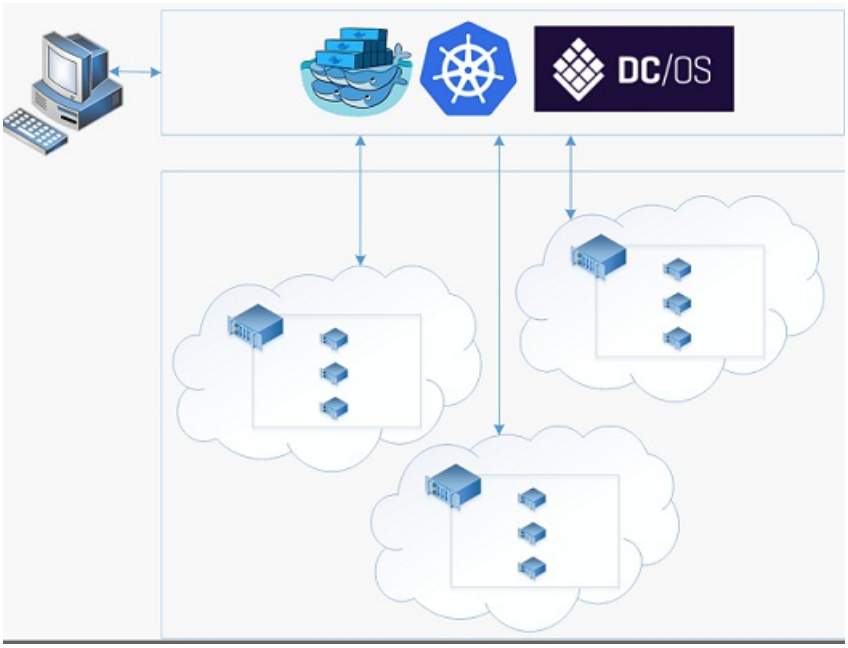
Table of Contents

Introduction to Docker container hosting solutions with Azure Container Service

6/27/2017 • 4 min to read • Edit Online

Azure Container Service makes it simpler for you to create, configure, and manage a cluster of virtual machines that are preconfigured to run containerized applications. It uses an optimized configuration of popular open-source scheduling and orchestration tools. 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.



Azure Container Service leverages the Docker container format to ensure that your application containers are fully portable. It also supports your choice of Marathon and DC/OS, Docker Swarm, or Kubernetes so that you can scale these applications to thousands of containers, or even tens of thousands.

By using Azure Container Service, you can take advantage of the enterprise-grade features of Azure, while still maintaining application portability--including portability at the orchestration layers.

Using Azure Container Service

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 API endpoints for your chosen orchestrator (DC/OS, Docker Swarm, or Kubernetes). By using these endpoints, you can leverage any software that is capable of talking to those endpoints. For example, in the case of the Docker Swarm endpoint, you might choose to use the Docker command-line interface (CLI). For DC/OS, you might choose the DCOS CLI. For Kubernetes, you might choose `kubectl`.

Creating a Docker cluster by using Azure Container Service

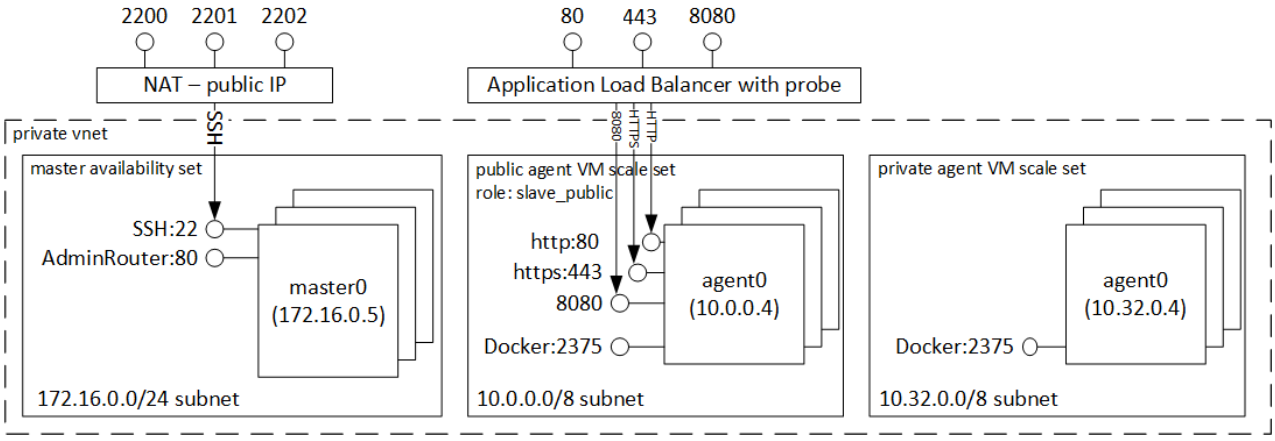
To begin using Azure Container Service, you deploy an Azure Container Service cluster via the portal (search the Marketplace for **Azure Container Service**), by using an Azure Resource Manager template (Docker Swarm, DC/OS, or Kubernetes), or with the Azure CLI 2.0. The provided quickstart templates can be modified to include additional or advanced Azure configuration. For more information, see Deploy an Azure Container Service cluster.

Deploying an application

Azure Container Service provides a choice of Docker Swarm, DC/OS, or Kubernetes for orchestration. How you deploy your application depends on your choice of orchestrator.

Using DC/OS

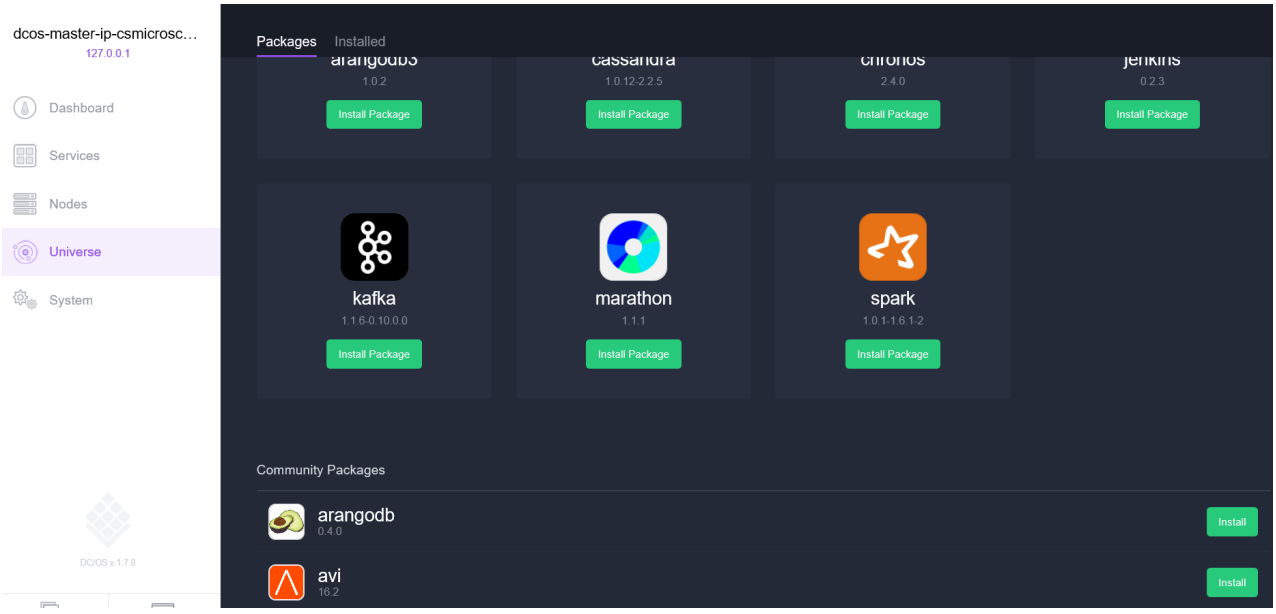
DC/OS is a distributed operating system based on the Apache Mesos distributed systems kernel. Apache Mesos is housed at the Apache Software Foundation and lists some of the biggest names in IT as users and contributors.



DC/OS and Apache Mesos include an impressive feature set:

- Proven scalability
- Fault-tolerant replicated master and slaves using Apache ZooKeeper
- Support for Docker-formatted containers
- Native isolation between tasks with Linux containers
- Multiresource scheduling (memory, CPU, disk, and ports)
- Java, Python, and C++ APIs for developing new parallel applications
- A web UI for viewing cluster state

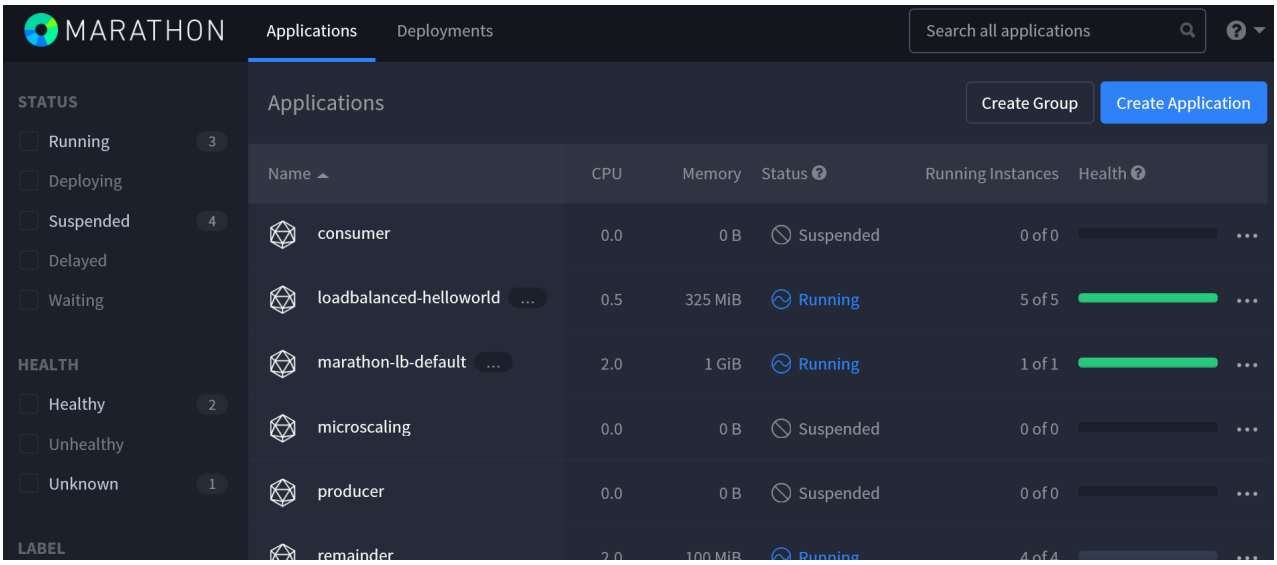
By default, DC/OS running on Azure Container Service includes the Marathon orchestration platform for scheduling workloads. However, included with the DC/OS deployment of ACS is the Mesosphere Universe of services that can be added to your service. Services in the Universe include Spark, Hadoop, Cassandra, and much more.



Using Marathon

Marathon is a cluster-wide init and control system for services in cgroups--or, in the case of Azure Container

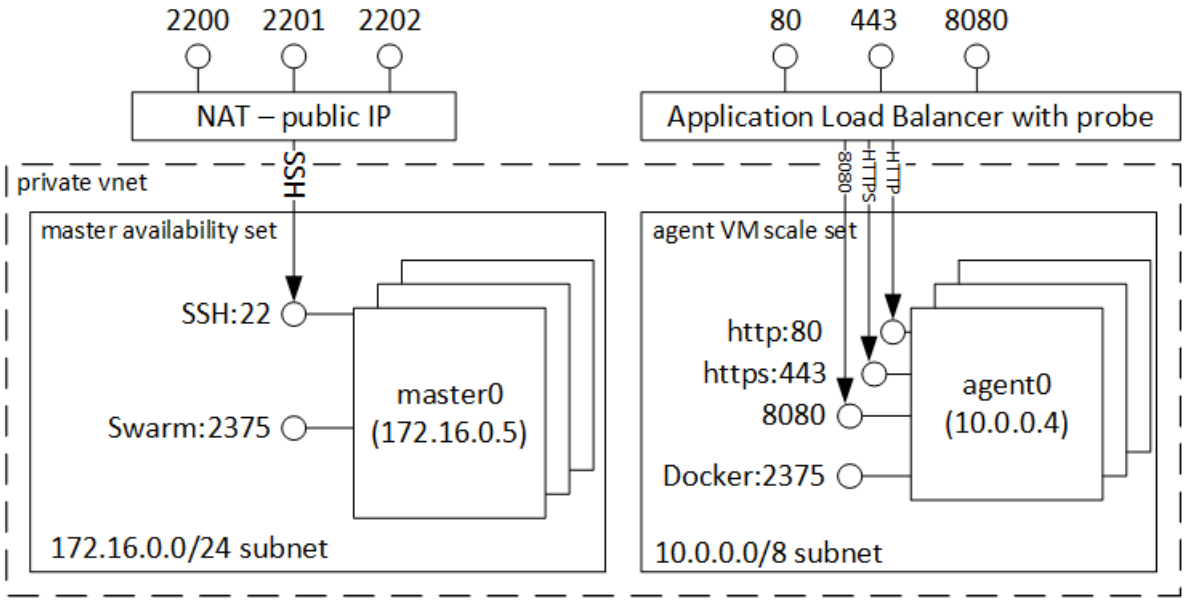
Service, Docker-formatted containers. Marathon provides a web UI from which you can deploy your applications. You can access this at a URL that looks something like `http://DNS_PREFIX.REGION.cloudapp.azure.com` where DNS_PREFIX and REGION are both defined at deployment time. Of course, you can also provide your own DNS name. For more information on running a container using the Marathon web UI, see DC/OS container management through the Marathon web UI.



You can also use the REST APIs for communicating with Marathon. There are a number of client libraries that are available for each tool. They cover a variety of languages--and, of course, you can use the HTTP protocol in any language. In addition, many popular DevOps tools provide support for Marathon. This provides maximum flexibility for your operations team when you are working with an Azure Container Service cluster. For more information on running a container by using the Marathon REST API, see DC/OS container management through the Marathon REST API.

Using Docker Swarm

Docker Swarm provides native clustering for Docker. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts on Azure Container Service.



NOTE

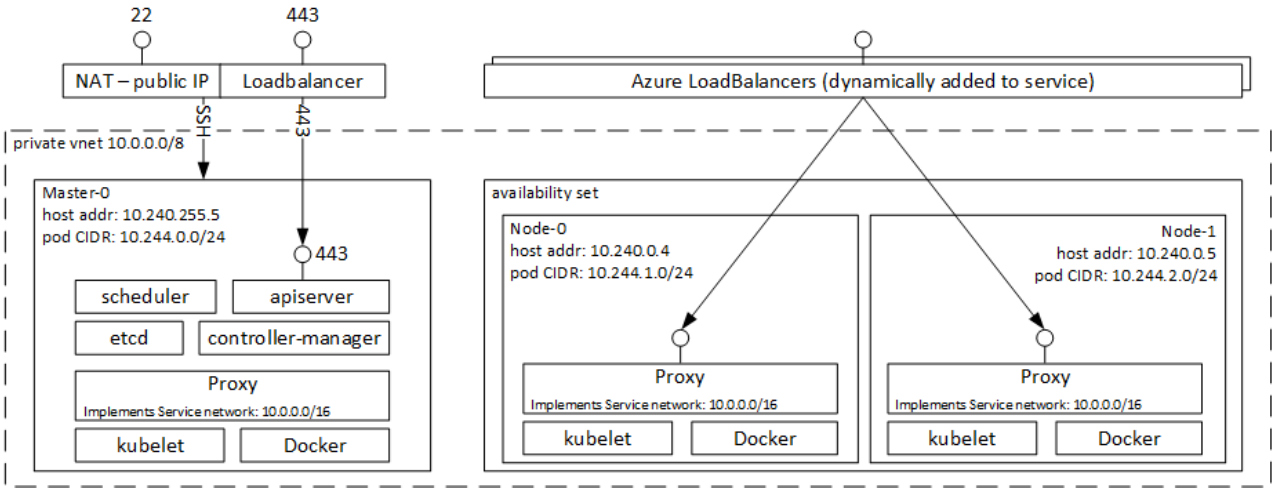
The Docker Swarm orchestrator in Azure Container Service uses legacy standalone Swarm. Currently, the integrated Swarm mode (in Docker 1.12 and higher) is not a supported orchestrator in Azure Container Service. If you want to deploy a Swarm mode cluster in Azure, use the open-source ACS Engine, a community-contributed quickstart template, or a Docker solution in the Azure Marketplace.

Supported tools for managing containers on a Swarm cluster include, but are not limited to, the following:

- Dokku
- Docker CLI and Docker Compose
- Krane
- Jenkins

Using Kubernetes

Kubernetes is a popular open-source, production-grade container orchestrator tool. Kubernetes automates deployment, scaling, and management of containerized applications. Because it is an open-source solution and is driven by the open-source community, it runs seamlessly on Azure Container Service and can be used to deploy containers at scale on Azure Container Service.



It has a rich set of features including:

- Horizontal scaling
- Service discovery and load balancing
- Secrets and configuration management
- API-based automated rollouts and rollbacks
- Self-healing

Videos

Getting started with Azure Container Service (101):

Next steps

Deploy a container service cluster using the portal or Azure CLI 2.0.

Deploy Kubernetes cluster for Linux containers

7/7/2017 • 3 min to read • Edit Online

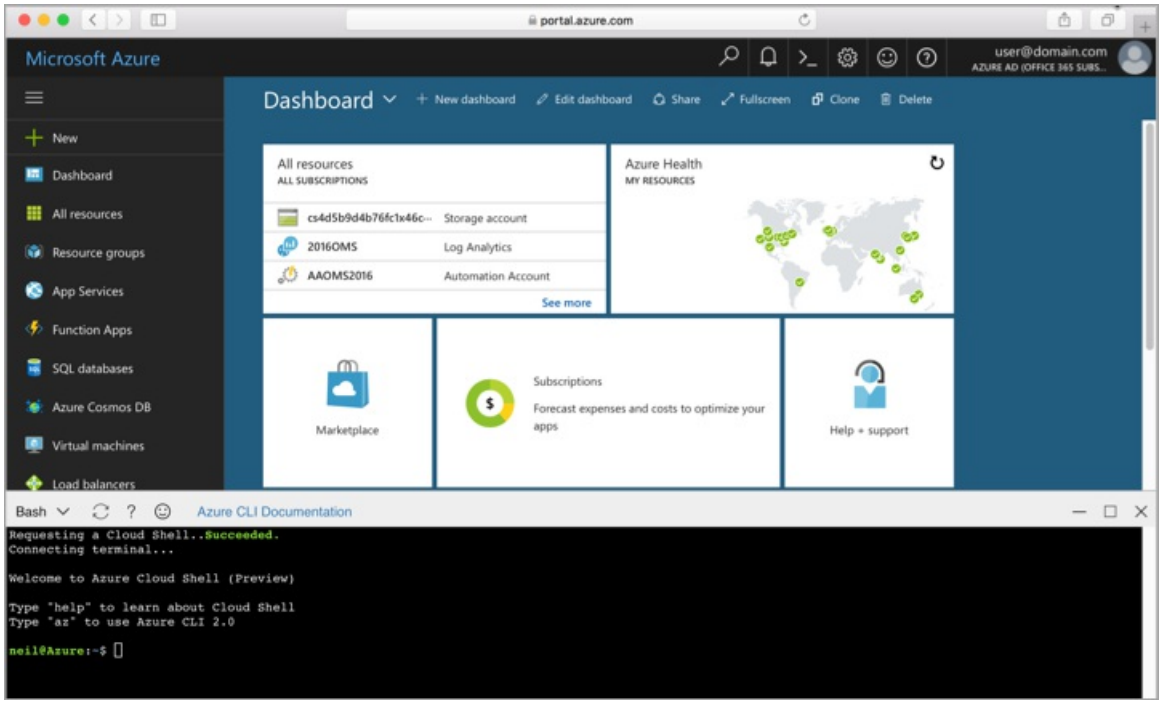
The Azure CLI is used to create and manage Azure resources from the command line or in scripts. This guide details using the Azure CLI to deploy a Kubernetes cluster in Azure Container Service. Once the cluster is deployed, you connect to it with the Kubernetes `kubectl` command-line tool, and you deploy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

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



If you don't have an Azure subscription, create a free account before you begin.

Log in to Azure

Log in to your Azure subscription with the az login command and follow the on-screen directions.

```
az login
```

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 Linux agent nodes. This example creates SSH keys if they don't already exist in the default locations. To use a specific set of keys, use the `--ssh-key-value` option. Update the cluster name to something appropriate to your environment.

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

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. If you are running the Azure CLI on macOS or Linux, you might need to run the command with `sudo`.

```
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 an NGINX container

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

The following command starts the NGINX Docker container in a Kubernetes pod on one of the nodes. In this case, the container runs the NGINX web server pulled from an image in Docker Hub.

```
kubectl run nginx --image nginx
```

To see that the container is running, run:

```
kubectl get pods
```

View the NGINX welcome page

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

```
kubectl expose deployments nginx --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.

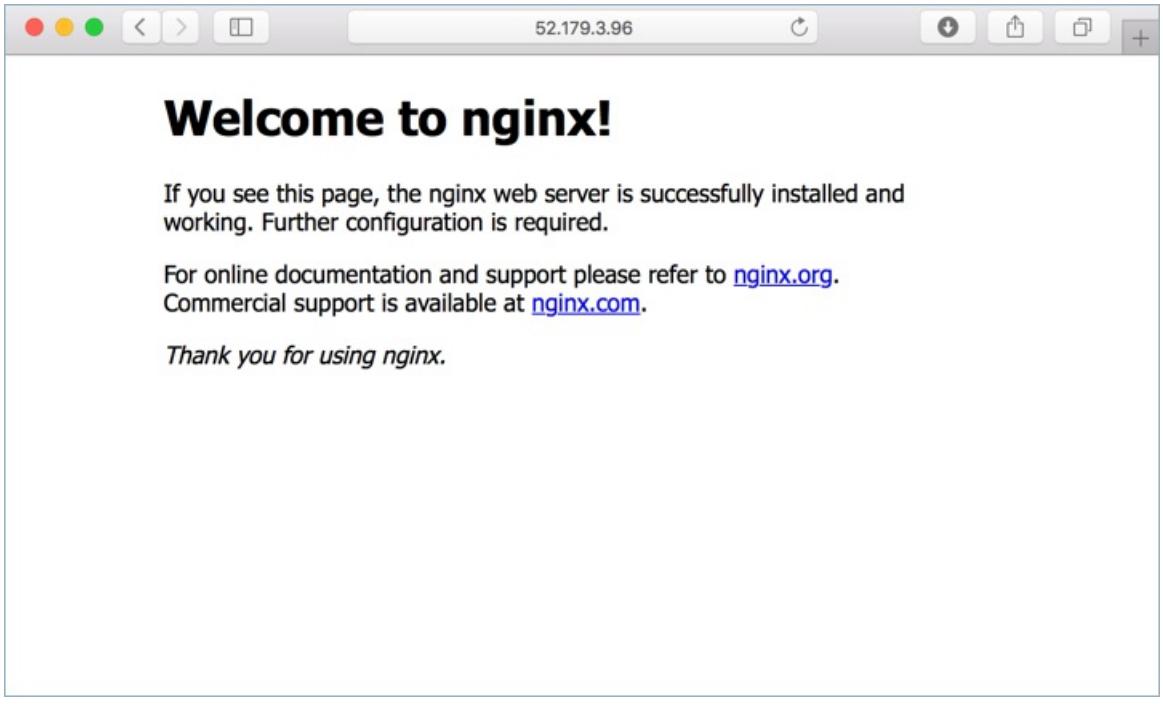
Run the following command to see the status of the service.

```
kubectl get svc
```

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

```
NAME         CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
kubernetes   10.0.0.1       <none>          443/TCP        21h
nginx        10.0.111.25    52.179.3.96     80/TCP         22m
```

You can use a web browser of your choice to see the default NGINX 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 `kubectl`, and deployed a pod with an NGINX container. To learn more about Azure Container Service, continue to the Kubernetes cluster tutorial.

Manage an ACS Kubernetes cluster

Deploy Kubernetes cluster for Windows containers

6/27/2017 • 4 min to read • Edit Online

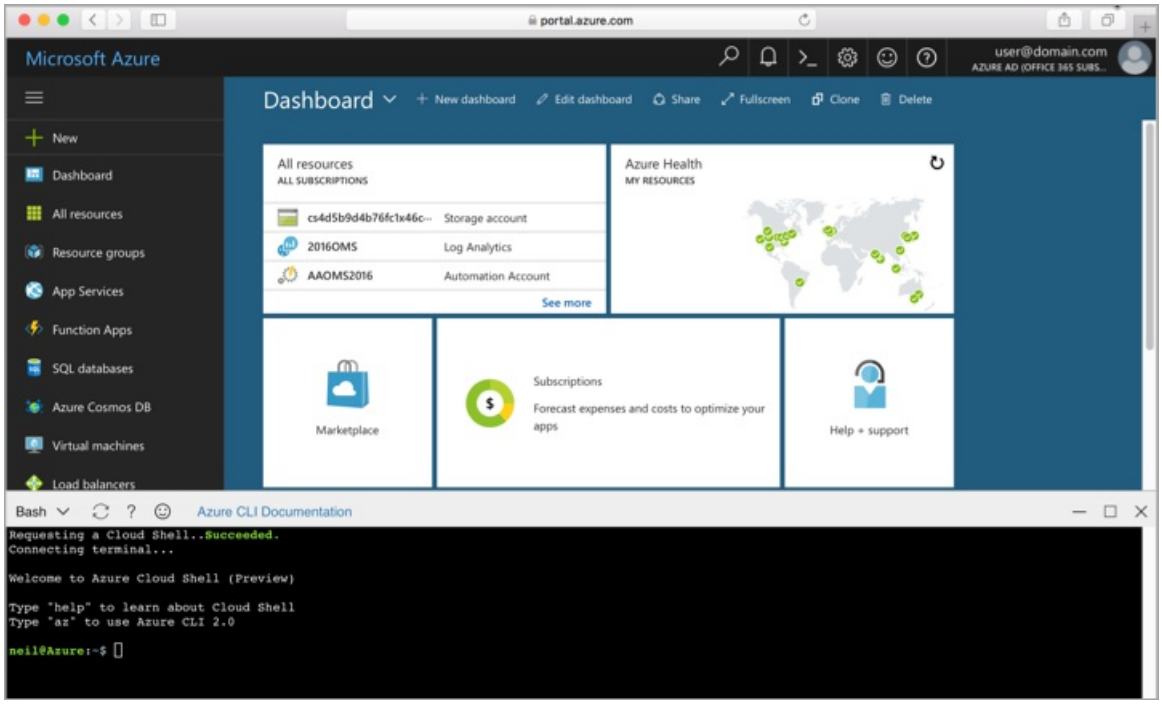
The Azure CLI is used to create and manage Azure resources from the command line or in scripts. This guide details using the Azure CLI to deploy a Kubernetes cluster in Azure Container Service. Once the cluster is deployed, you connect to it with the Kubernetes `kubectl` command-line tool, and you deploy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

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



If you don't have an Azure subscription, create a free account before you begin.

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

Log in to Azure

Log in to your Azure subscription with the az login command and follow the on-screen directions.

```
az login
```

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 `kubctl 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.

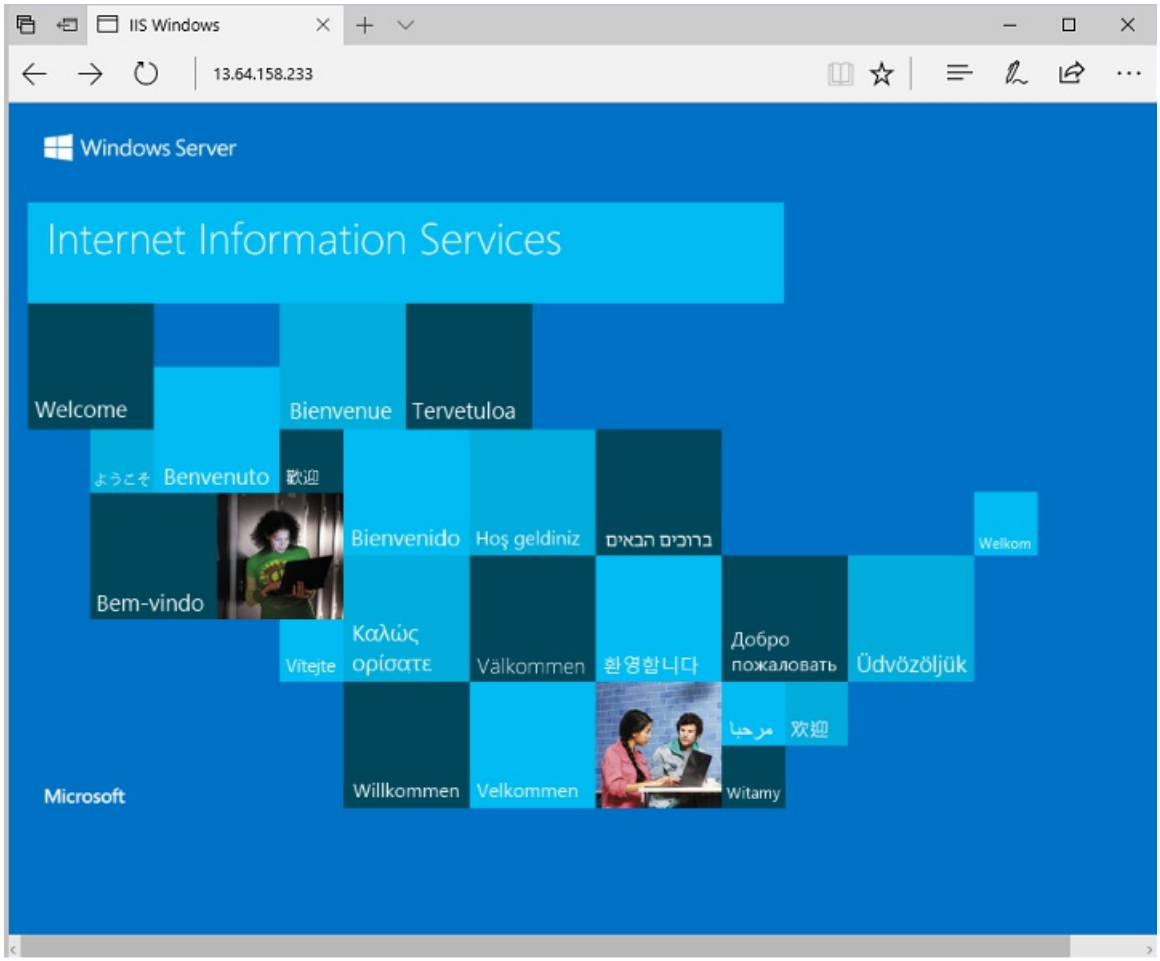
Run the following command to see the status of the service.

```
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 `kubectl`, 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

Deploy a DC/OS cluster

6/27/2017 • 3 min to read • Edit Online

DC/OS provides a distributed platform for running modern and containerized applications. With Azure Container Service, provisioning of a production ready DC/OS cluster is simple and quick. This quick start details the basic steps needed to deploy a DC/OS cluster and run basic workload.

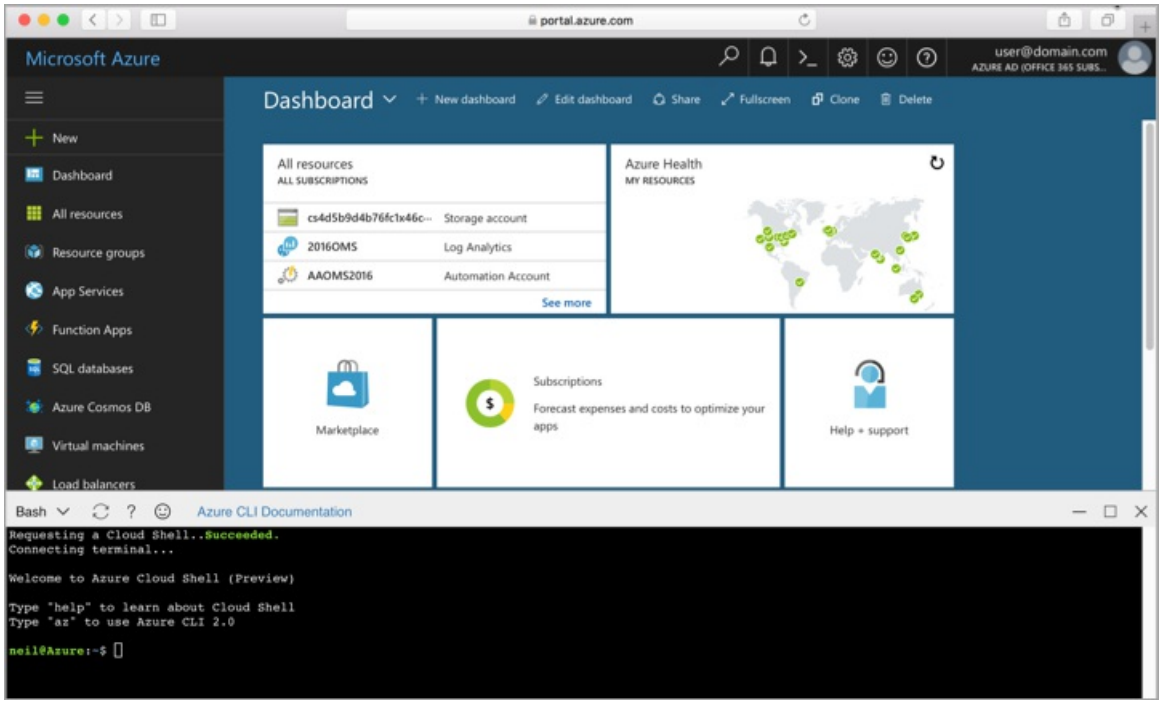
If you don't have an Azure subscription, create a free account before you begin.

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



Log in to Azure

Log in to your Azure subscription with the az login command and follow the on-screen directions.

```
az login
```

Create a resource group

Create a resource group with the az group create command. An Azure resource group is a logical container into

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 DC/OS cluster

Create a DC/OS cluster with the az acs create command.

The following example creates a DC/OS cluster named *myDCOSCluster* and creates SSH keys if they do not already exist. To use a specific set of keys, use the `--ssh-key-value` option.

```
az acs create \
  --orchestrator-type dcos \
  --resource-group myResourceGroup \
  --name myDCOSCluster \
  --generate-ssh-keys
```

After several minutes, the command completes, and returns information about the deployment.

Connect to DC/OS cluster

Once a DC/OS cluster has been created, it can be accesses through an SSH tunnel. Run the following command to return the public IP address of the DC/OS master. This IP address is stored in a variable and used in the next step.

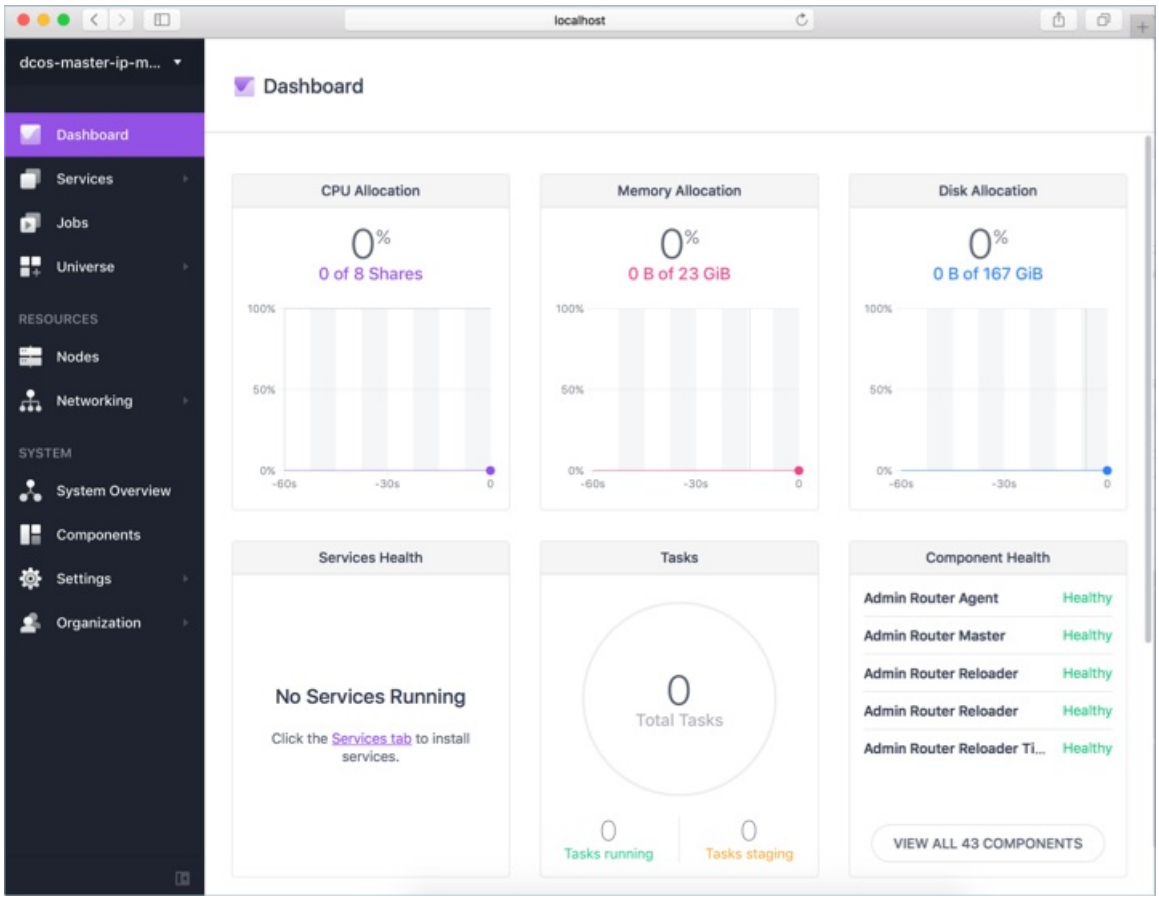
```
ip=$(az network public-ip list --resource-group myResourceGroup --query "[?contains(name,'dcos-master')].
[ipAddress]" -o tsv)
```

To create the SSH tunnel, run the following command and follow the on-screen instructions. If port 80 is already in use, the command fails. Update the tunneled port to one not in use, such as `85:localhost:80`.

```
sudo ssh -i ~/.ssh/id_rsa -fNL 80:localhost:80 -p 2200 azureuser@$ip
```

The SSH tunnel can be tested by browsing to `http://localhost`. If a port other that 80 has been used, adjust the location to match.

If the SSH tunnel was successfully created, the DC/OS portal is returned.



Install DC/OS CLI

The DC/OS command line interface is used to manage a DC/OS cluster from the command-line. Install the DC/OS cli using the az acs dcos install-cli command. If you are using Azure CloudShell, the DC/OS CLI is already installed.

If you are running the Azure CLI on macOS or Linux, you might need to run the command with sudo.

```
az acs dcos install-cli
```

Before the CLI can be used with the cluster, it must be configured to use the SSH tunnel. To do so, run the following command, adjusting the port if needed.

```
dcos config set core.dcos_url http://localhost
```

Run an application

The default scheduling mechanism for an ACS DC/OS cluster is Marathon. Marathon is used to start an application and manage the state of the application on the DC/OS cluster. To schedule an application through Marathon, create a file named *marathon-app.json*, and copy the following contents into it.

```
{
  "id": "demo-app",
  "cmd": null,
  "cpus": 1,
  "mem": 32,
  "disk": 0,
  "instances": 1,
  "container": {
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80,
          "protocol": "tcp",
          "name": "80",
          "labels": null
        }
      ]
    },
    "type": "DOCKER"
  },
  "acceptedResourceRoles": [
    "slave_public"
  ]
}
```

Run the following command to schedule the application to run on the DC/OS cluster.

```
dcos marathon app add marathon-app.json
```

To see the deployment status for the app, run the following command.

```
dcos marathon app list
```

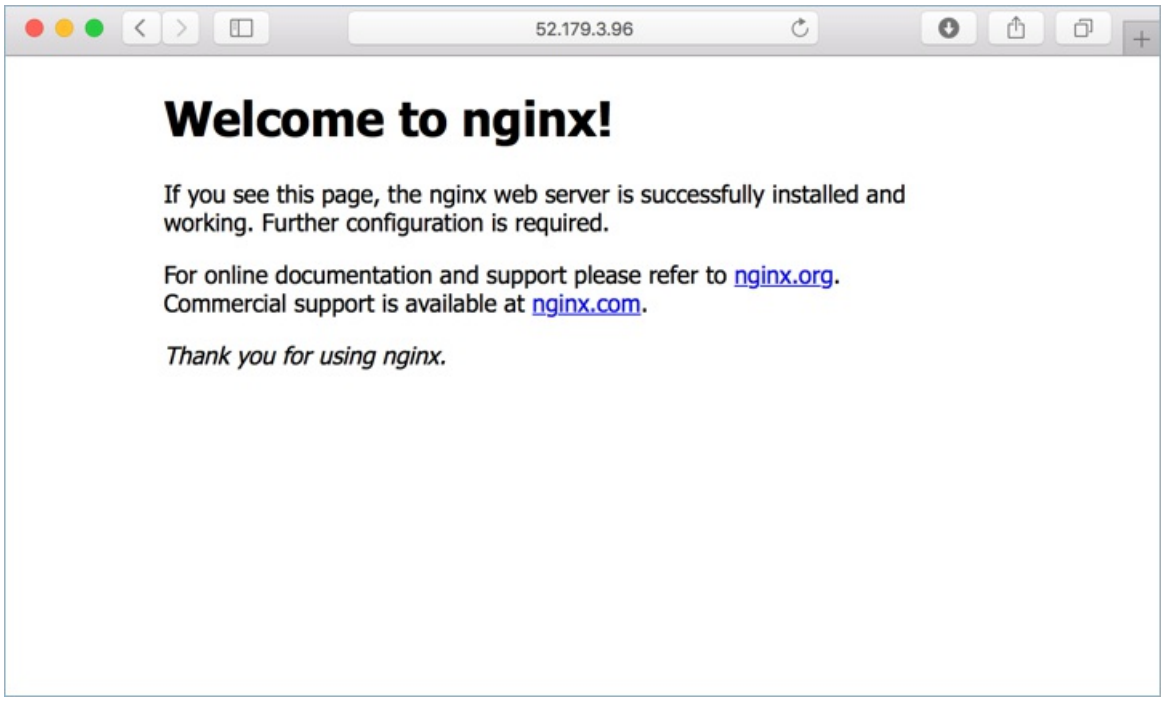
When the **WAITING** column value switches from *True* to *False*, application deployment has completed.

```
ID     MEM  CPUS  TASKS  HEALTH  DEPLOYMENT  WAITING  CONTAINER  CMD
/test   32   1     1/1    ---       ---      False      DOCKER   None
```

Get the public IP address of the DC/OS cluster agents.

```
az network public-ip list --resource-group myResourceGroup --query "[?contains(name,'dcos-agent')].[ipAddress]"
-o tsv
```

Browsing to this address returns the default NGINX site.



Delete DC/OS cluster

When no longer needed, you can use the az group delete command to remove the resource group, DC/OS cluster, and all related resources.

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

Next steps

In this quick start, you've deployed a DC/OS cluster and have run a simple Docker container on the cluster. To learn more about Azure Container Service, continue to the ACS tutorials.

Manage an ACS DC/OS Cluster

Create container images to be used with Azure Container Service

6/27/2017 • 3 min to read • Edit Online

In this tutorial, an application is prepared for Kubernetes. Steps completed include:

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

In subsequent tutorials, these container images are 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 database.

Use git to download a copy of the application to your development environment.

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

Inside the application directory, pre-created Dockerfiles and Kubernetes manifest files can be found. These files are used to create assets throughout the tutorial set.

Create container images

To create a container image for the application front-end, use the docker build command.

```
docker build ./azure-voting-app/azure-vote -t azure-vote-front
```

Repeat the command, this time for the back-end container image.

```
docker build ./azure-voting-app/azure-vote-mysql -t azure-vote-back
```

When completed, use the `docker images` command to see the created images.

```
docker images
```

Output:

```
REPOSITORY                  TAG                 IMAGE ID            CREATED              SIZE
azure-vote-front            latest              c13c4f50ede1        39 seconds ago       716 MB
azure-vote-back             latest              33fe5afc1885        About a minute ago   407 MB
mysql                       latest              e799c7f9ae9c        4 weeks ago          407 MB
tiangolo/uwsgi-nginx-flask  flask               788ca94b2313        8 months ago         694 MB
```

Test application

Now that two container images have been created, test these images in your local development environment.

First, create a Docker network. This network is used for communication between the containers.

```
docker network create azure-vote
```

Run an instance of the back-end container image using the `docker run` command.

In this example, the mysql database file is stored inside the container. Once this application is moved to the Kubernete clusters, an external data volume is used to store the database file. Also, environment variables are being used to set MySQL credentials.

```
docker run -p 3306:3306 --name azure-vote-back -d --network azure-vote -e MYSQL_ROOT_PASSWORD=Password12 -e
MYSQL_USER=dbuser -e MYSQL_PASSWORD=Password12 -e MYSQL_DATABASE=azurevote azure-vote-back
```

Run an instance of the front-end container image.

Environment variables are being used to configure the database connection information.

```
docker run -d -p 8080:80 --name azure-vote-front --network=azure-vote -e MYSQL_USER=dbuser -e
MYSQL_PASSWORD=Password12 -e MYSQL_DATABASE=azurevote -e MYSQL_HOST=azure-vote-back azure-vote-front
```

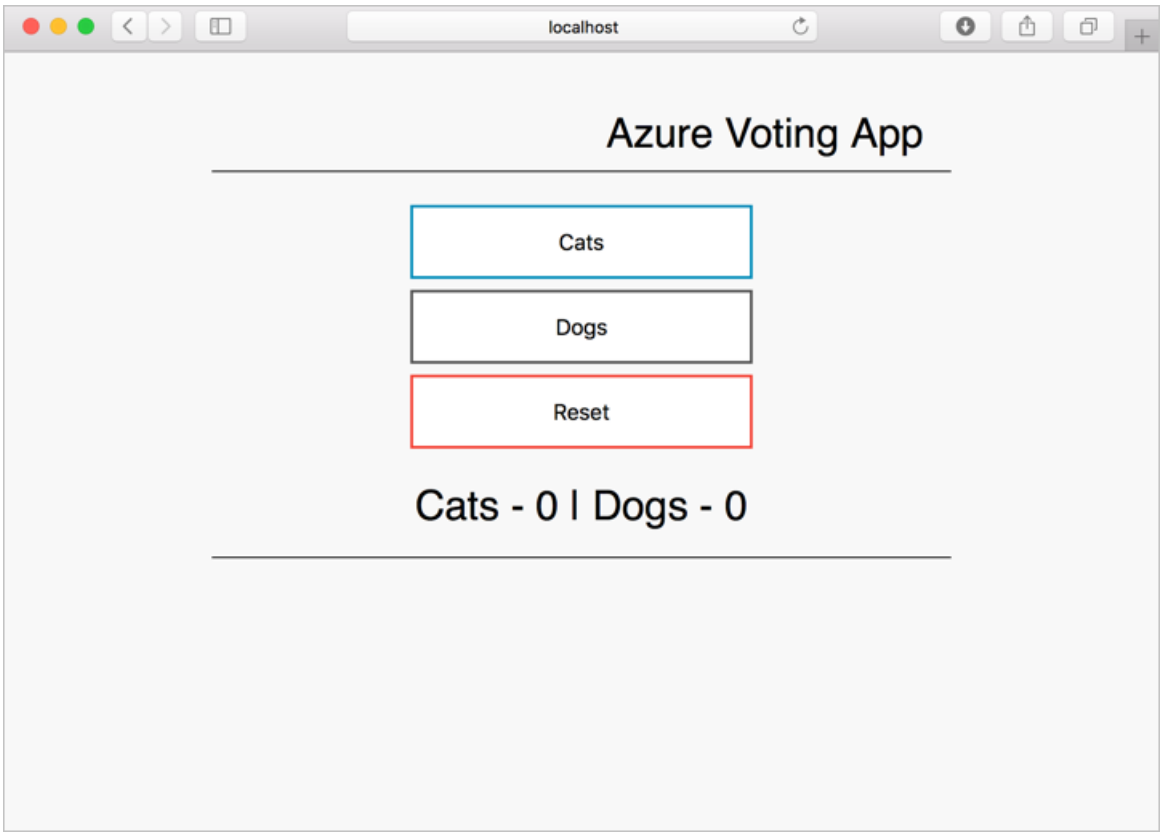
When complete, run `docker ps` to see the running containers.

```
docker ps
```

Output:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                            NAMES
3aa02e8ae965        azure-vote-front     "/usr/bin/supervisord"   59 seconds ago      Up 57 seconds
443/tcp, 0.0.0.0:8080->80/tcp   flaskmysqlvote_azure-vote-front_1
5ae60b3ba181        azure-vote-backend   "docker-entrypoint..."   59 seconds ago      Up 58 seconds
0.0.0.0:3306->3306/tcp          azure-vote-back
```

Browse to `http://localhost:8080` to see the running application. The application takes a few seconds to initialize. If an error is encountered, try again.



Clean up resources

Now that application functionality has been validated, the running containers can be stopped and removed. Do not delete the container images. These images are uploaded to an Azure Container Registry instance in the next tutorial.

Stop and delete the front-end container with the docker rm command.

```
docker rm -f azure-vote-front
```

Stop and delete the back-end container with the docker rm command.

```
docker rm -f azure-vote-back
```

Delete the network with the docker network rm command.

```
docker network rm azure-vote
```

At completion, you have two container images that make up 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
- Creating container images from application source
- Testing the images 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

7/6/2017 • 4 min to read • Edit Online

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

- Deploying an Azure Container Registry instance
- Tagging container images for ACR
- Uploading images to ACR

In subsequent tutorials, this ACR instance is integrated with an Azure Container Service Kubernetes cluster, for securely running container images.

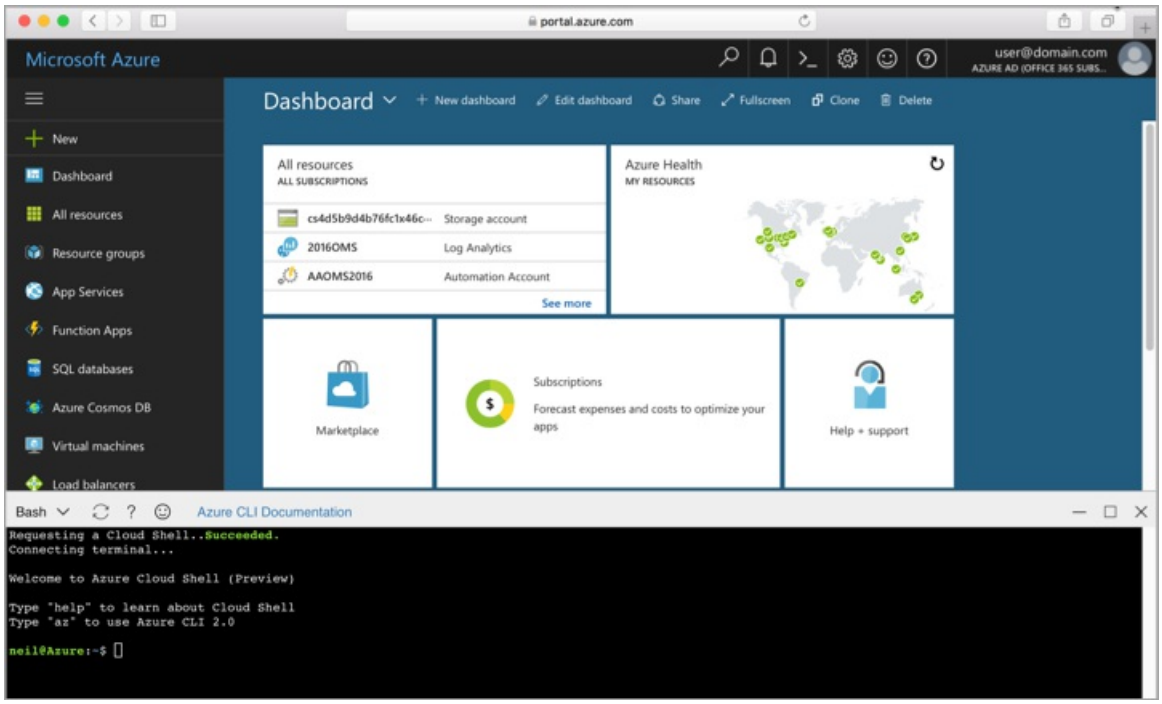
Before you begin

In the previous tutorial, container images were created for a simple Azure Voting application. In this tutorial, these images are pushed to an Azure Container Registry. If you have not created the Azure Voting app images, return to Tutorial 1 – Create container images. Alternatively, the steps detailed here work with any container image.

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 all of the steps in this topic:



If you choose to install and use the CLI locally, 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 *eastus* region.

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

Create an Azure Container registry with the az acr create command. The name of a Container Registry **must be unique**. Using the following example, update the name with some random characters.

```
az acr create --resource-group myResourceGroup --name myContainerRegistry007 --sku Basic --admin-enabled true
```

Get ACR information

Once the ACR instance has been created, the name, login server name, and authentication password are needed. The following code returns each of these values. Note each value down, they are referenced throughout this tutorial.

ACR Name and Login Server:

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

ACR Password - update with the ACR name.

```
az acr credential show --name <acrName> --query passwords[0].value -o tsv
```

Container registry login

You must log in to your ACR instance before pushing images to it. Use the docker login command to complete the operation. When running docker login, you need to provide th ACR login server name and ACR credentials.

```
docker login --username=<acrName> --password=<acrPassword> <acrLoginServer>
```

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

Tag container images

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 see a list of current images, use the `docker images` command.

```
docker images
```

Output:

```
REPOSITORY                  TAG                 IMAGE ID            CREATED              SIZE
azure-vote-front            latest              c13c4f50ede1        39 seconds ago       716 MB
azure-vote-back             latest              33fe5afc1885        About a minute ago   407 MB
mysql                       latest              e799c7f9ae9c        4 weeks ago          407 MB
tiangolo/uwsgi-nginx-flask  flask               788ca94b2313        8 months ago         694 MB
```

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

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

Repeat the command with the *azure-vote-back* image.

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

Once tagged, run `docker images` to verify the operation.

```
docker images
```

Output:

```
REPOSITORY                                          TAG                 IMAGE ID            CREATED
SIZE
azure-vote-back                                     latest              a9dace4e1a17        7 minutes ago
407 MB
mycontainerregistry082.azurecr.io/azure-vote-back   v1                  a9dace4e1a17        7 minutes ago
407 MB
azure-vote-front                                    latest              eaf2b9c57e5e        8 minutes ago
716 MB
mycontainerregistry082.azurecr.io/azure-vote-front  v1                  eaf2b9c57e5e        8 minutes ago
716 MB
mysql                                               latest              e799c7f9ae9c        6 weeks ago
407 MB
tiangolo/uwsgi-nginx-flask                          flask               788ca94b2313        8 months ago
694 MB
```

Push images to ACR

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

Using the following example, replace the ACR loginServer name with the loginServer from your environment. This takes a couple of minutes to complete.

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

Do the same to the *azure-vote-back* image.

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

List images in ACR

To return a list of images that have been pushed to your Azure Container registry, user the az acr repository list

command. Update the command with the ACR instance name.

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

Output:

```
Result
----------------
azure-vote-back
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> --username <acrName> --password <acrPassword> --repository azure-vote-front --output table
```

Output:

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

At tutorial completion, the two container images have been stored in a private Azure Container Registry instance. These images are 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:

- Deploying an Azure Container Registry instance
- Tagging container images for ACR
- Uploading images 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

7/6/2017 • 2 min to read • Edit Online

Kubernetes provides a distributed platform for running modern and containerized applications. With Azure Container Service, provisioning of a production ready Kubernetes cluster is simple and quick. This quick start details basic steps needed to deploy a Kubernetes cluster. Steps completed include:

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

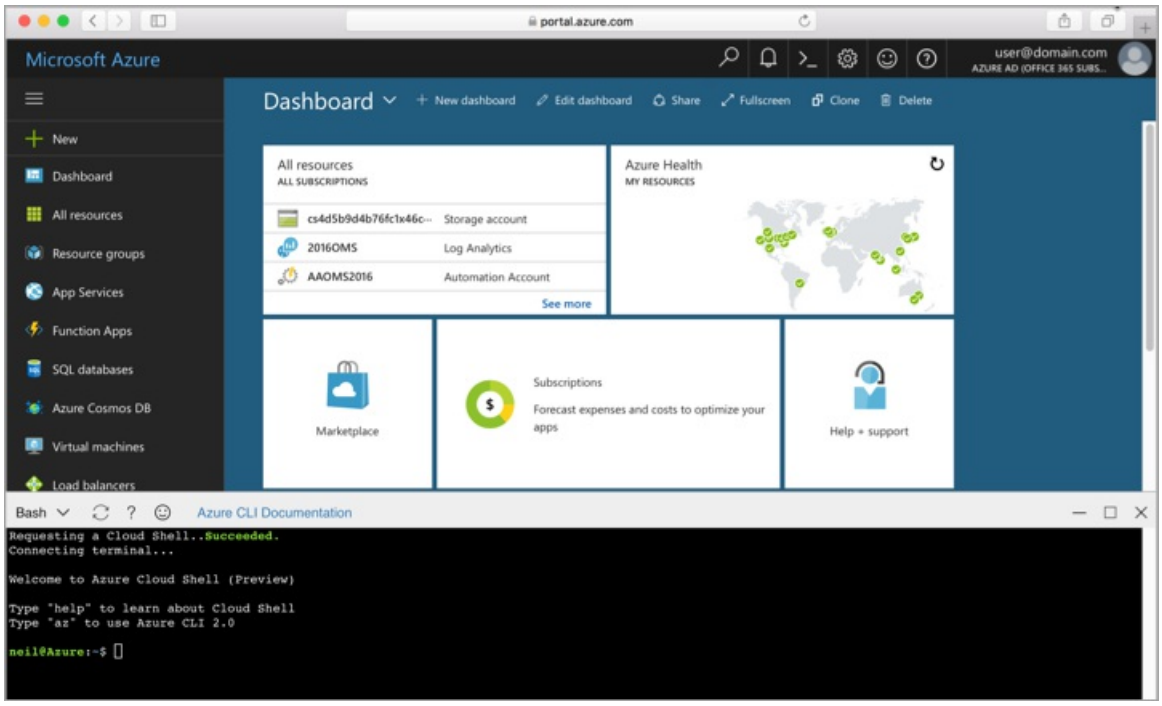
Before you begin

In previous tutorials, container images were 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.

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 all of the steps in this topic:



If you choose to install and use the CLI locally, 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.

Create Kubernetes cluster

In the previous tutorial, a resource group named *myResourceGroup* was created. If you have not done so, create this resource group now.

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

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. If they do not already exist, SSH keys are created. To use a specific set of keys in a non-default location, use the `--ssh-key-value` option.

```
az acs create --orchestrator-type=kubernetes --resource-group myResourceGroup --name=myK8SCluster --generate-ssh-keys
```

After several minutes, the command completes, and returns 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 competition, 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:

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

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

Deploy application in Kubernetes

Run applications in Kubernetes

7/6/2017 • 4 min to read • Edit Online

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

- Kubernetes objects introduction
- Download Kubernetes manifest files
- Run application in Kubernetes
- Test the application

In subsequent tutorials, this application is scaled out, updated, and the Kubernetes cluster monitored.

Before you begin

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

At minimum, this tutorial requires a Kubernetes cluster.

Kubernetes objects

When deploying a containerized application into Kubernetes, many different Kubernetes objects are created. Each object represents the desired state for the cluster. For example, a simple application may consist of a pod, which is a grouping of closely related containers, a persistent volume, which is a piece of networked storage, and a deployment, which manages the state of the application.

For details on all Kubernetes objects, see Kubernetes Concepts on kubernetes.io.

Get manifest files

For this tutorial, Kubernetes objects are deployed using Kubernetes manifests. A Kubernetes manifest is a YAML file containing object configuration instructions.

The manifest files for each object in this tutorial are available in the Azure Vote application repo, which was cloned in a previous tutorial. If you have not already done so, clone the repo with the following command:

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

The manifest files are found in the following directory of the cloned repo.

```
/azure-voting-app/kubernetes-manifests/
```

Run application

Storage objects

Because the Azure Vote application includes a MySQL database, you want to store the database file on a volume that can be shared between pods. In this configuration, if the MySQL pod is recreated, the database file remains intact.

The `storage-resources.yaml` manifest file creates a storage class object, which defines how and where a persistent volume is created. Several volume plug-ins are available for Kubernetes. In this case, the Azure disk plug-in is used.

A persistent volume claim is also created, which configures a piece of storage (using a storage class), and assigns it to a pod.

Run the following to create the storage objects.

```
kubectl create -f storage-resources.yaml
```

Once completed, a virtual disk is created and attached to the resulting Kubernetes pod. The virtual disk is automatically created in a storage account residing in the same resource group as the Kubernetes cluster, and of the same configuration as the storage class object (Standard_LRS).

Secure sensitive values

Kubernetes secrets provide secure storage for sensitive information. Using the `pod-secrets.yaml` file, the Azure Vote database credentials are stored in a secret.

Run the following to create the secrets objects.

```
kubectl create -f pod-secrets.yaml
```

Create deployments

A Kubernetes deployment manages the state of Kubernetes pods. This management includes things like ensuring that the desired replica counts are running, volumes are mounted, and the proper container images are being used.

The `azure-vote-deployment.yaml` manifest file creates a deployment for the front-end and back-end portions of the Azure Vote application.

Update image names

If using Azure Container Registry to store images, the image names need to be prepended with the ACR login server name.

Get the ACR login server name with the az acr list command.

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

Update the *azure-vote-front* and *azure-vote-back* container image names in the `azure-vote-deployment.yaml` file.

Front-end image name example:

```
containers:
      - name: azure-vote-front
        image: <acrLoginServer>/azure-vote-front:v1
```

Back-end image name example:

```
containers:
      - name: azure-vote-back
        image: <acrLoginServer>/azure-vote-front:v1
```

Create deployment objects

Run kubectl create to start the Azure Vote application.

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

Expose application

A Kubernetes service defines how a pod is accessed. With the Azure Vote app, the back-end deployment must be internally accessible by deployment name. The font-end deployment must be accessible over the internet. The Azure Vote app service configurations are defined in the `services.yaml` manifest file.

Run the following to create the services.

```
kubectl create -f services.yaml
```

Test application

Once all objects have been created, the application can be accessed over the external IP address for the azure-vote-front service. This service can take a few minutes to create. To monitor the service creation process, run the following command. When the *EXTERNAL-IP* value for the *azure-vote-front* service switches from *pending* to an IP address, the application is ready, and can be accessed on the external IP address.

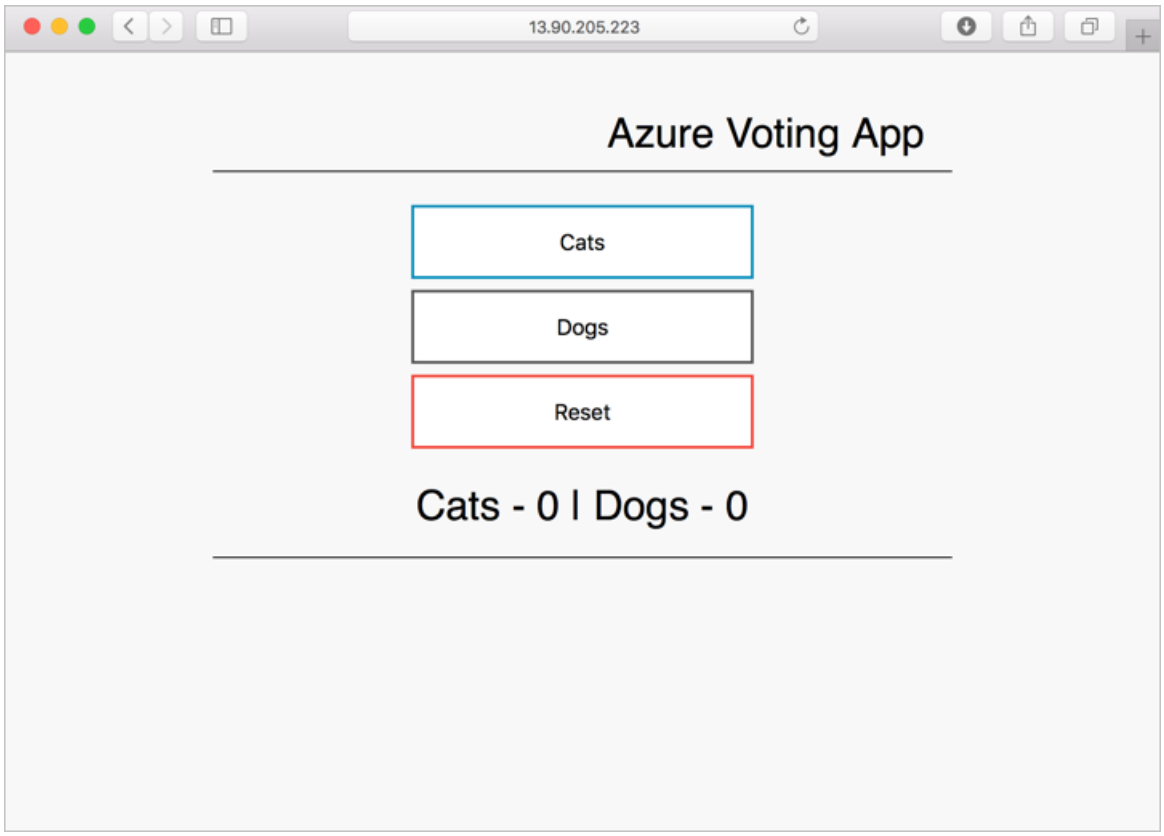
```
kubectl get service -w
```

Output:

```
NAME               CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
azure-vote-back    10.0.77.30    <none>          3306/TCP       4m
azure-vote-front   10.0.120.96   40.71.227.124   80:31482/TCP   4m
kubernetes         10.0.0.1      <none>          443/TCP        7m
```

After the service is ready, run `CTRL-C` to terminate kubectl watch.

Browse to the returned external IP address to see the application.



Next steps

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

- Kubernetes objects introduction
- Download Kubernetes manifest files
- Run application in Kubernetes
- Test 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

7/6/2017 • 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, you scale out the pods in the app and try pod autoscaling. You also learn how to scale the number of 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

Before you begin

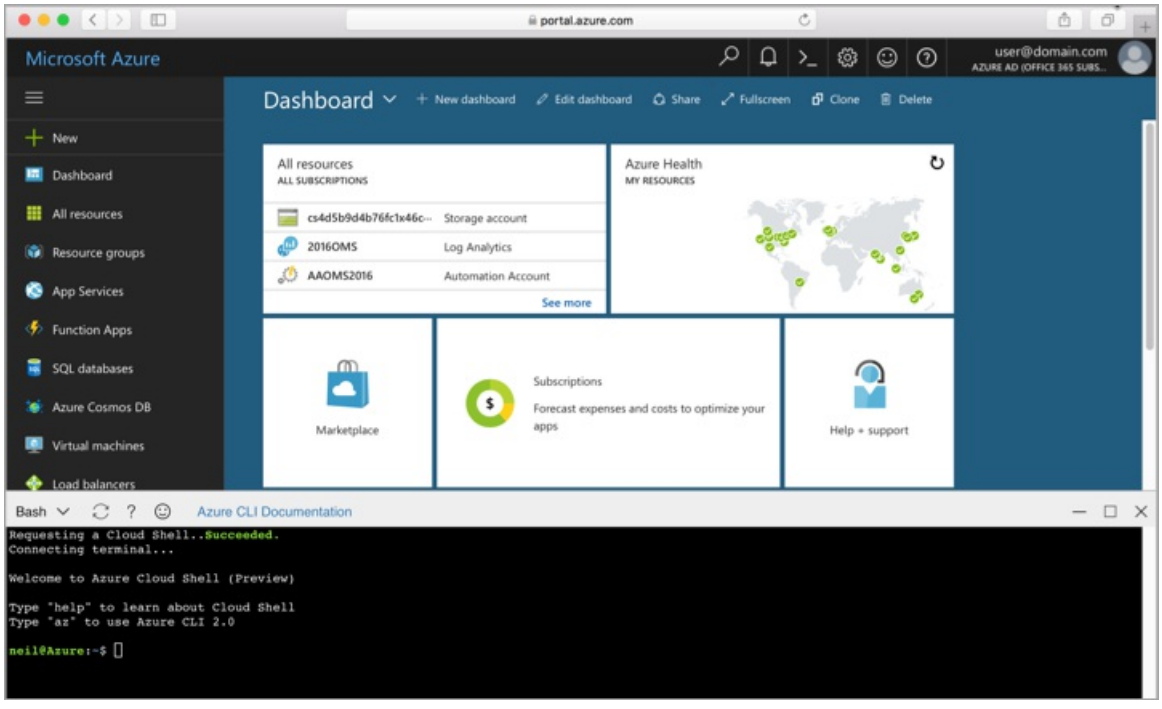
In previous tutorials, an application was packaged into container images, these images uploaded to Azure Container Registry, and a Kubernetes cluster created. The application was then run on the Kubernetes cluster. If you have not done these steps, and would like to follow along, return to the Tutorial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

At minimum, this tutorial requires a Kubernetes cluster with a running application.

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 all of the steps in this topic:



If you choose to install and use the CLI locally, 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.

Manually scale pods

The previous tutorial deployed the Azure Vote front-end and back-end each in a single pod. To verify, run the kubectl get command.

```
kubectl get pods
```

Output is similar to the following:

```
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 metrics.

To use the autoscaler, your pods must have CPU requests and limits defined. In the `azure-vote-front` deployment, each 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

7/7/2017 • 3 min to read • Edit Online

After you deploy an application in Kubernetes, you can update it by specifying a new container image or image version. When you update an application, the update rollout 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, and provides a rollback mechanism if a deployment failure occurs.

In this tutorial, 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 an updated application.

Before you begin

In previous tutorials, we packaged an application into container images, uploaded the images to Azure Container Registry, and created a Kubernetes cluster. We then ran the application on the Kubernetes cluster.

If you haven't taken these steps, and want to try them now, return to Tutorial 1 – Create container images.

At a minimum, this tutorial requires a Kubernetes cluster with a running application.

Update application

To complete the steps in this tutorial, you must have cloned a copy of the Azure Vote application. If necessary, create this cloned copy with the following command:

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

Open the `config_file.cfg` file with any code or text editor. You can find this file under the following directory of the cloned repo.

```
 /azure-voting-app/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 = 'Half Full'
VOTE2VALUE = 'Half Empty'
SHOWHOST = 'false'
```

Use `docker build` to re-create the front-end image.

```
docker build --no-cache ./azure-voting-app/azure-vote -t azure-vote-front:v2
```

Test application

Create a Docker network. This network is used for communication between the containers.

```
docker network create azure-vote
```

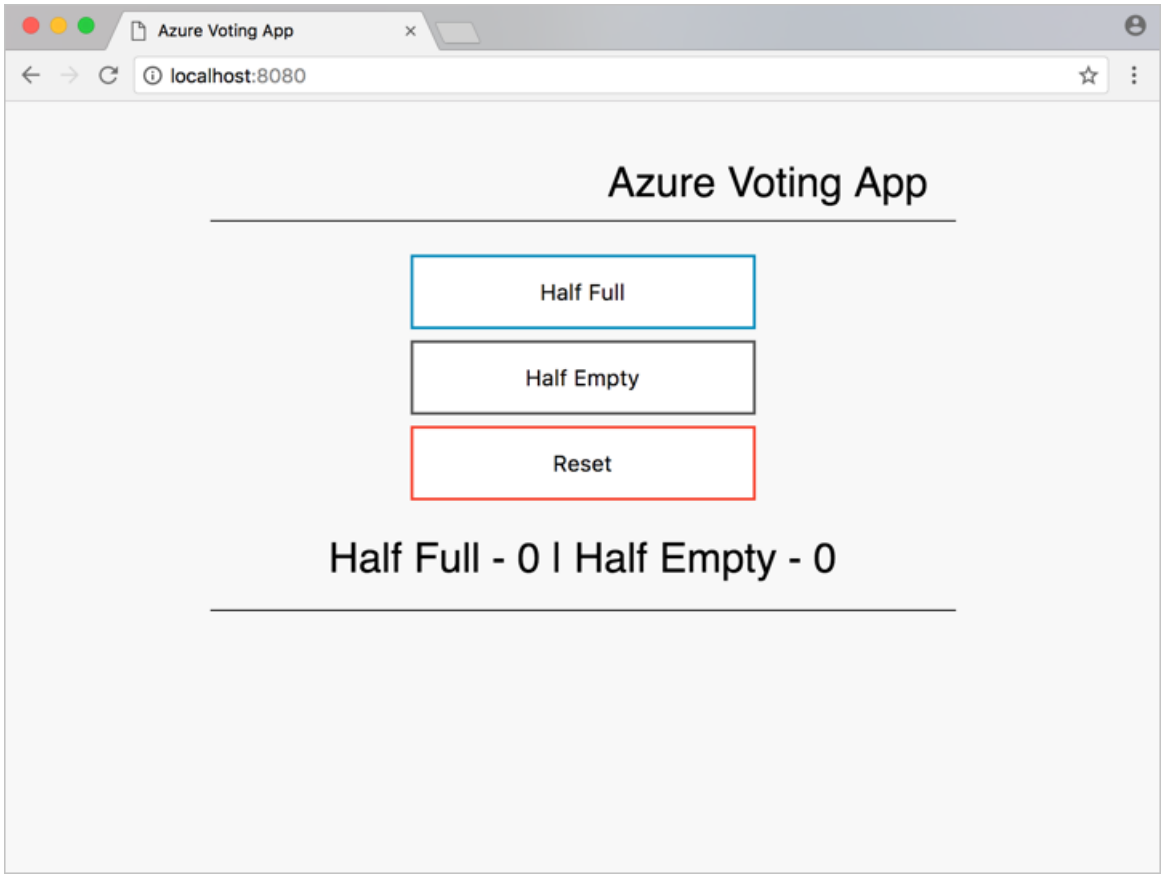
Run an instance of the back-end container image by using the `docker run` command.

```
docker run -p 3306:3306 --name azure-vote-back -d --network azure-vote -e MYSQL_ROOT_PASSWORD=Password12 -e
MYSQL_USER=dbuser -e MYSQL_PASSWORD=Password12 -e MYSQL_DATABASE=azurevote azure-vote-back
```

Run an instance of the front-end container image.

```
docker run -d -p 8080:80 --name azure-vote-front --network=azure-vote -e MYSQL_USER=dbuser -e
MYSQL_PASSWORD=Password12 -e MYSQL_DATABASE=azurevote -e MYSQL_HOST=azure-vote-back azure-vote-front:v2
```

Go to `http://localhost:8080` to see the updated application. The application takes a few seconds to initialize. If you get an error, try again.



Tag and push images

Tag the *azure-vote-front* image with the loginServer of the container registry.

If you're using Azure 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, making sure to update the command with your Azure Container Registry login server or public registry hostname.

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

Push the image to your registry. Replace `<acrLoginServer>` with your Azure Container Registry login server name or public registry hostname.

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

Deploy update to Kubernetes

Verify multiple POD replicas

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 don't have multiple pods running the azure-vote-front image, scale the *azure-vote-front* deployment.

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

Update application

To update the application, run the following 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: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:

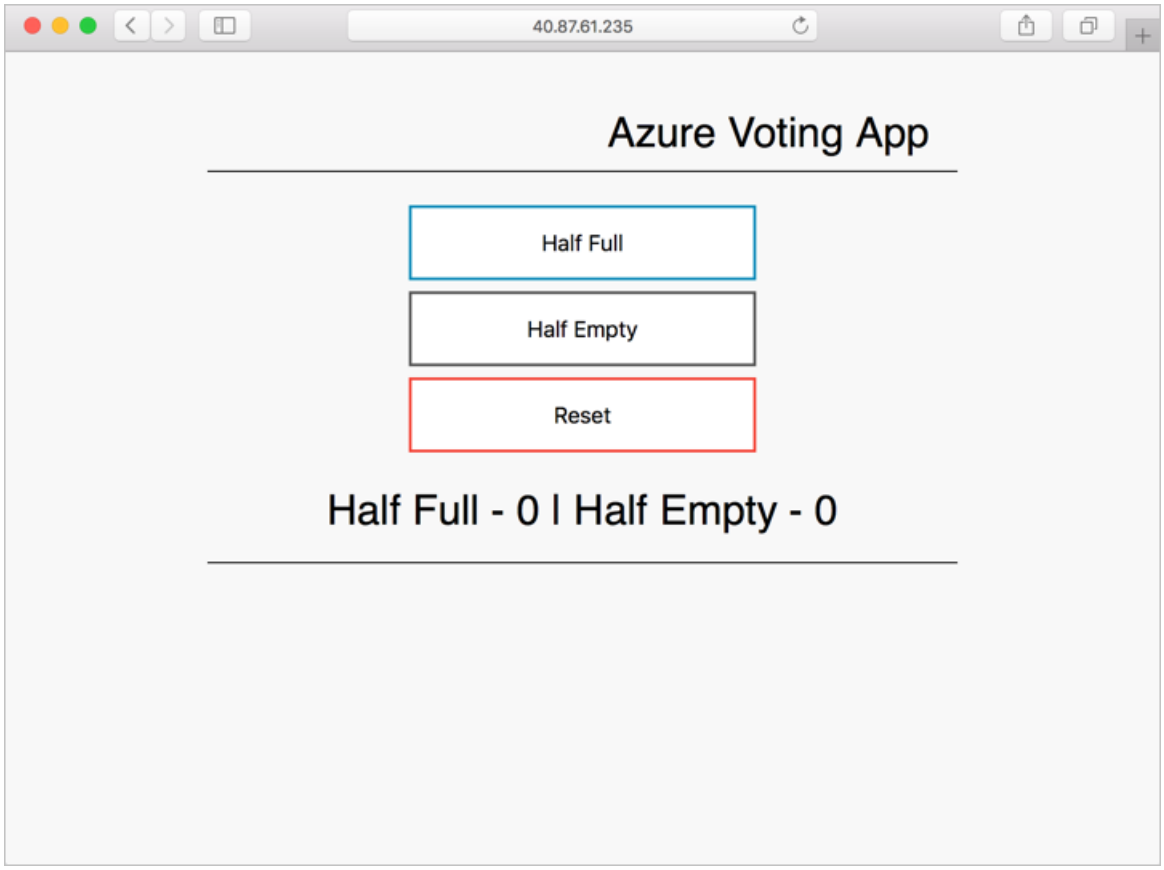
```
NAME                               READY     STATUS    RESTARTS   AGE
azure-vote-back-2978095810-gq9g0   1/1       Running   0          5m
azure-vote-front-1297194256-tpjlg   1/1       Running   0         1m
azure-vote-front-1297194256-tptnx   1/1       Running   0         5m
azure-vote-front-1297194256-zktw9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

Test updated application

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

```
kubectl get service
```

Go to the IP address to see the updated application.



Next steps

In this tutorial, we updated an application and rolled out this update to a Kubernetes cluster. We completed the following tasks:

- 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

6/27/2017 • 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 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.

At minimum, this tutorial requires a Kubernetes cluster with Linux agent nodes, and an Operations Management Suite (OMS) account. If needed, sign up for a free OMS trial.

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
```

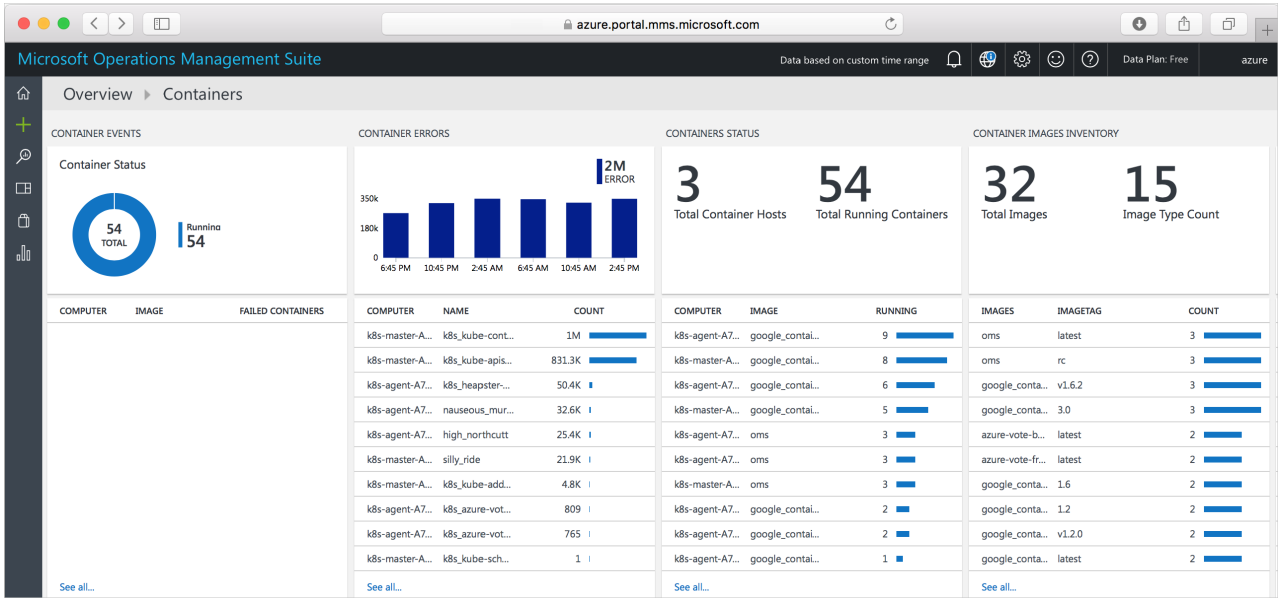
After the agents are running, it takes several minutes for OMS to ingest and process the data.

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 Container Service tutorial - Manage DC/OS

6/27/2017 • 5 min to read • Edit Online

DC/OS provides a distributed platform for running modern and containerized applications. With Azure Container Service, provisioning of a production ready DC/OS cluster is simple and quick. This quick start details basic steps needed to deploy a DC/OS cluster and run basic workload.

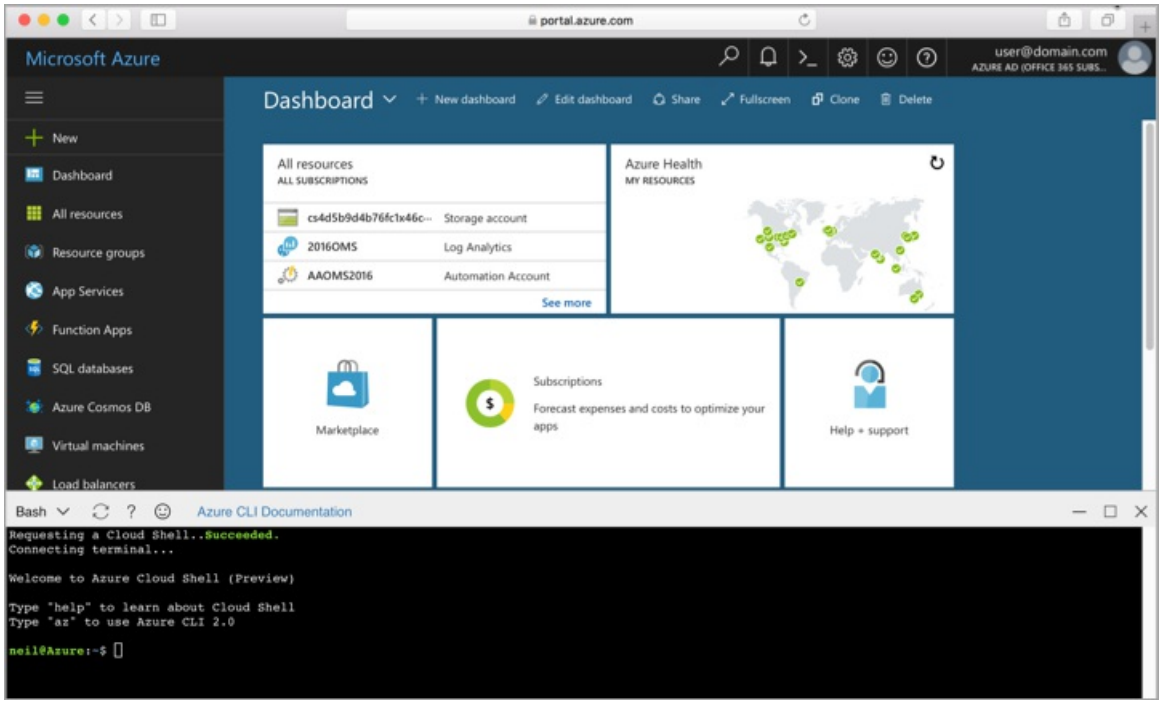
- Create an ACS DC/OS cluster
- Connect to the cluster
- Install the DC/OS CLI
- Deploy an application to the cluster
- Scale an application on the cluster
- Scale the DC/OS cluster nodes
- Basic DC/OS management
- Delete the DC/OS cluster

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



Create DC/OS cluster

First, create a resource group with the az group create command. An Azure resource group is a logical container into 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
```

Next, create a DC/OS cluster with the az acs create command.

The following example creates a DC/OS cluster named *myDCOSCluster* and creates SSH keys if they do not already exist. To use a specific set of keys, use the `--ssh-key-value` option.

```
az acs create \
  --orchestrator-type dcos \
  --resource-group myResourceGroup \
  --name myDCOSCluster \
  --generate-ssh-keys
```

After several minutes, the command completes, and returns information about the deployment.

Connect to DC/OS cluster

Once a DC/OS cluster has been created, it can be accesses through an SSH tunnel. Run the following command to return the public IP address of the DC/OS master. This IP address is stored in a variable and used in the next step.

```
ip=$(az network public-ip list --resource-group myResourceGroup --query "[?contains(name,'dcos-master')].
[ipAddress]" -o tsv)
```

To create the SSH tunnel, run the following command and follow the on-screen instructions. If port 80 is already in use, the command fails. Update the tunneled port to one not in use, such as `85:localhost:80`.

```
sudo ssh -i ~/.ssh/id_rsa -fNL 80:localhost:80 -p 2200 azureuser@$ip
```

Install DC/OS CLI

Install the DC/OS cli using the az acs dcos install-cli command. If you are using Azure CloudShell, the DC/OS CLI is already installed. If you are running the Azure CLI on macOS or Linux, you might need to run the command with sudo.

```
az acs dcos install-cli
```

Before the CLI can be used with the cluster, it must be configured to use the SSH tunnel. To do so, run the following command, adjusting the port if needed.

```
dcos config set core.dcos_url http://localhost
```

Run an application

The default scheduling mechanism for an ACS DC/OS cluster is Marathon. Marathon is used to start an application and manage the state of the application on the DC/OS cluster. To schedule an application through Marathon, create a file named **marathon-app.json**, and copy the following contents into it.

```
{
  "id": "demo-app-private",
  "cmd": null,
  "cpus": 1,
  "mem": 32,
  "disk": 0,
  "instances": 1,
  "container": {
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "protocol": "tcp",
          "name": "80",
          "labels": null
        }
      ]
    },
    "type": "DOCKER"
  }
}
```

Run the following command to schedule the application to run on the DC/OS cluster.

```
dcos marathon app add marathon-app.json
```

To see the deployment status for the app, run the following command.

```
dcos marathon app list
```

When the **TASKS** column value switches from *0/1* to *1/1*, application deployment has completed.

```
ID     MEM  CPUS  TASKS  HEALTH  DEPLOYMENT  WAITING  CONTAINER  CMD
/test   32   1     0/1    ---       ---      False      DOCKER   None
```

Scale Marathon application

In the previous example, a single instance application was created. To update this deployment so that three instances of the application are available, open up the **marathon-app.json** file, and update the instance property to 3.

```
{
  "id": "demo-app-private",
  "cmd": null,
  "cpus": 1,
  "mem": 32,
  "disk": 0,
  "instances": 3,
  "container": {
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "protocol": "tcp",
          "name": "80",
          "labels": null
        }
      ]
    },
    "type": "DOCKER"
  }
}
```

Update the application using the `dcos marathon app update` command.

```
dcos marathon app update demo-app-private < marathon-app.json
```

To see the deployment status for the app, run the following command.

```
dcos marathon app list
```

When the **TASKS** column value switches from *1/3* to *3/1*, application deployment has completed.

```
ID     MEM  CPUS  TASKS  HEALTH  DEPLOYMENT  WAITING  CONTAINER  CMD
/test   32   1     1/3    ---       ---      False      DOCKER   None
```

Run internet accessible app

The ACS DC/OS cluster consists of two node sets, one public which is accessible on the internet, and one private which is not accessible on the internet. The default set is the private nodes, which was used in the last example.

To make an application accessible on the internet, deploy them to the public node set. To do so, give the `acceptedResourceRoles` object a value of `slave_public`.

Create a file named **nginx-public.json** and copy the following contents into it.

```
{
  "id": "demo-app",
  "cmd": null,
  "cpus": 1,
  "mem": 32,
  "disk": 0,
  "instances": 1,
  "container": {
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80,
          "protocol": "tcp",
          "name": "80",
          "labels": null
        }
      ]
    },
    "type": "DOCKER"
  },
  "acceptedResourceRoles": [
    "slave_public"
  ]
}
```

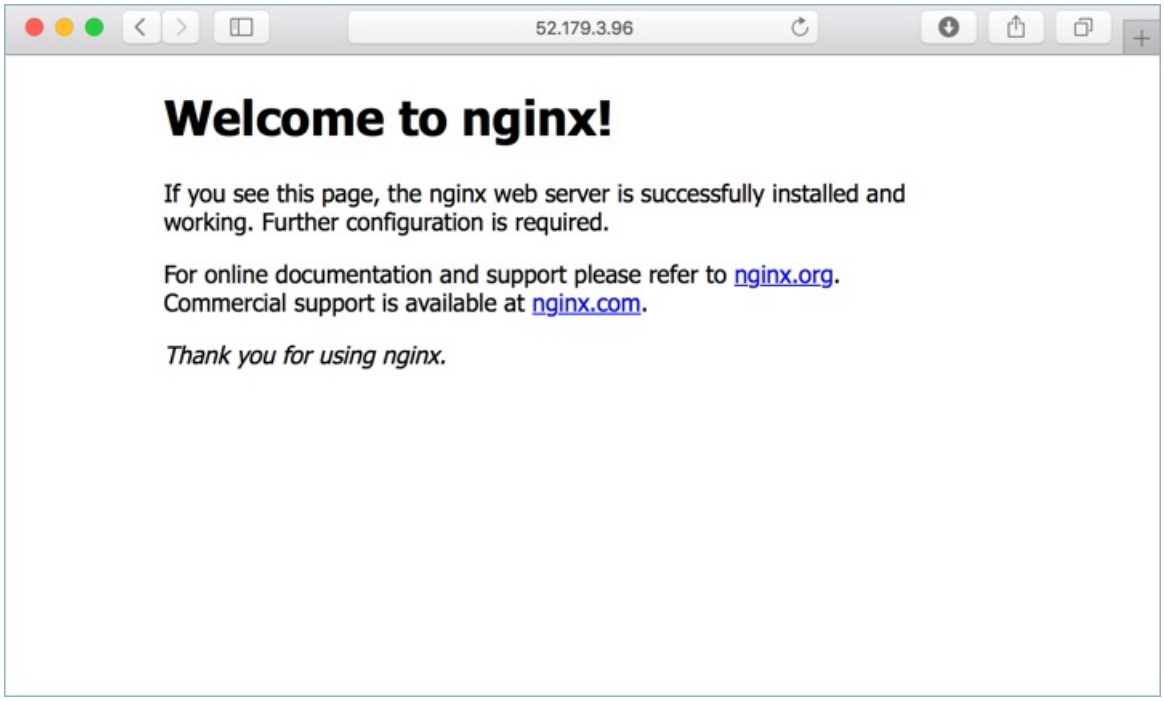
Run the following command to schedule the application to run on the DC/OS cluster.

```
dcos marathon app add nginx-public.json
```

Get the public IP address of the DC/OS public cluster agents.

```
az network public-ip list --resource-group myResourceGroup --query "[?contains(name,'dcos-agent')].
[ipAddress]" -o tsv
```

Browsing to this address returns the default NGINX site.



Scale DC/OS cluster

In the previous examples, an application was scaled to multiple instance. The DC/OS infrastructure can also be scaled to provide more or less compute capacity. This is done with the command.

To see the current count of DC/OS agents, use the az acs show command.

```
az acs show --resource-group myResourceGroup --name myDCOSCluster --query "agentPoolProfiles[0].count"
```

To increase the count to 5, use the az acs scale command.

```
az acs scale --resource-group myResourceGroup --name myDCOSCluster --new-agent-count 5
```

Delete DC/OS cluster

When no longer needed, you can use the az group delete command to remove the resource group, DC/OS cluster, and all related resources.

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

Next steps

In this tutorial, you have learned about basic DC/OS management task including the following.

- Create an ACS DC/OS cluster
- Connect to the cluster
- Install the DC/OS CLI
- Deploy an application to the cluster
- Scale an application on the cluster
- Scale the DC/OS cluster nodes
- Delete the DC/OS cluster

Advance to the next tutorial to learn about load balancing application in DC/OS on Azure.

Load balance applications

Load balance containers in an Azure Container Service DC/OS cluster

6/27/2017 • 3 min to read • Edit Online

In this article, we explore how to create an internal load balancer in a DC/OS managed Azure Container Service using Marathon-LB. This configuration enables you to scale your applications horizontally. It also allows you to take advantage of the public and private agent clusters by placing your load balancers on the public cluster and your application containers on the private cluster. In this tutorial, you:

- Configure a Marathon Load Balancer
- Deploy an application using the load balancer
- Configure and Azure load balancer

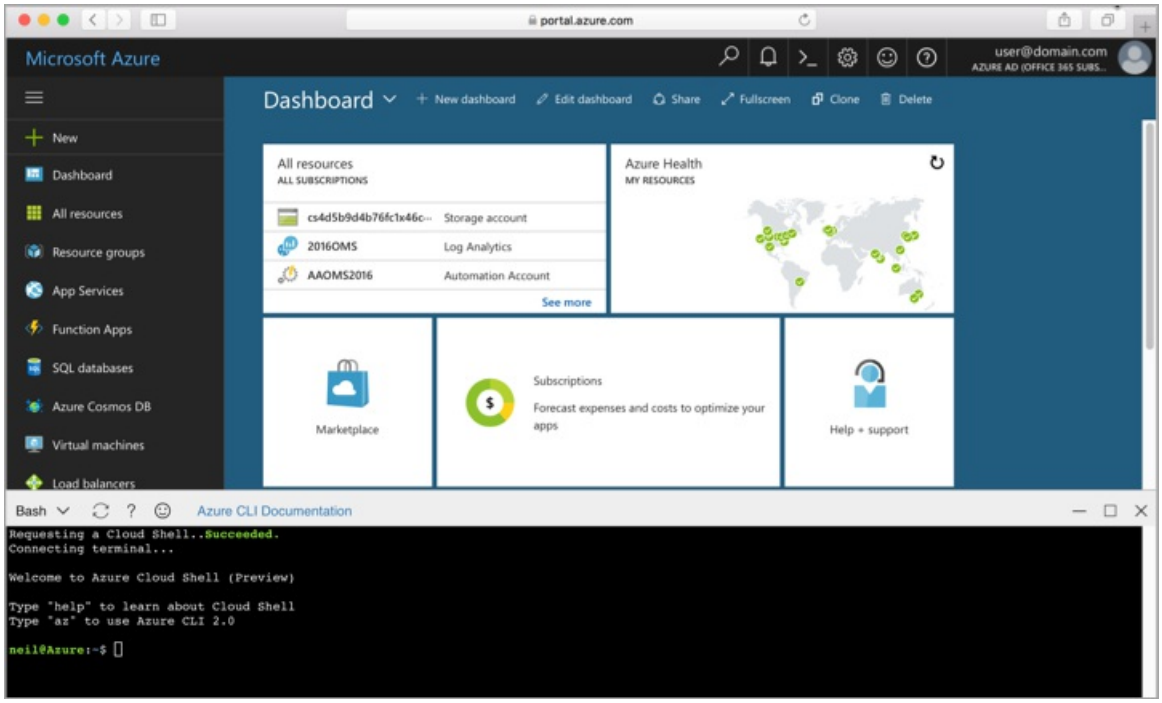
You need an ACS DC/OS cluster to complete the steps in this tutorial. If needed, this script sample can create one for you.

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



Load balancing overview

There are two load-balancing layers in an Azure Container Service DC/OS cluster:

Azure Load Balancer provides public entry points (the ones that end users access). An Azure LB is provided automatically by Azure Container Service and is, by default, configured to expose port 80, 443 and 8080.

The Marathon Load Balancer (marathon-lb) routes inbound requests to container instances that service these requests. As we scale the containers providing our web service, the marathon-lb dynamically adapts. This load balancer is not provided by default in your Container Service, but it is easy to install.

Configure Marathon Load Balancer

Marathon Load Balancer dynamically reconfigures itself based on the containers that you've deployed. It's also resilient to the loss of a container or an agent - if this occurs, Apache Mesos restarts the container elsewhere and marathon-lb adapts.

Run the following command to install the marathon load balancer on the public agent's cluster.

```
dcos package install marathon-lb
```

Deploy load balanced application

Now that we have the marathon-lb package, we can deploy an application container that we wish to load balance.

First, get the FQDN of the publicly exposed agents.

```
az acs list --resource-group myResourceGroup --query "[0].agentPoolProfiles[0].fqdn" --output tsv
```

Next, create a file named *hello-web.json* and copy in the following contents. The `HAPROXY_0_VHOST` label needs to be updated with the FQDN of the DC/OS agents.

```
{
  "id": "web",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "yeasy/simple-web",
      "network": "BRIDGE",
      "portMappings": [
        { "hostPort": 0, "containerPort": 80, "servicePort": 10000 }
      ],
      "forcePullImage":true
    }
  },
  "instances": 3,
  "cpus": 0.1,
  "mem": 65,
  "healthChecks": [{
      "protocol": "HTTP",
      "path": "/",
      "portIndex": 0,
      "timeoutSeconds": 10,
      "gracePeriodSeconds": 10,
      "intervalSeconds": 2,
      "maxConsecutiveFailures": 10
  }],
  "labels":{
    "HAPROXY_GROUP":"external",
    "HAPROXY_0_VHOST":"YOUR FQDN",
    "HAPROXY_0_MODE":"http"
  }
}
```

Use the DC/OS CLI to run the application. By default Marathon deploys the the applicaton to the private cluster. This means that the above deployment is only accessible via your load balancer, which is usually the desired behavior.

```
dcos marathon app add hello-web.json
```

Once the application has been deployed, browse to the FQDN of the agent cluster to view load balanced application.

Configure Azure Load Balancer

By default, Azure Load Balancer exposes ports 80, 8080, and 443. If you're using one of these three ports (as we do in the above example), then there is nothing you need to do. You should be able to hit your agent load balancer's FQDN, and each time you refresh, you'll hit one of your three web servers in a round-robin fashion.

If you use a different port, you need to add a round-robin rule and a probe on the load balancer for the port that

you used. You can do this from the Azure CLI, with the commands `azure network lb rule create` and `azure network lb probe create`.

Next steps

In this tutorial, you learned about load balancing in ACS with both the Marathon and Azure load balancers including the following actions:

- Configure a Marathon Load Balancer
- Deploy an application using the load balancer
- Configure and Azure load balancer

Advance to the next tutorial to learn about integrating Azure storage with DC/OS in Azure.

Mount Azure File Share in DC/OS cluster

Create and mount a file share to a DC/OS cluster

6/27/2017 • 4 min to read • Edit Online

This tutorial details how to create a file share in Azure and mount it on each agent and master of the DC/OS cluster. Setting up a file share makes it easier to share files across your cluster such as configuration, access, logs, and more. The following tasks are completed in this tutorial:

- Create an Azure storage account
- Create a file share
- Mount the share in the DC/OS cluster

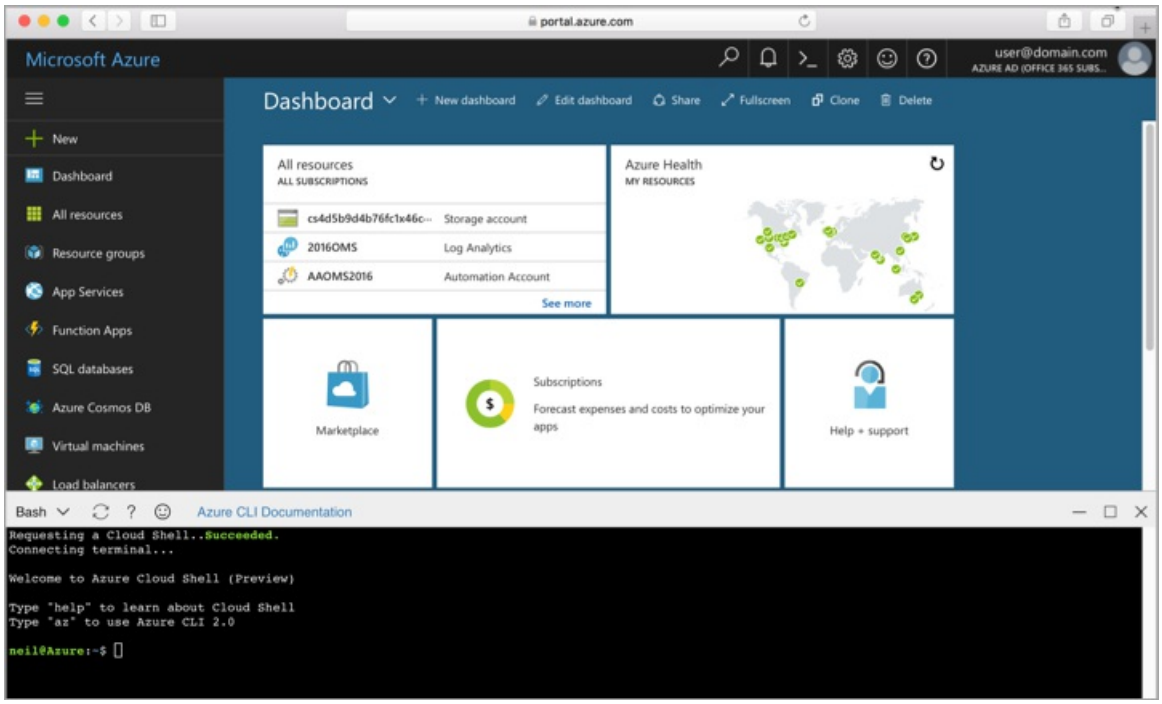
You need an ACS DC/OS cluster to complete the steps in this tutorial. If needed, this script sample can create one for you.

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



Create a file share on Microsoft Azure

Before using an Azure file share with an ACS DC/OS cluster, the storage account and file share must be created. Run the following script to create the storage and file share. Update the parameters with thoes from your environment.

```
# Change these four parameters
DCOS_PERS_STORAGE_ACCOUNT_NAME=mystorageaccount$RANDOM
DCOS_PERS_RESOURCE_GROUP=myResourceGroup
DCOS_PERS_LOCATION=eastus
DCOS_PERS_SHARE_NAME=dcosshare

# Create the storage account with the parameters
az storage account create -n $DCOS_PERS_STORAGE_ACCOUNT_NAME -g $DCOS_PERS_RESOURCE_GROUP -l
$DCOS_PERS_LOCATION --sku Standard_LRS

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

# Create the share
az storage share create -n $DCOS_PERS_SHARE_NAME
```

Mount the share in your cluster

Next, the file share needs to be mounted on every virtual machine inside your cluster. This task is completed using the cifs tool/protocol. The mount operation can be completed manually on each node of the cluster, or by running a script against each node in the cluster.

In this example, two scripts are run, one to mount the Azure file share, and a second to run this script on each node of the DC/OS cluster.

First, the Azure storage account name, and access key are needed. Run the following commands to get this information. Take note of each, these values are used in a later step.

Storage account name:

```
STORAGE_ACCT=$(az storage account list --resource-group myResourceGroup --query "[?
contains(name,'mystorageaccount')].[name]" -o tsv)
echo $STORAGE_ACCT
```

Storage account access key:

```
az storage account keys list --resource-group myResourceGroup --account-name $STORAGE_ACCT --query "[0].value"
-o tsv
```

Next, get the FQDN of the DC/OS master and store it in a variable.

```
FQDN=$(az acs list --resource-group myResourceGroup --query "[0].masterProfile.fqdn" --output tsv)
```

Copy your private key to the master node. This key is needed to create an ssh connection with all nodes in the cluster. Update the user name if a non-default value was used when creating the cluster.

```
scp ~/.ssh/id_rsa azureuser@$FQDN:~/.ssh
```

Create an SSH connection with the master (or the first master) of your DC/OS-based cluster. Update the user name if a non-default value was used when creating the cluster.

```
ssh azureuser@$FQDN
```

Create a file named **cifsMount.sh**, and copy the following contents into it.

This script is used to mount the Azure file share. Update the `STORAGE_ACCT_NAME` and `ACCESS_KEY` variables with the information collected earlier.

```
#!/bin/bash

# Azure storage account name and access key
STORAGE_ACCT_NAME=mystorageaccount
ACCESS_KEY=mystorageaccountKey

# Install the cifs utils, should be already installed
sudo apt-get update && sudo apt-get -y install cifs-utils

# Create the local folder that will contain our share
if [ ! -d "/mnt/share/dcosshare" ]; then sudo mkdir -p "/mnt/share/dcosshare" ; fi

# Mount the share under the previous local folder created
sudo mount -t cifs //$STORAGE_ACCT_NAME.file.core.windows.net/dcosshare /mnt/share/dcosshare -o
vers=3.0,username=$STORAGE_ACCT_NAME,password=$ACCESS_KEY,dir_mode=0777,file_mode=0777
```

Create a second file named **getNodesRunScript.sh** and copy the following contents into the file.

This script discovers all cluster nodes, and then runs the **cifsMount.sh** script to mount the file share on each.

```
#!/bin/bash

# Install jq used for the next command
sudo apt-get install jq -y

# Get the IP address of each node using the mesos API and store it inside a file called nodes
curl http://leader.mesos:1050/system/health/v1/nodes | jq '.nodes[].host_ip' | sed 's/\"//g' | sed '/172/d' >
nodes

# From the previous file created, run our script to mount our share on each node
cat nodes | while read line
do
  ssh `whoami`@$line -o StrictHostKeyChecking=no < ./cifsMount.sh
  done
```

Run the script to mount the Azure file share on all nodes of the cluster.

```
sh ./getNodesRunScript.sh
```

The file share is now accessible at `/mnt/share/dcosshare` on each node of the cluster.

Next steps

In this tutorial an Azure file share was made available to a DC/OS cluster using the steps:

- Create an Azure storage account
- Create a file share
- Mount the share in the DC/OS cluster

Advance to the next tutorial to learn about integrating an Azure Container Registry with DC/OS in Azure.

Load balance applications

Use ACR with a DC/OS cluster to deploy your application

7/3/2017 • 4 min to read • Edit Online

In this article, we explore how to use Azure Container Registry with a DC/OS cluster. Using ACR allows you to privately store and manage container images. This tutorial covers the following tasks:

- Deploy Azure Container Registry (if needed)
- Configure ACR authentication on a DC/OS cluster
- Uploaded an image to the Azure Container Registry
- Run a container image from the Azure Container Registry

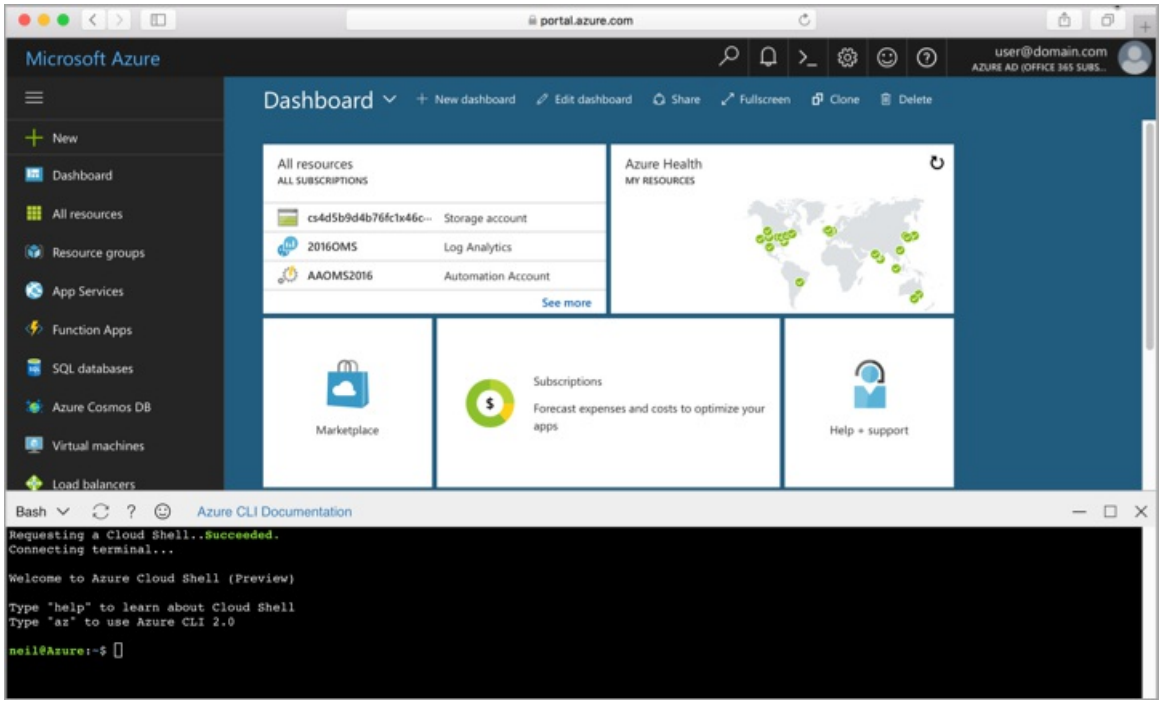
You need an ACS DC/OS cluster to complete the steps in this tutorial. If needed, this script sample can create one for you.

This tutorial requires the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to upgrade, see Install Azure CLI 2.0.

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 all of the steps in this topic:



Deploy Azure Container Registry

If needed, create an Azure Container registry with the az acr create command.

The following example creates a registry with a randomly generate name. The registry is also configured with an admin account using the `--admin-enabled` argument.

```
az acr create --resource-group myResourceGroup --name myContainerRegistry$RANDOM --sku Basic --admin-enabled
true
```

Once the registry has been created, the Azure CLI outputs data similar to the following. Take note of the `name` and `loginServer`, these are used in later steps.

```
{
  "adminUserEnabled": false,
  "creationDate": "2017-06-06T03:40:56.511597+00:00",
  "id": "/subscriptions/f2799821-a08a-434e-9128-
454ec4348b10/resourcegroups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/myContainerRegist
ry23489",
  "location": "eastus",
  "loginServer": "mycontainerregistry23489.azurecr.io",
  "name": "myContainerRegistry23489",
  "provisioningState": "Succeeded",
  "sku": {
    "name": "Basic",
    "tier": "Basic"
  },
  "storageAccount": {
    "name": "mycontainerregistr034017"
  },
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```

Get the container registry credentials using the az acr credential show command. Substitute the `--name` with the one noted in the last step. Take note of one password, it is needed in a later step.

```
az acr credential show --name myContainerRegistry23489
```

For more information on Azure Container Registry, see Introduction to private Docker container registries.

Manage ACR authentication

The conventional way to push and pull image from a private registry is to first authenticate with the registry. To do so, you would use the `docker login` command on any client that needs to access the private registry. Because a DC/OS cluster can contain many nodes, all of which need to be authenticated with the ACR, it is helpful to automate this process across each node.

Create shared storage

This process uses an Azure file share that has been mounted on each node in the cluster. If you have not already set up shared storage, see Setup a file share inside a DC/OS cluster.

Configure ACR authentication

First, get the FQDN of the DC/OS master and store it in a variable.

```
FQDN=$(az acs list --resource-group myResourceGroup --query "[0].masterProfile.fqdn" --output tsv)
```

Create an SSH connection with the master (or the first master) of your DC/OS-based cluster. Update the user name if a non-default value was used when creating the cluster.

```
ssh azureuser@$FQDN
```

Run the following command to login to the Azure Container Registry. Replace the `--username` with the name of the container registry, and the `--password` with one of the provided passwords. Replace the last argument *mycontainerregistry.azurecr.io* in the example with the loginServer name of the container registry.

This command stores the authentication values locally under the `~/.docker` path.

```
docker -H tcp://localhost:2375 login --username=myContainerRegistry23489 --
password=//=ls++q/m+w+pQDb/xCi0OhD=2c/hST mycontainerregistry.azurecr.io
```

Create a compressed file that contains the container registry authentication values.

```
tar czf docker.tar.gz .docker
```

Copy this file to the cluster shared storage. This step makes the file available on all nodes of the DC/OS cluster.

```
cp docker.tar.gz /mnt/share/dcosshare
```

Upload image to ACR

Now from a development machine, or any other system with Docker installed, create an image and upload it to the Azure Container Registry.

Create a container from the Ubuntu image.

```
docker run ubunut --name base-image
```

Now capture the container into a new image. The image name needs to include the `loginServer` name of the container registrywith a format of `loginServer/imageName`.

```
docker -H tcp://localhost:2375 commit base-image mycontainerregistry30678.azurecr.io/dcos-demo
```

Login into the Azure Container Registry. Replace the name with the loginServer name, the --username with the name of the container registry, and the --password with one of the provided passwords.

```
docker login --username=myContainerRegistry23489 --password=//=ls++q/m+w+pQDb/xCi0OhD=2c/hST
mycontainerregistry2675.azurecr.io
```

Finally, upload the image to the ACR registry. This example uploads an image named dcos-demo.

```
docker push mycontainerregistry30678.azurecr.io/dcos-demo
```

Run an image from ACR

To use an image from the ACR registry, create a file names *acrDemo.json* and copy the following text into it. Replace the image name with the container registry loginServer name and image name, for example `loginServer/imageName`. Take note of the `uris` property. This property holds the location of the container registry authentication file, which in this case is the Azure file share that is mounted on each node in the DC/OS cluster.

```
{
  "id": "myapp",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "mycontainerregistry30678.azurecr.io/dcos-demo",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80,
          "protocol": "tcp",
          "name": "80",
          "labels": null
        }
      ],
      "forcePullImage":true
    }
  },
  "instances": 3,
  "cpus": 0.1,
  "mem": 65,
  "healthChecks": [{
      "protocol": "HTTP",
      "path": "/",
      "portIndex": 0,
      "timeoutSeconds": 10,
      "gracePeriodSeconds": 10,
      "intervalSeconds": 2,
      "maxConsecutiveFailures": 10
  }],
  "uris":  [
       "file:///mnt/share/dcosshare/docker.tar.gz"
   ]
}
```

Deploy the application with the DC/OC CLI.

```
dcos marathon app add acrDemo.json
```

Next steps

In this tutorial you have configure DC/OS to use Azure Container Registry including the following tasks:

- Deploy Azure Container Registry (if needed)
- Configure ACR authentication on a DC/OS cluster
- Uploaded an image to the Azure Container Registry
- Run a container image from the Azure Container Registry

Azure CLI Samples for Azure Container Service

6/27/2017 • 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 DC/OS cluster	Create a DC/OS cluster for Linux based containers.
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

6/27/2017 • 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.

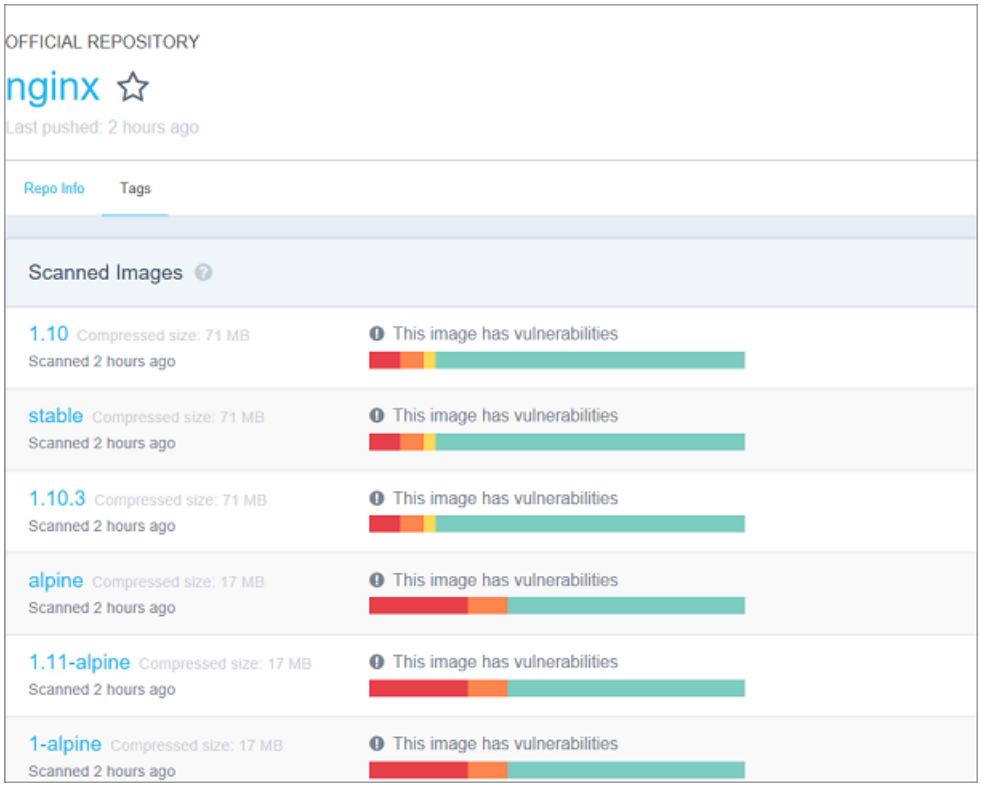
Image security

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.



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 `kubectl` 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

6/27/2017 • 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 resource group in the subscription used to deploy the Kubernetes cluster, or (less restrictively) 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.

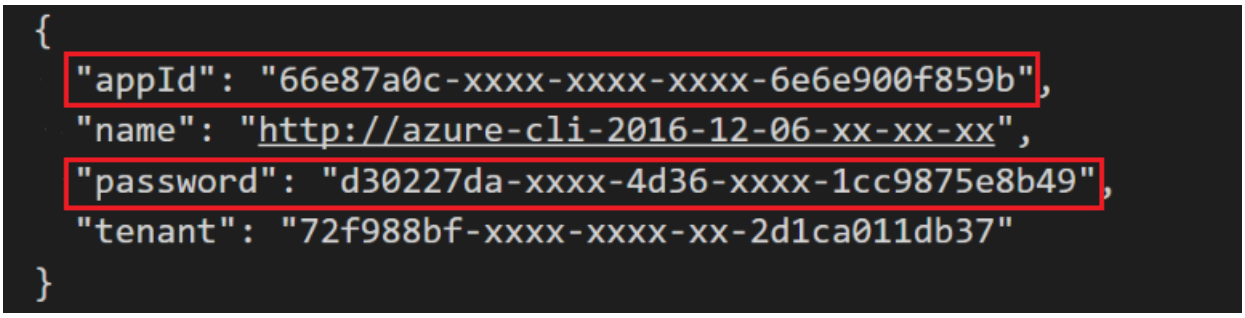
```
az login

az account set --subscription "mySubscriptionID"

az group create -n "myResourceGroupName" -l "westus"

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

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



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.

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.

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

6/27/2017 • 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 `kubectl` 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 `kubectl` on your computer.

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

Install kubectl

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/kubectl]

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

Alternatively, you can download the latest `kubectl` client directly from the Kubernetes releases page. For more information, see Installing and Setting up kubectl.

Download cluster credentials

Once you have `kubectl` 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

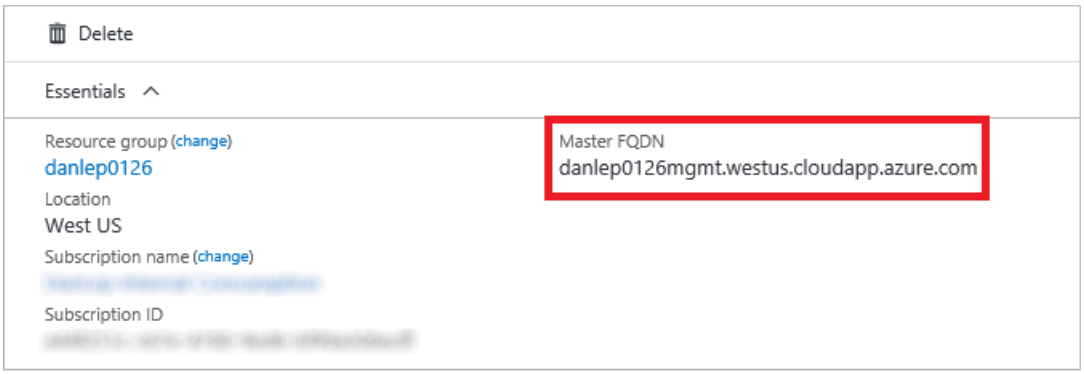
To use the DC/OS and Docker Swarm clusters deployed by Azure Container Service, follow these instructions to create a SSH tunnel from your local Linux, macOS, or Windows system.

> **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.



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

3. 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
```

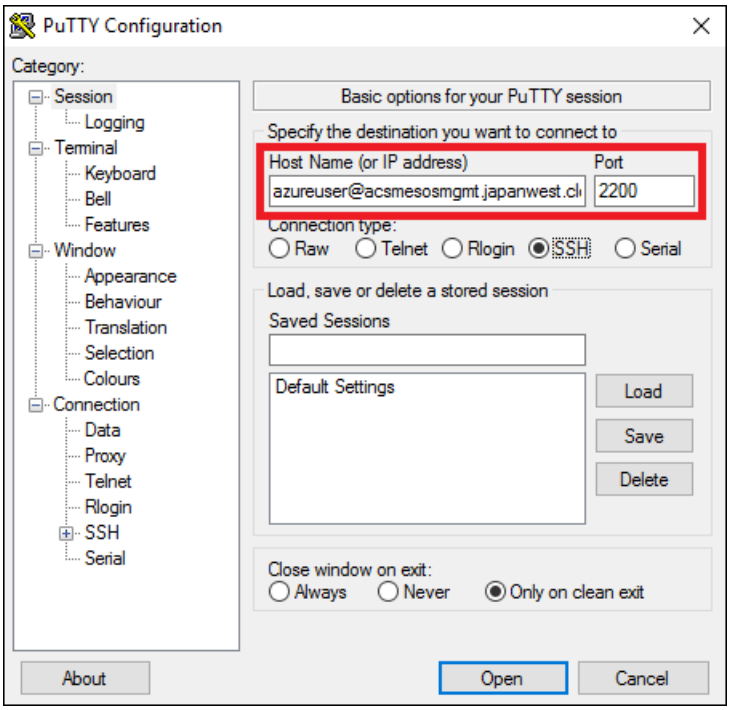
Run Docker commands that tunnel to the Docker Swarm cluster. For example:

```
docker info
```

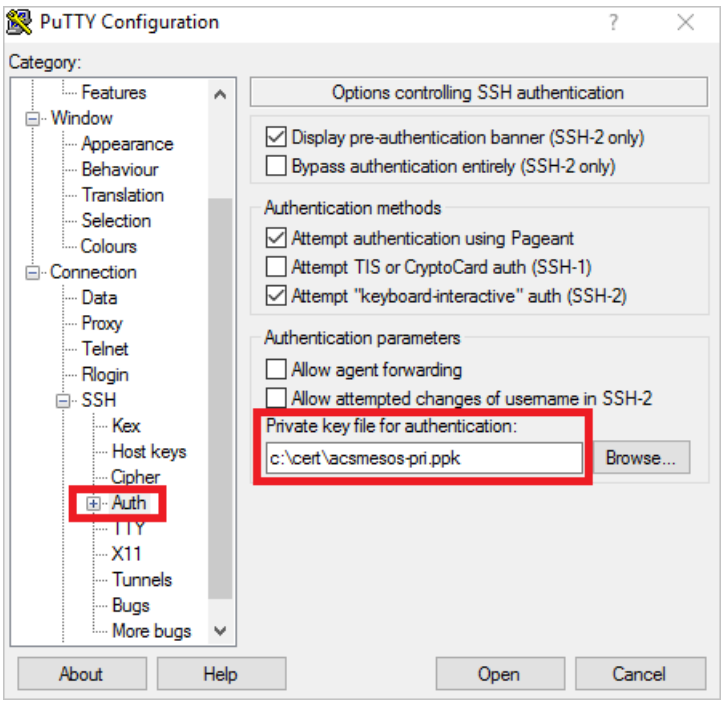
Create an SSH tunnel on Windows

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.

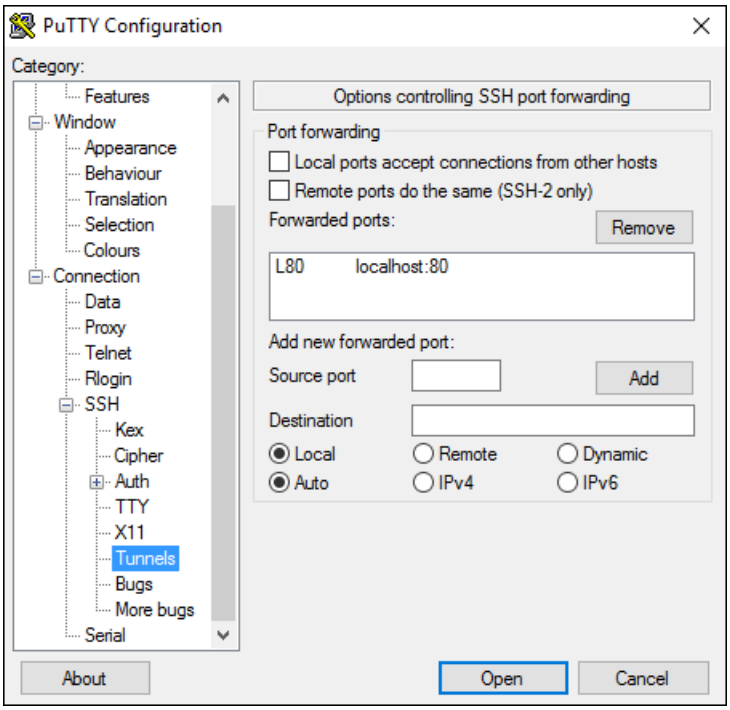


5. Select **SSH > Tunnels** and configure the following forwarded ports:

   - **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.

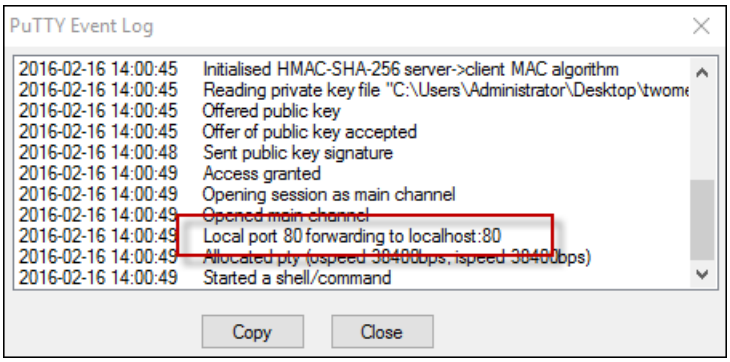
     > **NOTE**
     > Port 80 must not be in use when you create this tunnel.



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

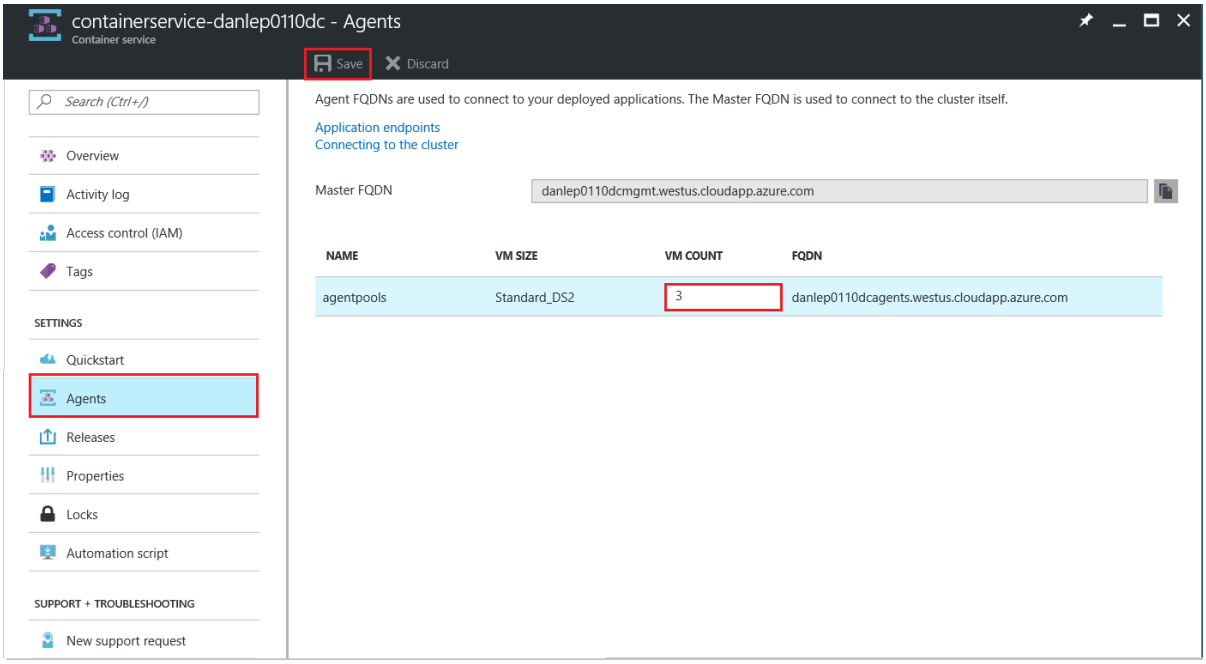
6/27/2017 • 2 min to read • Edit Online

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.



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:

```
azure 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.

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

6/27/2017 • 5 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, ACR, and Azure DNS to create a live CI/CD developer pipeline using Draft.

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

```
az acr create -g draft -n draftacs --sku Basic --admin-enabled true -l eastus
```

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-acs --dns-prefix draft-cluster --orchestrator-type
kubernetes
```

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

The installation instructions for Draft are in the Draft repository. They are relatively simple, but do require some configuration, as it depends on Helm to create and deploy a Helm chart into the Kubernetes cluster.

1. Download and install Helm.
2. Use Helm to search for and install `stable/traefik`, and 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.

Wire up deployment domain

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 `'*'` 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.

4. Create an A record-set entry for your deployment domain mapping to the `ingress` IP from step 2 of the previous section.

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

   The output looks something like:

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

5. 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, `draft`)
   - your registry key, or password, from `az acr credential show -n <registry name> --output tsv --query "passwords[0].value"`.
   - the root deployment domain that you have configured to map to the Kubernetes ingress external IP address (here, `squillace.io`)

   Call `draft init` and the configuration process prompts you for the values above. The process looks something like the following the first time you run it.

```
draft init
Creating pack ruby...
Creating pack node...
Creating pack gradle...
Creating pack maven...
Creating pack php...
Creating pack python...
Creating pack dotnetcore...
Creating pack golang...
$DRAFT_HOME has been configured at /Users/ralphsquillace/.draft.

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

1. Enter your Docker registry URL (e.g. docker.io, quay.io, myregistry.azurecr.io): draft.azurecr.io
2. Enter your username: draft
3. Enter your password:
4. Enter your org where Draft will push images [draft]: draft
5. Enter your top-level domain for ingress (e.g. draft.example.com): squillace.io
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 Python example. Change into the examples/Python directory, and type `draft create` to build the application. It should look like the following example.

```
$ draft create
--> Python app detected
--> 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 begins like the following example.

```
$ draft up
--> Building Dockerfile
Step 1 : FROM python:onbuild
onbuild: Pulling from library/python
10a267c67f42: Pulling fs layer
fb5937da9414: Pulling fs layer
9021b2326a1e: Pulling fs layer
dbed9b09434e: Pulling fs layer
ea8a37f15161: Pulling fs layer
<snip>
```

and when successful ends with something similar to the following example.

```
ab68189731eb: Pushed
53c0ab0341bee12d01be3d3c192fbd63562af7f1: digest:
sha256:bb0450ec37acf67ed461c1512ef21f58a500ff9326ce3ec623ce1e4427df9765 size: 2841
--> Deploying to Kubernetes
--> Status: DEPLOYED
--> Notes:

  http://gangly-bronco.squillace.io to access your application

Watching local files for changes...
```

Whatever your chart's name is, you can now `curl http://gangly-bronco.squillace.io` to receive the reply, `Hello World!`.

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.

Manage an Azure Container Service DC/OS cluster through the Marathon web UI

6/27/2017 • 2 min to read • Edit Online

DC/OS provides an environment for deploying and scaling clustered workloads, while abstracting the underlying hardware. On top of DC/OS, there is a framework that manages scheduling and executing compute workloads.

While frameworks are available for many popular workloads, this document describes how to get started deploying containers with Marathon.

Prerequisites

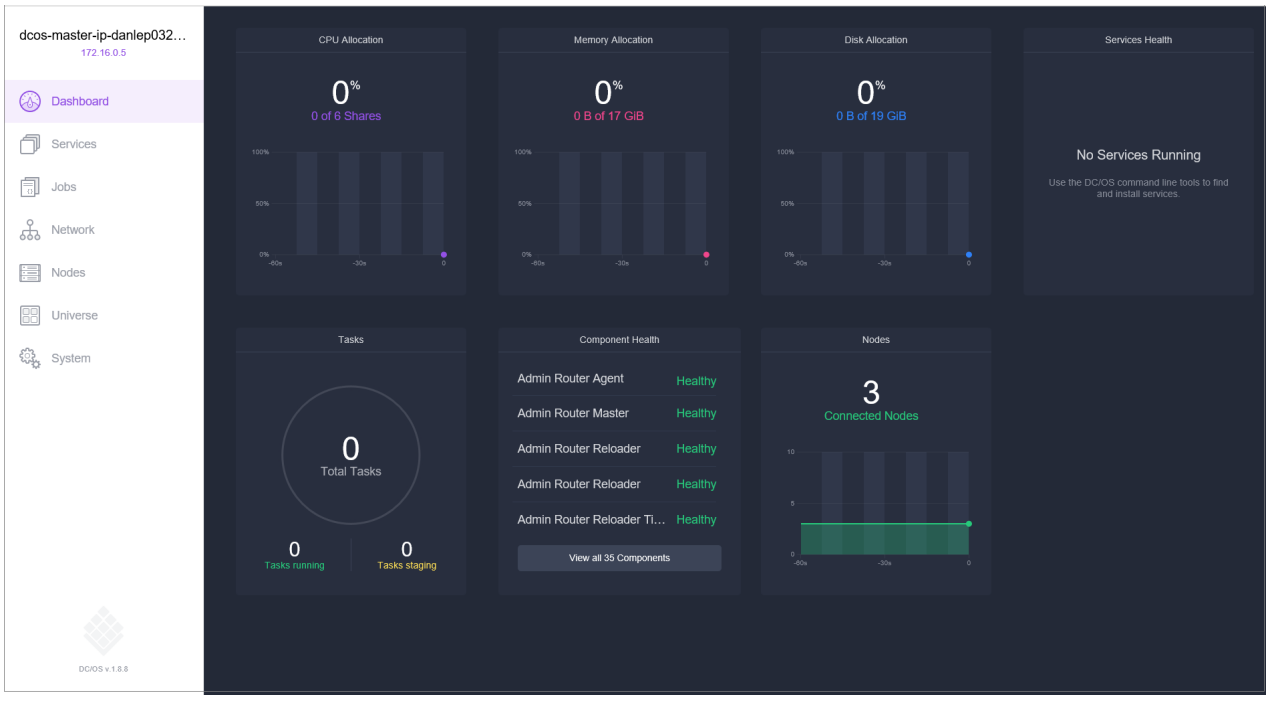
Before working through these examples, you need a DC/OS cluster that is configured in Azure Container Service. You also need to have remote connectivity to this cluster. For more information on these items, see the following articles:

- Deploy an Azure Container Service cluster
- Connect to an Azure Container Service cluster

> **NOTE**
>
> This article assumes you are tunneling to the DC/OS cluster through your local port 80.

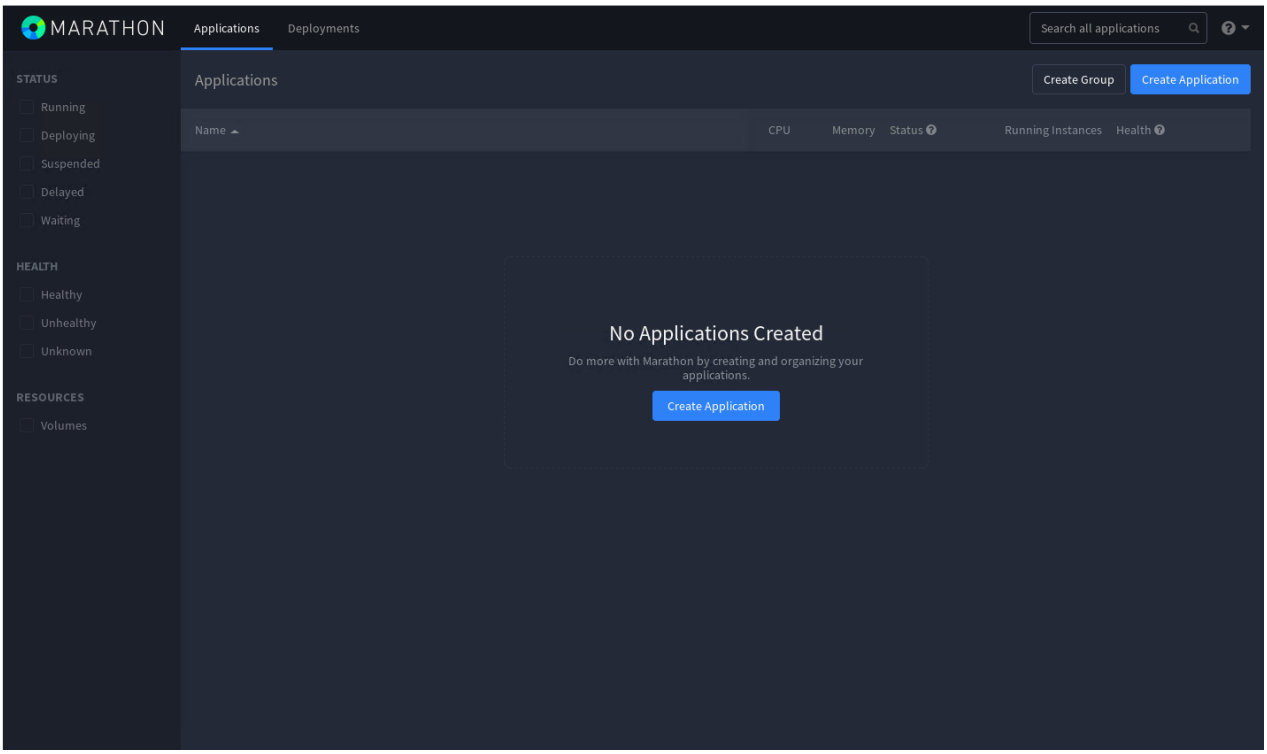
Explore the DC/OS UI

With a Secure Shell (SSH) tunnel established, browse to http://localhost/. This loads the DC/OS web UI and shows information about the cluster, such as used resources, active agents, and running services.



Explore the Marathon UI

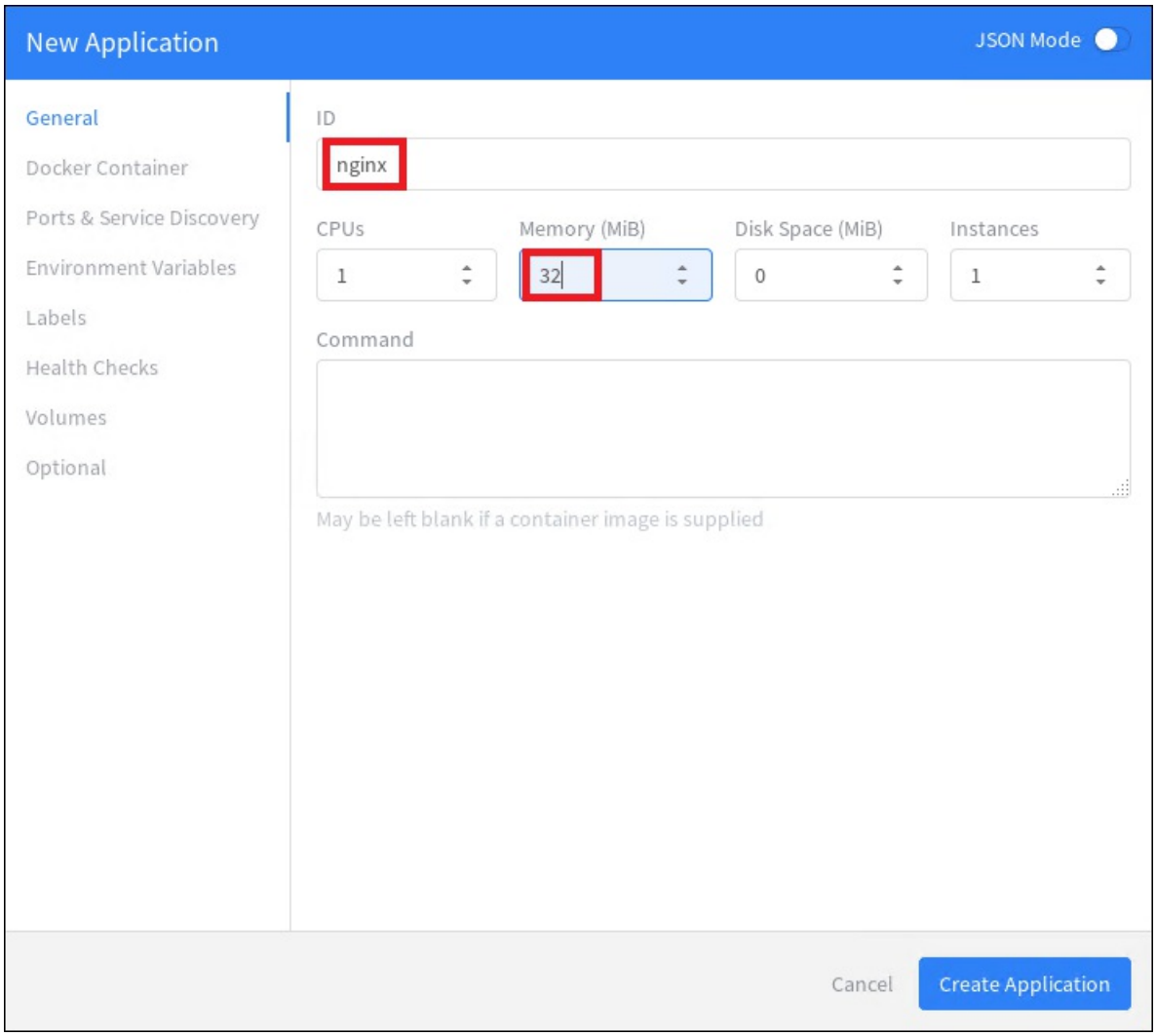
To see the Marathon UI, browse to http://localhost/marathon. From this screen, you can start a new container or another application on the Azure Container Service DC/OS cluster. You can also see information about running containers and applications.

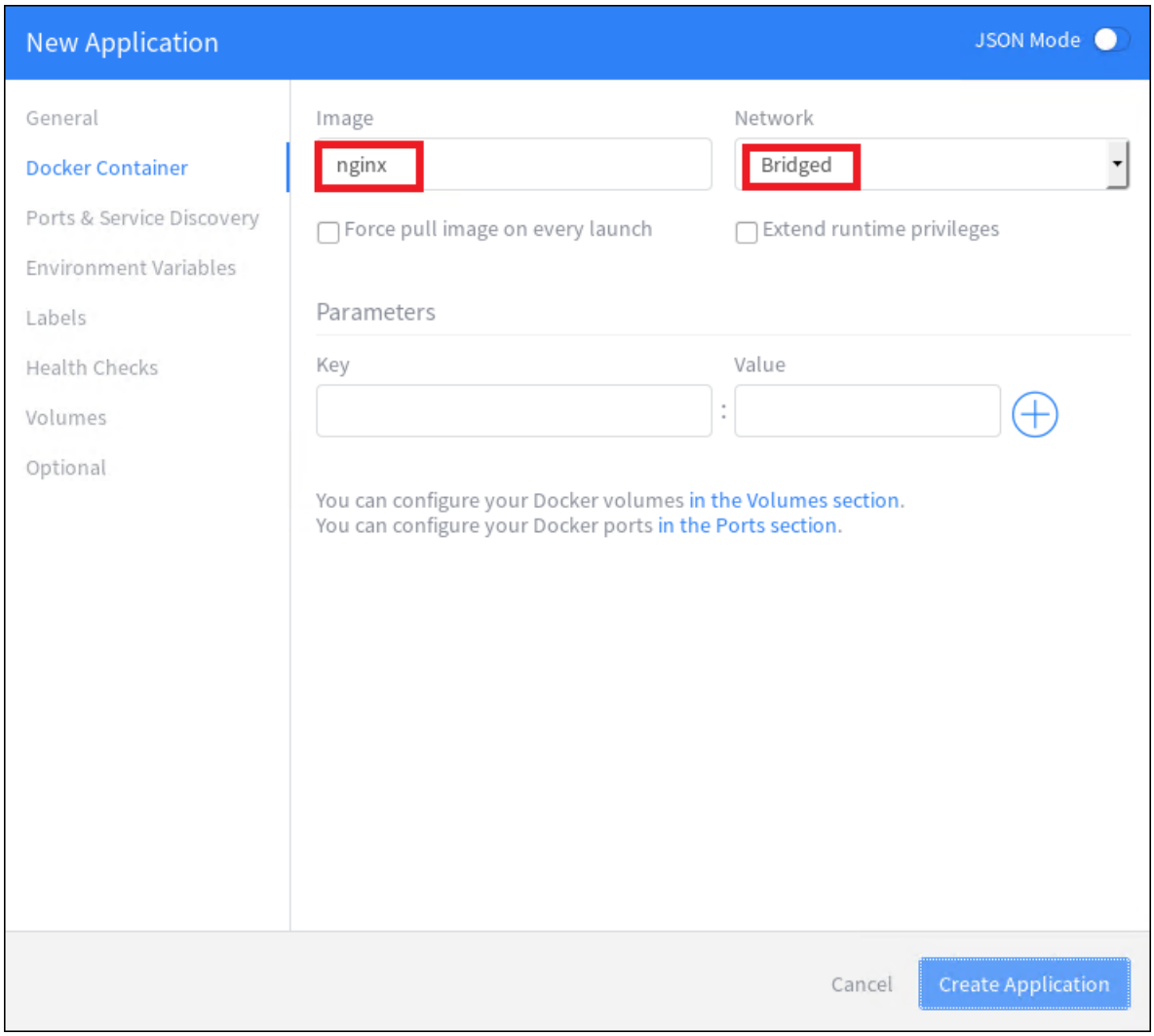


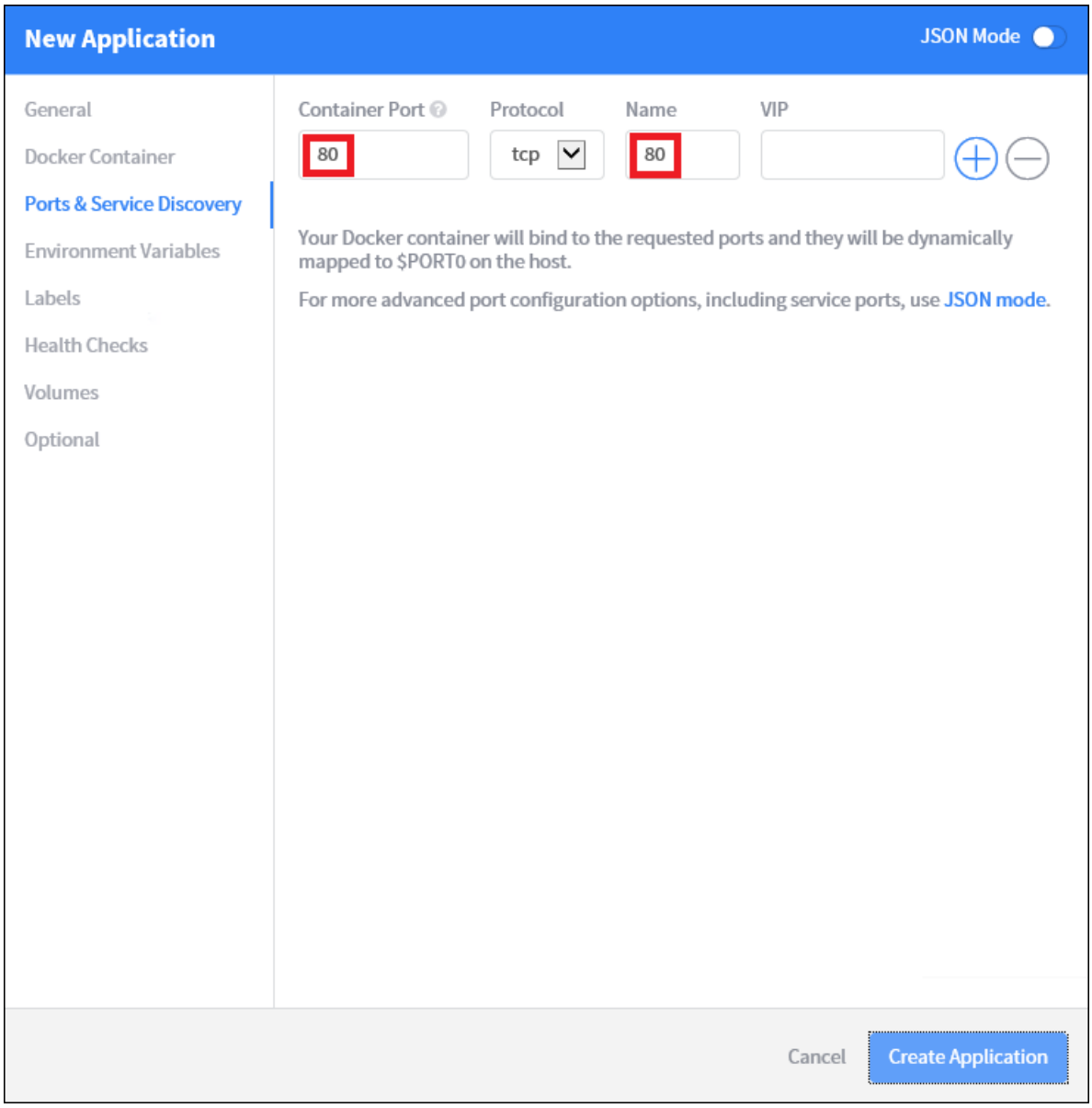
Deploy a Docker-formatted container

To deploy a new container by using Marathon, click **Create Application**, and enter the following information into the form tabs:

FIELD	VALUE
ID	nginx
Memory	32
Image	nginx
Network	Bridged
Host Port	80
Protocol	TCP

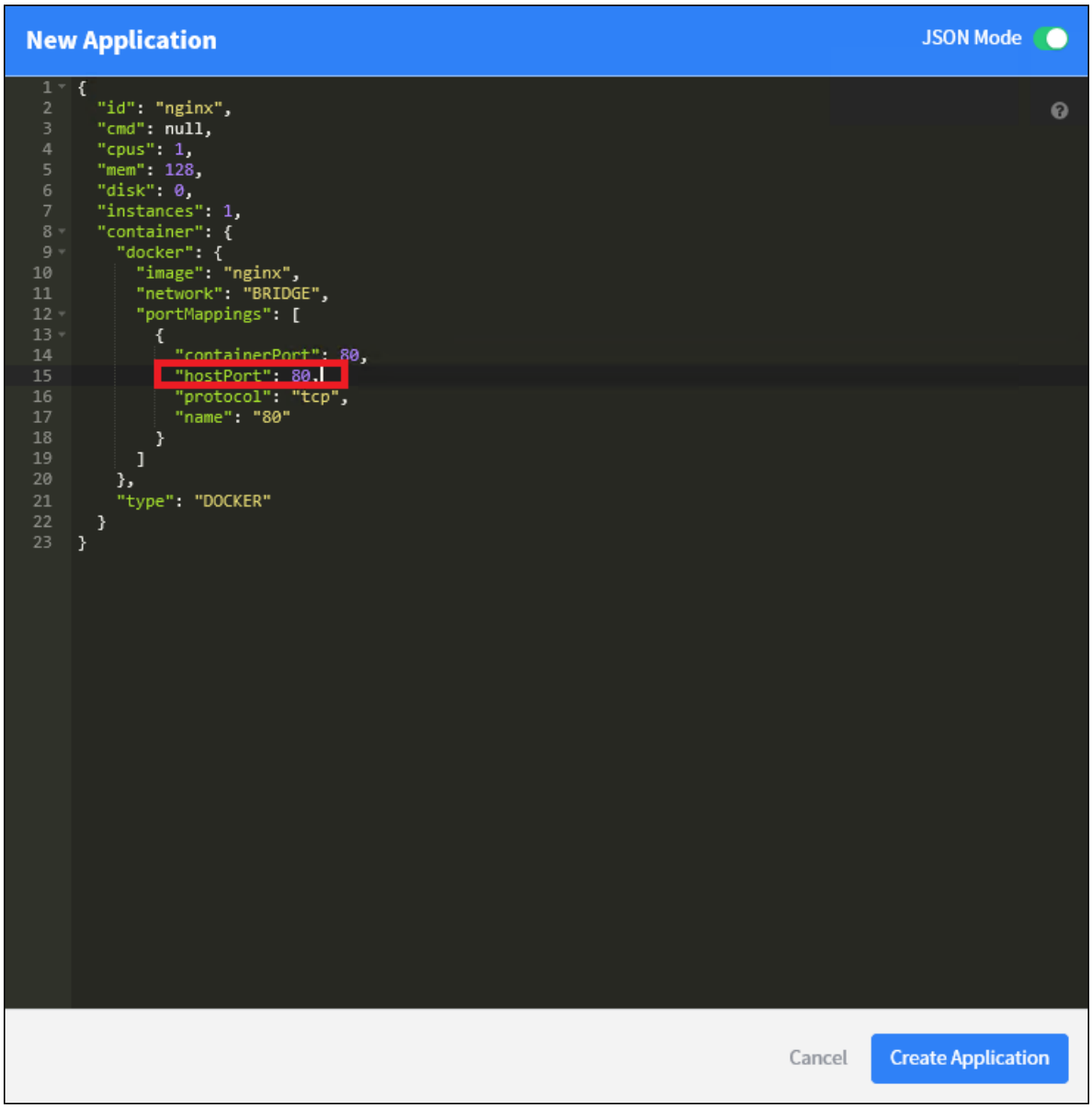

New Application
JSON Mode
General
Docker Container
Ports & Service Discovery
Environment Variables
Labels
Health Checks
Volumes
Optional
ID
nginx
CPUs
1
Memory (MiB)
32
Disk Space (MiB)
0
Instances
1
Command
May be left blank if a container image is supplied
Cancel
Create Application


New Application
JSON Mode
General
Docker Container
Ports & Service Discovery
Environment Variables
Labels
Health Checks
Volumes
Optional
Image
nginx
Network
Bridged
Force pull image on every launch
Extend runtime privileges
Parameters
Key
Value
:
You can configure your Docker volumes in the Volumes section.
You can configure your Docker ports in the Ports section.
Cancel
Create Application

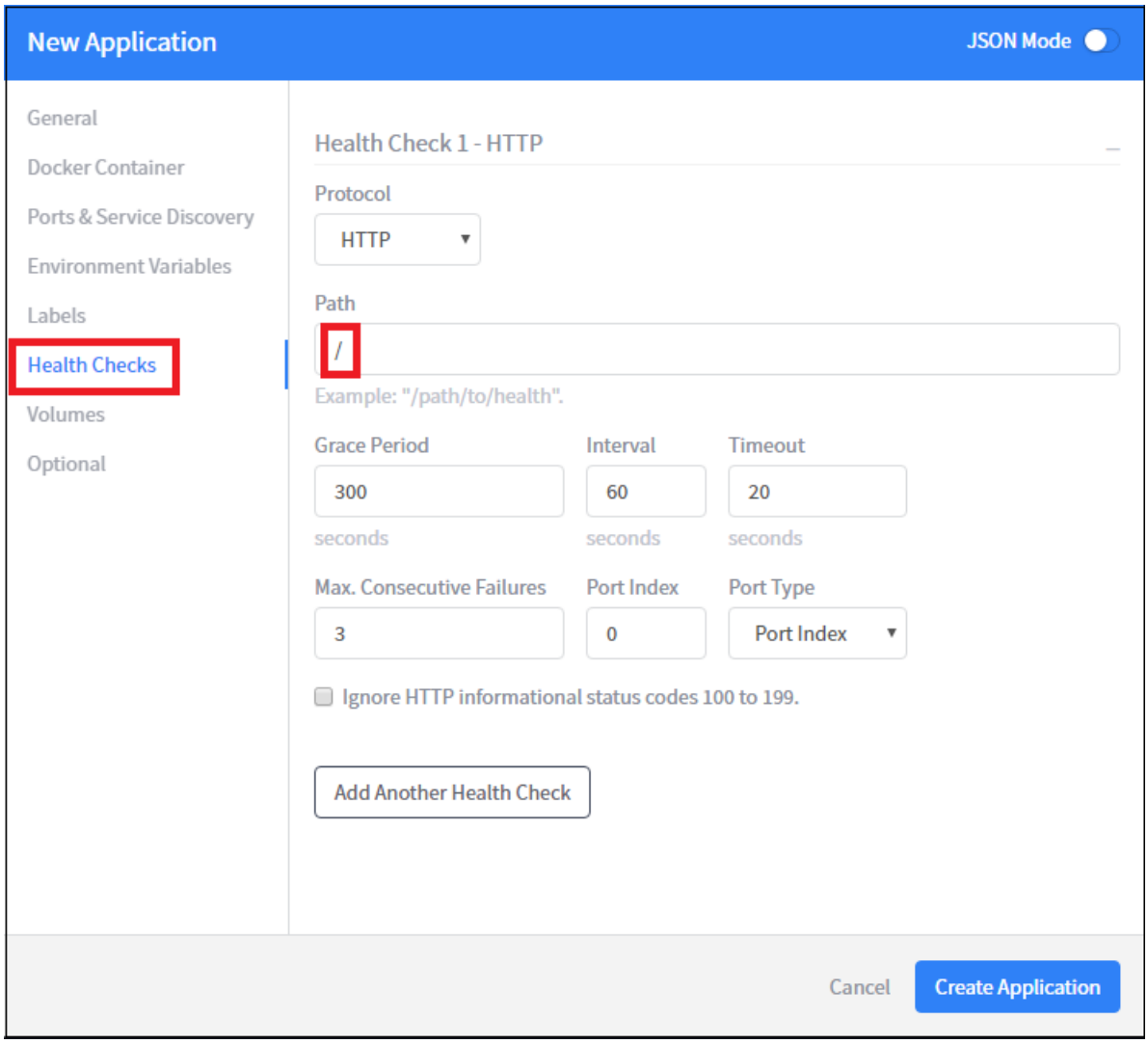


If you want to statically map the container port to a port on the agent, you need to use JSON Mode. To do so, switch the New Application wizard to **JSON Mode** by using the toggle. Then enter the following setting under the `portMappings` section of the application definition. This example binds port 80 of the container to port 80 of the DC/OS agent. You can switch this wizard out of JSON Mode after you make this change.

```
"hostPort": 80,
```

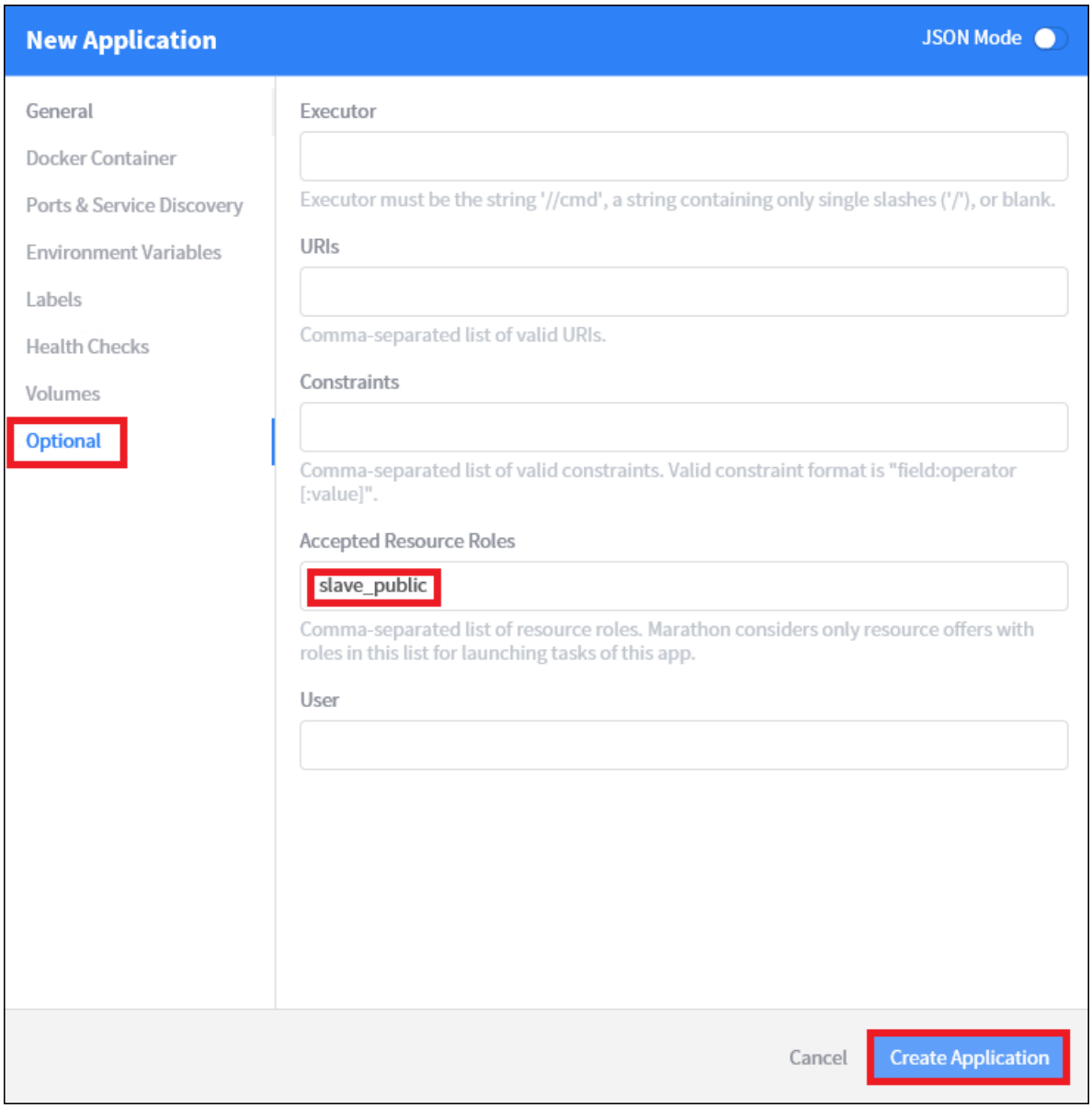


If you want to enable health checks, set a path on the **Health Checks** tab.

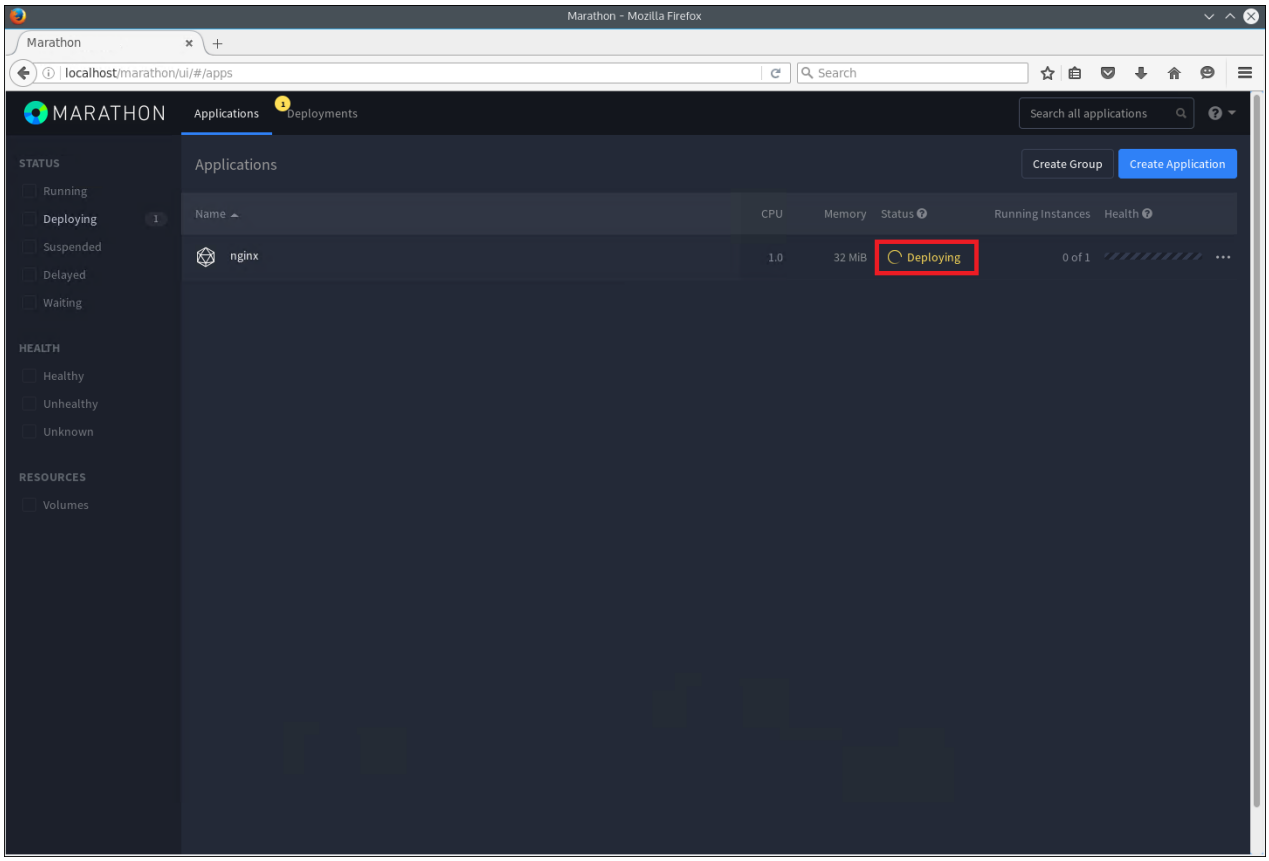


The DC/OS cluster is deployed with set of private and public agents. For the cluster to be able to access applications from the Internet, you need to deploy the applications to a public agent. To do so, select the **Optional** tab of the New Application wizard and enter **slave_public** for the **Accepted Resource Roles**.

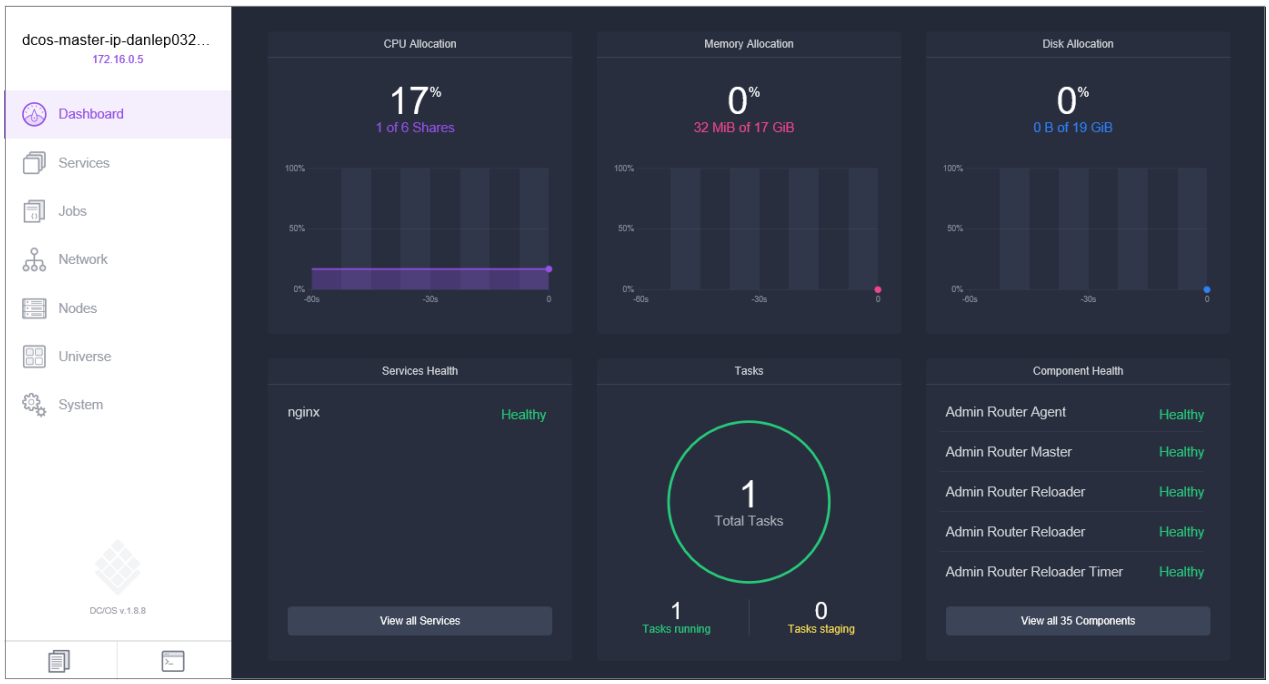
Then click **Create Application**.



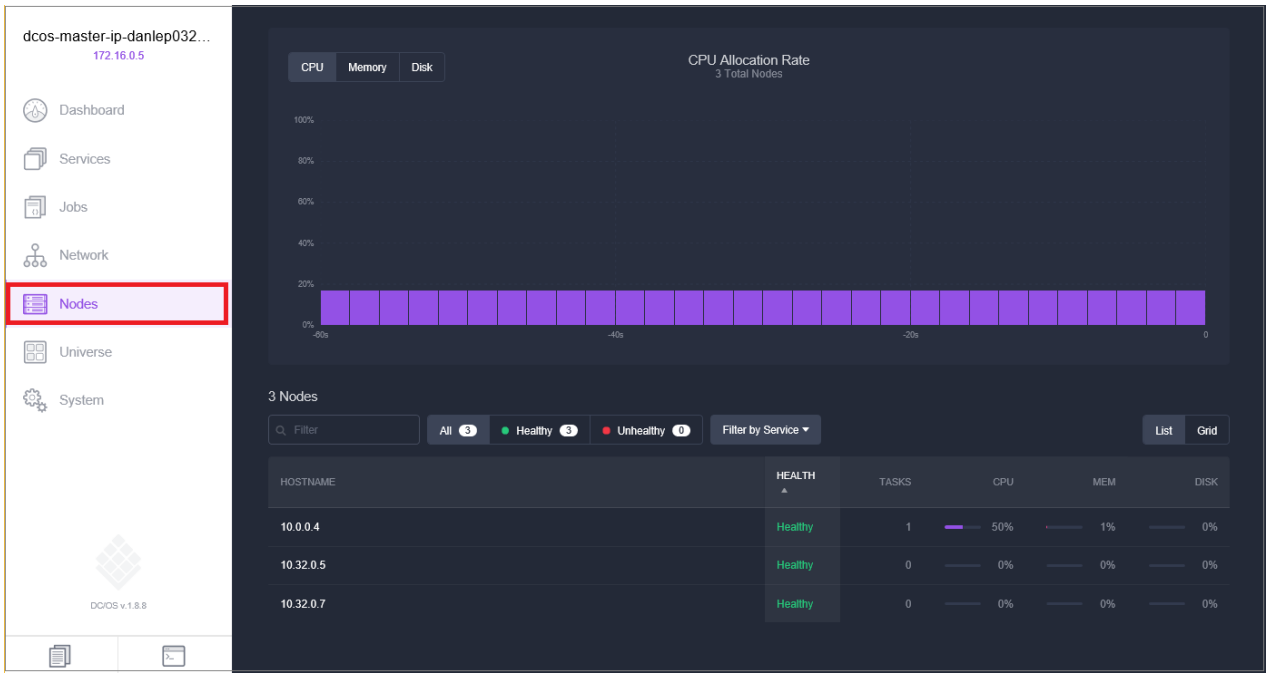
Back on the Marathon main page, you can see the deployment status for the container. Initially you see a status of **Deploying**. After a successful deployment, the status changes to **Running**.



When you switch back to the DC/OS web UI (http://localhost/), you see that a task (in this case, a Docker-formatted container) is running on the DC/OS cluster.



To see the cluster node that the task is running on, click the **Nodes** tab.



Reach the container

In this example, the application is running on a public agent node. You reach the application from the internet by browsing to the agent FQDN of the cluster: `http://[DNSPREFIX]agents.[REGION].cloudapp.azure.com`, where:

- **DNSPREFIX** is the DNS prefix that you provided when you deployed the cluster.
- **REGION** is the region in which your resource group is located.

Next steps

- Work with DC/OS and the Marathon API
- Deep dive on the Azure Container Service with Mesos

DC/OS container management through the Marathon REST API

6/27/2017 • 4 min to read • Edit Online

DC/OS provides an environment for deploying and scaling clustered workloads, while abstracting the underlying hardware. On top of DC/OS, there is a framework that manages scheduling and executing compute workloads. Although frameworks are available for many popular workloads, this document gets you started creating and scaling container deployments by using the Marathon REST API.

Prerequisites

Before working through these examples, you need a DC/OS cluster that is configured in Azure Container Service. You also need to have remote connectivity to this cluster. For more information on these items, see the following articles:

- Deploying an Azure Container Service cluster
- Connecting to an Azure Container Service cluster

Access the DC/OS APIs

After you are connected to the Azure Container Service cluster, you can access the DC/OS and related REST APIs through http://localhost:local-port. The examples in this document assume that you are tunneling on port 80. For example, the Marathon endpoints can be reached at URIs beginning with `http://localhost/marathon/v2/`.

For more information on the various APIs, see the Mesosphere documentation for the Marathon API and the Chronos API, and the Apache documentation for the Mesos Scheduler API.

Gather information from DC/OS and Marathon

Before you deploy containers to the DC/OS cluster, gather some information about the DC/OS cluster, such as the names and status of the DC/OS agents. To do so, query the `master/slaves` endpoint of the DC/OS REST API. If everything goes well, the query returns a list of DC/OS agents and several properties for each.

```
curl http://localhost/mesos/master/slaves
```

Now, use the Marathon `/apps` endpoint to check for current application deployments to the DC/OS cluster. If this is a new cluster, you see an empty array for apps.

```
curl localhost/marathon/v2/apps

{"apps":[]}
```

Deploy a Docker-formatted container

You deploy Docker-formatted containers through the Marathon REST API by using a JSON file that describes the intended deployment. The following sample deploys an Nginx container to a private agent in the cluster.

```
{
  "id": "nginx",
  "cpus": 0.1,
  "mem": 32.0,
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "servicePort": 9000, "protocol": "tcp" }
      ]
    }
  }
}
```

To deploy a Docker-formatted container, store the JSON file in an accessible location. Next, to deploy the container, run the following command. Specify the name of the JSON file (`marathon.json` in this example).

```
curl -X POST http://localhost/marathon/v2/apps -d @marathon.json -H "Content-type: application/json"
```

The output is similar to the following:

```
{"version":"2015-11-20T18:59:00.494Z","deploymentId":"b12f8a73-f56a-4eb1-9375-4ac026d6cdec"}
```

Now, if you query Marathon for applications, this new application appears in the output.

```
curl localhost/marathon/v2/apps
```

Reach the container

You can verify that the Nginx is running in a container on one of the private agents in the cluster. To find the host and port where the container is running, query Marathon for the running tasks:

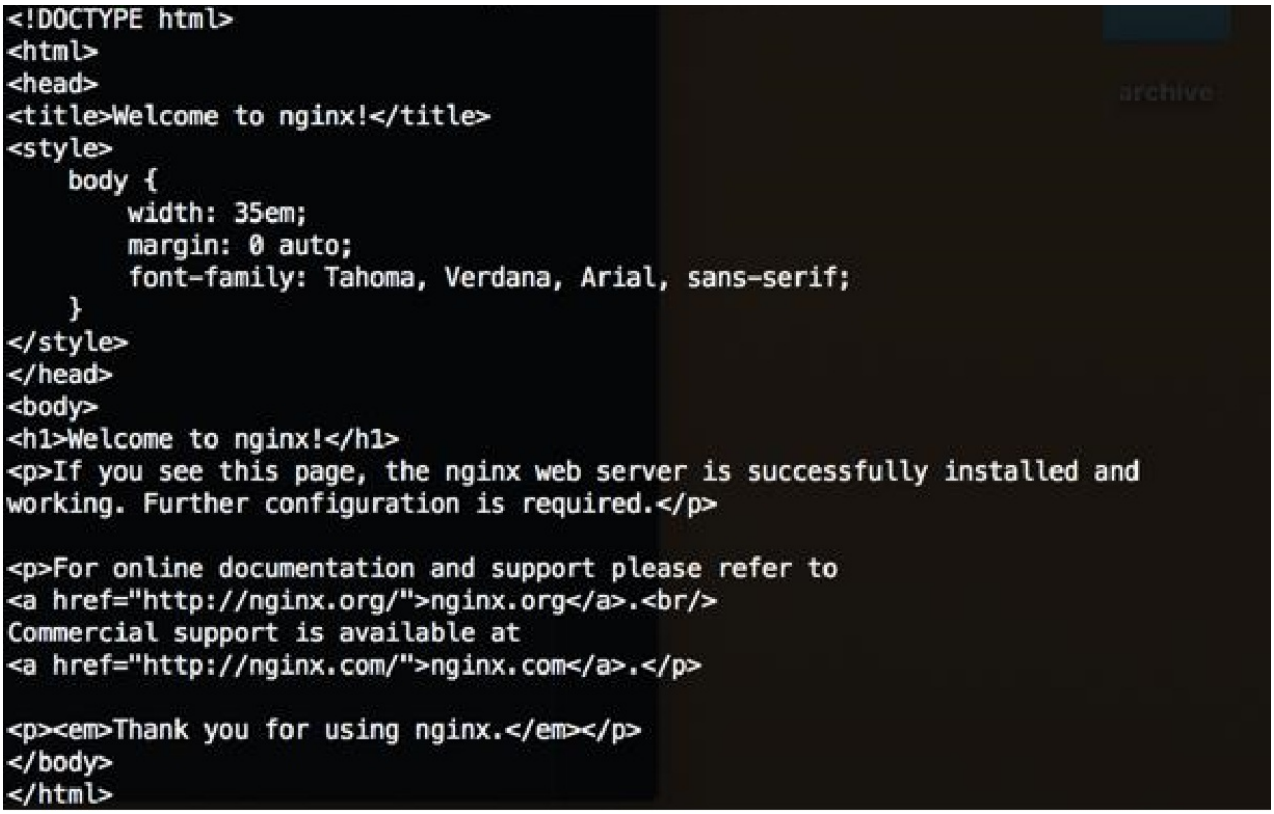
```
curl localhost/marathon/v2/tasks
```

Find the value of `host` in the output (an IP address similar to `10.32.0.x`), and the value of `ports`.

Now make an SSH terminal connection (not a tunneled connection) to the management FQDN of the cluster. Once connected, make the following request, substituting the correct values of `host` and `ports`:

```
curl http://host:ports
```

The Nginx server output is similar to the following:



Scale your containers

You can use the Marathon API to scale out or scale in application deployments. In the previous example, you deployed one instance of an application. Let's scale this out to three instances of an application. To do so, create a JSON file by using the following JSON text, and store it in an accessible location.

```
{ "instances": 3 }
```

From your tunneled connection, run the following command to scale out the application.

> **NOTE**
> The URI is http://localhost/marathon/v2/apps/ followed by the ID of the application to scale. If you are using the Nginx sample that is provided here, the URI would be http://localhost/marathon/v2/apps/nginx.

```
curl http://localhost/marathon/v2/apps/nginx -H "Content-type: application/json" -X PUT -d @scale.json
```

Finally, query the Marathon endpoint for applications. You see that there are now three Nginx containers.

```
curl localhost/marathon/v2/apps
```

Equivalent PowerShell commands

You can perform these same actions by using PowerShell commands on a Windows system.

To gather information about the DC/OS cluster, such as agent names and agent status, run the following command:

```
Invoke-WebRequest -Uri http://localhost/mesos/master/slaves
```

You deploy Docker-formatted containers through Marathon by using a JSON file that describes the intended deployment. The following sample deploys the Nginx container, binding port 80 of the DC/OS agent to port 80 of the container.

```
{
  "id": "nginx",
  "cpus": 0.1,
  "mem": 32.0,
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "servicePort": 9000, "protocol": "tcp" }
      ]
    }
  }
}
```

To deploy a Docker-formatted container, store the JSON file in an accessible location. Next, to deploy the container, run the following command. Specify the path to the JSON file (`marathon.json` in this example).

```
Invoke-WebRequest -Method Post -Uri http://localhost/marathon/v2/apps -ContentType application/json -InFile
'c:\marathon.json'
```

You can also use the Marathon API to scale out or scale in application deployments. In the previous example, you deployed one instance of an application. Let's scale this out to three instances of an application. To do so, create a JSON file by using the following JSON text, and store it in an accessible location.

```
{ "instances": 3 }
```

Run the following command to scale out the application:

> **NOTE**
> The URI is http://localhost/marathon/v2/apps/ followed by the ID of the application to scale. If you are using the Nginx sample provided here, the URI would be http://localhost/marathon/v2/apps/nginx.

```
Invoke-WebRequest -Method Put -Uri http://localhost/marathon/v2/apps/nginx -ContentType application/json -
InFile 'c:\scale.json'
```

Next steps

- Read more about the Mesos HTTP endpoints
- Read more about the Marathon REST API

DC/OS agent pools for Azure Container Service

6/27/2017 • 1 min to read • Edit Online

DC/OS clusters in Azure Container Service contain agent nodes in two pools, a public pool and a private pool. An application can be deployed to either pool, affecting accessibility between machines in your container service. The machines can be exposed to the internet (public) or kept internal (private). This article gives a brief overview of why there are public and private pools.

- **Private agents**: Private agent nodes run through a non-routable network. This network is only accessible from the admin zone or through the public zone edge router. By default, DC/OS launches apps on private agent nodes.
- **Public agents**: Public agent nodes run DC/OS apps and services through a publicly accessible network.

For more information about DC/OS network security, see the DC/OS documentation.

Deploy agent pools

The DC/OS agent pools In Azure Container Service are created as follows:

- The **private pool** contains the number of agent nodes that you specify when you deploy the DC/OS cluster.
- The **public pool** initially contains a predetermined number of agent nodes. This pool is added automatically when the DC/OS cluster is provisioned.

The private pool and the public pool are Azure virtual machine scale sets. You can resize these pools after deployment.

Use agent pools

By default, **Marathon** deploys any new application to the *private* agent nodes. You have to explicitly deploy the application to the *public* nodes during the creation of the application. Select the **Optional** tab and enter **slave_public** for the **Accepted Resource Roles** value. This process is documented here and in the DC/OS documentation.

Next steps

- Read more about managing your DC/OS containers.
- Learn how to open the firewall provided by Azure to allow public access to your DC/OS containers.

Enable public access to an Azure Container Service application

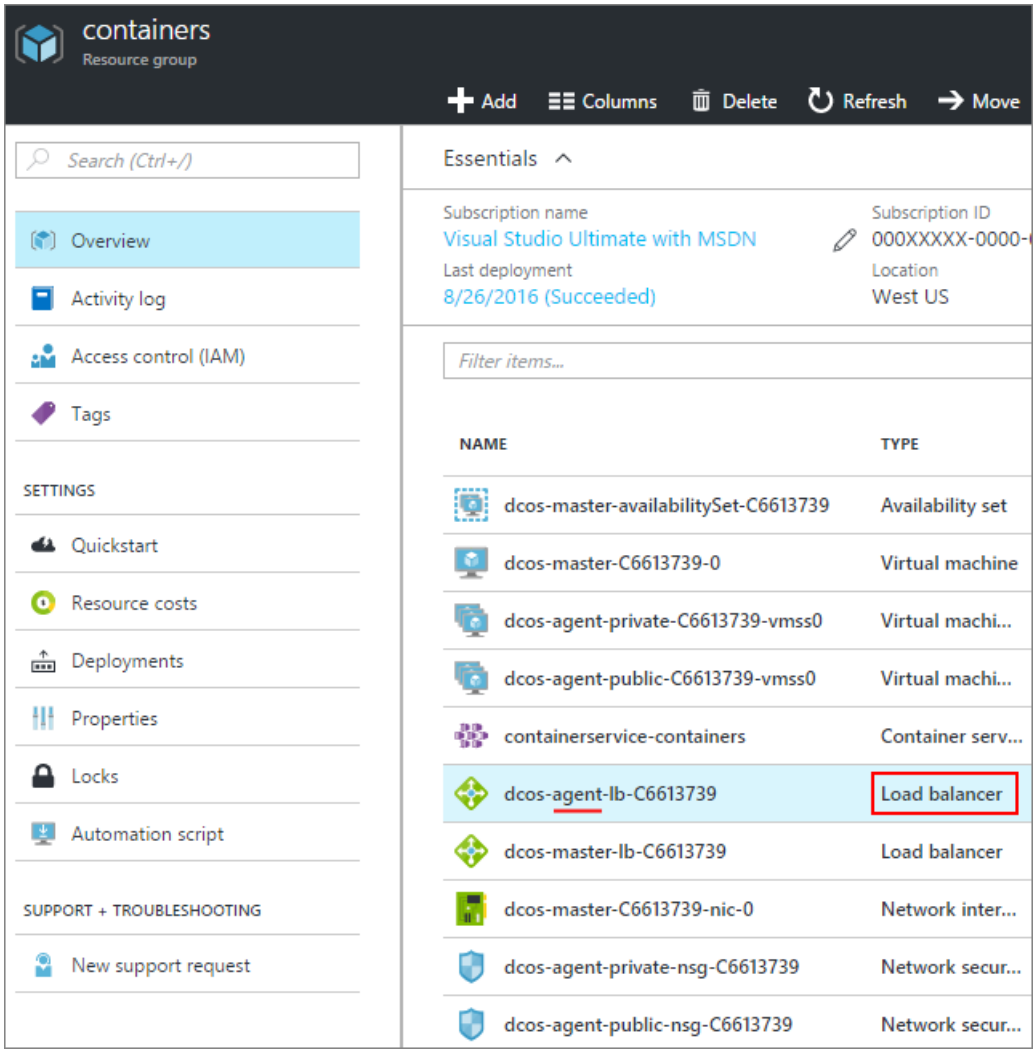
6/27/2017 • 2 min to read • Edit Online

Any DC/OS container in the ACS public agent pool is automatically exposed to the internet. By default, ports **80**, **443**, **8080** are opened, and any (public) container listening on those ports are accessible. This article shows you how to open more ports for your applications in Azure Container Service.

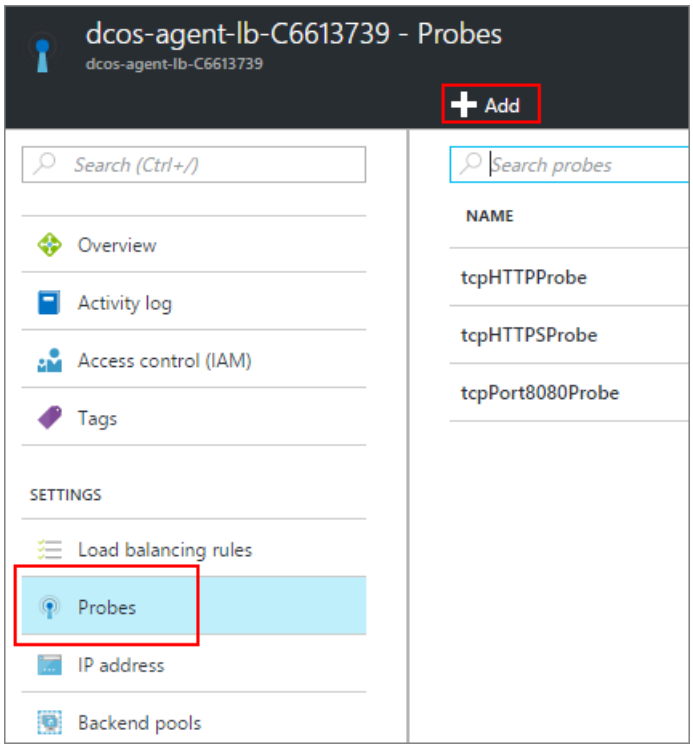
Open a port (portal)

First, we need to open the port we want.

1. Log in to the portal.
2. Find the resource group that you deployed the Azure Container Service to.
3. Select the agent load balancer (which is named similar to **XXXX-agent-lb-XXXX**).



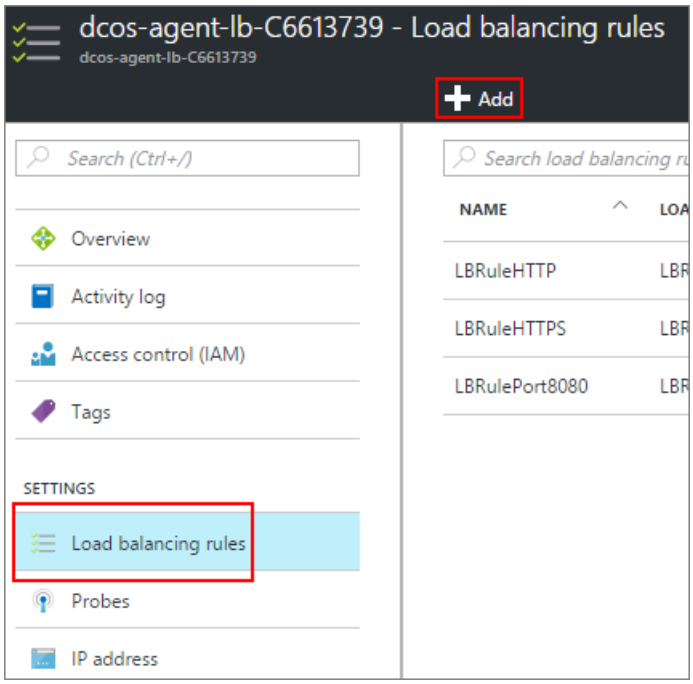
4. Click **Probes** and then **Add**.



5. Fill out the probe form and click **OK**.

FIELD	DESCRIPTION
Name	A descriptive name of the probe.
Port	The port of the container to test.
Path	(When in HTTP mode) The relative website path to probe. HTTPS not supported.
Interval	The amount of time between probe attempts, in seconds.
Unhealthy threshold	Number of consecutive probe attempts before considering the container unhealthy.

6. Back at the properties of the agent load balancer, click **Load balancing rules** and then **Add**.



7. Fill out the load balancer form and click **OK**.

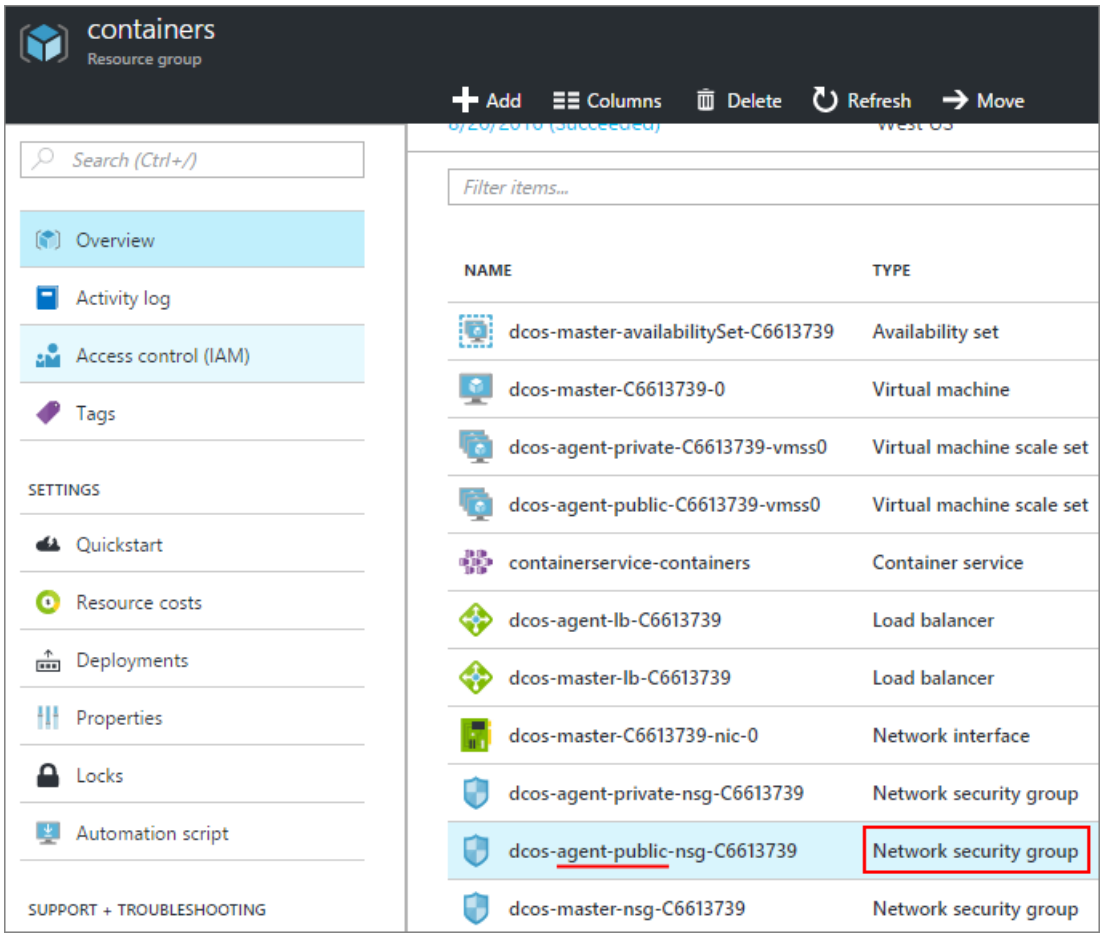
FIELD	DESCRIPTION
Name	A descriptive name of the load balancer.
Port	The public incoming port.
Backend port	The internal-public port of the container to route traffic to.
Backend pool	The containers in this pool will be the target for this load balancer.
Probe	The probe used to determine if a target in the **Backend pool** is healthy.
Session persistence	Determines how traffic from a client should be handled for the duration of the session. **None**: Successive requests from the same client can be handled by any container. **Client IP**: Successive requests from the same client IP are handled by the same container. **Client IP and protocol**: Successive requests from the same client IP and protocol combination are handled by the same container.
Idle timeout	(TCP only) In minutes, the time to keep a TCP/HTTP client open without relying on *keep-alive* messages.

Add a security rule (portal)

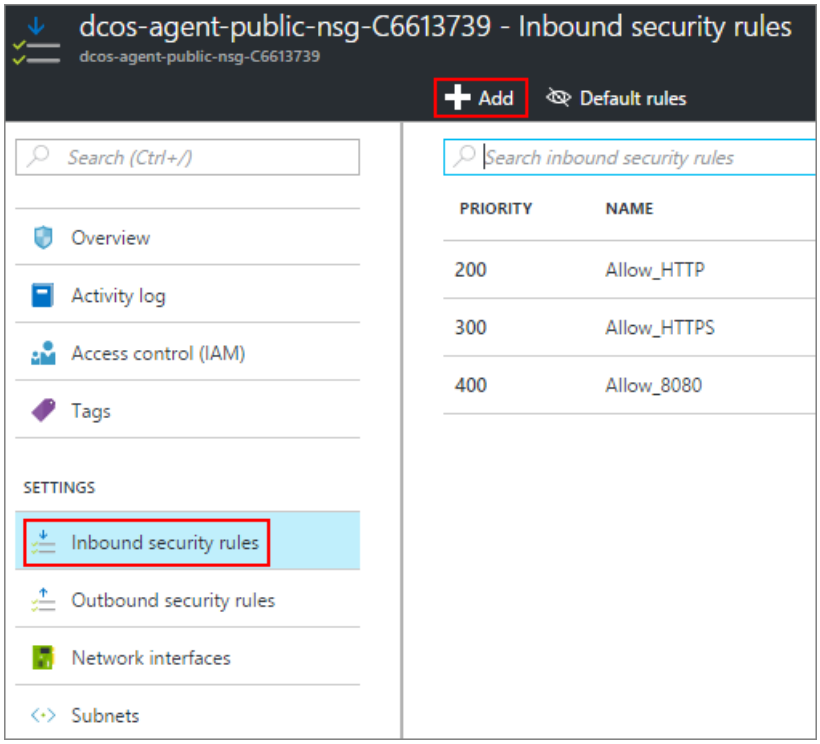
Next, we need to add a security rule that routes traffic from our opened port through the firewall.

1. Log in to the portal.
2. Find the resource group that you deployed the Azure Container Service to.
3. Select the **public** agent network security group (which is named similar to **XXXX-agent-public-nsg-**

XXXX).



4. Select **Inbound security rules** and then **Add**.



5. Fill out the firewall rule to allow your public port and click **OK**.

FIELD	DESCRIPTION
Name	A descriptive name of the firewall rule.

FIELD	DESCRIPTION
Priority	Priority rank for the rule. The lower the number the higher the priority.
Source	Restrict the incoming IP address range to be allowed or denied by this rule. Use **Any** to not specify a restriction.
Service	Select a set of predefined services this security rule is for. Otherwise use **Custom** to create your own.
Protocol	Restrict traffic based on **TCP** or **UDP**. Use **Any** to not specify a restriction.
Port range	When **Service** is **Custom**, specifies the range of ports that this rule affects. You can use a single port, such as **80**, or a range like **1024-1500**.
Action	Allow or deny traffic that meets the criteria.

Next steps

Learn about the difference between public and private DC/OS agents.

Read more information about managing your DC/OS containers.

Create an application or user-specific Marathon service

6/27/2017 • 2 min to read • Edit Online

Azure Container Service provides a set of master servers on which we preconfigure Apache Mesos and Marathon. These can be used to orchestrate your applications on the cluster, but it's best not to use the master servers for this purpose. For example, tweaking the configuration of Marathon requires logging into the master servers themselves and making changes--this encourages unique master servers that are a little different from the standard and need to be cared for and managed independently. Additionally, the configuration required by one team might not be the optimal configuration for another team.

In this article, we'll explain how to add an application or user-specific Marathon service.

Because this service will belong to a single user or team, they are free to configure it in any way that they desire. Also, Azure Container Service will ensure that the service continues to run. If the service fails, Azure Container Service will restart it for you. Most of the time you won't even notice it had downtime.

Prerequisites

Deploy an instance of Azure Container Service with orchestrator type DC/OS and ensure that your client can connect to your cluster. Also, do the following steps.

> **NOTE**
> This is for working with DC/OS-based ACS clusters. There is no need to do this for Swarm-based ACS clusters.

First, connect to your DC/OS-based ACS cluster. Once you have done this, you can install the DC/OS CLI on your client machine with the commands below:

```
sudo pip install virtualenv
mkdir dcos && cd dcos
wget https://raw.githubusercontent.com/mesosphere/dcos-cli/master/bin/install/install-optout-dcos-cli.sh
chmod +x install-optout-dcos-cli.sh
./install-optout-dcos-cli.sh . http://localhost --add-path yes
```

If you are using an old version of Python, you may notice some "InsecurePlatformWarnings". You can safely ignore these.

In order to get started without restarting your shell, run:

```
source ~/.bashrc
```

This step will not be necessary when you start new shells.

Now you can confirm that the CLI is installed:

```
dcos --help
```

Create an application or user-specific Marathon service

Begin by creating a JSON configuration file that defines the name of the application service that you want to create. Here we use `marathon-alice` as the framework name. Save the file as something like `marathon-alice.json`:

```
{"marathon": {"framework-name": "marathon-alice" }}
```

Next, use the DC/OS CLI to install the Marathon instance with the options that are set in your configuration file:

```
dcos package install --options=marathon-alice.json marathon
```

You should now see your `marathon-alice` service running in the Services tab of your DC/OS UI. The UI will be `http://<hostname>/service/marathon-alice/` if you want to access it directly.

Set the DC/OS CLI to access the service

You can optionally configure your DC/OS CLI to access this new service by setting the `marathon.url` property to point to the `marathon-alice` instance as follows:

```
dcos config set marathon.url http://<hostname>/service/marathon-alice/
```

You can verify which instance of Marathon that your CLI is working against with the `dcos config show` command. You can revert to using your master Marathon service with the command `dcos config unset marathon.url`.

Canary release microservices with Vamp on an Azure Container Service DC/OS cluster

6/27/2017 • 8 min to read • Edit Online

In this walkthrough, we set up Vamp on Azure Container Service with a DC/OS cluster. We canary release the Vamp demo service "sava", and then resolve an incompatibility of the service with Firefox by applying smart traffic filtering.

> **TIP**
> In this walkthrough Vamp runs on a DC/OS cluster, but you can also use Vamp with Kubernetes as the orchestrator.

About canary releases and Vamp

Canary releasing is a smart deployment strategy adopted by innovative organizations like Netflix, Facebook, and Spotify. It's an approach that makes sense, because it reduces issues, introduces safety-nets, and increases innovation. So why aren't all companies using it? Extending a CI/CD pipeline to include canary strategies adds complexity, and requires extensive devops knowledge and experience. That's enough to block smaller companies and enterprises alike before they even get started.

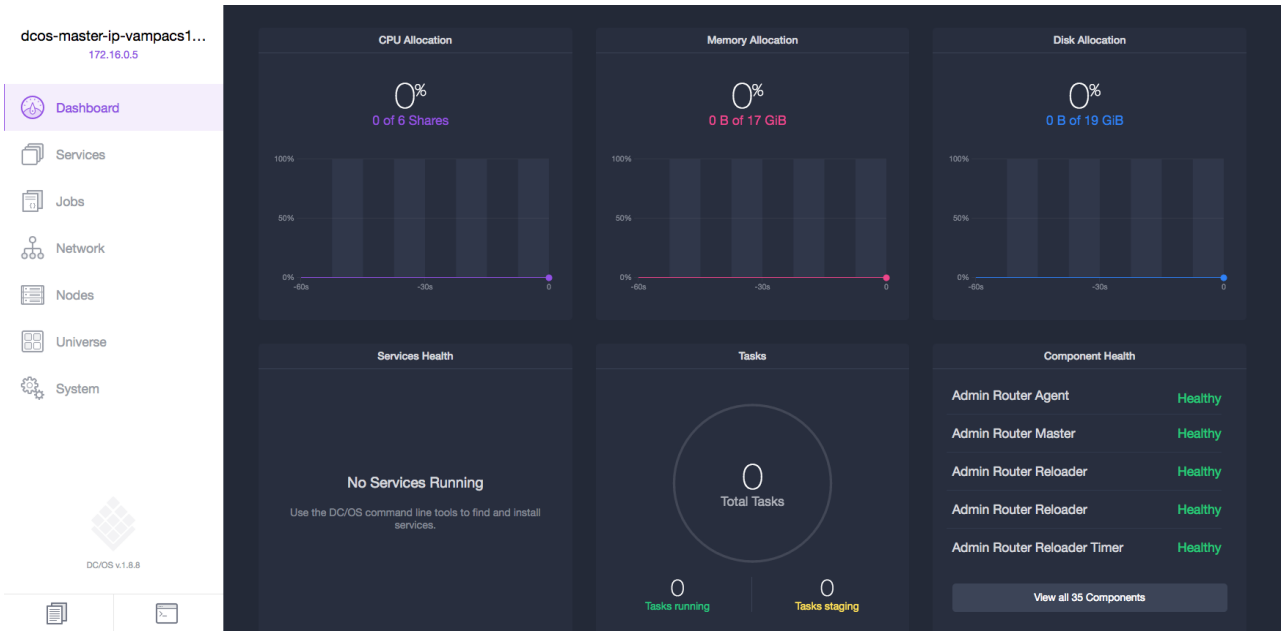
Vamp is an open-source system designed to ease this transition and bring canary releasing features to your preferred container scheduler. Vamp's canary functionality goes beyond percentage-based rollouts. Traffic can be filtered and split on a wide range of conditions, for example to target specific users, IP-ranges, or devices. Vamp tracks and analyzes performance metrics, allowing for automation based on real-world data. You can set up automatic rollback on errors, or scale individual service variants based on load or latency.

Set up Azure Container Service with DC/OS

1. Deploy a DC/OS cluster with one master and two agents of default size.
2. Create an SSH tunnel to connect to the DC/OS cluster. This article assumes that you tunnel to the cluster on local port 80.

Set up Vamp

Now that you have a running DC/OS cluster, you can install Vamp from the DC/OS UI (http://localhost:80).



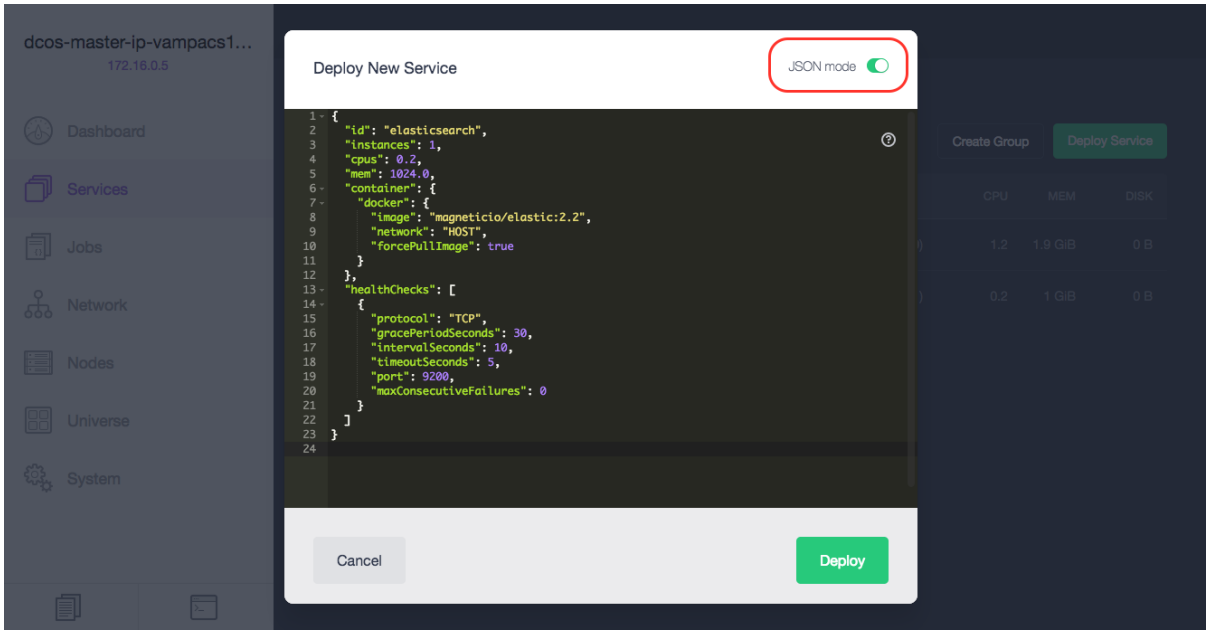
Installation is done in two stages:

1. **Deploy Elasticsearch**.
2. Then **deploy Vamp** by installing the Vamp DC/OS universe package.

Deploy Elasticsearch

Vamp requires Elasticsearch for metrics collection and aggregation. You can use the magneticio Docker images to deploy a compatible Vamp Elasticsearch stack.

1. In the DC/OS UI, go to **Services** and click **Deploy Service**.
2. Select **JSON mode** from the **Deploy New Service** pop-up.

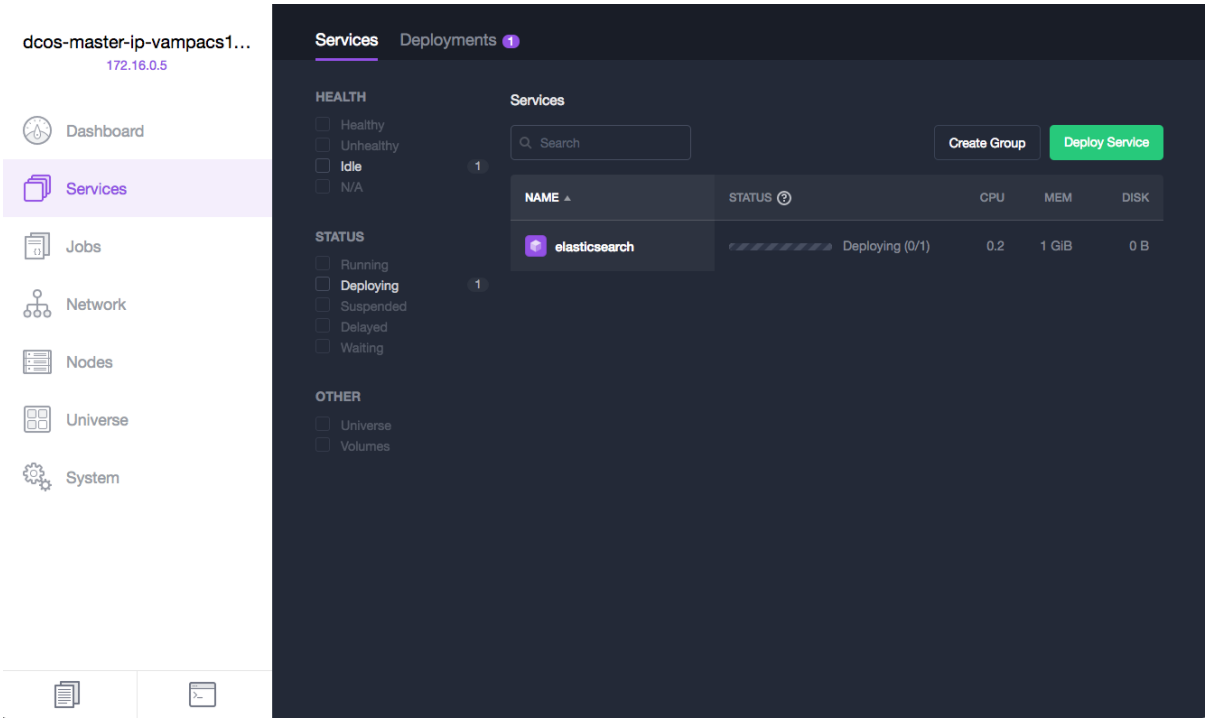


3. Paste in the following JSON. This configuration runs the container with 1 GB of RAM and a basic health check on the Elasticsearch port.

```
{
 "id": "elasticsearch",
 "instances": 1,
 "cpus": 0.2,
 "mem": 1024.0,
 "container": {
   "docker": {
     "image": "magneticio/elastic:2.2",
     "network": "HOST",
     "forcePullImage": true
   }
 },
 "healthChecks": [
   {
     "protocol": "TCP",
     "gracePeriodSeconds": 30,
     "intervalSeconds": 10,
     "timeoutSeconds": 5,
     "port": 9200,
     "maxConsecutiveFailures": 0
   }
 ]
}
```

4. Click **Deploy**.

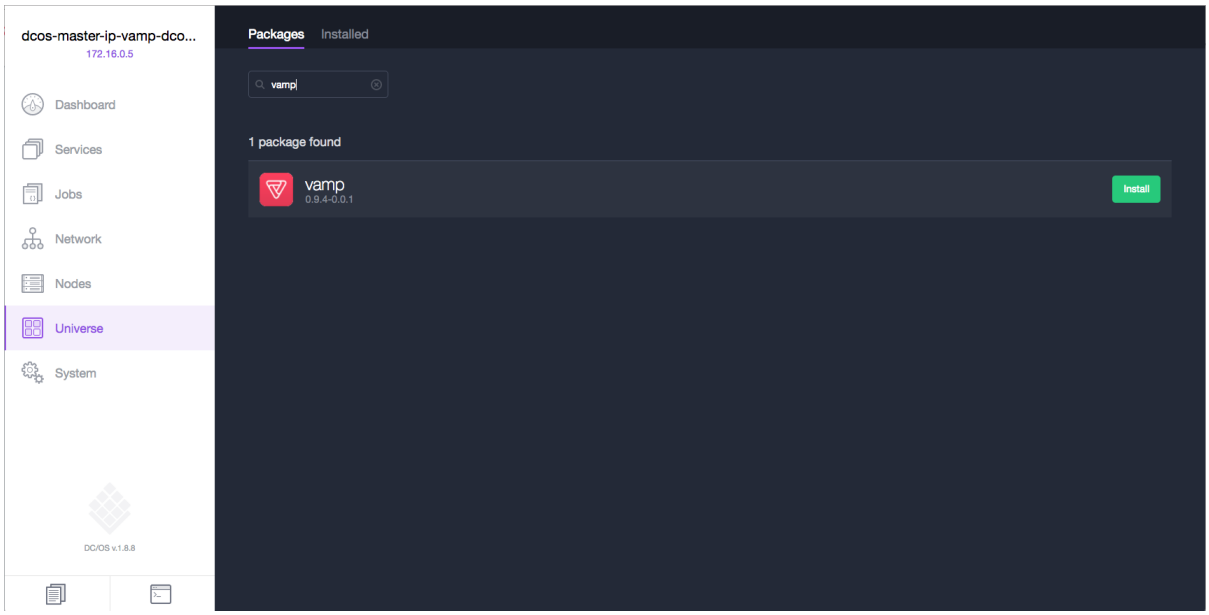
   DC/OS deploys the Elasticsearch container. You can track progress on the **Services** page.



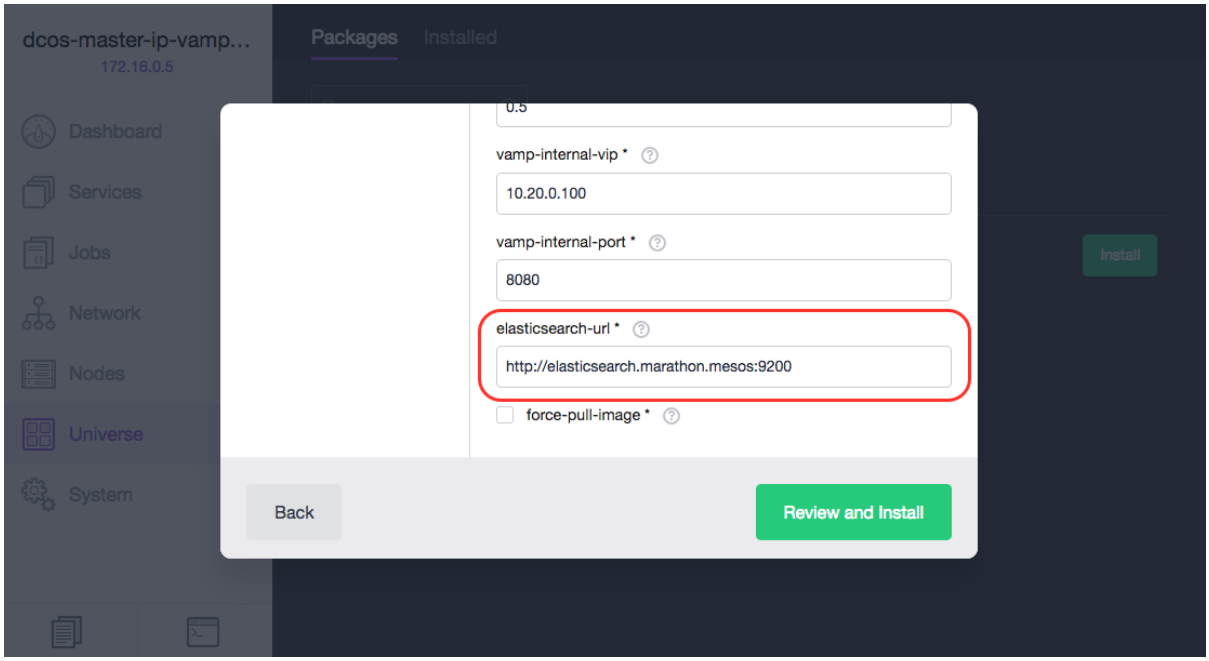
Deploy Vamp

Once Elasticsearch reports as **Running**, you can add the Vamp DC/OS Universe package.

1. Go to **Universe** and search for **vamp**.

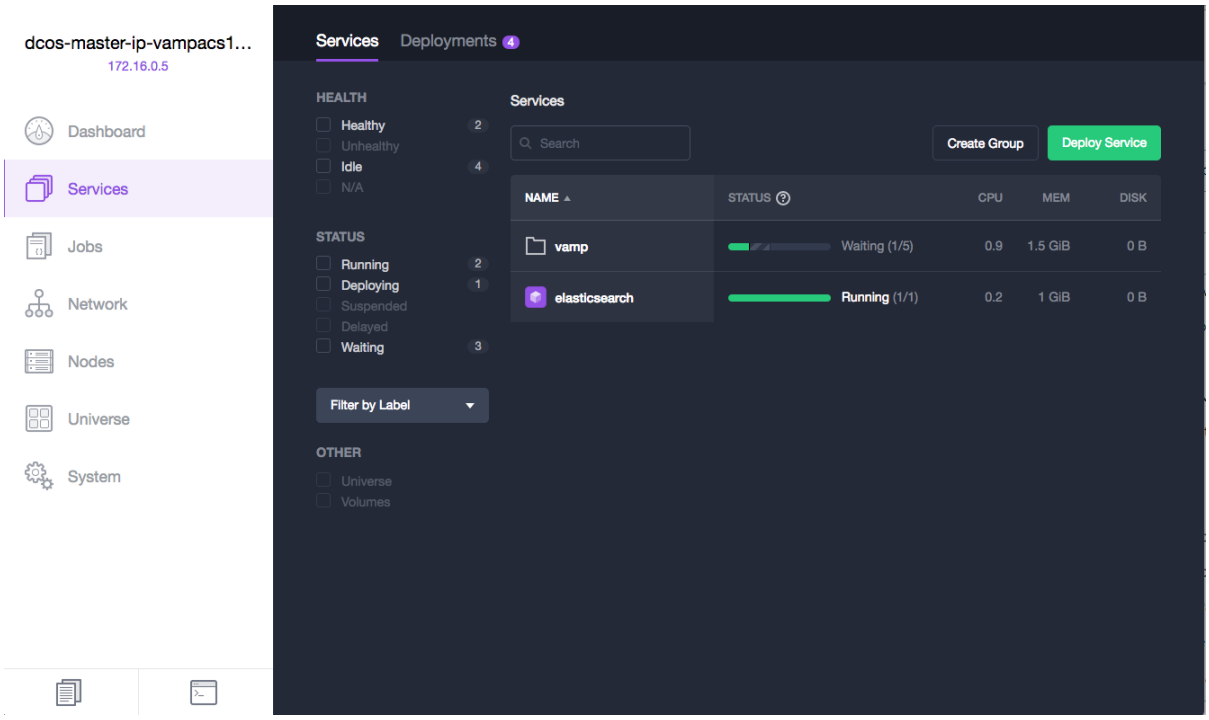


2. Click **install** next to the vamp package, and choose **Advanced Installation**.
3. Scroll down and enter the following elasticsearch-url: `http://elasticsearch.marathon.mesos:9200`.

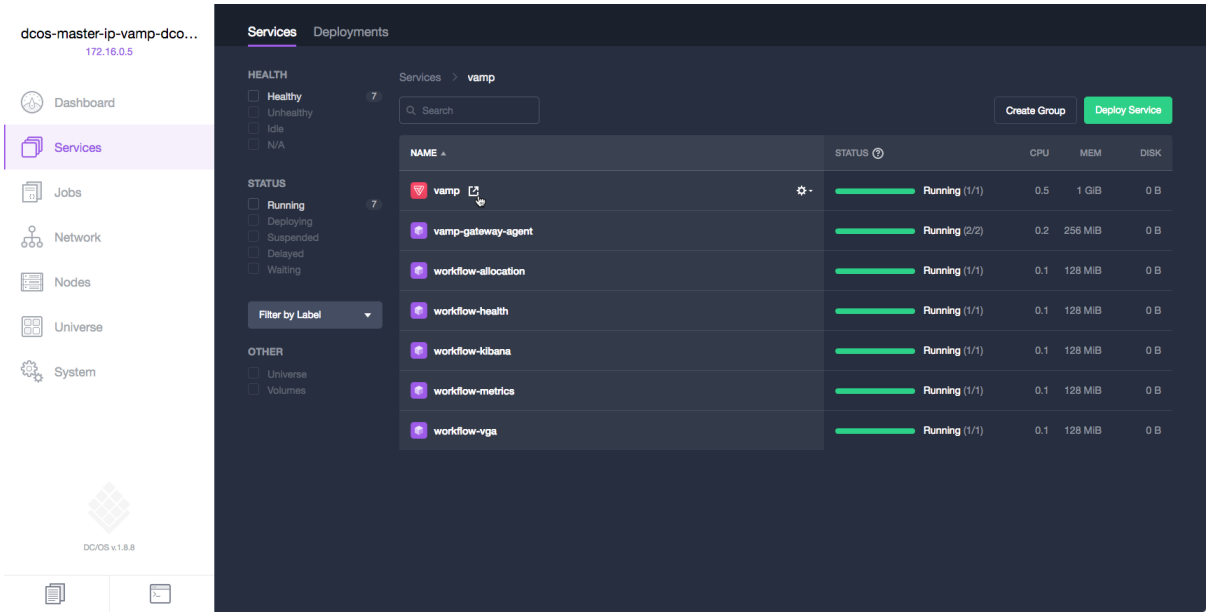


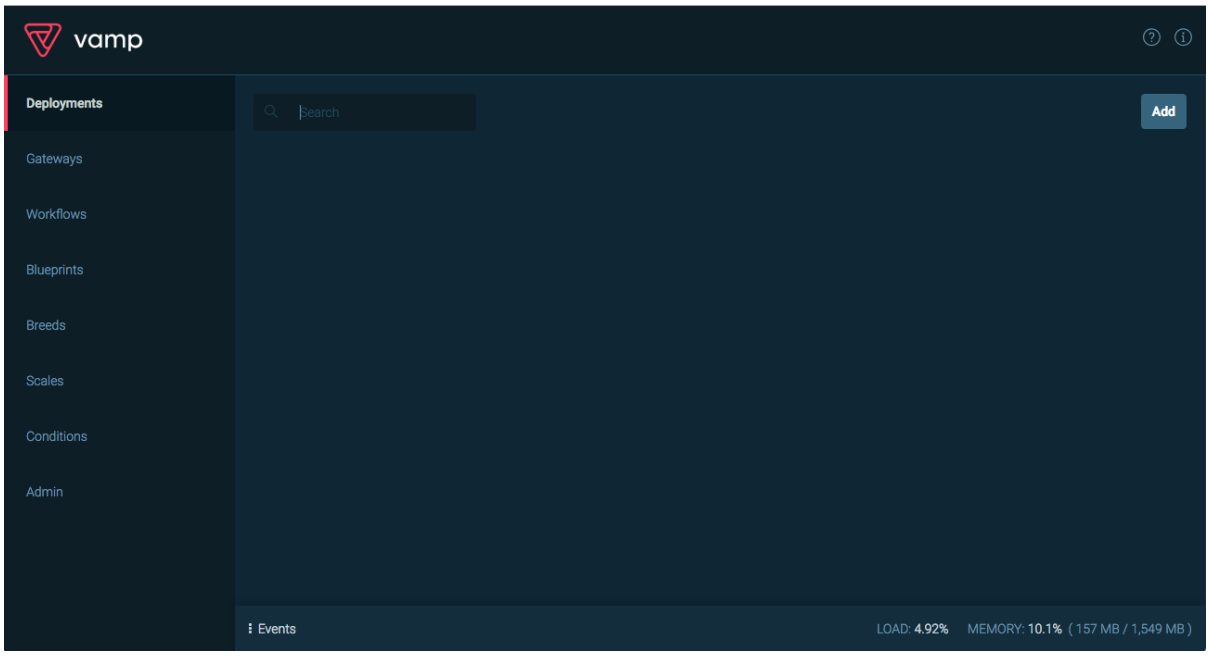
4. Click **Review and Install**, then click **Install** to start the deployment.

   DC/OS deploys all required Vamp components. You can track progress on the **Services** page.



5. Once deployment has completed, you can access the Vamp UI:





Deploy your first service

Now that Vamp is up and running, deploy a service from a blueprint.

In its simplest form, a Vamp blueprint describes the endpoints (gateways), clusters, and services to deploy. Vamp uses clusters to group different variants of the same service into logical groups for canary releasing or A/B testing.

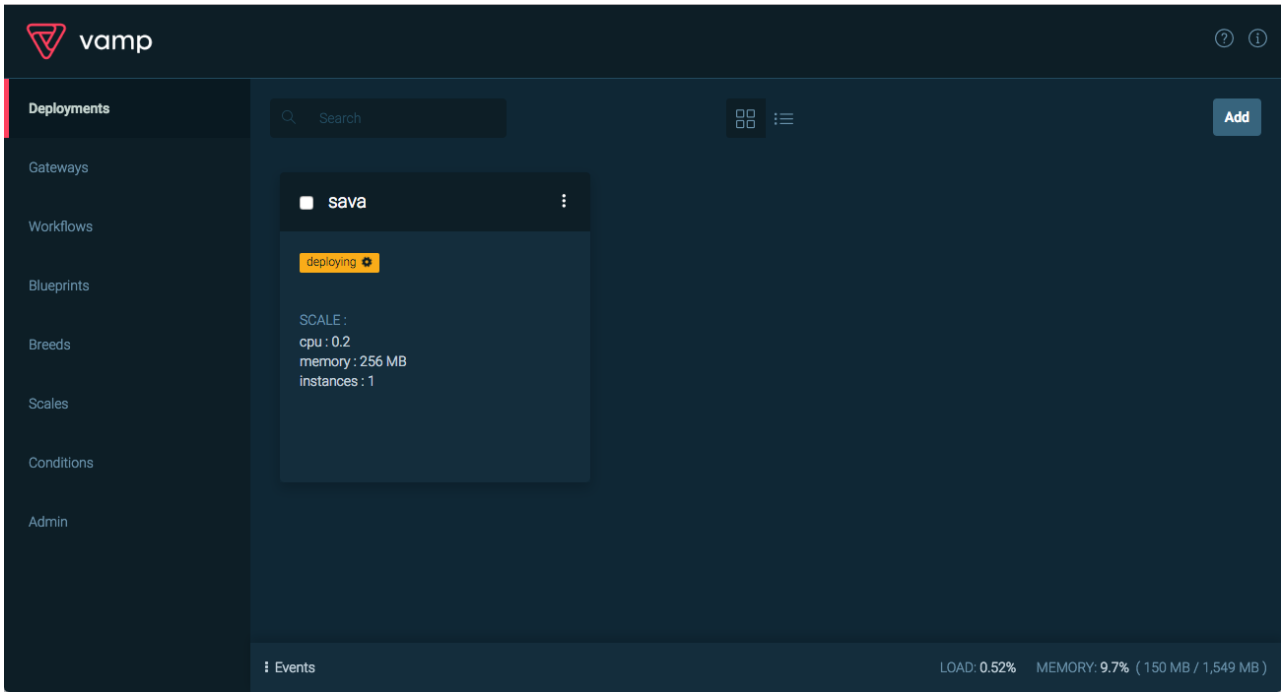
This scenario uses a sample monolithic application called **sava**, which is at version 1.0. The monolith is packaged in a Docker container, which is in Docker Hub under magneticio/sava:1.0.0. The app normally runs on port 8080, but you want to expose it under port 9050 in this case. Deploy the app through Vamp using a simple blueprint.

1. Go to **Deployments**.
2. Click **Add**.
3. Paste in the following blueprint YAML. This blueprint contains one cluster with only one service variant, which we change in a later step:

```
name: sava                       # deployment name
gateways:
 9050: sava_cluster/webport      # stable endpoint
clusters:
 sava_cluster:               # cluster to create
  services:
     -
       breed:
         name: sava:1.0.0        # service variant name
         deployable: magneticio/sava:1.0.0
         ports:
           webport: 8080/http # cluster endpoint, used for canary releasing
```

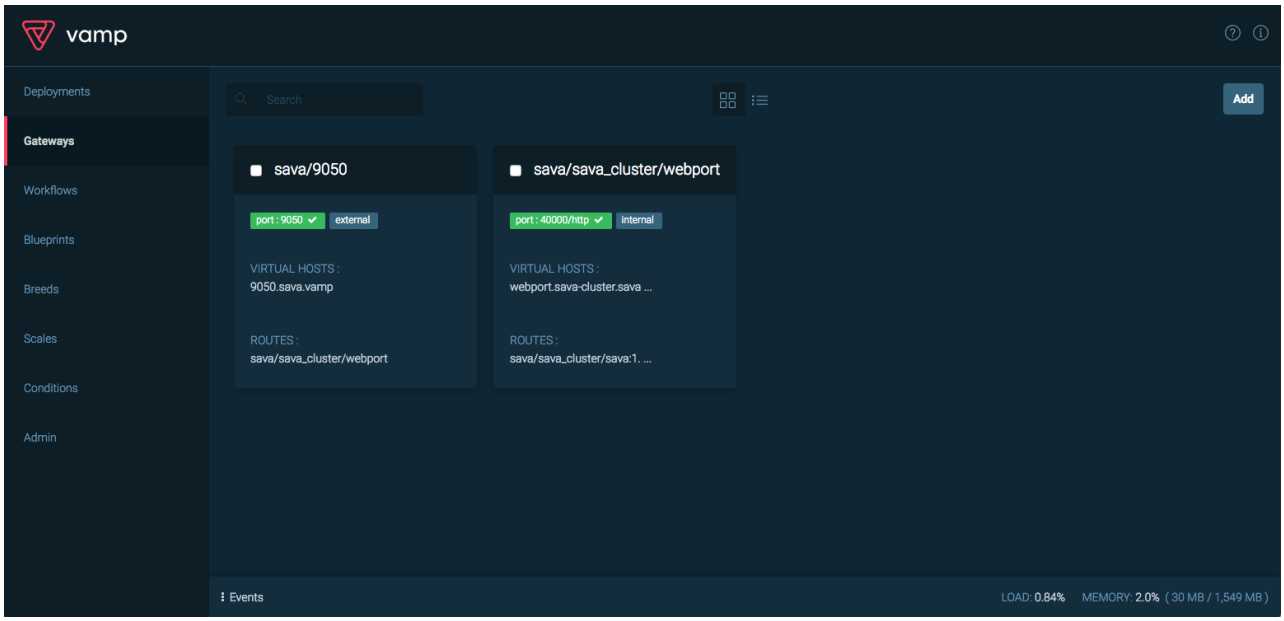
4. Click **Save**. Vamp initiates the deployment.

The deployment is listed on the **Deployments** page. Click the deployment to monitor its status.



Two gateways are created, which are listed on the **Gateways** page:

- a stable endpoint to access the running service (port 9050)
- a Vamp-managed internal gateway (more on this gateway later).



The sava service has now deployed, but you can't access it externally because the Azure Load Balancer doesn't know to forward traffic to it yet. To access the service, update the Azure networking configuration.

Update the Azure network configuration

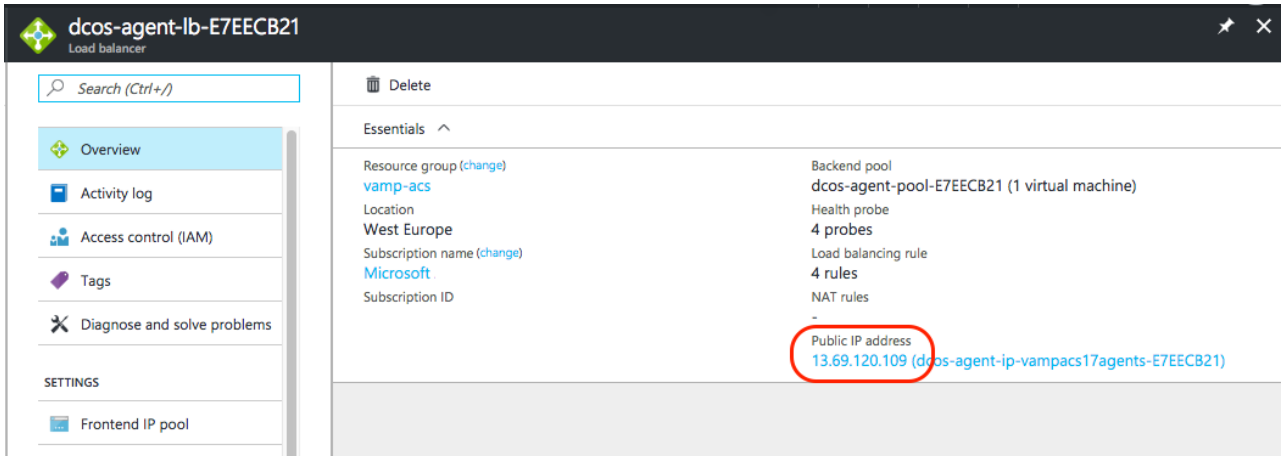
Vamp deployed the sava service on the DC/OS agent nodes, exposing a stable endpoint at port 9050. To access the service from outside the DC/OS cluster, make the following changes to the Azure network configuration in your cluster deployment:

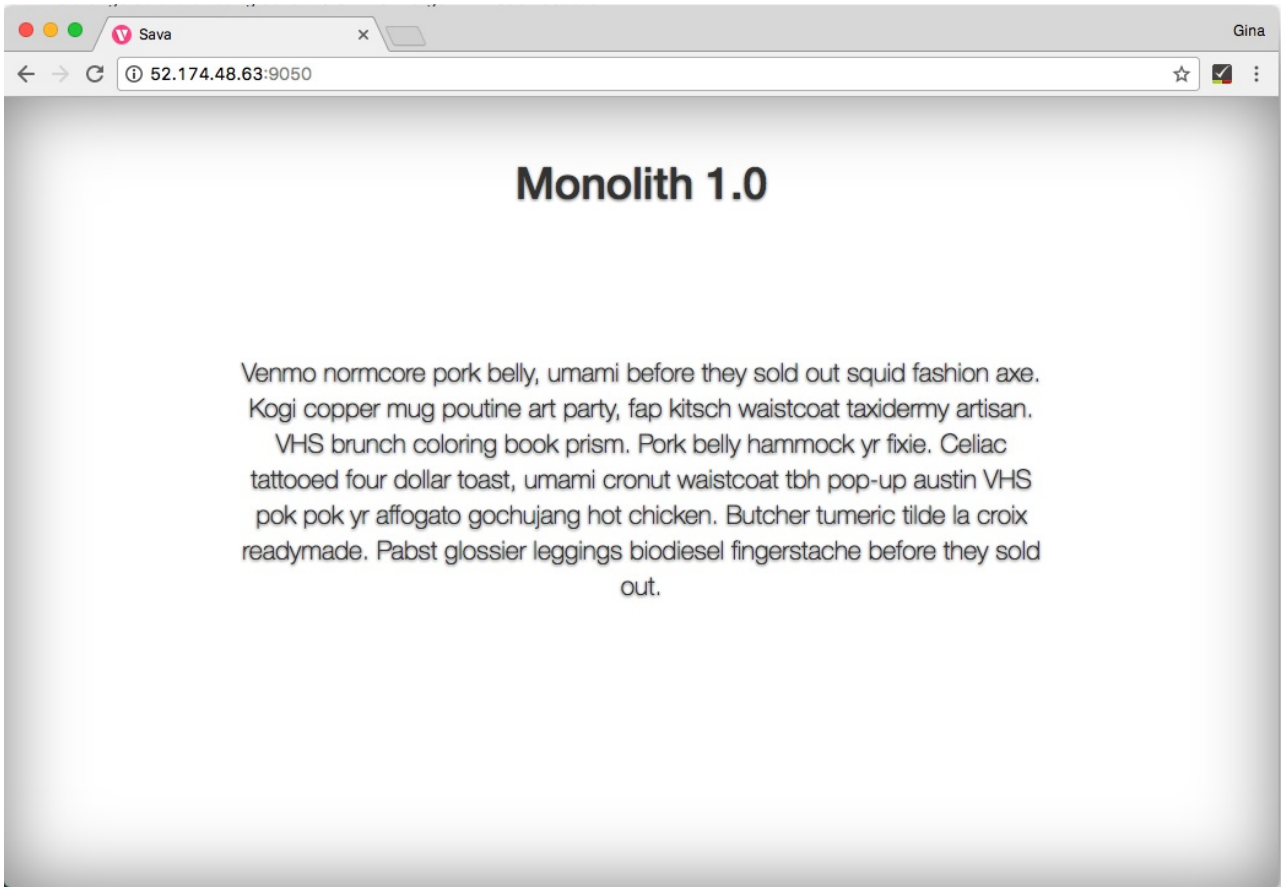
1. **Configure the Azure Load Balancer** for the agents (the resource named **dcos-agent-lb-xxxx**) with a health probe and a rule to forward traffic on port 9050 to the sava instances.
2. **Update the network security group** for the public agents (the resource named **XXXX-agent-public-nsg-XXXX**) to allow traffic on port 9050.

For detailed steps to complete these tasks using the Azure portal, see Enable public access to an Azure Container

Service application. Specify port 9050 for all port settings.

Once everything has been created, go to the **Overview** blade of the DC/OS agent load balancer (the resource named **dcos-agent-lb-xxxx**). Find the **Public IP address**, and use the address to access sava at port 9050.





Run a canary release

Suppose you have a new version of this application that you want to canary release into production. You have it containerized as magneticio/sava:1.1.0 and are ready to go. Vamp lets you easily add new services to the running deployment. These "merged" services are deployed alongside the existing services in the cluster, and assigned a weight of 0%. No traffic is routed to a newly merged service until you adjust the traffic distribution. The weight slider in the Vamp UI gives you complete control over the distribution, allowing for incremental adjustments (canary release) or an instant rollback.

Merge a new service variant

To merge the new sava 1.1 service with the running deployment:

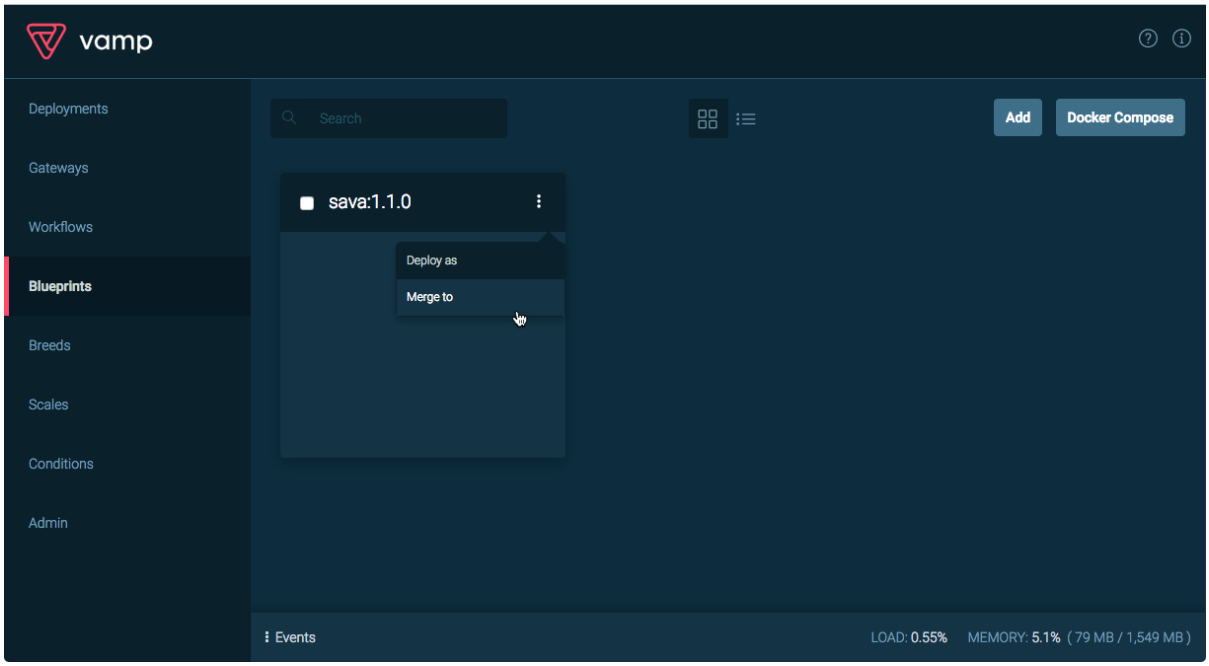
1. In the Vamp UI, click **Blueprints**.

2. Click **Add** and paste in the following blueprint YAML: This blueprint describes a new service variant (sava:1.1.0) to deploy within the existing cluster (sava_cluster).

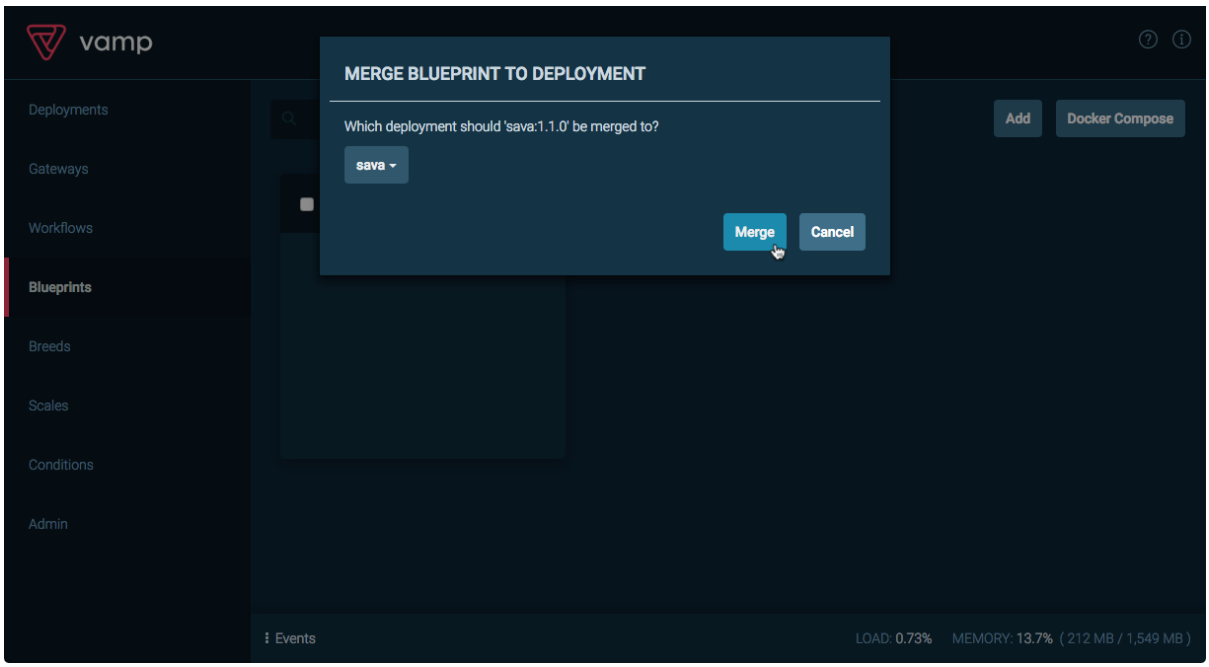
```
name: sava:1.1.0      # blueprint name
clusters:
 sava_cluster:       # cluster to update
   services:
     -
       breed:
         name: sava:1.1.0    # service variant name
         deployable: magneticio/sava:1.1.0
         ports:
           webport: 8080/http # cluster endpoint to update
```

3. Click **Save**. The blueprint is stored and listed on the **Blueprints** page.

4. Open the action menu on the sava:1.1 blueprint and click **Merge to**.

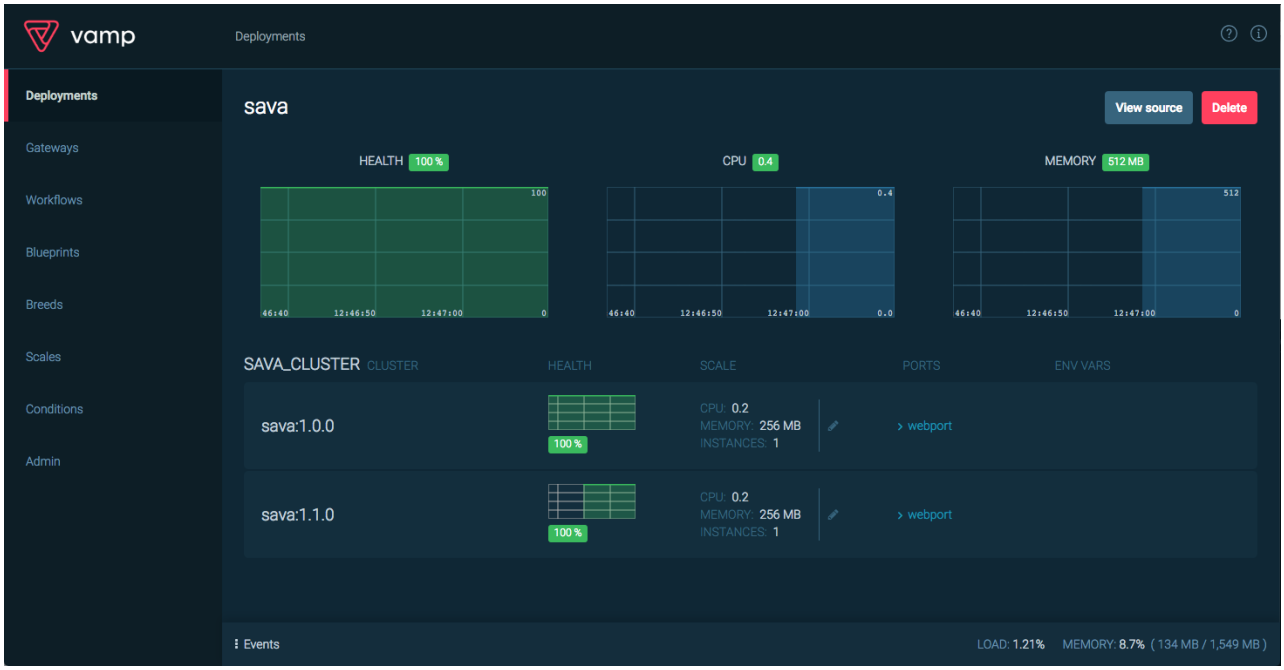


5. Select the **sava** deployment and click **Merge**.

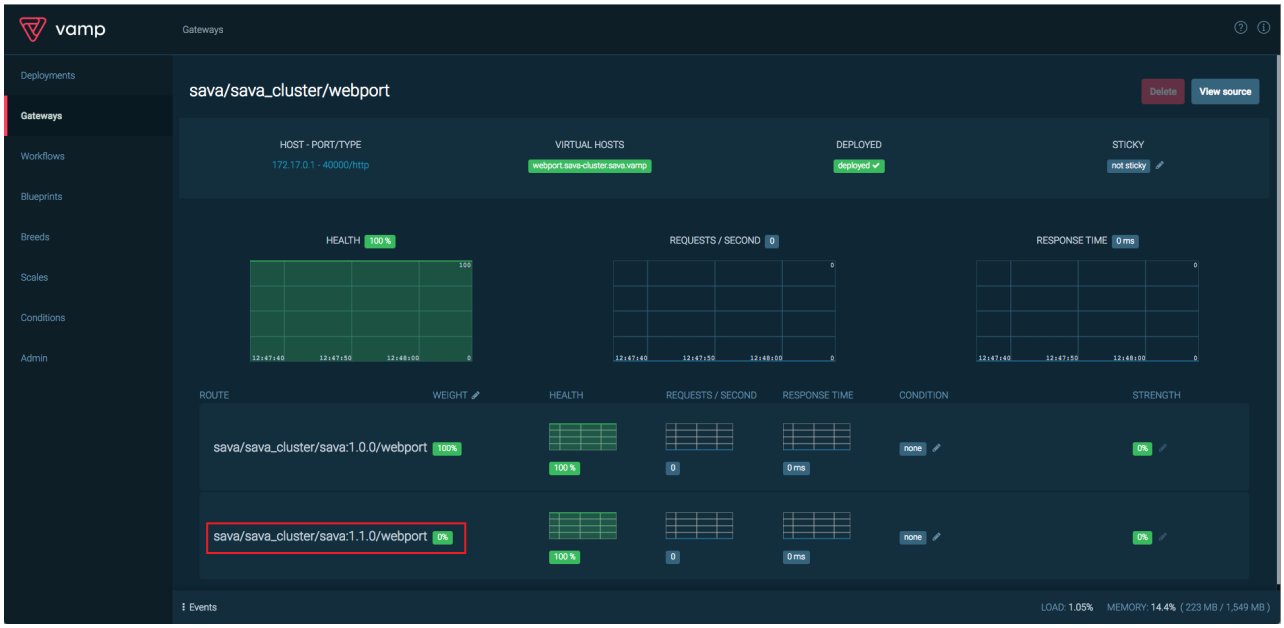


Vamp deploys the new sava:1.1.0 service variant described in the blueprint alongside sava:1.0.0 in the **sava_cluster**

of the running deployment.



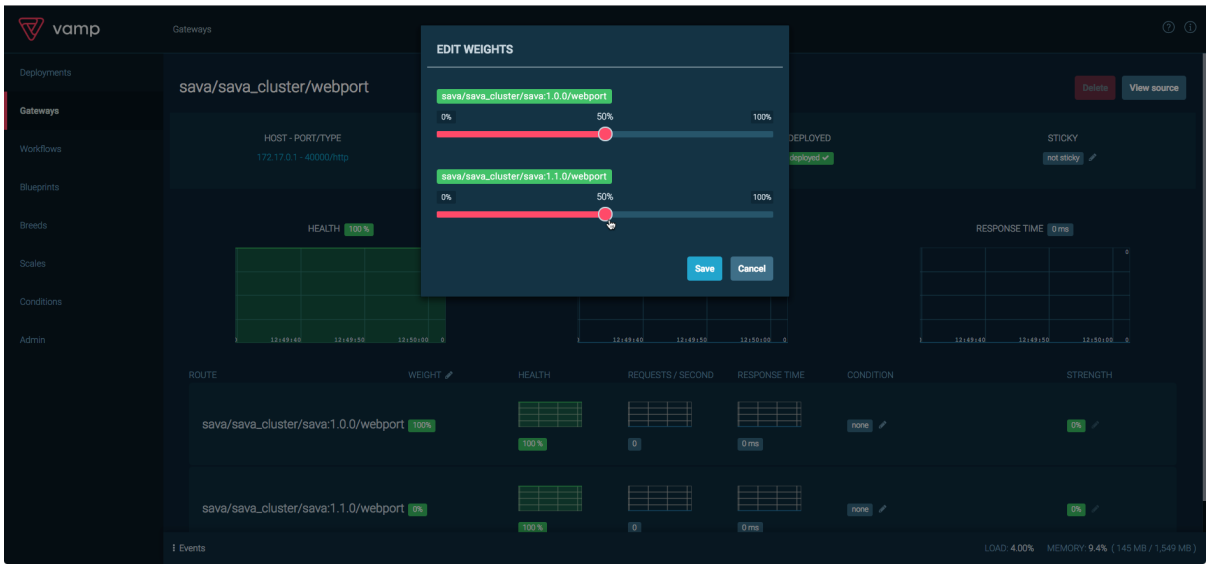
The **sava/sava_cluster/webport** gateway (the cluster endpoint) is also updated, adding a route to the newly deployed sava:1.1.0. At this point, no traffic is routed here (the **WEIGHT** is set to 0%).



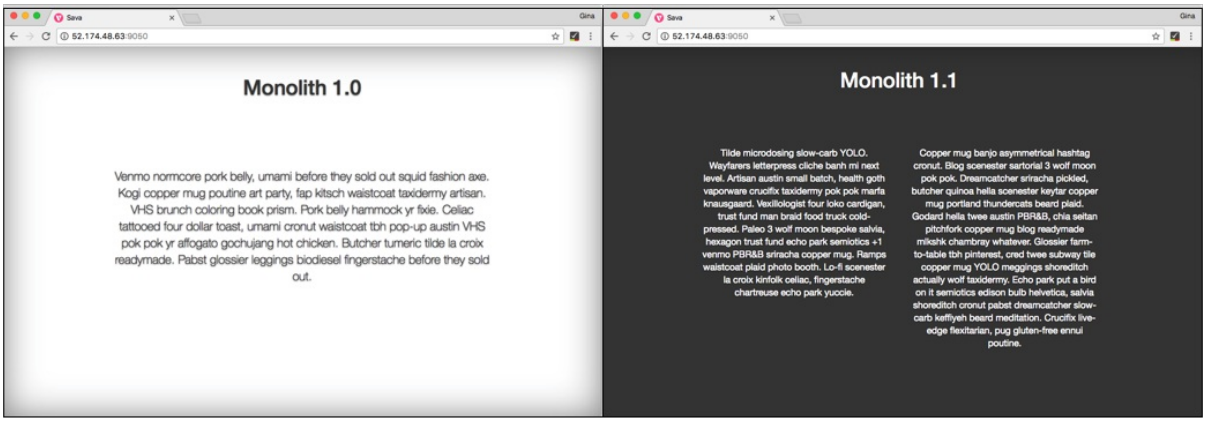
Canary release

With both versions of sava deployed in the same cluster, adjust the distribution of traffic between them by moving the **WEIGHT** slider.

1. Click  next to **WEIGHT**.
2. Set the weight distribution to 50%/50% and click **Save**.



3. Go back to your browser and refresh the sava page a few more times. The sava application now switches between a sava:1.0 page and a sava:1.1 page.



> **NOTE**
>
> This alternation of the page works best with the "Incognito" or “Anonymous” mode of your browser because of the caching of static assets.

Filter traffic

Suppose after deployment that you discovered an incompatibility in sava:1.1.0 that causes display problems in Firefox browsers. You can set Vamp to filter incoming traffic and direct all Firefox users back to the known stable sava:1.0.0. This filter instantly resolves the disruption for Firefox users, while everybody else continues to enjoy the benefits of the improved sava:1.1.0.

Vamp uses **conditions** to filter traffic between routes in a gateway. Traffic is first filtered and directed according to the conditions applied to each route. All remaining traffic is distributed according to the gateway weight setting.

You can create a condition to filter all Firefox users and direct them to the old sava:1.0.0:

1. On the sava/sava_cluster/webport **Gateways** page, click  to add a **CONDITION** to the route sava/sava_cluster/sava:1.0.0/webport.

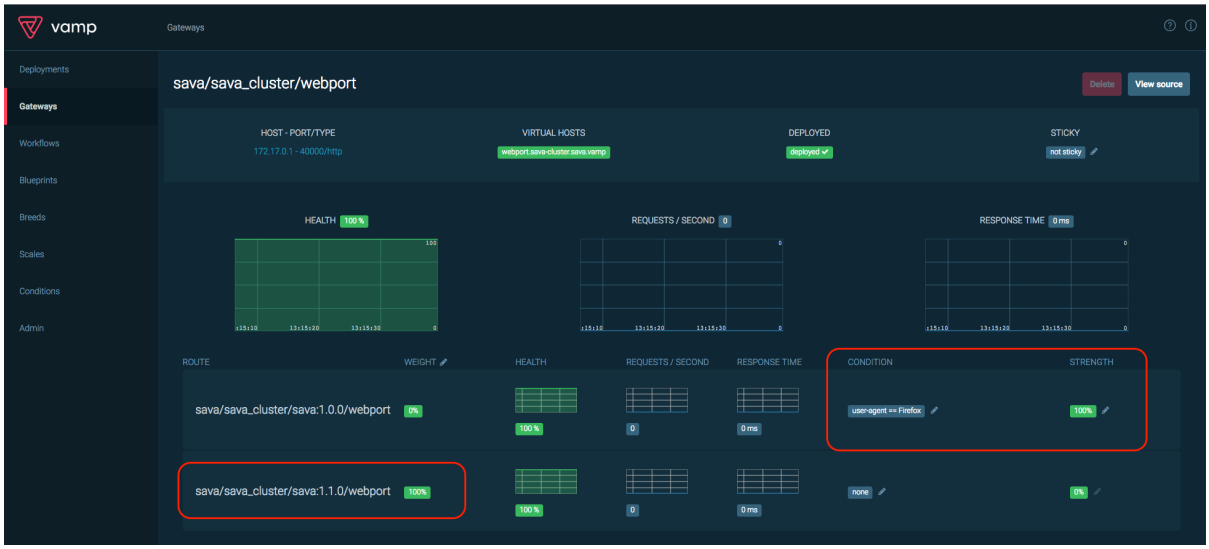
2. Enter the condition **user-agent == Firefox** and click  .

   Vamp adds the condition with a default strength of 0%. To start filtering traffic, you need to adjust the condition strength.

3. Click  to change the **STRENGTH** applied to the condition.

4. Set the **STRENGTH** to 100% and click  to save.

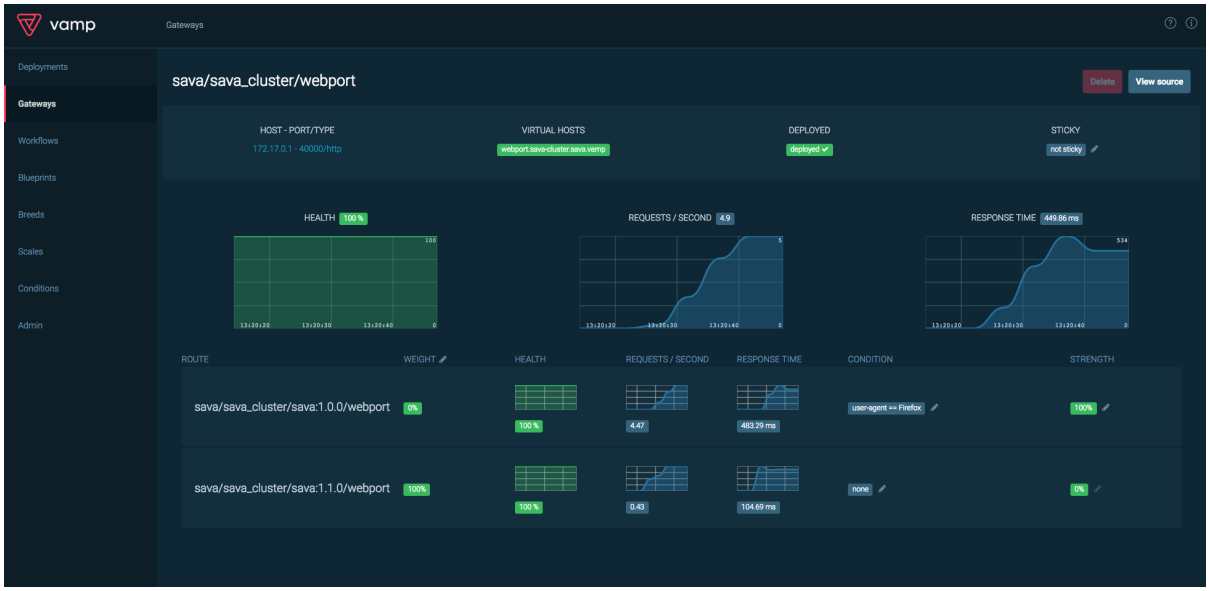
   Vamp now sends all traffic matching the condition (all Firefox users) to sava:1.0.0.



5. Finally, adjust the gateway weight to send all remaining traffic (all non-Firefox users) to the new sava:1.1.0. Click  next to **WEIGHT** and set the weight distribution so 100% is directed to the route sava/sava_cluster/sava:1.1.0/webport.

   All traffic not filtered by the condition is now directed to the new sava:1.1.0.

6. To see the filter in action, open two different browsers (one Firefox and one other browser) and access the sava service from both. All Firefox requests are sent to sava:1.0.0, while all other browsers are directed to sava:1.1.0.



Summing up

This article was a quick introduction to Vamp on a DC/OS cluster. For starters, you got Vamp up and running on your Azure Container Service DC/OS cluster, deployed a service with a Vamp blueprint, and accessed it at the exposed endpoint (gateway).

We also touched on some powerful features of Vamp: merging a new service variant to the running deployment and introducing it incrementally, then filtering traffic to resolve a known incompatibility.

Next steps

- Learn about managing Vamp actions through the Vamp REST API.
- Build Vamp automation scripts in Node.js and run them as Vamp workflows.
- See additional VAMP tutorials.

Monitor an Azure Container Service cluster with ELK

6/27/2017 • 1 min to read • Edit Online

In this article, we demonstrate how to deploy the ELK (Elasticsearch, Logstash, Kibana) stack on a DC/OS cluster in Azure Container Service.

Prerequisites

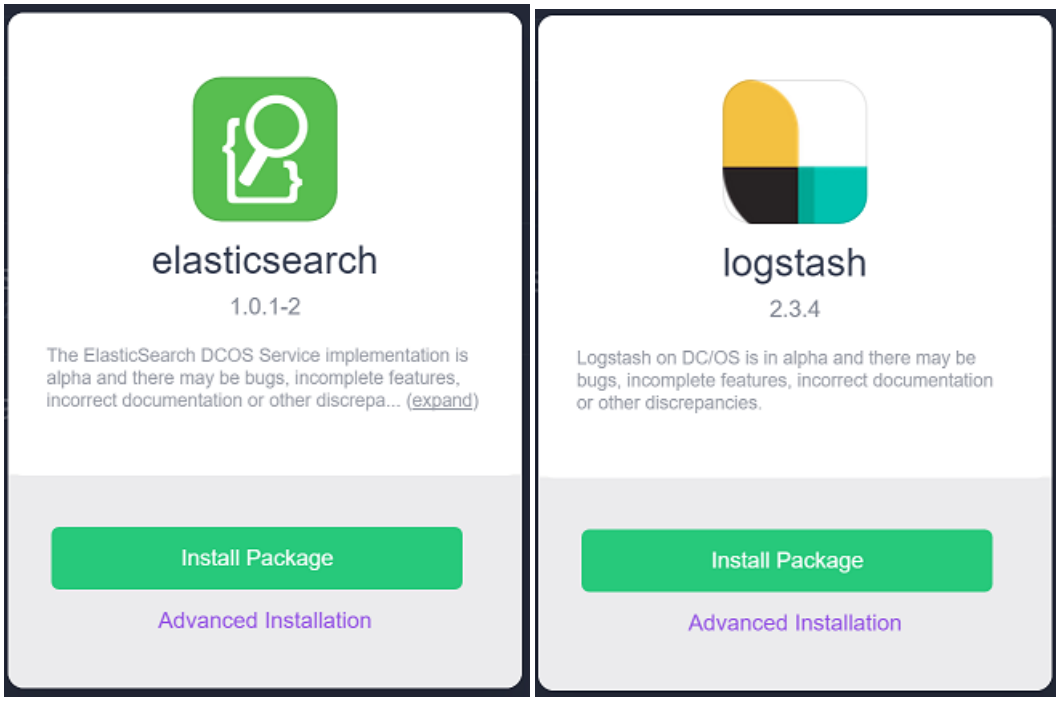
Deploy and connect a DC/OS cluster configured by Azure Container Service. Explore the DC/OS dashboard and Marathon services here. Also install the Marathon Load Balancer.

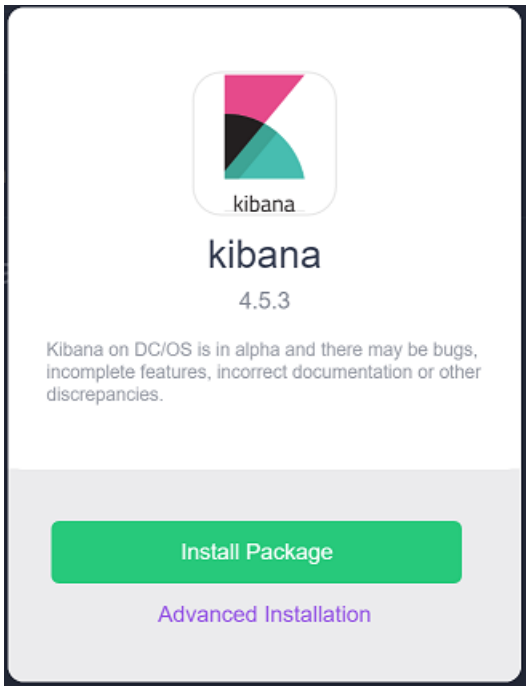
ELK (Elasticsearch, Logstash, Kibana)

ELK stack is a combination of Elasticsearch, Logstash, and Kibana that provides an end to end stack that can be used to monitor and analyze logs in your cluster.

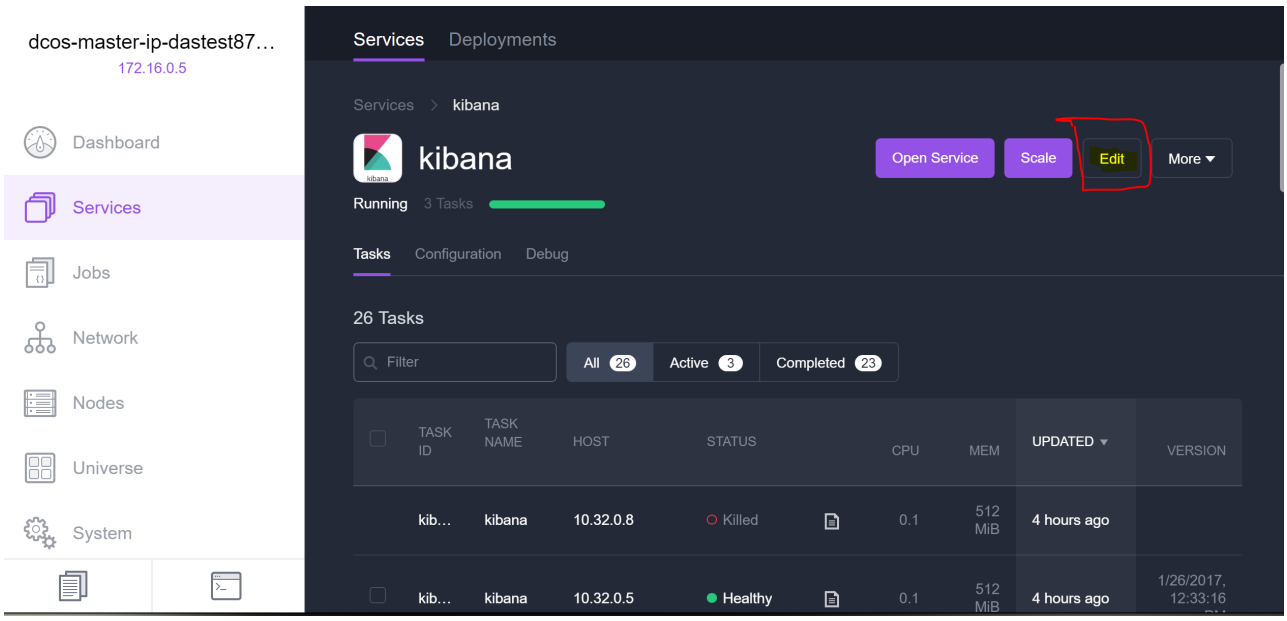
Configure the ELK stack on a DC/OS cluster

Access your DC/OS UI via http://localhost:80/ Once in the DC/OS UI navigate to **Universe**. Search and install Elasticsearch, Logstash, and Kibana from the DC/OS Universe and in that specific order. You can learn more about configuration if you go to the **Advanced Installation** link.

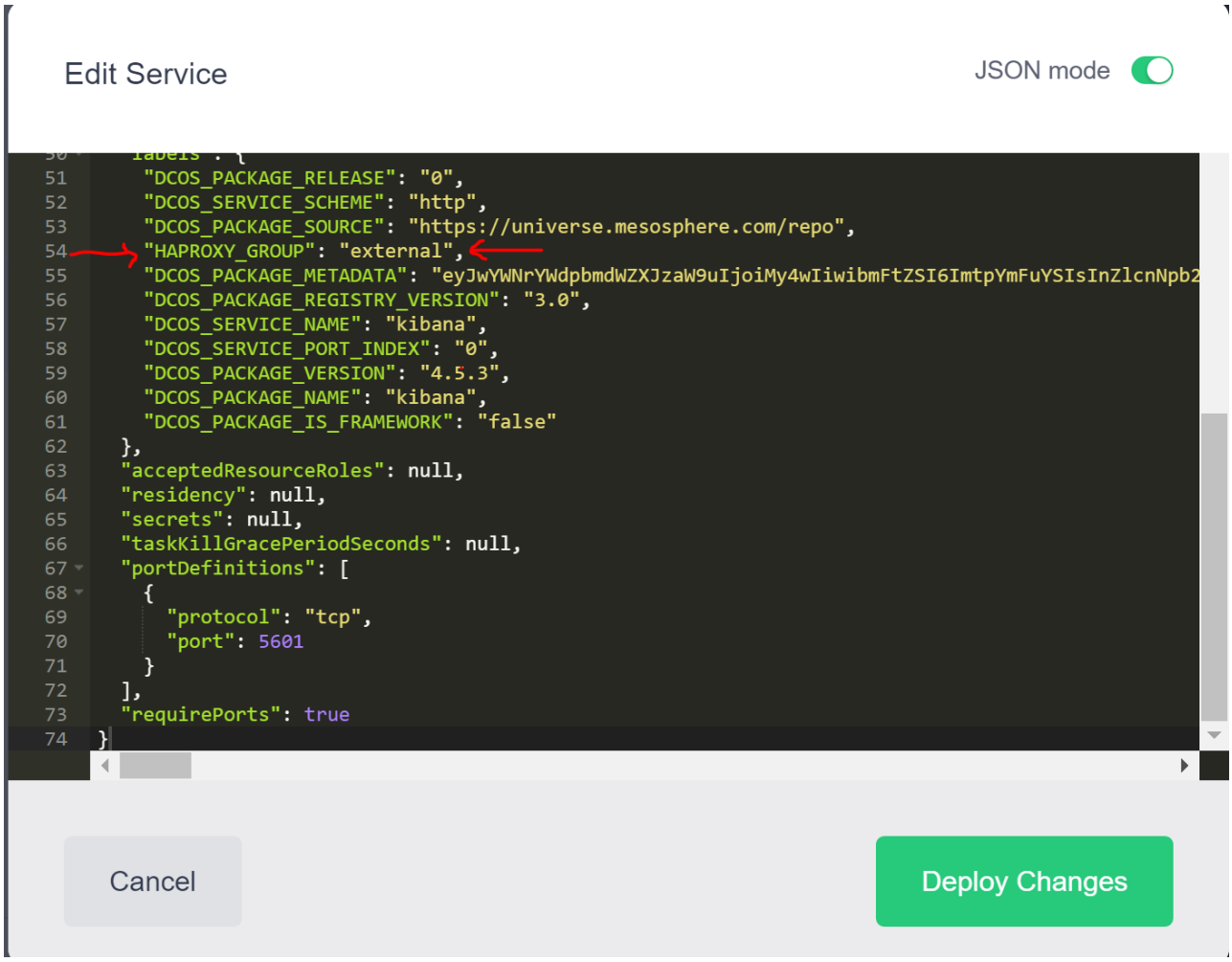




Once the ELK containers and are up and running, you need to enable Kibana to be accessed through Marathon-LB. Navigate to **Services** > **kibana**, and click **Edit** as shown below.

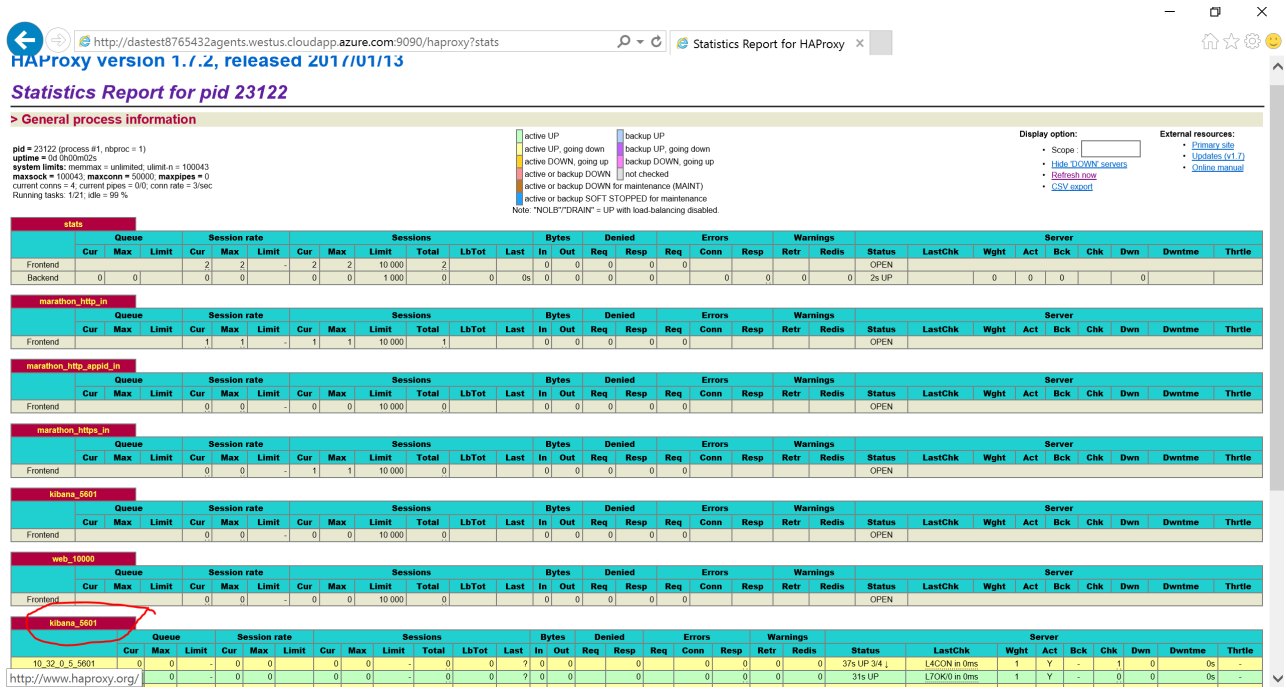


Toggle to **JSON mode** and scroll down to the labels section. You need to add a `"HAPROXY_GROUP": "external"` entry here as shown below. Once you click **Deploy changes**, your container restarts.



If you want to verify that Kibana is registered as a service in the HAPROXY dashboard, you need to open port 9090 on the agent cluster as HAPROXY runs on port 9090. By default, we open ports 80, 8080, and 443 in the DC/OS agent cluster. Instructions to open a port and provide public assess are provided here.

To access the HAPROXY dashboard, open the Marathon-LB admin interface at: `http://$PUBLIC_NODE_IP_ADDRESS:9090/haproxy?stats`. Once you navigate to the URL, you should see the HAPROXY dashboard as shown below and you should see a service entry for Kibana.



To access the Kibana dashboard, which is deployed on port 5601, you need to open port 5601. Follow instructions

here. Then open the Kibana dashboard at: `http://localhost:5601`.

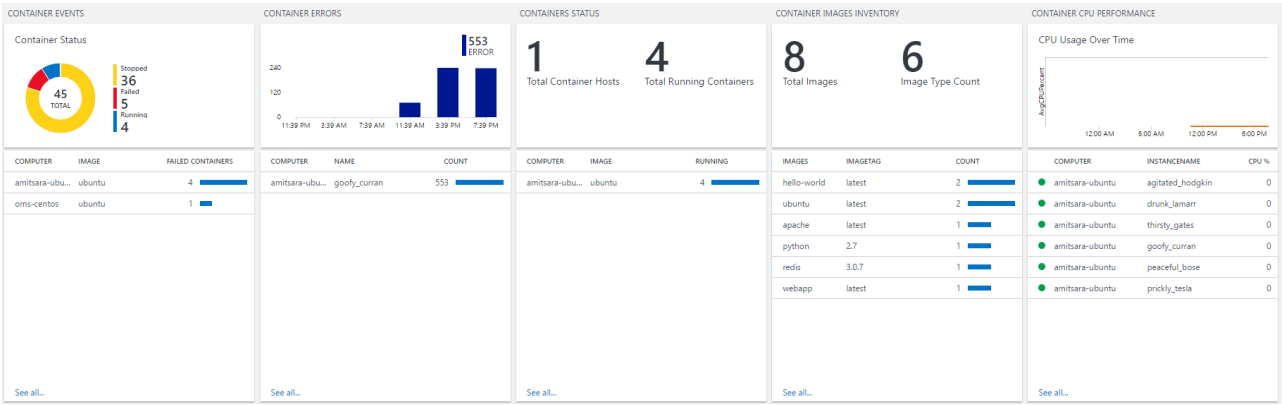
Next steps

- For system and application log forwarding and setup, see Log Management in DC/OS with ELK.
- To filter logs, see Filtering Logs with ELK.

Monitor an Azure Container Service DC/OS cluster with Operations Management Suite

6/27/2017 • 3 min to read • Edit Online

Microsoft Operations Management Suite (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.

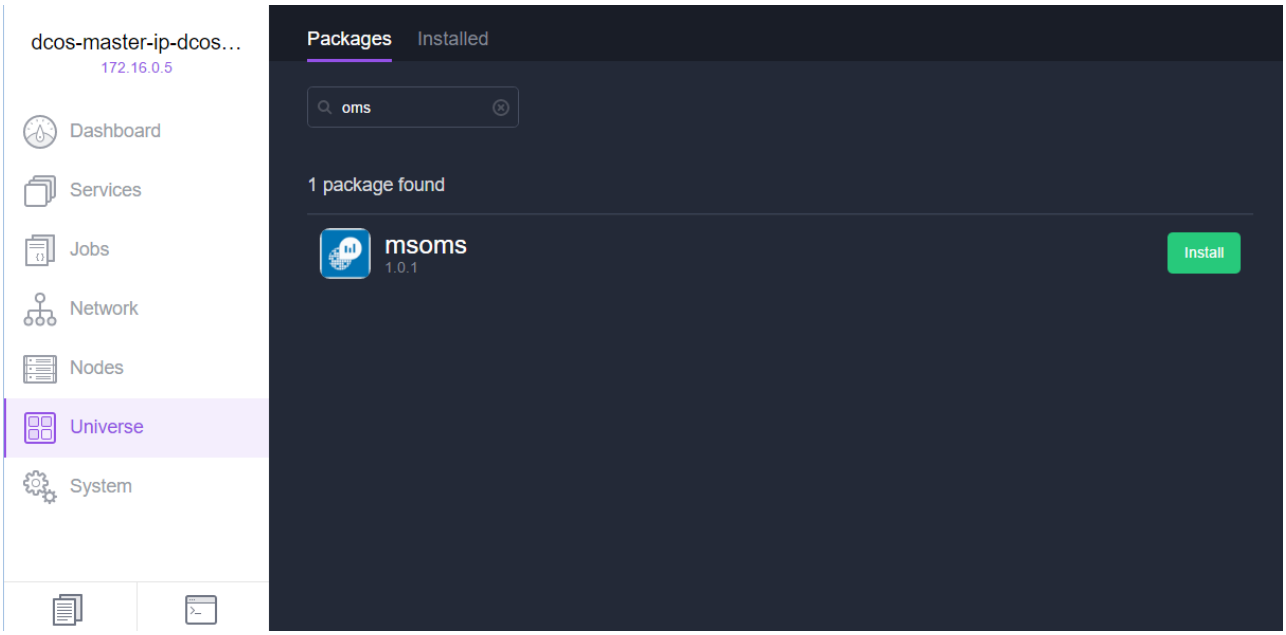
Setting up OMS from the DC/OS universe

This article assumes that you have set up an DC/OS and have deployed simple web container applications on the cluster.

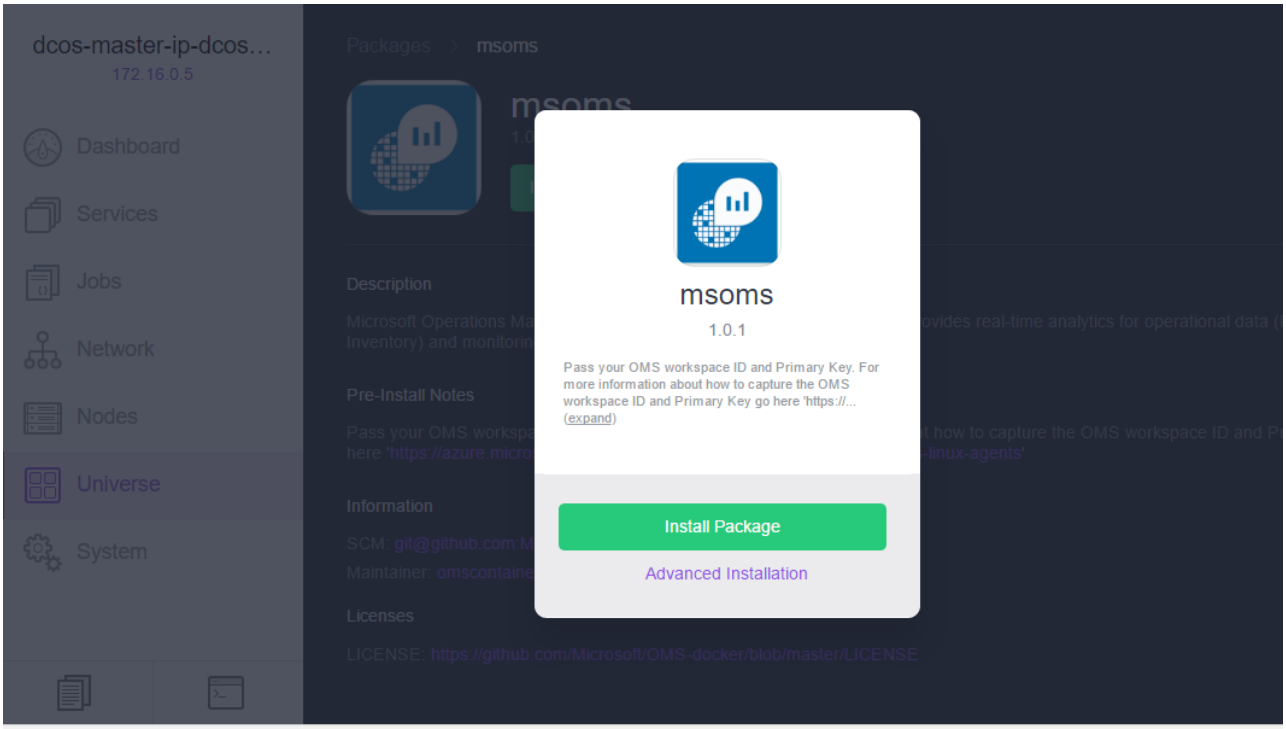
Pre-requisite

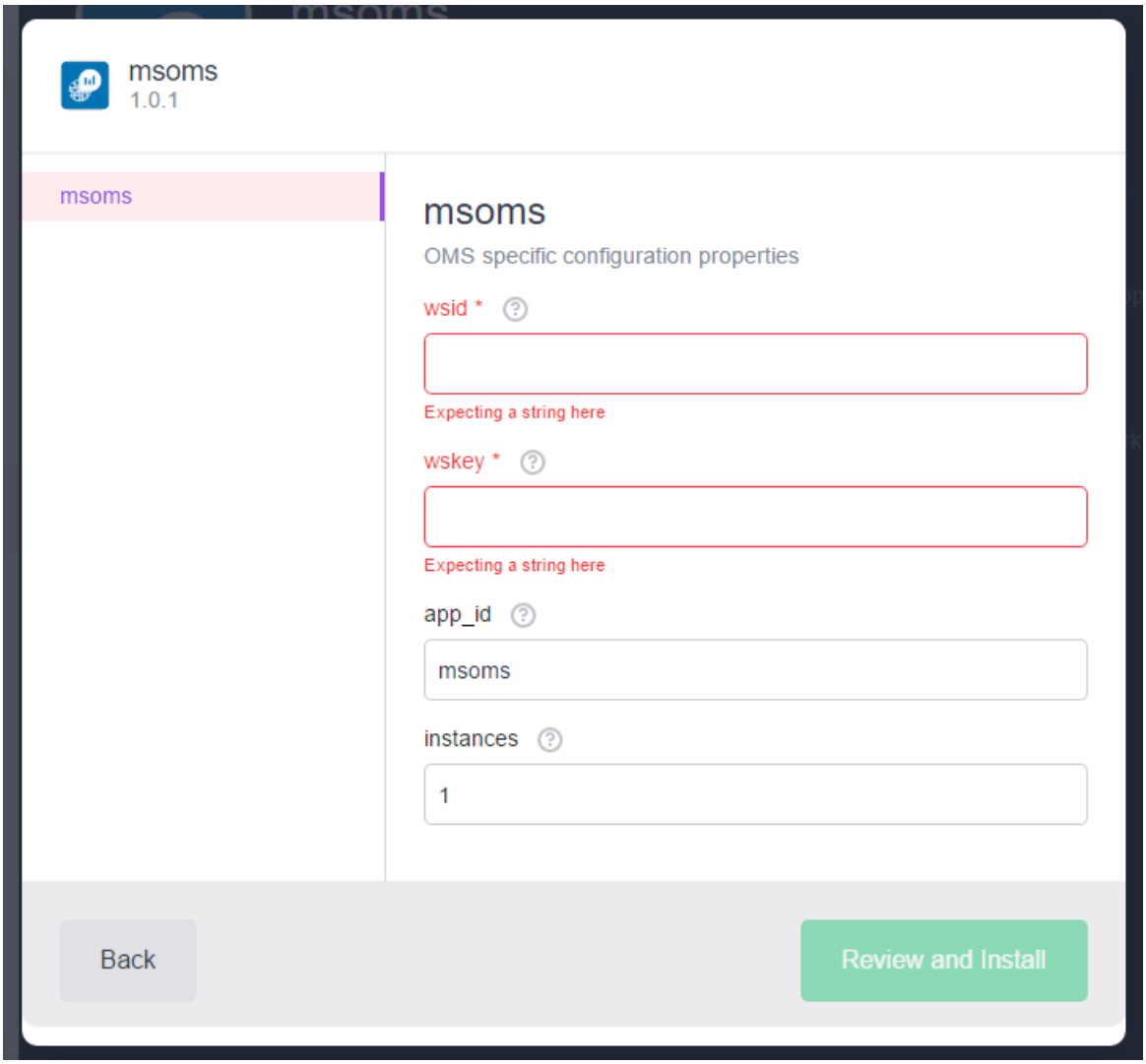
- Microsoft Azure Subscription - You can get this for free.
- Microsoft OMS Workspace Setup - see "Step 3" below
- DC/OS CLI installed.

1. In the DC/OS dashboard, click on Universe and search for 'OMS' as shown below.

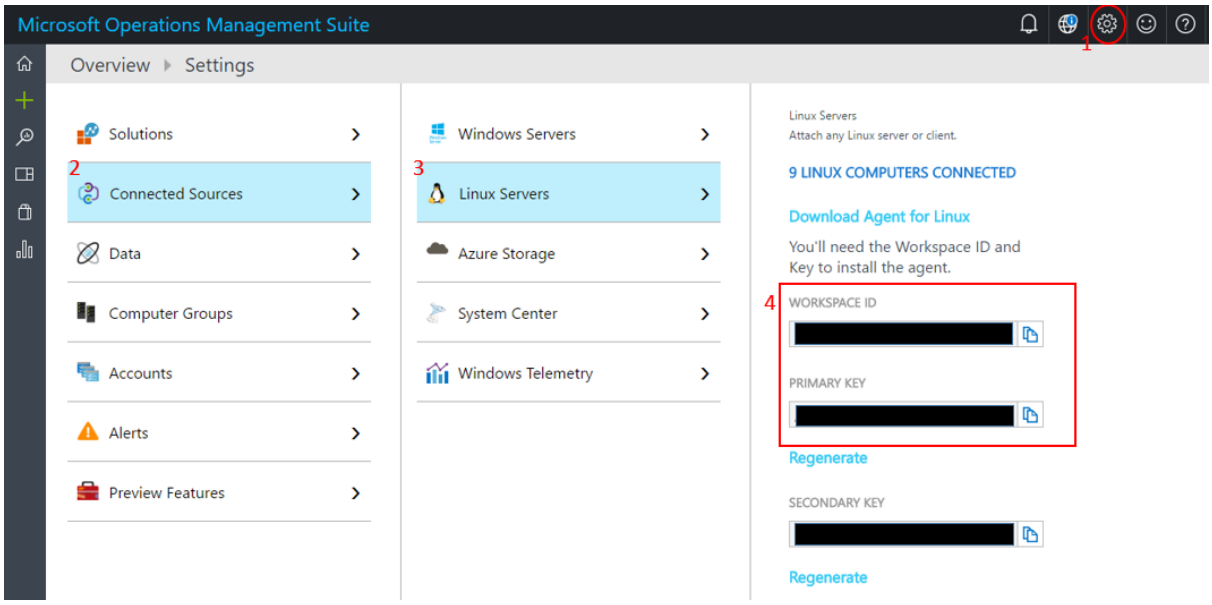


1. Click **Install**. You will see a pop up with the OMS version information and an **Install Package** or **Advanced Installation** button. When you click **Advanced Installation**, which leads you to the **OMS specific configuration properties** page.





1. Here, you will be asked to enter the `wsid` (the OMS workspace ID) and `wskey` (the OMS primary key for the workspace id). To get both `wsid` and `wskey` 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 `wsid` and `wskey` by clicking **Settings**, then **Connected Sources**, and then **Linux Servers**, as shown below.



2. Select the number you OMS instances that you want and click the 'Review and Install' button. Typically, you will want to have the number of OMS instances equal to the number of VM's you have in your agent cluster. OMS Agent for Linux is installs as individual containers on each VM that it wants to collect information for

monitoring and logging information.

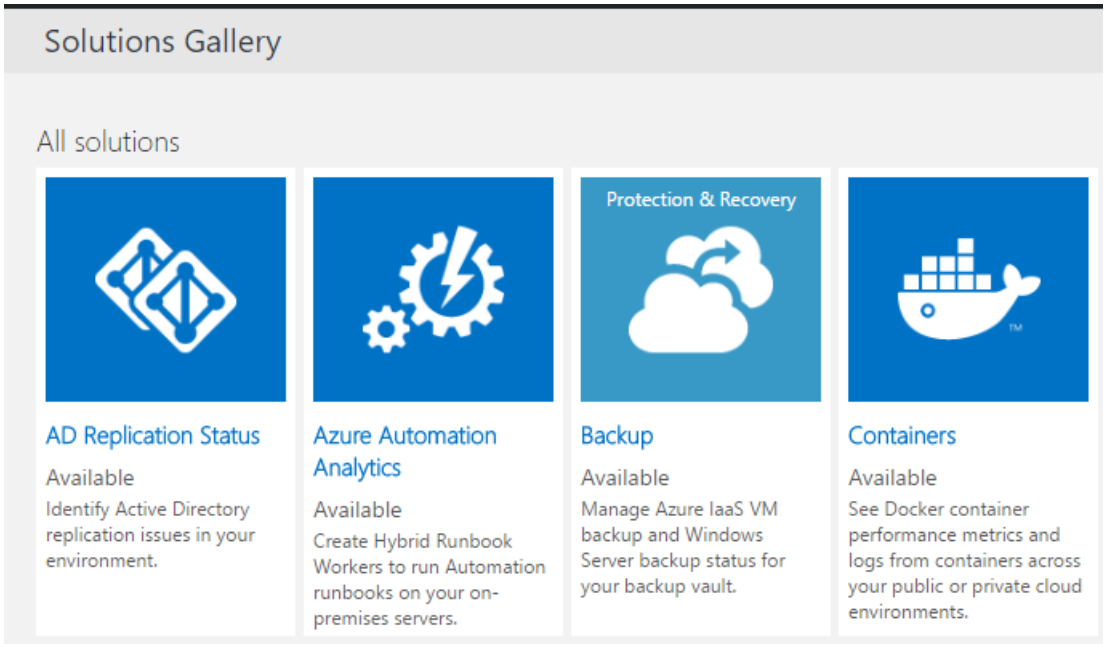
Setting up a simple OMS dashboard

Once you have installed the OMS Agent for Linux on the VMs, next step is to set up the OMS dashboard. There are two ways to do this: OMS Portal or Azure Portal.

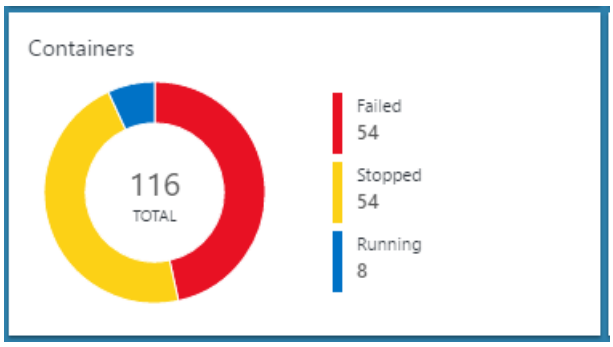
OMS Portal

Log in to the OMS portal (https://mms.microsoft.com) and go to the **Solution Gallery**.

Once you are in the **Solution Gallery**, select **Containers**.

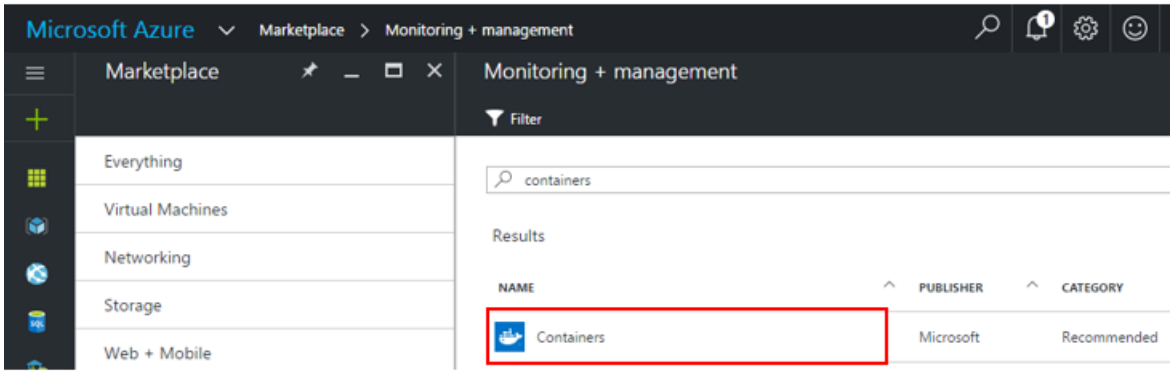


Once you've selected the Container Solution, you will see the tile on the OMS Overview Dashboard page. Once the ingested container data is indexed, you will see the tile populated with information on the solution view tiles.

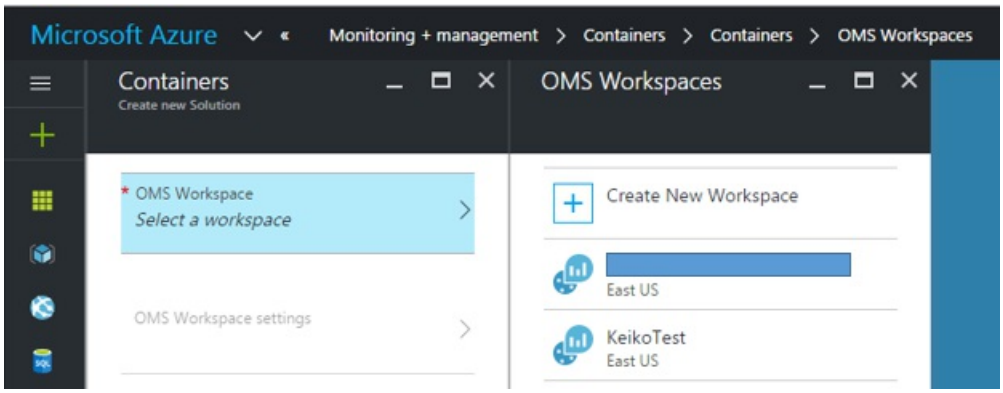


Azure Portal

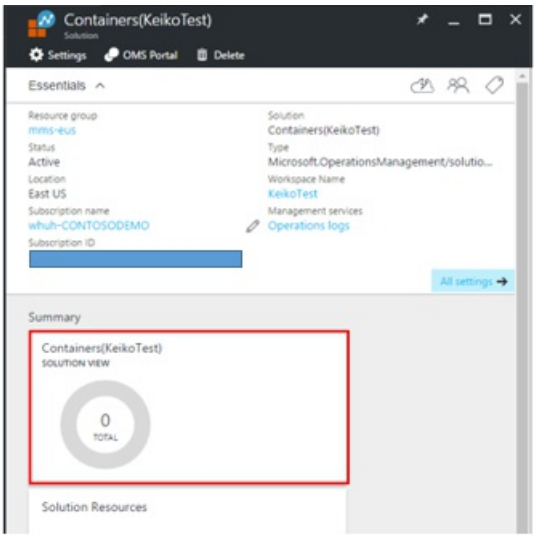
Login to Azure portal at https://portal.microsoft.com/. Go to **Marketplace**, select **Monitoring + management** and click **See All**. Then Type `containers` in search. You will see "containers" in the search results. Select **Containers** and click **Create**.



Once you click **Create**, it will ask you for your workspace. Select your workspace or if you do not have one, create a new workspace.



Once you've selected your workspace, click **Create**.

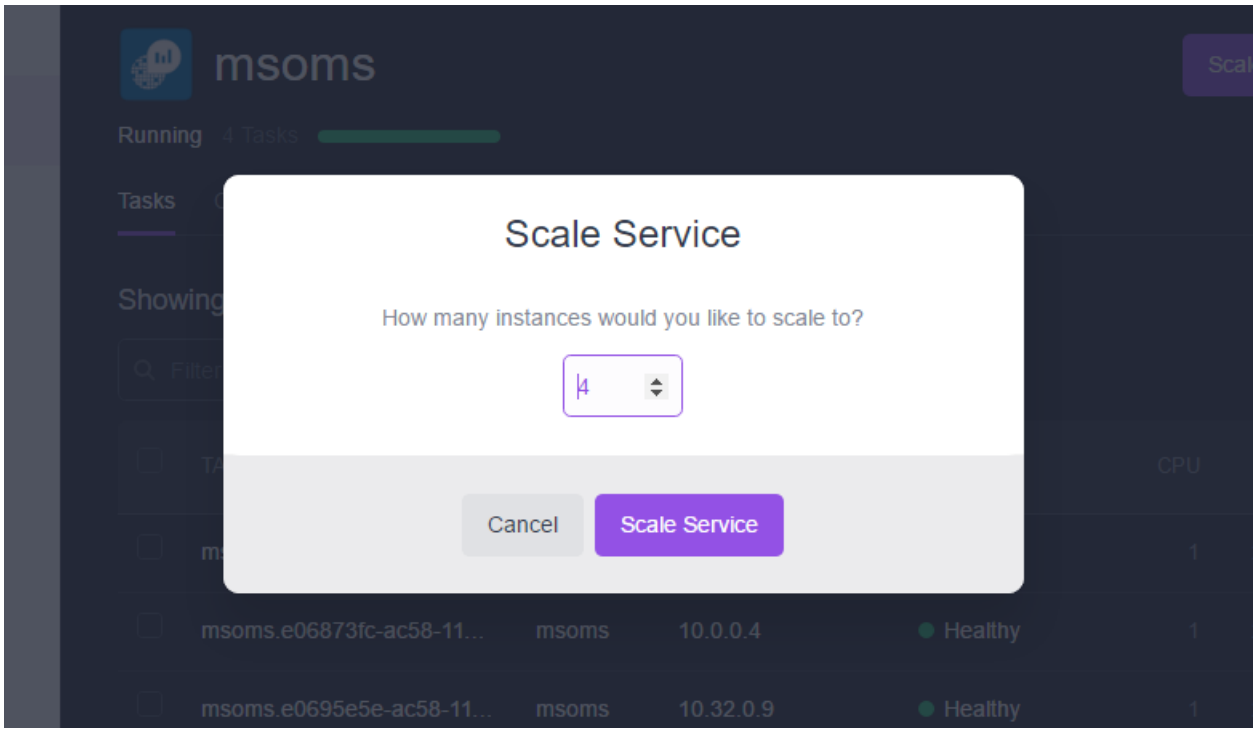


For more information about the OMS Container Solution, please refer to the Container Solution Log Analytics.

How to scale OMS Agent with ACS DC/OS

In case you need to have installed OMS agent short of the actual node count or you are scaling up VMSS by adding more VM, you can do so by scaling the `msoms` service.

You can either go to Marathon or the DC/OS UI Services tab and scale up your node count.



This will deploy to other nodes which have not yet deployed the OMS agent.

Uninstall MS OMS

To uninstall MS OMS enter the following command:

```
$ dcos package uninstall msoms
```

Let us know!!!

What works? What is missing? What else do you need for this to be useful for you? Let us know at OMSContainers.

Next steps

Now that you have set up OMS to monitor your containers,see your container dashboard.

Monitor an Azure Container Service DC/OS cluster with Datadog

6/27/2017 • 1 min to read • Edit Online

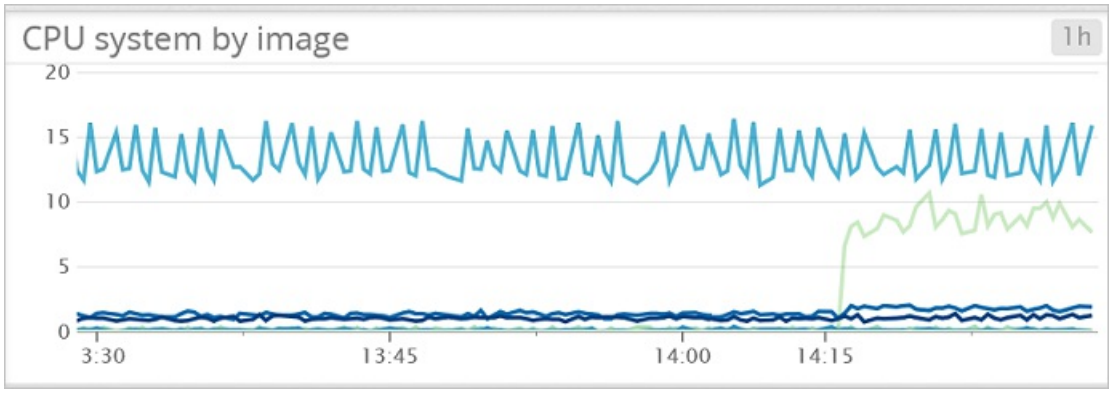
In this article we will deploy Datadog agents to all the agent nodes in your Azure Container Service cluster. You will need an account with Datadog for this configuration.

Prerequisites

Deploy and connect a cluster configured by Azure Container Service. Explore the Marathon UI. Go to http://datadoghq.com to set up a Datadog account.

Datadog

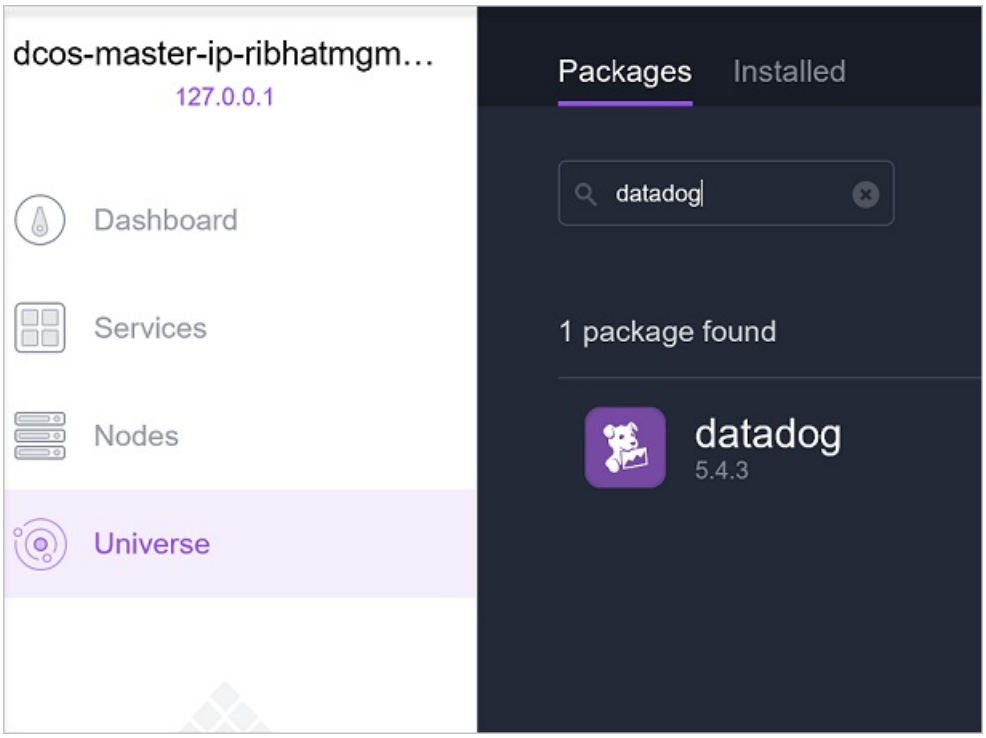
Datadog is a monitoring service that gathers monitoring data from your containers within your Azure Container Service cluster. Datadog has a Docker Integration Dashboard where you can see specific metrics within your containers. Metrics gathered from your containers are organized by CPU, Memory, Network and I/O. Datadog splits metrics into containers and images. An example of what the UI looks like for CPU usage is below.



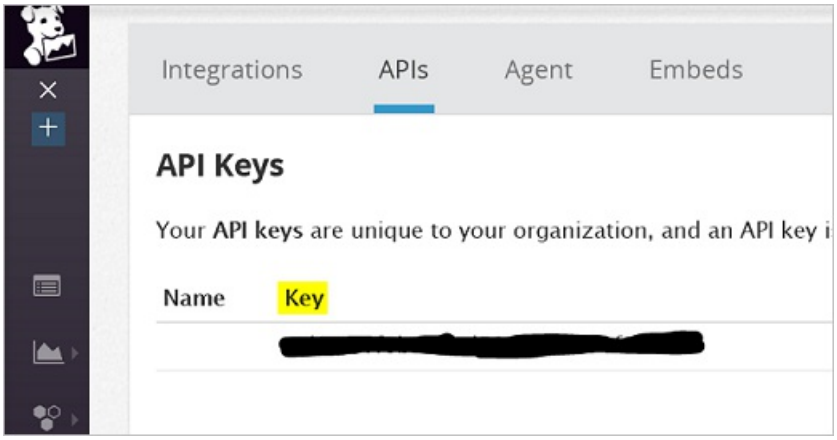
Configure a Datadog deployment with Marathon

These steps will show you how to configure and deploy Datadog applications to your cluster with Marathon.

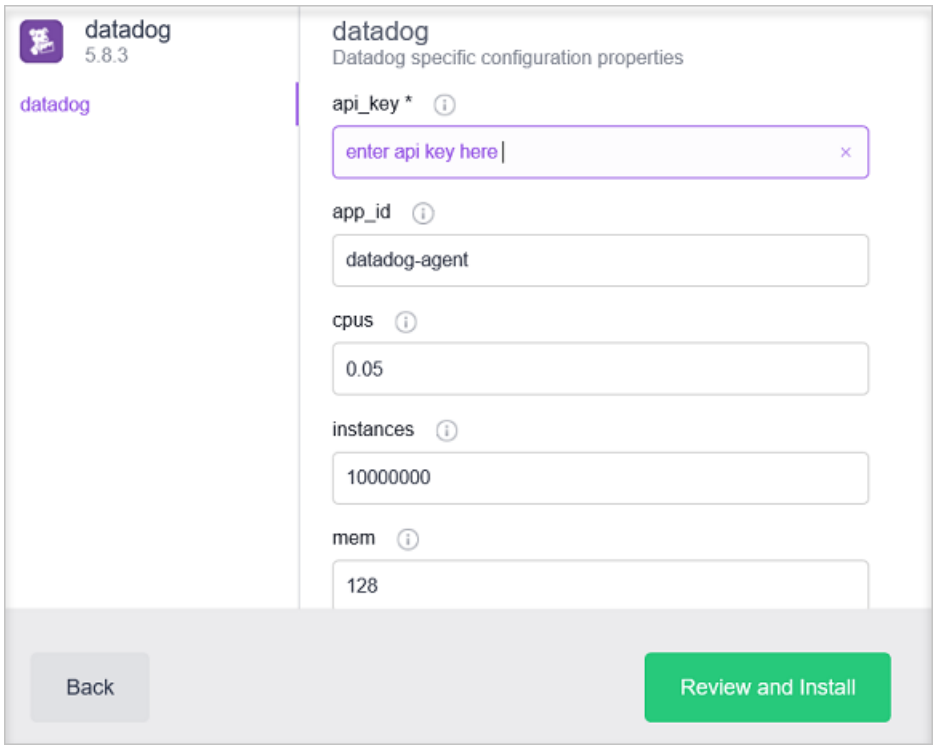
Access your DC/OS UI via http://localhost:80/. Once in the DC/OS UI navigate to the "Universe" which is on the bottom left and then search for "Datadog" and click "Install."



Now to complete the configuration you will need a Datadog account or a free trial account. Once you're logged in to the Datadog website look to the left and go to Integrations -> then APIs.



Next enter your API key into the Datadog configuration within the DC/OS Universe.



In the above configuration instances are set to 10000000 so whenever a new node is added to the cluster Datadog will automatically deploy an agent to that node. This is an interim solution. Once you've installed the package you should navigate back to the Datadog website and find "Dashboards." From there you will see Custom and Integration Dashboards. The Docker dashboard will have all the container metrics you need for monitoring your cluster.

Monitor an Azure Container Service cluster with Sysdig

6/27/2017 • 1 min to read • Edit Online

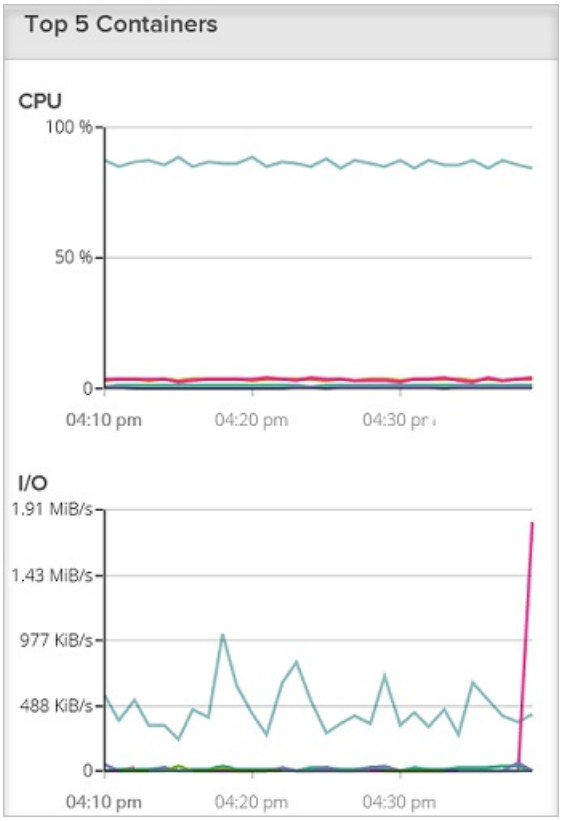
In this article, we will deploy Sysdig agents to all the agent nodes in your Azure Container Service cluster. You need an account with Sysdig for this configuration.

Prerequisites

Deploy and connect a cluster configured by Azure Container Service. Explore the Marathon UI. Go to http://app.sysdigcloud.com to set up a Sysdig cloud account.

Sysdig

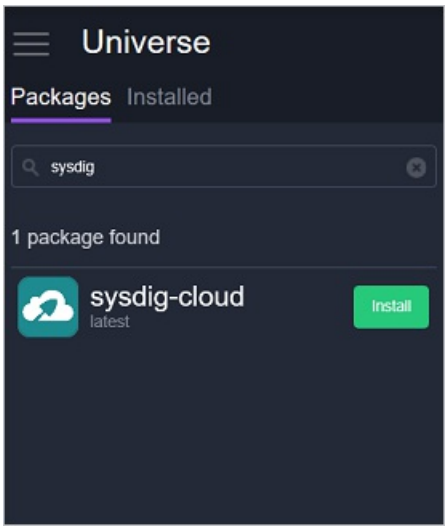
Sysdig is a monitoring service that allows you to monitor your containers within your cluster. Sysdig is known to help with troubleshooting but it also has your basic monitoring metrics for CPU, Networking, Memory, and I/O. Sysdig makes it easy to see which containers are working the hardest or essentially using the most memory and CPU. This view is in the "Overview" section, which is currently in beta.



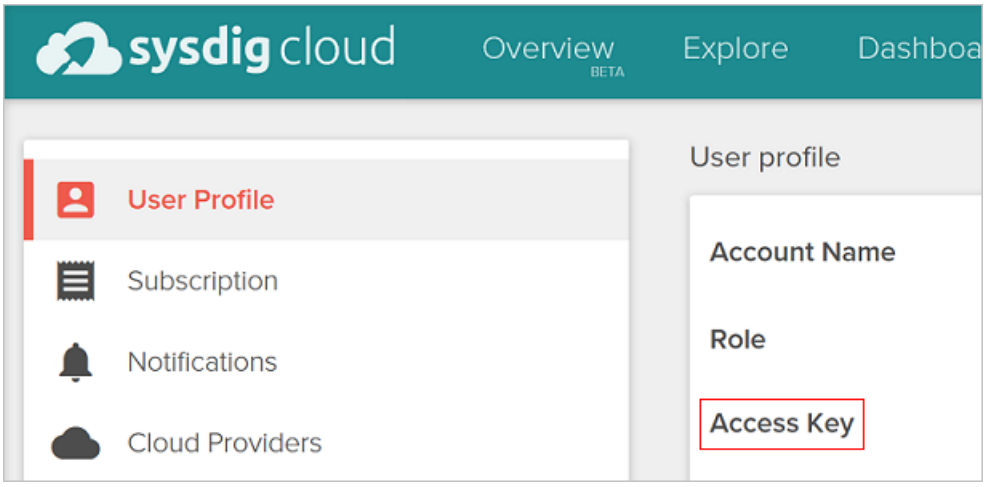
Configure a Sysdig deployment with Marathon

These steps will show you how to configure and deploy Sysdig applications to your cluster with Marathon.

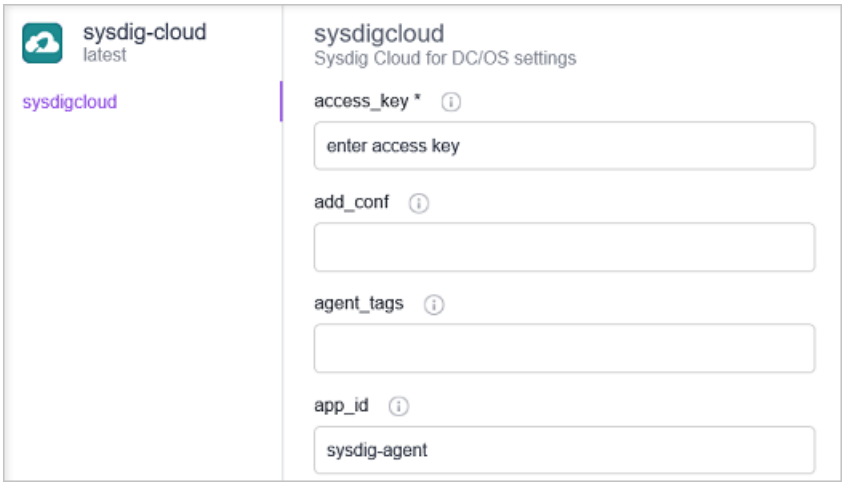
Access your DC/OS UI via http://localhost:80/ Once in the DC/OS UI navigate to the "Universe", which is on the bottom left and then search for "Sysdig."



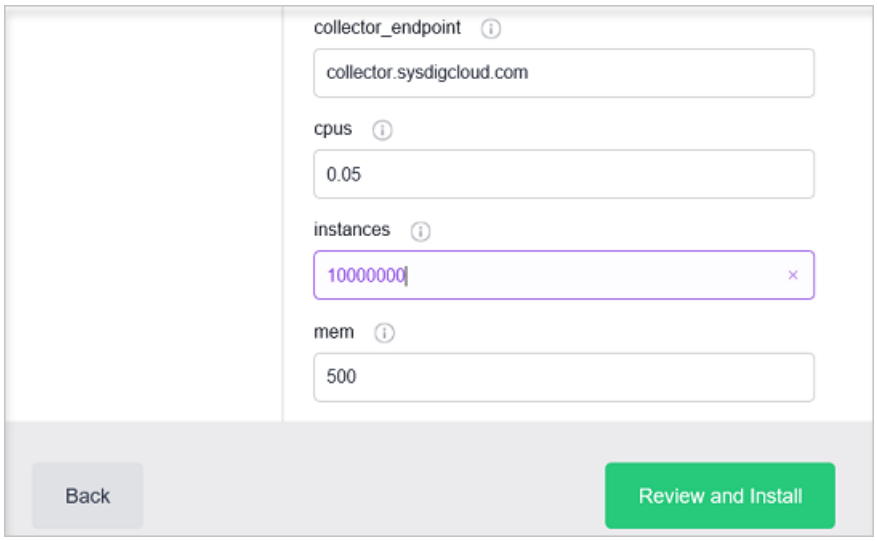
Now to complete the configuration you need a Sysdig cloud account or a free trial account. Once you're logged in to the Sysdig cloud website, click on your user name, and on the page you should see your "Access Key."



Next enter your Access Key into the Sysdig configuration within the DC/OS Universe.



Now set the instances to 10000000 so whenever a new node is added to the cluster Sysdig will automatically deploy an agent to that new node. This is an interim solution to make sure Sysdig will deploy to all new agents within the cluster.



Once you've installed the package navigate back to the Sysdig UI and you'll be able to explore the different usage metrics for the containers within your cluster.

You can also install Mesos and Marathon specific dashboards via the new dashboard wizard.

Monitor an Azure Container Service DC/OS cluster with Dynatrace SaaS/Managed

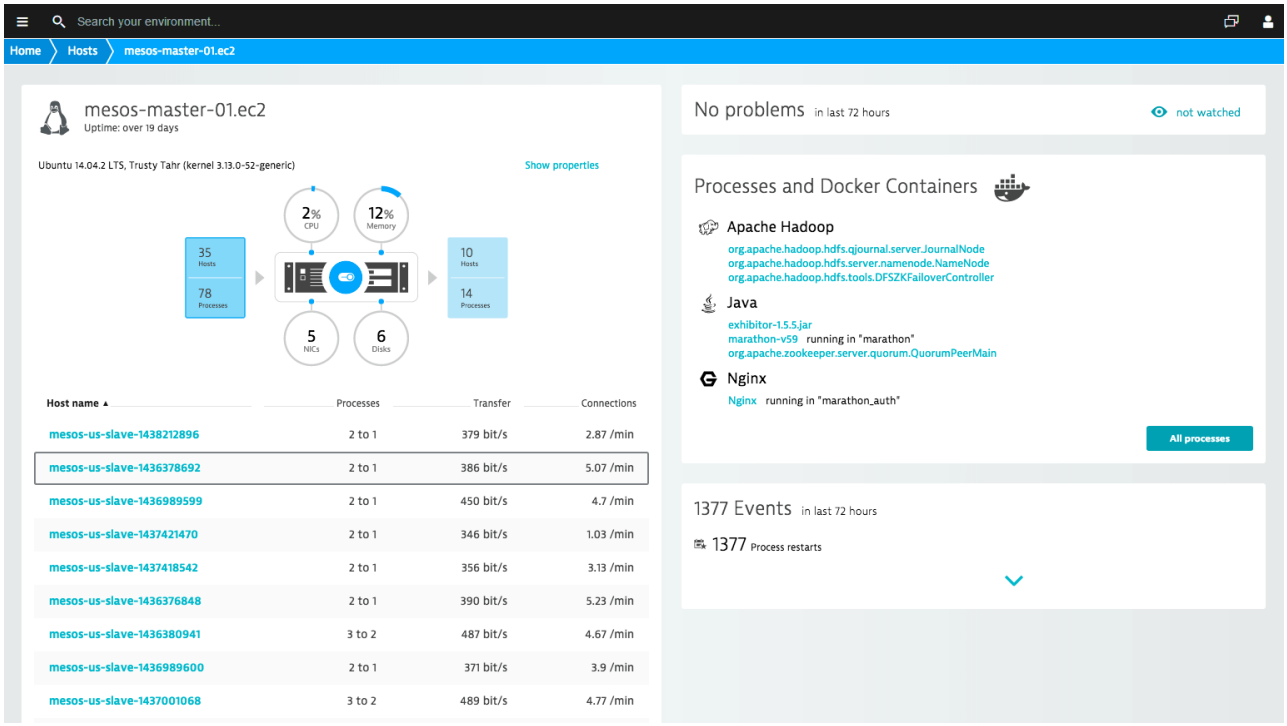
6/27/2017 • 1 min to read • Edit Online

In this article, we show you how to deploy the Dynatrace OneAgent to monitor all the agent nodes in your Azure Container Service cluster. You need an account with Dynatrace SaaS/Managed for this configuration.

Dynatrace SaaS/Managed

Dynatrace is a cloud-native monitoring solution for highly dynamic container and cluster environments. It allows you to better optimize your container deployments and memory allocations by using real-time usage data. It is capable of automatically pinpointing application and infrastructure issues by providing automated baselining, problem correlation, and root-cause detection.

The following figure shows the Dynatrace UI:



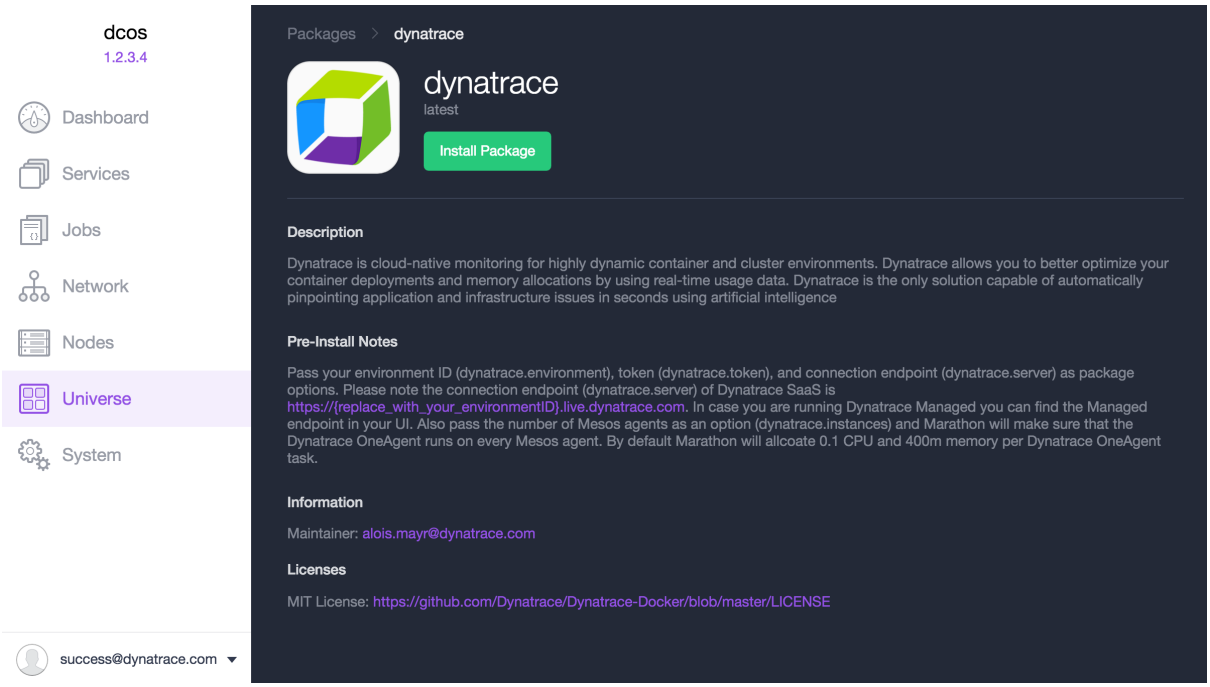
Prerequisites

Deploy and connect to a cluster configured by Azure Container Service. Explore the Marathon UI. Go to https://www.dynatrace.com/trial/ to set up a Dynatrace SaaS account.

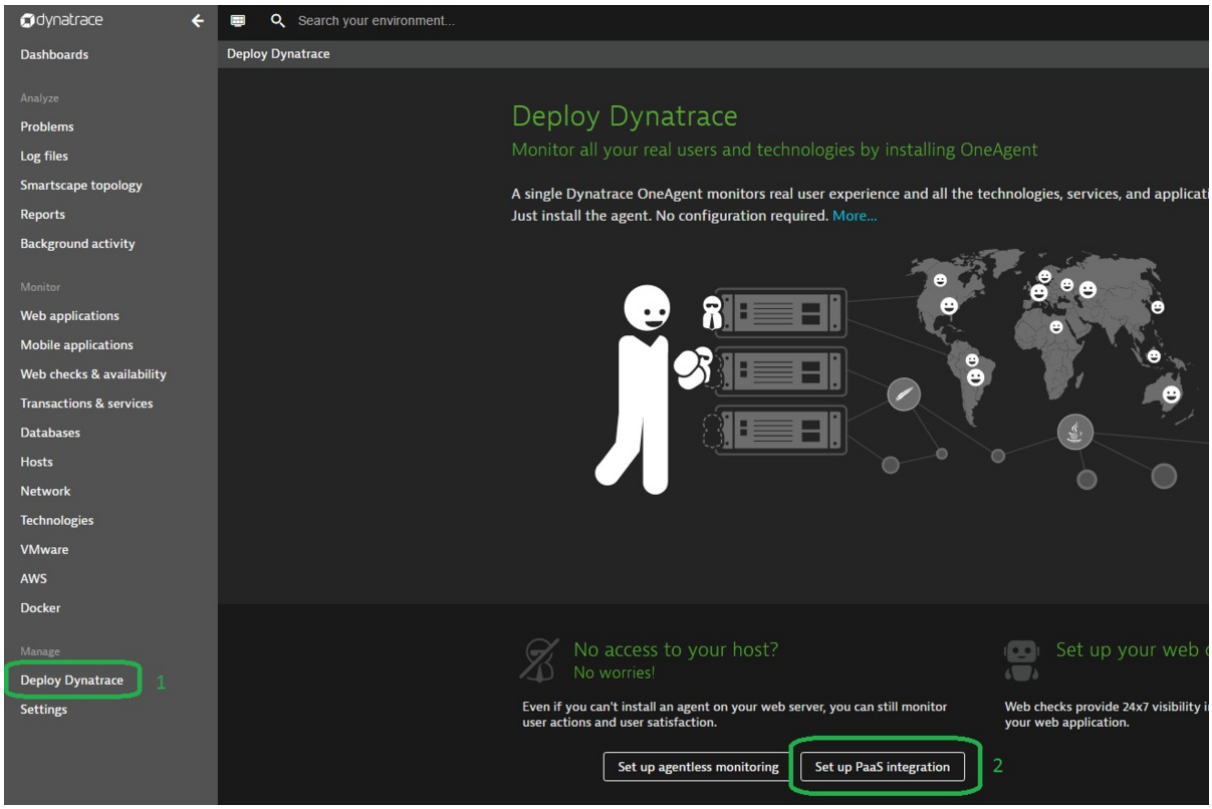
Configure a Dynatrace deployment with Marathon

These steps show you how to configure and deploy Dynatrace applications to your cluster with Marathon.

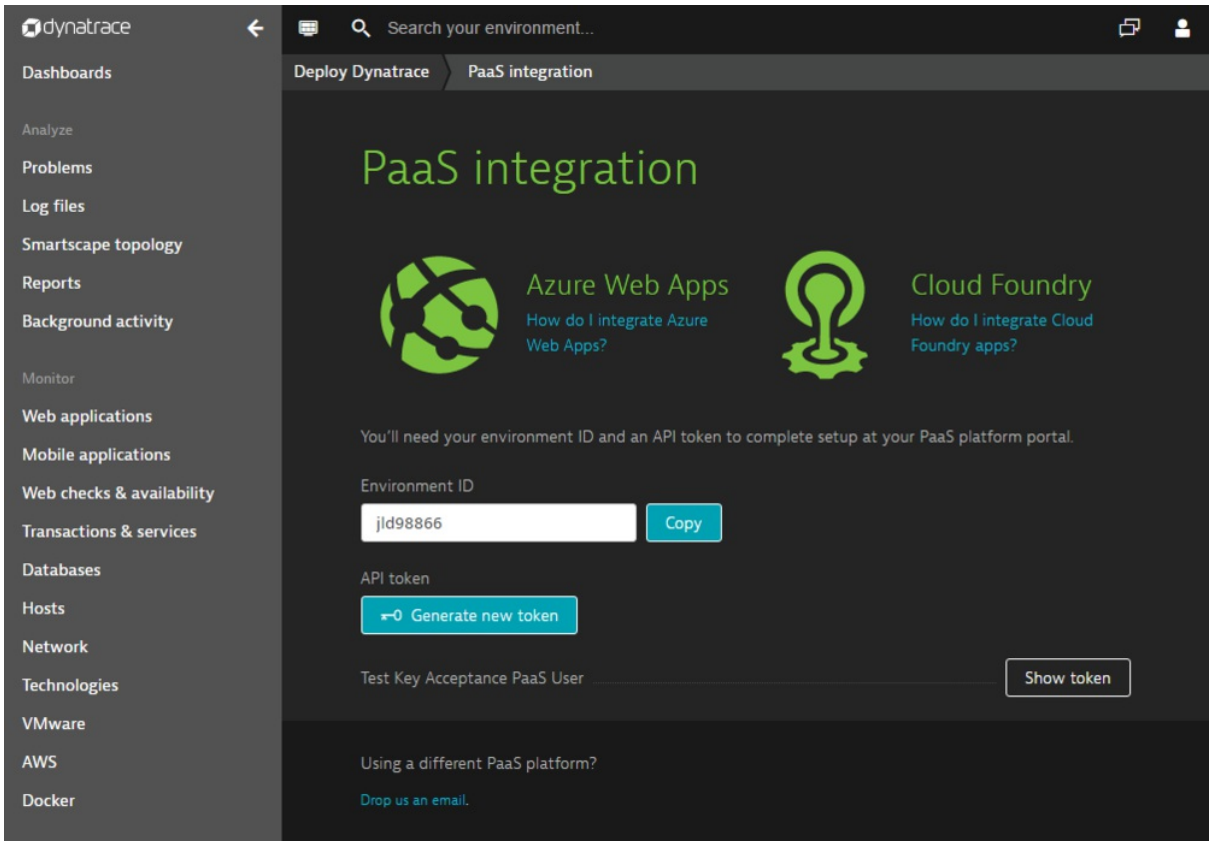
1. Access your DC/OS UI via http://localhost:80/. Once in the DC/OS UI, navigate to the **Universe** tab and then search for **Dynatrace**.



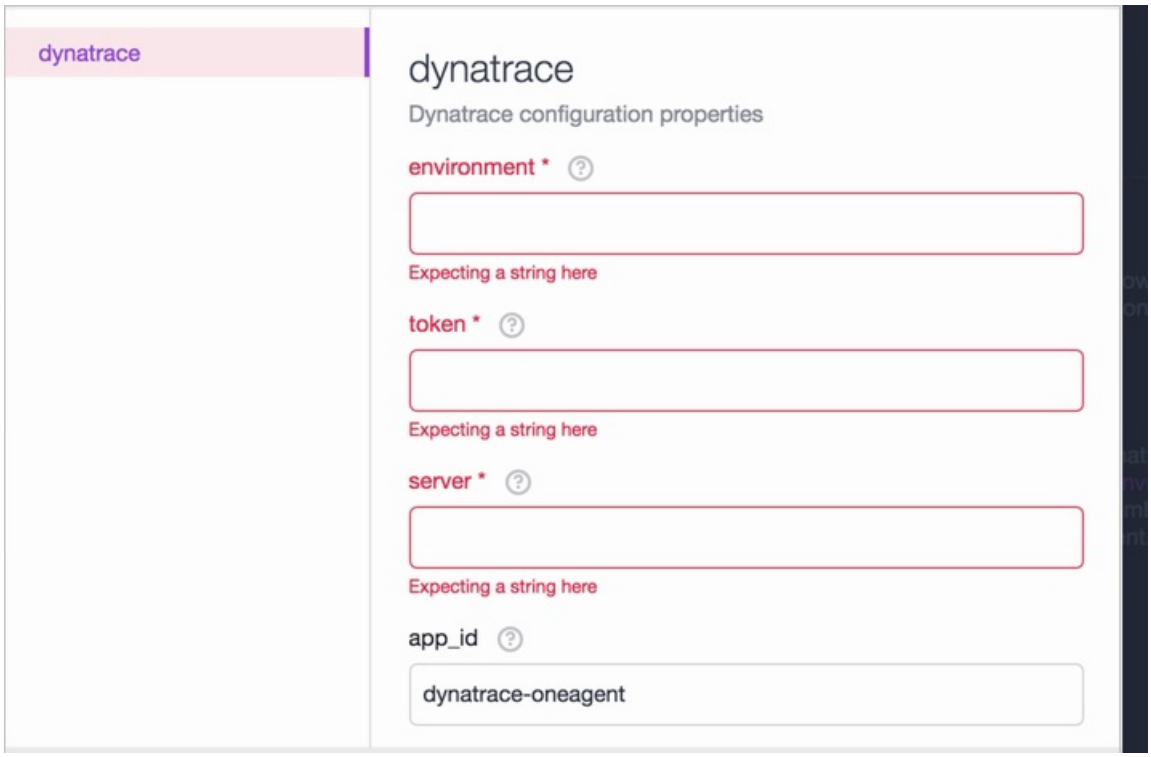
2. To complete the configuration, you need a Dynatrace SaaS account or a free trial account. Once you log into the Dynatrace dashboard, select **Deploy Dynatrace**.



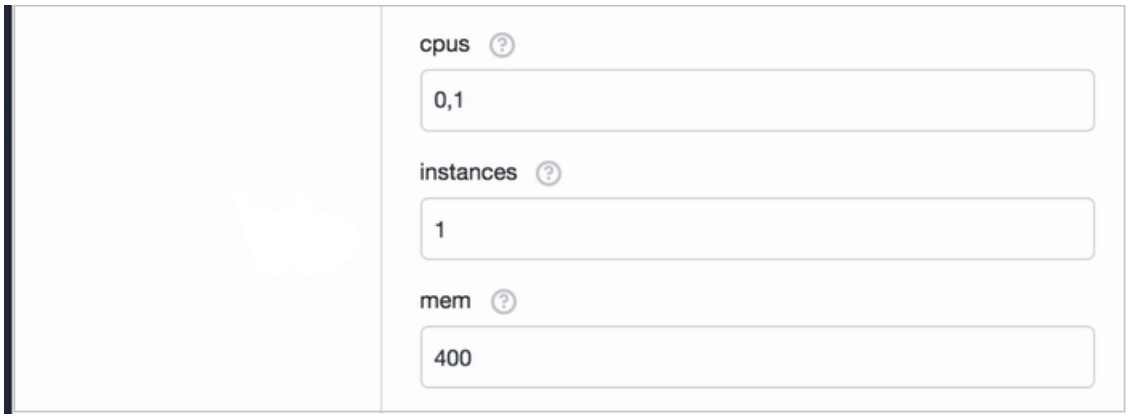
3. On the page, select **Set up PaaS integration**.



4. Enter your API token into the Dynatrace OneAgent configuration within the DC/OS Universe.



5. Set the instances to the number of nodes you intend to run. Setting a higher number also works, but DC/OS will keep trying to find new instances until that number is actually reached. If you prefer, you can also set this to a value like 1000000. In this case, whenever a new node is added to the cluster, Dynatrace automatically deploys an agent to that new node, at the price of DC/OS constantly trying to deploy further instances.



Next steps

Once you've installed the package, navigate back to the Dynatrace dashboard. You can explore the different usage metrics for the containers within your cluster.

Using the Kubernetes web UI with Azure Container Service

6/27/2017 • 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

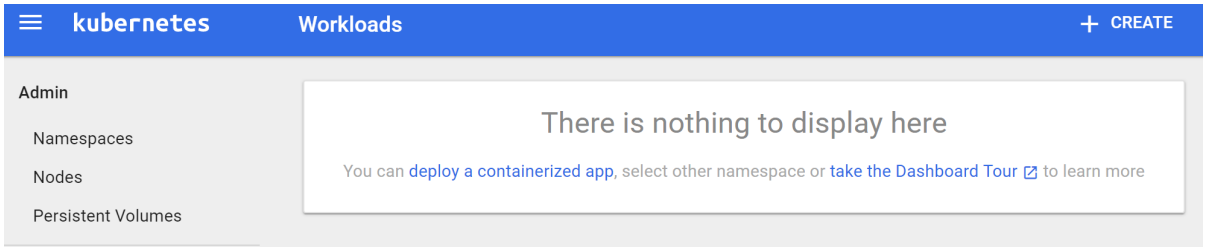
You can launch the Kubernetes web UI by running:

```
$ 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.

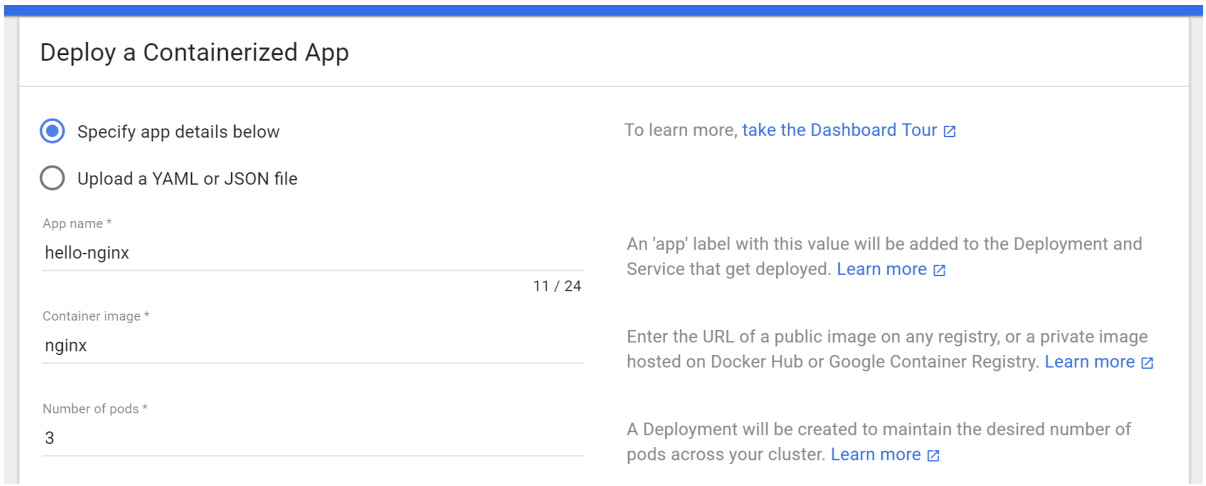
Create and expose a service

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



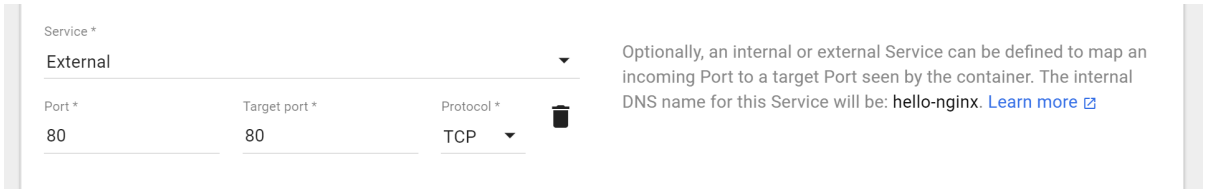
   A dialog box opens where you can start creating your application.

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



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

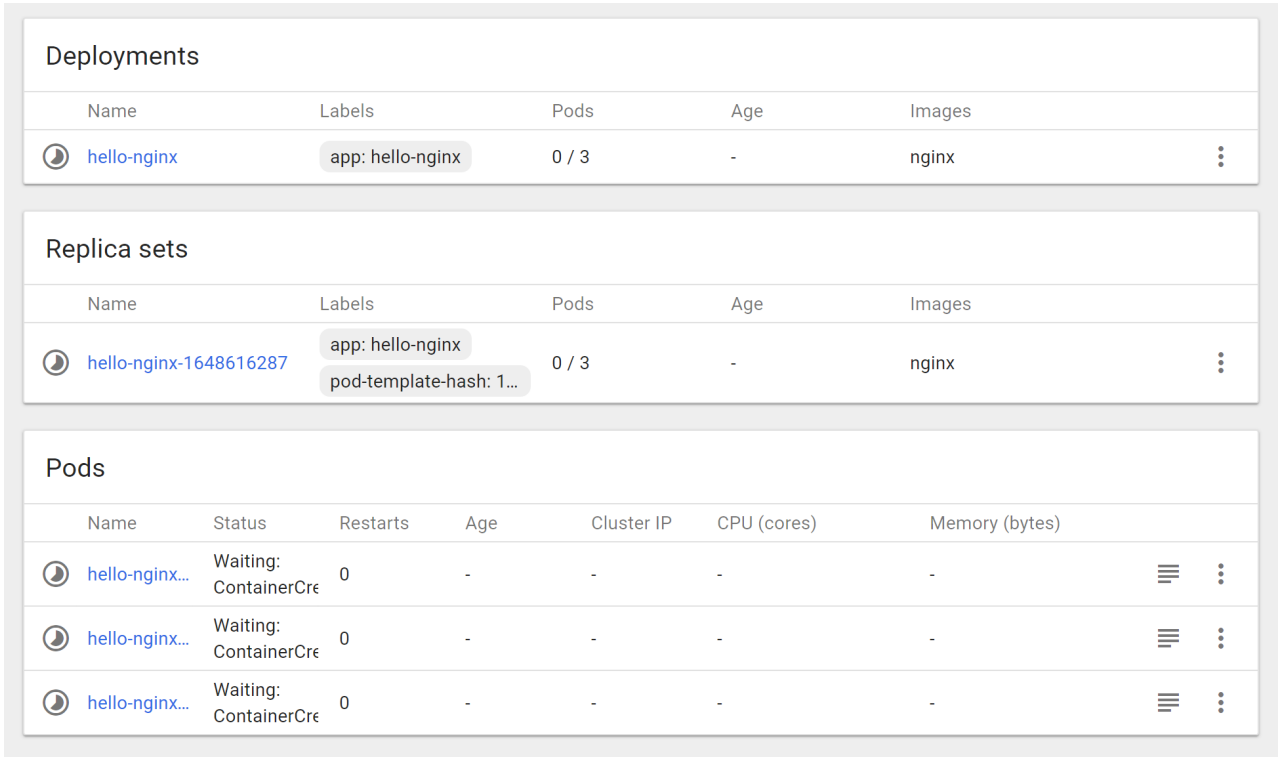
   This setting load-balances traffic to the three replicas.



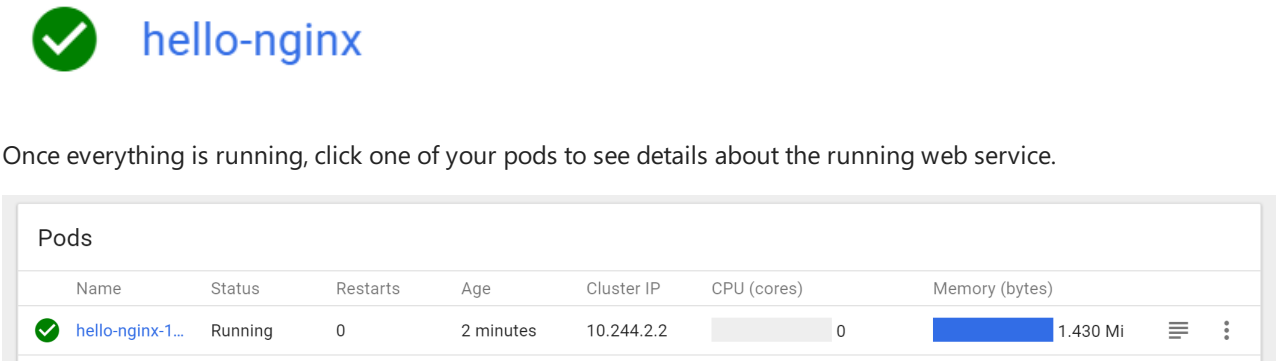
4. Click **Deploy** to deploy these containers and services.

View your containers

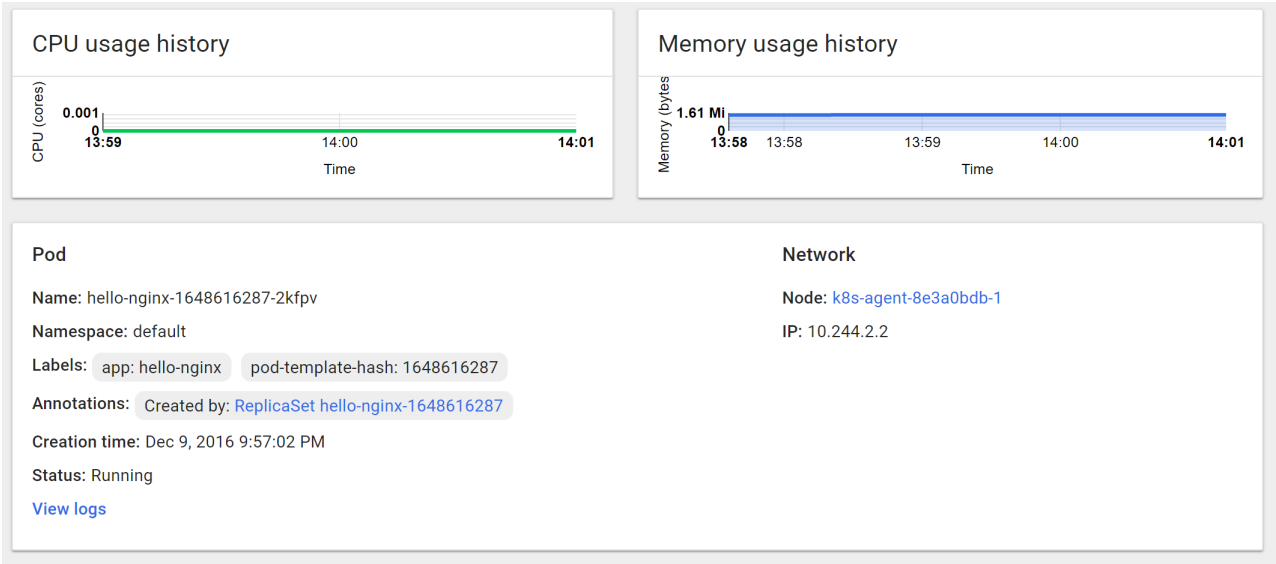
After you click **Deploy**, the UI shows a view of your service as it deploys:



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:

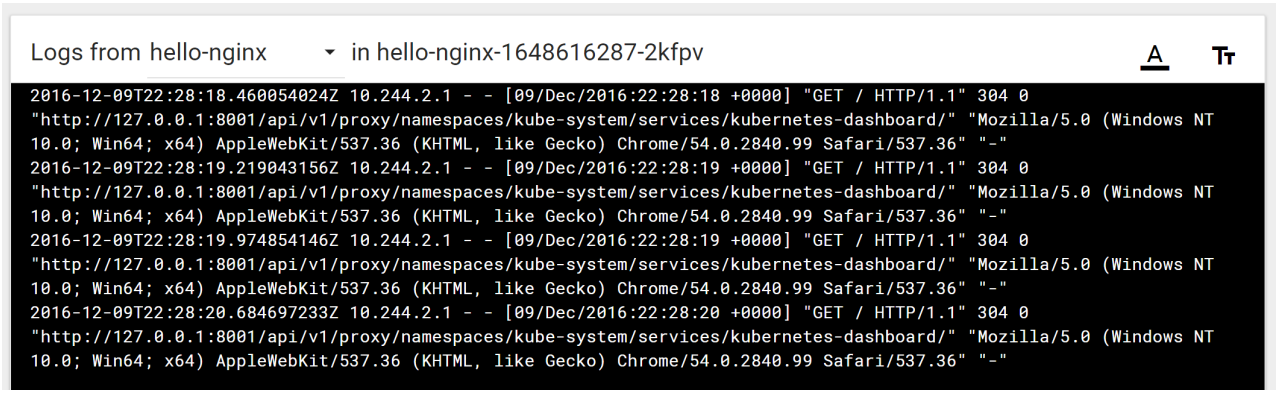


In the **Pods** view, you can see information about the containers in the pod as well as the CPU and memory resources used by those containers:



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

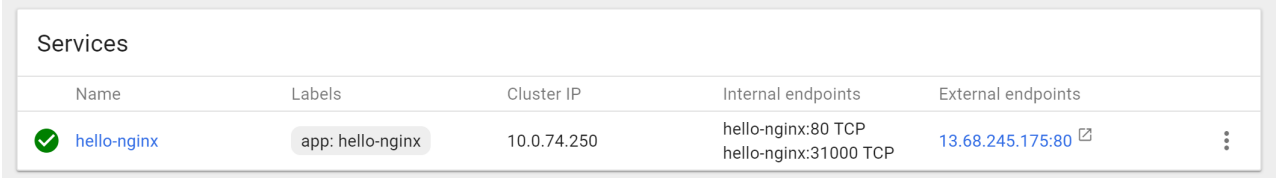
To see the logs for your container, click **View logs**.



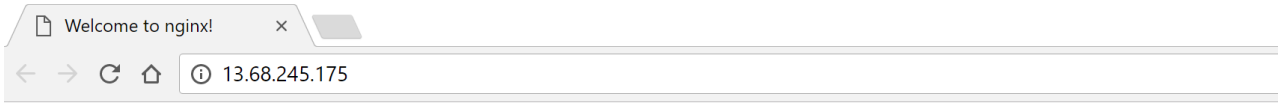
Viewing your service

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).



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

6/27/2017 • 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

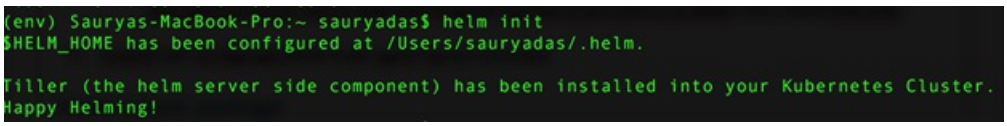
To view information about the Kubernetes cluster that you are installing Tiller and deploying your applications to, type the following command:

```
kubectl cluster-info
```

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:



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/chaoskube                0.4.0   Chaoskube 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

6/27/2017 • 4 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.
2. Install Jenkins via these instructions.
3. A more detailed tutorial is at howtoforge.com.
4. Update the Azure network security group to allow port 8080 and then browse the public IP at port 8080 to manage Jenkins in your browser.
5. Initial Jenkins admin password is stored at /var/lib/jenkins/secrets/initialAdminPassword.
6. Install Docker on the Jenkins machine via these instructions. This allows for Docker commands to be run in Jenkins jobs.
7. Configure Docker permissions to allow Jenkins to access endpoint.

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

8. Install `kubectl` CLI on Jenkins. More details are at Installing and Setting up kubectl.

```
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.

```
export KUBE_MASTER=<your_cluster_master_fqdn>

sudo scp -3 -i ~/.ssh/id_rsa azureuser@$KUBE_MASTER:.kube/config
user@<your_jenkins_server>:~/.kube/config

sudo ssh user@<your_jenkins_server> sudo chmod 777 /home/user/.kube/config

sudo ssh -i ~/.ssh/id_rsa user@<your_jenkins_server> sudo chmod 777 /home/user/.kube/config

sudo ssh -i ~/.ssh/id_rsa user@<your_jenkins_server> sudo cp /home/user/.kube/config
/var/lib/jenkins/config
```

2. Validate from Jenkins that the Kubernetes cluster is accessible.

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
```

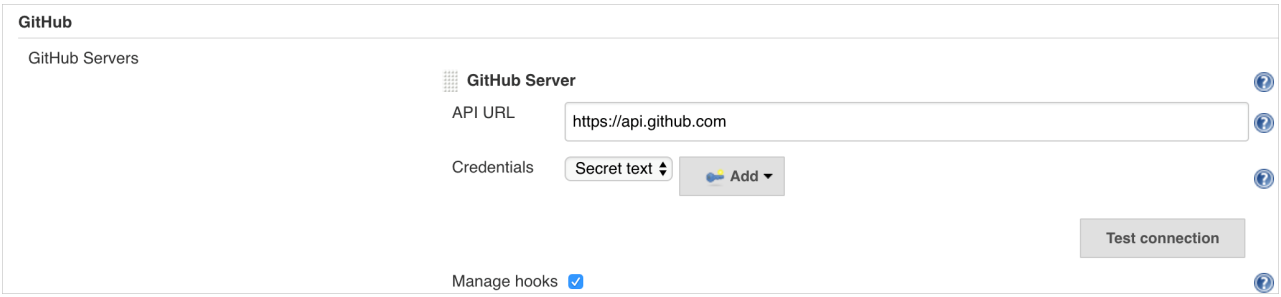
3. Deploy to the Kubernetes cluster.

> **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.



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
```

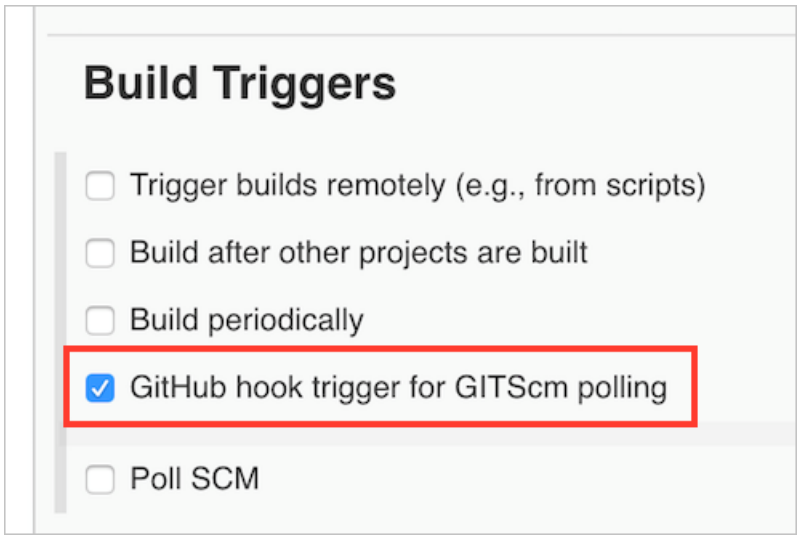
6. Add another **Build Step** of type **Execute shell** and use the following text:

```
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
```

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".



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.

Monitor an Azure Container Service cluster with DataDog

6/27/2017 • 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 `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
```

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

6/27/2017 • 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
```

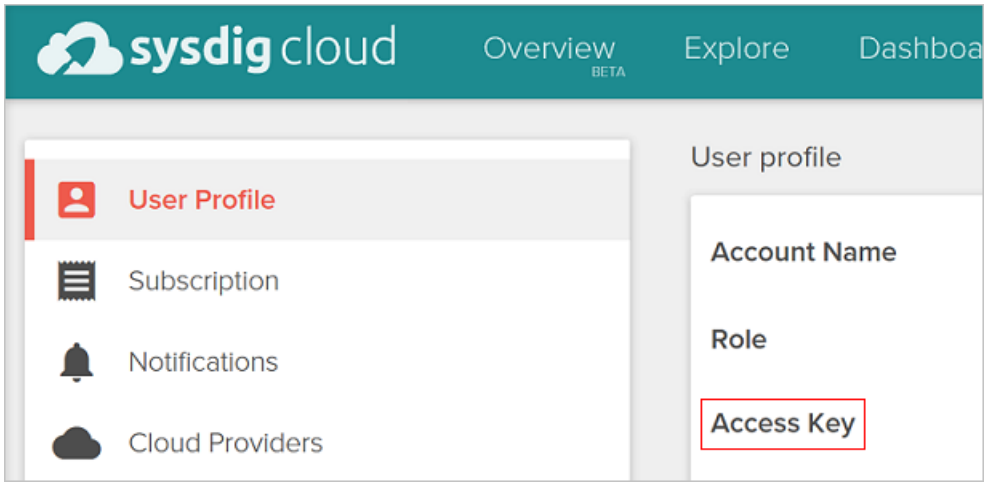
If you don't have `kubectl` installed, you can run:

```
$ 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.

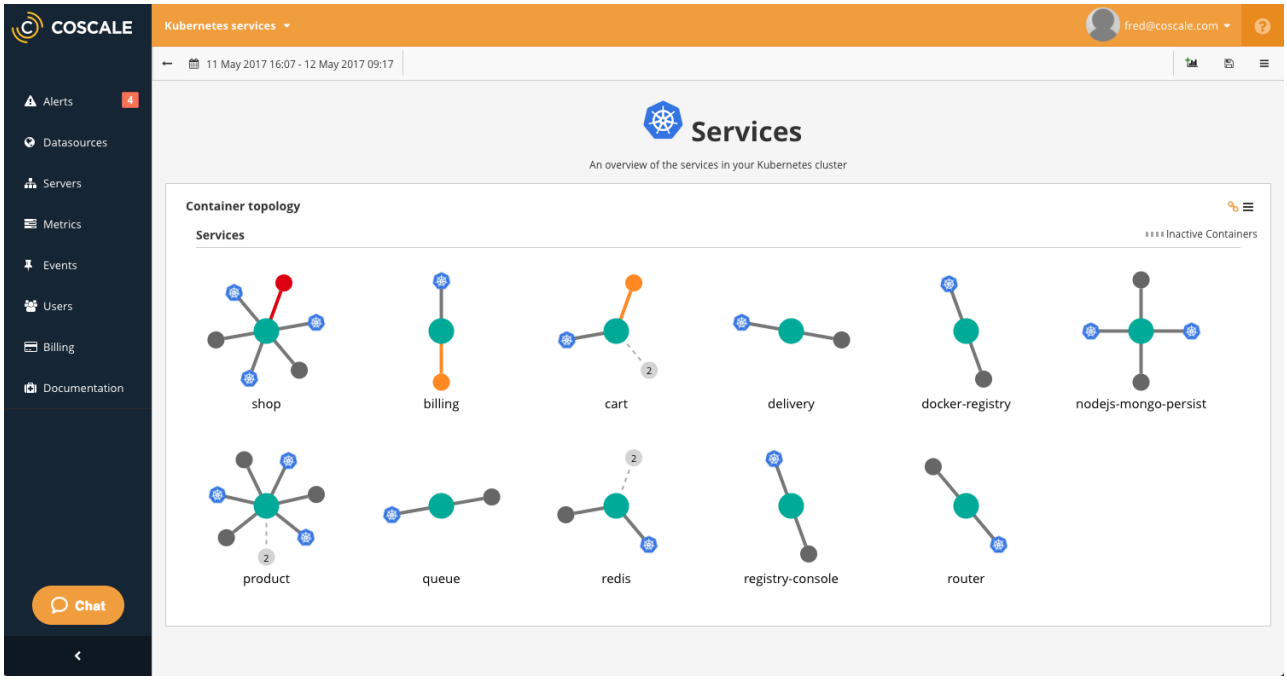
Monitor an Azure Container Service Kubernetes cluster with CoScale

6/27/2017 • 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 `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
```

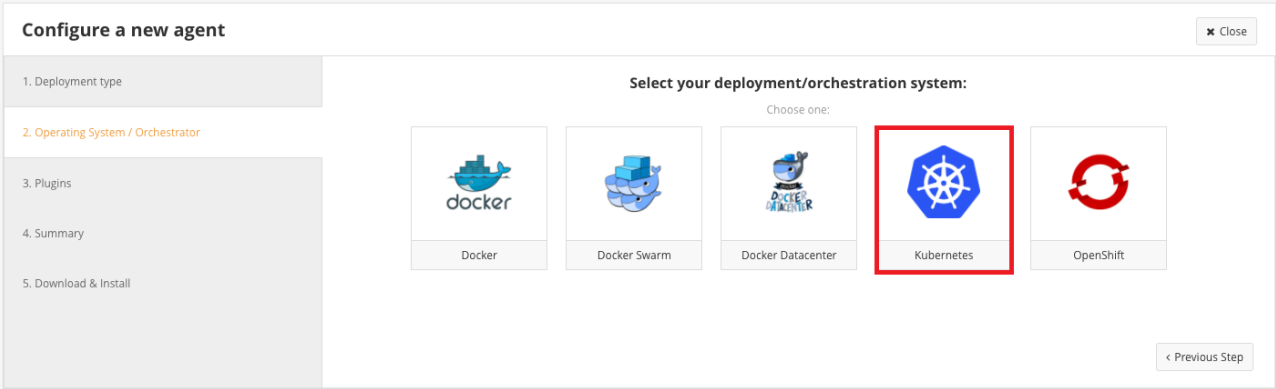
If you don't have `kubectl` installed, you can run:

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

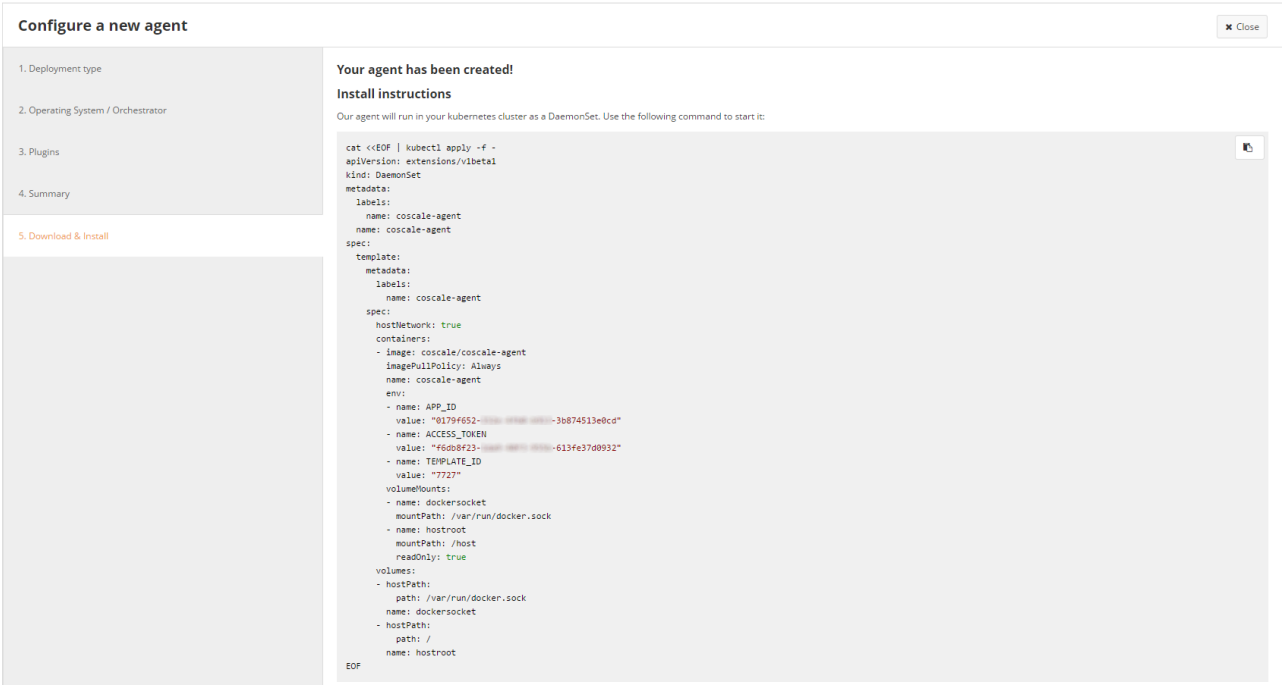
Installing the CoScale agent with a DaemonSet

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.

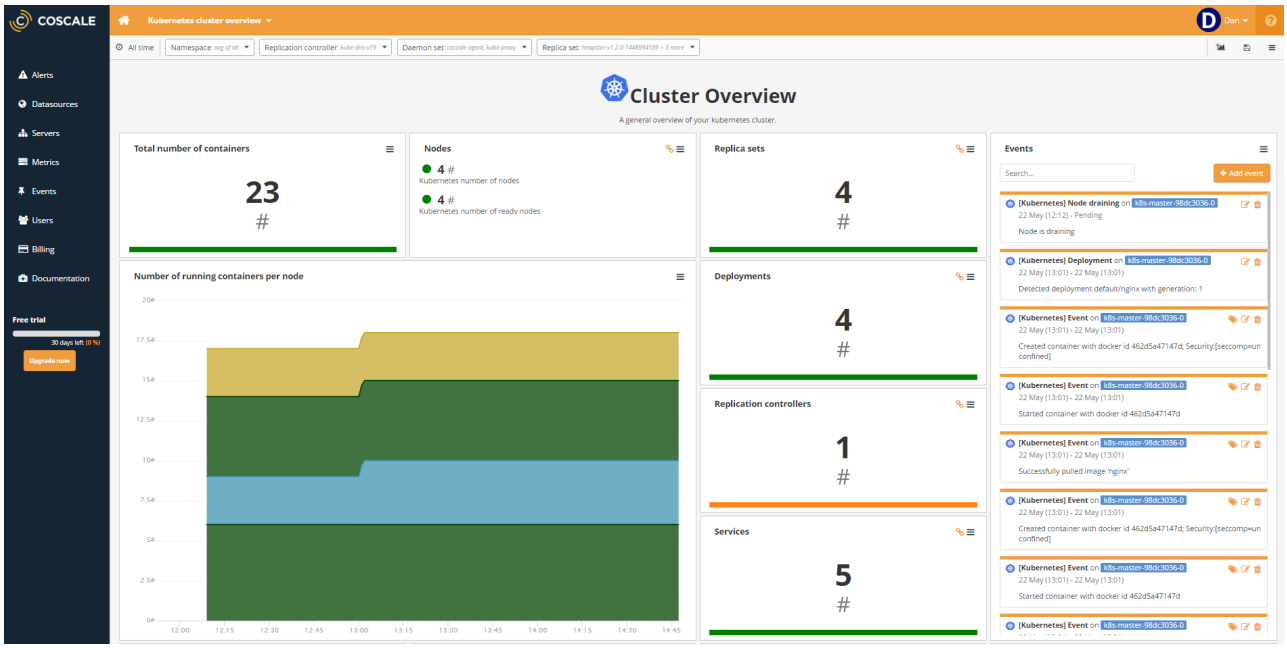
After you log in to CoScale, go to the agent page to install CoScale agents on your cluster using a DaemonSet. The CoScale UI provides guided configuration steps to create an agent and start monitoring your complete Kubernetes cluster.



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



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.

Container management with Docker Swarm

6/27/2017 • 3 min to read • Edit Online

Docker Swarm provides an environment for deploying containerized workloads across a pooled set of Docker hosts. Docker Swarm uses the native Docker API. The workflow for managing containers on a Docker Swarm is almost identical to what it would be on a single container host. This document provides simple examples of deploying containerized workloads in an Azure Container Service instance of Docker Swarm. For more in-depth documentation on Docker Swarm, see Docker Swarm on Docker.com.

> **NOTE**
>
> The Docker Swarm orchestrator in Azure Container Service uses legacy standalone Swarm. Currently, the integrated Swarm mode (in Docker 1.12 and higher) is not a supported orchestrator in Azure Container Service. If you want to deploy a Swarm mode cluster in Azure, use the open-source ACS Engine, a community-contributed quickstart template, or a Docker solution in the Azure Marketplace.

Prerequisites to the exercises in this document:

Create a Swarm cluster in Azure Container Service

Connect with the Swarm cluster in Azure Container Service

Deploy a new container

To create a new container in the Docker Swarm, use the `docker run` command (ensuring that you have opened an SSH tunnel to the masters as per the prerequisites above). This example creates a container from the `yeasy/simple-web` image:

```
user@ubuntu:~$ docker run -d -p 80:80 yeasy/simple-web

4298d397b9ab6f37e2d1978ef3c8c1537c938e98a8bf096ff00def2eab04bf72
```

After the container has been created, use `docker ps` to return information about the container. Notice here that the Swarm agent that is hosting the container is listed:

```
user@ubuntu:~$ docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
4298d397b9ab        yeasy/simple-web    "/bin/sh -c 'python i"   31 seconds ago      Up 9 seconds
10.0.0.5:80->80/tcp   swarm-agent-34A73819-1/happy_allen
```

You can now access the application that is running in this container through the public DNS name of the Swarm agent load balancer. You can find this information in the Azure portal:

By default the Load Balancer has ports 80, 8080 and 443 open. If you want to connect on another port you will need to open that port on the Azure Load Balancer for the Agent Pool.

Deploy multiple containers

As multiple containers are started, by executing 'docker run' multiple times, you can use the `docker ps` command to see which hosts the containers are running on. In the example below, three containers are spread evenly across the three Swarm agents:

```
user@ubuntu:~$ docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
11be062ff602        yeasy/simple-web    "/bin/sh -c 'python i"   11 seconds ago      Up 10 seconds
10.0.0.6:83->80/tcp   swarm-agent-34A73819-2/clever_banach
1ff421554c50        yeasy/simple-web    "/bin/sh -c 'python i"   49 seconds ago      Up 48 seconds
10.0.0.4:82->80/tcp   swarm-agent-34A73819-0/stupefied_ride
4298d397b9ab        yeasy/simple-web    "/bin/sh -c 'python i"   2 minutes ago       Up 2 minutes
10.0.0.5:80->80/tcp   swarm-agent-34A73819-1/happy_allen
```

Deploy containers by using Docker Compose

You can use Docker Compose to automate the deployment and configuration of multiple containers. To do so, ensure that a Secure Shell (SSH) tunnel has been created and that the DOCKER_HOST variable has been set (see the pre-requisites above).

Create a docker-compose.yml file on your local system. To do this, use this sample.

```
web:
  image: adtd/web:0.1
  ports:
    - "80:80"
  links:
    - rest:rest-demo-azure.marathon.mesos
rest:
  image: adtd/rest:0.1
  ports:
    - "8080:8080"
```

Run `docker-compose up -d` to start the container deployments:

```
user@ubuntu:~/compose$ docker-compose up -d
Pulling rest (adtd/rest:0.1)...
swarm-agent-3B7093B8-0: Pulling adtd/rest:0.1... : downloaded
swarm-agent-3B7093B8-2: Pulling adtd/rest:0.1... : downloaded
swarm-agent-3B7093B8-3: Pulling adtd/rest:0.1... : downloaded
Creating compose_rest_1
Pulling web (adtd/web:0.1)...
swarm-agent-3B7093B8-3: Pulling adtd/web:0.1... : downloaded
swarm-agent-3B7093B8-0: Pulling adtd/web:0.1... : downloaded
swarm-agent-3B7093B8-2: Pulling adtd/web:0.1... : downloaded
Creating compose_web_1
```

Finally, the list of running containers will be returned. This list reflects the containers that were deployed by using Docker Compose:

```
user@ubuntu:~/compose$ docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS
NAMES
caf185d221b7        adtd/web:0.1        "apache2-foreground"   2 minutes ago       Up About a minute
10.0.0.4:80->80/tcp       swarm-agent-3B7093B8-0/compose_web_1
040efc0ea937        adtd/rest:0.1       "catalina.sh run"      3 minutes ago       Up 2 minutes
10.0.0.4:8080->8080/tcp   swarm-agent-3B7093B8-0/compose_rest_1
```

Naturally, you can use `docker-compose ps` to examine only the containers defined in your `compose.yml` file.

Next steps

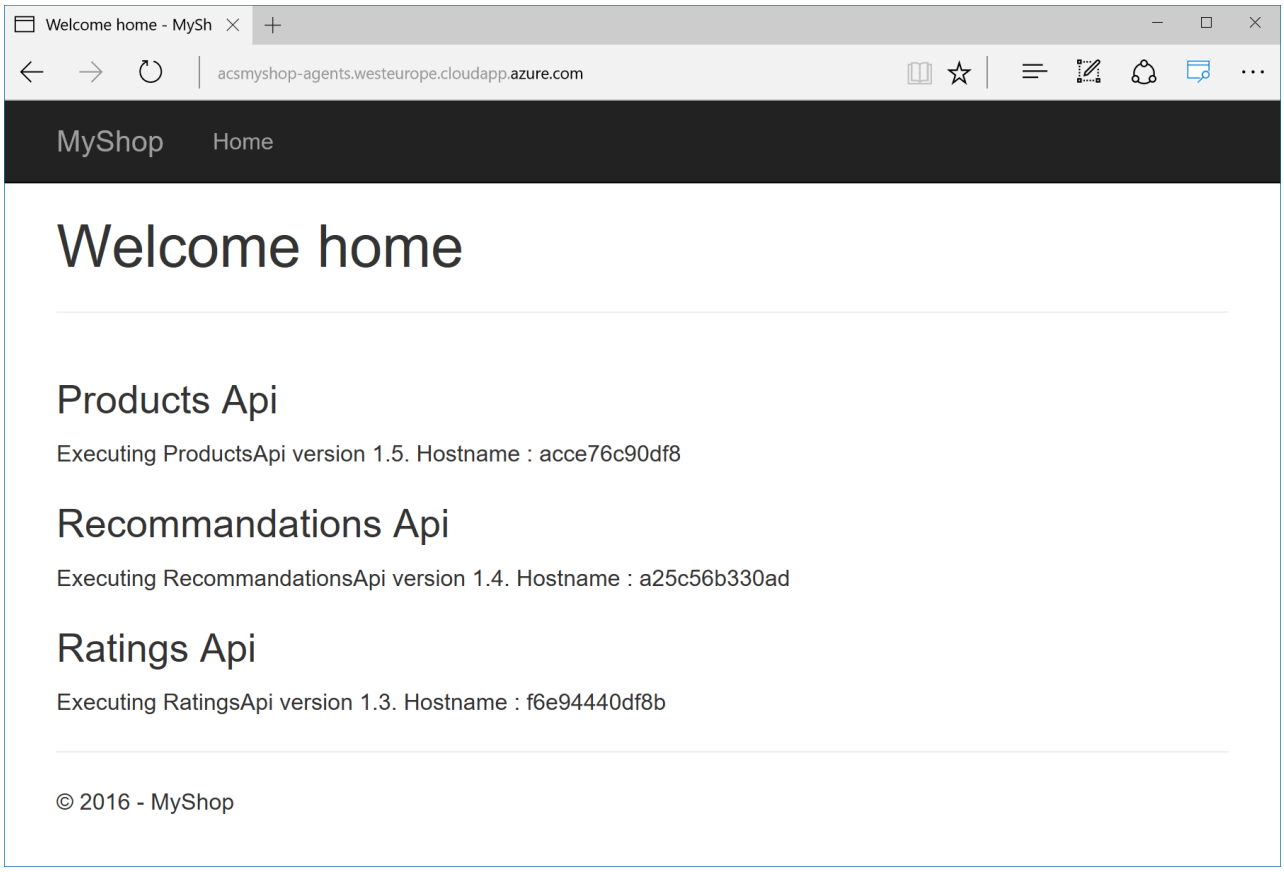
Learn more about Docker Swarm

Full CI/CD pipeline to deploy a multi-container application on Azure Container Service with Docker Swarm using Visual Studio Team Services

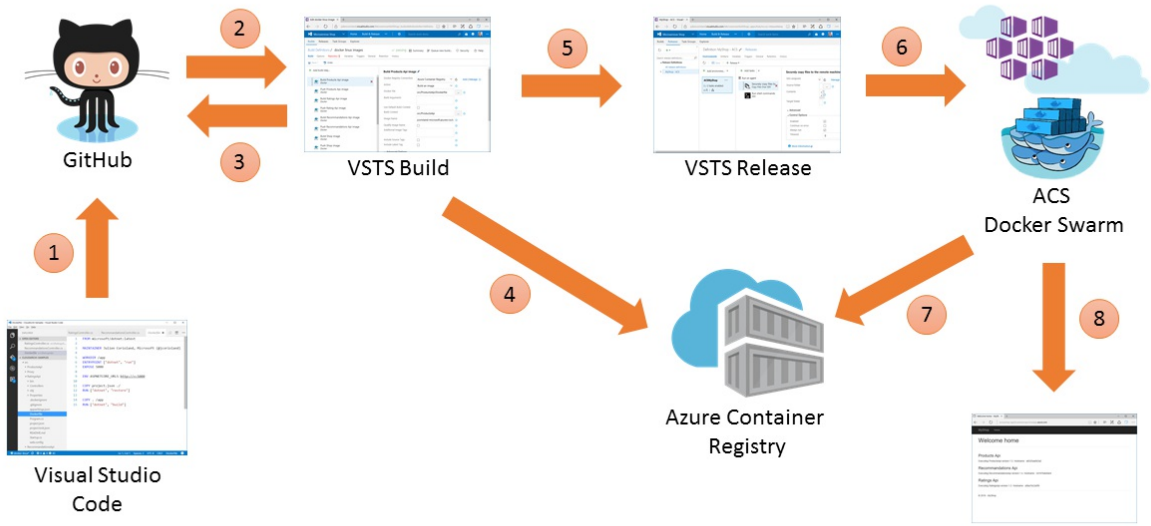
6/27/2017 • 8 min to read • Edit Online

One of the biggest challenges when developing modern applications for the cloud is being able to deliver these applications continuously. In this article, you learn how to implement a full continuous integration and deployment (CI/CD) pipeline using Azure Container Service with Docker Swarm, Azure Container Registry, and Visual Studio Team Services build and release management.

This article is based on a simple application, available on GitHub, developed with ASP.NET Core. The application is composed of four different services: three web APIs and one web front end:



The objective is to deliver this application continuously in a Docker Swarm cluster, using Visual Studio Team Services. The following figure details this continuous delivery pipeline:



Here is a brief explanation of the steps:

1. Code changes are committed to the source code repository (here, GitHub)
2. GitHub triggers a build in Visual Studio Team Services
3. Visual Studio Team Services gets the latest version of the sources and builds all the images that compose the application
4. Visual Studio Team Services pushes each image to a Docker registry created using the Azure Container Registry service
5. Visual Studio Team Services triggers a new release
6. The release runs some commands using SSH on the Azure container service cluster master node
7. Docker Swarm on the cluster pulls the latest version of the images
8. The new version of the application is deployed using Docker Compose

Prerequisites

Before starting this tutorial, you need to complete the following tasks:

- Create a Swarm cluster in Azure Container Service
- Connect with the Swarm cluster in Azure Container Service
- Create an Azure container registry
- Have a Visual Studio Team Services account and team project created
- Fork the GitHub repository to your GitHub account

> **NOTE**
> The Docker Swarm orchestrator in Azure Container Service uses legacy standalone Swarm. Currently, the integrated Swarm mode (in Docker 1.12 and higher) is not a supported orchestrator in Azure Container Service. If you want to deploy a Swarm mode cluster in Azure, use the open-source ACS Engine, a community-contributed quickstart template, or a Docker solution in the Azure Marketplace.

You also need an Ubuntu (14.04 or 16.04) machine with Docker installed. This machine is used by Visual Studio Team Services during the build and release processes. One way to create this machine is to use the image available in the Azure Marketplace.

Step 1: Configure your Visual Studio Team Services account

In this section, you configure your Visual Studio Team Services account.

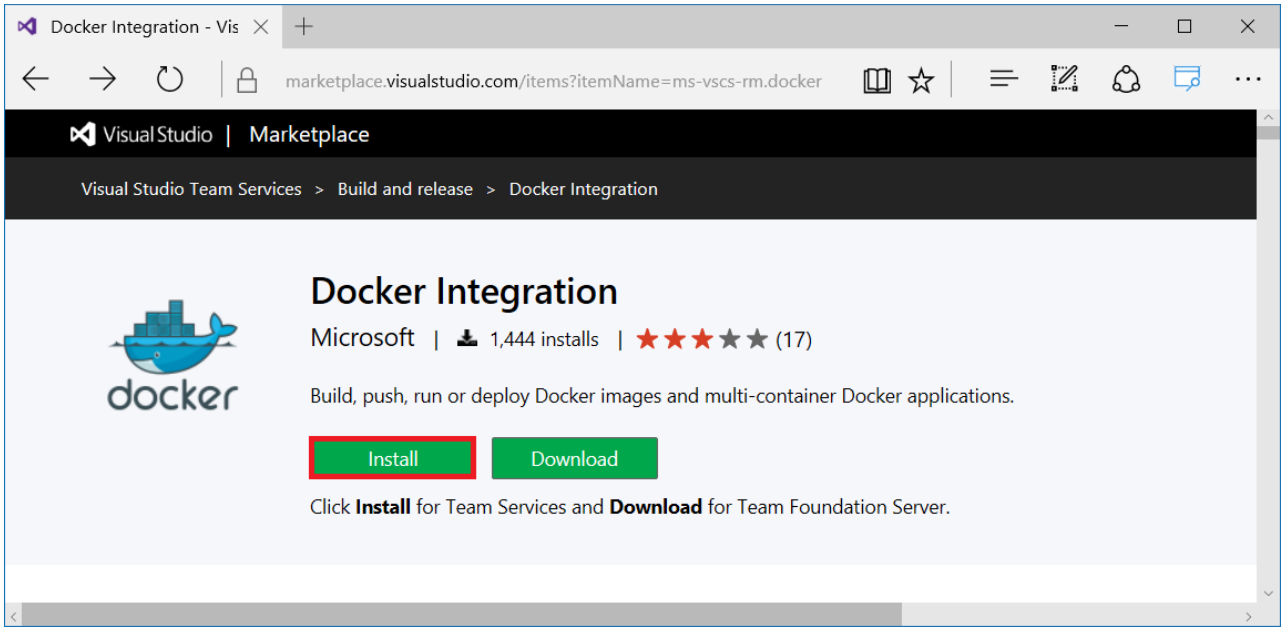
Configure a Visual Studio Team Services Linux build agent

To create Docker images and push these images into an Azure container registry from a Visual Studio Team Services build, you need to register a Linux agent. You have these installation options:

- Deploy an agent on Linux
- Use Docker to run the VSTS agent

Install the Docker Integration VSTS extension

Microsoft provides a VSTS extension to work with Docker in build and release processes. This extension is available in the VSTS Marketplace. Click **Install** to add this extension to your VSTS account:

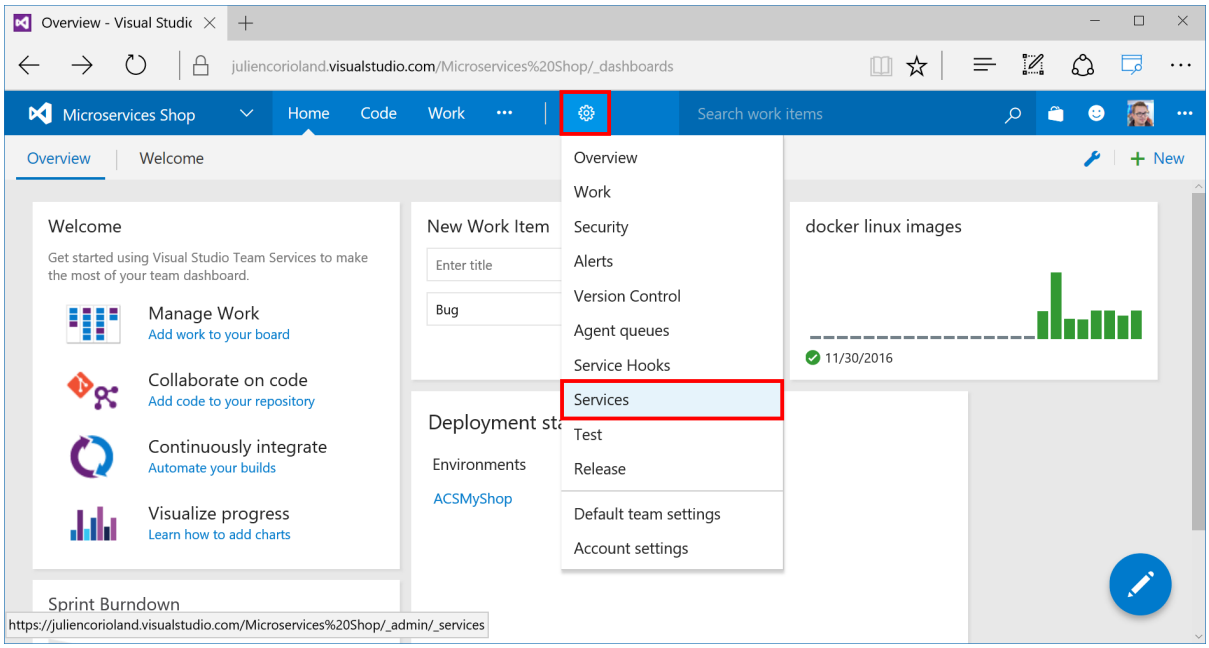


You are asked to connect to your VSTS account using your credentials.

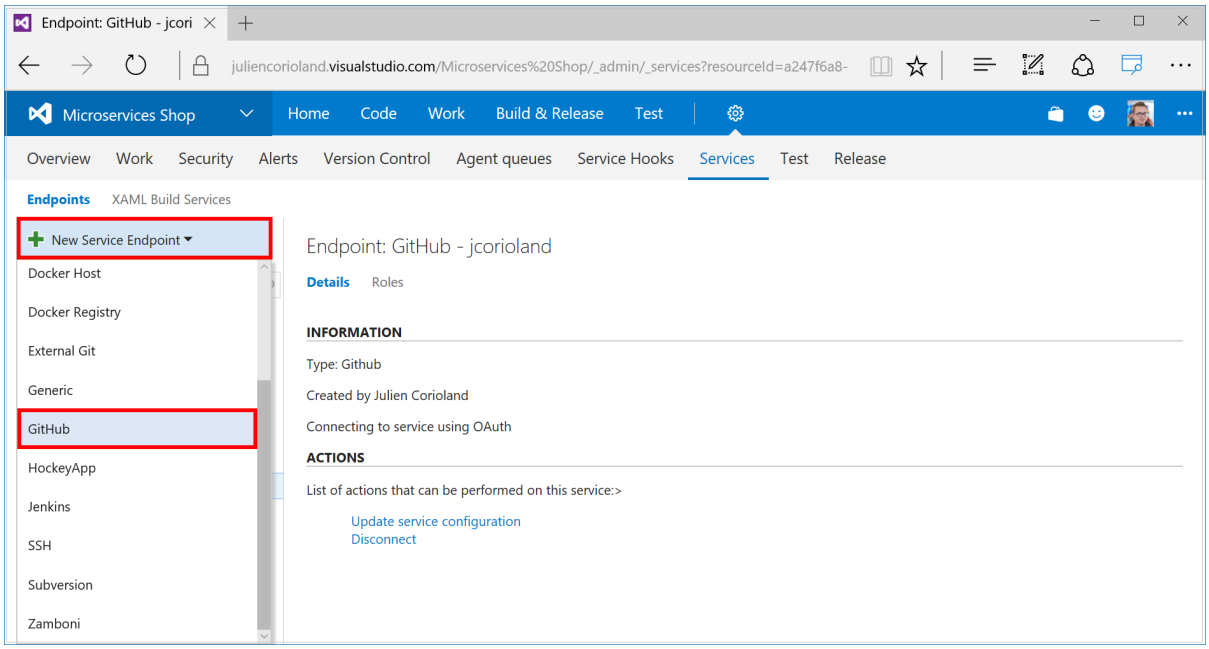
Connect Visual Studio Team Services and GitHub

Set up a connection between your VSTS project and your GitHub account.

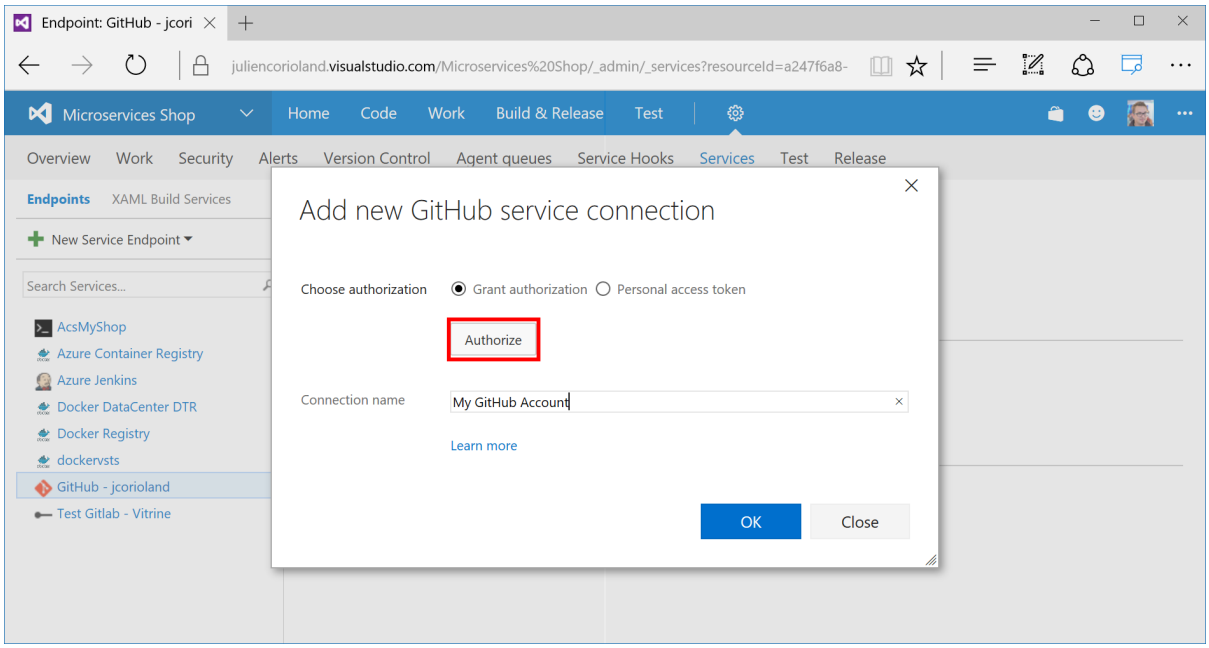
1. In your Visual Studio Team Services project, click the **Settings** icon in the toolbar, and select **Services**.



2. On the left, click **New Service Endpoint** > **GitHub**.



3. To authorize VSTS to work with your GitHub account, click **Authorize** and follow the procedure in the window that opens.

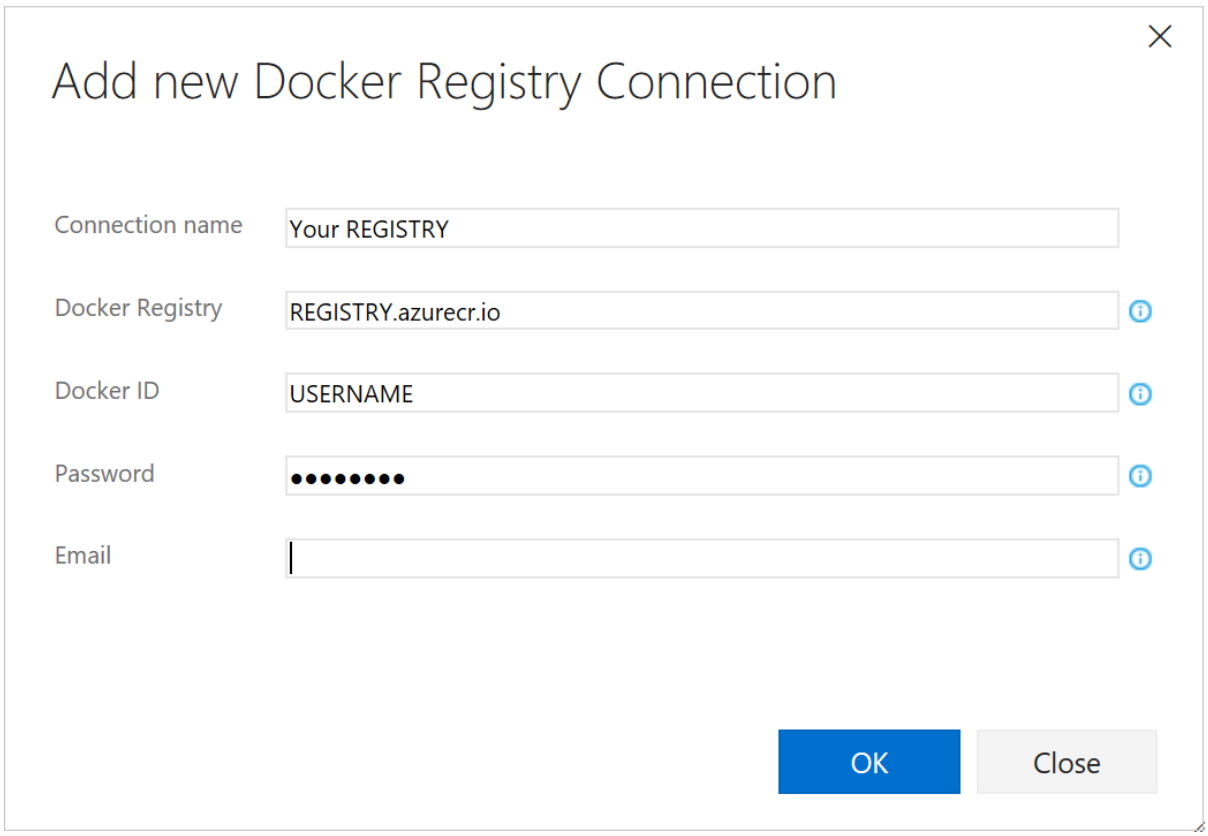


Connect VSTS to your Azure container registry and Azure Container Service cluster

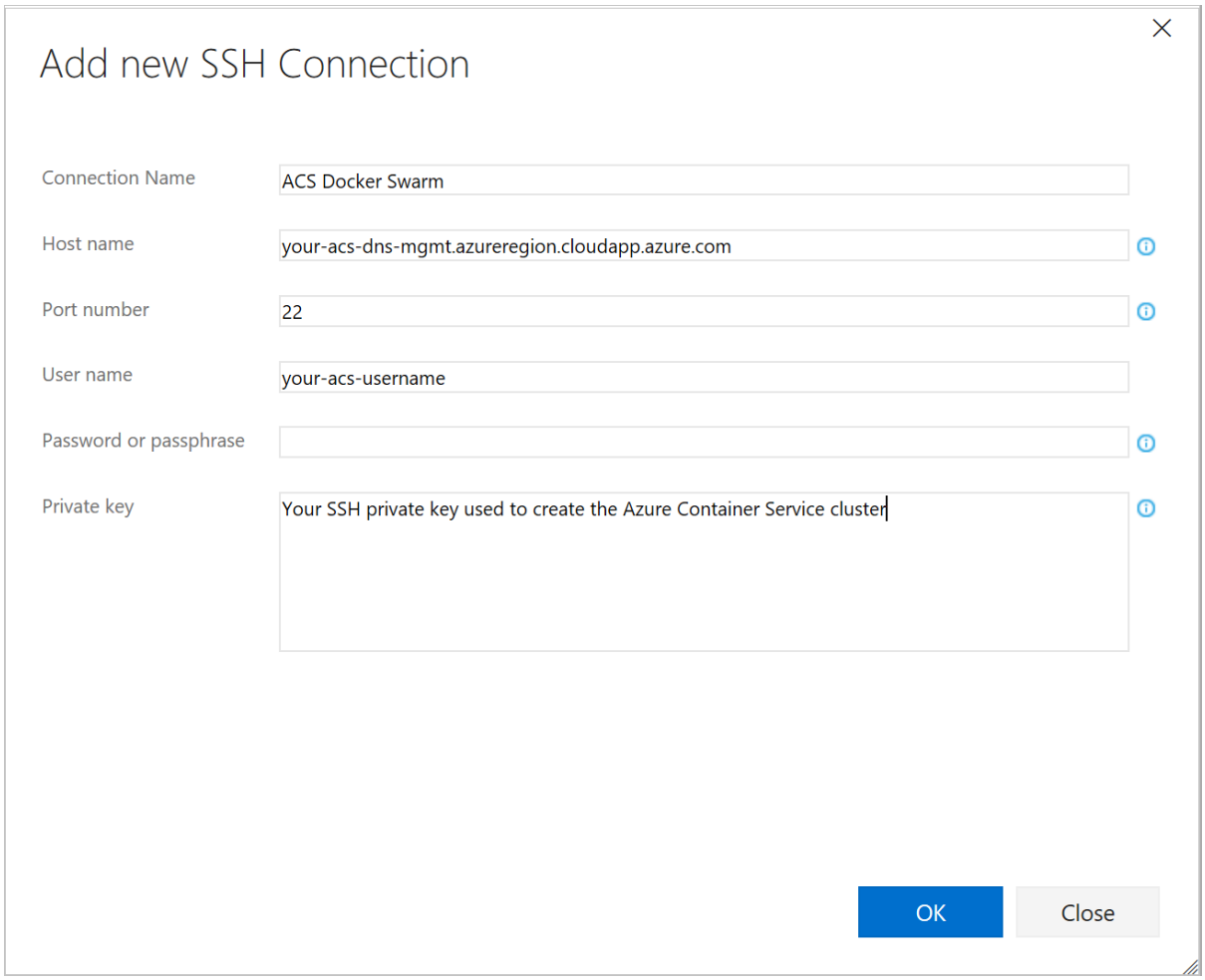
The last steps before getting into the CI/CD pipeline are to configure external connections to your container registry and your Docker Swarm cluster in Azure.

1. In the **Services** settings of your Visual Studio Team Services project, add a service endpoint of type **Docker Registry**.

2. In the popup that opens, enter the URL and the credentials of your Azure container registry.



3. For the Docker Swarm cluster, add an endpoint of type **SSH**. Then enter the SSH connection information of your Swarm cluster.



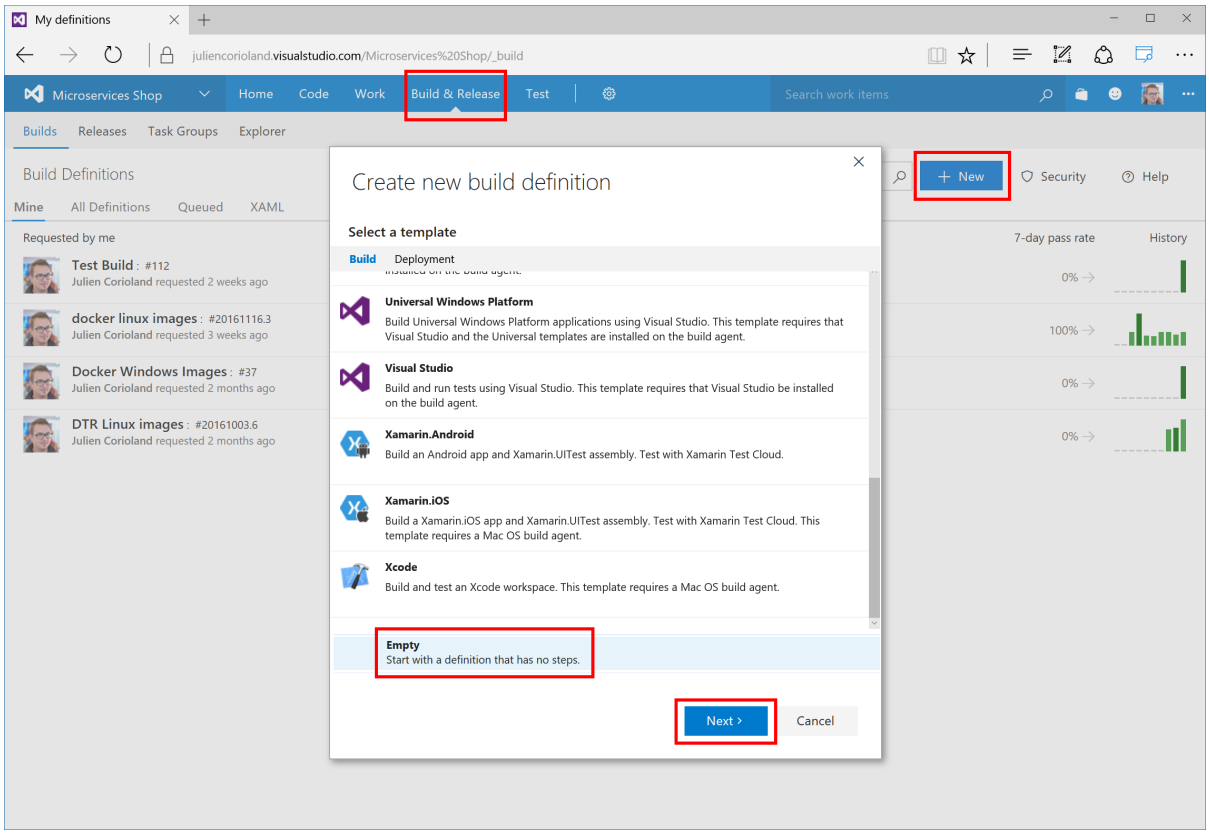
All the configuration is done now. In the next steps, you create the CI/CD pipeline that builds and deploys the application to the Docker Swarm cluster.

Step 2: Create the build definition

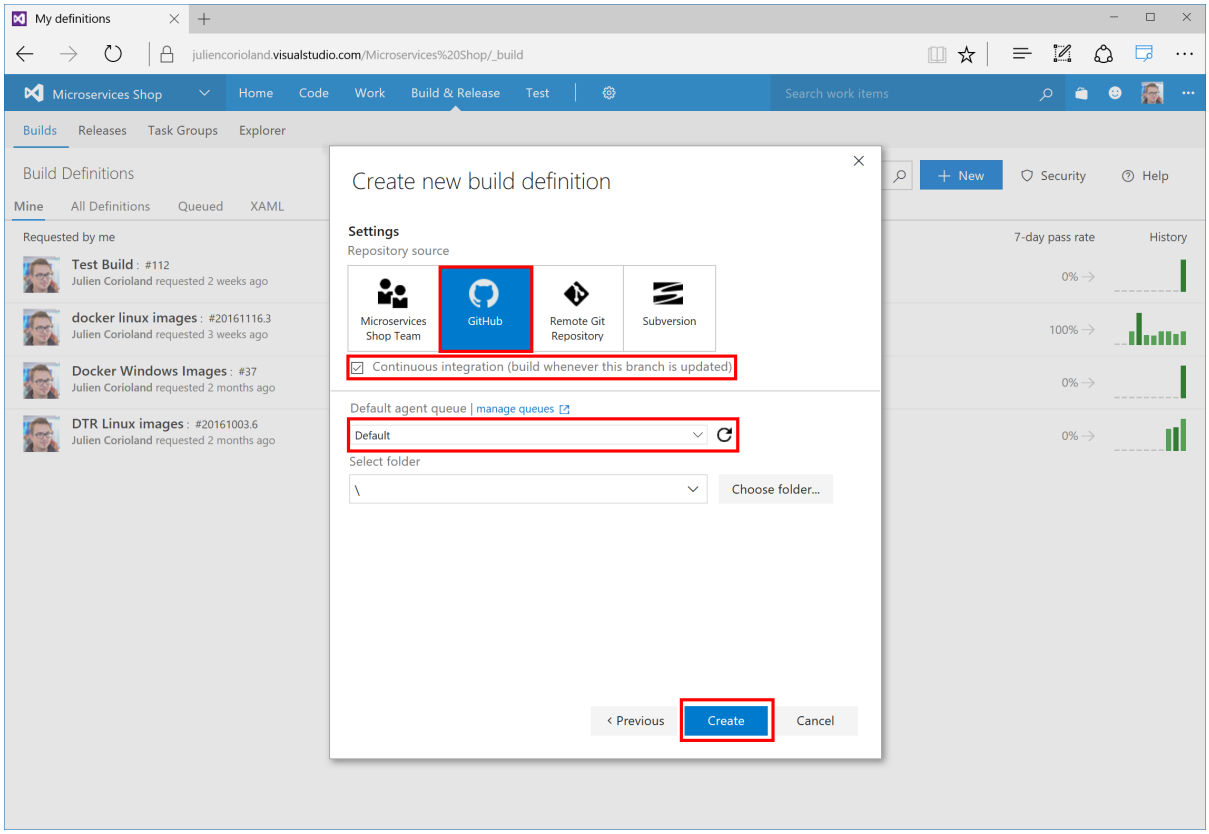
In this step, you set up a build definitionfor your VSTS project and define the build workflow for your container images

Initial definition setup

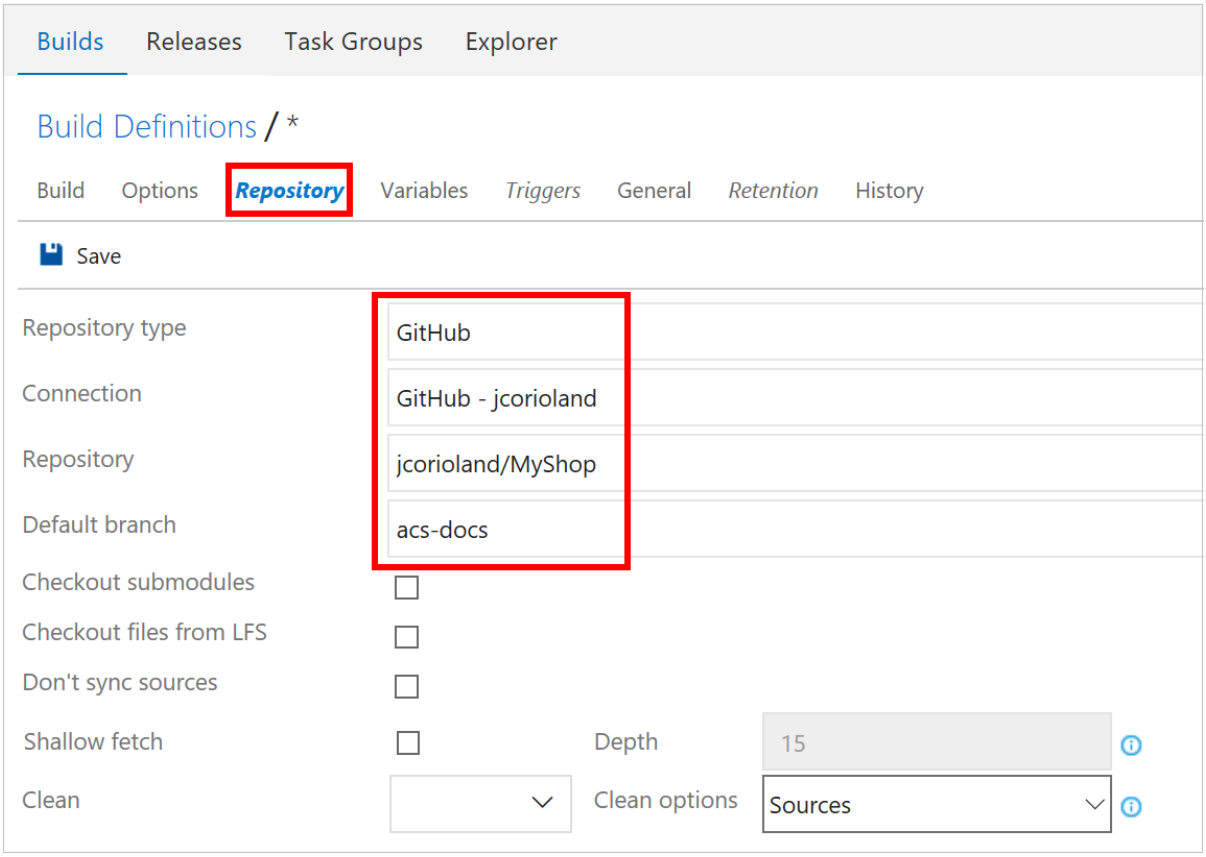
1. To create a build definition, connect to your Visual Studio Team Services project and click **Build & Release**.
2. In the **Build definitions** section, click **+ New**. Select the **Empty** template.



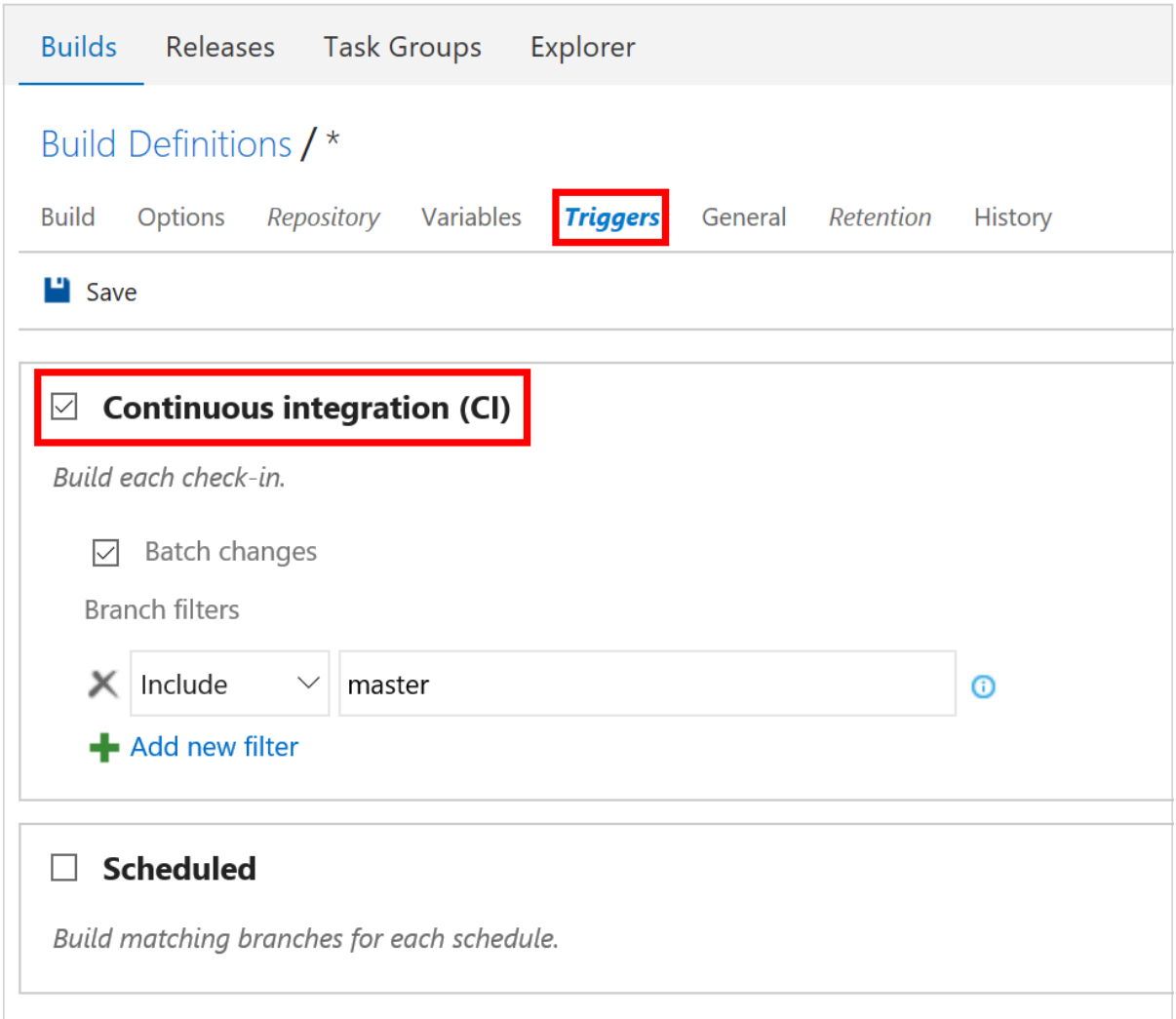
3. Configure the new build with a GitHub repository source, check **Continuous integration**, and select the agent queue where you registered your Linux agent. Click **Create** to create the build definition.



4. On the **Build Definitions** page, first open the **Repository** tab and configure the build to use the fork of the MyShop project that you created in the prerequisites. Make sure that you select *acs-docs* as the **Default branch**.



5. On the **Triggers** tab, configure the build to be triggered after each commit. Select **Continuous integration** and **Batch changes**.



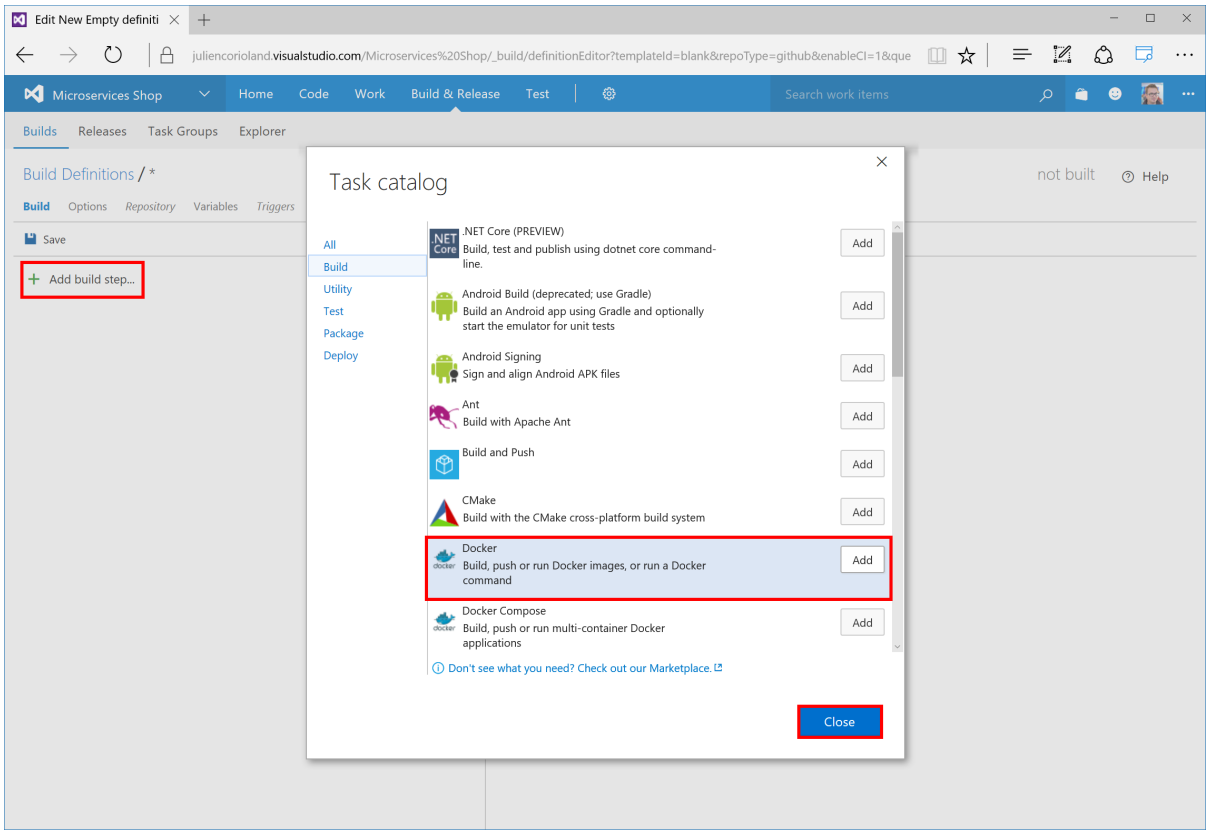
Define the build workflow

The next steps define the build workflow. There are five container images to build for the *MyShop* application. Each image is built using the Dockerfile located in the project folders:

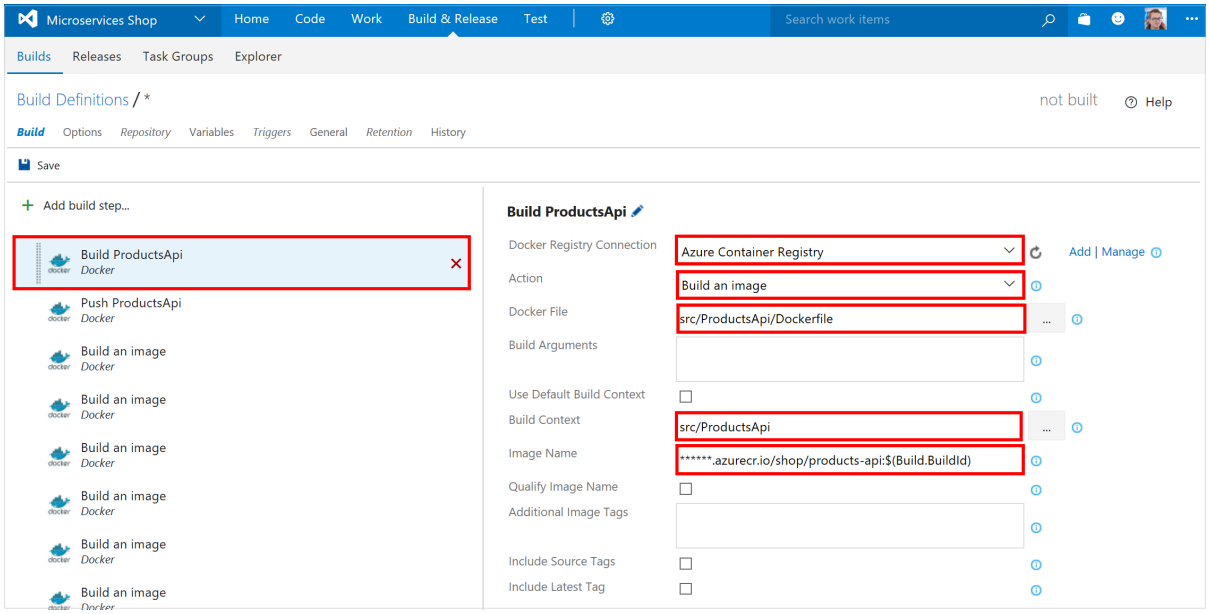
- ProductsApi
- Proxy
- RatingsApi
- RecommandationsApi
- ShopFront

You need to add two Docker steps for each image, one to build the image, and one to push the image in the Azure container registry.

1. To add a step in the build workflow, click **+ Add build step** and select **Docker**.



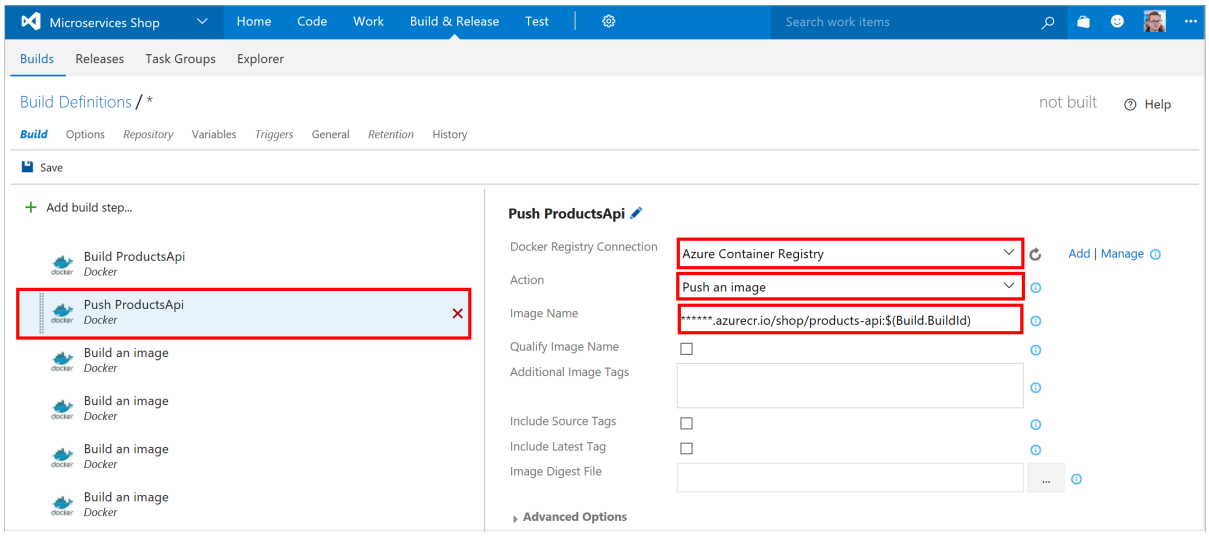
2. For each image, configure one step that uses the `docker build` command.



For the build operation, select your Azure container registry, the **Build an image** action, and the Dockerfile that defines each image. Set the **Build context** as the Dockerfile root directory, and define the **Image Name**.

As shown on the preceding screen, start the image name with the URI of your Azure container registry. (You can also use a build variable to parameterize the tag of the image, such as the build identifier in this example.)

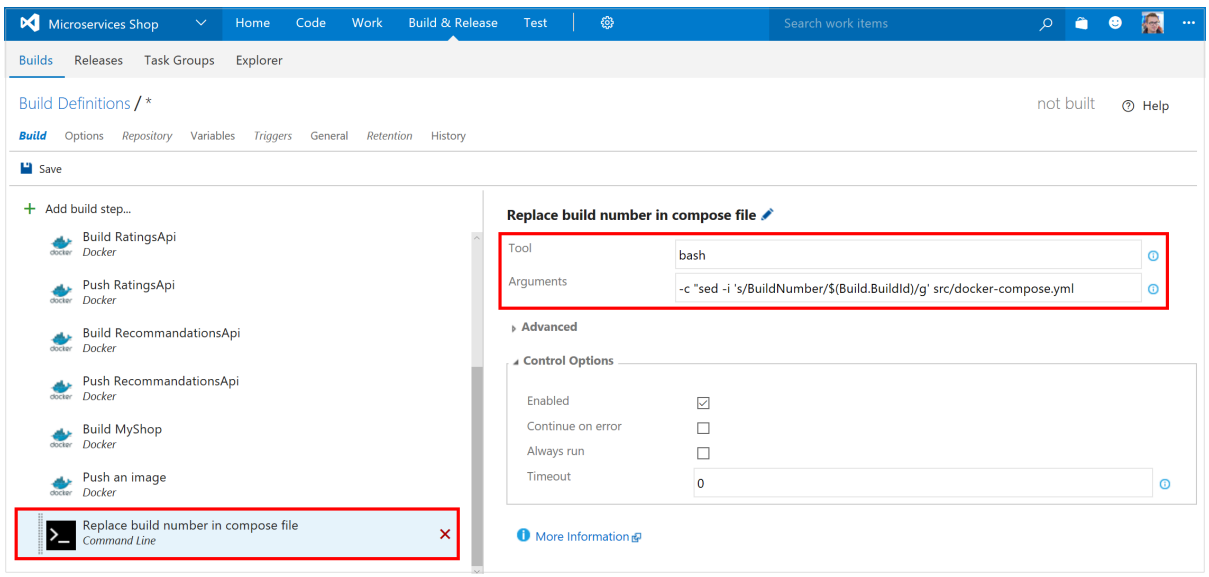
3. For each image, configure a second step that uses the `docker push` command.



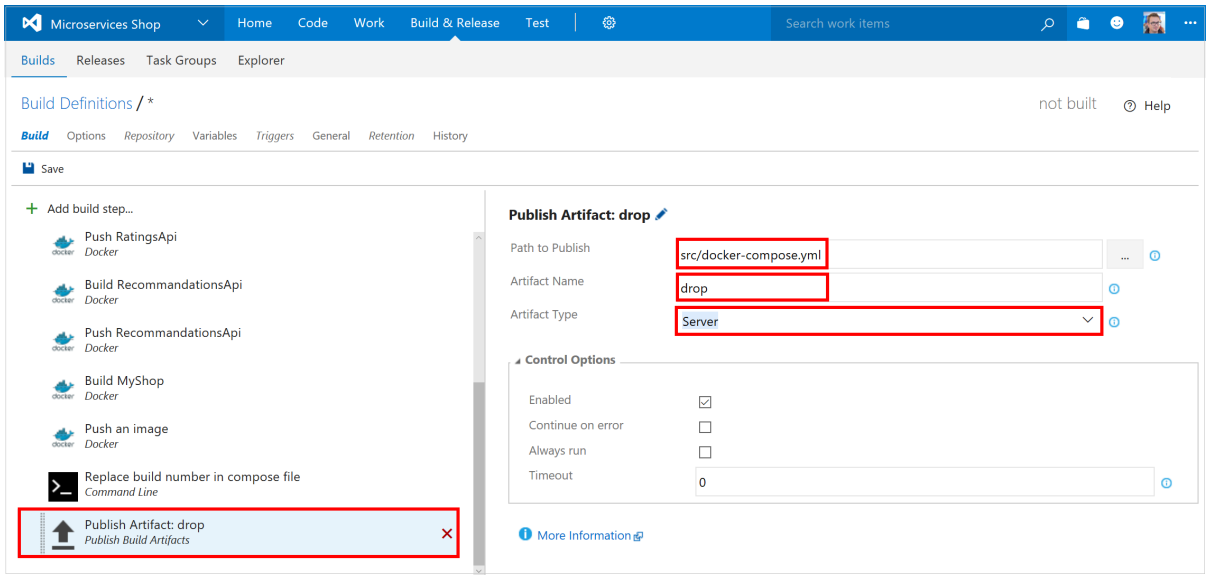
For the push operation, select your Azure container registry, the **Push an image** action, and enter the **Image Name** that is built in the previous step.

4. After you configure the build and push steps for each of the five images, add two more steps in the build workflow.

a. A command-line task that uses a bash script to replace the *BuildNumber* occurrence in the docker-compose.yml file with the current build Id. See the following screen for details.



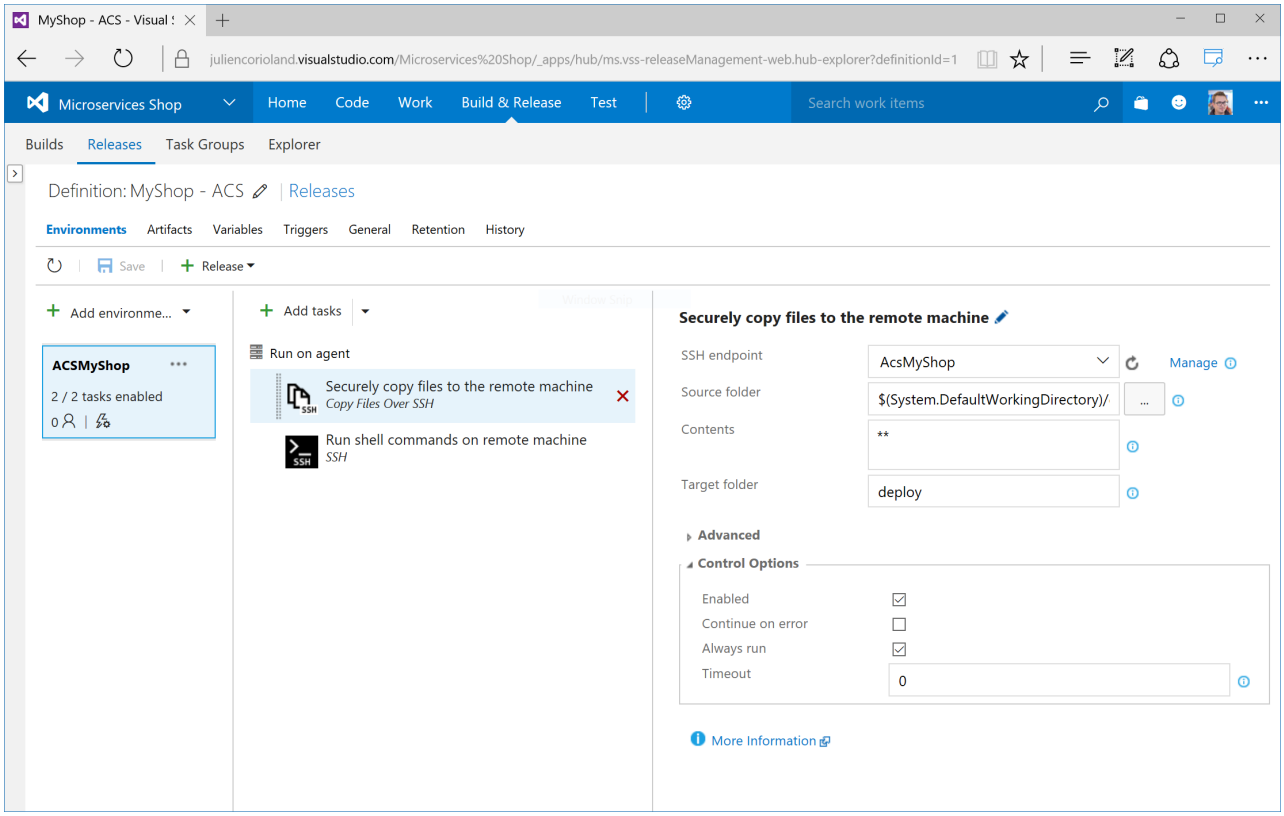
b. A task that drops the updated Compose file as a build artifact so it can be used in the release. See the following screen for details.



5. Click **Save** and name your build definition.

Step 3: Create the release definition

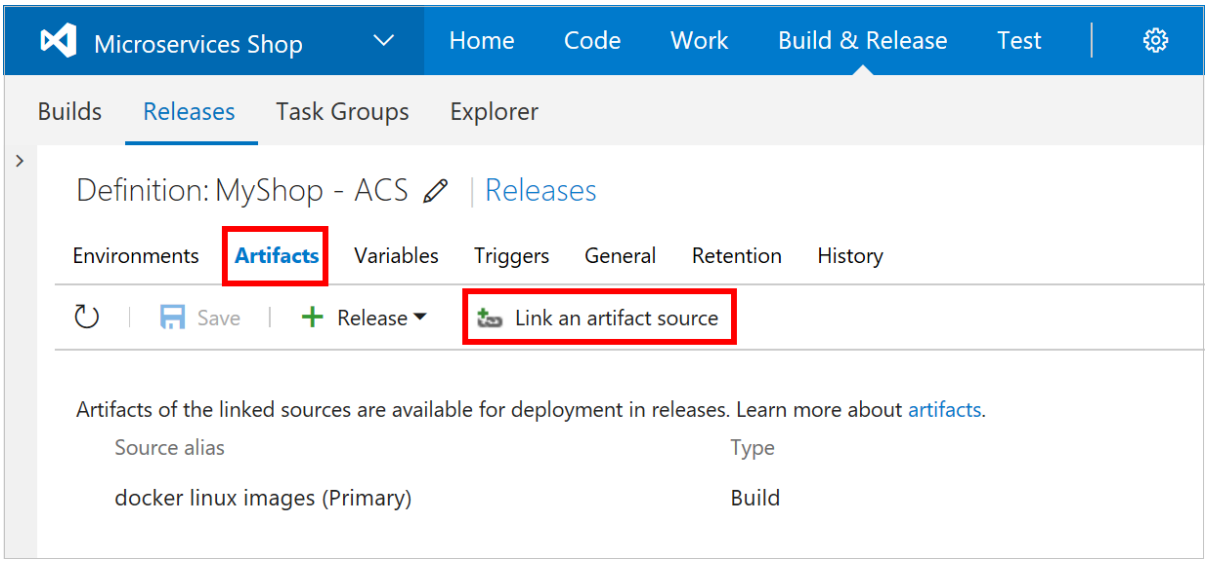
Visual Studio Team Services allows you to manage releases across environments. You can enable continuous deployment to make sure that your application is deployed on your different environments (such as dev, test, pre-production, and production) in a smooth way. You can create a new environment that represents your Azure Container Service Docker Swarm cluster.



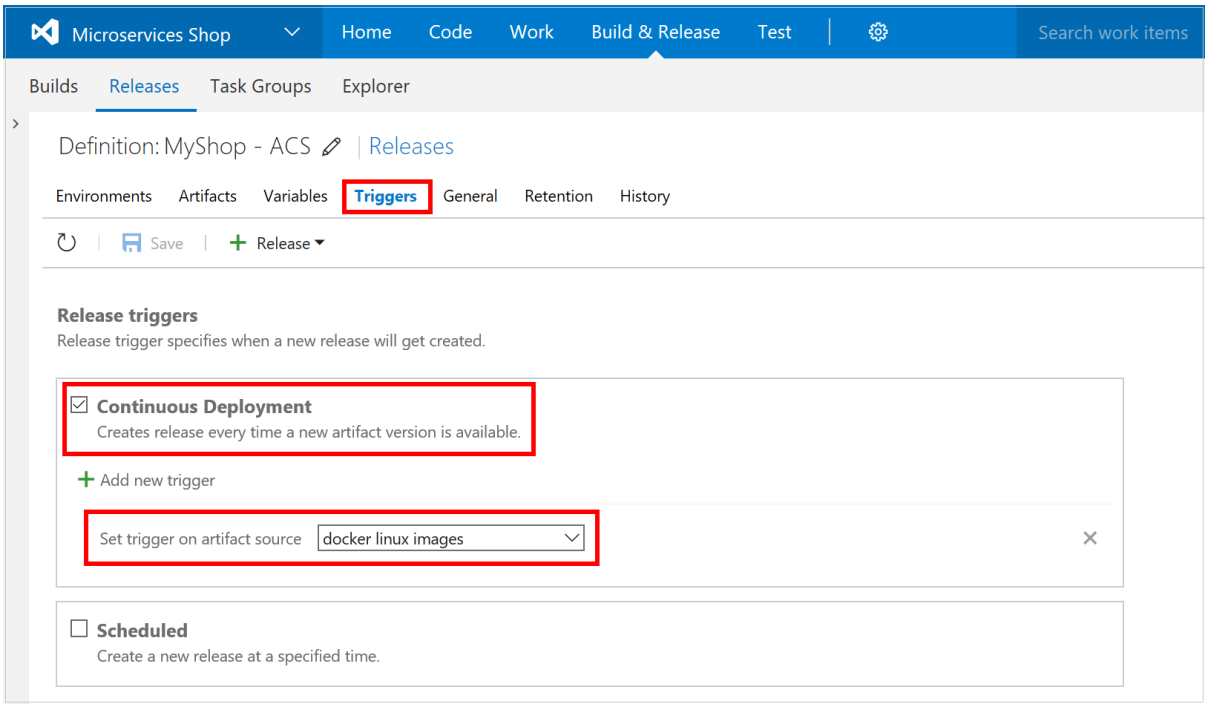
Initial release setup

1. To create a release definition, click **Releases** > **+ Release**

2. To configure the artifact source, Click **Artifacts** > **Link an artifact source**. Here, link this new release definition to the build that you defined in the previous step. By doing this, the docker-compose.yml file is available in the release process.



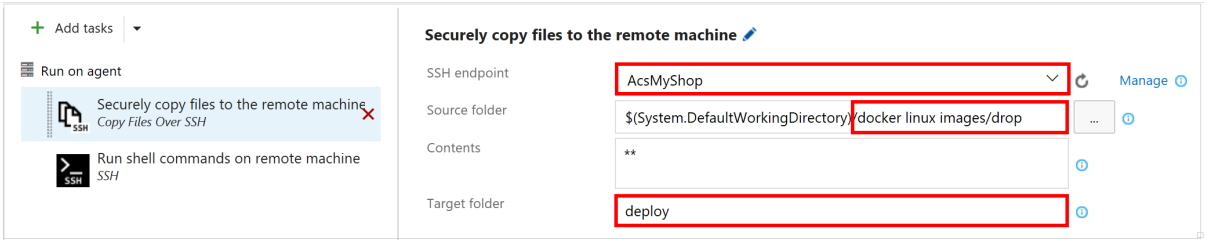
3. To configure the release trigger, click **Triggers** and select **Continuous Deployment**. Set the trigger on the same artifact source. This setting ensures that a new release starts as soon as the build completes successfully.



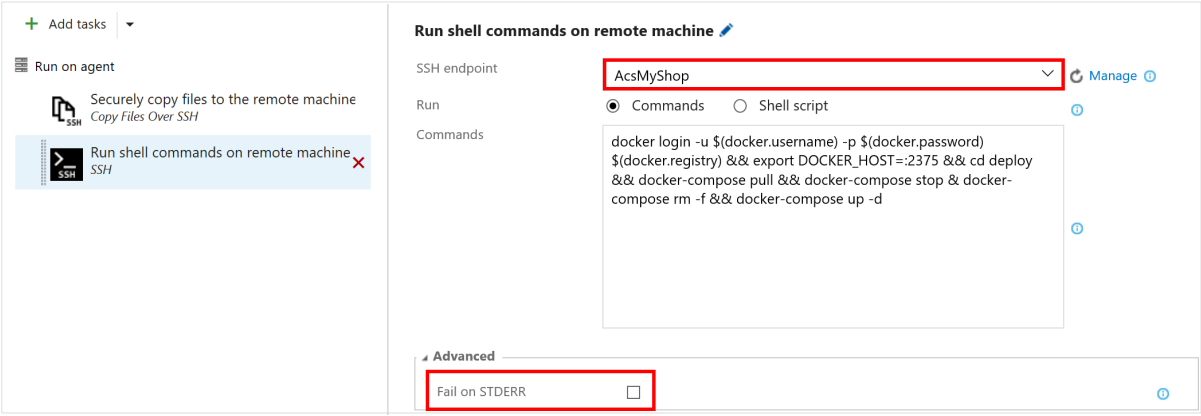
Define the release workflow

The release workflow is composed of two tasks that you add.

1. Configure a task to securely copy the compose file to a *deploy* folder on the Docker Swarm master node, using the SSH connection you configured previously. See the following screen for details.



2. Configure a second task to execute a bash command to run `docker` and `docker-compose` commands on the master node. See the following screen for details.



The command executed on the master use the Docker CLI and the Docker-Compose CLI to do the following tasks:

- Login to the Azure container registry (it uses three build variab`les that are defined in the **Variables** tab)
- Define the **DOCKER_HOST** variable to work with the Swarm endpoint (:2375)
- Navigate to the *deploy* folder that was created by the preceding secure copy task and that contains the docker-compose.yml file
- Execute `docker-compose` commands that pull the new images, stop the services, remove the services, and create the containers.

> **IMPORTANT**
>
> As shown on the preceding screen, leave the **Fail on STDERR** checkbox unchecked. This is an important setting, because `docker-compose` prints several diagnostic messages, such as containers are stopping or being deleted, on the standard error output. If you check the checkbox, Visual Studio Team Services reports that errors occurred during the release, even if all goes well.

3. Save this new release definition.

> **NOTE**
>
> This deployment includes some downtime because we are stopping the old services and running the new one. It is possible to avoid this by doing a blue-green deployment.

Step 4. Test the CI/CD pipeline

Now that you are done with the configuration, it's time to test this new CI/CD pipeline. The easiest way to test it is to update the source code and commit the changes into your GitHub repository. A few seconds after you push the code, you will see a new build running in Visual Studio Team Services. Once completed successfully, a new release will be triggered and will deploy the new version of the application on the Azure Container Service cluster.

Next Steps

- For more information about CI/CD with Visual Studio Team Services, see the VSTS Build overview.

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

6/27/2017 • 7 min to read • Edit Online

The **Spring Framework** is an open-source solution which helps Java developers create enterprise-level applications. One of the more-popular projects which 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 which helps developers automate the deployment, scaling, and management of their applications which 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:

- 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. Optional Step: If you want to run the embedded Tomcat server to run on port 80 instead of the default port of 8080, (for example if you are going to be testing your Spring Boot project locally), you can configure the port by using the following steps:

   a. Change directory to the resources directory; for example:

```
cd src/main/resources
```

   b. Open the *application.yml* file in a text editor.

   c. Modify the **server:** setting so that the server will run on port 80; for example:

```
server:
   port: 80
```

   d. Save and close the *application.yml* file.

   e. Change directory back to the root folder for the completed project; for example:

```
cd ../../..
```

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

```
mvn package
```

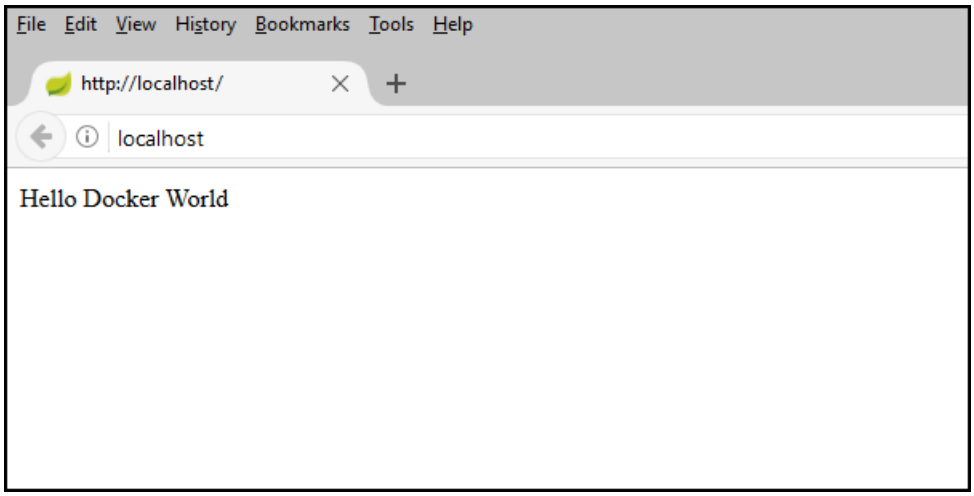
6. Once the web app has been created, change directory to the `target` directory where the JAR file is located and start the web app; for example:

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

7. 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
```

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



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

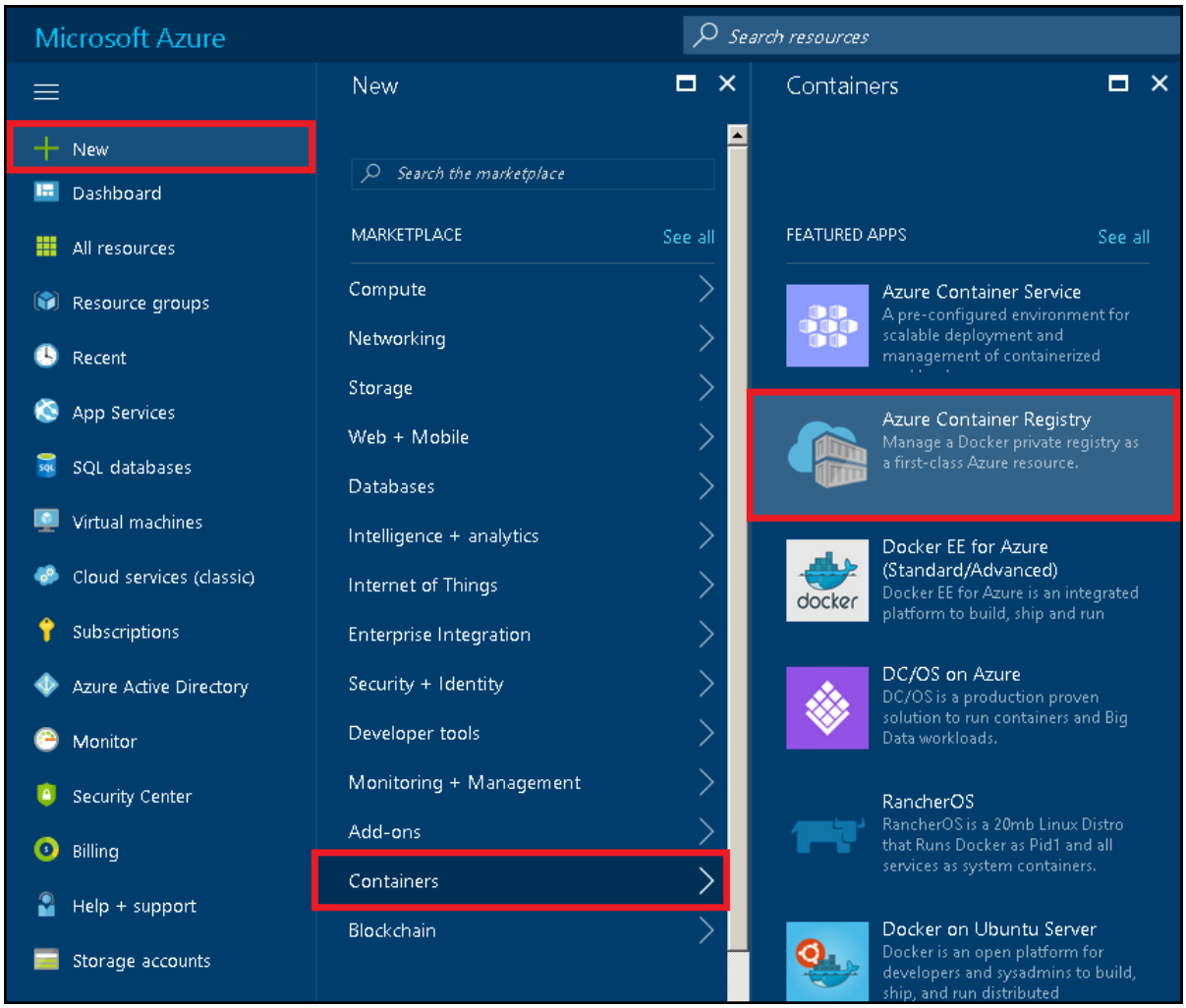
The following steps walk you through using the Azure portal to create an Azure Container Registry.

> **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.

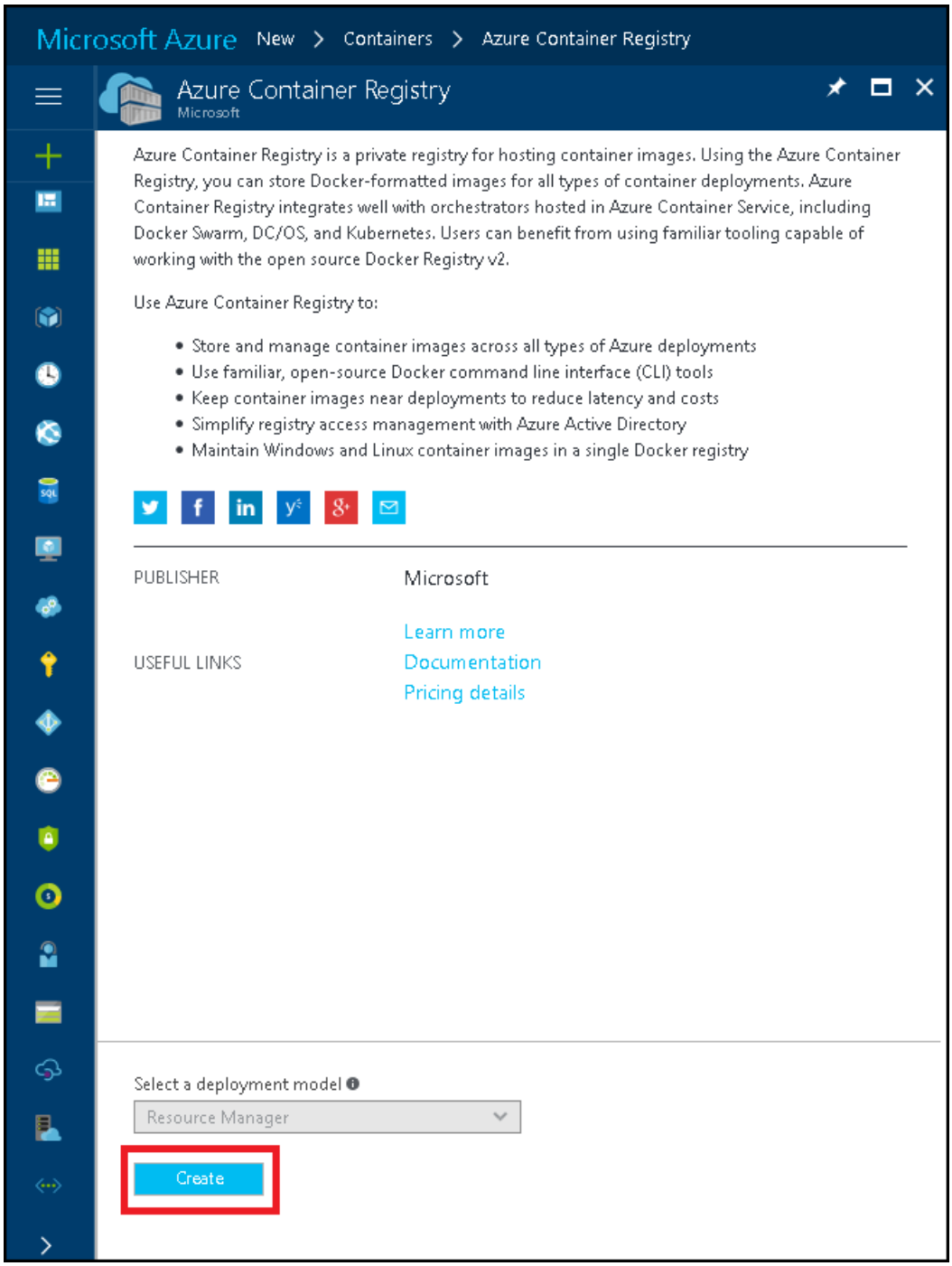
1. Browse to the Azure portal and log in.

   Once you have logged into 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.

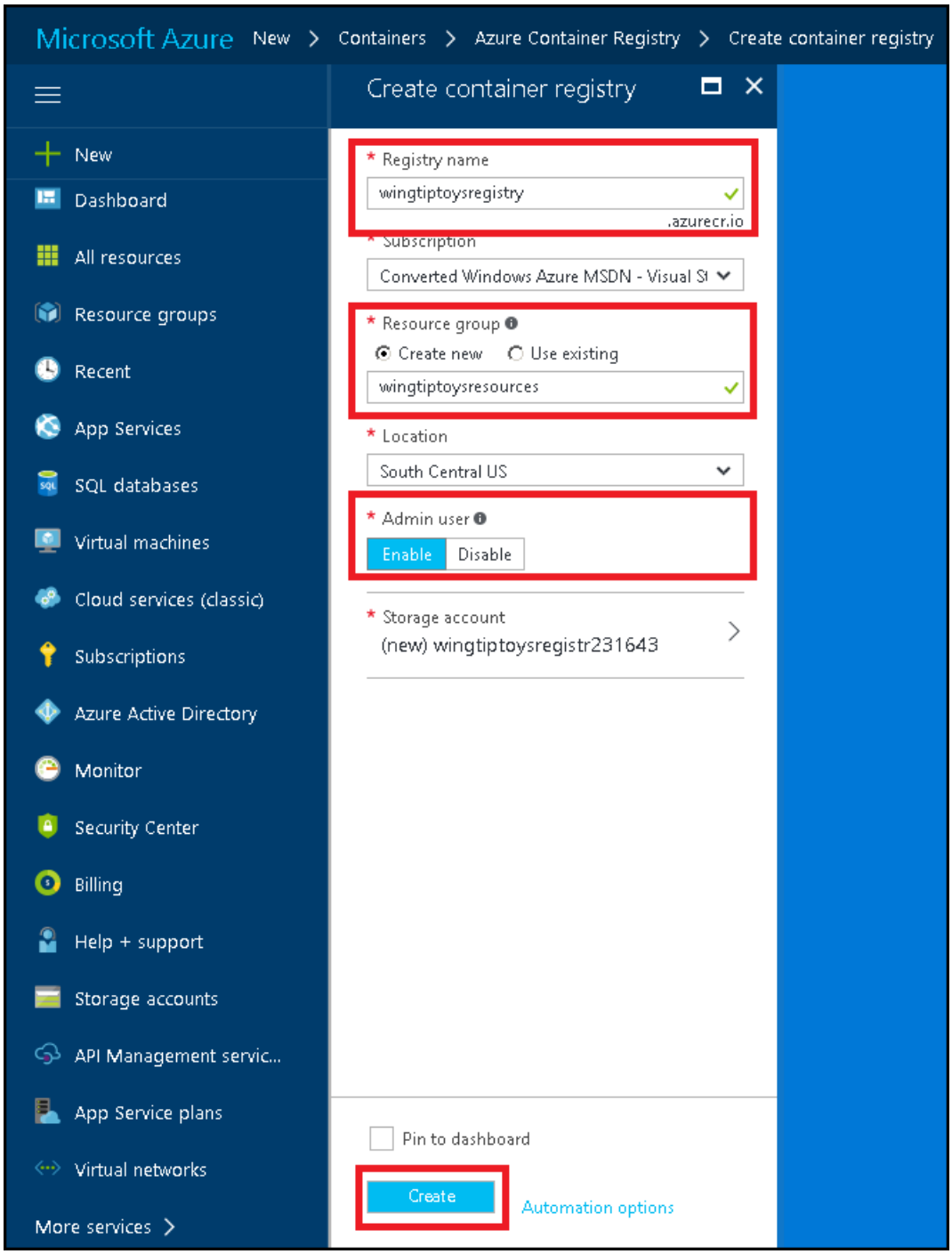
2. Click the menu icon for **+ New**, then click **Containers**, and then click **Azure Container Registry**.



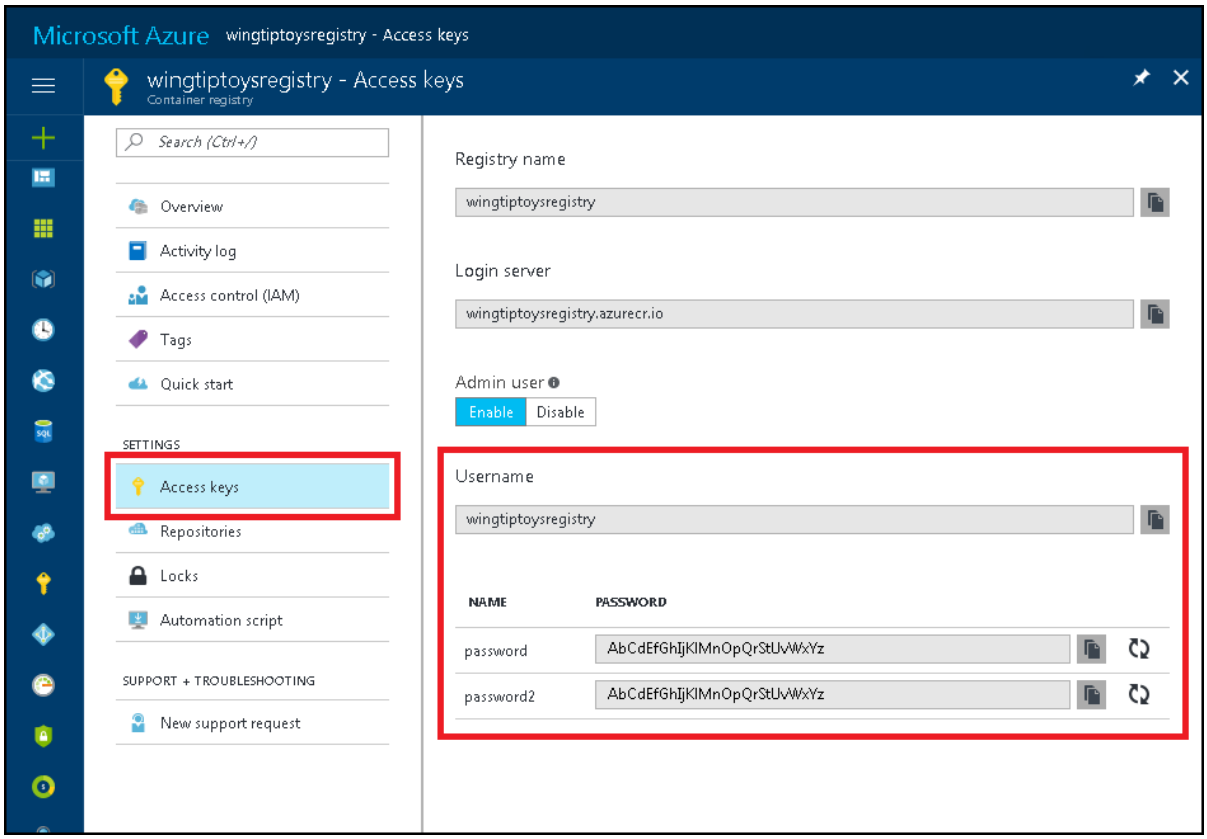
3. When the information page for the Azure Container Registry template is displayed, click **Create**.



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



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

3. Navigate to the completed project directory for your Spring Boot application, (e.g. "*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>wingtiptoysregistry.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>wingtiptoysregistry</serverId>
      <registryUrl>https://wingtiptoysregistry.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 log into Azure from the Docker command line; for example:

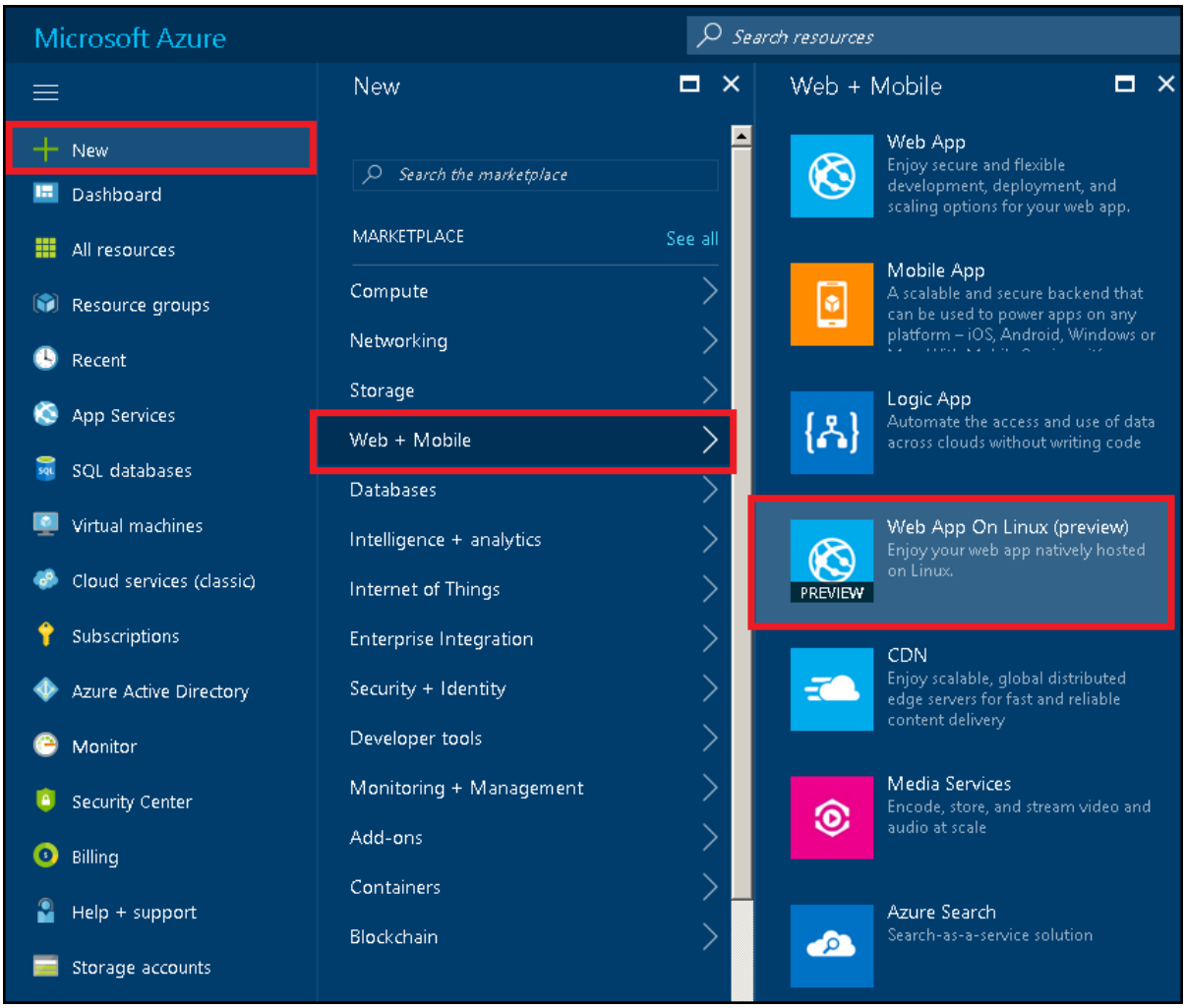
```
docker login -u wingtiptoysregistry -p "AbCdEfGhIjKlMnOpQrStUvWxYz" wingtiptoysregistry.azurecr.io
```

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

```
docker push wingtiptoysregistry.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 log 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** blade is displayed, enter the following information:

   a. Enter a unique name for the **App name**; for example: "*wingtiptoyslinux*."

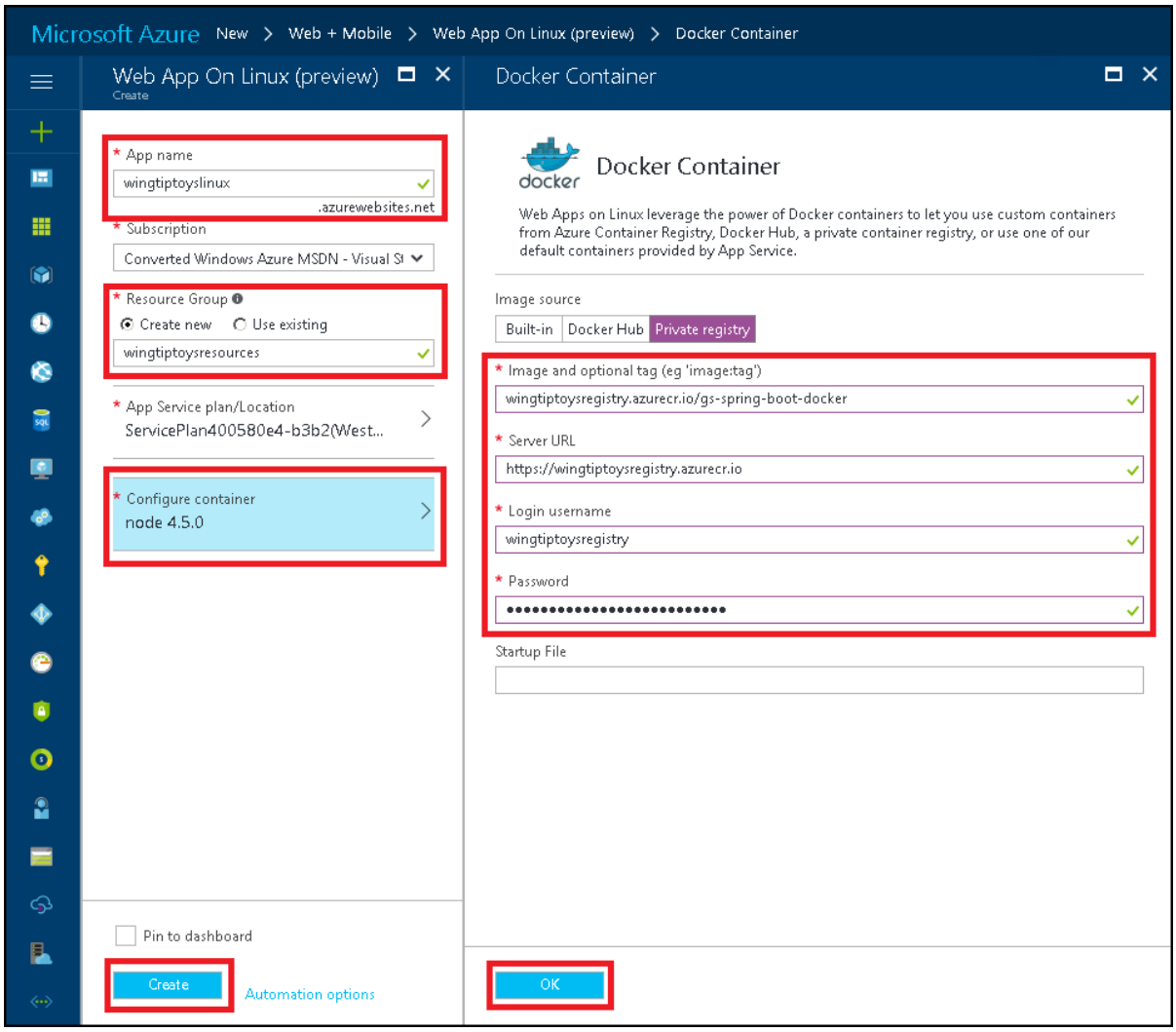
   b. Choose your **Subscription** from the drop-down list.

   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: "*wingtiptoysregistry.azurecr.io/gs-spring-boot-docker:latest*"
   - **Server URL**: Specify your registry URL from earlier; for example: "*https://wingtiptoysregistry.azurecr.io*"
   - **Login username** and **Password**: Specify your login credentials from your **Access Keys** which you used in previous steps.

   e. Once you have entered all of the above information, click **OK**.

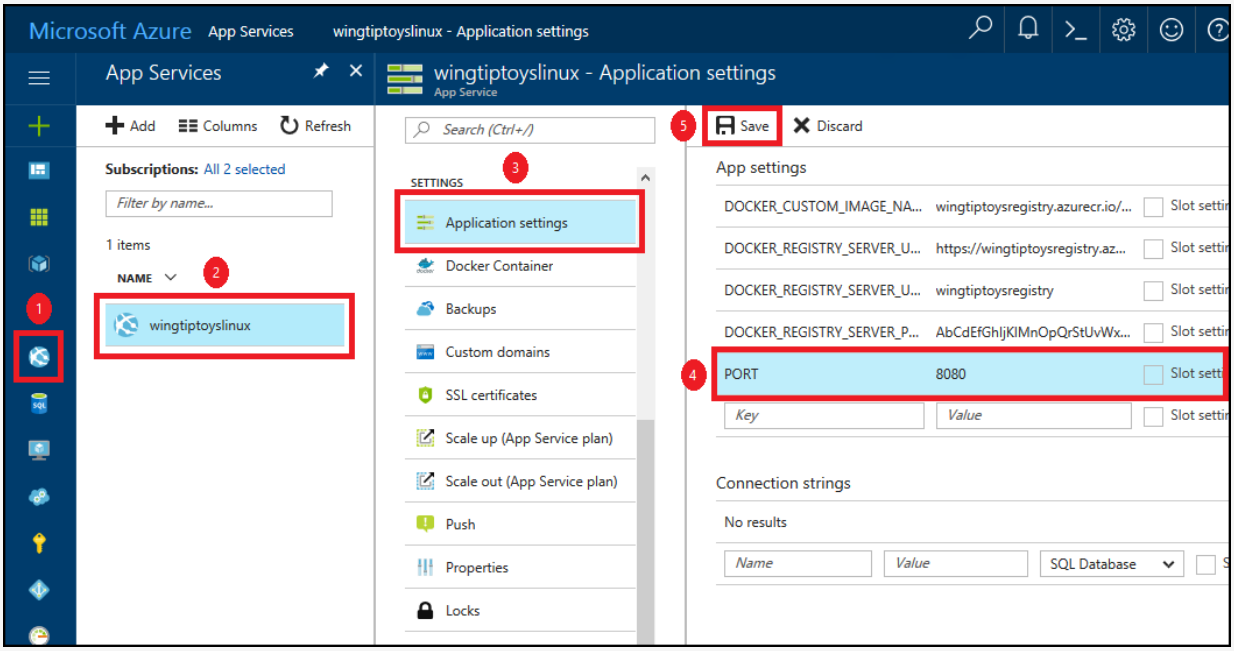


4. Click **Create**.

NOTE

Azure will automatically map Internet requests to embedded Tomcat server which 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 will need to add an environment variable to your web app which defines the port for your embedded Tomcat server. To do so, use the following steps:

1. Browse to the Azure portal and log 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
- Running a Spring Boot Application on a Kubernetes Cluster in the Azure Container Service

Additional resources

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

6/27/2017 • 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 though 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
```

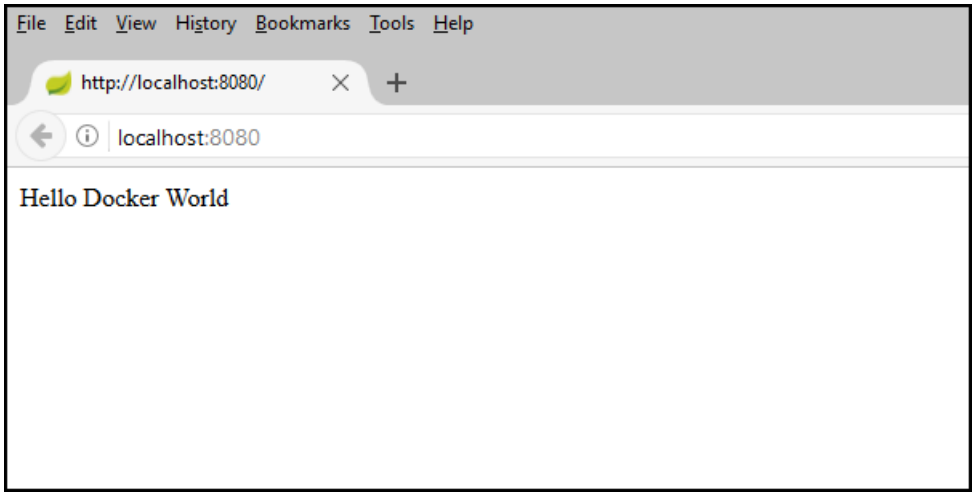
4. Use Maven to build and run the sample app.

```
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=wingtiptoys-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 `wingtiptoysregistry` with a unique name for your registry.

```
az acr create --admin-enabled --resource-group wingtiptoys-kubernetes--location eastus \
 --name wingtiptoysregistry --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 wingtiptoysregistry --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>wingtiptoysregistry</id>
      <username>wingtiptoysregistry</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>wingtiptoysregistry.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>wingtiptoysregistry</serverId>
      <registryUrl>https://wingtiptoysregistry.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 wingtiptoysregistry -p "AbCdEfGhIjKlMnOpQrStUvWxYz" wingtiptoysregistry.azurecr.io
```

Then push your container:

```
docker push wingtiptoysregistry.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 *wingtiptoys-kubernetes* resource group, with *wingtiptoys-containerservice* as the cluster name, and *wingtiptoys-kubernetes* as the DNS prefix:

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

   This command may take a while to complete.

2. Install `kubectl` 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 `kubectl`.

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

Deploy the image to your Kubernetes cluster

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

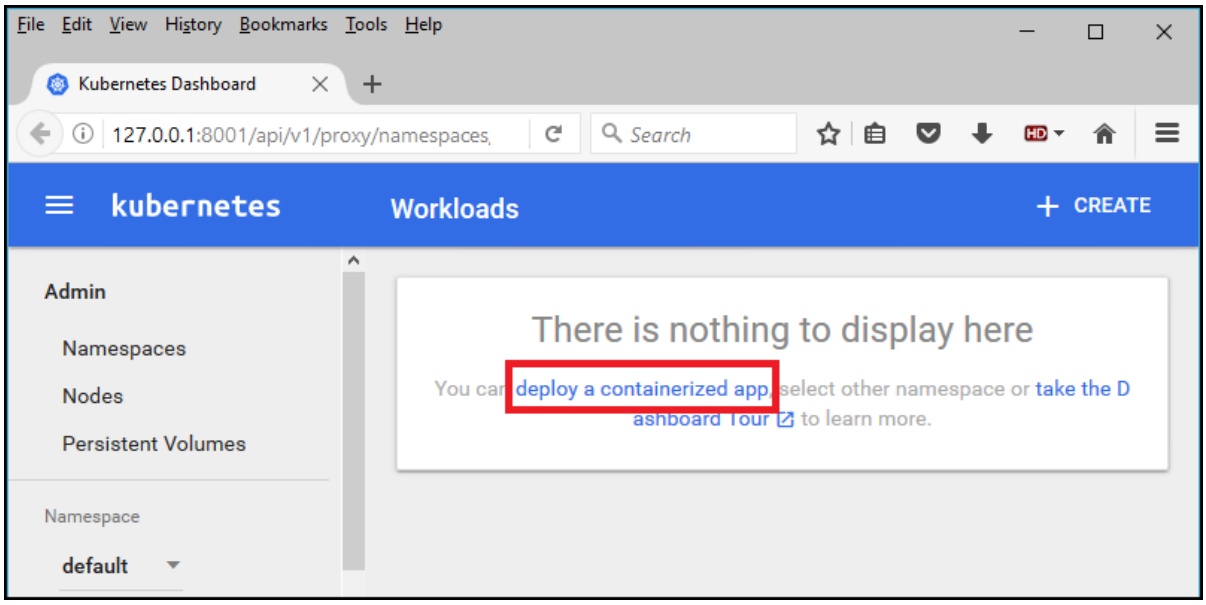
Deploy with the Kubernetes web interface

1. Open a command prompt.

2. Open the configuration website for your Kubernetes cluster in your default browser:

```
az acs 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:

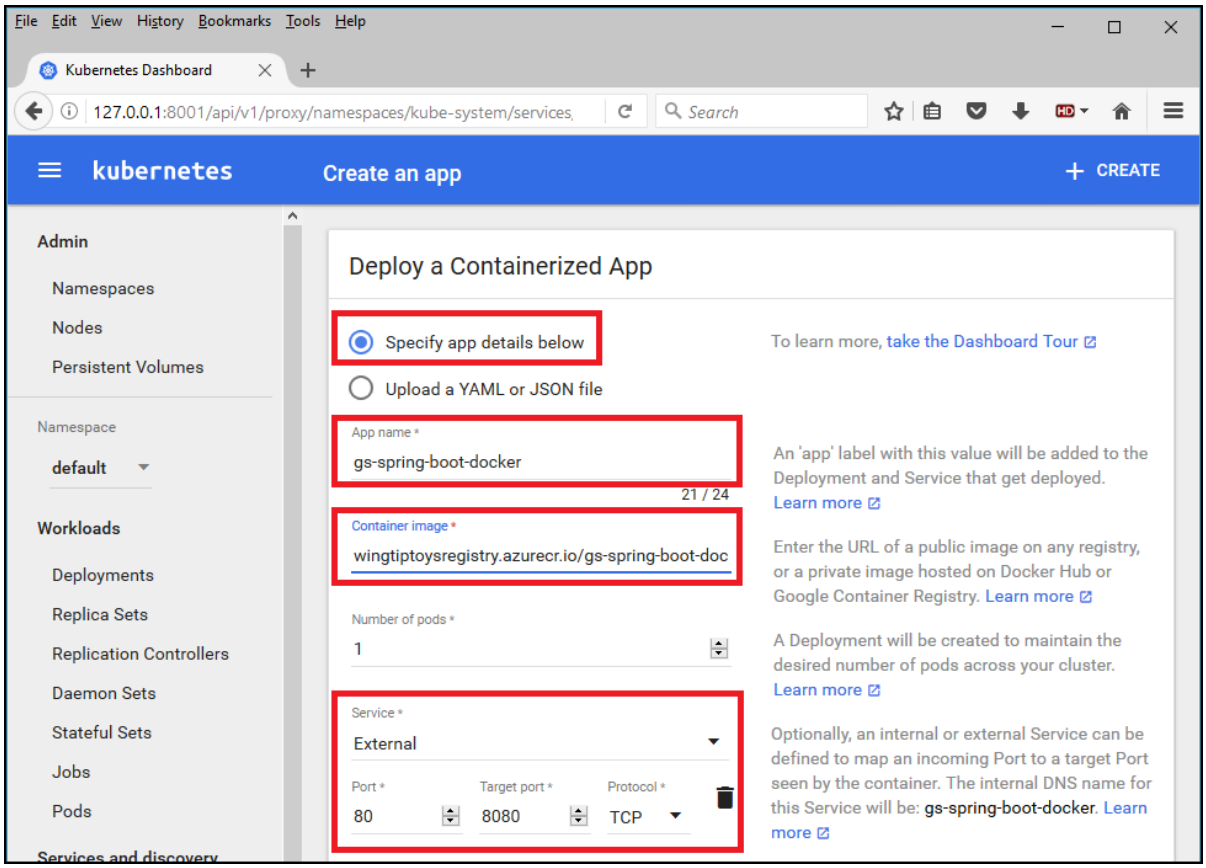
   a. Select **Specify app details below**.

   b. Enter your Spring Boot application name for the **App name**; for example: "*gs-spring-boot-docker*".

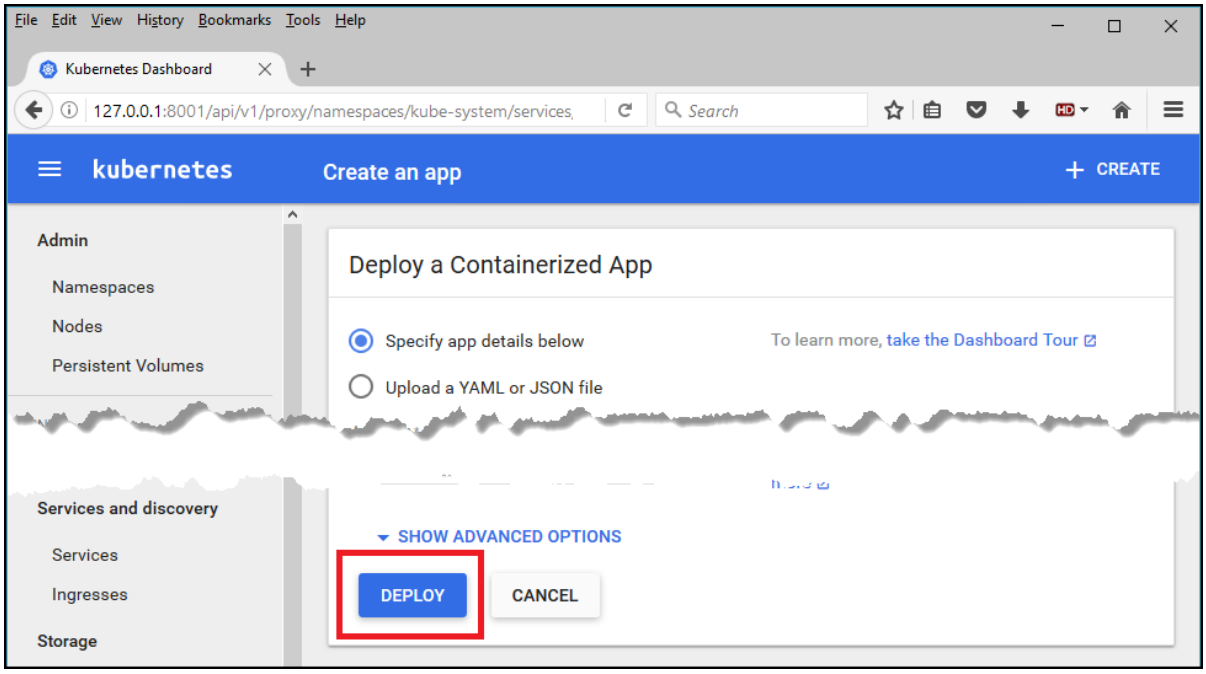
   c. Enter your login server and container image from earlier for the **Container image**; for example: "*wingtiptoysregistry.azurecr.io/gs-spring-boot-docker:latest*".

   d. Choose **External** for the **Service**.

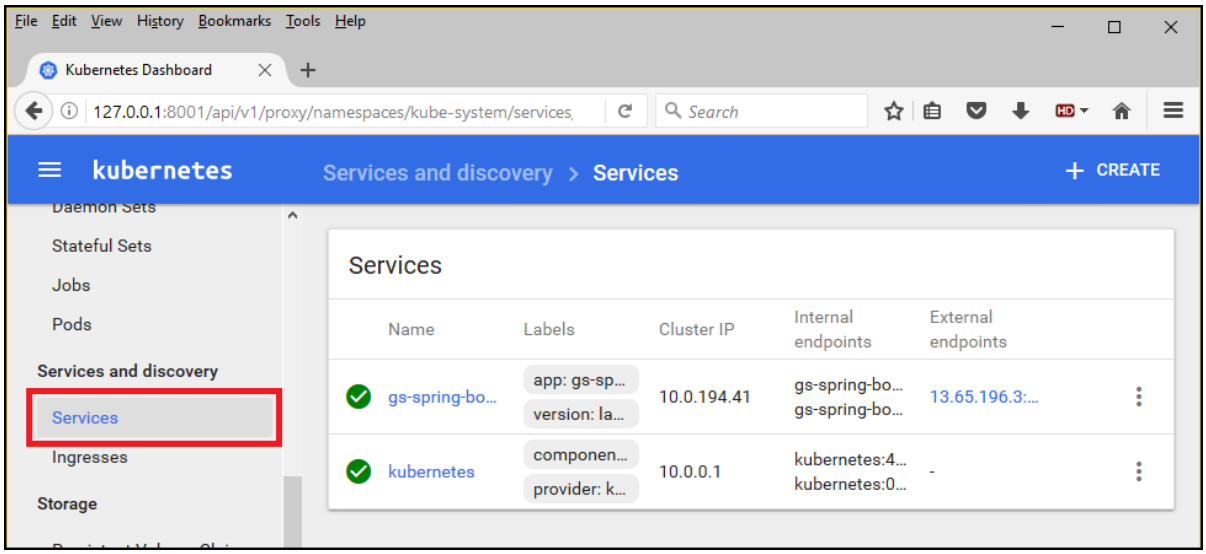
   e. Specify your external and internal ports in the **Port** and **Target port** text boxes.



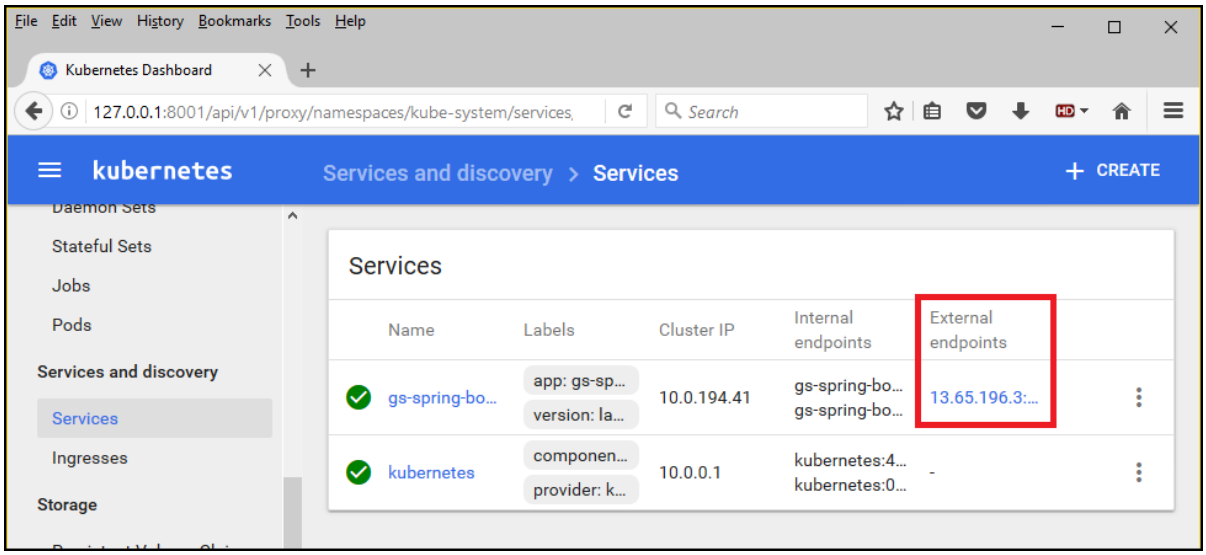
5. Click **Deploy** to deploy the container.

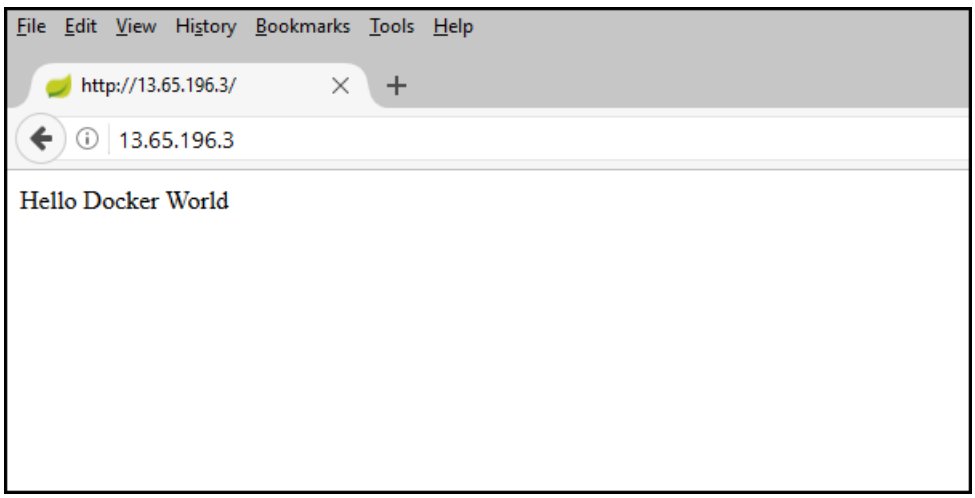


6. Once your application has been deployed, you will see your Spring Boot application listed under **Services**.



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=wingtiptoysregistry.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 `wingtiptoysregistry.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:

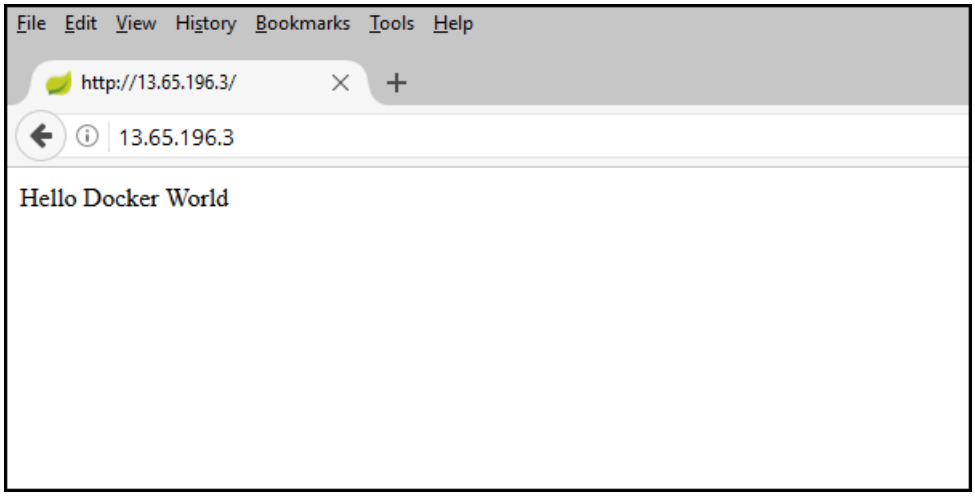
```
kubectl expose deployment gs-spring-boot-docker --type=LoadBalancer --port=80 --target-port=8080
```

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
- Running a Spring Boot Application on Linux in the Azure Container Service

Additional Resources

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.

Container Service frequently asked questions

6/27/2017 • 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.

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 `kubectl 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-acs** followed by a deployment date. Example: microsoft-acs-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.