Contents

Introduction to private Docker container registries in Azure

2/11/2020 • 3 minutes to read • Edit Online

Azure Container Registry is a managed, private Docker registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your private Docker container images and related artifacts.

Use Azure container registries with your existing container development and deployment pipelines, or use Azure Container Registry Tasks to build container images in Azure. Build on demand, or fully automate builds with triggers such as source code commits and base image updates.

For more about Docker and registry concepts, see the Docker overview and About registries, repositories, and images.

Use cases

Pull images from an Azure container registry to various deployment targets:

- **Scalable orchestration systems** that manage containerized applications across clusters of hosts, including Kubernetes, DC/OS, and Docker Swarm.
- **Azure services** that support building and running applications at scale, including Azure Kubernetes Service (AKS), App Service, Batch, Service Fabric, and others.

Developers can also push to a container registry as part of a container development workflow. For example, target a container registry from a continuous integration and delivery tool such as Azure Pipelines or Jenkins.

Configure ACR Tasks to automatically rebuild application images when their base images are updated, or automate image builds when your team commits code to a Git repository. Create multi-step tasks to automate building, testing, and patching multiple container images in parallel in the cloud.

Azure provides tooling including Azure Command-Line Interface, Azure portal, and API support to manage your Azure container registries. Optionally install the Docker Extension for Visual Studio Code and the Azure Account extension to work with your Azure container registries. Pull and push images to an Azure container registry, or run ACR Tasks, all within Visual Studio Code.

Key features

- **Registry SKUs** - Create one or more container registries in your Azure subscription. Registries are available in three SKUs: Basic, Standard, and Premium, each of which supports webhook integration, registry authentication with Azure Active Directory, and delete functionality. Take advantage of local, network-close storage of your container images by creating a registry in the same Azure location as your deployments. Use the geo-replication feature of Premium registries for advanced replication and container image distribution scenarios.
- **Security and access** - You log in to a registry using the Azure CLI or the standard `docker login` command. Azure Container Registry transfers container images over HTTPS, and supports TLS to secure client connections.

> **IMPORTANT**
> Starting January 13, 2020, Azure Container Registry will require all secure connections from servers and applications to use TLS 1.2. Enable TLS 1.2 by using any recent docker client (version 18.03.0 or later). Support for TLS 1.0 and 1.1 will be retired.

You control access to a container registry using an Azure identity, an Azure Active Directory-backed service principal, or a provided admin account. Use role-based access control (RBAC) to assign users or systems fine-grained permissions to a registry.

Security features of the Premium SKU include content trust for image tag signing, and firewalls and virtual networks (preview) to restrict access to the registry. Azure Security Center optionally integrates with Azure Container Registry to scan images whenever an image is pushed to a registry.

- **Supported images and artifacts** - Grouped in a repository, each image is a read-only snapshot of a Docker-compatible container. Azure container registries can include both Windows and Linux images. You control image names for all your container deployments. Use standard Docker commands to push images into a repository, or pull an image from a repository. In addition to Docker container images, Azure Container Registry stores related content formats such as Helm charts and images built to the Open Container Initiative (OCI) Image Format Specification.

- **Automated image builds** - Use Azure Container Registry Tasks (ACR Tasks) to streamline building, testing, pushing, and deploying images in Azure. For example, use ACR Tasks to extend your development inner-loop to the cloud by offloading `docker build` operations to Azure. Configure build tasks to automate your container OS and framework patching pipeline, and build images automatically when your team commits code to source control.

  Multi-step tasks provide step-based task definition and execution for building, testing, and patching container images in the cloud. Task steps define individual container image build and push operations. They can also define the execution of one or more containers, with each step using the container as its execution environment.

Next steps

- Create a container registry using the Azure portal
- Create a container registry using the Azure CLI
- Automate container builds and maintenance with ACR Tasks

Quickstart: Create a private container registry using the Azure CLI

11/24/2019 • 3 minutes to read • Edit Online

Azure Container Registry is a managed Docker container registry service used for storing private Docker container images. This guide details creating an Azure Container Registry instance using the Azure CLI. Then, use Docker commands to push a container image into the registry, and finally pull and run the image from your registry.

This quickstart requires that you are running the Azure CLI (version 2.0.55 or later recommended). Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

You must also have Docker installed locally. Docker provides packages that easily configure Docker on any macOS, Windows, or Linux system.

Because the Azure Cloud Shell doesn't include all required Docker components (the `dockerd` daemon), you can't use the Cloud Shell for this quickstart.

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 a container registry

In this quickstart you create a *Basic* registry, which is a cost-optimized option for developers learning about Azure Container Registry. For details on available service tiers, see Container registry SKUs.

Create an ACR instance using the az acr create command. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. In the following example, *myContainerRegistry007* is used. Update this to a unique value.

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

When the registry is created, the output is similar to the following:

```
{
  "adminUserEnabled": false,
  "creationDate": "2019-01-08T22:32:13.175925+00:00",
  "id": "/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/myContainerRegi
stry007",
  "location": "eastus",
  "loginServer": "mycontainerregistry007.azurecr.io",
  "name": "myContainerRegistry007",
  "provisioningState": "Succeeded",
  "resourceGroup": "myResourceGroup",
  "sku": {
    "name": "Basic",
    "tier": "Basic"
  },
  "status": null,
  "storageAccount": null,
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```


Take note of `loginServer` in the output, which is the fully qualified registry name (all lowercase). Throughout the rest of this quickstart `<acrName>` is a placeholder for the container registry name.

Log in to registry

Before pushing and pulling container images, you must log in to the registry. To do so, use the az acr login command.

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

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

Push image to registry

To push an image to an Azure Container registry, you must first have an image. If you don't yet have any local container images, run the following docker pull command to pull an existing image from Docker Hub. For this example, pull the `hello-world` image.

```
docker pull hello-world
```

Before you can push an image to your registry, you must tag it with the fully qualified name of your ACR login server. The login server name is in the format *<registry-name>.azurecr.io* (all lowercase), for example, *mycontainerregistry007.azurecr.io*.

Tag the image using the docker tag command. Replace `<acrLoginServer>` with the login server name of your ACR instance.

```
docker tag hello-world <acrLoginServer>/hello-world:v1
```

Finally, use docker push to push the image to the ACR instance. Replace `<acrLoginServer>` with the login server name of your ACR instance. This example creates the **hello-world** repository, containing the `hello-world:v1` image.

```
docker push <acrLoginServer>/hello-world:v1
```

After pushing the image to your container registry, remove the `hello-world:v1` image from your local Docker environment. (Note that this docker rmi command does not remove the image from the **hello-world** repository in your Azure container registry.)

```
docker rmi <acrLoginServer>/hello-world:v1
```

List container images

The following example lists the repositories in your registry:

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

Output:

```
Result
----------------
hello-world
```

The following example lists the tags on the **hello-world** repository.

```
az acr repository show-tags --name <acrName> --repository hello-world --output table
```

Output:

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

Run image from registry

Now, you can pull and run the `hello-world:v1` container image from your container registry by using docker run:

```
docker run <acrLoginServer>/hello-world:v1
```

Example output:

```
Unable to find image 'mycontainerregistry007.azurecr.io/hello-world:v1' locally
v1: Pulling from hello-world
Digest: sha256:662dd8e65ef7ccf13f417962c2f77567d3b132f12c95909de6c85ac3c326a345
Status: Downloaded newer image for mycontainerregistry007.azurecr.io/hello-world:v1

Hello from Docker!
This message shows that your installation appears to be working correctly.

[...]
```

Clean up resources

When no longer needed, you can use the az group delete command to remove the resource group, the container registry, and the container images stored there.

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

Next steps

In this quickstart, you created an Azure Container Registry with the Azure CLI, pushed a container image to the registry, and pulled and ran the image from the registry. Continue to the Azure Container Registry tutorials for a deeper look at ACR.

Azure Container Registry tutorials

Quickstart: Create a private container registry using the Azure portal

1/2/2020 • 3 minutes to read • Edit Online

An Azure container registry is a private Docker registry in Azure where you can store and manage your private Docker container images. In this quickstart, you create a container registry with the Azure portal. Then, use Docker commands to push a container image into the registry, and finally pull and run the image from your registry.

To log in to the registry to work with container images, this quickstart requires that you are running the Azure CLI (version 2.0.55 or later recommended). Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

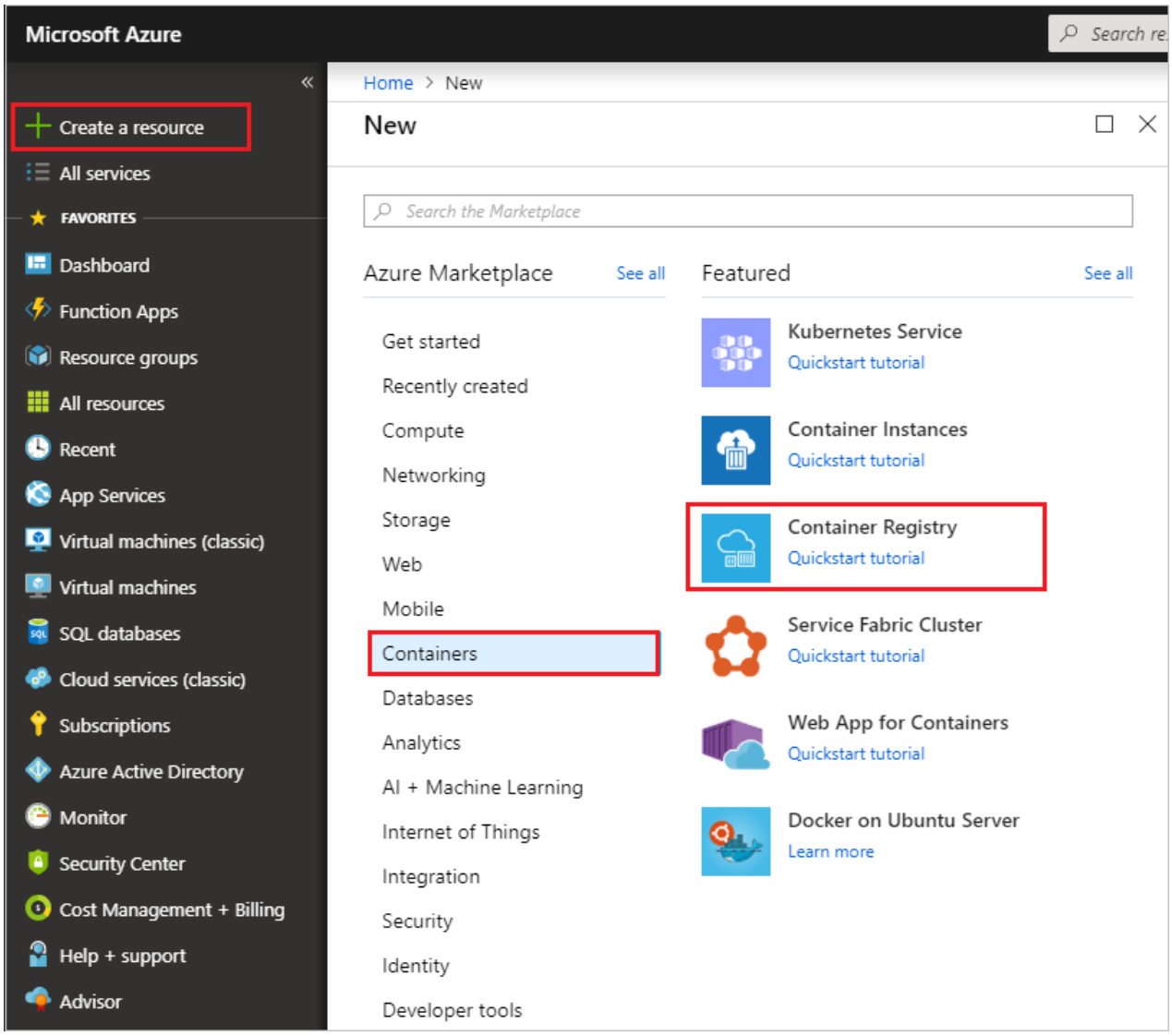
You must also have Docker installed locally. Docker provides packages that easily configure Docker on any Mac, Windows, or Linux system.

Sign in to Azure

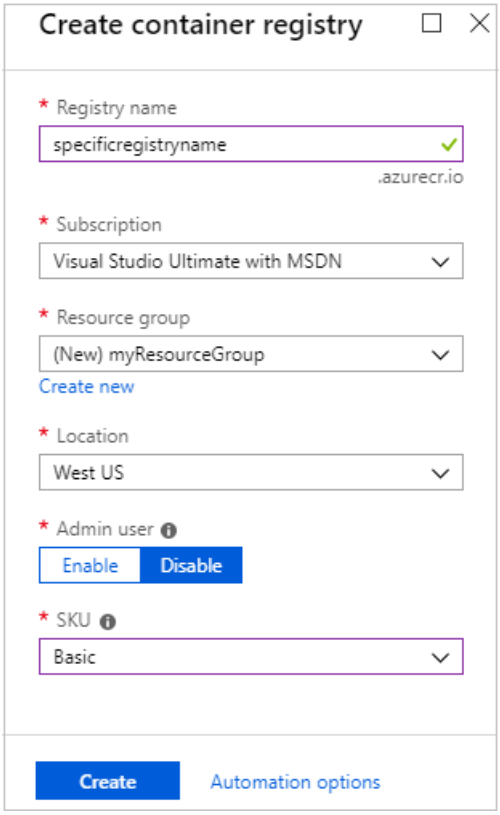
Sign in to the Azure portal at https://portal.azure.com.

Create a container registry

Select **Create a resource** > **Containers** > **Container Registry**.

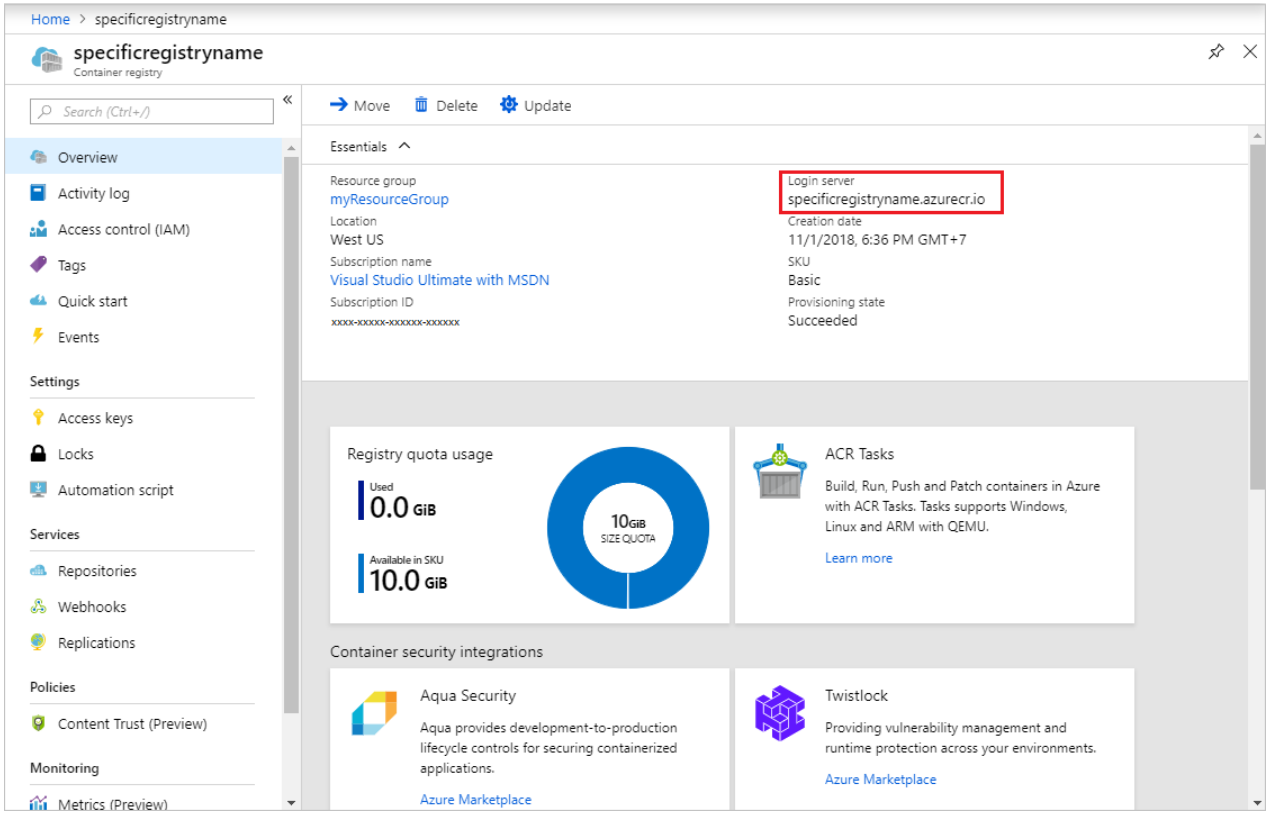


Enter values for **Registry name** and **Resource group**. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. For this quickstart create a new resource group in the `West US` location named `myResourceGroup`, and for **SKU**, select 'Basic'. Select **Create** to deploy the ACR instance.



In this quickstart you create a *Basic* registry, which is a cost-optimized option for developers learning about Azure Container Registry. For details on available service tiers, see Container registry SKUs.

When the **Deployment succeeded** message appears, select the container registry in the portal.



Take note of the value of the **Login server**. You use this value in the following steps while working with your registry with the Azure CLI and Docker.

Log in to registry

Before pushing and pulling container images, you must log in to the ACR instance. Open a command shell in your

operating system, and use the az acr login command in the Azure CLI.(Specify only Container Name. Do not include 'azurecr.io')

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

The command returns `Login Succeeded` once completed.

Push image to registry

To push an image to an Azure Container registry, you must first have an image. If you don't yet have any local container images, run the following docker pull command to pull an existing image from Docker Hub. For this example, pull the `hello-world` image.

```
docker pull hello-world
```

Before you can push an image to your registry, you must tag it with the fully qualified name of your ACR login server. The login server name is in the format *<registry-name>.azurecr.io* (all lowercase), for example, *mycontainerregistry007.azurecr.io*.

Tag the image using the docker tag command. Replace `<acrLoginServer>` with the login server name of your ACR instance.

```
docker tag hello-world <acrLoginServer>/hello-world:v1
```

Finally, use docker push to push the image to the ACR instance. Replace `<acrLoginServer>` with the login server name of your ACR instance. This example creates the **hello-world** repository, containing the `hello-world:v1` image.

```
docker push <acrLoginServer>/hello-world:v1
```

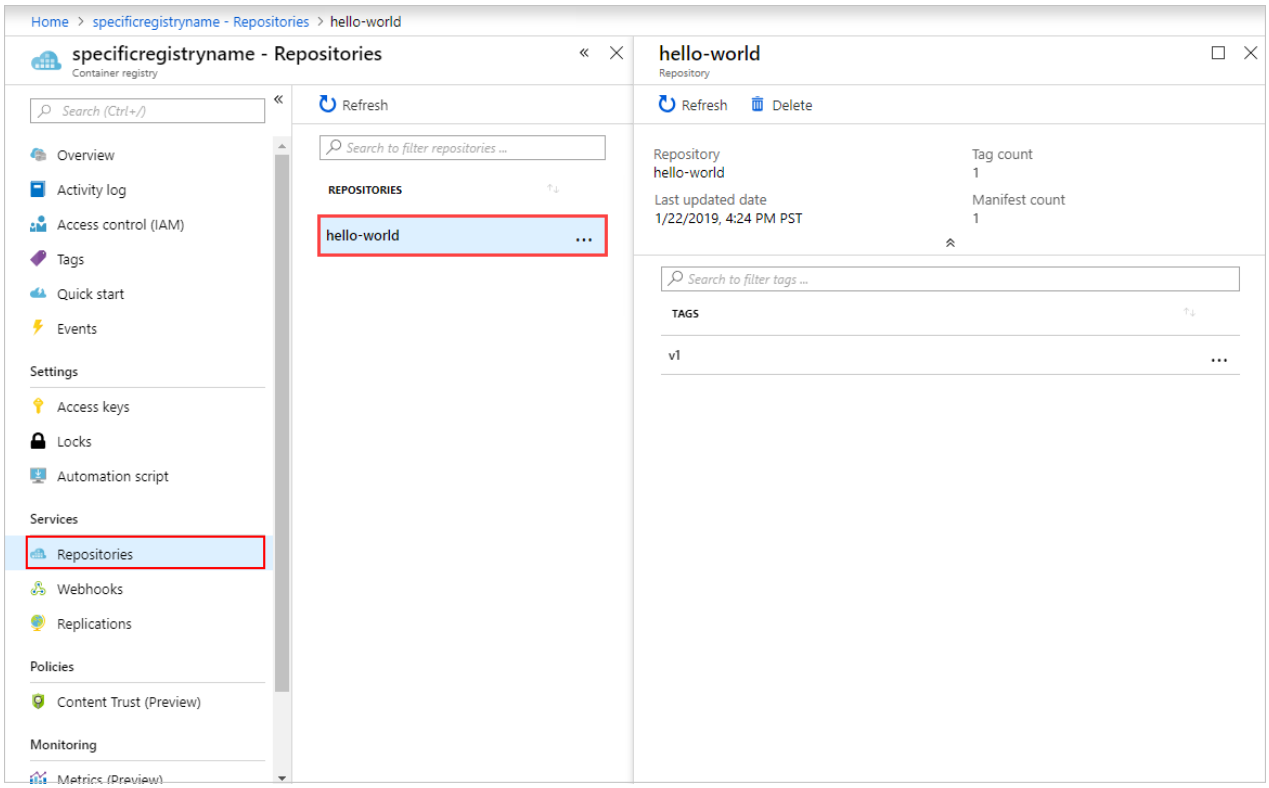
After pushing the image to your container registry, remove the `hello-world:v1` image from your local Docker environment. (Note that this docker rmi command does not remove the image from the **hello-world** repository in your Azure container registry.)

```
docker rmi <acrLoginServer>/hello-world:v1
```

List container images

To list the images in your registry, navigate to your registry in the portal and select **Repositories**, then select the repository you created with `docker push`.

In this example, we select the **hello-world** repository, and we can see the `v1`-tagged image under **TAGS**.



Run image from registry

Now, you can pull and run the `hello-world:v1` container image from your container registry by using docker run:

```
docker run <acrLoginServer>/hello-world:v1
```

Example output:

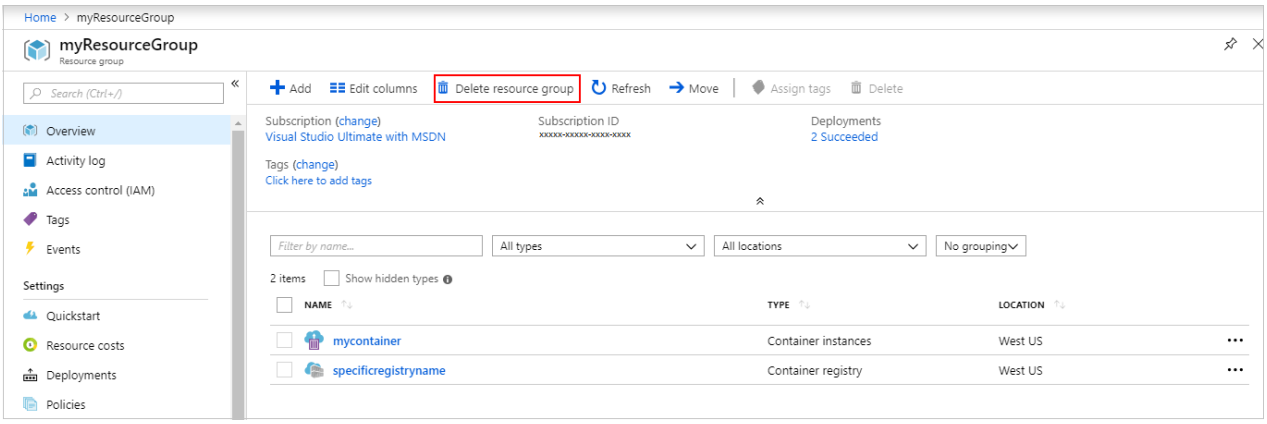
```
Unable to find image 'mycontainerregistry007.azurecr.io/hello-world:v1' locally
v1: Pulling from hello-world
Digest: sha256:662dd8e65ef7ccf13f417962c2f77567d3b132f12c95909de6c85ac3c326a345
Status: Downloaded newer image for mycontainerregistry007.azurecr.io/hello-world:v1

Hello from Docker!
This message shows that your installation appears to be working correctly.

[...]
```

Clean up resources

To clean up your resources, navigate to the **myResourceGroup** resource group in the portal. Once the resource group is loaded click on **Delete resource group** to remove the resource group, the container registry, and the container images stored there.



Next steps

In this quickstart, you created an Azure Container Registry with the Azure portal, pushed a container image, and pulled and ran the image from the registry. Continue to the Azure Container Registry tutorials for a deeper look at ACR.

Azure Container Registry tutorials

Quickstart: Create a private container registry using Azure PowerShell

11/24/2019 • 3 minutes to read • Edit Online

Azure Container Registry is a managed, private Docker container registry service for building, storing, and serving Docker container images. In this quickstart, you learn how to create an Azure container registry using PowerShell. Then, use Docker commands to push a container image into the registry, and finally pull and run the image from your registry.

Prerequisites

> **NOTE**
> This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see Introducing the new Azure PowerShell Az module. For Az module installation instructions, see Install Azure PowerShell.

This quickstart requires Azure PowerShell module. Run `Get-Module -ListAvailable Az` to determine your installed version. If you need to install or upgrade, see Install Azure PowerShell module.

You must also have Docker installed locally. Docker provides packages for macOS, Windows, and Linux systems.

Because the Azure Cloud Shell doesn't include all required Docker components (the `dockerd` daemon), you can't use the Cloud Shell for this quickstart.

Sign in to Azure

Sign in to your Azure subscription with the Connect-AzAccount command, and follow the on-screen directions.

```
Connect-AzAccount
```

Create resource group

Once you're authenticated with Azure, create a resource group with New-AzResourceGroup. A resource group is a logical container in which you deploy and manage your Azure resources.

```
New-AzResourceGroup -Name myResourceGroup -Location EastUS
```

Create container registry

Next, create a container registry in your new resource group with the New-AzContainerRegistry command.

The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. The following example creates a registry named "myContainerRegistry007." Replace *myContainerRegistry007* in the following command, then run it to create the registry:

```
$registry = New-AzContainerRegistry -ResourceGroupName "myResourceGroup" -Name "myContainerRegistry007" -
EnableAdminUser -Sku Basic
```

In this quickstart you create a *Basic* registry, which is a cost-optimized option for developers learning about Azure Container Registry. For details on available service tiers, see Container registry SKUs.

Log in to registry

Before pushing and pulling container images, you must log in to your registry. In production scenarios you should use an individual identity or service principal for container registry access, but to keep this quickstart brief, enable the admin user on your registry with the Get-AzContainerRegistryCredential command:

```
$creds = Get-AzContainerRegistryCredential -Registry $registry
```

Next, run docker login to log in:

```
$creds.Password | docker login $registry.LoginServer -u $creds.Username --password-stdin
```

The command returns `Login Succeeded` once completed.

Push image to registry

To push an image to an Azure Container registry, you must first have an image. If you don't yet have any local container images, run the following docker pull command to pull an existing image from Docker Hub. For this example, pull the `hello-world` image.

```
docker pull hello-world
```

Before you can push an image to your registry, you must tag it with the fully qualified name of your ACR login server. The login server name is in the format *<registry-name>.azurecr.io* (all lowercase), for example, *mycontainerregistry007.azurecr.io*.

Tag the image using the docker tag command. Replace `<acrLoginServer>` with the login server name of your ACR instance.

```
docker tag hello-world <acrLoginServer>/hello-world:v1
```

Finally, use docker push to push the image to the ACR instance. Replace `<acrLoginServer>` with the login server name of your ACR instance. This example creates the **hello-world** repository, containing the `hello-world:v1` image.

```
docker push <acrLoginServer>/hello-world:v1
```

After pushing the image to your container registry, remove the `hello-world:v1` image from your local Docker environment. (Note that this docker rmi command does not remove the image from the **hello-world** repository in your Azure container registry.)

```
docker rmi <acrLoginServer>/hello-world:v1
```

Run image from registry

Now, you can pull and run the `hello-world:v1` container image from your container registry by using docker run:

```
docker run <acrLoginServer>/hello-world:v1
```

Example output:

```
Unable to find image 'mycontainerregistry007.azurecr.io/hello-world:v1' locally
v1: Pulling from hello-world
Digest: sha256:662dd8e65ef7ccf13f417962c2f77567d3b132f12c95909de6c85ac3c326a345
Status: Downloaded newer image for mycontainerregistry007.azurecr.io/hello-world:v1

Hello from Docker!
This message shows that your installation appears to be working correctly.

[...]
```

Clean up resources

Once you're done working with the resources you created in this quickstart, use the Remove-AzResourceGroup command to remove the resource group, the container registry, and the container images stored there:

```
Remove-AzResourceGroup -Name myResourceGroup
```

Next steps

In this quickstart, you created an Azure Container Registry with Azure PowerShell, pushed a container image, and pulled and ran the image from the registry. Continue to the Azure Container Registry tutorials for a deeper look at ACR.

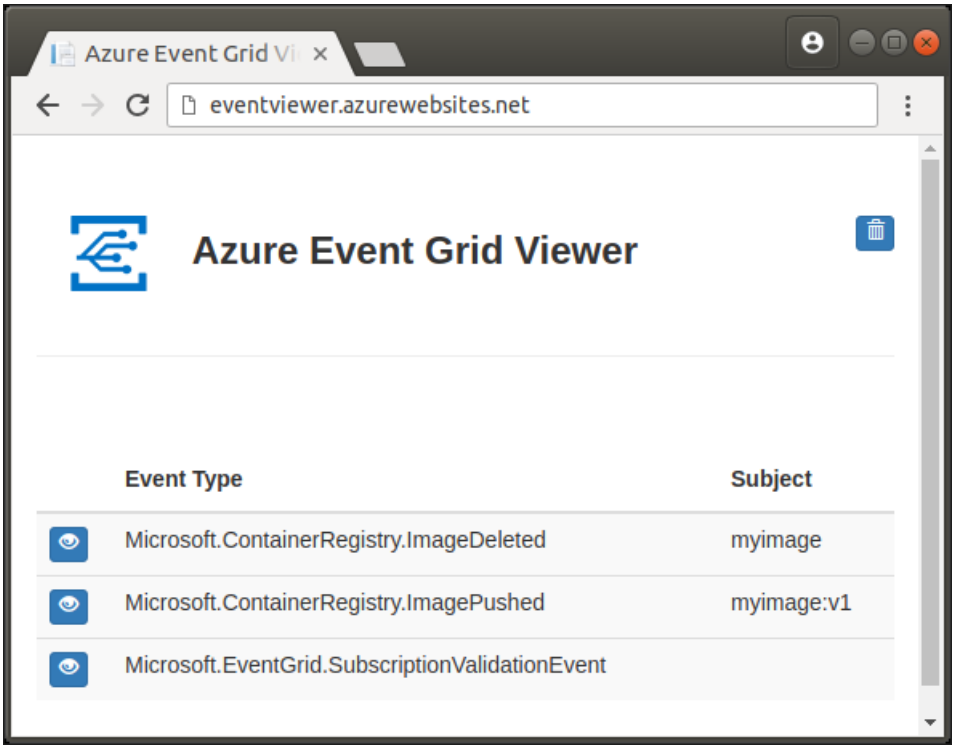
Azure Container Registry tutorials

Quickstart: Send events from private container registry to Event Grid

11/24/2019 • 7 minutes to read • Edit Online

Azure Event Grid is a fully managed event routing service that provides uniform event consumption using a publish-subscribe model. In this quickstart, you use the Azure CLI to create a container registry, subscribe to registry events, then deploy a sample web application to receive the events. Finally, you trigger container image `push` and `delete` events and view the event payload in the sample application.

After you complete the steps in this article, events sent from your container registry to Event Grid appear in the sample web app:



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

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	

OPTION	EXAMPLE/LINK
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by selecting **Cmd**+**Shift**+**V** on macOS.
4. Select **Enter** to run the code.

The Azure CLI commands in this article are formatted for the **Bash** shell. If you're using a different shell like PowerShell or Command Prompt, you may need to adjust line continuation characters or variable assignment lines accordingly. This article uses variables to minimize the amount of command editing required.

Create a resource group

An Azure resource group is a logical container in which you deploy and manage your Azure resources. The following az group create command creates a resource group named *myResourceGroup* in the *eastus* region. If you want to use a different name for your resource group, set `RESOURCE_GROUP_NAME` to a different value.

```
RESOURCE_GROUP_NAME=myResourceGroup

az group create --name $RESOURCE_GROUP_NAME --location eastus
```

Create a container registry

Next, deploy a container registry into the resource group with the following commands. Before you run the az acr create command, set `ACR_NAME` to a name for your registry. The name must be unique within Azure, and is restricted to 5-50 alphanumeric characters.

```
ACR_NAME=<acrName>

az acr create --resource-group $RESOURCE_GROUP_NAME --name $ACR_NAME --sku Basic
```

Once the registry has been created, the Azure CLI returns output similar to the following:

```
{
  "adminUserEnabled": false,
  "creationDate": "2018-08-16T20:02:46.569509+00:00",
  "id": "/subscriptions/<Subscription
ID>/resourceGroups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/myregistry",
  "location": "eastus",
  "loginServer": "myregistry.azurecr.io",
  "name": "myregistry",
  "provisioningState": "Succeeded",
  "resourceGroup": "myResourceGroup",
  "sku": {
    "name": "Basic",
    "tier": "Basic"
  },
  "status": null,
  "storageAccount": null,
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```

Create an event endpoint

In this section, you use a Resource Manager template located in a GitHub repository to deploy a pre-built sample web application to Azure App Service. Later, you subscribe to your registry's Event Grid events and specify this app as the endpoint to which the events are sent.

To deploy the sample app, set `SITE_NAME` to a unique name for your web app, and execute the following commands. The site name must be unique within Azure because it forms part of the fully qualified domain name (FQDN) of the web app. In a later section, you navigate to the app's FQDN in a web browser to view your registry's events.

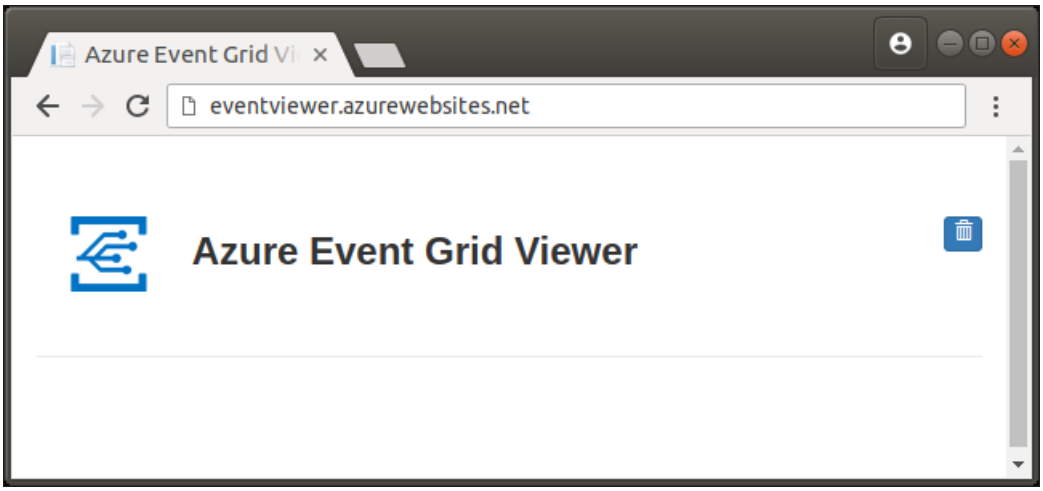
```
SITE_NAME=<your-site-name>

az group deployment create \
    --resource-group $RESOURCE_GROUP_NAME \
    --template-uri "https://raw.githubusercontent.com/Azure-Samples/azure-event-grid-
viewer/master/azuredeploy.json" \
    --parameters siteName=$SITE_NAME hostingPlanName=$SITE_NAME-plan
```

Once the deployment has succeeded (it might take a few minutes), open a browser and navigate to your web app to make sure it's running:

`http://<your-site-name>.azurewebsites.net`

You should see the sample app rendered with no event messages displayed:



Enable Event Grid resource provider

If you haven't previously used Event Grid in your Azure subscription, you may need to register the Event Grid resource provider. Run the following command to register the provider:

```
az provider register --namespace Microsoft.EventGrid
```

It may take a moment for the registration to finish. To check the status, run:

```
az provider show --namespace Microsoft.EventGrid --query "registrationState"
```

When `registrationState` is `Registered`, you're ready to continue.

Subscribe to registry events

In Event Grid, you subscribe to a *topic* to tell it which events you want to track, and where to send them. The following az eventgrid event-subscription create command subscribes to the container registry you created, and specifies your web app's URL as the endpoint to which it should send events. The environment variables you populated in earlier sections are reused here, so no edits are required.

```
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)
APP_ENDPOINT=https://$SITE_NAME.azurewebsites.net/api/updates

az eventgrid event-subscription create \
    --name event-sub-acr \
    --source-resource-id $ACR_REGISTRY_ID \
    --endpoint $APP_ENDPOINT
```

When the subscription is completed, you should see output similar to the following:

```
{
  "destination": {
    "endpointBaseUrl": "https://eventgridviewer.azurewebsites.net/api/updates",
    "endpointType": "WebHook",
    "endpointUrl": null
  },
  "filter": {
    "includedEventTypes": [
      "All"
    ],
    "isSubjectCaseSensitive": null,
    "subjectBeginsWith": "",
    "subjectEndsWith": ""
  },
  "id": "/subscriptions/<Subscription
ID>/resourceGroups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/myregistry/providers/Micros
oft.EventGrid/eventSubscriptions/event-sub-acr",
  "labels": null,
  "name": "event-sub-acr",
  "provisioningState": "Succeeded",
  "resourceGroup": "myResourceGroup",
  "topic": "/subscriptions/<Subscription
ID>/resourceGroups/myresourcegroup/providers/microsoft.containerregistry/registries/myregistry",
  "type": "Microsoft.EventGrid/eventSubscriptions"
}
```


Trigger registry events

Now that the sample app is up and running and you've subscribed to your registry with Event Grid, you're ready to generate some events. In this section, you use ACR Tasks to build and push a container image to your registry. ACR Tasks is a feature of Azure Container Registry that allows you to build container images in the cloud, without needing the Docker Engine installed on your local machine.

Build and push image

Execute the following Azure CLI command to build a container image from the contents of a GitHub repository. By default, ACR Tasks automatically pushes a successfully built image to your registry, which generates the `ImagePushed` event.

```
az acr build --registry $ACR_NAME --image myimage:v1 -f Dockerfile https://github.com/Azure-Samples/acr-build-
helloworld-node.git
```

You should see output similar to the following while ACR Tasks builds and then pushes your image. The following sample output has been truncated for brevity.

```
$ az acr build -r $ACR_NAME --image myimage:v1 -f Dockerfile https://github.com/Azure-Samples/acr-build-
helloworld-node.git
Sending build context to ACR...
Queued a build with build ID: aa2
Waiting for build agent...
2018/08/16 22:19:38 Using acb_vol_27a2afa6-27dc-4ae4-9e52-6d6c8b7455b2 as the home volume
2018/08/16 22:19:38 Setting up Docker configuration...
2018/08/16 22:19:39 Successfully set up Docker configuration
2018/08/16 22:19:39 Logging in to registry: myregistry.azurecr.io
2018/08/16 22:19:55 Successfully logged in
Sending build context to Docker daemon  94.72kB
Step 1/5 : FROM node:9-alpine
...
```

To verify that the built image is in your registry, execute the following command to view the tags in the "myimage"

repository:

```
az acr repository show-tags --name $ACR_NAME --repository myimage
```

The "v1" tag of the image you built should appear in the output, similar to the following:

```
$ az acr repository show-tags --name $ACR_NAME --repository myimage
[
  "v1"
]
```

Delete the image

Now, generate an `ImageDeleted` event by deleting the image with the az acr repository delete command:

```
az acr repository delete --name $ACR_NAME --image myimage:v1
```

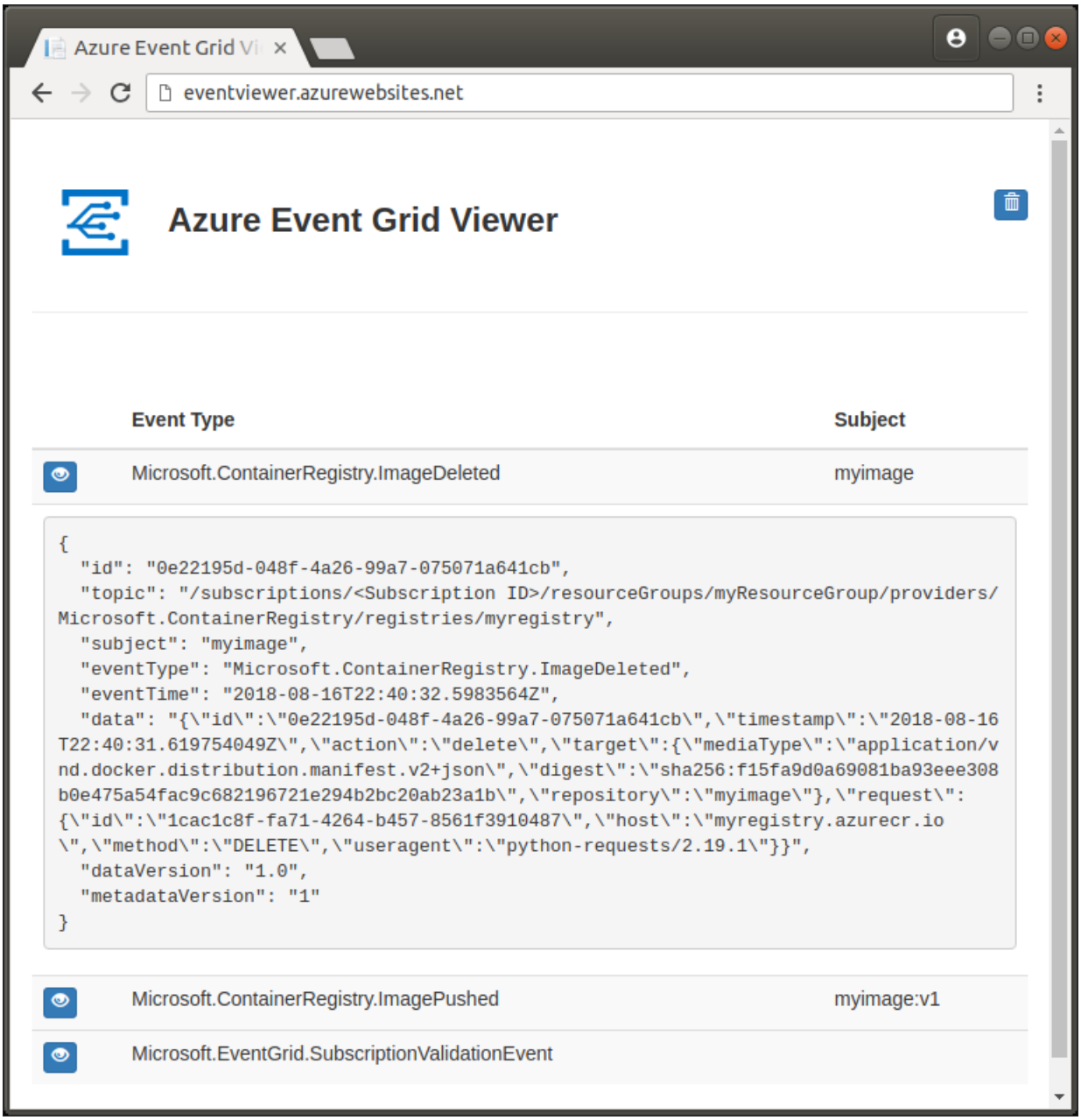
You should see output similar to the following, asking for confirmation to delete the manifest and associated images:

```
$ az acr repository delete --name $ACR_NAME --image myimage:v1
This operation will delete the manifest
'sha256:f15fa9d0a69081ba93eee308b0e475a54fac9c682196721e294b2bc20ab23a1b' and all the following images:
'myimage:v1'.
Are you sure you want to continue? (y/n): y
```

View registry events

You've now pushed an image to your registry and then deleted it. Navigate to your Event Grid Viewer web app, and you should see both `ImageDeleted` and `ImagePushed` events. You might also see a subscription validation event generated by executing the command in the Subscribe to registry events section.

The following screenshot shows the sample app with the three events, and the `ImageDeleted` event is expanded to show its details.



Congratulations! If you see the `ImagePushed` and `ImageDeleted` events, your registry is sending events to Event Grid, and Event Grid is forwarding those events to your web app endpoint.

Clean up resources

Once you're done with the resources you created in this quickstart, you can delete them all with the following Azure CLI command. When you delete a resource group, all of the resources it contains are permanently deleted.

WARNING: This operation is irreversible. Be sure you no longer need any of the resources in the group before running the command.

```
az group delete --name $RESOURCE_GROUP_NAME
```

Event Grid event schema

You can find the Azure Container Registry event message schema reference in the Event Grid documentation:

Azure Event Grid event schema for Container Registry

Next steps

In this quickstart, you deployed a container registry, built an image with ACR Tasks, deleted it, and have consumed your registry's events from Event Grid with a sample application. Next, move on to the ACR Tasks tutorial to learn more about building container images in the cloud, including automated builds on base image update:

Build container images in the cloud with ACR Tasks

Tutorial: Prepare a geo-replicated Azure container registry

11/24/2019 • 5 minutes to read • Edit Online

An Azure container registry is a private Docker registry deployed in Azure that you can keep network-close to your deployments. In this set of three tutorial articles, you learn how to use geo-replication to deploy an ASP.NET Core web application running in a Linux container to two Web Apps for Containers instances. You'll see how Azure automatically deploys the image to each Web App instance from the closest geo-replicated repository.

In this tutorial, part one in a three-part series:

- Create a geo-replicated Azure container registry
- Clone application source code from GitHub
- Build a Docker container image from application source
- Push the container image to your registry

In subsequent tutorials, you deploy the container from your private registry to a web app running in two Azure regions. You then update the code in the application, and update both Web App instances with a single `docker push` to your registry.

Before you begin

This tutorial requires a local installation of the Azure CLI (version 2.0.31 or later). Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

You should be familiar with core Docker concepts such as containers, container images, and basic Docker CLI commands. For a primer on container basics, see Get started with Docker.

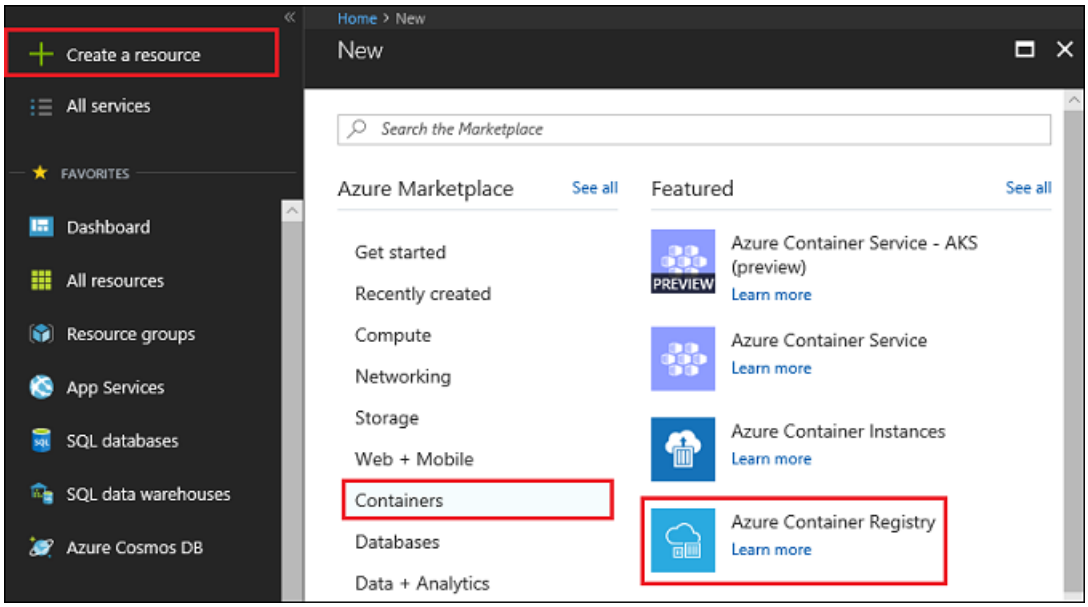
To complete this tutorial, you need a local Docker installation. Docker provides installation instructions for macOS, Windows, and Linux systems.

Azure Cloud Shell does not include the Docker components required to complete every step this tutorial. Therefore, we recommend a local installation of the Azure CLI and Docker development environment.

Create a container registry

Sign in to the Azure portal.

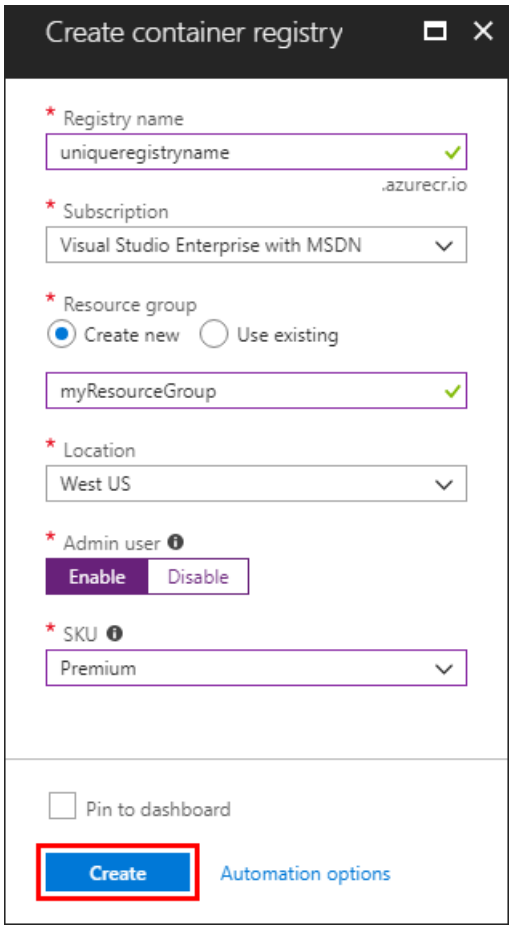
Select **Create a resource** > **Containers** > **Azure Container Registry**.



Configure your new registry with the following settings:

- **Registry name**: Create a registry name that's globally unique within Azure, and contains 5-50 alphanumeric characters
- **Resource Group**: **Create new** > `myResourceGroup`
- **Location**: `West US`
- **Admin user**: `Enable` (required for Web App for Containers to pull images)
- **SKU**: `Premium` (required for geo-replication)

Select **Create** to deploy the ACR instance.



Throughout the rest of this tutorial, we use `<acrName>` as a placeholder for the container **Registry name** that you chose.

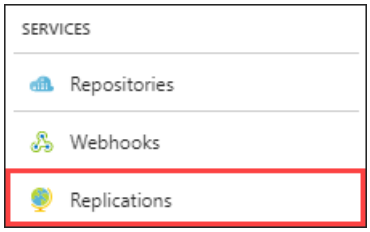
TIP

Because Azure container registries are typically long-lived resources that are used across multiple container hosts, we recommend that you create your registry in its own resource group. As you configure geo-replicated registries and webhooks, these additional resources are placed in the same resource group.

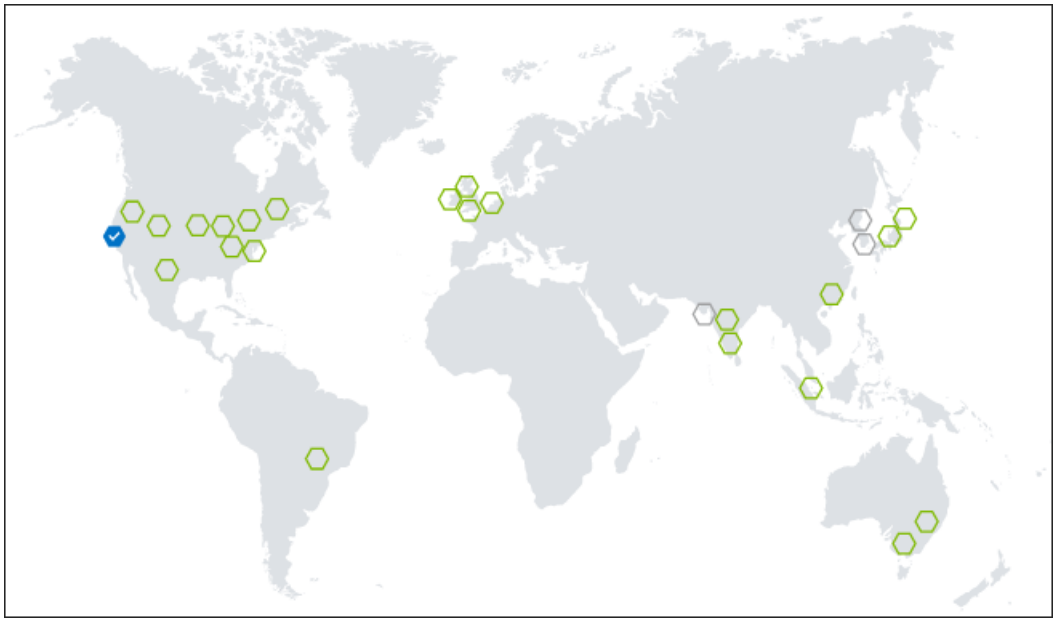
Configure geo-replication

Now that you have a Premium registry, you can configure geo-replication. Your web app, which you configure in the next tutorial to run in two regions, can then pull its container images from the nearest registry.

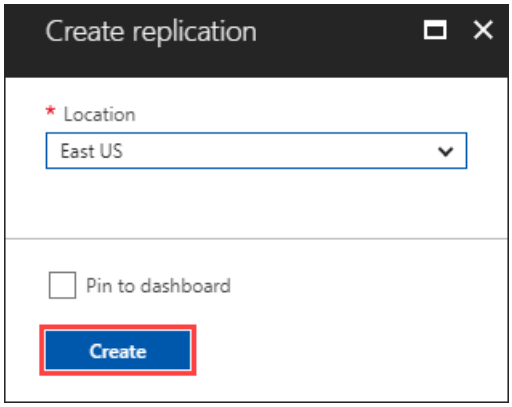
Navigate to your new container registry in the Azure portal and select **Replications** under **SERVICES**:



A map is displayed showing green hexagons representing Azure regions available for geo-replication:



Replicate your registry to the East US region by selecting its green hexagon, then select **Create** under **Create replication**:



When the replication is complete, the portal reflects *Ready* for both regions. Use the **Refresh** button to refresh the status of the replication; it can take a minute or so for the replicas to be created and synchronized.

NAME	LOCATION	PROVISIONING STATE	STATUS	
eastus	East US	Succeeded	Ready	...
westus	West US	Succeeded	Ready	...

Container registry login

Now that you've configured geo-replication, build a container image and push it to your registry. You must first log in to your ACR instance before pushing images to it.

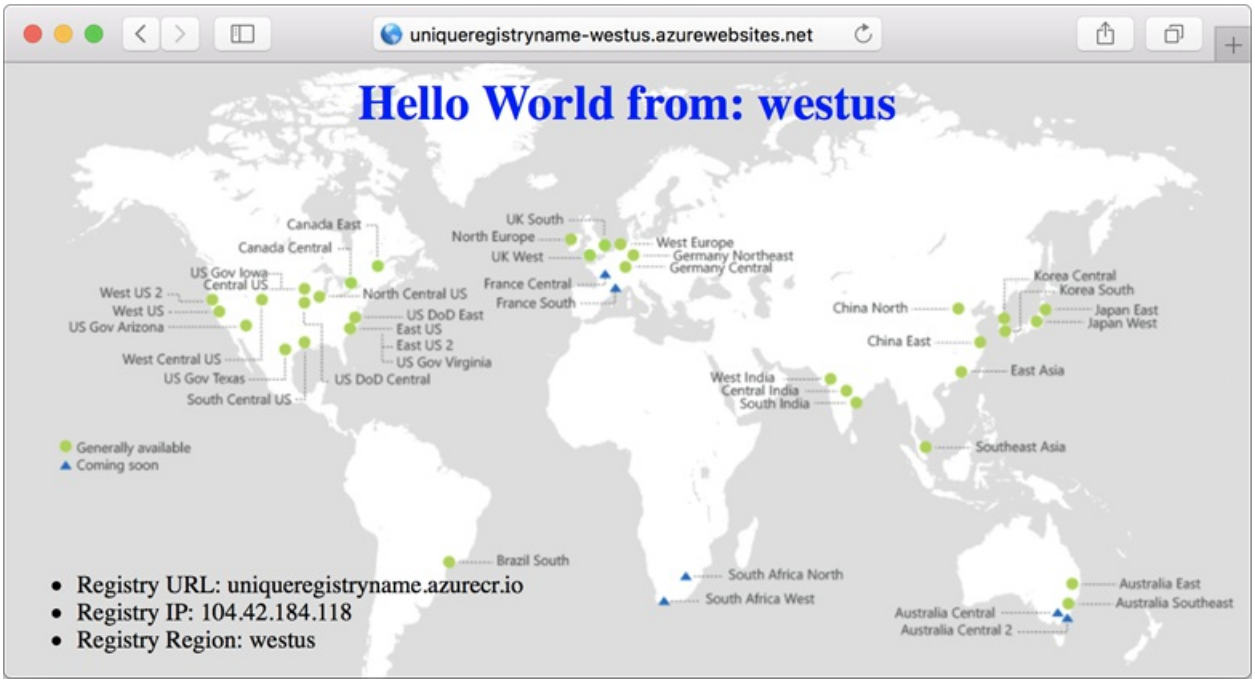
Use the az acr login command to authenticate and cache the credentials for your registry. Replace `<acrName>` with the name of the registry you created earlier.

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

The command returns `Login Succeeded` when complete.

Get application code

The sample in this tutorial includes a small web application built with ASP.NET Core. The app serves an HTML page that displays the region from which the image was deployed by Azure Container Registry.



Use git to download the sample into a local directory, and `cd` into the directory:

```
git clone https://github.com/Azure-Samples/acr-helloworld.git
cd acr-helloworld
```

If you don't have `git` installed, you can download the ZIP archive directly from GitHub.

Update Dockerfile

The Dockerfile included in the sample shows how the container is built. It starts from an official aspnetcore image, copies the application files into the container, installs dependencies, compiles the output using the official aspnetcore-build image, and finally, builds an optimized aspnetcore image.

The Dockerfile is located at `./AcrHelloworld/Dockerfile` in the cloned source.

```
FROM microsoft/aspnetcore:2.0 AS base
# Update <acrName> with the name of your registry
# Example: uniqueregistryname.azurecr.io
ENV DOCKER_REGISTRY <acrName>.azurecr.io
WORKDIR /app
EXPOSE 80

FROM microsoft/aspnetcore-build:2.0 AS build
WORKDIR /src
COPY *.sln ./
COPY AcrHelloworld/AcrHelloworld.csproj AcrHelloworld/
RUN dotnet restore
COPY . .
WORKDIR /src/AcrHelloworld
RUN dotnet build -c Release -o /app

FROM build AS publish
RUN dotnet publish -c Release -o /app

FROM base AS production
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "AcrHelloworld.dll"]
```

The application in the *acr-helloworld* image tries to determine the region from which its container was deployed by querying DNS for information about the registry's login server. You must specify your registry login server's fully qualified domain name (FQDN) in the `DOCKER_REGISTRY` environment variable in the Dockerfile.

First, get the registry's login server with the `az acr show` command. Replace `<acrName>` with the name of the registry you created in previous steps.

```
az acr show --name <acrName> --query "{acrLoginServer:loginServer}" --output table
```

Output:

```
AcrLoginServer
----------------------------
uniqueregistryname.azurecr.io
```

Next, update the `ENV DOCKER_REGISTRY` line with the FQDN of your registry's login server. This example reflects the example registry name, *uniqueregistryname*:

```
ENV DOCKER_REGISTRY uniqueregistryname.azurecr.io
```

Build container image

Now that you've updated the Dockerfile with the FQDN of your registry login server, you can use `docker build` to create the container image. Run the following command to build the image and tag it with the URL of your private registry, again replacing `<acrName>` with the name of your registry:

```
docker build . -f ./AcrHelloworld/Dockerfile -t <acrName>.azurecr.io/acr-helloworld:v1
```

Several lines of output are displayed as the Docker image is built (shown here truncated):

```
Sending build context to Docker daemon  523.8kB
Step 1/18 : FROM microsoft/aspnetcore:2.0 AS base
2.0: Pulling from microsoft/aspnetcore
3e17c6eae66c: Pulling fs layer

[...]

Step 18/18 : ENTRYPOINT dotnet AcrHelloworld.dll
 ---> Running in 6906d98c47a1
 ---> c9ca1763cfb1
Removing intermediate container 6906d98c47a1
Successfully built c9ca1763cfb1
Successfully tagged uniqueregistryname.azurecr.io/acr-helloworld:v1
```

Use `docker images` to see the built and tagged image:

```
$ docker images
REPOSITORY                                      TAG     IMAGE ID        CREATED               SIZE
uniqueregistryname.azurecr.io/acr-helloworld    v1      01ac48d5c8cf    About a minute ago    284MB
[...]
```

Push image to Azure Container Registry

Next, use the `docker push` command to push the *acr-helloworld* image to your registry. Replace `<acrName>` with the name of your registry.

```
docker push <acrName>.azurecr.io/acr-helloworld:v1
```

Because you've configured your registry for geo-replication, your image is automatically replicated to both the *West US* and *East US* regions with this single `docker push` command.

```
$ docker push uniqueregistryname.azurecr.io/acr-helloworld:v1
The push refers to a repository [uniqueregistryname.azurecr.io/acr-helloworld]
cd54739c444b: Pushed
d6803756744a: Pushed
b7b1f3a15779: Pushed
a89567dff12d: Pushed
59c7b561ff56: Pushed
9a2f9413d9e4: Pushed
a75caa09eb1f: Pushed
v1: digest: sha256:0799014f91384bda5b87591170b1242bcd719f07a03d1f9a1ddbae72b3543970 size: 1792
```

Next steps

In this tutorial, you created a private, geo-replicated container registry, built a container image, and then pushed that image to your registry.

Advance to the next tutorial to deploy your container to multiple Web Apps for Containers instances, using geo-replication to serve the images locally.

Deploy web app from Azure Container Registry

Tutorial: Deploy a web app from a geo-replicated Azure container registry

11/24/2019 • 4 minutes to read • Edit Online

This is part two in a three-part tutorial series. In part one, a private, geo-replicated container registry was created, and a container image was built from source and pushed to the registry. In this article, you take advantage of the network-close aspect of the geo-replicated registry by deploying the container to Web App instances in two different Azure regions. Each instance then pulls the container image from the closest registry.

In this tutorial, part two in the series:

- Deploy a container image to two *Web Apps for Containers* instances
- Verify the deployed application

If you haven't yet created a geo-replicated registry and pushed the image of the containerized sample application to the registry, return to the previous tutorial in the series, Prepare a geo-replicated Azure container registry.

In the next article in the series, you update the application, then push the updated container image to the registry. Finally, you browse to each running Web App instance to see the change automatically reflected in both, showing Azure Container Registry geo-replication and webhooks in action.

Automatic deployment to Web Apps for Containers

Azure Container Registry provides support for deploying containerized applications directly to Web Apps for Containers. In this tutorial, you use the Azure portal to deploy the container image created in the previous tutorial to two web app plans located in different Azure regions.

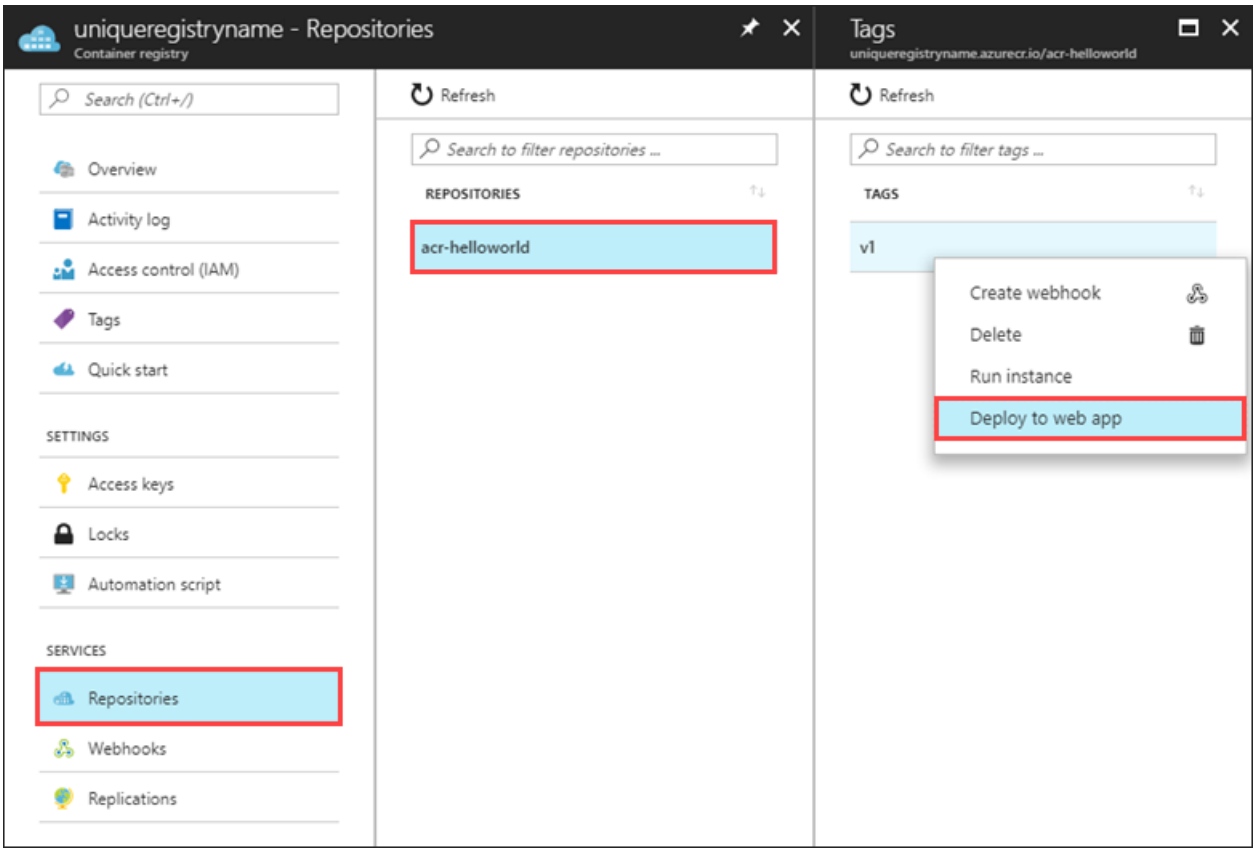
When you deploy a web app from a container image in your registry, and you have a geo-replicated registry in the same region, Azure Container Registry creates an image deployment webhook for you. When you push a new image to your container repository, the webhook picks up the change and automatically deploys the new container image to your web app.

Deploy a Web App for Containers instance

In this step, you create a Web App for Containers instance in the *West US* region.

Sign in to the Azure portal and navigate to the registry you created in the previous tutorial.

Select **Repositories** > **acr-helloworld**, then right-click on the **v1** tag under **Tags** and select **Deploy to web app**:



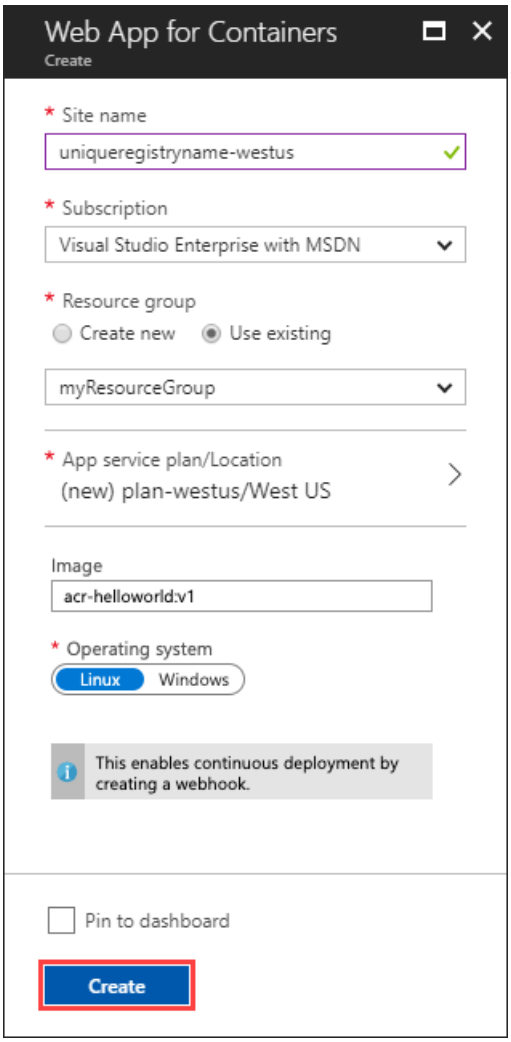
If "Deploy to web app" is disabled, you might not have enabled the registry admin user as directed in Create a container registry in the first tutorial. You can enable the admin user in **Settings** > **Access keys** in the Azure portal.

Under **Web App for Containers** that's displayed after you select "Deploy to web app," specify the following values for each setting:

SETTING	VALUE
Site Name	A globally unique name for the web app. In this example, we use the format `<acrName>-westus` to easily identify the registry and region the web app is deployed from.
Resource Group	**Use existing** > `myResourceGroup`
App service plan/Location	Create a new plan named `plan-westus` in the **West US** region.
Image	`acr-helloworld:v1`
Operating system	Linux

> **NOTE**
> When you create a new app service plan to deploy your containerized app, a default plan is automatically selected to host your application. The default plan depends on the operating system setting.

Select **Create** to provision the web app to the *West US* region.

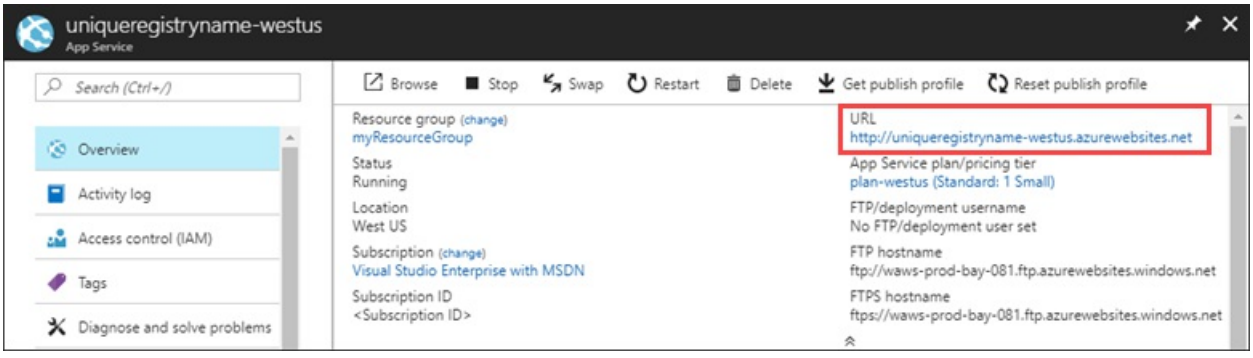


View the deployed web app

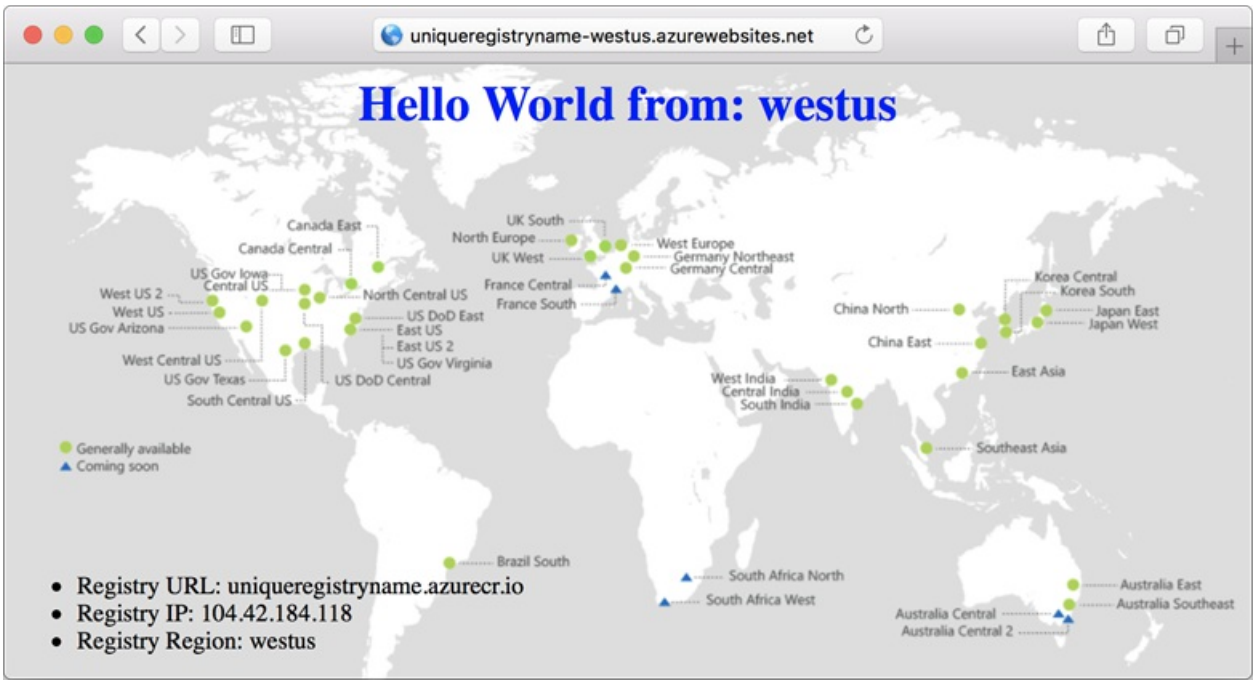
When deployment is complete, you can view the running application by navigating to its URL in your browser.

In the portal, select **App Services**, then the web app you provisioned in the previous step. In this example, the web app is named *uniqueregistryname-westus*.

Select the hyperlinked URL of the web app in the top-right of the **App Service** overview to view the running application in your browser.



Once the Docker image is deployed from your geo-replicated container registry, the site displays an image representing the Azure region hosting the container registry.

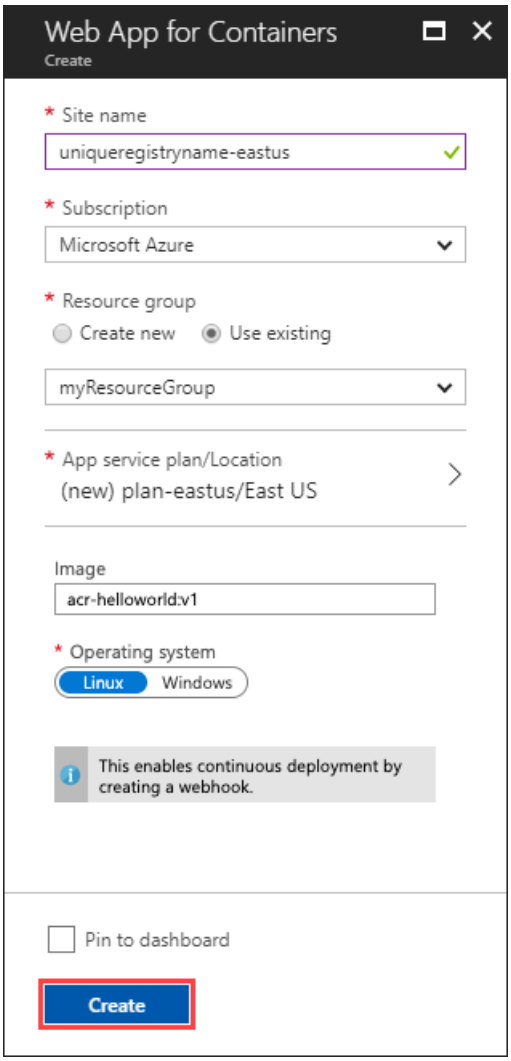


Deploy second Web App for Containers instance

Use the procedure outlined in the previous section to deploy a second web app to the *East US* region. Under **Web App for Containers**, specify the following values:

SETTING	VALUE
Site Name	A globally unique name for the web app. In this example, we use the format `<acrName>-eastus` to easily identify the registry and region the web app is deployed from.
Resource Group	**Use existing** > `myResourceGroup`
App service plan/Location	Create a new plan named `plan-eastus` in the **East US** region.
Image	`acr-helloworld:v1`
Operating system	Linux

Select **Create** to provision the web app to the *East US* region.

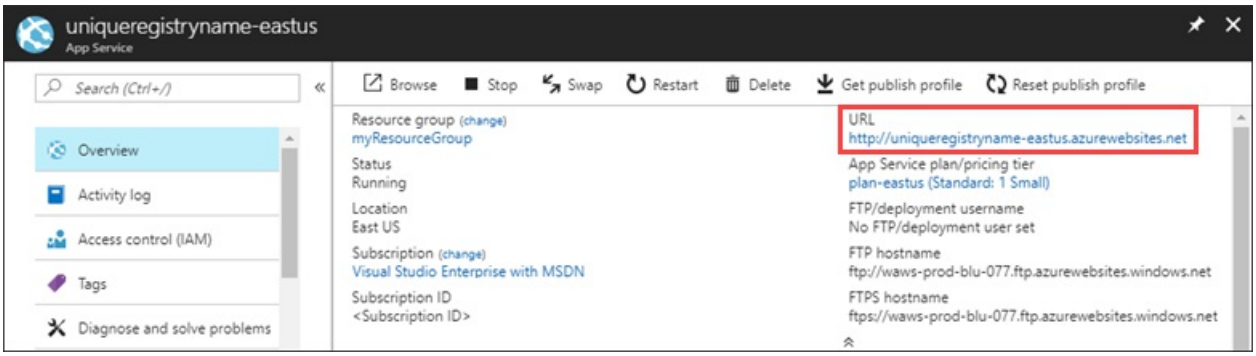


View the deployed web app

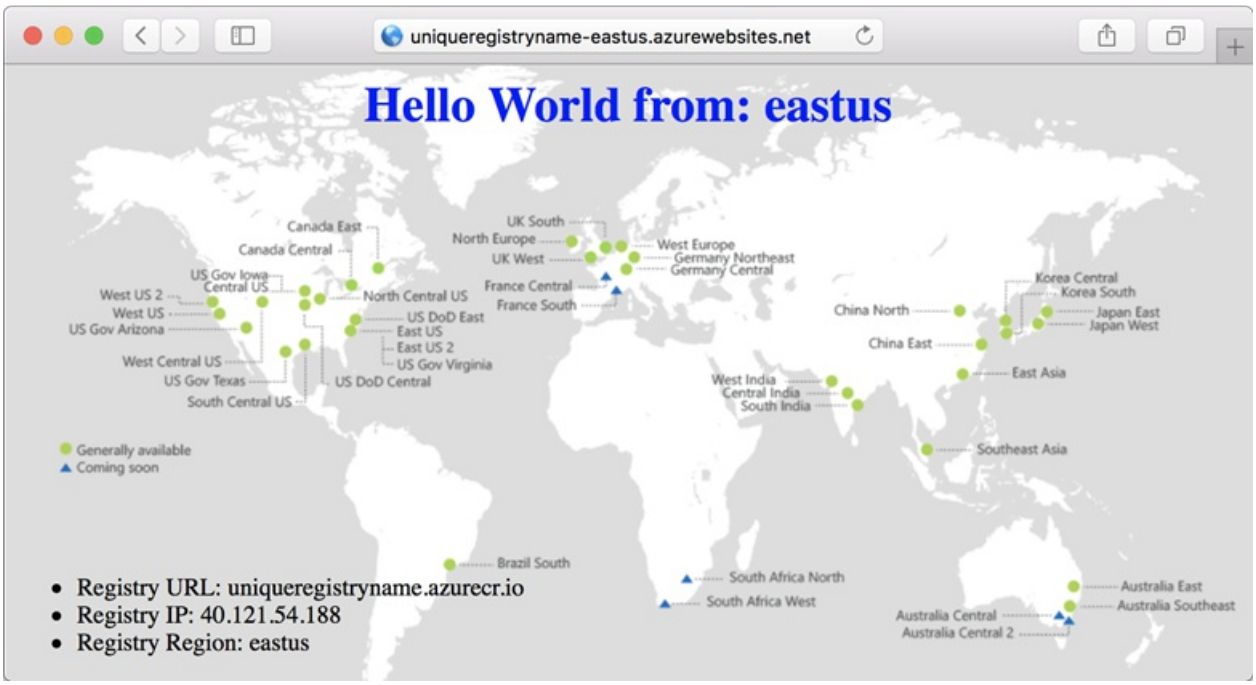
As before, you can view the running application by navigating to its URL in your browser.

In the portal, select **App Services**, then the web app you provisioned in the previous step. In this example, the web app is named *uniqueregistryname-eastus*.

Select the hyperlinked URL of the web app in the top-right of the **App Service overview** to view the running application in your browser.



Once the Docker image is deployed from your geo-replicated container registry, the site displays an image representing the Azure region hosting the container registry.



Next steps

In this tutorial, you deployed two Web App for Containers instances from a geo-replicated Azure container registry.

Advance to the next tutorial to update and then deploy a new container image to the container registry, then verify that the web apps running in both regions were updated automatically.

Deploy an update to geo-replicated container image

Tutorial: Push an updated container image to a geo-replicated container registry for regional web app deployments

11/24/2019 • 3 minutes to read • Edit Online

This is part three in a three-part tutorial series. In the previous tutorial, geo-replication was configured for two different regional Web App deployments. In this tutorial, you first modify the application, then build a new container image and push it to your geo-replicated registry. Finally, you view the change, deployed automatically by Azure Container Registry webhooks, in both Web App instances.

In this tutorial, the final part in the series:

- Modify the web application HTML
- Build and tag the Docker image
- Push the change to Azure Container Registry
- View the updated app in two different regions

If you've not yet configured the two *Web App for Containers* regional deployments, return to the previous tutorial in the series, Deploy web app from Azure Container Registry.

Modify the web application

In this step, make a change to the web application that will be highly visible once you push the updated container image to Azure Container Registry.

Find the `AcrHelloworld/Views/Home/Index.cshtml` file in the application source you cloned from GitHub in a previous tutorial and open it in your favorite text editor. Add the following line below the existing `<h1>` line:

```
<h1>MODIFIED</h1>
```

Your modified `Index.cshtml` should look similar to:

```
@{
    ViewData["Title"] = "Azure Container Registry :: Geo-replication";
}
<style>
    body {
        background-image: url('images/azure-regions.png');
        background-size: cover;
    }
    .footer {
        position: fixed;
        bottom: 0px;
        width: 100%;
    }
</style>

<h1 style="text-align:center;color:blue">Hello World from:  @ViewData["REGION"]</h1>
<h1>MODIFIED</h1>
<div class="footer">
    <ul>
        <li>Registry URL: @ViewData["REGISTRYURL"]</li>
        <li>Registry IP: @ViewData["REGISTRYIP"]</li>
        <li>Registry Region: @ViewData["REGION"]</li>
    </ul>
</div>
```

Rebuild the image

Now that you've updated the web application, rebuild its container image. As before, use the fully qualified image name, including the login server's fully qualified domain name (FQDN), for the tag:

```
docker build . -f ./AcrHelloworld/Dockerfile -t <acrName>.azurecr.io/acr-helloworld:v1
```

Push image to Azure Container Registry

Next, push the updated *acr-helloworld* container image to your geo-replicated registry. Here, you're executing a single `docker push` command to deploy the updated image to the registry replicas in both the *West US* and *East US* regions.

```
docker push <acrName>.azurecr.io/acr-helloworld:v1
```

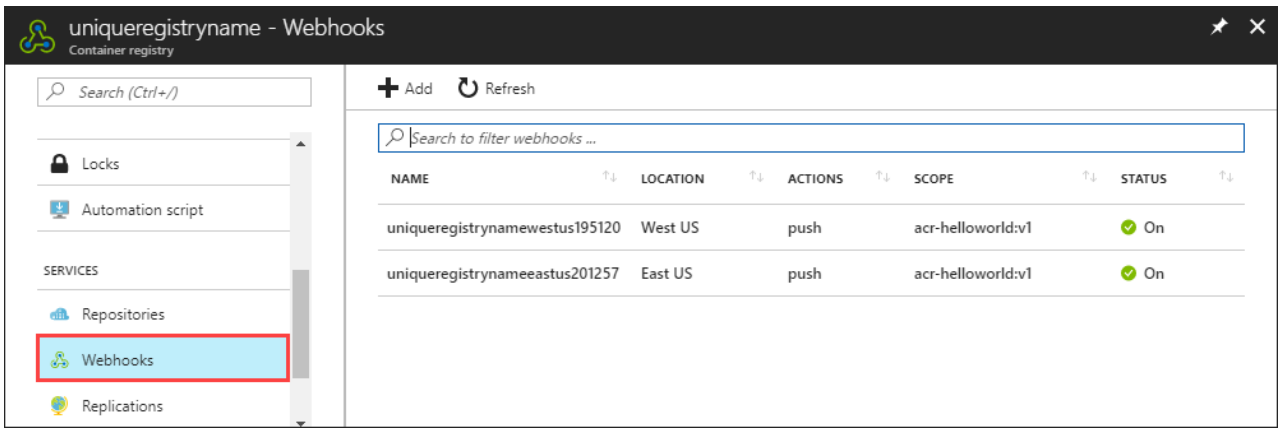
Your `docker push` output should be similar to the following:

```
$ docker push uniqueregistryname.azurecr.io/acr-helloworld:v1
The push refers to a repository [uniqueregistryname.azurecr.io/acr-helloworld]
5b9454e91555: Pushed
d6803756744a: Layer already exists
b7b1f3a15779: Layer already exists
a89567dff12d: Layer already exists
59c7b561ff56: Layer already exists
9a2f9413d9e4: Layer already exists
a75caa09eb1f: Layer already exists
v1: digest: sha256:4c3f2211569346fbe2d1006c18cbea2a4a9dcc1eb3a078608cef70d3a186ec7a size: 1792
```

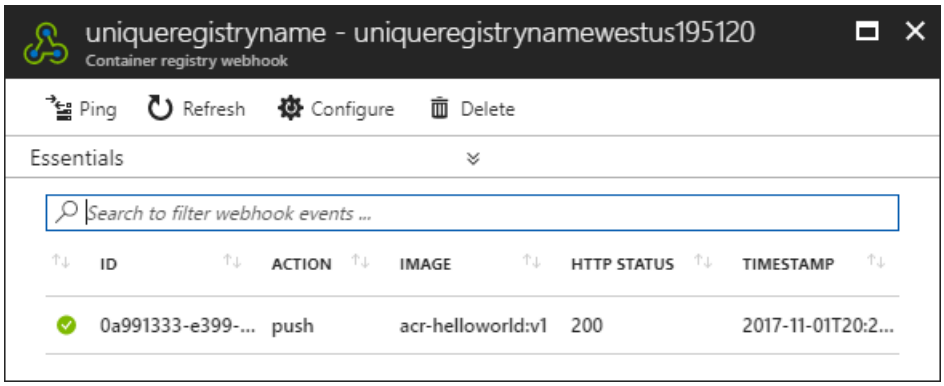
View the webhook logs

While the image is being replicated, you can see the Azure Container Registry webhooks being triggered.

To see the regional webhooks that were created when you deployed the container to *Web Apps for Containers* in a previous tutorial, navigate to your container registry in the Azure portal, then select **Webhooks** under **SERVICES**.



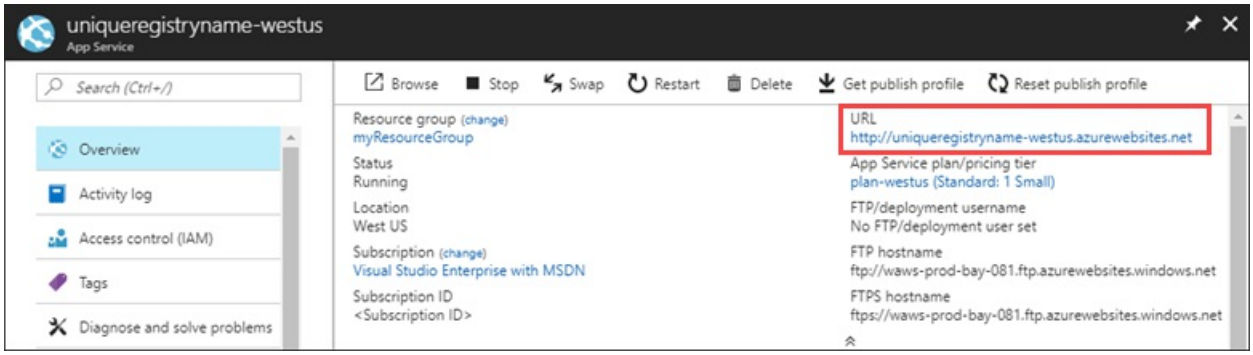
Select each Webhook to see the history of its calls and responses. You should see a row for the **push** action in the logs of both Webhooks. Here, the log for the Webhook located in the *West US* region shows the **push** action triggered by the `docker push` in the previous step:



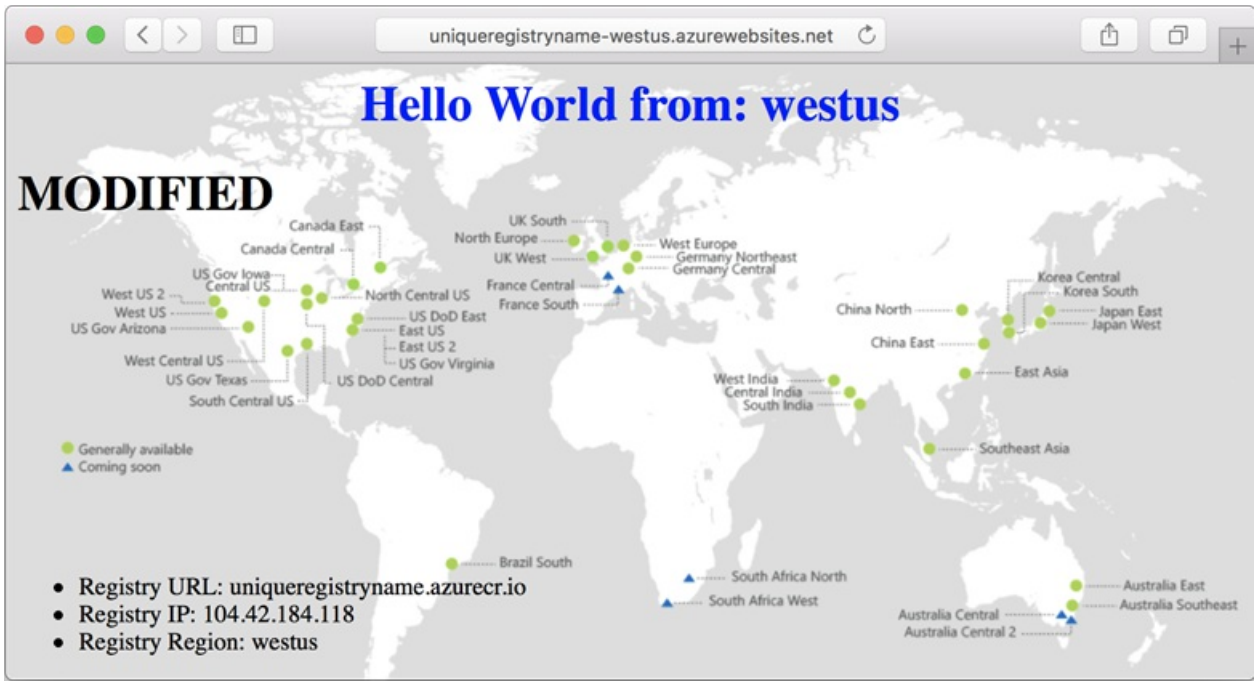
View the updated web app

The Webhooks notify Web Apps that a new image has been pushed to the registry, which automatically deploys the updated container to the two regional web apps.

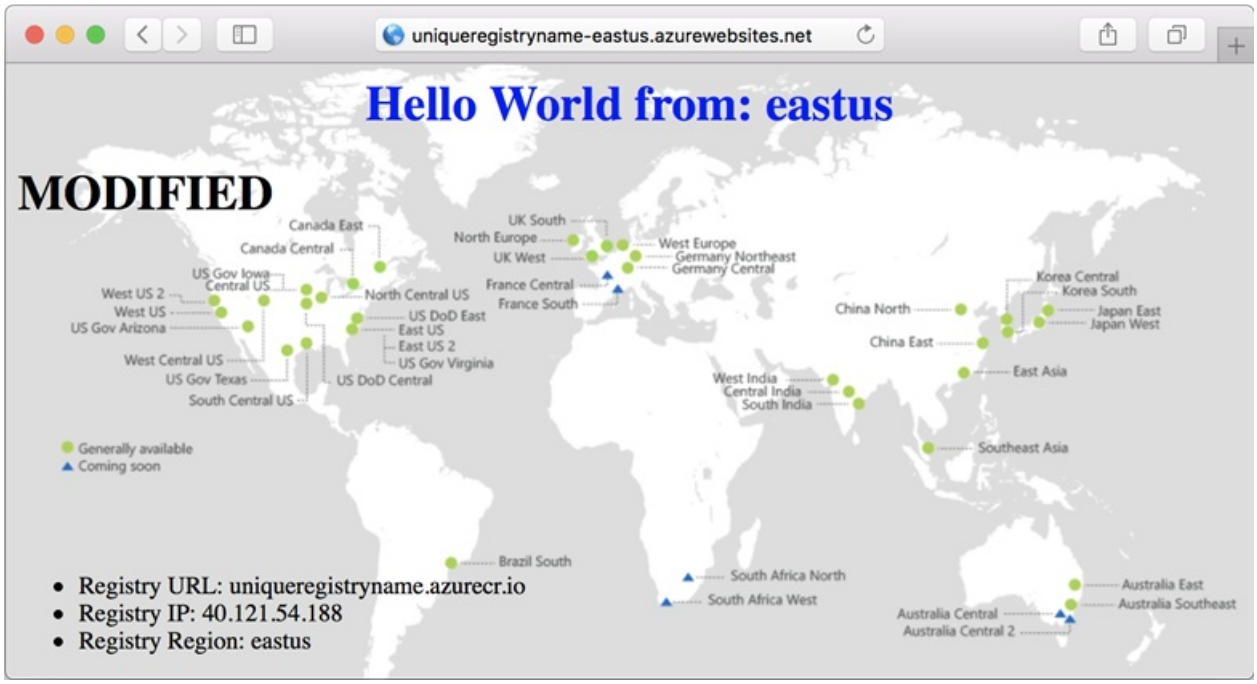
Verify that the application has been updated in both deployments by navigating to both regional Web App deployments in your web browser. As a reminder, you can find the URL for the deployed web app in the top-right of each App Service overview tab.



To see the updated application, select the link in the App Service overview. Here's an example view of the app running in *West US*:



Verify that the updated container image was also deployed to the *East US* deployment by viewing it in your browser.



With a single `docker push`, you've automatically updated the web application running in both regional Web App deployments. And, Azure Container Registry served the container images from the repositories located closest to each deployment.

Next steps

In this tutorial, you updated and pushed a new version of the web application container to your geo-replicated registry. Webhooks in Azure Container Registry notified Web Apps for Containers of the update, which triggered a local pull from the nearest registry replica.

ACR Build: Automated image build and patch

In addition to geo-replication, ACR Build is another feature of Azure Container Registry that can help optimize your container deployment pipeline. Start with the ACR Build overview to get an idea of its capabilities:

Automate OS and framework patching with ACR Build

Azure Container Registry SKUs

11/24/2019 • 3 minutes to read • Edit Online

Azure Container Registry (ACR) is available in multiple service tiers, known as SKUs. These SKUs provide predictable pricing and several options for aligning to the capacity and usage patterns of your private Docker registry in Azure.

SKU	DESCRIPTION
Basic	A cost-optimized entry point for developers learning about Azure Container Registry. Basic registries have the same programmatic capabilities as Standard and Premium (such as Azure Active Directory authentication integration, image deletion, and webhooks). However, the included storage and image throughput are most appropriate for lower usage scenarios.
Standard	Standard registries offer the same capabilities as Basic, with increased included storage and image throughput. Standard registries should satisfy the needs of most production scenarios.
Premium	Premium registries provide the highest amount of included storage and concurrent operations, enabling high-volume scenarios. In addition to higher image throughput, Premium adds features such as geo-replication for managing a single registry across multiple regions, content trust for image tag signing, firewalls and virtual networks (preview) to restrict access to the registry.

The Basic, Standard, and Premium SKUs all provide the same programmatic capabilities. They also all benefit from image storage managed entirely by Azure. Choosing a higher-level SKU provides more performance and scale. With multiple service tiers, you can get started with Basic, then convert to Standard and Premium as your registry usage increases.

SKU features and limits

The following table details the features and limits of the Basic, Standard, and Premium service tiers.

RESOURCE	BASIC	STANDARD	PREMIUM
Storage[1]	10 GiB	100 GiB	500 GiB
Maximum image layer size	200 GiB	200 GiB	200 GiB
ReadOps per minute[2, 3]	1,000	3,000	10,000
WriteOps per minute[2, 4]	100	500	2,000
Download bandwidth MBps[2]	30	60	100

RESOURCE	BASIC	STANDARD	PREMIUM
Upload bandwidth MBps[2]	10	20	50
Webhooks	2	10	500
Geo-replication	N/A	N/A	Supported
Content trust	N/A	N/A	Supported
Virtual network access	N/A	N/A	Preview
Repository-scoped permissions	N/A	N/A	Preview
• Tokens	N/A	N/A	20,000
• Scope maps	N/A	N/A	20,000
• Repositories per scope map	N/A	N/A	500

[1]The specified storage limits are the amount of *included* storage for each tier. You're charged an additional daily rate per GiB for image storage above these limits. For rate information, see Azure Container Registry pricing.

[2]*ReadOps*, *WriteOps*, and *Bandwidth* are minimum estimates. Azure Container Registry strives to improve performance as usage requires.

[3]A docker pull translates to multiple read operations based on the number of layers in the image, plus the manifest retrieval.

[4]A docker push translates to multiple write operations, based on the number of layers that must be pushed. A `docker push` includes *ReadOps* to retrieve a manifest for an existing image.

Changing SKUs

You can change a registry's SKU with the Azure CLI or in the Azure portal. You can move freely between SKUs as long as the SKU you're switching to has the required maximum storage capacity.

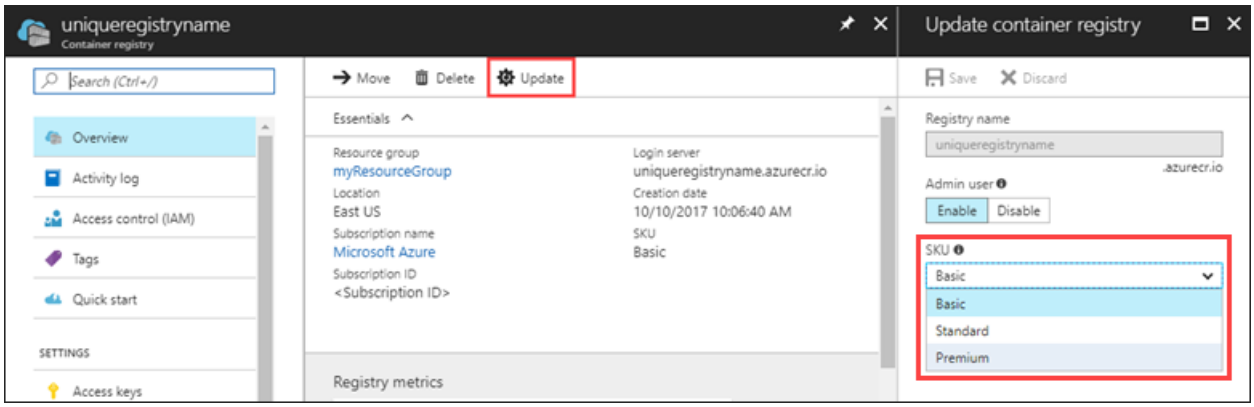
Azure CLI

To move between SKUs in the Azure CLI, use the az acr update command. For example, to switch to Premium:

```
az acr update --name myregistry --sku Premium
```

Azure portal

In the container registry **Overview** in the Azure portal, select **Update**, then select a new **SKU** from the SKU drop-down.



Pricing

For pricing information on each of the Azure Container Registry SKUs, see Container Registry pricing.

For details about pricing for data transfers, see Bandwidth Pricing Details.

Next steps

Azure Container Registry Roadmap

Visit the ACR Roadmap on GitHub to find information about upcoming features in the service.

Azure Container Registry UserVoice

Submit and vote on new feature suggestions in ACR UserVoice.

About registries, repositories, and images

11/24/2019 • 4 minutes to read • Edit Online

This article introduces the key concepts of container registries, repositories, and container images and related artifacts.

Registry

A container *registry* is a service that stores and distributes container images. Docker Hub is a public container registry that supports the open source community and serves as a general catalog of images. Azure Container Registry provides users with direct control of their images, with integrated authentication, geo-replication supporting global distribution and reliability for network-close deployments, virtual network and firewall configuration, tag locking, and many other enhanced features.

In addition to Docker container images, Azure Container Registry supports related content artifacts including Open Container Initiative (OCI) image formats.

Content addressable elements of an artifact

The address of an artifact in an Azure container registry includes the following elements.

```
[loginUrl]/[namespace]/[artifact:][tag]
```

- **loginUrl** - The fully qualified name of the registry host. The registry host in an Azure container registry is in the format *myregistry*.azurecr.io (all lowercase). You must specify the loginUrl when using Docker or other client tools to pull or push artifacts to an Azure container registry.
- **namespace** - Slash-delimited logical grouping of related images or artifacts - for example, for a workgroup or app
- **artifact** - The name of a repository for a particular image or artifact
- **tag** - A specific version of an image or artifact stored in a repository

For example, the full name of an image in an Azure container registry might look like:

```
myregistry.azurecr.io/marketing/campaign10-18/email-sender:v2
```

See the following sections for details about these elements.

Repository name

Container registries manage *repositories*, collections of container images or other artifacts with the same name, but different tags. For example, the following three images are in the "acr-helloworld" repository:

```
acr-helloworld:latest
acr-helloworld:v1
acr-helloworld:v2
```

Repository names can also include namespaces. Namespaces allow you to group images using forward slash-delimited repository names, for example:

```
marketing/campaign10-18/web:v2
marketing/campaign10-18/api:v3
marketing/campaign10-18/email-sender:v2
product-returns/web-submission:20180604
product-returns/legacy-integrator:20180715
```

Image

A container image or other artifact within a registry is associated with one or more tags, has one or more layers, and is identified by a manifest. Understanding how these components relate to each other can help you manage your registry effectively.

Tag

The *tag* for an image or other artifact specifies its version. A single artifact within a repository can be assigned one or many tags, and may also be "untagged." That is, you can delete all tags from an image, while the image's data (its layers) remain in the registry.

The repository (or repository and namespace) plus a tag defines an image's name. You can push and pull an image by specifying its name in the push or pull operation.

How you tag container images is guided by your scenarios to develop or deploy them. For example, stable tags are recommended for maintaining your base images, and unique tags for deploying images. For more information, see Recommendations for tagging and versioning container images.

Layer

Container images are made up of one or more *layers*, each corresponding to a line in the Dockerfile that defines the image. Images in a registry share common layers, increasing storage efficiency. For example, several images in different repositories might share the same Alpine Linux base layer, but only one copy of that layer is stored in the registry.

Layer sharing also optimizes layer distribution to nodes with multiple images sharing common layers. For example, if an image already on a node includes the Alpine Linux layer as its base, the subsequent pull of a different image referencing the same layer doesn't transfer the layer to the node. Instead, it references the layer already existing on the node.

To provide secure isolation and protection from potential layer manipulation, layers are not shared across registries.

Manifest

Each container image or artifact pushed to a container registry is associated with a *manifest*. The manifest, generated by the registry when the image is pushed, uniquely identifies the image and specifies its layers. You can list the manifests for a repository with the Azure CLI command az acr repository show-manifests:

```
az acr repository show-manifests --name <acrName> --repository <repositoryName>
```

For example, list the manifests for the "acr-helloworld" repository:

```
$ az acr repository show-manifests --name myregistry --repository acr-helloworld
[
  {
    "digest": "sha256:0a2e01852872580b2c2fea9380ff8d7b637d3928783c55beb3f21a6e58d5d108",
    "tags": [
      "latest",
      "v3"
    ],
    "timestamp": "2018-07-12T15:52:00.2075864Z"
  },
  {
    "digest": "sha256:3168a21b98836dda7eb7a846b3d735286e09a32b0aa2401773da518e7eba3b57",
    "tags": [
      "v2"
    ],
    "timestamp": "2018-07-12T15:50:53.5372468Z"
  },
  {
    "digest": "sha256:7ca0e0ae50c95155dbb0e380f37d7471e98d2232ed9e31eece9f9fb9078f2728",
    "tags": [
      "v1"
    ],
    "timestamp": "2018-07-11T21:38:35.9170967Z"
  }
]
```

Manifest digest

Manifests are identified by a unique SHA-256 hash, or *manifest digest*. Each image or artifact--whether tagged or not--is identified by its digest. The digest value is unique even if the image's layer data is identical to that of another image. This mechanism is what allows you to repeatedly push identically tagged images to a registry. For example, you can repeatedly push `myimage:latest` to your registry without error because each image is identified by its unique digest.

You can pull an image from a registry by specifying its digest in the pull operation. Some systems may be configured to pull by digest because it guarantees the image version being pulled, even if an identically tagged image is subsequently pushed to the registry.

For example, pull an image from the "acr-helloworld" repository by manifest digest:

```
$ docker pull myregistry.azurecr.io/acr-
helloworld@sha256:0a2e01852872580b2c2fea9380ff8d7b637d3928783c55beb3f21a6e58d5d108
```

IMPORTANT

If you repeatedly push modified images with identical tags, you might create orphaned images--images that are untagged, but still consume space in your registry. Untagged images are not shown in the Azure CLI or in the Azure portal when you list or view images by tag. However, their layers still exist and consume space in your registry. Deleting an untagged image frees registry space when the manifest is the only one, or the last one, pointing to a particular layer. For information about freeing space used by untagged images, see Delete container images in Azure Container Registry.

Next steps

Learn more about image storage and supported content formats in Azure Container Registry.

Container image storage in Azure Container Registry

11/24/2019 • 2 minutes to read • Edit Online

Every Basic, Standard, and Premium Azure container registry benefits from advanced Azure storage features like encryption-at-rest for image data security and geo-redundancy for image data protection. The following sections describe both the features and limits of image storage in Azure Container Registry (ACR).

Encryption-at-rest

All container images in your registry are encrypted at rest. Azure automatically encrypts an image before storing it, and decrypts it on-the-fly when you or your applications and services pull the image.

Geo-redundant storage

Azure uses a geo-redundant storage scheme to guard against loss of your container images. Azure Container Registry automatically replicates your container images to multiple geographically distant data centers, preventing their loss in the event of a regional storage failure.

Geo-replication

For scenarios requiring even more high-availability assurance, consider using the geo-replication feature of Premium registries. Geo-replication helps guard against losing access to your registry in the event of a *total* regional failure, not just a storage failure. Geo-replication provides other benefits, too, like network-close image storage for faster pushes and pulls in distributed development or deployment scenarios.

Image limits

The following table describes the container image and storage limits in place for Azure container registries.

RESOURCE	LIMIT
Repositories	No limit
Images	No limit
Layers	No limit
Tags	No limit
Storage	5 TB

Very high numbers of repositories and tags can impact the performance of your registry. Periodically delete unused repositories, tags, and images as part of your registry maintenance routine. Deleted registry resources like repositories, images, and tags *cannot* be recovered after deletion. For more information about deleting registry resources, see Delete container images in Azure Container Registry.

Storage cost

For full details about pricing, see Azure Container Registry pricing.

Next steps

For more information about the different Azure Container Registry SKUs (Basic, Standard, Premium), see Azure Container Registry SKUs.

Content formats supported in Azure Container Registry

11/24/2019 • 2 minutes to read • Edit Online

Use a private repository in Azure Container Registry to manage one of the following content formats.

Docker-compatible container images

The following Docker container image formats are supported:

- Docker Image Manifest V2, Schema 1
- Docker Image Manifest V2, Schema 2 - includes Manifest Lists which allow registries to store multiplatform images under a single "image:tag" reference

OCI images

Azure Container Registry supports images that meet the Open Container Initiative (OCI) Image Format Specification. Packaging formats include Singularity Image Format (SIF).

OCI artifacts

Azure Container Registry supports the OCI Distribution Specification, a vendor-neutral, cloud-agnostic spec to store, share, secure, and deploy container images and other content types (artifacts). The specification allows a registry to store a wide range of artifacts in addition to container images. You use tooling appropriate to the artifact to push and pull artifacts. For an example, see Push and pull an OCI artifact using an Azure container registry.

To learn more about OCI artifacts, see the OCI Registry as Storage (ORAS) repo and the OCI Artifacts repo on GitHub.

Helm charts

Azure Container Registry can host repositories for Helm charts, a packaging format used to quickly manage and deploy applications for Kubernetes. Helm client version 2 (2.11.0 or later) is supported.

Next steps

- See how to push and pull images with Azure Container Registry.
- Use ACR tasks to build and test container images.
- Use the Moby BuildKit to build and package containers in OCI format.
- Set up a Helm repository hosted in Azure Container Registry.

Recommendations for tagging and versioning container images

12/18/2019 • 4 minutes to read • Edit Online

When pushing deploying container images to a container registry and then deploying them, you need a strategy for image tagging and versioning. This article discusses two approaches and where each fits during the container lifecycle:

- **Stable tags** - Tags that you reuse, for example, to indicate a major or minor version such as *mycontainerimage:1.0*.
- **Unique tags** - A different tag for each image you push to a registry, such as *mycontainerimage:abc123*.

Stable tags

Recommendation: Use stable tags to maintain **base images** for your container builds. Avoid deployments with stable tags, because those tags continue to receive updates and can introduce inconsistencies in production environments.

Stable tags mean a developer, or a build system, can continue to pull a specific tag, which continues to get updates. Stable doesn't mean the contents are frozen. Rather, stable implies the image should be stable for the intent of that version. To stay "stable", it might be serviced to apply security patches or framework updates.

Example

A framework team ships version 1.0. They know they'll ship updates, including minor updates. To support stable tags for a given major and minor version, they have two sets of stable tags.

- `:1` – a stable tag for the major version. `1` represents the "newest" or "latest" 1.* version.
- `:1.0` - a stable tag for version 1.0, allowing a developer to bind to updates of 1.0, and not be rolled forward to 1.1 when it is released.

The team also uses the `:latest` tag, which points to the latest stable tag, no matter what the current major version is.

When base image updates are available, or any type of servicing release of the framework, images with the stable tags are updated to the newest digest that represents the most current stable release of that version.

In this case, both the major and minor tags are continually being serviced. From a base image scenario, this allows the image owner to provide serviced images.

Delete untagged manifests

If an image with a stable tag is updated, the previously tagged image is untagged, resulting in an orphaned image. The previous image's manifest and unique layer data remain in the registry. To maintain your registry size, you can periodically delete untagged manifests resulting from stable image updates. For example, auto-purge untagged manifests older than a specified duration, or set a retention policy for untagged manifests.

Unique tags

Recommendation: Use unique tags for **deployments**, especially in an environment that could scale on multiple nodes. You likely want deliberate deployments of a consistent version of components. If your container restarts or an orchestrator scales out more instances, your hosts won't accidentally pull a newer version, inconsistent with the other nodes.

Unique tagging simply means that every image pushed to a registry has a unique tag. Tags are not reused. There are several patterns you can follow to generate unique tags, including:

- **Date-time stamp** - This approach is fairly common, since you can clearly tell when the image was built. But, how to correlate it back to your build system? Do you have to find the build that was completed at the same time? What time zone are you in? Are all your build systems calibrated to UTC?
- **Git commit** – This approach works until you start supporting base image updates. If a base image update happens, your build system kicks off with the same Git commit as the previous build. However, the base image has new content. In general, a Git commit provides a *semi*-stable tag.
- **Manifest digest** - Each container image pushed to a container registry is associated with a manifest, identified by a unique SHA-256 hash, or digest. While unique, the digest is long, difficult to read, and uncorrelated with your build environment.
- **Build ID** - This option may be best since it's likely incremental, and it allows you to correlate back to the specific build to find all the artifacts and logs. However, like a manifest digest, it might be difficult for a human to read.

  If your organization has several build systems, prefixing the tag with the build system name is a variation on this option: `<build-system>-<build-id>`. For example, you could differentiate builds from the API team's Jenkins build system and the web team's Azure Pipelines build system.

Lock deployed image tags

As a best practice, we recommend that you lock any deployed image tag, by setting its `write-enabled` attribute to `false`. This practice prevents you from inadvertently removing an image from the registry and possibly disrupting your deployments. You can include the locking step in your release pipeline.

Locking a deployed image still allows you to remove other, undeployed images from your registry using Azure Container Registry features to maintain your registry. For example, auto-purge untagged manifests or unlocked images older than a specified duration, or set a retention policy for untagged manifests.

Next steps

For a more detailed discussion of the concepts in this article, see the blog post Docker Tagging: Best practices for tagging and versioning docker images.

To help maximize the performance and cost-effective use of your Azure container registry, see Best practices for Azure Container Registry.

Geo-replication in Azure Container Registry

11/24/2019 • 6 minutes to read • Edit Online

Companies that want a local presence, or a hot backup, choose to run services from multiple Azure regions. As a best practice, placing a container registry in each region where images are run allows network-close operations, enabling fast, reliable image layer transfers. Geo-replication enables an Azure container registry to function as a single registry, serving multiple regions with multi-master regional registries.

A geo-replicated registry provides the following benefits:

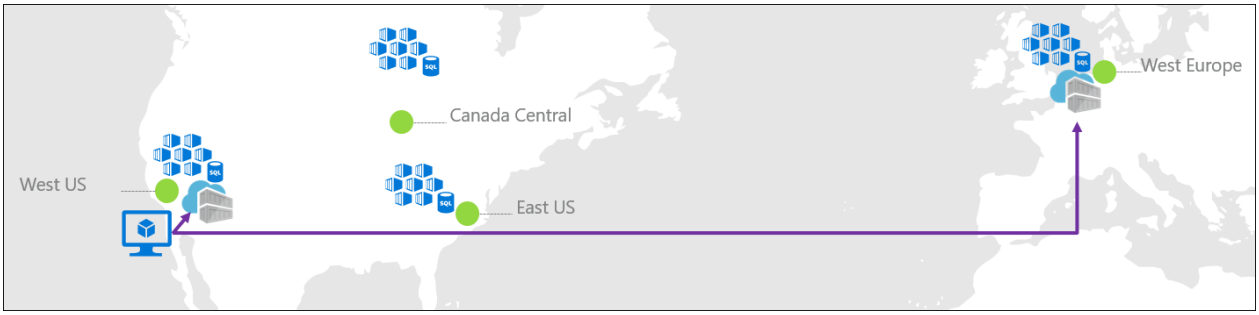
- Single registry/image/tag names can be used across multiple regions
- Network-close registry access from regional deployments
- No additional egress fees, as images are pulled from a local, replicated registry in the same region as your container host
- Single management of a registry across multiple regions

> **NOTE**
> If you need to maintain copies of container images in more than one Azure container registry, Azure Container Registry also supports image import. For example, in a DevOps workflow, you can import an image from a development registry to a production registry, without needing to use Docker commands.

Example use case

Contoso runs a public presence website located across the US, Canada, and Europe. To serve these markets with local and network-close content, Contoso runs Azure Kubernetes Service (AKS) clusters in West US, East US, Canada Central, and West Europe. The website application, deployed as a Docker image, utilizes the same code and image across all regions. Content, local to that region, is retrieved from a database, which is provisioned uniquely in each region. Each regional deployment has its unique configuration for resources like the local database.

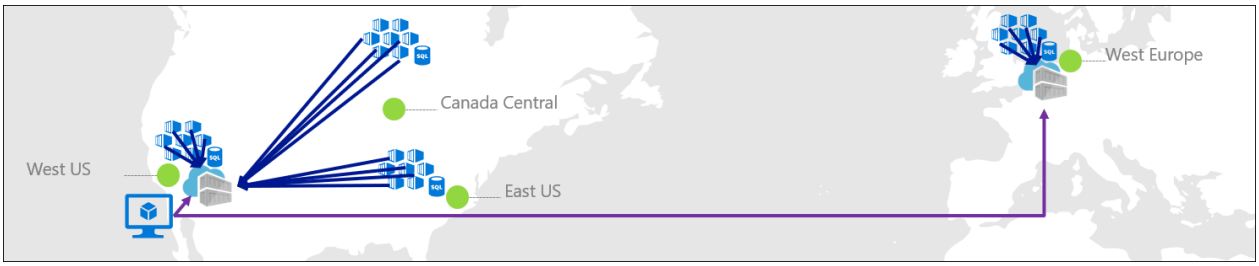
The development team is located in Seattle WA, utilizing the West US data center.



Pushing to multiple registries

Prior to using the geo-replication features, Contoso had a US-based registry in West US, with an additional registry in West Europe. To serve these different regions, the development team pushed images to two different registries.

```
docker push contoso.azurecr.io/public/products/web:1.2
docker push contosowesteu.azurecr.io/public/products/web:1.2
```

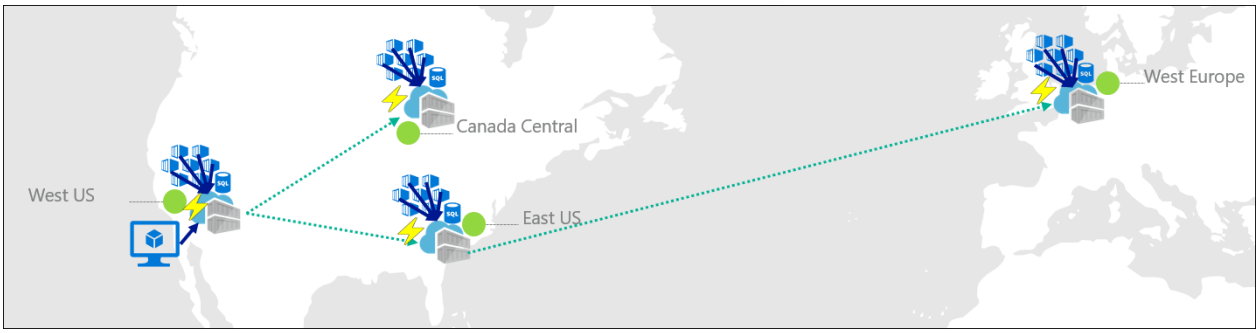


Pulling from multiple registries

Typical challenges of multiple registries include:

- The East US, West US, and Canada Central clusters all pull from the West US registry, incurring egress fees as each of these remote container hosts pull images from West US data centers.
- The development team must push images to West US and West Europe registries.
- The development team must configure and maintain each regional deployment with image names referencing the local registry.
- Registry access must be configured for each region.

Benefits of geo-replication



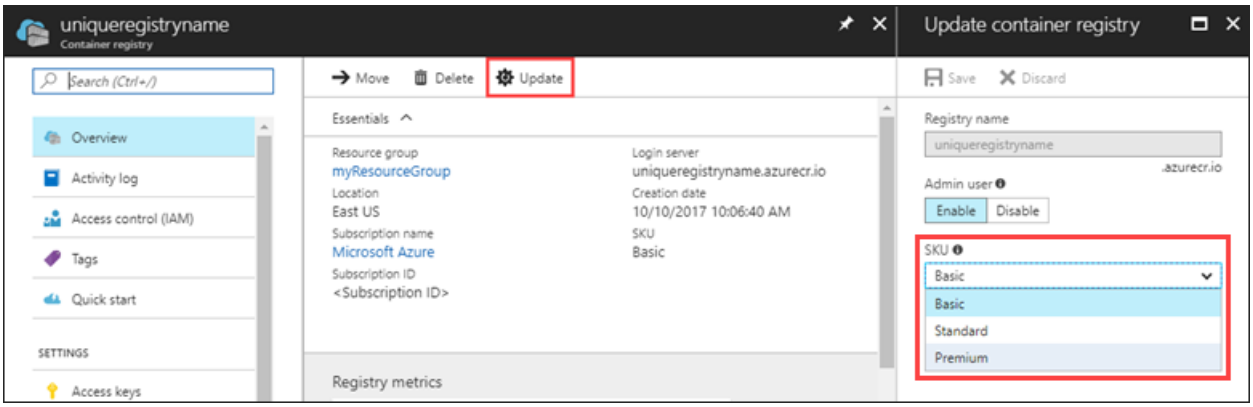
Using the geo-replication feature of Azure Container Registry, these benefits are realized:

- Manage a single registry across all regions: `contoso.azurecr.io`
- Manage a single configuration of image deployments as all regions used the same image URL: `contoso.azurecr.io/public/products/web:1.2`
- Push to a single registry, while ACR manages the geo-replication. You can configure regional webhooks to notify you of events in specific replicas.

Configure geo-replication

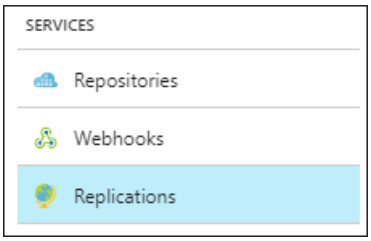
Configuring geo-replication is as easy as clicking regions on a map. You can also manage geo-replication using tools including the az acr replication commands in the Azure CLI, or deploy a registry enabled for geo-replication with an Azure Resource Manager template.

Geo-replication is a feature of Premium registries only. If your registry isn't yet Premium, you can change from Basic and Standard to Premium in the Azure portal:

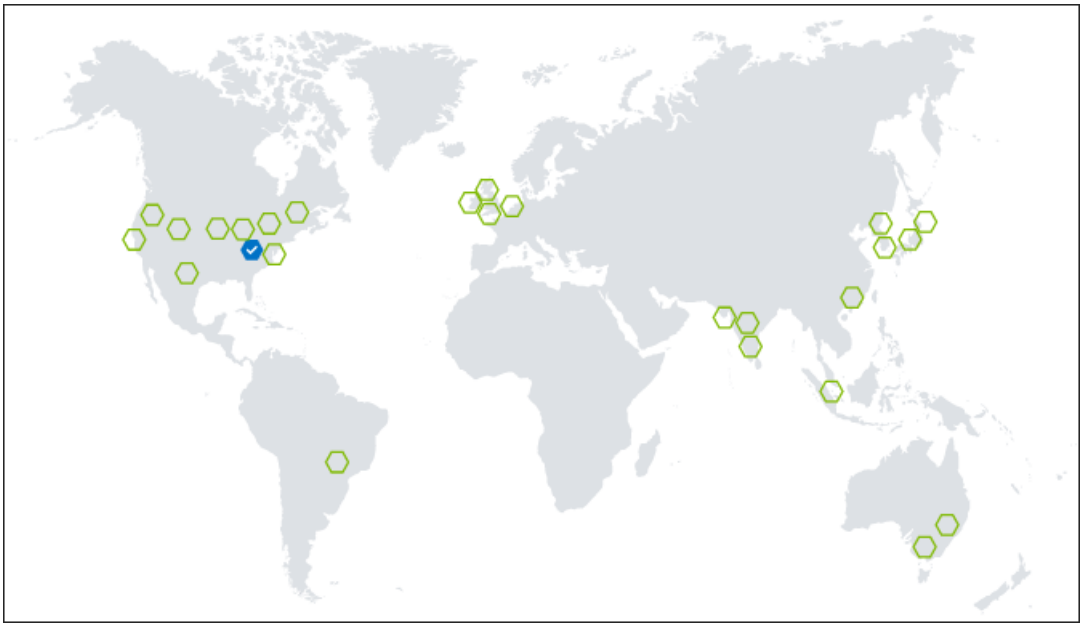


To configure geo-replication for your Premium registry, log in to the Azure portal at https://portal.azure.com.

Navigate to your Azure Container Registry, and select **Replications**:

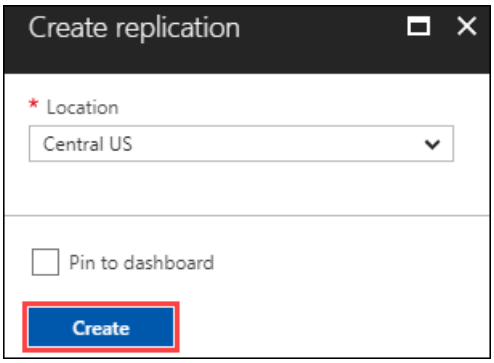


A map is displayed showing all current Azure Regions:



- Blue hexagons represent current replicas
- Green hexagons represent possible replica regions
- Gray hexagons represent Azure regions not yet available for replication

To configure a replica, select a green hexagon, then select **Create**:



To configure additional replicas, select the green hexagons for other regions, then click **Create**.

ACR begins syncing images across the configured replicas. Once complete, the portal reflects *Ready*. The replica status in the portal doesn't automatically update. Use the refresh button to see the updated status.

Considerations for using a geo-replicated registry

- Each region in a geo-replicated registry is independent once set up. Azure Container Registry SLAs apply to each geo-replicated region.
- When you push or pull images from a geo-replicated registry, Azure Traffic Manager in the background sends the request to the registry located in the region closest to you.
- After you push an image or tag update to the closest region, it takes some time for Azure Container Registry to replicate the manifests and layers to the remaining regions you opted into. Larger images take longer to replicate than smaller ones. Images and tags are synchronized across the replication regions with an eventual consistency model.
- To manage workflows that depend on push updates to a geo-replicated , we recommend that you configure webhooks to respond to the push events. You can set up regional webhooks within a geo-replicated registry to track push events as they complete across the geo-replicated regions.

Delete a replica

After you've configured a replica for your registry, you can delete it at any time if it's no longer needed. Delete a replica using the Azure portal or other tools such as the az acr replication delete command in the Azure CLI.

To delete a replica in the Azure portal:

1. Navigate to your Azure Container Registry, and select **Replications**.
2. Select the name of a replica, and select **Delete**. Confirm that you want to delete the replica.

> **NOTE**
> You can't delete the registry replica in the *home region* of the registry, that is, the location where you created the registry. You can only delete the home replica by deleting the registry itself.

Geo-replication pricing

Geo-replication is a feature of the Premium SKU of Azure Container Registry. When you replicate a registry to your desired regions, you incur Premium registry fees for each region.

In the preceding example, Contoso consolidated two registries down to one, adding replicas to East US, Canada Central, and West Europe. Contoso would pay four times Premium per month, with no additional configuration or management. Each region now pulls their images locally, improving performance, reliability without network egress fees from West US to Canada and East US.

Troubleshoot push operations with geo-replicated registries

A Docker client that pushes an image to a geo-replicated registry may not push all image layers and its manifest to a single replicated region. This may occur because Azure Traffic Manager routes registry requests to the network-closest replicated registry. If the registry has two *nearby* replication regions, image layers and the manifest could be distributed to the two sites, and the push operation fails when the manifest is validated. This problem occurs because of the way the DNS name of the registry is resolved on some Linux hosts. This issue doesn't occur on Windows, which provides a client-side DNS cache.

If this problem occurs, one solution is to apply a client-side DNS cache such as `dnsmasq` on the Linux host. This helps ensure that the registry's name is resolved consistently. If you're using a Linux VM in Azure to push to a registry, see options in DNS Name Resolution options for Linux virtual machines in Azure.

To optimize DNS resolution to the closest replica when pushing images, configure a geo-replicated registry in the same Azure regions as the source of the push operations, or the closest region when working outside of Azure.

Next steps

Check out the three-part tutorial series, Geo-replication in Azure Container Registry. Walk through creating a geo-replicated registry, building a container, and then deploying it with a single `docker push` command to multiple regional Web Apps for Containers instances.

Geo-replication in Azure Container Registry

Best practices for Azure Container Registry

12/20/2019 • 3 minutes to read • Edit Online

By following these best practices, you can help maximize the performance and cost-effective use of your private Docker registry in Azure.

See also Recommendations for tagging and versioning container images for strategies to tag and version images in your registry.

Network-close deployment

Create your container registry in the same Azure region in which you deploy containers. Placing your registry in a region that is network-close to your container hosts can help lower both latency and cost.

Network-close deployment is one of the primary reasons for using a private container registry. Docker images have an efficient layering construct that allows for incremental deployments. However, new nodes need to pull all layers required for a given image. This initial `docker pull` can quickly add up to multiple gigabytes. Having a private registry close to your deployment minimizes the network latency. Additionally, all public clouds, Azure included, implement network egress fees. Pulling images from one datacenter to another adds network egress fees, in addition to the latency.

Geo-replicate multi-region deployments

Use Azure Container Registry's geo-replication feature if you're deploying containers to multiple regions. Whether you're serving global customers from local data centers or your development team is in different locations, you can simplify registry management and minimize latency by geo-replicating your registry. Geo-replication is available only with Premium registries.

To learn how to use geo-replication, see the three-part tutorial, Geo-replication in Azure Container Registry.

Repository namespaces

By leveraging repository namespaces, you can allow sharing a single registry across multiple groups within your organization. Registries can be shared across deployments and teams. Azure Container Registry supports nested namespaces, enabling group isolation.

For example, consider the following container image tags. Images that are used corporate-wide, like `aspnetcore`, are placed in the root namespace, while container images owned by the Products and Marketing groups each use their own namespaces.

```
contoso.azurecr.io/aspnetcore:2.0
contoso.azurecr.io/products/widget/web:1
contoso.azurecr.io/products/bettermousetrap/refundapi:12.3
contoso.azurecr.io/marketing/2017-fall/concertpromotions/campaign:218.42
```

Dedicated resource group

Because container registries are resources that are used across multiple container hosts, a registry should reside in its own resource group.

Although you might experiment with a specific host type, such as Azure Container Instances, you'll likely want to

delete the container instance when you're done. However, you might also want to keep the collection of images you pushed to Azure Container Registry. By placing your registry in its own resource group, you minimize the risk of accidentally deleting the collection of images in the registry when you delete the container instance resource group.

Authentication

When authenticating with an Azure container registry, there are two primary scenarios: individual authentication, and service (or "headless") authentication. The following table provides a brief overview of these scenarios, and the recommended method of authentication for each.

TYPE	EXAMPLE SCENARIO	RECOMMENDED METHOD
Individual identity	A developer pulling images to or pushing images from their development machine.	az acr login
Headless/service identity	Build and deployment pipelines where the user isn't directly involved.	Service principal

For in-depth information about Azure Container Registry authentication, see Authenticate with an Azure container registry.

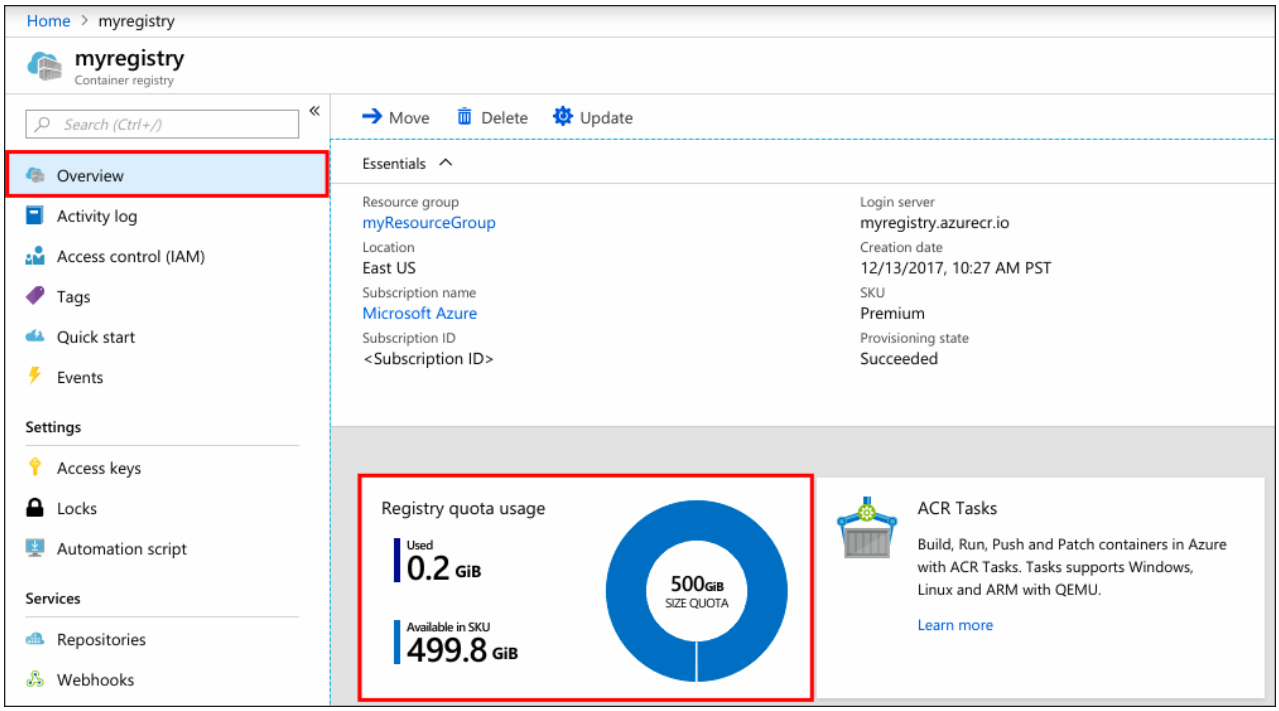
Manage registry size

The storage constraints of each container registry SKU are intended to align with a typical scenario: **Basic** for getting started, **Standard** for the majority of production applications, and **Premium** for hyper-scale performance and geo-replication. Throughout the life of your registry, you should manage its size by periodically deleting unused content.

Use the Azure CLI command az acr show-usage to display the current size of your registry:

```
$ az acr show-usage --resource-group myResourceGroup --name myregistry --output table
NAME      LIMIT         CURRENT VALUE    UNIT
--------  ------------  ---------------  ------
Size      536870912000  185444288        Bytes
Webhooks  100                            Count
```

You can also find the current storage used in the **Overview** of your registry in the Azure portal:



Delete image data

Azure Container Registry supports several methods for deleting image data from your container registry. You can delete images by tag or manifest digest, or delete a whole repository.

For details on deleting image data from your registry, including untagged (sometimes called "dangling" or "orphaned") images, see Delete container images in Azure Container Registry.

Next steps

Azure Container Registry is available in several tiers, called SKUs, that each provide different capabilities. For details on the available SKUs, see Azure Container Registry SKUs.

Push your first image to a private Docker container registry using the Docker CLI

11/24/2019 • 3 minutes to read • Edit Online

An Azure container registry stores and manages private Docker container images, similar to the way Docker Hub stores public Docker images. You can use the Docker command-line interface (Docker CLI) for login, push, pull, and other operations on your container registry.

In the following steps, you download an official Nginx image from the public Docker Hub registry, tag it for your private Azure container registry, push it to your registry, and then pull it from the registry.

Prerequisites

- **Azure container registry** - Create a container registry in your Azure subscription. For example, use the Azure portal or the Azure CLI.
- **Docker CLI** - You must also have Docker installed locally. Docker provides packages that easily configure Docker on any macOS, Windows, or Linux system.

Log in to a registry

There are several ways to authenticate to your private container registry. The recommended method when working in a command line is with the Azure CLI command az acr login. For example, to log in to a registry named *myregistry*:

```
az acr login --name myregistry
```

You can also log in with docker login. For example, you might have assigned a service principal to your registry for an automation scenario. When you run the following command, interactively provide the service principal appID (username) and password when prompted. For best practices to manage login credentials, see the docker login command reference:

```
docker login myregistry.azurecr.io
```

Both commands return `Login Succeeded` once completed.

> **TIP**
> Always specify the fully qualified registry name (all lowercase) when you use `docker login` and when you tag images for pushing to your registry. In the examples in this article, the fully qualified name is *myregistry.azurecr.io*.

Pull the official Nginx image

First, pull the public Nginx image to your local computer.

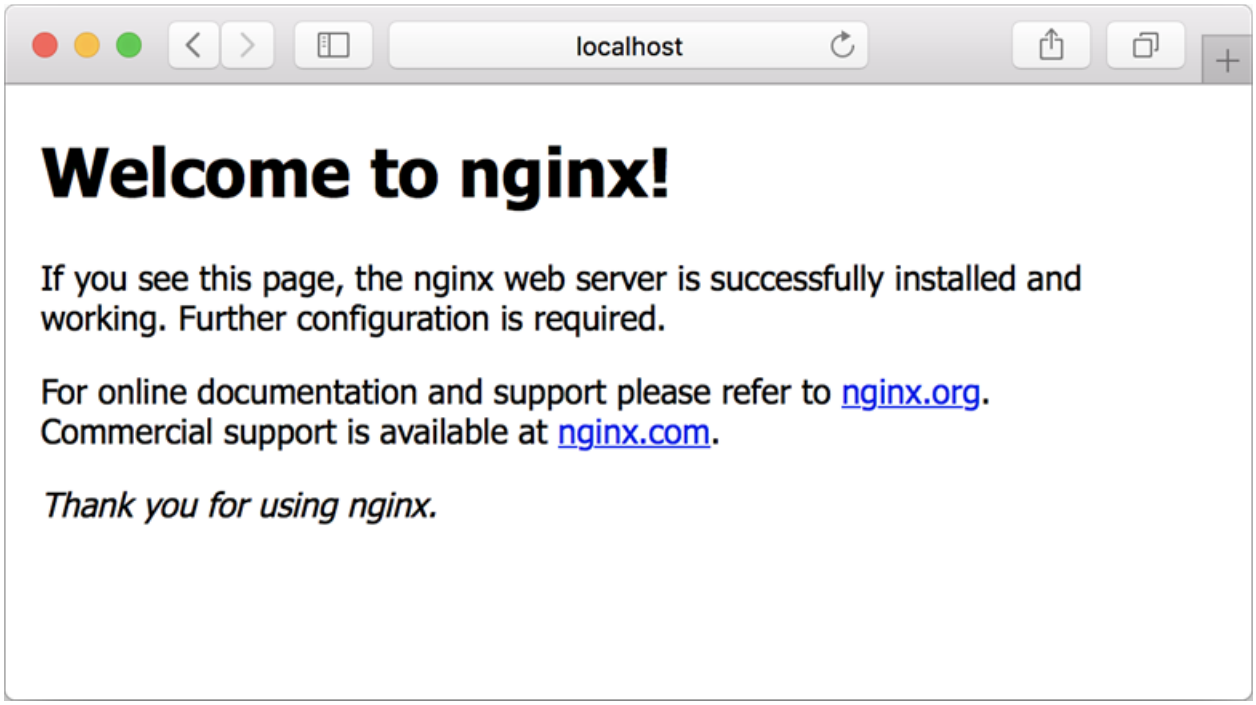
```
docker pull nginx
```

Run the container locally

Execute following docker run command to start a local instance of the Nginx container interactively (`-it`) on port 8080. The `--rm` argument specifies that the container should be removed when you stop it.

```
docker run -it --rm -p 8080:80 nginx
```

Browse to `http://localhost:8080` to view the default web page served by Nginx in the running container. You should see a page similar to the following:



Because you started the container interactively with `-it` , you can see the Nginx server's output on the command line after navigating to it in your browser.

To stop and remove the container, press `Control` + `C` .

Create an alias of the image

Use docker tag to create an alias of the image with the fully qualified path to your registry. This example specifies the `samples` namespace to avoid clutter in the root of the registry.

```
docker tag nginx myregistry.azurecr.io/samples/nginx
```

For more information about tagging with namespaces, see the Repository namespaces section of Best practices for Azure Container Registry.

Push the image to your registry

Now that you've tagged the image with the fully qualified path to your private registry, you can push it to the registry with docker push:

```
docker push myregistry.azurecr.io/samples/nginx
```

Pull the image from your registry

Use the docker pull command to pull the image from your registry:

```
docker pull myregistry.azurecr.io/samples/nginx
```

Start the Nginx container

Use the docker run command to run the image you've pulled from your registry:

```
docker run -it --rm -p 8080:80 myregistry.azurecr.io/samples/nginx
```

Browse to `http://localhost:8080` to view the running container.

To stop and remove the container, press `Control` + `C`.

Remove the image (optional)

If you no longer need the Nginx image, you can delete it locally with the docker rmi command.

```
docker rmi myregistry.azurecr.io/samples/nginx
```

To remove images from your Azure container registry, you can use the Azure CLI command az acr repository delete. For example, the following command deletes the manifest referenced by the `samples/nginx:latest` tag, any unique layer data, and all other tags referencing the manifest.

```
az acr repository delete --name myregistry --image samples/nginx:latest
```

Next steps

Now that you know the basics, you're ready to start using your registry! For example, deploy container images from your registry to:

- Azure Kubernetes Service (AKS)
- Azure Container Instances
- Service Fabric

Optionally install the Docker Extension for Visual Studio Code and the Azure Account extension to work with your Azure container registries. Pull and push images to an Azure container registry, or run ACR Tasks, all within Visual Studio Code.

Push and pull an OCI artifact using an Azure container registry

11/24/2019 • 3 minutes to read • Edit Online

You can use an Azure container registry to store and manage Open Container Initiative (OCI) artifacts as well as Docker and Docker-compatible container images.

To demonstrate this capability, this article shows how to use the OCI Registry as Storage (ORAS) tool to push a sample artifact - a text file - to an Azure container registry. Then, pull the artifact from the registry. You can manage a variety of OCI artifacts in an Azure container registry using different command-line tools appropriate to each artifact.

Prerequisites

- **Azure container registry** - Create a container registry in your Azure subscription. For example, use the Azure portal or the Azure CLI.
- **ORAS tool** - Download and install a current ORAS release for your operating system from the GitHub repo. The tool is released as a compressed tarball (`.tar.gz` file). Extract and install the file using standard procedures for your operating system.
- **Azure Active Directory service principal (optional)** - To authenticate directly with ORAS, create a service principal to access your registry. Ensure that the service principal is assigned a role such as AcrPush so that it has permissions to push and pull artifacts.
- **Azure CLI (optional)** - To use an individual identity, you need a local installation of the Azure CLI. Version 2.0.71 or later is recommended. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.
- **Docker (optional)** - To use an individual identity, you must also have Docker installed locally, to authenticate with the registry. Docker provides packages that easily configure Docker on any macOS, Windows, or Linux system.

Sign in to a registry

This section shows two suggested workflows to sign into the registry, depending on the identity used. Choose the method appropriate for your environment.

Sign in with ORAS

Using a service principal with push rights, run the `oras login` command to sign in to the registry using the service principal application ID and password. Specify the fully qualified registry name (all lowercase), in this case *myregistry.azurecr.io*. The service principal application ID is passed in the environment variable `$SP_APP_ID`, and the password in the variable `$SP_PASSWD`.

```
oras login myregistry.azurecr.io --username $SP_APP_ID --password $SP_PASSWD
```

To read the password from Stdin, use `--password-stdin`.

Sign in with Azure CLI

Sign in to the Azure CLI with your identity to push and pull artifacts from the container registry.

Then, use the Azure CLI command az acr login to access the registry. For example, to authenticate to a registry

named *myregistry*:

```
az login
az acr login --name myregistry
```

> **NOTE**
> `az acr login` uses the Docker client to set an Azure Active Directory token in the `docker.config` file. The Docker client must be installed and running to complete the individual authentication flow.

Push an artifact

Create a text file in a local working working directory with some sample text. For example, in a bash shell:

```
echo "Here is an artifact!" > artifact.txt
```

Use the `oras push` command to push this text file to your registry. The following example pushes the sample text file to the `samples/artifact` repo. The registry is identified with the fully qualified registry name *myregistry.azurecr.io* (all lowercase). The artifact is tagged `1.0`. The artifact has an undefined type, by default, identified by the *media type* string following the filename `artifact.txt`. See OCI Artifacts for additional types.

```
oras push myregistry.azurecr.io/samples/artifact:1.0 \
    --manifest-config /dev/null:application/vnd.unknown.config.v1+json \
    ./artifact.txt:application/vnd.unknown.layer.v1+txt
```

Output for a successful push is similar to the following:

```
Uploading 33998889555f artifact.txt
Pushed myregistry.azurecr.io/samples/artifact:1.0
Digest: sha256:xxxxxxbc912ef63e69136f05f1078dbf8d00960a79ee73c210eb2a5f65xxxxxx
```

To manage artifacts in your registry, if you are using the Azure CLI, run standard `az acr` commands for managing images. For example, get the attributes of the artifact using the az acr repository show command:

```
az acr repository show \
    --name myregistry \
    --image samples/artifact:1.0
```

Output is similar to the following:

```
{
  "changeableAttributes": {
    "deleteEnabled": true,
    "listEnabled": true,
    "readEnabled": true,
    "writeEnabled": true
  },
  "createdTime": "2019-08-28T20:43:31.0001687Z",
  "digest": "sha256:xxxxxxbc912ef63e69136f05f1078dbf8d00960a79ee73c210eb2a5f65xxxxxx",
  "lastUpdateTime": "2019-08-28T20:43:31.0001687Z",
  "name": "1.0",
  "signed": false
}
```

Pull an artifact

Run the `oras pull` command to pull the artifact from your registry.

First remove the text file from your local working directory:

```
rm artifact.txt
```

Run `oras pull` to pull the artifact, and specify the media type used to push the artifact:

```
oras pull myregistry.azurecr.io/samples/artifact:1.0 \
    --media-type application/vnd.unknown.layer.v1+txt
```

Verify that the pull was successful:

```
$ cat artifact.txt
Here is an artifact!
```

Remove the artifact (optional)

To remove the artifact from your Azure container registry, use the az acr repository delete command. The following example removes the artifact you stored there:

```
az acr repository delete \
    --name myregistry \
    --image samples/artifact:1.0
```

Next steps

- Learn more about the ORAS Library, including how to configure a manifest for an artifact
- Visit the OCI Artifacts repo for reference information about new artifact types

Push and pull Helm charts to an Azure container registry

2/21/2020 • 12 minutes to read • Edit Online

To quickly manage and deploy applications for Kubernetes, you can use the open-source Helm package manager. With Helm, application packages are defined as charts, which are collected and stored in a Helm chart repository.

This article shows you how to host Helm charts in repositories in an Azure container registry, using either a Helm 3 or Helm 2 installation. For this example, you store an existing Helm chart from the public Helm *stable* repo. In many scenarios, you would build and upload your own charts for the applications you develop. For more information on how to build your own Helm charts, see the Chart Template Developer's Guide.

> **IMPORTANT**
> Support for Helm charts in Azure Container Registry is currently in preview. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

Helm 3 or Helm 2?

To store, manage, and install Helm charts, you use a Helm client and the Helm CLI. Major releases of the Helm client include Helm 3 and Helm 2. Helm 3 supports a new chart format and no longer installs the Tiller server-side component. For details on the version differences, see the version FAQ. If you've previously deployed Helm 2 charts, see Migrating Helm v2 to v3.

You can use either Helm 3 or Helm 2 to host Helm charts in Azure Container Registry, with workflows specific to each version:

- Helm 3 client - use `helm chart` commands to manage charts in your registry as OCI artifacts
- Helm 2 client - use az acr helm commands in the Azure CLI to add and manage your container registry as a Helm chart repository

Additional information

- We recommend using the Helm 3 workflow with native `helm chart` commands to manage charts as OCI artifacts.
- You can use legacy az acr helm Azure CLI commands and workflow with the Helm 3 client and charts. However, certain commands such as `az acr helm list` aren't compatible with Helm 3 charts.
- As of Helm 3, az acr helm commands are supported mainly for compatibility with the Helm 2 client and chart format. Future development of these commands isn't currently planned.

Use the Helm 3 client

Prerequisites

- **An Azure container registry** in your Azure subscription. If needed, create a registry using the Azure portal or the Azure CLI.
- **Helm client version 3.0.0 or later** - Run `helm version` to find your current version. For more information on how to install and upgrade Helm, see Installing Helm.
- **A Kubernetes cluster** where you will install a Helm chart. If needed, create an Azure Kubernetes Service

cluster.

- **Azure CLI version 2.0.71 or later** - Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

High level workflow

With **Helm 3** you:

- Can create one or more Helm repositories in an Azure container registry
- Store Helm 3 charts in a registry as OCI artifacts. Currently, Helm 3 support for OCI is considered *experimental*.
- Use `helm chart` commands directly from the Helm CLI to push, pull, and manage Helm charts in a registry
- Authenticate with your registry via the Azure CLI, which then updates your Helm client automatically with the registry URI and credentials. You don't need to manually specify this registry information, so the credentials aren't exposed in the command history.
- Use `helm install` to install charts to a Kubernetes cluster from a local repository cache.

See the following sections for examples.

Enable OCI support

Set the following environment variable to enable OCI support in the Helm 3 client. Currently, this support is experimental.

```
export HELM_EXPERIMENTAL_OCI=1
```

Pull an existing Helm package

If you haven't already added the `stable` Helm chart repo, run the `helm repo add` command:

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com
```

Pull a chart package from the `stable` repo locally. For example, create a local directory such as *~/acr-helm*, then download the existing *stable/wordpress* chart package. (This example and other commands in this article are formatted for the Bash shell.)

```
mkdir ~/acr-helm && cd ~/acr-helm
helm pull stable/wordpress --untar
```

The `helm pull stable/wordpress` command didn't specify a particular version, so the *latest* version was pulled and uncompressed in the `wordpress` subdirectory.

Save chart to local registry cache

Change directory to the `wordpress` subdirectory, which contains the Helm chart files. Then, run `helm chart save` to save a copy of the chart locally and also create an alias with the fully qualified name of the registry and the target repository and tag.

In the following example, the registry name is *mycontainerregistry*, the target repo is *wordpress*, and the target chart tag is *latest*, but substitute values for your environment:

```
cd wordpress
helm chart save . wordpress:latest
helm chart save . mycontainerregistry.azurecr.io/helm/wordpress:latest
```

Run `helm chart list` to confirm you saved the charts in the local registry cache. Output is similar to:

```
REF                                                 NAME           VERSION DIGEST  SIZE
CREATED
wordpress:latest                                    wordpress      8.1.0   5899db0 29.1 KiB        1 day
mycontainerregistry.azurecr.io/helm/wordpress:latest    wordpress      8.1.0   5899db0 29.1 KiB        1 day
```

Push chart to Azure Container Registry

Run the `helm chart push` command in the Helm 3 CLI to push the Helm chart to a repository in your Azure container registry. If it doesn't exist, the repository is created.

First use the Azure CLI command az acr login to authenticate to your registry:

```
az acr login --name mycontainerregistry
```

Push the chart to the fully qualified target repository:

```
helm chart push mycontainerregistry.azurecr.io/helm/wordpress:latest
```

After a successful push, output is similar to:

```
The push refers to repository [mycontainerregistry.azurecr.io/helm/wordpress]
ref:     mycontainerregistry.azurecr.io/helm/wordpress:latest
digest:  5899db028dcf96aeaabdadfa5899db025899db025899db025899db025899db02
size:    29.1 KiB
name:    wordpress
version: 8.1.0
```

List charts in the repository

As with images stored in an Azure container registry, you can use az acr repository commands to show the repositories hosting your charts, and chart tags and manifests.

For example, run az acr repository show to see the properties of the repo you created in the previous step:

```
az acr repository show \
  --name mycontainerregistry \
  --repository helm/wordpress
```

Output is similar to:

```
{
  "changeableAttributes": {
    "deleteEnabled": true,
    "listEnabled": true,
    "readEnabled": true,
    "writeEnabled": true
  },
  "createdTime": "2020-01-29T16:54:30.1514833Z",
  "imageName": "helm/wordpress",
  "lastUpdateTime": "2020-01-29T16:54:30.4992247Z",
  "manifestCount": 1,
  "registry": "mycontainerregistry.azurecr.io",
  "tagCount": 1
}
```

Run the az acr repository show-manifests command to see details of the chart stored in the repository. For example:

```
az acr repository show-manifests \
  --name mycontainerregistry \
  --repository helm/wordpress --detail
```

Output, abbreviated in this example, shows a `configMediaType` of `application/vnd.cncf.helm.config.v1+json`:

```
[
  {
    [...]
    "configMediaType": "application/vnd.cncf.helm.config.v1+json",
    "createdTime": "2020-01-29T16:54:30.2382436Z",
    "digest": "sha256:xxxxxxxx51bc0807bfa97cb647e493ac381b96c1f18749b7388c24bbxxxxxxxxx",
    "imageSize": 29995,
    "lastUpdateTime": "2020-01-29T16:54:30.3492436Z",
    "mediaType": "application/vnd.oci.image.manifest.v1+json",
    "tags": [
      "latest"
    ]
  }
]
```

Pull chart to local cache

To install a Helm chart to Kubernetes, the chart must be in the local cache. In this example, first run `helm chart remove` to remove the existing local chart named `mycontainerregistry.azurecr.io/helm/wordpress:latest`:

```
helm chart remove mycontainerregistry.azurecr.io/helm/wordpress:latest
```

Run `helm chart pull` to download the chart from the Azure container registry to your local cache:

```
helm chart pull mycontainerregistry.azurecr.io/helm/wordpress:latest
```

Export Helm chart

To work further with the chart, export it to a local directory using `helm chart export`. For example, export the chart you pulled to the `install` directory:

```
helm chart export mycontainerregistry.azurecr.io/helm/wordpress:latest --destination ./install
```

To view information for the exported chart in the repo, run the `helm inspect chart` command in the directory where you exported the chart.

```
cd install
helm inspect chart wordpress
```

When no version number is provided, the *latest* version is used. Helm returns detailed information about your chart, as shown in the following condensed output:

```
apiVersion: v1
appVersion: 5.3.2
dependencies:
- condition: mariadb.enabled
  name: mariadb
  repository: https://kubernetes-charts.storage.googleapis.com/
  tags:
  - wordpress-database
  version: 7.x.x
description: Web publishing platform for building blogs and websites.
home: http://www.wordpress.com/
icon: https://bitnami.com/assets/stacks/wordpress/img/wordpress-stack-220x234.png
keywords:
- wordpress
- cms
- blog
- http
- web
- application
- php
maintainers:
- email: containers@bitnami.com
  name: Bitnami
name: wordpress
sources:
- https://github.com/bitnami/bitnami-docker-wordpress
version: 8.1.0
```

Install Helm chart

Run `helm install` to install the Helm chart you pulled to the local cache and exported. Specify a release name or pass the `--generate-name` parameter. For example:

```
helm install wordpress --generate-name
```

As the installation proceeds, follow the instructions in the command output to see the WorPress URLs and credentials. You can also run the `kubectl get pods` command to see the Kubernetes resources deployed through the Helm chart:

```
NAME                                READY   STATUS    RESTARTS   AGE
wordpress-1598530621-67c77b6d86-7ldv4   1/1     Running   0          2m48s
wordpress-1598530621-mariadb-0          1/1     Running   0          2m48s
[...]
```

Delete a Helm chart from the repository

To delete a chart from the repository, use the az acr repository delete command. Run the following command and confirm the operation when prompted:

```
az acr repository delete --name mycontainerregistry --image helm/wordpress:latest
```

Use the Helm 2 client

Prerequisites

- **An Azure container registry** in your Azure subscription. If needed, create a registry using the Azure portal or the Azure CLI.
- **Helm client version 2.11.0 (not an RC version) or later** - Run `helm version` to find your current version.

You also need a Helm server (Tiller) initialized within a Kubernetes cluster. If needed, create an Azure Kubernetes Service cluster. For more information on how to install and upgrade Helm, see Installing Helm.

- **Azure CLI version 2.0.46 or later** - Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

High level workflow

With **Helm 2** you:

- Configure your Azure container registry as a *single* Helm chart repository. Azure Container Registry manages the index definition as you add and remove charts to the repository.
- Use the az acr helm commands in the Azure CLI to add your Azure container registry as a Helm chart repository, and to push and manage charts. These Azure CLI commands wrap Helm 2 client commands.
- Add the chart repository in your Azure container registry to your local Helm repo index, supporting chart search
- Authenticate with your Azure container registry via the Azure CLI, which then updates your Helm client automatically with the registry URI and credentials. You don't need to manually specify this registry information, so the credentials aren't exposed in the command history.
- Use `helm install` to install charts to a Kubernetes cluster from a local repository cache.

See the following sections for examples.

Add repository to Helm client

Add your Azure Container Registry Helm chart repository to your Helm client using the az acr helm repo add command. This command gets an authentication token for your Azure container registry that is used by the Helm client. The authentication token is valid for 3 hours. Similar to `docker login`, you can run this command in future CLI sessions to authenticate your Helm client with your Azure Container Registry Helm chart repository:

```
az acr helm repo add --name mycontainerregistry
```

Add a chart to the repository

First, create a local directory at *~/acr-helm*, then download the existing *stable/wordpress* chart:

```
mkdir ~/acr-helm && cd ~/acr-helm
helm repo update
helm fetch stable/wordpress
```

Type `ls` to list the downloaded chart, and note the Wordpress version included in the filename. The `helm fetch stable/wordpress` command didn't specify a particular version, so the *latest* version was fetched. In the following example output, the Wordpress chart is version *8.1.0*:

```
wordpress-8.1.0.tgz
```

Push the chart to your Helm chart repository in Azure Container Registry using the az acr helm push command in the Azure CLI. Specify the name of your Helm chart downloaded in the previous step, such as *wordpress-8.1.0.tgz*:

```
az acr helm push --name mycontainerregistry wordpress-8.1.0.tgz
```

After a few moments, the Azure CLI reports that your chart is saved, as shown in the following example output:

```
{
  "saved": true
}
```

List charts in the repository

To use the chart uploaded in the previous step, the local Helm repository index must be updated. You can reindex the repositories in the Helm client, or use the Azure CLI to update the repository index. Each time you add a chart to your repository, this step must be completed:

```
az acr helm repo add --name mycontainerregistry
```

With a chart stored in your repository and the updated index available locally, you can use the regular Helm client commands to search or install. To see all the charts in your repository, use the `helm search` command, providing your own Azure Container Registry name:

```
helm search mycontainerregistry
```

The Wordpress chart pushed in the previous step is listed, as shown in the following example output:

```
NAME                CHART VERSION APP VERSION DESCRIPTION
helmdocs/wordpress 8.1.0        5.3.2       Web publishing platform for building blogs and websites.
```

You can also list the charts with the Azure CLI, using az acr helm list:

```
az acr helm list --name mycontainerregistry
```

Show information for a Helm chart

To view information for a specific chart in the repo, you can use the `helm inspect` command.

```
helm inspect mycontainerregistry/wordpress
```

When no version number is provided, the *latest* version is used. Helm returns detailed information about your chart, as shown in the following condensed example output:

```
apiVersion: v1
appVersion: 5.3.2
description: Web publishing platform for building blogs and websites.
engine: gotpl
home: http://www.wordpress.com/
icon: https://bitnami.com/assets/stacks/wordpress/img/wordpress-stack-220x234.png
keywords:
- wordpress
- cms
- blog
- http
- web
- application
- php
maintainers:
- email: containers@bitnami.com
  name: Bitnami
name: wordpress
sources:
- https://github.com/bitnami/bitnami-docker-wordpress
version: 8.1.0
[...]
```

You can also show the information for a chart with the Azure CLI az acr helm show command. Again, the *latest* version of a chart is returned by default. You can append `--version` to list a specific version of a chart, such as *8.1.0*:

```
az acr helm show --name mycontainerregistry wordpress
```

Install a Helm chart from the repository

The Helm chart in your repository is installed by specifying the repository name and the chart name. Use the Helm client to install the Wordpress chart:

```
helm install mycontainerregistry/wordpress
```

> **TIP**
> If you push to your Azure Container Registry Helm chart repository and later return in a new CLI session, your local Helm client needs an updated authentication token. To obtain a new authentication token, use the az acr helm repo add command.

The following steps are completed during the install process:

- The Helm client searches the local repository index.
- The corresponding chart is downloaded from the Azure Container Registry repository.
- The chart is deployed using the Tiller in your Kubernetes cluster.

As the installation proceeds, follow the instructions in the command output to see the WorPress URLs and credentials. You can also run the `kubectl get pods` command to see the Kubernetes resources deployed through the Helm chart:

```
NAME                                    READY   STATUS    RESTARTS   AGE
wordpress-1598530621-67c77b6d86-7ldv4   1/1     Running   0          2m48s
wordpress-1598530621-mariadb-0          1/1     Running   0          2m48s
[...]
```

Delete a Helm chart from the repository

To delete a chart from the repository, use the az acr helm delete command. Specify the name of the chart, such as *wordpress*, and the version to delete, such as *8.1.0*.

```
az acr helm delete --name mycontainerregistry wordpress --version 8.1.0
```

If you wish to delete all versions of the named chart, leave out the `--version` parameter.

The chart continues to be returned when you run `helm search`. Again, the Helm client doesn't automatically update the list of available charts in a repository. To update the Helm client repo index, use the az acr helm repo add command again:

```
az acr helm repo add --name mycontainerregistry
```

Next steps

This article used an existing Helm chart from the public *stable* repository. For more information on how to create and deploy Helm charts, see Developing Helm charts.

Helm charts can be used as part of the container build process. For more information, see Use Azure Container Registry Tasks.

View container registry repositories in the Azure portal

11/24/2019 • 2 minutes to read • Edit Online

Azure Container Registry allows you to store Docker container images in repositories. By storing images in repositories, you can store groups of images (or versions of images) in isolated environments. You can specify these repositories when you push images to your registry, and view their contents in the Azure portal.

Prerequisites

- **Container registry**: Create a container registry in your Azure subscription. For example, use the Azure portal or the Azure CLI.
- **Docker CLI**: Install Docker on your local machine, which provides you with the Docker command-line interface.
- **Container image**: Push an image to your container registry. For guidance on how to push and pull images, see Push and pull an image.

View repositories in Azure portal

You can see a list of the repositories hosting your images, as well as the image tags, in the Azure portal.

If you followed the steps in Push and pull an image (and didn't subsequently delete the image), you should have an Nginx image in your container registry. The instructions in that article specified that you tag the image with a namespace, the "samples" in `/samples/nginx`. As a refresher, the docker push command specified in that article was:

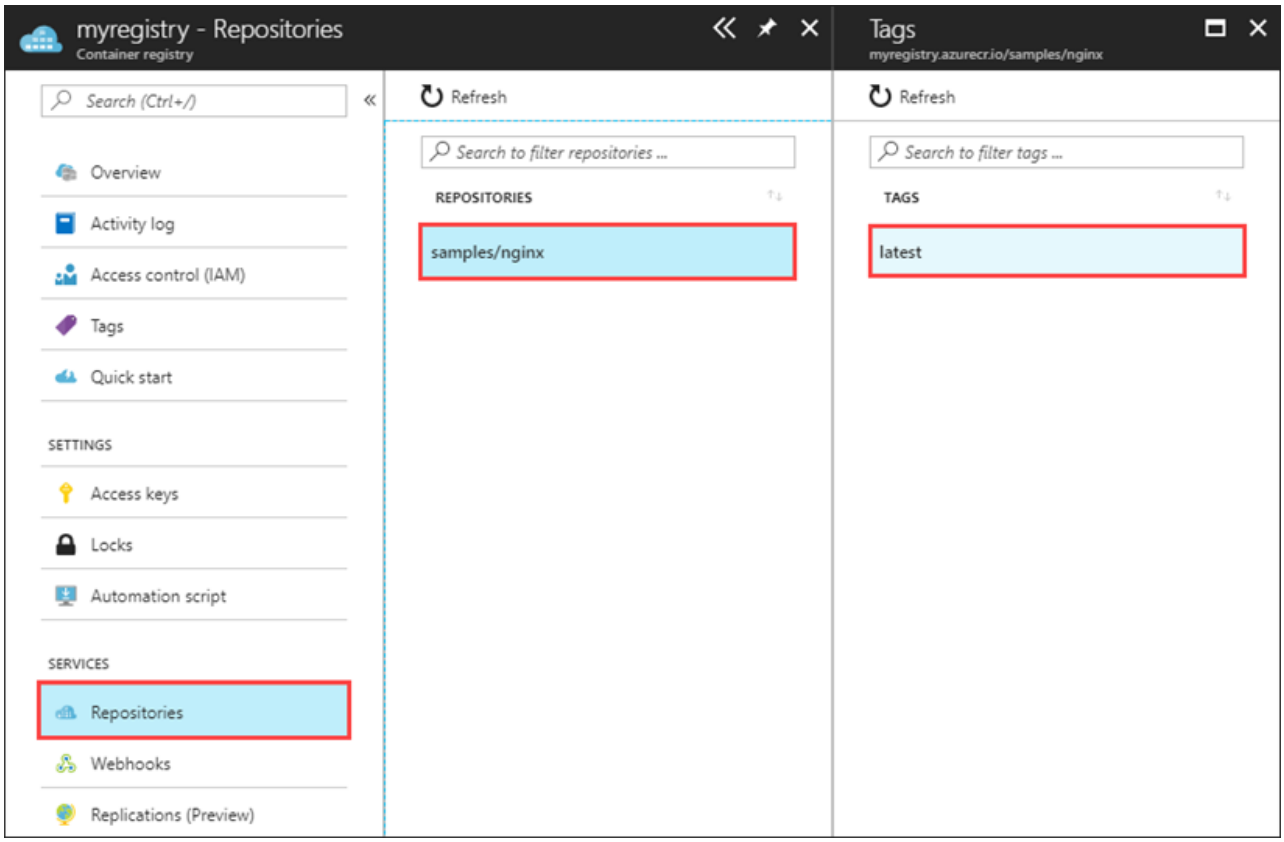
```
docker push myregistry.azurecr.io/samples/nginx
```

Because Azure Container Registry supports such multilevel repository namespaces, you can scope collections of images related to a specific app, or a collection of apps, to different development or operational teams. To read more about repositories in container registries, see Private Docker container registries in Azure.

To view a repository:

1. Sign in to the Azure portal
2. Select the **Azure Container Registry** to which you pushed the Nginx image
3. Select **Repositories** to see a list of the repositories that contain the images in the registry
4. Select a repository to see the image tags within that repository

For example, if you pushed the Nginx image as instructed in Push and pull an image, you should see something similar to:



Next steps

Now that you know the basics of viewing and working with repositories in the portal, try using Azure Container Registry with an Azure Kubernetes Service (AKS) cluster.

Import container images to a container registry

11/24/2019 • 4 minutes to read • Edit Online

You can easily import (copy) container images to an Azure container registry, without using Docker commands. For example, import images from a development registry to a production registry, or copy base images from a public registry.

Azure Container Registry handles a number of common scenarios to copy images from an existing registry:

- Import from a public registry
- Import from another Azure container registry, in the same or a different Azure subscription
- Import from a non-Azure private container registry

Image import into an Azure container registry has the following benefits over using Docker CLI commands:

- Because your client environment doesn't need a local Docker installation, import any container image, regardless of the supported OS type.
- When you import multi-architecture images (such as official Docker images), images for all architectures and platforms specified in the manifest list get copied.

To import container images, this article requires that you run the Azure CLI in Azure Cloud Shell or locally (version 2.0.55 or later recommended). Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

> **NOTE**
> If you need to distribute identical container images across multiple Azure regions, Azure Container Registry also supports geo-replication. By geo-replicating a registry (Premium SKU required), you can serve multiple regions with identical image and tag names from a single registry.

Prerequisites

If you don't already have an Azure container registry, create a registry. For steps, see Quickstart: Create a private container registry using the Azure CLI.

To import an image to an Azure container registry, your identity must have write permissions to the target registry (at least Contributor role). See Azure Container Registry roles and permissions.

Import from a public registry

Import from Docker Hub

For example, use the az acr import command to import the multi-architecture `hello-world:latest` image from Docker Hub to a registry named *myregistry*. Because `hello-world` is an official image from Docker Hub, this image is in the default `library` repository. Include the repository name and optionally a tag in the value of the `--source` image parameter. (You can optionally identify an image by its manifest digest instead of by tag, which guarantees a particular version of an image.)

```
az acr import --name myregistry --source docker.io/library/hello-world:latest --image hello-world:latest
```

You can verify that multiple manifests are associated with this image by running the `az acr repository show-manifests` command:

```
az acr repository show-manifests --name myregistry --repository hello-world
```

The following example imports a public image from the `tensorflow` repository in Docker Hub:

```
az acr import --name myregistry --source docker.io/tensorflow/tensorflow:latest-gpu --image tensorflow:latest-
gpu
```

Import from Microsoft Container Registry

For example, import the latest Windows Server Core image from the `windows` repository in Microsoft Container Registry.

```
az acr import --name myregistry --source mcr.microsoft.com/windows/servercore:latest --image servercore:latest
```

Import from another Azure container registry

You can import an image from another Azure container registry using integrated Azure Active Directory permissions.

- Your identity must have Azure Active Directory permissions to read from the source registry (Reader role) and to write to the target registry (Contributor role).
- The registry can be in the same or a different Azure subscription in the same Active Directory tenant.

Import from a registry in the same subscription

For example, import the `aci-helloworld:latest` image from a source registry *mysourceregistry* to *myregistry* in the same Azure subscription.

```
az acr import --name myregistry --source mysourceregistry.azurecr.io/aci-helloworld:latest --image hello-
world:latest
```

The following example imports an image by manifest digest (SHA-256 hash, represented as `sha256:...`) instead of by tag:

```
az acr import --name myregistry --source mysourceregistry.azurecr.io/aci-helloworld@sha256:123456abcdefg
```

Import from a registry in a different subscription

In the following example, *mysourceregistry* is in a different subscription from *myregistry* in the same Active Directory tenant. Supply the resource ID of the source registry with the `--registry` parameter. Notice that the `--source` parameter specifies only the source repository and image name, not the registry login server name.

```
az acr import --name myregistry --source sourcerepo/aci-helloworld:latest --image aci-hello-world:latest --
registry /subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/sourceResourceGroup/providers/Microsoft.ContainerRegistry/registries/mysourceregis
try
```

Import from a registry using service principal credentials

To import from a registry that you can't access using Active Directory permissions, you can use service principal

credentials (if available). Supply the appID and password of an Active Directory service principal that has ACRPull access to the source registry. Using a service principal is useful for build systems and other unattended systems that need to import images to your registry.

```
az acr import --name myregistry --source sourceregistry.azurecr.io/sourcerepo/sourceimage:tag --image
targetimage:tag --username <SP_App_ID> --password <SP_Passwd>
```

Import from a non-Azure private container registry

Import an image from a private registry by specifying credentials that enable pull access to the registry. For example, pull an image from a private Docker registry:

```
az acr import --name myregistry --source docker.io/sourcerepo/sourceimage:tag --image sourceimage:tag --
username <username> --password <password>
```

Next steps

In this article, you learned about importing container images to an Azure container registry from a public registry or another private registry. For additional image import options, see the az acr import command reference.

Lock a container image in an Azure container registry

2/27/2020 • 3 minutes to read • Edit Online

In an Azure container registry, you can lock an image version or a repository so that it can't be deleted or updated. To lock an image or a repository, update its attributes using the Azure CLI command az acr repository update.

This article requires that you run the Azure CLI in Azure Cloud Shell or locally (version 2.0.55 or later recommended). Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

> **IMPORTANT**
>
> This article doesn't apply to locking an entire registry, for example, using **Settings > Locks** in the Azure portal, or `az lock` commands in the Azure CLI. Locking a registry resource doesn't prevent you from creating, updating, or deleting data in repositories. Locking a registry only affects management operations such as adding or deleting replications, or deleting the registry itself. More information in Lock resources to prevent unexpected changes.

Scenarios

By default, a tagged image in Azure Container Registry is *mutable*, so with appropriate permissions you can repeatedly update and push an image with the same tag to a registry. Container images can also be deleted as needed. This behavior is useful when you develop images and need to maintain a size for your registry.

However, when you deploy a container image to production, you might need an *immutable* container image. An immutable image is one that you can't accidentally delete or overwrite.

See Recommendations for tagging and versioning container images for strategies to tag and version images in your registry.

Use the az acr repository update command to set repository attributes so you can:

- Lock an image version, or an entire repository
- Protect an image version or repository from deletion, but allow updates
- Prevent read (pull) operations on an image version, or an entire repository

See the following sections for examples.

Lock an image or repository

Show the current repository attributes

To see the current attributes of a repository, run the following az acr repository show command:

```
az acr repository show \
    --name myregistry --repository myrepo \
    --output jsonc
```

Show the current image attributes

To see the current attributes of a tag, run the following az acr repository show command:

```
az acr repository show \
    --name myregistry --image image:tag \
    --output jsonc
```

Lock an image by tag

To lock the *myrepo/myimage:tag* image in *myregistry*, run the following az acr repository update command:

```
az acr repository update \
    --name myregistry --image myrepo/myimage:tag \
    --write-enabled false
```

Lock an image by manifest digest

To lock a *myrepo/myimage* image identified by manifest digest (SHA-256 hash, represented as `sha256:...`), run the following command. (To find the manifest digest associated with one or more image tags, run the az acr repository show-manifests command.)

```
az acr repository update \
    --name myregistry --image myrepo/myimage@sha256:123456abcdefg \
    --write-enabled false
```

Lock a repository

To lock the *myrepo/myimage* repository and all images in it, run the following command:

```
az acr repository update \
    --name myregistry --repository myrepo/myimage \
    --write-enabled false
```

Protect an image or repository from deletion

Protect an image from deletion

To allow the *myrepo/myimage:tag* image to be updated but not deleted, run the following command:

```
az acr repository update \
    --name myregistry --image myrepo/myimage:tag \
    --delete-enabled false --write-enabled true
```

Protect a repository from deletion

The following command sets the *myrepo/myimage* repository so it can't be deleted. Individual images can still be updated or deleted.

```
az acr repository update \
    --name myregistry --repository myrepo/myimage \
    --delete-enabled false --write-enabled true
```

Prevent read operations on an image or repository

To prevent read (pull) operations on the *myrepo/myimage:tag* image, run the following command:

```
az acr repository update \
    --name myregistry --image myrepo/myimage:tag \
    --read-enabled false
```

To prevent read operations on all images in the *myrepo/myimage* repository, run the following command:

```
az acr repository update \
    --name myregistry --repository myrepo/myimage \
    --read-enabled false
```

Unlock an image or repository

To restore the default behavior of the *myrepo/myimage:tag* image so that it can be deleted and updated, run the following command:

```
az acr repository update \
    --name myregistry --image myrepo/myimage:tag \
    --delete-enabled true --write-enabled true
```

To restore the default behavior of the *myrepo/myimage* repository and all images so that they can be deleted and updated, run the following command:

```
az acr repository update \
    --name myregistry --repository myrepo/myimage \
    --delete-enabled true --write-enabled true
```

Next steps

In this article, you learned about using the az acr repository update command to prevent deletion or updating of image versions in a repository. To set additional attributes, see the az acr repository update command reference.

To see the attributes set for an image version or repository, use the az acr repository show command.

For details about delete operations, see Delete container images in Azure Container Registry.

Delete container images in Azure Container Registry using the Azure CLI

11/24/2019 • 8 minutes to read • Edit Online

To maintain the size of your Azure container registry, you should periodically delete stale image data. While some container images deployed into production may require longer-term storage, others can typically be deleted more quickly. For example, in an automated build and test scenario, your registry can quickly fill with images that might never be deployed, and can be purged shortly after completing the build and test pass.

Because you can delete image data in several different ways, it's important to understand how each delete operation affects storage usage. This article covers several methods for deleting image data:

- Delete a repository: Deletes all images and all unique layers within the repository.
- Delete by tag: Deletes an image, the tag, all unique layers referenced by the image, and all other tags associated with the image.
- Delete by manifest digest: Deletes an image, all unique layers referenced by the image, and all tags associated with the image.

Sample scripts are provided to help automate delete operations.

For an introduction to these concepts, see About registries, repositories, and images.

Delete repository

Deleting a repository deletes all of the images in the repository, including all tags, unique layers, and manifests. When you delete a repository, you recover the storage space used by the images that reference unique layers in that repository.

The following Azure CLI command deletes the "acr-helloworld" repository and all tags and manifests within the repository. If layers referenced by the deleted manifests are not referenced by any other images in the registry, their layer data is also deleted, recovering the storage space.

```
az acr repository delete --name myregistry --repository acr-helloworld
```

Delete by tag

You can delete individual images from a repository by specifying the repository name and tag in the delete operation. When you delete by tag, you recover the storage space used by any unique layers in the image (layers not shared by any other images in the registry).

To delete by tag, use az acr repository delete and specify the image name in the `--image` parameter. All layers unique to the image, and any other tags associated with the image are deleted.

For example, deleting the "acr-helloworld:latest" image from registry "myregistry":

```
$ az acr repository delete --name myregistry --image acr-helloworld:latest
This operation will delete the manifest
'sha256:0a2e01852872580b2c2fea9380ff8d7b637d3928783c55beb3f21a6e58d5d108' and all the following images:
'acr-helloworld:latest', 'acr-helloworld:v3'.
Are you sure you want to continue? (y/n): y
```

TIP

Deleting *by tag* shouldn't be confused with deleting a tag (untagging). You can delete a tag with the Azure CLI command az acr repository untag. No space is freed when you untag an image because its manifest and layer data remain in the registry. Only the tag reference itself is deleted.

Delete by manifest digest

A manifest digest can be associated with one, none, or multiple tags. When you delete by digest, all tags referenced by the manifest are deleted, as is layer data for any layers unique to the image. Shared layer data is not deleted.

To delete by digest, first list the manifest digests for the repository containing the images you wish to delete. For example:

```
$ az acr repository show-manifests --name myregistry --repository acr-helloworld
[
  {
    "digest": "sha256:0a2e01852872580b2c2fea9380ff8d7b637d3928783c55beb3f21a6e58d5d108",
    "tags": [
      "latest",
      "v3"
    ],
    "timestamp": "2018-07-12T15:52:00.2075864Z"
  },
  {
    "digest": "sha256:3168a21b98836dda7eb7a846b3d735286e09a32b0aa2401773da518e7eba3b57",
    "tags": [
      "v2"
    ],
    "timestamp": "2018-07-12T15:50:53.5372468Z"
  }
]
```

Next, specify the digest you wish to delete in the az acr repository delete command. The format of the command is:

```
az acr repository delete --name <acrName> --image <repositoryName>@<digest>
```

For example, to delete the last manifest listed in the preceding output (with the tag "v2"):

```
$ az acr repository delete --name myregistry --image acr-
helloworld@sha256:3168a21b98836dda7eb7a846b3d735286e09a32b0aa2401773da518e7eba3b57
This operation will delete the manifest
'sha256:3168a21b98836dda7eb7a846b3d735286e09a32b0aa2401773da518e7eba3b57' and all the following images:
'acr-helloworld:v2'.
Are you sure you want to continue? (y/n): y
```

The `acr-helloworld:v2` image is deleted from the registry, as is any layer data unique to that image. If a manifest is associated with multiple tags, all associated tags are also deleted.

Delete digests by timestamp

To maintain the size of a repository or registry, you might need to periodically delete manifest digests older than a certain date.

The following Azure CLI command lists all manifest digest in a repository older than a specified timestamp, in ascending order. Replace `<acrName>` and `<repositoryName>` with values appropriate for your environment. The timestamp could be a full date-time expression or a date, as in this example.

```
az acr repository show-manifests --name <acrName> --repository <repositoryName> \
--orderby time_asc -o tsv --query "[?timestamp < '2019-04-05'].[digest, timestamp]"
```

After identifying stale manifest digests, you can run the following Bash script to delete manifest digests older than a specified timestamp. It requires the Azure CLI and **xargs**. By default, the script performs no deletion. Change the `ENABLE_DELETE` value to `true` to enable image deletion.

> **WARNING**
> Use the following sample script with caution--deleted image data is UNRECOVERABLE. If you have systems that pull images by manifest digest (as opposed to image name), you should not run these scripts. Deleting the manifest digests will prevent those systems from pulling the images from your registry. Instead of pulling by manifest, consider adopting a *unique tagging* scheme, a recommended best practice.

```
#!/bin/bash

# WARNING! This script deletes data!
# Run only if you do not have systems
# that pull images via manifest digest.

# Change to 'true' to enable image delete
ENABLE_DELETE=false

# Modify for your environment
# TIMESTAMP can be a date-time string such as 2019-03-15T17:55:00.
REGISTRY=myregistry
REPOSITORY=myrepository
TIMESTAMP=2019-04-05

# Delete all images older than specified timestamp.

if [ "$ENABLE_DELETE" = true ]
then
    az acr repository show-manifests --name $REGISTRY --repository $REPOSITORY \
    --orderby time_asc --query "[?timestamp < '$TIMESTAMP'].digest" -o tsv \
    | xargs -I% az acr repository delete --name $REGISTRY --image $REPOSITORY@% --yes
else
    echo "No data deleted."
    echo "Set ENABLE_DELETE=true to enable deletion of these images in $REPOSITORY:"
    az acr repository show-manifests --name $REGISTRY --repository $REPOSITORY \
   --orderby time_asc --query "[?timestamp < '$TIMESTAMP'].[digest, timestamp]" -o tsv
fi
```

Delete untagged images

As mentioned in the Manifest digest section, pushing a modified image using an existing tag **untags** the previously pushed image, resulting in an orphaned (or "dangling") image. The previously pushed image's manifest--and its layer data--remains in the registry. Consider the following sequence of events:

1. Push image *acr-helloworld* with tag **latest**: `docker push myregistry.azurecr.io/acr-helloworld:latest`
2. Check manifests for repository *acr-helloworld*:

```
$ az acr repository show-manifests --name myregistry --repository acr-helloworld
[
  {
    "digest": "sha256:d2bdc0c22d78cde155f53b4092111d7e13fe28ebf87a945f94b19c248000ceec",
    "tags": [
      "latest"
    ],
    "timestamp": "2018-07-11T21:32:21.1400513Z"
  }
]
```

3. Modify *acr-helloworld* Dockerfile
4. Push image *acr-helloworld* with tag **latest**: `docker push myregistry.azurecr.io/acr-helloworld:latest`
5. Check manifests for repository *acr-helloworld*:

```
$ az acr repository show-manifests --name myregistry --repository acr-helloworld
[
  {
    "digest": "sha256:7ca0e0ae50c95155dbb0e380f37d7471e98d2232ed9e31eece9f9fb9078f2728",
    "tags": [
      "latest"
    ],
    "timestamp": "2018-07-11T21:38:35.9170967Z"
  },
  {
    "digest": "sha256:d2bdc0c22d78cde155f53b4092111d7e13fe28ebf87a945f94b19c248000ceec",
    "tags": [],
    "timestamp": "2018-07-11T21:32:21.1400513Z"
  }
]
```

As you can see in the output of the last step in the sequence, there is now an orphaned manifest whose `"tags"` property is an empty list. This manifest still exists within the registry, along with any unique layer data that it references. **To delete such orphaned images and their layer data, you must delete by manifest digest**.

Delete all untagged images

You can list all untagged images in your repository using the following Azure CLI command. Replace `<acrName>` and `<repositoryName>` with values appropriate for your environment.

```
az acr repository show-manifests --name <acrName> --repository <repositoryName> --query "[?
tags[0]==null].digest"
```

Using this command in a script, you can delete all untagged images in a repository.

> **WARNING**
> Use the following sample scripts with caution--deleted image data is UNRECOVERABLE. If you have systems that pull images by manifest digest (as opposed to image name), you should not run these scripts. Deleting untagged images will prevent those systems from pulling the images from your registry. Instead of pulling by manifest, consider adopting a *unique tagging* scheme, a recommended best practice.

Azure CLI in Bash

The following Bash script deletes all untagged images from a repository. It requires the Azure CLI and **xargs**. By

default, the script performs no deletion. Change the `ENABLE_DELETE` value to `true` to enable image deletion.

```
#!/bin/bash

# WARNING! This script deletes data!
# Run only if you do not have systems
# that pull images via manifest digest.

# Change to 'true' to enable image delete
ENABLE_DELETE=false

# Modify for your environment
REGISTRY=myregistry
REPOSITORY=myrepository

# Delete all untagged (orphaned) images
if [ "$ENABLE_DELETE" = true ]
then
    az acr repository show-manifests --name $REGISTRY --repository $REPOSITORY  --query "[?
tags[0]==null].digest" -o tsv \
    | xargs -I% az acr repository delete --name $REGISTRY --image $REPOSITORY@% --yes
else
    echo "No data deleted."
    echo "Set ENABLE_DELETE=true to enable image deletion of these images in $REPOSITORY:"
    az acr repository show-manifests --name $REGISTRY --repository $REPOSITORY --query "[?tags[0]==null]" -o
tsv
fi
```

Azure CLI in PowerShell

The following PowerShell script deletes all untagged images from a repository. It requires PowerShell and the Azure CLI. By default, the script performs no deletion. Change the `$enableDelete` value to `$TRUE` to enable image deletion.

```
# WARNING! This script deletes data!
# Run only if you do not have systems
# that pull images via manifest digest.

# Change to '$TRUE' to enable image delete
$enableDelete = $FALSE

# Modify for your environment
$registry = "myregistry"
$repository = "myrepository"

if ($enableDelete) {
    az acr repository show-manifests --name $registry --repository $repository --query "[?
tags[0]==null].digest" -o tsv `
    | %{ az acr repository delete --name $registry --image $repository@$_ --yes }
} else {
    Write-Host "No data deleted."
    Write-Host "Set `$enableDelete = `$TRUE to enable image deletion."
    az acr repository show-manifests --name $registry --repository $repository --query "[?tags[0]==null]" -o
tsv
}
```

Automatically purge tags and manifests (preview)

As an alternative to scripting Azure CLI commands, run an on-demand or scheduled ACR task to delete all tags that are older than a certain duration or match a specified name filter. For more information, see Automatically purge images from an Azure container registry.

Optionally set a retention policy for each registry, to manage untagged manifests. When you enable a retention policy, image manifests in the registry that don't have any associated tags, and the underlying layer data, are automatically deleted after a set period.

Next steps

For more information about image storage in Azure Container Registry see Container image storage in Azure Container Registry.

Set a retention policy for untagged manifests

11/24/2019 • 4 minutes to read • Edit Online

Azure Container Registry gives you the option to set a *retention policy* for stored image manifests that don't have any associated tags (*untagged manifests*). When a retention policy is enabled, untagged manifests in the registry are automatically deleted after a number of days you set. This feature prevents the registry from filling up with artifacts that aren't needed and helps you save on storage costs. If the `delete-enabled` attribute of an untagged manifest is set to `false`, the manifest can't be deleted, and the retention policy doesn't apply.

You can use the Azure Cloud Shell or a local installation of the Azure CLI to run the command examples in this article. If you'd like to use it locally, version 2.0.74 or later is required. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

IMPORTANT

This feature is currently in preview, and some limitations apply. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

WARNING

Set a retention policy with care--deleted image data is UNRECOVERABLE. If you have systems that pull images by manifest digest (as opposed to image name), you should not set a retention policy for untagged manifests. Deleting untagged images will prevent those systems from pulling the images from your registry. Instead of pulling by manifest, consider adopting a *unique tagging* scheme, a recommended best practice.

Preview limitations

- Only a **Premium** container registry can be configured with a retention policy. For information about registry service tiers, see Azure Container Registry SKUs.
- You can only set a retention policy for untagged manifests.
- The retention policy currently applies only to manifests that are untagged *after* the policy is enabled. Existing untagged manifests in the registry aren't subject to the policy. To delete existing untagged manifests, see examples in Delete container images in Azure Container Registry.

About the retention policy

Azure Container Registry does reference counting for manifests in the registry. When a manifest is untagged, it checks the retention policy. If a retention policy is enabled, a manifest delete operation is queued, with a specific date, according to the number of days set in the policy.

A separate queue management job constantly processes messages, scaling as needed. As an example, suppose you untagged two manifests, 1 hour apart, in a registry with a retention policy of 30 days. Two messages would be queued. Then, 30 days later, approximately 1 hour apart, the messages would be retrieved from the queue and processed, assuming the policy was still in effect.

Set a retention policy - CLI

The following example shows you how to use the Azure CLI to set a retention policy for untagged manifests in a registry.

Enable a retention policy

By default, no retention policy is set in a container registry. To set or update a retention policy, run the az acr config retention update command in the Azure CLI. You can specify a number of days between 0 and 365 to retain the untagged manifests. If you don't specify a number of days, the command sets a default of 7 days. After the retention period, all untagged manifests in the registry are automatically deleted.

The following example sets a retention policy of 30 days for untagged manifests in the registry *myregistry*:

```
az acr config retention update --registry myregistry --status enabled --days 30 --type UntaggedManifests
```

The following example sets a policy to delete any manifest in the registry as soon as it's untagged. Create this policy by setting a retention period of 0 days.

```
az acr config retention update --registry myregistry --status enabled --days 0 --type UntaggedManifests
```

Validate a retention policy

If you enable the preceding policy with a retention period of 0 days, you can quickly verify that untagged manifests are deleted:

1. Push a test image `hello-world:latest` image to your registry, or substitute another test image of your choice.
2. Untag the `hello-world:latest` image, for example, using the az acr repository untag command. The untagged manifest remains in the registry.

   ```
   az acr repository untag --name myregistry --image hello-world:latest
   ```

3. Within a few seconds, the untagged manifest is deleted. You can verify the deletion by listing manifests in the repository, for example, using the az acr repository show-manifests command. If the test image was the only one in the repository, the repository itself is deleted.

Disable a retention policy

To see the retention policy set in a registry, run the az acr config retention show command:

```
az acr config retention show --registry myregistry
```

To disable a retention policy in a registry, run the az acr config retention update command and set `--status disabled`:

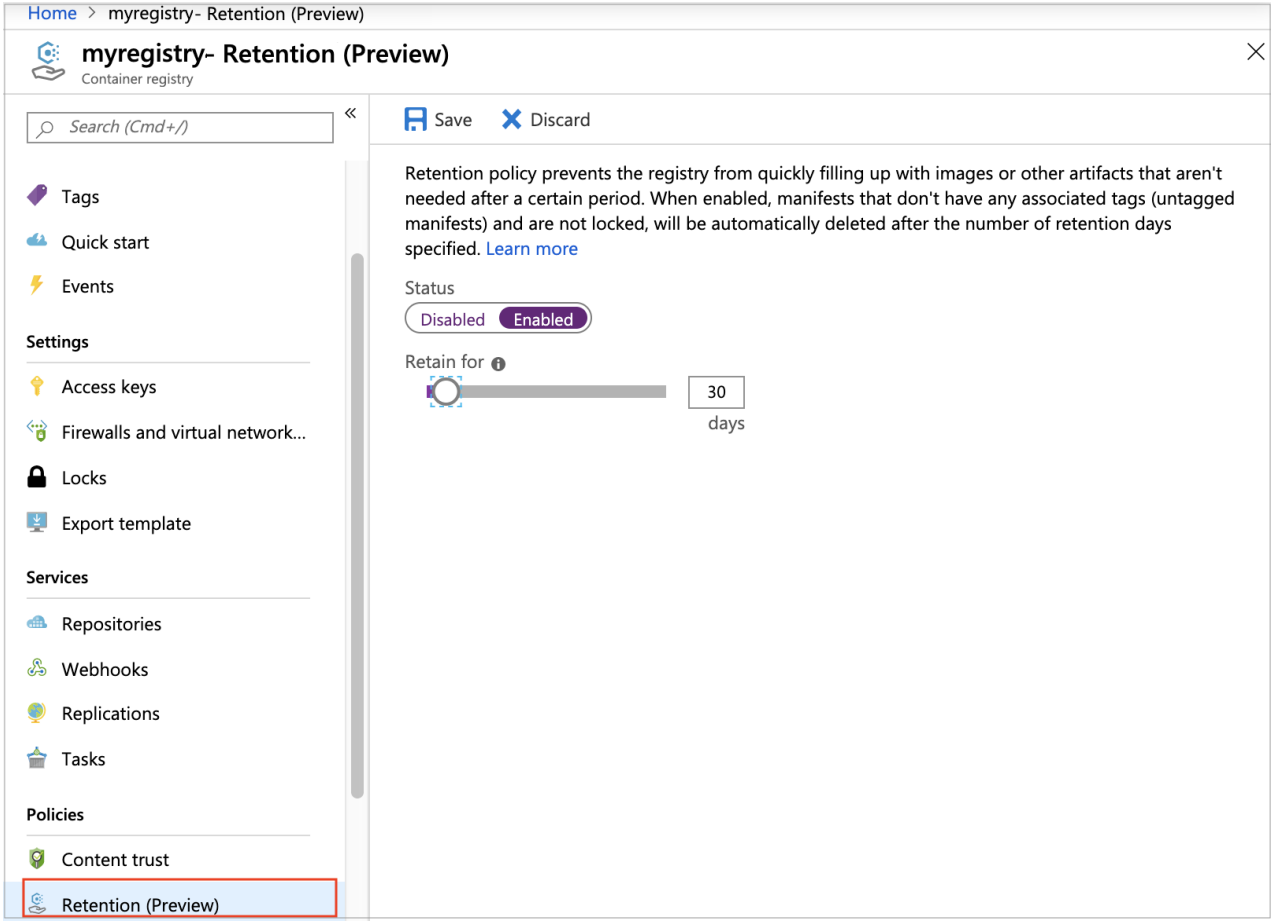
```
az acr config retention update --registry myregistry --status disabled --type UntaggedManifests
```

Set a retention policy - portal

You can also set a registry's retention policy in the Azure portal. The following example shows you how to use the portal to set a retention policy for untagged manifests in a registry.

Enable a retention policy

1. Navigate to your Azure container registry. Under **Policies**, select **Retention** (Preview).
2. In **Status**, select **Enabled**.
3. Select a number of days between 0 and 365 to retain the untagged manifests. Select **Save**.



Disable a retention policy

1. Navigate to your Azure container registry. Under **Policies**, select **Retention** (Preview).
2. In **Status**, select **Disabled**. Select **Save**.

Next steps

- Learn more about options to delete images and repositories in Azure Container Registry
- Learn how to automatically purge selected images and manifests from a registry
- Learn more about options to lock images and manifests in a registry

Automatically purge images from an Azure container registry

12/8/2019 • 6 minutes to read • Edit Online

When you use an Azure container registry as part of a development workflow, the registry can quickly fill up with images or other artifacts that aren't needed after a short period. You might want to delete all tags that are older than a certain duration or match a specified name filter. To delete multiple artifacts quickly, this article introduces the `acr purge` command you can run as an on-demand or scheduled ACR Task.

The `acr purge` command is currently distributed in a public container image (`mcr.microsoft.com/acr/acr-cli:0.1`), built from source code in the acr-cli repo in GitHub.

You can use the Azure Cloud Shell or a local installation of the Azure CLI to run the ACR task examples in this article. If you'd like to use it locally, version 2.0.69 or later is required. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

IMPORTANT

This feature is currently in preview. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

WARNING

Use the `acr purge` command with caution--deleted image data is UNRECOVERABLE. If you have systems that pull images by manifest digest (as opposed to image name), you should not purge untagged images. Deleting untagged images will prevent those systems from pulling the images from your registry. Instead of pulling by manifest, consider adopting a *unique tagging* scheme, a recommended best practice.

If you want to delete single image tags or manifests using Azure CLI commands, see Delete container images in Azure Container Registry.

Use the purge command

The `acr purge` container command deletes images by tag in a repository that match a name filter and that are older than a specified duration. By default, only tag references are deleted, not the underlying manifests and layer data. The command has an option to also delete manifests.

NOTE

`acr purge` does not delete an image tag or repository where the `write-enabled` attribute is set to `false`. For information, see Lock a container image in an Azure container registry.

`acr purge` is designed to run as a container command in an ACR Task, so that it authenticates automatically with the registry where the task runs and performs actions there. The task examples in this article use the `acr purge` command alias in place of a fully qualified container image command.

At a minimum, specify the following when you run `acr purge`:

- `--filter` - A repository and a *regular expression* to filter tags in the repository. Examples:

`--filter "hello-world:.*"` matches all tags in the `hello-world` repository, and `--filter "hello-world:^1.*"` matches tags beginning with `1`. Pass multiple `--filter` parameters to purge multiple repositories.

- `--ago` - A Go-style duration string to indicate a duration beyond which images are deleted. The duration consists of a sequence of one or more decimal numbers, each with a unit suffix. Valid time units include "d" for days, "h" for hours, and "m" for minutes. For example, `--ago 2d3h6m` selects all filtered images last modified more than 2 days, 3 hours, and 6 minutes ago, and `--ago 1.5h` selects images last modified more than 1.5 hours ago.

`acr purge` supports several optional parameters. The following two are used in examples in this article:

- `--untagged` - Specifies that manifests that don't have associated tags (*untagged manifests*) are deleted.
- `--dry-run` - Specifies that no data is deleted, but the output is the same as if the command is run without this flag. This parameter is useful for testing a purge command to make sure it does not inadvertently delete data you intend to preserve.

For additional parameters, run `acr purge --help`.

`acr purge` supports other features of ACR Tasks commands including run variables and task run logs that are streamed and also saved for later retrieval.

Run in an on-demand task

The following example uses the az acr run command to run the `acr purge` command on-demand. This example deletes all image tags and manifests in the `hello-world` repository in *myregistry* that were modified more than 1 day ago. The container command is passed using an environment variable. The task runs without a source context.

```
# Environment variable for container command line
PURGE_CMD="acr purge --filter 'hello-world:.*' \
  --untagged --ago 1d"

az acr run \
  --cmd "$PURGE_CMD" \
  --registry myregistry \
  /dev/null
```

Run in a scheduled task

The following example uses the az acr task create command to create a daily scheduled ACR task. The task purges tags modified more than 7 days ago in the `hello-world` repository. The container command is passed using an environment variable. The task runs without a source context.

```
# Environment variable for container command line
PURGE_CMD="acr purge --filter 'hello-world:.*' \
  --ago 7d"

az acr task create --name purgeTask \
  --cmd "$PURGE_CMD" \
  --schedule "0 0 * * *" \
  --registry myregistry \
  --context /dev/null
```

Run the az acr task show command to see that the timer trigger is configured.

Purge large numbers of tags and manifests

Purging a large number of tags and manifests could take several minutes or longer. To purge thousands of tags and manifests, the command might need to run longer than the default timeout time of 600 seconds for an on-demand task, or 3600 seconds for a scheduled task. If the timeout time is exceeded, only a subset of tags and

manifests are deleted. To ensure that a large-scale purge is complete, pass the `--timeout` parameter to increase the value.

For example, the following on-demand task sets a timeout time of 3600 seconds (1 hour):

```
# Environment variable for container command line
PURGE_CMD="acr purge --filter 'hello-world:.*' \
  --ago 1d --untagged"

az acr run \
  --cmd "$PURGE_CMD" \
  --registry myregistry \
  --timeout 3600 \
  /dev/null
```

Example: Scheduled purge of multiple repositories in a registry

This example walks through using `acr purge` to periodically clean up multiple repositories in a registry. For example, you might have a development pipeline that pushes images to the `samples/devimage1` and `samples/devimage2` repositories. You periodically import development images into a production repository for your deployments, so you no longer need the development images. On a weekly basis, you purge the `samples/devimage1` and `samples/devimage2` repositories, in preparation for the coming week's work.

Preview the purge

Before deleting data, we recommend running an on-demand purge task using the `--dry-run` parameter. This option allows you to see the tags and manifests that the command will purge, without removing any data.

In the following example, the filter in each repository selects all tags. The `--ago 0d` parameter matches images of all ages in the repositories that match the filters. Modify the selection criteria as needed for your scenario. The `--untagged` parameter indicates to delete manifests in addition to tags. The container command is passed to the az acr run command using an environment variable.

```
# Environment variable for container command line
PURGE_CMD="acr purge \
  --filter 'samples/devimage1:.*' --filter 'samples/devimage2:.*' \
  --ago 0d --untagged --dry-run"

az acr run \
  --cmd "$PURGE_CMD" \
  --registry myregistry \
  /dev/null
```

Review the command output to see the tags and manifests that match the selection parameters. Because the command is run with `--dry-run`, no data is deleted.

Sample output:

```
[...]
Deleting tags for repository: samples/devimage1
myregistry.azurecr.io/samples/devimage1:232889b
myregistry.azurecr.io/samples/devimage1:a21776a
Deleting manifests for repository: samples/devimage1
myregistry.azurecr.io/samples/devimage1@sha256:81b6f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e78
8b
myregistry.azurecr.io/samples/devimage1@sha256:3ded859790e68bd02791a972ab0bae727231dc8746f233a7949e40f8ea90c8
b3
Deleting tags for repository: samples/devimage2
myregistry.azurecr.io/samples/devimage2:5e788ba
myregistry.azurecr.io/samples/devimage2:f336b7c
Deleting manifests for repository: samples/devimage2
myregistry.azurecr.io/samples/devimage2@sha256:8d2527cde610e1715ad095cb12bc7ed169b60c495e5428eefdf336b7cb7c03
71
myregistry.azurecr.io/samples/devimage2@sha256:ca86b078f89607bc03ded859790e68bd02791a972ab0bae727231dc8746f23
3a

Number of deleted tags: 4
Number of deleted manifests: 4
[...]
```

Schedule the purge

After you've verified the dry run, create a scheduled task to automate the purge. The following example schedules a weekly task on Sunday at 1:00 UTC to run the previous purge command:

```
# Environment variable for container command line
PURGE_CMD="acr purge \
  --filter 'samples/devimage1:.*' --filter 'samples/devimage2:.*' \
  --ago 0d --untagged"

az acr task create --name weeklyPurgeTask \
  --cmd "$PURGE_CMD" \
  --schedule "0 1 * * Sun" \
  --registry myregistry \
  --context /dev/null
```

Run the az acr task show command to see that the timer trigger is configured.

Next steps

Learn about other options to delete image data in Azure Container Registry.

For more information about image storage, see Container image storage in Azure Container Registry.

Using Azure Container Registry webhooks

11/24/2019 • 3 minutes to read • Edit Online

An Azure container registry stores and manages private Docker container images, similar to the way Docker Hub stores public Docker images. It can also host repositories for Helm charts (preview), a packaging format to deploy applications to Kubernetes. You can use webhooks to trigger events when certain actions take place in one of your registry repositories. Webhooks can respond to events at the registry level, or they can be scoped down to a specific repository tag. With a geo-replicated registry, you configure each webhook to respond to events in a specific regional replica.

For details on webhook requests, see Azure Container Registry webhook schema reference.

Prerequisites

- Azure container registry - Create a container registry in your Azure subscription. For example, use the Azure portal or the Azure CLI. The Azure Container Registry SKUs have different webhooks quotas.
- Docker CLI - To set up your local computer as a Docker host and access the Docker CLI commands, install Docker Engine.

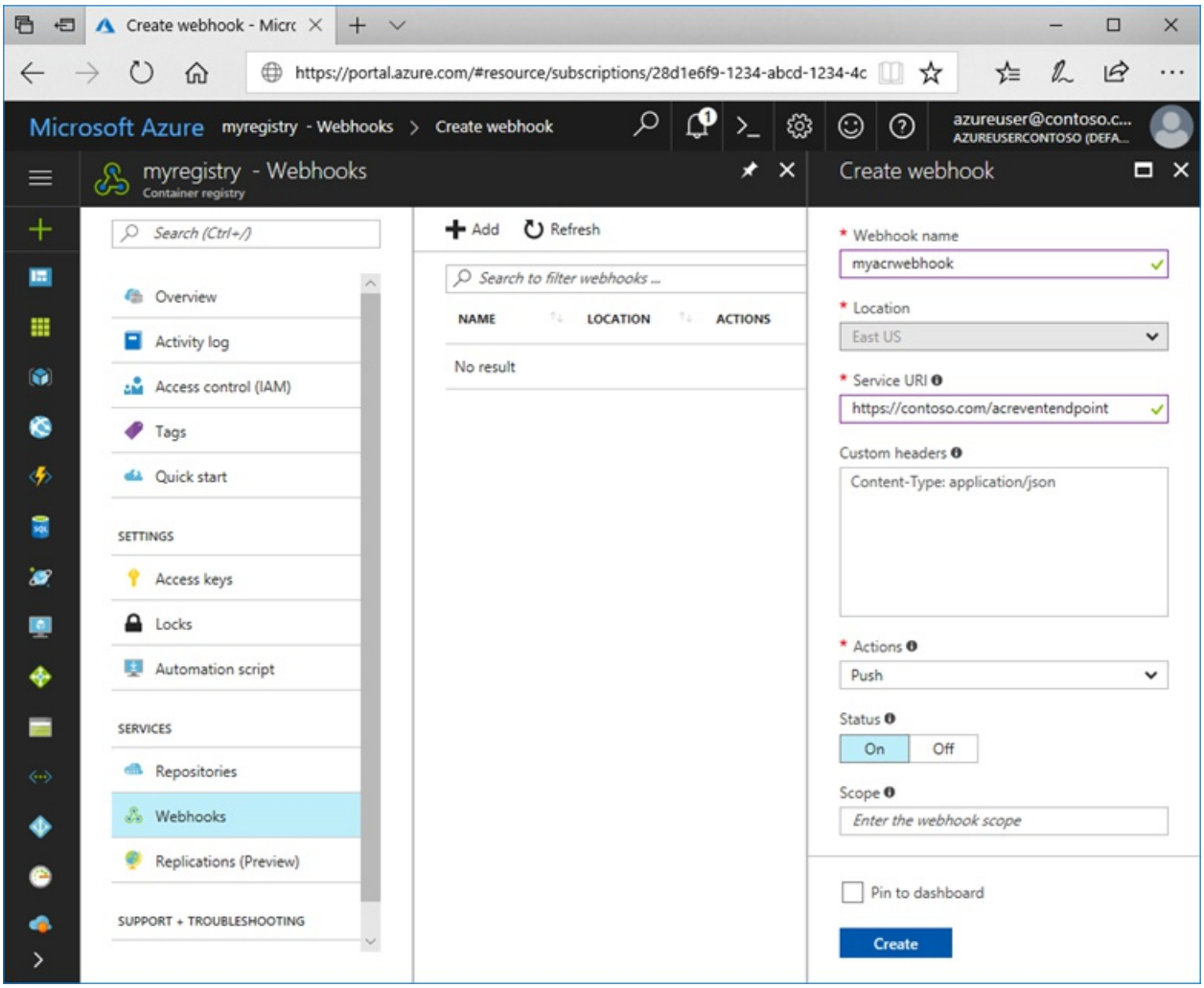
Create webhook - Azure portal

1. Sign in to the Azure portal.
2. Navigate to the container registry in which you want to create a webhook.
3. Under **Services**, select **Webhooks**.
4. Select **Add** in the webhook toolbar.
5. Complete the *Create webhook* form with the following information:

VALUE	DESCRIPTION
Webhook name	The name you want to give to the webhook. It may contain only letters and numbers, and must be 5-50 characters in length.
Location	For a geo-replicated registry, specify the Azure region of the registry replica.
Service URI	The URI where the webhook should send POST notifications.
Custom headers	Headers you want to pass along with the POST request. They should be in "key: value" format.
Trigger actions	Actions that trigger the webhook. Actions include image push, image delete, Helm chart push, Helm chart delete, and image quarantine. You can choose one or more actions to trigger the webhook.
Status	The status for the webhook after it's created. It's enabled by default.

VALUE	DESCRIPTION
Scope	The scope at which the webhook works. If not specified, the scope is for all events in the registry. It can be specified for a repository or a tag by using the format "repository:tag", or "repository:*" for all tags under a repository.

Example webhook form:



Create webhook - Azure CLI

To create a webhook using the Azure CLI, use the az acr webhook create command. The following command creates a webhook for all image delete events in the registry *mycontainerregistry*:

```
az acr webhook create --registry mycontainerregistry --name myacrwebhook01 --actions delete --uri
http://webhookuri.com
```

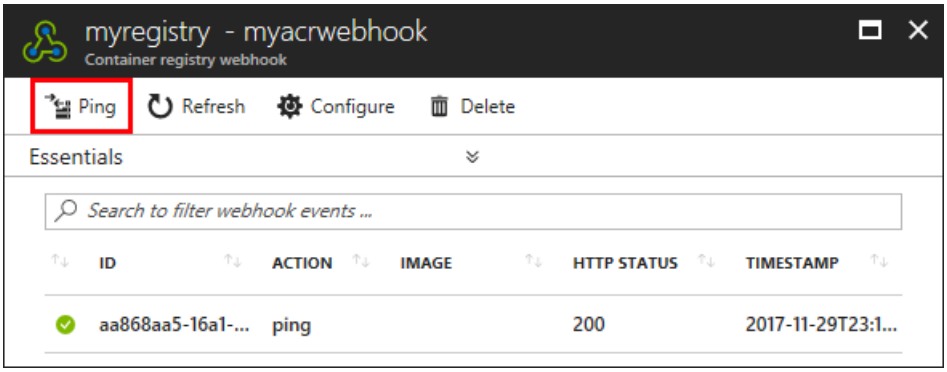
Test webhook

Azure portal

Prior to using the webhook, you can test it with the **Ping** button. Ping sends a generic POST request to the specified endpoint and logs the response. Using the ping feature can help you verify you've correctly configured the webhook.

1. Select the webhook you want to test.
2. In the top toolbar, select **Ping**.

3. Check the endpoint's response in the **HTTP STATUS** column.



Azure CLI

To test an ACR webhook with the Azure CLI, use the az acr webhook ping command.

```
az acr webhook ping --registry mycontainerregistry --name myacrwebhook01
```

To see the results, use the az acr webhook list-events command.

```
az acr webhook list-events --registry mycontainerregistry08 --name myacrwebhook01
```

Delete webhook

Azure portal

Each webhook can be deleted by selecting the webhook and then the **Delete** button in the Azure portal.

Azure CLI

```
az acr webhook delete --registry mycontainerregistry --name myacrwebhook01
```

Next steps

Webhook schema reference

For details on the format and properties of the JSON event payloads emitted by Azure Container Registry, see the webhook schema reference:

Azure Container Registry webhook schema reference

Event Grid events

In addition to the native registry webhook events discussed in this article, Azure Container Registry can emit events to Event Grid:

Quickstart: Send container registry events to Event Grid

Restrict access to an Azure container registry using an Azure virtual network or firewall rules

11/24/2019 • 12 minutes to read • Edit Online

Azure Virtual Network provides secure, private networking for your Azure and on-premises resources. By limiting access to your private Azure container registry from an Azure virtual network, you ensure that only resources in the virtual network access the registry. For cross-premises scenarios, you can also configure firewall rules to allow registry access only from specific IP addresses.

This article shows two scenarios to configure inbound network access rules on a container registry: from a virtual machine deployed in a virtual network, or from a VM's public IP address.

> **IMPORTANT**
> This feature is currently in preview, and some limitations apply. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

If instead you need to set up access rules for resources to reach a container registry from behind a firewall, see Configure rules to access an Azure container registry behind a firewall.

Preview limitations

- Only a **Premium** container registry can be configured with network access rules. For information about registry service tiers, see Azure Container Registry SKUs.
- Only an Azure Kubernetes Service cluster or Azure virtual machine can be used as a host to access a container registry in a virtual network. *Other Azure services including Azure Container Instances aren't currently supported.*
- ACR Tasks operations aren't currently supported in a container registry accessed in a virtual network.
- Each registry supports a maximum of 100 virtual network rules.

Prerequisites

- To use the Azure CLI steps in this article, Azure CLI version 2.0.58 or later is required. If you need to install or upgrade, see Install Azure CLI.
- If you don't already have a container registry, create one (Premium SKU required) and push a sample image such as `hello-world` from Docker Hub. For example, use the Azure portal or the Azure CLI to create a registry.
- If you want to restrict registry access using a virtual network in a different Azure subscription, you need to register the resource provider for Azure Container Registry in that subscription. For example:

```
az account set --subscription <Name or ID of subscription of virtual network>

az provider register --namespace Microsoft.ContainerRegistry
```

About network rules for a container registry

An Azure container registry by default accepts connections over the internet from hosts on any network. With a virtual network, you can allow only Azure resources such as an AKS cluster or Azure VM to securely access the registry, without crossing a network boundary. You can also configure network firewall rules to allow only specific public internet IP address ranges.

To limit access to a registry, first change the default action of the registry so that it denies all network connections. Then, add network access rules. Clients granted access via the network rules must continue to authenticate to the container registry and be authorized to access the data.

Service endpoint for subnets

To allow access from a subnet in a virtual network, you need to add a service endpoint for the Azure Container Registry service.

Multi-tenant services, like Azure Container Registry, use a single set of IP addresses for all customers. A service endpoint assigns an endpoint to access a registry. This endpoint gives traffic an optimal route to the resource over the Azure backbone network. The identities of the virtual network and the subnet are also transmitted with each request.

Firewall rules

For IP network rules, provide allowed internet address ranges using CIDR notation such as *16.17.18.0/24* or an individual IP addresses like *16.17.18.19*. IP network rules are only allowed for *public* internet IP addresses. IP address ranges reserved for private networks (as defined in RFC 1918) aren't allowed in IP rules.

Create a Docker-enabled virtual machine

For this article, use a Docker-enabled Ubuntu VM to access an Azure container registry. To use Azure Active Directory authentication to the registry, also install the Azure CLI on the VM. If you already have an Azure virtual machine, skip this creation step.

You may use the same resource group for your virtual machine and your container registry. This setup simplifies clean-up at the end but isn't required. If you choose to create a separate resource group for the virtual machine and virtual network, run az group create. The following example creates a resource group named *myResourceGroup* in the *westcentralus* location:

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

Now deploy a default Ubuntu Azure virtual machine with az vm create. The following example creates a VM named *myDockerVM*:

```
az vm create \
    --resource-group myResourceGroup \
    --name myDockerVM \
    --image UbuntuLTS \
    --admin-username azureuser \
    --generate-ssh-keys
```

It takes a few minutes for the VM to be created. When the command completes, take note of the `publicIpAddress` displayed by the Azure CLI. Use this address to make SSH connections to the VM, and optionally for later setup of firewall rules.

Install Docker on the VM

After the VM is running, make an SSH connection to the VM. Replace *publicIpAddress* with the public IP address of your VM.

```
ssh azureuser@publicIpAddress
```

Run the following command to install Docker on the Ubuntu VM:

```
sudo apt install docker.io -y
```

After installation, run the following command to verify that Docker is running properly on the VM:

```
sudo docker run -it hello-world
```

Output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
[...]
```

Install the Azure CLI

Follow the steps in Install Azure CLI with apt to install the Azure CLI on your Ubuntu virtual machine. For this article, ensure that you install version 2.0.58 or later.

Exit the SSH connection.

Allow access from a virtual network

In this section, configure your container registry to allow access from a subnet in an Azure virtual network. Equivalent steps using the Azure CLI and Azure portal are provided.

Allow access from a virtual network - CLI

Add a service endpoint to a subnet

When you create a VM, Azure by default creates a virtual network in the same resource group. The name of the virtual network is based on the name of the virtual machine. For example, if you name your virtual machine *myDockerVM*, the default virtual network name is *myDockerVMVNET*, with a subnet named *myDockerVMSubnet*. Verify this in the Azure portal or by using the az network vnet list command:

```
az network vnet list --resource-group myResourceGroup --query "[].{Name: name, Subnet: subnets[0].name}"
```

Output:

```
[
  {
    "Name": "myDockerVMVNET",
    "Subnet": "myDockerVMSubnet"
  }
]
```

Use the az network vnet subnet update command to add a **Microsoft.ContainerRegistry** service endpoint to your subnet. Substitute the names of your virtual network and subnet in the following command:

```
az network vnet subnet update \
  --name myDockerVMSubnet \
  --vnet-name myDockerVMVNET \
  --resource-group myResourceGroup \
  --service-endpoints Microsoft.ContainerRegistry
```

Use the az network vnet subnet show command to retrieve the resource ID of the subnet. You need this in a later step to configure a network access rule.

```
az network vnet subnet show \
  --name myDockerVMSubnet \
  --vnet-name myDockerVMVNET \
  --resource-group myResourceGroup \
  --query "id"
  --output tsv
```

Output:

```
/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourceGroups/myResourceGroup/providers/Microsoft.Network/virtualNetworks/myDockerVMVNET/subnets
/myDockerVMSubnet
```

Change default network access to registry

By default, an Azure container registry allows connections from hosts on any network. To limit access to a selected network, change the default action to deny access. Substitute the name of your registry in the following az acr update command:

```
az acr update --name myContainerRegistry --default-action Deny
```

Add network rule to registry

Use the az acr network-rule add command to add a network rule to your registry that allows access from the VM's subnet. Substitute the container registry's name and the resource ID of the subnet in the following command:

```
az acr network-rule add --name mycontainerregistry --subnet <subnet-resource-id>
```

Continue to Verify access to the registry.

Allow access from a virtual network - portal

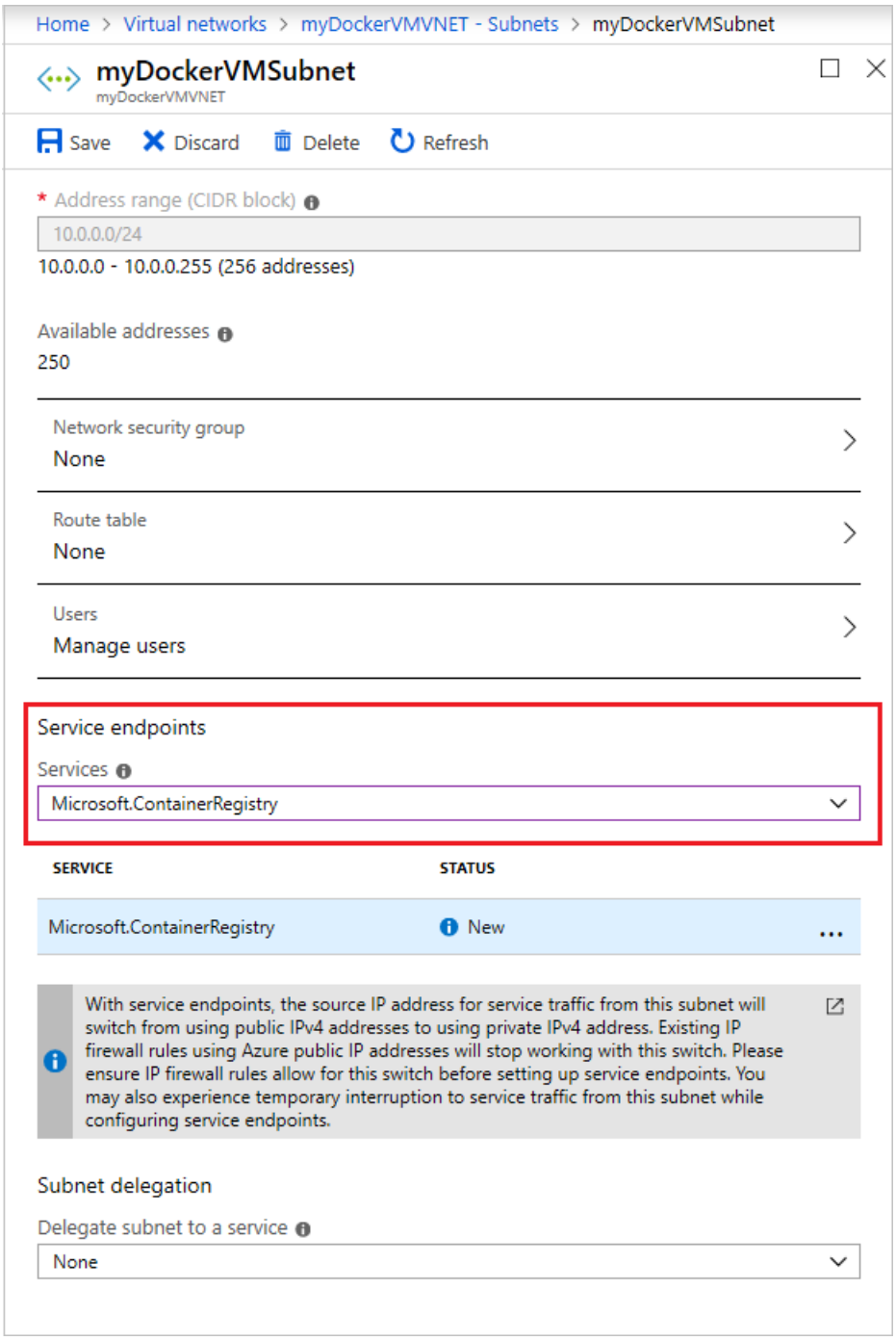
Add service endpoint to subnet

When you create a VM, Azure by default creates a virtual network in the same resource group. The name of the virtual network is based on the name of the virtual machine. For example, if you name your virtual machine *myDockerVM*, the default virtual network name is *myDockerVMVNET*, with a subnet named *myDockerVMSubnet*.

To add a service endpoint for Azure Container Registry to a subnet:

1. In the search box at the top of the Azure portal, enter *virtual networks*. When **Virtual networks** appear in the search results, select it.
2. From the list of virtual networks, select the virtual network where your virtual machine is deployed, such as *myDockerVMVNET*.
3. Under **Settings**, select **Subnets**.
4. Select the subnet where your virtual machine is deployed, such as *myDockerVMSubnet*.

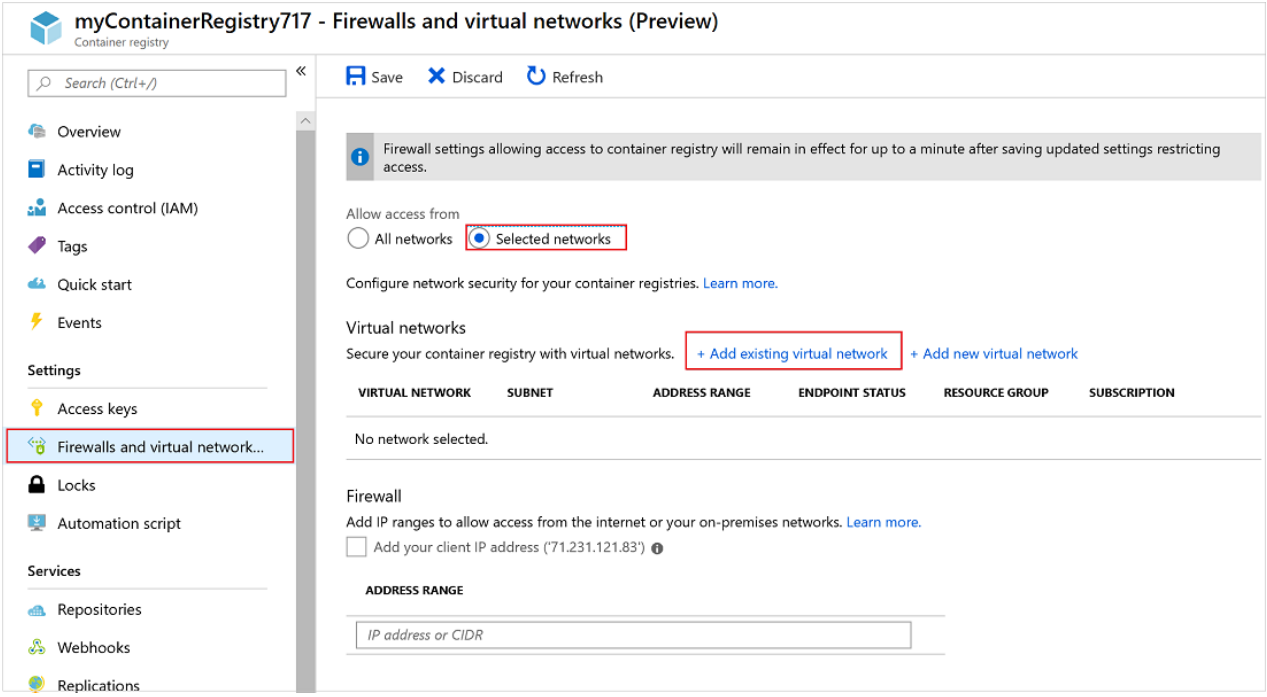
5. Under **Service endpoints**, select **Microsoft.ContainerRegistry**.
6. Select **Save**.



Configure network access for registry

By default, an Azure container registry allows connections from hosts on any network. To limit access to the virtual network:

1. In the portal, navigate to your container registry.
2. Under **Settings**, select **Firewall and virtual networks**.
3. To deny access by default, choose to allow access from **Selected networks**.
4. Select **Add existing virtual network**, and select the virtual network and subnet you configured with a service endpoint. Select **Add**.
5. Select **Save**.



Continue to Verify access to the registry.

Allow access from an IP address

In this section, configure your container registry to allow access from a specific IP address or range. Equivalent steps using the Azure CLI and Azure portal are provided.

Allow access from an IP address - CLI

Change default network access to registry

If you haven't already done so, update the registry configuration to deny access by default. Substitute the name of your registry in the following az acr update command:

```
az acr update --name myContainerRegistry --default-action Deny
```

Remove network rule from registry

If you previously added a network rule to allow access from the VM's subnet, remove the subnet's service endpoint and the network rule. Substitute the container registry's name and the resource ID of the subnet you retrieved in an earlier step in the az acr network-rule remove command:

```
# Remove service endpoint

az network vnet subnet update \
  --name myDockerVMSubnet \
  --vnet-name myDockerVMVNET \
  --resource-group myResourceGroup \
  --service-endpoints ""

# Remove network rule

az acr network-rule remove --name mycontainerregistry --subnet <subnet-resource-id>
```

Add network rule to registry

Use the az acr network-rule add command to add a network rule to your registry that allows access from the VM's IP address. Substitute the container registry's name and the public IP address of the VM in the following command.

```
az acr network-rule add --name mycontainerregistry --ip-address <public-IP-address>
```

Continue to Verify access to the registry.

Allow access from an IP address - portal

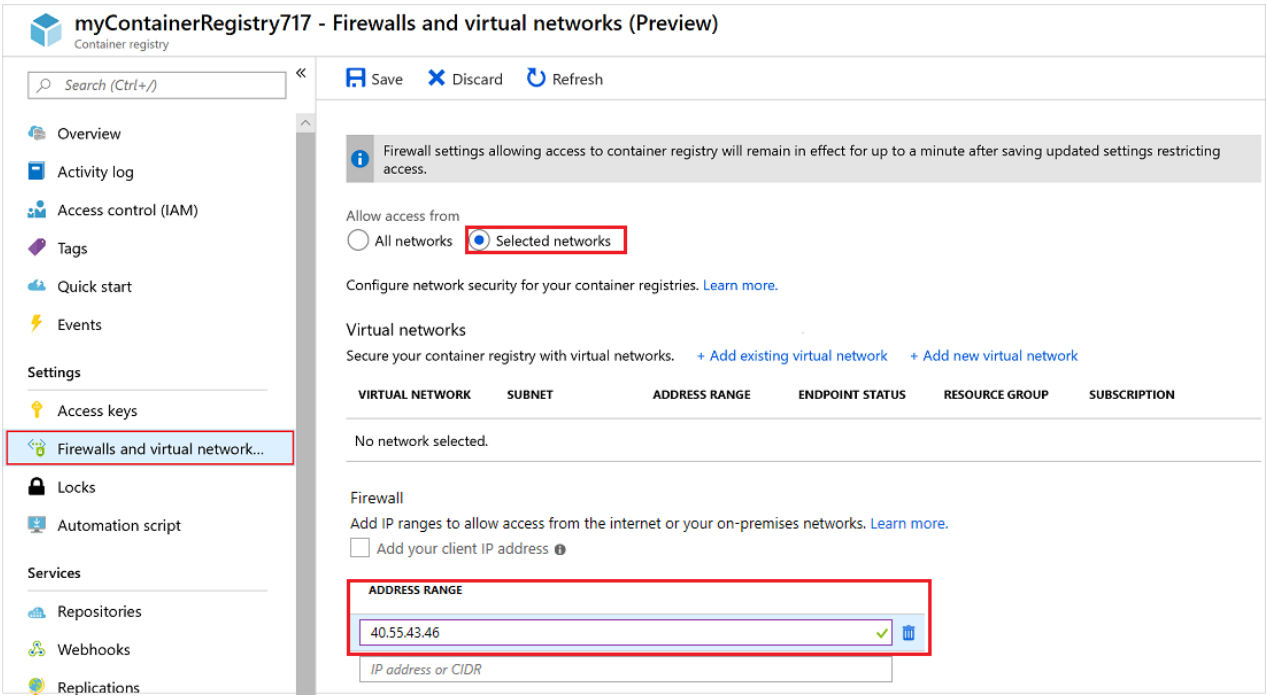
Remove existing network rule from registry

If you previously added a network rule to allow access from the VM's subnet, remove the existing rule. Skip this section if you want to access the registry from a different VM.

- Update the subnet settings to remove the subnet's service endpoint for Azure Container Registry.
    1. In the Azure portal, navigate to the virtual network where your virtual machine is deployed.
    2. Under **Settings**, select **Subnets**.
    3. Select the subnet where your virtual machine is deployed.
    4. Under **Service endpoints**, remove the checkbox for **Microsoft.ContainerRegistry**.
    5. Select **Save**.
- Remove the network rule that allows the subnet to access the registry.
    1. In the portal, navigate to your container registry.
    2. Under **Settings**, select **Firewall and virtual networks**.
    3. Under **Virtual networks**, select the name of the virtual network, and then select **Remove**.
    4. Select **Save**.

Add network rule to registry

1. In the portal, navigate to your container registry.
2. Under **Settings**, select **Firewall and virtual networks**.
3. If you haven't already done so, choose to allow access from **Selected networks**.
4. Under **Virtual networks**, ensure no network is selected.
5. Under **Firewall**, enter the public IP address of a VM. Or, enter an address range in CIDR notation that contains the VM's IP address.
6. Select **Save**.



Continue to Verify access to the registry.

Verify access to the registry

After waiting a few minutes for the configuration to update, verify that the VM can access the container registry. Make an SSH connection to your VM, and run the az acr login command to login to your registry.

```
az acr login --name mycontainerregistry
```

You can perform registry operations such as run `docker pull` to pull a sample image from the registry. Substitute an image and tag value appropriate for your registry, prefixed with the registry login server name (all lowercase):

```
docker pull mycontainerregistry.azurecr.io/hello-world:v1
```

Docker successfully pulls the image to the VM.

This example demonstrates that you can access the private container registry through the network access rule. However, the registry can't be accessed from a different login host that doesn't have a network access rule configured. If you attempt to login from another host using the `az acr login` command or `docker login` command, output is similar to the following:

```
Error response from daemon: login attempt to https://xxxxxxx.azurecr.io/v2/ failed with status: 403 Forbidden
```

Restore default registry access

To restore the registry to allow access by default, remove any network rules that are configured. Then set the default action to allow access. Equivalent steps using the Azure CLI and Azure portal are provided.

Restore default registry access - CLI

Remove network rules

To see a list of network rules configured for your registry, run the following az acr network-rule list command:

```
az acr network-rule list--name mycontainerregistry
```

For each rule that is configured, run the az acr network-rule remove command to remove it. For example:

```
# Remove a rule that allows access for a subnet. Substitute the subnet resource ID.

az acr network-rule remove \
  --name mycontainerregistry \
  --subnet /subscriptions/ \
  xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourceGroups/myResourceGroup/providers/Microsoft.Network/virtualNetworks/myDockerVMVNET/subnets
/myDockerVMSubnet

# Remove a rule that allows access for an IP address or CIDR range such as 23.45.1.0/24.

az acr network-rule remove \
  --name mycontainerregistry \
  --ip-address 23.45.1.0/24
```

Allow access

Substitute the name of your registry in the following az acr update command:

```
az acr update --name myContainerRegistry --default-action Allow
```

Restore default registry access - portal

1. In the portal, navigate to your container registry and select **Firewall and virtual networks**.
2. Under **Virtual networks**, select each virtual network, and then select **Remove**.
3. Under **Firewall**, select each address range, and then select the Delete icon.
4. Under **Allow access from**, select **All networks**.
5. Select **Save**.

Clean up resources

If you created all the Azure resources in the same resource group and no longer need them, you can optionally delete the resources by using a single az group delete command:

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

To clean up your resources in the portal, navigate to the myResourceGroup resource group. Once the resource group is loaded, click on **Delete resource group** to remove the resource group and the resources stored there.

Next steps

Several virtual network resources and features were discussed in this article, though briefly. The Azure Virtual Network documentation covers these topics extensively:

- Virtual network
- Subnet
- Service endpoints

Configure rules to access an Azure container registry behind a firewall

2/12/2020 • 3 minutes to read • Edit Online

This article explains how to configure rules on your firewall to allow access to an Azure container registry. For example, an Azure IoT Edge device behind a firewall or proxy server might need to access a container registry to pull a container image. Or, a locked-down server in an on-premises network might need access to push an image.

If instead you want to configure inbound network access rules on a container registry only within an Azure virtual network or from a public IP address range, see Restrict access to an Azure container registry from a virtual network.

About registry endpoints

To pull or push images or other artifacts to an Azure container registry, a client such as a Docker daemon needs to interact over HTTPS with two distinct endpoints.

- **Registry REST API endpoint** - Authentication and registry management operations are handled through the registry's public REST API endpoint. This endpoint is the login server name of the registry, or an associated IP address range.
- **Storage endpoint** - Azure allocates blob storage in Azure Storage accounts on behalf of each registry to manage the data for container images and other artifacts. When a client accesses image layers in an Azure container registry, it makes requests using a storage account endpoint provided by the registry.

If your registry is geo-replicated, a client might need to interact with REST and storage endpoints in a specific region or in multiple replicated regions.

Allow access to REST and storage domain names

- **REST endpoint** - Allow access to the fully qualified registry login server name, such as `myregistry.azurecr.io`
- **Storage (data) endpoint** - Allow access to all Azure blob storage accounts using the wildcard `*.blob.core.windows.net`

Allow access by IP address range

If your organization has policies to allow access only to specific IP addresses or address ranges, download Azure IP Ranges and Service Tags – Public Cloud.

To find the ACR REST endpoint IP ranges for which you need to allow access, search for **AzureContainerRegistry** in the JSON file.

> **IMPORTANT**
>
> IP address ranges for Azure services can change, and updates are published weekly. Download the JSON file regularly, and make necessary updates in your access rules. If your scenario involves configuring network security group rules in an Azure virtual network to access Azure Container Registry, use the **AzureContainerRegistry** service tag instead.

REST IP addresses for all regions

```
{
  "name": "AzureContainerRegistry",
  "id": "AzureContainerRegistry",
  "properties": {
    "changeNumber": 10,
    "region": "",
    "platform": "Azure",
    "systemService": "AzureContainerRegistry",
    "addressPrefixes": [
      "13.66.140.72/29",
    [...]
```

REST IP addresses for a specific region

Search for the specific region, such as **AzureContainerRegistry.AustraliaEast**.

```
{
  "name": "AzureContainerRegistry.AustraliaEast",
  "id": "AzureContainerRegistry.AustraliaEast",
  "properties": {
    "changeNumber": 1,
    "region": "australiaeast",
    "platform": "Azure",
    "systemService": "AzureContainerRegistry",
    "addressPrefixes": [
      "13.70.72.136/29",
    [...]
```

Storage IP addresses for all regions

```
{
  "name": "Storage",
  "id": "Storage",
  "properties": {
    "changeNumber": 19,
    "region": "",
    "platform": "Azure",
    "systemService": "AzureStorage",
    "addressPrefixes": [
      "13.65.107.32/28",
    [...]
```

Storage IP addresses for specific regions

Search for the specific region, such as **Storage.AustraliaCentral**.

```
{
  "name": "Storage.AustraliaCentral",
  "id": "Storage.AustraliaCentral",
  "properties": {
    "changeNumber": 1,
    "region": "australiacentral",
    "platform": "Azure",
    "systemService": "AzureStorage",
    "addressPrefixes": [
      "52.239.216.0/23"
    [...]
```

Allow access by service tag

In an Azure virtual network, use network security rules to filter traffic from a resource such as a virtual machine to

a container registry. To simplify the creation of the Azure network rules, use the **AzureContainerRegistry** service tag. A service tag represents a group of IP address prefixes to access an Azure service globally or per Azure region. The tag is automatically updated when addresses change.

For example, create an outbound network security group rule with destination **AzureContainerRegistry** to allow traffic to an Azure container registry. To allow access to the service tag only in a specific region, specify the region in the following format: **AzureContainerRegistry**.[*region name*].

Configure client firewall rules for MCR

If you need to access Microsoft Container Registry (MCR) from behind a firewall, see the guidance to configure MCR client firewall rules. MCR is the primary registry for all Microsoft-published docker images, such as Windows Server images.

Next steps

- Learn about Azure best practices for network security
- Learn more about security groups in an Azure virtual network

Authenticate with an Azure container registry

2/11/2020 • 4 minutes to read • Edit Online

There are several ways to authenticate with an Azure container registry, each of which is applicable to one or more registry usage scenarios.

Recommended ways include authenticating to a registry directly via individual login, or your applications and container orchestrators can perform unattended, or "headless," authentication by using an Azure Active Directory (Azure AD) service principal.

Authentication options

The following table lists available authentication methods and recommended scenarios. See linked content for details.

METHOD	HOW TO AUTHENTICATE	SCENARIOS	RBAC	LIMITATIONS
Individual AD identity	`az acr login` in Azure CLI	Interactive push/pull by developers, testers	Yes	AD token must be renewed every 3 hours
AD service principal	`docker login` `az acr login` in Azure CLI Registry login settings in APIs or tooling Kubernetes pull secret	Unattended push from CI/CD pipeline Unattended pull to Azure or external services	Yes	SP password default expiry is 1 year
Integrate with AKS	Attach registry when AKS cluster created or updated	Unattended pull to AKS cluster	No, pull access only	Only available with AKS cluster
Managed identity for Azure resources	`docker login` `az acr login` in Azure CLI	Unattended push from Azure CI/CD pipeline Unattended pull to Azure services	Yes	Use only from Azure services that support managed identities for Azure resources
Admin user	`docker login`	Interactive push/pull by individual developer or tester	No, always pull and push access	Single account per registry, not recommended for multiple users

METHOD	HOW TO AUTHENTICATE	SCENARIOS	RBAC	LIMITATIONS
Repository-scoped access token	`docker login` `az acr login` in Azure CLI	Interactive push/pull to repository by individual developer or tester Unattended push/pull to repository by individual system or external device	Yes	Not currently integrated with AD identity

Individual login with Azure AD

When working with your registry directly, such as pulling images to and pushing images from a development workstation, authenticate by using the az acr login command in the Azure CLI:

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

When you log in with `az acr login`, the CLI uses the token created when you executed az login to seamlessly authenticate your session with your registry. To complete the authentication flow, Docker must be installed and running in your environment. `az acr login` uses the Docker client to set an Azure Active Directory token in the `docker.config` file. Once you've logged in this way, your credentials are cached, and subsequent `docker` commands in your session do not require a username or password.

> **TIP**
> Also use `az acr login` to authenticate an individual identity when you want to push or pull artifacts other than Docker images to your registry, such as OCI artifacts.

For registry access, the token used by `az acr login` is valid for **3 hours**, so we recommend that you always log in to the registry before running a `docker` command. If your token expires, you can refresh it by using the `az acr login` command again to reauthenticate.

Using `az acr login` with Azure identities provides role-based access. For some scenarios, you may want to log in to a registry with your own individual identity in Azure AD. For cross-service scenarios or to handle the needs of a workgroup or a development workflow where you don't want to manage individual access, you can also log in with a managed identity for Azure resources.

Service principal

If you assign a service principal to your registry, your application or service can use it for headless authentication. Service principals allow role-based access to a registry, and you can assign multiple service principals to a registry. Multiple service principals allow you to define different access for different applications.

The available roles for a container registry include:

- **AcrPull**: pull
- **AcrPush**: pull and push
- **Owner**: pull, push, and assign roles to other users

For a complete list of roles, see Azure Container Registry roles and permissions.

For CLI scripts to create a service principal for authenticating with an Azure container registry, and more guidance, see Azure Container Registry authentication with service principals.

Admin account

Each container registry includes an admin user account, which is disabled by default. You can enable the admin user and manage its credentials in the Azure portal, or by using the Azure CLI or other Azure tools.

> **IMPORTANT**
> The admin account is designed for a single user to access the registry, mainly for testing purposes. We do not recommend sharing the admin account credentials among multiple users. All users authenticating with the admin account appear as a single user with push and pull access to the registry. Changing or disabling this account disables registry access for all users who use its credentials. Individual identity is recommended for users and service principals for headless scenarios.

The admin account is provided with two passwords, both of which can be regenerated. Two passwords allow you to maintain connection to the registry by using one password while you regenerate the other. If the admin account is enabled, you can pass the username and either password to the `docker login` command when prompted for basic authentication to the registry. For example:

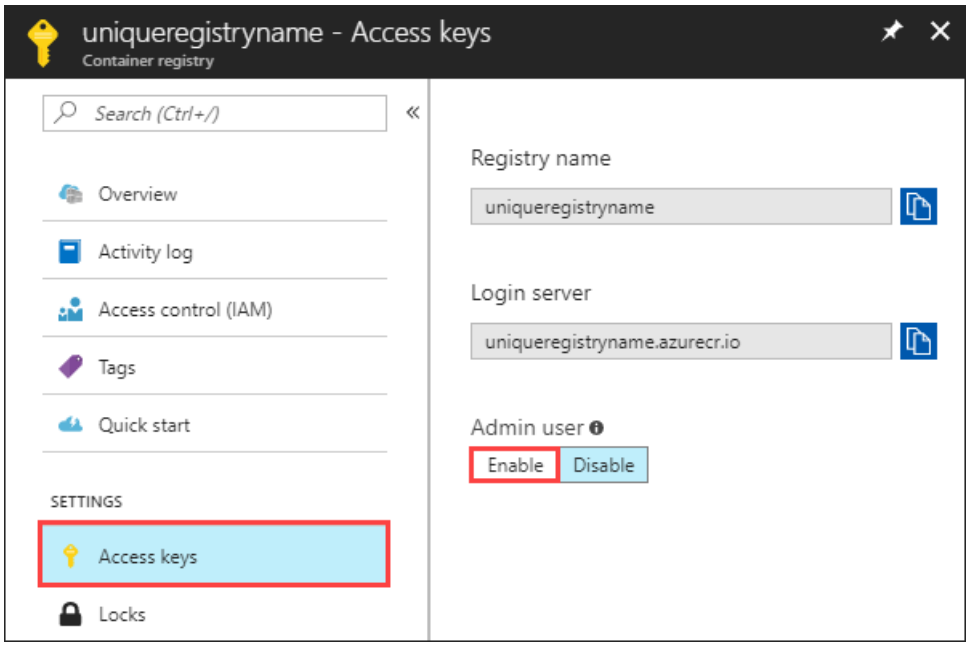
```
docker login myregistry.azurecr.io
```

For best practices to manage login credentials, see the docker login command reference.

To enable the admin user for an existing registry, you can use the `--admin-enabled` parameter of the az acr update command in the Azure CLI:

```
az acr update -n <acrName> --admin-enabled true
```

You can enable the admin user in the Azure portal by navigating your registry, selecting **Access keys** under **SETTINGS**, then **Enable** under **Admin user**.



Next steps

- Push your first image using the Azure CLI

Azure Container Registry authentication with service principals

11/24/2019 • 7 minutes to read • Edit Online

You can use an Azure Active Directory (Azure AD) service principal to provide container image `docker push` and `pull` access to your container registry. By using a service principal, you can provide access to "headless" services and applications.

What is a service principal?

Azure AD *service principals* provide access to Azure resources within your subscription. You can think of a service principal as a user identity for a service, where "service" is any application, service, or platform that needs to access the resources. You can configure a service principal with access rights scoped only to those resources you specify. Then, configure your application or service to use the service principal's credentials to access those resources.

In the context of Azure Container Registry, you can create an Azure AD service principal with pull, push and pull, or other permissions to your private registry in Azure. For a complete list, see Azure Container Registry roles and permissions.

Why use a service principal?

By using an Azure AD service principal, you can provide scoped access to your private container registry. Create different service principals for each of your applications or services, each with tailored access rights to your registry. And, because you can avoid sharing credentials between services and applications, you can rotate credentials or revoke access for only the service principal (and thus the application) you choose.

For example, configure your web application to use a service principal that provides it with image `pull` access only, while your build system uses a service principal that provides it with both `push` and `pull` access. If development of your application changes hands, you can rotate its service principal credentials without affecting the build system.

When to use a service principal

You should use a service principal to provide registry access in **headless scenarios**. That is, any application, service, or script that must push or pull container images in an automated or otherwise unattended manner. For example:

- *Pull*: Deploy containers from a registry to orchestration systems including Kubernetes, DC/OS, and Docker Swarm. You can also pull from container registries to related Azure services such as Azure Kubernetes Service (AKS), Azure Container Instances, App Service, Batch, Service Fabric, and others.
- *Push*: Build container images and push them to a registry using continuous integration and deployment solutions like Azure Pipelines or Jenkins.

For individual access to a registry, such as when you manually pull a container image to your development workstation, we recommend using your own Azure AD identity instead for registry access (for example, with az acr login).

Create a service principal

To create a service principal with access to your container registry, run the following script in the Azure Cloud Shell or a local installation of the Azure CLI. The script is formatted for the Bash shell.

Before running the script, update the `ACR_NAME` variable with the name of your container registry. The `SERVICE_PRINCIPAL_NAME` value must be unique within your Azure Active Directory tenant. If you receive an "`'http://acr-service-principal' already exists.`" error, specify a different name for the service principal.

You can optionally modify the `--role` value in the az ad sp create-for-rbac command if you want to grant different permissions. For a complete list of roles, see ACR roles and permissions.

After you run the script, take note of the service principal's **ID** and **password**. Once you have its credentials, you can configure your applications and services to authenticate to your container registry as the service principal.

```
#!/bin/bash

# Modify for your environment.
# ACR_NAME: The name of your Azure Container Registry
# SERVICE_PRINCIPAL_NAME: Must be unique within your AD tenant
ACR_NAME=<container-registry-name>
SERVICE_PRINCIPAL_NAME=acr-service-principal

# Obtain the full registry ID for subsequent command args
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Create the service principal with rights scoped to the registry.
# Default permissions are for docker pull access. Modify the '--role'
# argument value as desired:
# acrpull:     pull only
# acrpush:     push and pull
# owner:       push, pull, and assign roles
SP_PASSWD=$(az ad sp create-for-rbac --name http://$SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role
acrpull --query password --output tsv)
SP_APP_ID=$(az ad sp show --id http://$SERVICE_PRINCIPAL_NAME --query appId --output tsv)

# Output the service principal's credentials; use these in your services and
# applications to authenticate to the container registry.
echo "Service principal ID: $SP_APP_ID"
echo "Service principal password: $SP_PASSWD"
```

Use an existing service principal

To grant registry access to an existing service principal, you must assign a new role to the service principal. As with creating a new service principal, you can grant pull, push and pull, and owner access, among others.

The following script uses the az role assignment create command to grant *pull* permissions to a service principal you specify in the `SERVICE_PRINCIPAL_ID` variable. Adjust the `--role` value if you'd like to grant a different level of access.

```
#!/bin/bash

# Modify for your environment. The ACR_NAME is the name of your Azure Container
# Registry, and the SERVICE_PRINCIPAL_ID is the service principal's 'appId' or
# one of its 'servicePrincipalNames' values.
ACR_NAME=mycontainerregistry
SERVICE_PRINCIPAL_ID=<service-principal-ID>

# Populate value required for subsequent command args
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Assign the desired role to the service principal. Modify the '--role' argument
# value as desired:
# acrpull:     pull only
# acrpush:     push and pull
# owner:       push, pull, and assign roles
az role assignment create --assignee $SERVICE_PRINCIPAL_ID --scope $ACR_REGISTRY_ID --role acrpull
```

Sample scripts

You can find the preceding sample scripts for Azure CLI on GitHub, as well as versions for Azure PowerShell:

- Azure CLI
- Azure PowerShell

Authenticate with the service principal

Once you have a service principal that you've granted access to your container registry, you can configure its credentials for access to "headless" services and applications, or enter them using the `docker login` command. Use the following values:

- **User name** - service principal application ID (also called *client ID*)
- **Password** - service principal password (also called *client secret*)

Each value is a GUID of the form `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.

> **TIP**
> You can regenerate the password of a service principal by running the az ad sp reset-credentials command.

Use credentials with Azure services

You can use service principal credentials from any Azure service that authenticates with an Azure container registry. Use service principal credentials in place of the registry's admin credentials for a variety of scenarios.

For example, use the credentials to pull an image from an Azure container registry to Azure Container Instances.

Use with docker login

You can run `docker login` using a service principal. In the following example, the service principal application ID is passed in the environment variable `$SP_APP_ID`, and the password in the variable `$SP_PASSWD`. For best practices to manage Docker credentials, see the docker login command reference.

```
# Log in to Docker with service principal credentials
docker login myregistry.azurecr.io --username $SP_APP_ID --password $SP_PASSWD
```

Once logged in, Docker caches the credentials.

Use with certificate

If you've added a certificate to your service principal, you can sign into the Azure CLI with certificate-based authentication, and then use the az acr login command to access a registry. Using a certificate as a secret instead of a password provides additional security when you use the CLI.

A self-signed certificate can be created when you create a service principal. Or, add one or more certificates to an existing service principal. For example, if you use one of the scripts in this article to create or update a service principal with rights to pull or push images from a registry, add a certificate using the az ad sp credential reset command.

To use the service principal with certificate to sign into the Azure CLI, the certificate must be in PEM format and include the private key. If your certificate isn't in the required format, use a tool such as `openssl` to convert it. When you run az login to sign into the CLI using the service principal, also provide the service principal's application ID and the Active Directory tenant ID. The following example shows these values as environment variables:

```
az login --service-principal --username $SP_APP_ID --tenant $SP_TENANT_ID  --password /path/to/cert/pem/file
```

Then, run az acr login to authenticate with the registry:

```
az acr login --name myregistry
```

The CLI uses the token created when you ran `az login` to authenticate your session with the registry.

Next steps

- See the authentication overview for other scenarios to authenticate with an Azure container registry.
- For an example of using an Azure key vault to store and retrieve service principal credentials for a container registry, see the tutorial to build and deploy a container image using ACR Tasks.

Use an Azure managed identity to authenticate to an Azure container registry

11/24/2019 • 7 minutes to read • Edit Online

Use a managed identity for Azure resources to authenticate to an Azure container registry from another Azure resource, without needing to provide or manage registry credentials. For example, set up a user-assigned or system-assigned managed identity on a Linux VM to access container images from your container registry, as easily as you use a public registry.

For this article, you learn more about managed identities and how to:

- Enable a user-assigned or system-assigned identity on an Azure VM
- Grant the identity access to an Azure container registry
- Use the managed identity to access the registry and pull a container image

To create the Azure resources, this article requires that you run the Azure CLI version 2.0.55 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

To set up a container registry and push a container image to it, you must also have Docker installed locally. Docker provides packages that easily configure Docker on any macOS, Windows, or Linux system.

Why use a managed identity?

A managed identity for Azure resources provides Azure services with an automatically managed identity in Azure Active Directory (Azure AD). You can configure certain Azure resources, including virtual machines, with a managed identity. Then, use the identity to access other Azure resources, without passing credentials in code or scripts.

Managed identities are of two types:

- *User-assigned identities*, which you can assign to multiple resources and persist for as long as your want. User-assigned identities are currently in preview.
- A *system-managed identity*, which is unique to a specific resource like a single virtual machine and lasts for the lifetime of that resource.

After you set up an Azure resource with a managed identity, give the identity the access you want to another resource, just like any security principal. For example, assign a managed identity a role with pull, push and pull, or other permissions to a private registry in Azure. (For a complete list of registry roles, see Azure Container Registry roles and permissions.) You can give an identity access to one or more resources.

Then, use the identity to authenticate to any service that supports Azure AD authentication, without any credentials in your code. To use the identity to access an Azure container registry from a virtual machine, you authenticate with Azure Resource Manager. Choose how to authenticate using the managed identity, depending on your scenario:

- Acquire an Azure AD access token programmatically using HTTP or REST calls
- Use the Azure SDKs
- Sign into Azure CLI or PowerShell with the identity.

Create a container registry

If you don't already have an Azure container registry, create a registry and push a sample container image to it. For steps, see Quickstart: Create a private container registry using the Azure CLI.

This article assumes you have the `aci-helloworld:v1` container image stored in your registry. The examples use a registry name of *myContainerRegistry*. Replace with your own registry and image names in later steps.

Create a Docker-enabled VM

Create a Docker-enabled Ubuntu virtual machine. You also need to install the Azure CLI on the virtual machine. If you already have an Azure virtual machine, skip this step to create the virtual machine.

Deploy a default Ubuntu Azure virtual machine with az vm create. The following example creates a VM named *myDockerVM* in an existing resource group named *myResourceGroup*:

```
az vm create \
    --resource-group myResourceGroup \
    --name myDockerVM \
    --image UbuntuLTS \
    --admin-username azureuser \
    --generate-ssh-keys
```

It takes a few minutes for the VM to be created. When the command completes, take note of the `publicIpAddress` displayed by the Azure CLI. Use this address to make SSH connections to the VM.

Install Docker on the VM

After the VM is running, make an SSH connection to the VM. Replace *publicIpAddress* with the public IP address of your VM.

```
ssh azureuser@publicIpAddress
```

Run the following command to install Docker on the VM:

```
sudo apt install docker.io -y
```

After installation, run the following command to verify that Docker is running properly on the VM:

```
sudo docker run -it hello-world
```

Output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
[...]
```

Install the Azure CLI

Follow the steps in Install Azure CLI with apt to install the Azure CLI on your Ubuntu virtual machine. For this article, ensure that you install version 2.0.55 or later.

Exit the SSH session.

Example 1: Access with a user-assigned identity

Create an identity

Create an identity in your subscription using the az identity create command. You can use the same resource group you used previously to create the container registry or virtual machine, or a different one.

```
az identity create --resource-group myResourceGroup --name myACRId
```

To configure the identity in the following steps, use the az identity show command to store the identity's resource ID and service principal ID in variables.

```
# Get resource ID of the user-assigned identity
userID=$(az identity show --resource-group myResourceGroup --name myACRId --query id --output tsv)

# Get service principal ID of the user-assigned identity
spID=$(az identity show --resource-group myResourceGroup --name myACRId --query principalId --output tsv)
```

Because you need the identity's ID in a later step when you sign in to the CLI from your virtual machine, show the value:

```
echo $userID
```

The ID is of the form:

```
/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myACRI
d
```

Configure the VM with the identity

The following az vm identity assign command configures your Docker VM with the user-assigned identity:

```
az vm identity assign --resource-group myResourceGroup --name myDockerVM --identities $userID
```

Grant identity access to the container registry

Now configure the identity to access your container registry. First use the az acr show command to get the resource ID of the registry:

```
resourceID=$(az acr show --resource-group myResourceGroup --name myContainerRegistry --query id --output tsv)
```

Use the az role assignment create command to assign the AcrPull role to the registry. This role provides pull permissions to the registry. To provide both pull and push permissions, assign the ACRPush role.

```
az role assignment create --assignee $spID --scope $resourceID --role acrpull
```

Use the identity to access the registry

SSH into the Docker virtual machine that's configured with the identity. Run the following Azure CLI commands, using the Azure CLI installed on the VM.

First, authenticate to the Azure CLI with az login, using the identity you configured on the VM. For `<userID>`, substitute the ID of the identity you retrieved in a previous step.

```
az login --identity --username <userID>
```

Then, authenticate to the registry with az acr login. When you use this command, the CLI uses the Active Directory token created when you ran `az login` to seamlessly authenticate your session with the container registry. (Depending on your VM's setup, you might need to run this command and docker commands with `sudo`.)

```
az acr login --name myContainerRegistry
```

You should see a `Login succeeded` message. You can then run `docker` commands without providing credentials. For example, run docker pull to pull the `aci-helloworld:v1` image, specifying the login server name of your registry. The login server name consists of your container registry name (all lowercase) followed by `.azurecr.io` - for example, `mycontainerregistry.azurecr.io`.

```
docker pull mycontainerregistry.azurecr.io/aci-helloworld:v1
```

Example 2: Access with a system-assigned identity

Configure the VM with a system-managed identity

The following az vm identity assign command configures your Docker VM with a system-assigned identity:

```
az vm identity assign --resource-group myResourceGroup --name myDockerVM
```

Use the az vm show command to set a variable to the value of `principalId` (the service principal ID) of the VM's identity, to use in later steps.

```
spID=$(az vm show --resource-group myResourceGroup --name myDockerVM --query identity.principalId --out tsv)
```

Grant identity access to the container registry

Now configure the identity to access your container registry. First use the az acr show command to get the resource ID of the registry:

```
resourceID=$(az acr show --resource-group myResourceGroup --name myContainerRegistry --query id --output tsv)
```

Use the az role assignment create command to assign the AcrPull role to the identity. This role provides pull permissions to the registry. To provide both pull and push permissions, assign the ACRPush role.

```
az role assignment create --assignee $spID --scope $resourceID --role acrpull
```

Use the identity to access the registry

SSH into the Docker virtual machine that's configured with the identity. Run the following Azure CLI commands, using the Azure CLI installed on the VM.

First, authenticate the Azure CLI with az login, using the system-assigned identity on the VM.

```
az login --identity
```

Then, authenticate to the registry with az acr login. When you use this command, the CLI uses the Active Directory

token created when you ran `az login` to seamlessly authenticate your session with the container registry. (Depending on your VM's setup, you might need to run this command and docker commands with `sudo`.)

```
az acr login --name myContainerRegistry
```

You should see a `Login succeeded` message. You can then run `docker` commands without providing credentials. For example, run docker pull to pull the `aci-helloworld:v1` image, specifying the login server name of your registry. The login server name consists of your container registry name (all lowercase) followed by `.azurecr.io` - for example, `mycontainerregistry.azurecr.io`.

```
docker pull mycontainerregistry.azurecr.io/aci-helloworld:v1
```

Next steps

In this article, you learned about using managed identities with Azure Container Registry and how to:

- Enable a user-assigned or system-assigned identity in an Azure VM
- Grant the identity access to an Azure container registry
- Use the managed identity to access the registry and pull a container image

- Learn more about managed identities for Azure resources.

Authenticate with Azure Container Registry from Azure Container Instances

11/24/2019 • 3 minutes to read • Edit Online

You can use an Azure Active Directory (Azure AD) service principal to provide access to your private container registries in Azure Container Registry.

In this article, you learn to create and configure an Azure AD service principal with *pull* permissions to your registry. Then, you start a container in Azure Container Instances (ACI) that pulls its image from your private registry, using the service principal for authentication.

When to use a service principal

You should use a service principal for authentication from ACI in **headless scenarios**, such as in applications or services that create container instances in an automated or otherwise unattended manner.

For example, if you have an automated script that runs nightly and creates a task-based container instance to process some data, it can use a service principal with pull-only permissions to authenticate to the registry. You can then rotate the service principal's credentials or revoke its access completely without affecting other services and applications.

Service principals should also be used when the registry admin user is disabled.

Create a service principal

To create a service principal with access to your container registry, run the following script in the Azure Cloud Shell or a local installation of the Azure CLI. The script is formatted for the Bash shell.

Before running the script, update the `ACR_NAME` variable with the name of your container registry. The `SERVICE_PRINCIPAL_NAME` value must be unique within your Azure Active Directory tenant. If you receive an "`'http://acr-service-principal' already exists.`" error, specify a different name for the service principal.

You can optionally modify the `--role` value in the az ad sp create-for-rbac command if you want to grant different permissions. For a complete list of roles, see ACR roles and permissions.

After you run the script, take note of the service principal's **ID** and **password**. Once you have its credentials, you can configure your applications and services to authenticate to your container registry as the service principal.

```
#!/bin/bash

# Modify for your environment.
# ACR_NAME: The name of your Azure Container Registry
# SERVICE_PRINCIPAL_NAME: Must be unique within your AD tenant
ACR_NAME=<container-registry-name>
SERVICE_PRINCIPAL_NAME=acr-service-principal

# Obtain the full registry ID for subsequent command args
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Create the service principal with rights scoped to the registry.
# Default permissions are for docker pull access. Modify the '--role'
# argument value as desired:
# acrpull:     pull only
# acrpush:     push and pull
# owner:       push, pull, and assign roles
SP_PASSWD=$(az ad sp create-for-rbac --name http://$SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role
acrpull --query password --output tsv)
SP_APP_ID=$(az ad sp show --id http://$SERVICE_PRINCIPAL_NAME --query appId --output tsv)

# Output the service principal's credentials; use these in your services and
# applications to authenticate to the container registry.
echo "Service principal ID: $SP_APP_ID"
echo "Service principal password: $SP_PASSWD"
```

Use an existing service principal

To grant registry access to an existing service principal, you must assign a new role to the service principal. As with creating a new service principal, you can grant pull, push and pull, and owner access, among others.

The following script uses the az role assignment create command to grant *pull* permissions to a service principal you specify in the `SERVICE_PRINCIPAL_ID` variable. Adjust the `--role` value if you'd like to grant a different level of access.

```
#!/bin/bash

# Modify for your environment. The ACR_NAME is the name of your Azure Container
# Registry, and the SERVICE_PRINCIPAL_ID is the service principal's 'appId' or
# one of its 'servicePrincipalNames' values.
ACR_NAME=mycontainerregistry
SERVICE_PRINCIPAL_ID=<service-principal-ID>

# Populate value required for subsequent command args
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Assign the desired role to the service principal. Modify the '--role' argument
# value as desired:
# acrpull:     pull only
# acrpush:     push and pull
# owner:       push, pull, and assign roles
az role assignment create --assignee $SERVICE_PRINCIPAL_ID --scope $ACR_REGISTRY_ID --role acrpull
```

Authenticate using the service principal

To launch a container in Azure Container Instances using a service principal, specify its ID for `--registry-username`, and its password for `--registry-password`.

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer \
    --image mycontainerregistry.azurecr.io/myimage:v1 \
    --registry-login-server mycontainerregistry.azurecr.io \
    --registry-username <service-principal-ID> \
    --registry-password <service-principal-password>
```

Sample scripts

You can find the preceding sample scripts for Azure CLI on GitHub, as well versions for Azure PowerShell:

- Azure CLI
- Azure PowerShell

Next steps

The following articles contain additional details on working with service principals and ACR:

- Azure Container Registry authentication with service principals
- Authenticate with Azure Container Registry from Azure Kubernetes Service (AKS)

Pull images from an Azure container registry to a Kubernetes cluster

2/11/2020 • 4 minutes to read • Edit Online

You can use an Azure container registry as a source of container images with any Kubernetes cluster, including "local" Kubernetes clusters such as minikube and kind. This article shows how to create a Kubernetes pull secret based on an Azure Active Directory service principal. Then, use the secret to pull images from an Azure container registry in a Kubernetes deployment.

> **TIP**
>
> If you're using the managed Azure Kubernetes Service, you can also integrate your cluster with a target Azure container registry for image pulls.

This article assumes you already created a private Azure container registry. You also need to have a Kubernetes cluster running and accessible via the `kubectl` command-line tool.

Create a service principal

To create a service principal with access to your container registry, run the following script in the Azure Cloud Shell or a local installation of the Azure CLI. The script is formatted for the Bash shell.

Before running the script, update the `ACR_NAME` variable with the name of your container registry. The `SERVICE_PRINCIPAL_NAME` value must be unique within your Azure Active Directory tenant. If you receive an "`'http://acr-service-principal' already exists.`" error, specify a different name for the service principal.

You can optionally modify the `--role` value in the az ad sp create-for-rbac command if you want to grant different permissions. For a complete list of roles, see ACR roles and permissions.

After you run the script, take note of the service principal's **ID** and **password**. Once you have its credentials, you can configure your applications and services to authenticate to your container registry as the service principal.

```
#!/bin/bash

# Modify for your environment.
# ACR_NAME: The name of your Azure Container Registry
# SERVICE_PRINCIPAL_NAME: Must be unique within your AD tenant
ACR_NAME=<container-registry-name>
SERVICE_PRINCIPAL_NAME=acr-service-principal

# Obtain the full registry ID for subsequent command args
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Create the service principal with rights scoped to the registry.
# Default permissions are for docker pull access. Modify the '--role'
# argument value as desired:
# acrpull:     pull only
# acrpush:     push and pull
# owner:       push, pull, and assign roles
SP_PASSWD=$(az ad sp create-for-rbac --name http://$SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role
acrpull --query password --output tsv)
SP_APP_ID=$(az ad sp show --id http://$SERVICE_PRINCIPAL_NAME --query appId --output tsv)

# Output the service principal's credentials; use these in your services and
# applications to authenticate to the container registry.
echo "Service principal ID: $SP_APP_ID"
echo "Service principal password: $SP_PASSWD"
```

Use an existing service principal

To grant registry access to an existing service principal, you must assign a new role to the service principal. As with creating a new service principal, you can grant pull, push and pull, and owner access, among others.

The following script uses the az role assignment create command to grant *pull* permissions to a service principal you specify in the `SERVICE_PRINCIPAL_ID` variable. Adjust the `--role` value if you'd like to grant a different level of access.

```
#!/bin/bash

# Modify for your environment. The ACR_NAME is the name of your Azure Container
# Registry, and the SERVICE_PRINCIPAL_ID is the service principal's 'appId' or
# one of its 'servicePrincipalNames' values.
ACR_NAME=mycontainerregistry
SERVICE_PRINCIPAL_ID=<service-principal-ID>

# Populate value required for subsequent command args
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Assign the desired role to the service principal. Modify the '--role' argument
# value as desired:
# acrpull:     pull only
# acrpush:     push and pull
# owner:       push, pull, and assign roles
az role assignment create --assignee $SERVICE_PRINCIPAL_ID --scope $ACR_REGISTRY_ID --role acrpull
```

If you don't save or remember the service principal password, you can reset it with the az ad sp credential reset command:

```
az ad sp credential reset  --name http://<service-principal-name> --query password --output tsv
```

This command returns a new, valid password for your service principal.

Create an image pull secret

Kubernetes uses an *image pull secret* to store information needed to authenticate to your registry. To create the pull secret for an Azure container registry, you provide the service principal ID, password, and the registry URL.

Create an image pull secret with the following `kubectl` command:

```
kubectl create secret docker-registry <secret-name> \
  --namespace <namespace> \
  --docker-server=https://<container-registry-name>.azurecr.io \
  --docker-username=<service-principal-ID> \
  --docker-password=<service-principal-password>
```

where:

VALUE	DESCRIPTION
`secret-name`	Name of the image pull secret, for example, *acr-secret*
`namespace`	Kubernetes namespace to put the secret into Only needed if you want to place the secret in a namespace other than the default namespace
`container-registry-name`	Name of your Azure container registry
`service-principal-ID`	ID of the service principal that will be used by Kubernetes to access your registry
`service-principal-password`	Service principal password

Use the image pull secret

Once you've created the image pull secret, you can use it to create Kubernetes pods and deployments. Provide the name of the secret under `imagePullSecrets` in the deployment file. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: your-awesome-app-pod
  namespace: awesomeapps
spec:
  containers:
    - name: main-app-container
      image: your-awesome-app:v1
      imagePullPolicy: IfNotPresent
  imagePullSecrets:
    - name: acr-secret
```

In the preceding example, `your-awesome-app:v1` is the name of the image to pull from the Azure container registry, and `acr-secret` is the name of the pull secret you created to access the registry. When you deploy the pod, Kubernetes automatically pulls the image from your registry, if it is not already present on the cluster.

Next steps

- For more about working with service principals and Azure Container Registry, see Azure Container Registry authentication with service principals
- Learn more about image pull secrets in the Kubernetes documentation

Authenticate with Azure Container Registry from Azure Kubernetes Service

2/25/2020 • 2 minutes to read • Edit Online

When you're using Azure Container Registry (ACR) with Azure Kubernetes Service (AKS), an authentication mechanism needs to be established. This article provides examples for configuring authentication between these two Azure services.

You can set up the AKS to ACR integration in a few simple commands with the Azure CLI.

Before you begin

These examples require:

- **Owner** or **Azure account administrator** role on the **Azure subscription**
- Azure CLI version 2.0.73 or later

Create a new AKS cluster with ACR integration

You can set up AKS and ACR integration during the initial creation of your AKS cluster. To allow an AKS cluster to interact with ACR, an Azure Active Directory **service principal** is used. The following CLI command allows you to authorize an existing ACR in your subscription and configures the appropriate **ACRPull** role for the service principal. Supply valid values for your parameters below.

```
# set this to the name of your Azure Container Registry.  It must be globally unique
MYACR=myContainerRegistry

# Run the following line to create an Azure Container Registry if you do not already have one
az acr create -n $MYACR -g myContainerRegistryResourceGroup --sku basic

# Create an AKS cluster with ACR integration
az aks create -n myAKSCluster -g myResourceGroup --generate-ssh-keys --attach-acr $MYACR
```

Alternatively, you can specify the ACR name using an ACR resource ID, which has has the following format:

/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/Microsoft.ContainerRegistry/registries/<name>

```
az aks create -n myAKSCluster -g myResourceGroup --generate-ssh-keys --attach-acr
/subscriptions/<subscription-
id>/resourceGroups/myContainerRegistryResourceGroup/providers/Microsoft.ContainerRegistry/registries/myContain
erRegistry
```

This step may take several minutes to complete.

Configure ACR integration for existing AKS clusters

Integrate an existing ACR with existing AKS clusters by supplying valid values for **acr-name** or **acr-resource-id** as below.

```
az aks update -n myAKSCluster -g myResourceGroup --attach-acr <acrName>
```

or,

```
az aks update -n myAKSCluster -g myResourceGroup --attach-acr <acr-resource-id>
```

You can also remove the integration between an ACR and an AKS cluster with the following

```
az aks update -n myAKSCluster -g myResourceGroup --detach-acr <acrName>
```

or

```
az aks update -n myAKSCluster -g myResourceGroup --detach-acr <acr-resource-id>
```

Working with ACR & AKS

Import an image into your ACR

Import an image from docker hub into your ACR by running the following:

```
az acr import  -n <myContainerRegistry> --source docker.io/library/nginx:latest --image nginx:v1
```

Deploy the sample image from ACR to AKS

Ensure you have the proper AKS credentials

```
az aks get-credentials -g myResourceGroup -n myAKSCluster
```

Create a file called **acr-nginx.yaml** that contains the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx0-deployment
  labels:
    app: nginx0-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx0
  template:
    metadata:
      labels:
        app: nginx0
    spec:
      containers:
      - name: nginx
        image: <replace this image property with you acr login server, image and tag>
        ports:
        - containerPort: 80
```

Next, run this deployment in your AKS cluster:

```
kubectl apply -f acr-nginx.yaml
```

You can monitor the deployment by running:

```
kubectl get pods
```

You should have two running pods.

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx0-deployment-669dfc4d4b-x74kr   1/1     Running   0          20s
nginx0-deployment-669dfc4d4b-xdpd6   1/1     Running   0          20s
```

Azure Container Registry roles and permissions

12/6/2019 • 3 minutes to read • Edit Online

The Azure Container Registry service supports a set of built-in Azure roles that provide different levels of permissions to an Azure container registry. Use Azure role-based access control (RBAC) to assign specific permissions to users, service principals, or other identities that need to interact with a registry.

ROLE/PERMISSION	ACCESS RESOURCE MANAGER	CREATE/DELETE REGISTRY	PUSH IMAGE	PULL IMAGE	DELETE IMAGE DATA	CHANGE POLICIES	SIGN IMAGES
Owner	X	X	X	X	X	X	
Contributor	X	X	X	X	X	X	
Reader	X			X			
AcrPush			X	X			
AcrPull				X			
AcrDelete					X		
AcrImageSigner							X

Differentiate users and services

Any time permissions are applied, a best practice is to provide the most limited set of permissions for a person, or service, to accomplish a task. The following permission sets represent a set of capabilities that may be used by humans and headless services.

CI/CD solutions

When automating `docker build` commands from CI/CD solutions, you need `docker push` capabilities. For these headless service scenarios, we suggest assigning the **AcrPush** role. This role, unlike the broader **Contributor** role, prevents the account from performing other registry operations or accessing Azure Resource Manager.

Container host nodes

Likewise, nodes running your containers need the **AcrPull** role, but shouldn't require **Reader** capabilities.

Visual Studio Code Docker extension

For tools like the Visual Studio Code Docker extension, additional resource provider access is required to list the available Azure container registries. In this case, provide your users access to the **Reader** or **Contributor** role. These roles allow `docker pull`, `docker push`, `az acr list`, `az acr build`, and other capabilities.

Access Resource Manager

Azure Resource Manager access is required for the Azure portal and registry management with the Azure CLI. For example, to get a list of registries by using the `az acr list` command, you need this permission set.

Create and delete registry

The ability to create and delete Azure container registries.

Push image

The ability to `docker push` an image, or push another supported artifact such as a Helm chart, to a registry. Requires authentication with the registry using the authorized identity.

Pull image

The ability to `docker pull` a non-quarantined image, or pull another supported artifact such as a Helm chart, from a registry. Requires authentication with the registry using the authorized identity.

Delete image data

The ability to delete container images, or delete other supported artifacts such as Helm charts, from a registry.

Change policies

The ability to configure policies on a registry. Policies include image purging, enabling quarantine, and image signing.

Sign images

The ability to sign images, usually assigned to an automated process, which would use a service principal. This permission is typically combined with push image to allow pushing a trusted image to a registry. For details, see Content trust in Azure Container Registry.

Custom roles

As with other Azure resources, you can create your own custom roles with fine-grained permissions to Azure Container Registry. Then assign the custom roles to users, service principals, or other identities that need to interact with a registry.

To determine which permissions to apply to a custom role, see the list of Microsoft.ContainerRegistry actions, review the permitted actions of the built-in ACR roles, or run the following command:

```
az provider operation show --namespace Microsoft.ContainerRegistry
```

To define a custom role, see Steps to create a custom role.

> **IMPORTANT**
> In a custom role, Azure Container Registry doesn't currently support wildcards such as `Microsoft.ContainerRegistry/*` or `Microsoft.ContainerRegistry/registries/*` that grant access to all matching actions. Specify any required action individually in the role.

Next steps

- Learn more about assigning RBAC roles to an Azure identity by using the Azure portal, the Azure CLI, or other Azure tools.
- Learn about authentication options for Azure Container Registry.

- Learn about enabling repository-scoped permissions (preview) in a container registry.

Create a token with repository-scoped permissions

2/18/2020 • 12 minutes to read • Edit Online

This article describes how to create tokens and scope maps to manage repository-scoped permissions in your container registry. By creating tokens, a registry owner can provide users or services with scoped, time-limited access to repositories to pull or push images or perform other actions. A token provides more fine-grained permissions than other registry authentication options, which scope permissions to an entire registry.

Scenarios for creating a token include:

- Allow IoT devices with individual tokens to pull an image from a repository
- Provide an external organization with permissions to a specific repository
- Limit repository access to different user groups in your organization. For example, provide write and read access to developers who build images that target specific repositories, and read access to teams that deploy from those repositories.

> **IMPORTANT**
> This feature is currently in preview, and some limitations apply. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

Preview limitations

- This feature is only available in a **Premium** container registry. For information about registry service tiers and limits, see Azure Container Registry SKUs.
- You can't currently assign repository-scoped permissions to an Azure Active Directory identity, such as a service principal or managed identity.

Concepts

To configure repository-scoped permissions, you create a *token* with an associated *scope map*.

- A **token** along with a generated password lets the user authenticate with the registry. You can set an expiration date for a token password, or disable a token at any time.

  After authenticating with a token, the user or service can perform one or more *actions* scoped to one or more repositories.

ACTION	DESCRIPTION	EXAMPLE
`content/delete`	Remove data from the repository	Delete a repository or a manifest
`content/read`	Read data from the repository	Pull an artifact
`content/write`	Write data to the repository	Use with `content/read` to push an artifact
`metadata/read`	Read metadata from the repository	List tags or manifests

ACTION	DESCRIPTION	EXAMPLE
`metadata/write`	Write metadata to the repository	Enable or disable read, write, or delete operations

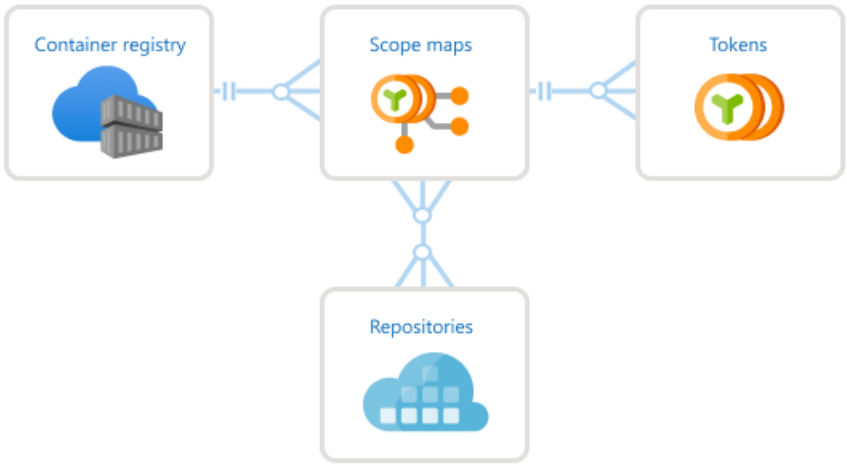
- A **scope map** groups the repository permissions you apply to a token, and can reapply to other tokens. Every token is associated with a single scope map.

  With a scope map:

  - Configure multiple tokens with identical permissions to a set of repositories
  - Update token permissions when you add or remove repository actions in the scope map, or apply a different scope map

  Azure Container Registry also provides several system-defined scope maps you can apply, with fixed permissions across all repositories.

The following image shows the relationship between tokens and scope maps.



Prerequisites

- **Azure CLI** - Azure CLI commands to create and manage tokens are available in Azure CLI version 2.0.76 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.
- **Docker** - To authenticate with the registry to pull or push images, you need a local Docker installation. Docker provides installation instructions for macOS, Windows, and Linux systems.
- **Container registry** - If you don't have one, create a Premium container registry in your Azure subscription, or upgrade an existing registry. For example, use the Azure portal or the Azure CLI.

Create token - CLI

Create token and specify repositories

Create a token using the az acr token create command. When creating a token, you can specify one or more repositories and associated actions on each repository. The repositories don't need to be in the registry yet. To create a token by specifying an existing scope map, see the next section.

The following example creates a token in the registry *myregistry* with the following permissions on the `samples/hello-world` repo: `content/write` and `content/read`. By default, the command sets the default token status to `enabled`, but you can update the status to `disabled` at any time.

```
az acr token create --name MyToken --registry myregistry \
  --repository samples/hello-world \
  content/write content/read
```

The output shows details about the token, including two generated passwords. It's recommended to save the passwords in a safe place to use later for authentication. The passwords can't be retrieved again, but new ones can be generated.


```
{
  "creationDate": "2020-01-18T00:15:34.066221+00:00",
  "credentials": {
    "certificates": [],
    "passwords": [
      {
        "creationTime": "2020-01-18T00:15:52.837651+00:00",
        "expiry": null,
        "name": "password1",
        "value": "uH54BxxxxK7KOxxxxRbr26dAs8JXxxxx"
      },
      {
        "creationTime": "2020-01-18T00:15:52.837651+00:00",
        "expiry": null,
        "name": "password2",
        "value": "kPX6Or/xxxxLXpqowxxxxkA0idwLtmxxxx"
      }
    ],
    "username": "MyToken"
  },
  "id": "/subscriptions/xxxxxxxx-adbd-4cb4-c864-
xxxxxxxxxxxx/resourceGroups/myresourcegroup/providers/Microsoft.ContainerRegistry/registries/myregistry/tokens
/MyToken",
  "name": "MyToken",
  "objectId": null,
  "provisioningState": "Succeeded",
  "resourceGroup": "myresourcegroup",
  "scopeMapId": "/subscriptions/xxxxxxxx-adbd-4cb4-c864-
xxxxxxxxxxxx/resourceGroups/myresourcegroup/providers/Microsoft.ContainerRegistry/registries/myregistry/scopeM
aps/MyToken-scope-map",
  "status": "enabled",
  "type": "Microsoft.ContainerRegistry/registries/tokens"
```


The output includes details about the scope map the command created. You can use the scope map, here named `MyToken-scope-map`, to apply the same repository actions to other tokens. Or, update the scope map later to change the permissions of the associated tokens.

Create token and specify scope map

An alternative way to create a token is to specify an existing scope map. If you don't already have a scope map, first create one by specifying repositories and associated actions. Then, specify the scope map when creating a token.

To create a scope map, use the az acr scope-map create command. The following command creates a scope map with the same permissions on the `samples/hello-world` repository used previously.

```
az acr scope-map create --name MyScopeMap --registry myregistry \
  --repository samples/hello-world \
  content/write content/read \
  --description "Sample scope map"
```

Run az acr token create to create a token, specifying the *MyScopeMap* scope map. As in the previous example, the

command sets the default token status to `enabled`.

```
az acr token create --name MyToken \
  --registry myregistry \
  --scope-map MyScopeMap
```

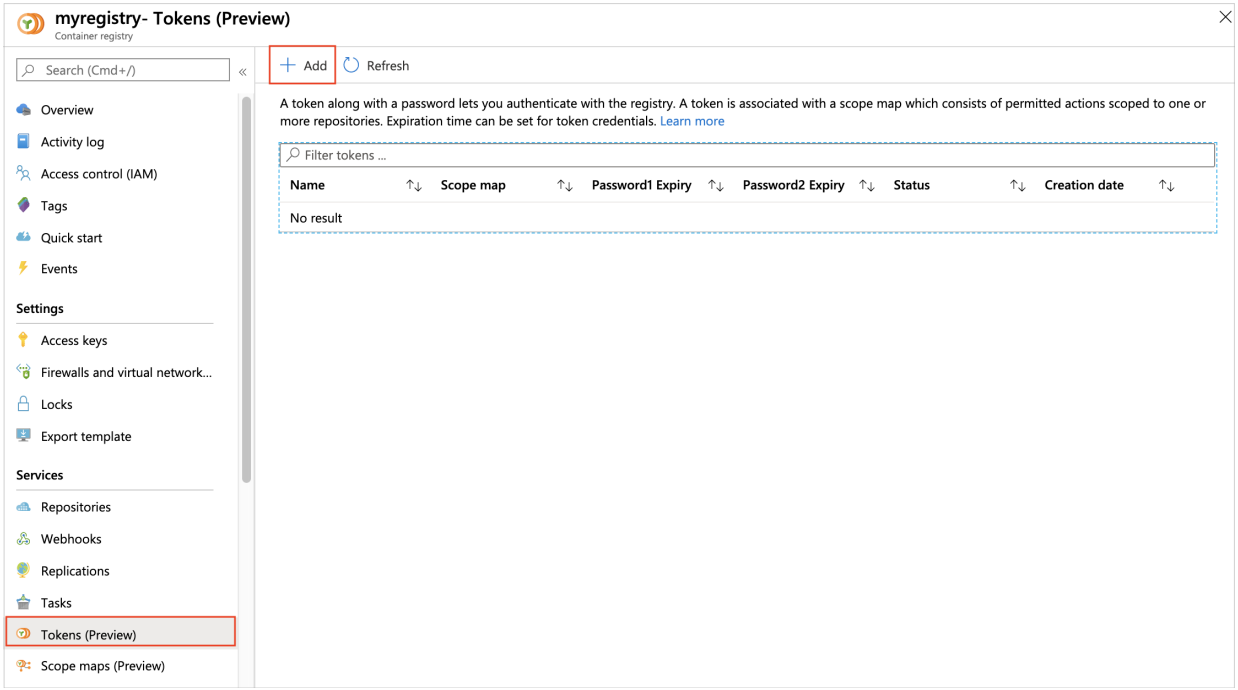
The output shows details about the token, including two generated passwords. It's recommended to save the passwords in a safe place to use later for authentication. The passwords can't be retrieved again, but new ones can be generated.

Create token - portal

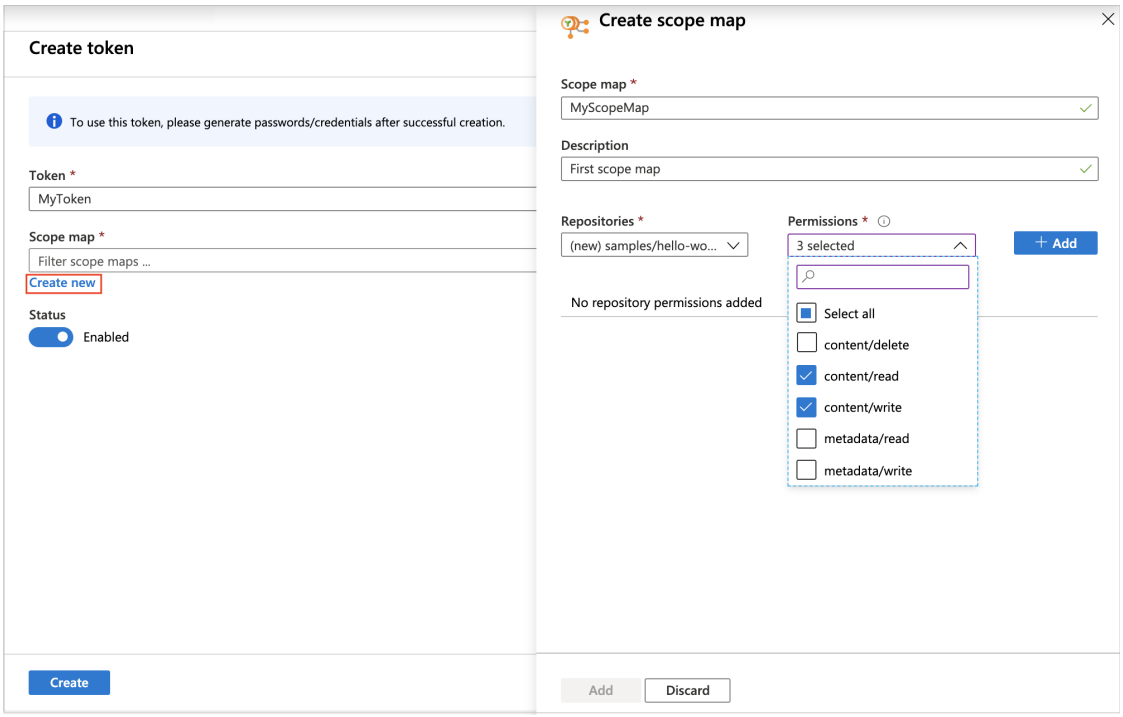
You can use the Azure portal to create tokens and scope maps. As with the `az acr token create` CLI command, you can apply an existing scope map, or create a scope map when you create a token by specifying one or more repositories and associated actions. The repositories don't need to be in the registry yet.

The following example creates a token, and creates a scope map with the following permissions on the `samples/hello-world` repository: `content/write` and `content/read`.

1. In the portal, navigate to your container registry.
2. Under **Services**, select **Tokens (Preview) > +Add**.



3. Enter a token name.
4. Under **Scope map**, select **Create new**.
5. Configure the scope map:
   a. Enter a name and description for the scope map.
   b. Under **Repositories**, enter `samples/hello-world`, and under **Permissions**, select `content/read` and `content/write`. Then select **+Add**.



c. After adding repositories and permissions, select **Add** to add the scope map.

6. Accept the default token **Status** of **Enabled** and then select **Create**.

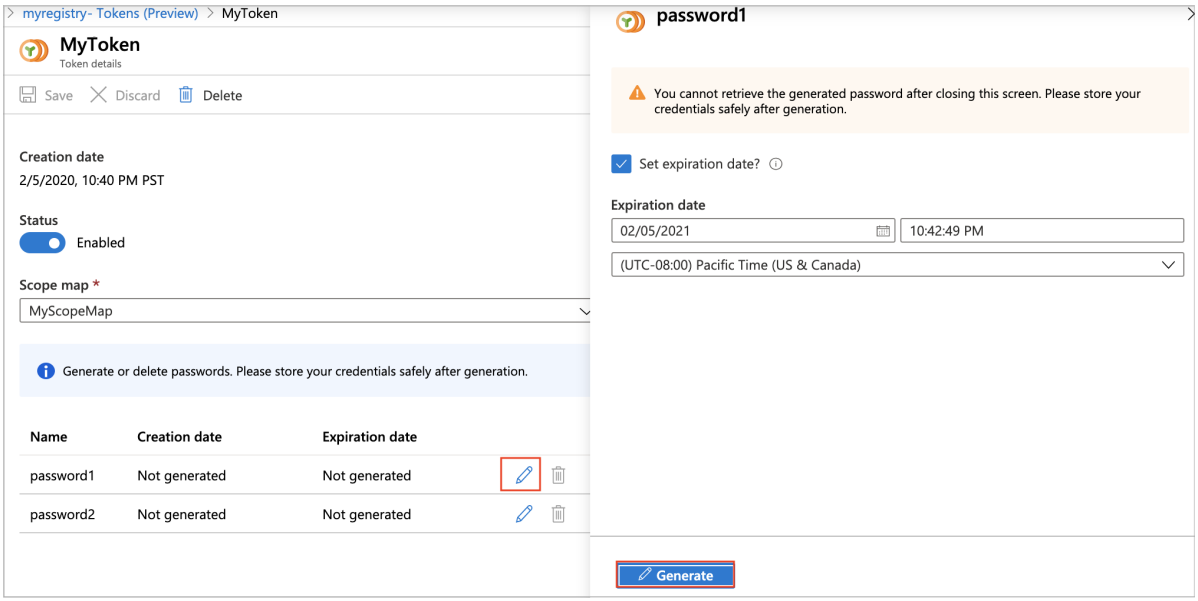
After the token is validated and created, token details appear in the **Tokens** screen.

Add token password

Generate a password after you create a token. To authenticate with the registry, the token must be enabled and have a valid password.

You can generate one or two passwords, and set an expiration date for each one.

1. In the portal, navigate to your container registry.
2. Under **Services**, select **Tokens (Preview)**, and select a token.
3. In the token details, select **password1** or **password2**, and select the Generate icon.
4. In the password screen, optionally set an expiration date for the password, and select **Generate**.
5. After generating a password, copy and save it to a safe location. You can't retrieve a generated password after closing the screen, but you can generate a new one.



Authenticate with token

When a user or service uses a token to authenticate with the target registry, it provides the token name as a user name and one of its generated passwords. The authentication method depends on the configured action or actions associated with the token.

ACTION	HOW TO AUTHENTICATE
`content/delete`	`az acr repository delete` in Azure CLI
`content/read`	`docker login` `az acr login` in Azure CLI
`content/write`	`docker login` `az acr login` in Azure CLI

`metadata/read`	`az acr repository show`
	`az acr repository show-tags`
	`az acr repository show-manifests` in Azure CLI

`metadata/write`	`az acr repository untag` `az acr repository update` in Azure CLI

Examples: Use token

The following examples use the token created earlier in this article to perform common operations on a repository: push and pull images, delete images, and list repository tags. The token was set up initially with push permissions (`content/write` and `content/read` actions) on the `samples/hello-world` repository.

Pull and tag test images

For the following examples, pull the `hello-world` and `alpine` images from Docker Hub, and tag them for your registry and repository.

```
docker pull hello-world
docker pull alpine
docker tag hello-world myregistry.azurecr.io/samples/hello-world:v1
docker tag hello-world myregistry.azurecr.io/samples/alpine:v1
```

Authenticate using token

Run `docker login` to authenticate with the registry, Provide the token name as the user name, and provide one of its passwords. The token must have the `Enabled` status.

The following example is formatted for the bash shell, and provides the values using environment variables.

```
TOKEN_NAME=MyToken
TOKEN_PWD=<token password>

echo $TOKEN_PWD | docker login --username $TOKEN_NAME --password-stdin myregistry.azurecr.io
```

Output should show successful authentication:

```
Login Succeeded
```

Push images to registry

After successful login, attempt to push the tagged images to the registry. Because the token has permissions to push images to the `samples/hello-world` repository, the following push succeeds:

```
docker push myregistry.azurecr.io/samples/hello-world:v1
```

The token doesn't have permissions to the `samples/alpine` repo, so the following push attempt fails with an error similar to `requested access to the resource is denied`:

```
docker push myregistry.azurecr.io/samples/alpine:v1
```

Change push/pull permissions

To update the permissions of a token, update the permissions in the associated scope map. The updated scope map is applied immediately to all associated tokens.

For example, update `MyToken-scope-map` with `content/write` and `content/read` actions on the `samples/alpine` repository, and remove the `content/write` action on the `samples/hello-world` repository.

To use the Azure CLI, run az acr scope-map update to update the scope map:

```
az acr scope-map update \
  --name MyScopeMap \
  --registry myregistry \
  --add samples/alpine content/write content/read \
  --remove samples/hello-world content/write
```

In the Azure portal:

1. Navigate to your container registry.
2. Under **Services**, select **Scope maps (Preview)**, and select the scope map to update.
3. Under **Repositories**, enter `samples/alpine`, and under **Permissions**, select `content/read` and `content/write`. Then select **+Add**.

4. Under **Repositories**, select `samples/hello-world` and under **Permissions**, deselect `content/write`. Then select **Save**.

After updating the scope map, the following push succeeds:

```
docker push myregistry.azurecr.io/samples/alpine:v1
```

Because the scope map only has the `content/read` permission on the `samples/hello-world` repository, a push attempt to the `samples/hello-world` repo now fails:

```
docker push myregistry.azurecr.io/samples/hello-world:v1
```

Pulling images from both repos succeeds, because the scope map provides `content/read` permissions on both repositories:

```
docker pull myregistry.azurecr.io/samples/alpine:v1
docker pull myregistry.azurecr.io/samples/hello-world:v1
```

Delete images

Update the scope map by adding the `content/delete` action to the `alpine` repository. This action allows deletion of images in the repository, or deletion of the entire repository.

For brevity, we show only the az acr scope-map update command to update the scope map:

```
az acr scope-map update \
  --name MyScopeMap \
  --registry myregistry \
  --add samples/alpine content/delete
```

To update the scope map using the portal, see the preceding section.

Use the following az acr repository delete command to delete the `samples/alpine` repository. To delete images or repositories, the token doesn't authenticate through `docker login`. Instead, pass the token's name and password to the command. The following example uses the environment variables created earlier in the article:

```
az acr repository delete \
  --name myregistry --repository samples/alpine \
  --username $TOKEN_NAME --password $TOKEN_PWD
```

Show repo tags

Update the scope map by adding the `metadata/read` action to the `hello-world` repository. This action allows reading manifest and tag data in the repository.

For brevity, we show only the az acr scope-map update command to update the scope map:

```
az acr scope-map update \
  --name MyScopeMap \
  --registry myregistry \
  --add samples/hello-world metadata/read
```

To update the scope map using the portal, see the preceding section.

To read metadata in the `samples/hello-world` repository, run the az acr repository show-manifests or az acr

repository show-tags command.

To read metadata, the token doesn't authenticate through `docker login`. Instead, pass the token's name and password to either command. The following example uses the environment variables created earlier in the article:

```
az acr repository show-tags \
  --name myregistry --repository samples/hello-world \
  --username $TOKEN_NAME --password $TOKEN_PWD
```

Sample output:

```
[
  "v1"
]
```

Manage tokens and scope maps

List scope maps

Use the az acr scope-map list command, or the **Scope maps (Preview)** screen in the portal, to list all the scope maps configured in a registry. For example:

```
az acr scope-map list \
  --registry myregistry --output table
```

The output shows the scope maps you defined and several system-defined scope maps you can use to configure tokens:

```
NAME                 TYPE           CREATION DATE         DESCRIPTION
-------------------  -------------  --------------------  ------------------------------------------------
--------
_repositories_admin  SystemDefined  2020-01-20T09:44:24Z  Can perform all read, write and delete operations on
the ...
_repositories_pull   SystemDefined  2020-01-20T09:44:24Z  Can pull any repository of the registry
_repositories_push   SystemDefined  2020-01-20T09:44:24Z  Can push to any repository of the registry
MyScopeMap           UserDefined    2019-11-15T21:17:34Z  Sample scope map
```

Show token details

To view the details of a token, such as its status and password expiration dates, run the az acr token show command, or select the token in the **Tokens (Preview)** screen in the portal. For example:

```
az acr scope-map show \
  --name MyScopeMap --registry myregistry
```

Use the az acr token list command, or the **Tokens (Preview)** screen in the portal, to list all the tokens configured in a registry. For example:

```
az acr token list --registry myregistry --output table
```

Generate passwords for token

If you don't have a token password, or you want to generate new passwords, run the az acr token credential generate command.

The following example generates a new value for password1 for the *MyToken* token, with an expiration period of 30 days. It stores the password in the environment variable `TOKEN_PWD`. This example is formatted for the bash shell.

```
TOKEN_PWD=$(az acr token credential generate \
  --name MyToken --registry myregistry --days 30 \
  --password1 --query 'passwords[0].value' --output tsv)
```

To use the Azure portal to generate a token password, see the steps in Create token - portal earlier in this article.

Update token with new scope map

If you want to update a token with a different scope map, run az acr token update and specify the new scope map. For example:

```
az acr token update --name MyToken --registry myregistry \
  --scope-map MyNewScopeMap
```

In the portal, on the **Tokens (preview)** screen, select the token, and under **Scope map**, select a different scope map.

> **TIP**
> After updating a token with a new scope map, you might want to generate new token passwords. Use the az acr token credential generate command or regenerate a token password in the Azure portal.

Disable or delete token

You might need to temporarily disable use of the token credentials for a user or service.

Using the Azure CLI, run the az acr token update command to set the `status` to `disabled`:

```
az acr token update --name MyToken --registry myregistry \
  --status disabled
```

In the portal, select the token in the **Tokens (Preview)** screen, and select **Disabled** under **Status**.

To delete a token to permanently invalidate access by anyone using its credentials, run the az acr token delete command.

```
az acr token delete --name MyToken --registry myregistry
```

In the portal, select the token in the **Tokens (Preview)** screen, and select **Discard**.

Next steps

- To manage scope maps and tokens, use additional commands in the az acr scope-map and az acr token command groups.
- See the authentication overview for other options to authenticate with an Azure container registry, including using an Azure Active Directory identity, a service principal, or an admin account.

Content trust in Azure Container Registry

11/24/2019 • 9 minutes to read • Edit Online

Azure Container Registry implements Docker's content trust model, enabling pushing and pulling of signed images. This article gets you started enabling content trust in your container registries.

> **NOTE**
> Content trust is a feature of the Premium SKU of Azure Container Registry.

How content trust works

Important to any distributed system designed with security in mind is verifying both the *source* and the *integrity* of data entering the system. Consumers of the data need to be able to verify both the publisher (source) of the data, as well as ensure it's not been modified after it was published (integrity).

As an image publisher, content trust allows you to **sign** the images you push to your registry. Consumers of your images (people or systems pulling images from your registry) can configure their clients to pull *only* signed images. When an image consumer pulls a signed image, their Docker client verifies the integrity of the image. In this model, consumers are assured that the signed images in your registry were indeed published by you, and that they've not been modified since being published.

Trusted images

Content trust works with the **tags** in a repository. Image repositories can contain images with both signed and unsigned tags. For example, you might sign only the `myimage:stable` and `myimage:latest` images, but not `myimage:dev`.

Signing keys

Content trust is managed through the use of a set of cryptographic signing keys. These keys are associated with a specific repository in a registry. There are several types of signing keys that Docker clients and your registry use in managing trust for the tags in a repository. When you enable content trust and integrate it into your container publishing and consumption pipeline, you must manage these keys carefully. For more information, see Key management later in this article and Manage keys for content trust in the Docker documentation.

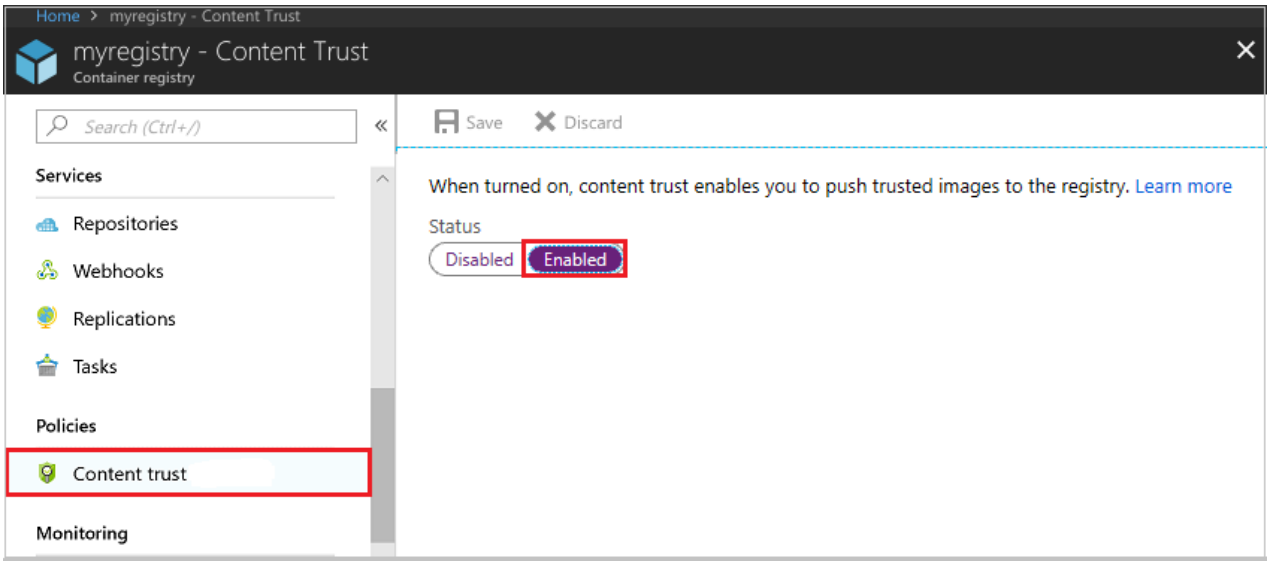
> **TIP**
> This was a very high-level overview of Docker's content trust model. For an in-depth discussion of content trust, see Content trust in Docker.

Enable registry content trust

Your first step is to enable content trust at the registry level. Once you enable content trust, clients (users or services) can push signed images to your registry. Enabling content trust on your registry does not restrict registry usage only to consumers with content trust enabled. Consumers without content trust enabled can continue to use your registry as normal. Consumers who have enabled content trust in their clients, however, will be able to see *only* signed images in your registry.

To enable content trust for your registry, first navigate to the registry in the Azure portal. Under **Policies**, select **Content Trust** > **Enabled** > **Save**. You can also use the az acr config content-trust update command in the Azure

CLI.



Enable client content trust

To work with trusted images, both image publishers and consumers need to enable content trust for their Docker clients. As a publisher, you can sign the images you push to a content trust-enabled registry. As a consumer, enabling content trust limits your view of a registry to signed images only. Content trust is disabled by default in Docker clients, but you can enable it per shell session or per command.

To enable content trust for a shell session, set the `DOCKER_CONTENT_TRUST` environment variable to **1**. For example, in the Bash shell:

```
# Enable content trust for shell session
export DOCKER_CONTENT_TRUST=1
```

If instead you'd like to enable or disable content trust for a single command, several Docker commands support the `--disable-content-trust` argument. To enable content trust for a single command:

```
# Enable content trust for single command
docker build --disable-content-trust=false -t myacr.azurecr.io/myimage:v1 .
```

If you've enabled content trust for your shell session and want to disable it for a single command:

```
# Disable content trust for single command
docker build --disable-content-trust -t myacr.azurecr.io/myimage:v1 .
```

Grant image signing permissions

Only the users or systems you've granted permission can push trusted images to your registry. To grant trusted image push permission to a user (or a system using a service principal), grant their Azure Active Directory identities the `AcrImageSigner` role. This is in addition to the `AcrPush` (or equivalent) role required for pushing images to the registry. For details, see Azure Container Registry roles and permissions.

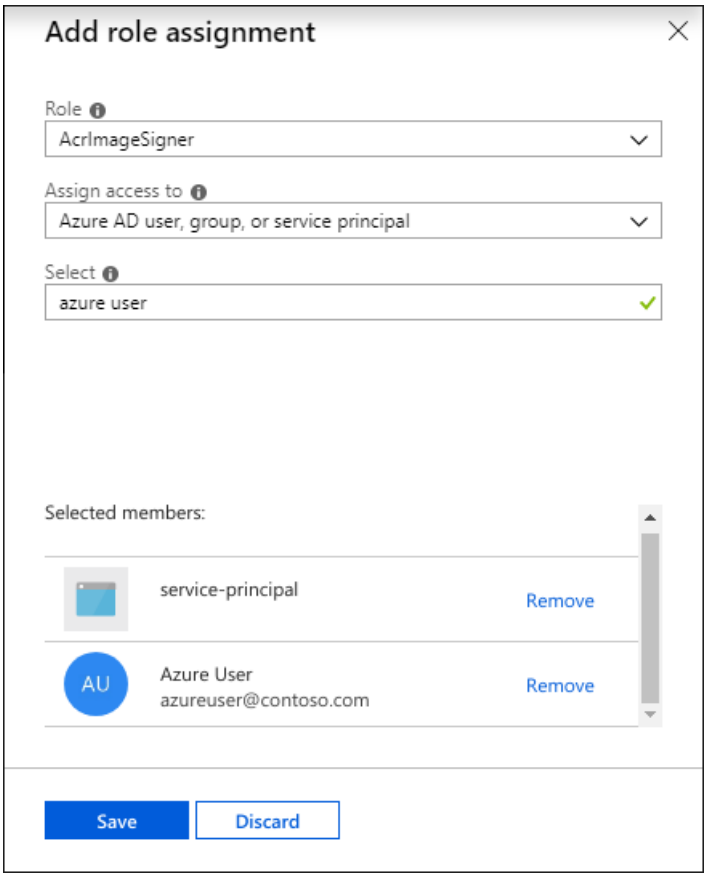
> **NOTE**
> You can't grant trusted image push permission to the admin account of an Azure container registry.

Details for granting the `AcrImageSigner` role in the Azure portal and the Azure CLI follow.

Azure portal

Navigate to your registry in the Azure portal, then select **Access control (IAM)** > **Add role assignment**. Under **Add role assignment**, select `AcrImageSigner` under **Role**, then **Select** one or more users or service principals, then **Save**.

In this example, two entities have been assigned the `AcrImageSigner` role: a service principal named "service-principal," and a user named "Azure User."



Azure CLI

To grant signing permissions to a user with the Azure CLI, assign the `AcrImageSigner` role to the user, scoped to your registry. The format of the command is:

```
az role assignment create --scope <registry ID> --role AcrImageSigner --assignee <user name>
```

For example, to grant yourself the role, you can run the following commands in an authenticated Azure CLI session. Modify the `REGISTRY` value to reflect the name of your Azure container registry.

```
# Grant signing permissions to authenticated Azure CLI user
REGISTRY=myregistry
USER=$(az account show --query user.name --output tsv)
REGISTRY_ID=$(az acr show --name $REGISTRY --query id --output tsv)

az role assignment create --scope $REGISTRY_ID --role AcrImageSigner --assignee $USER
```

You can also grant a service principal the rights to push trusted images to your registry. Using a service principal is useful for build systems and other unattended systems that need to push trusted images to your registry. The format is similar to granting a user permission, but specify a service principal ID for the `--assignee` value.

```
az role assignment create --scope $REGISTRY_ID --role AcrImageSigner --assignee <service principal ID>
```

The `<service principal ID>` can be the service principal's **appId**, **objectId**, or one of its **servicePrincipalNames**. For more information about working with service principals and Azure Container Registry, see Azure Container Registry authentication with service principals.

> **IMPORTANT**
> After any role changes, run `az acr login` to refresh the local identity token for the Azure CLI so that the new roles can take effect. For information about verifying roles for an identity, see Manage access to Azure resources using RBAC and Azure CLI and Troubleshoot RBAC for Azure resources.

Push a trusted image

To push a trusted image tag to your container registry, enable content trust and push the image with `docker push`. The first time you push a signed tag, you're asked to create a passphrase for both a root signing key and a repository signing key. Both the root and repository keys are generated and stored locally on your machine.

```
$ docker push myregistry.azurecr.io/myimage:v1
The push refers to repository [myregistry.azurecr.io/myimage]
ee83fc5847cb: Pushed
v1: digest: sha256:aca41a608e5eb015f1ec6755f490f3be26b48010b178e78c00eac21ffbe246f1 size: 524
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 4c6c56a:
Repeat passphrase for new root key with ID 4c6c56a:
Enter passphrase for new repository key with ID bcd6d98:
Repeat passphrase for new repository key with ID bcd6d98:
Finished initializing "myregistry.azurecr.io/myimage"
Successfully signed myregistry.azurecr.io/myimage:v1
```

After your first `docker push` with content trust enabled, the Docker client uses the same root key for subsequent pushes. On each subsequent push to the same repository, you're asked only for the repository key. Each time you push a trusted image to a new repository, you're asked to supply a passphrase for a new repository key.

Pull a trusted image

To pull a trusted image, enable content trust and run the `docker pull` command as normal. To pull trusted images, the `AcrPull` role is enough for normal users. No additional roles like an `AcrImageSigner` role are required. Consumers with content trust enabled can pull only images with signed tags. Here's an example of pulling a signed tag:

```
$ docker pull myregistry.azurecr.io/myimage:signed
Pull (1 of 1):
myregistry.azurecr.io/myimage:signed@sha256:0800d17e37fb4f8194495b1a188f121e5b54efb52b5d93dc9e0ed97fce49564b
sha256:0800d17e37fb4f8194495b1a188f121e5b54efb52b5d93dc9e0ed97fce49564b: Pulling from myimage
8e3ba11ec2a2: Pull complete
Digest: sha256:0800d17e37fb4f8194495b1a188f121e5b54efb52b5d93dc9e0ed97fce49564b
Status: Downloaded newer image for
myregistry.azurecr.io/myimage@sha256:0800d17e37fb4f8194495b1a188f121e5b54efb52b5d93dc9e0ed97fce49564b
Tagging myregistry.azurecr.io/myimage@sha256:0800d17e37fb4f8194495b1a188f121e5b54efb52b5d93dc9e0ed97fce49564b
as myregistry.azurecr.io/myimage:signed
```

If a client with content trust enabled tries to pull an unsigned tag, the operation fails:

```
$ docker pull myregistry.azurecr.io/myimage:unsigned
No valid trust data for unsigned
```

Behind the scenes

When you run `docker pull`, the Docker client uses the same library as in the Notary CLI to request the tag-to-SHA-256 digest mapping for the tag you're pulling. After validating the signatures on the trust data, the client instructs Docker Engine to do a "pull by digest." During the pull, the Engine uses the SHA-256 checksum as a content address to request and validate the image manifest from the Azure container registry.

Key management

As stated in the `docker push` output when you push your first trusted image, the root key is the most sensitive. Be sure to back up your root key and store it in a secure location. By default, the Docker client stores signing keys in the following directory:

```
~/.docker/trust/private
```

Back up your root and repository keys by compressing them in an archive and storing it in a secure location. For example, in Bash:

```
umask 077; tar -zcvf docker_private_keys_backup.tar.gz ~/.docker/trust/private; umask 022
```

Along with the locally generated root and repository keys, several others are generated and stored by Azure Container Registry when you push a trusted image. For a detailed discussion of the various keys in Docker's content trust implementation, including additional management guidance, see Manage keys for content trust in the Docker documentation.

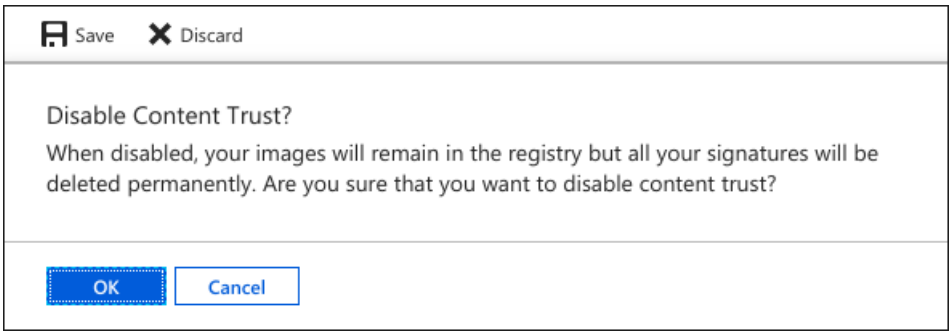
Lost root key

If you lose access to your root key, you lose access to the signed tags in any repository whose tags were signed with that key. Azure Container Registry cannot restore access to image tags signed with a lost root key. To remove all trust data (signatures) for your registry, first disable, then re-enable content trust for the registry.

> **WARNING**
> Disabling and re-enabling content trust in your registry **deletes all trust data for all signed tags in every repository in your registry**. This action is irreversible--Azure Container Registry cannot recover deleted trust data. Disabling content trust does not delete the images themselves.

To disable content trust for your registry, navigate to the registry in the Azure portal. Under **Policies**, select

Content Trust > **Disabled** > **Save**. You're warned of the loss of all signatures in the registry. Select **OK** to permanently delete all signatures in your registry.



Next steps

- See Content trust in Docker for additional information about content trust. While several key points were touched on in this article, content trust is an extensive topic and is covered more in-depth in the Docker documentation.
- See the Azure Pipelines documentation for an example of using content trust when you build and push a Docker image.

Azure Container Registry integration with Security Center (Preview)

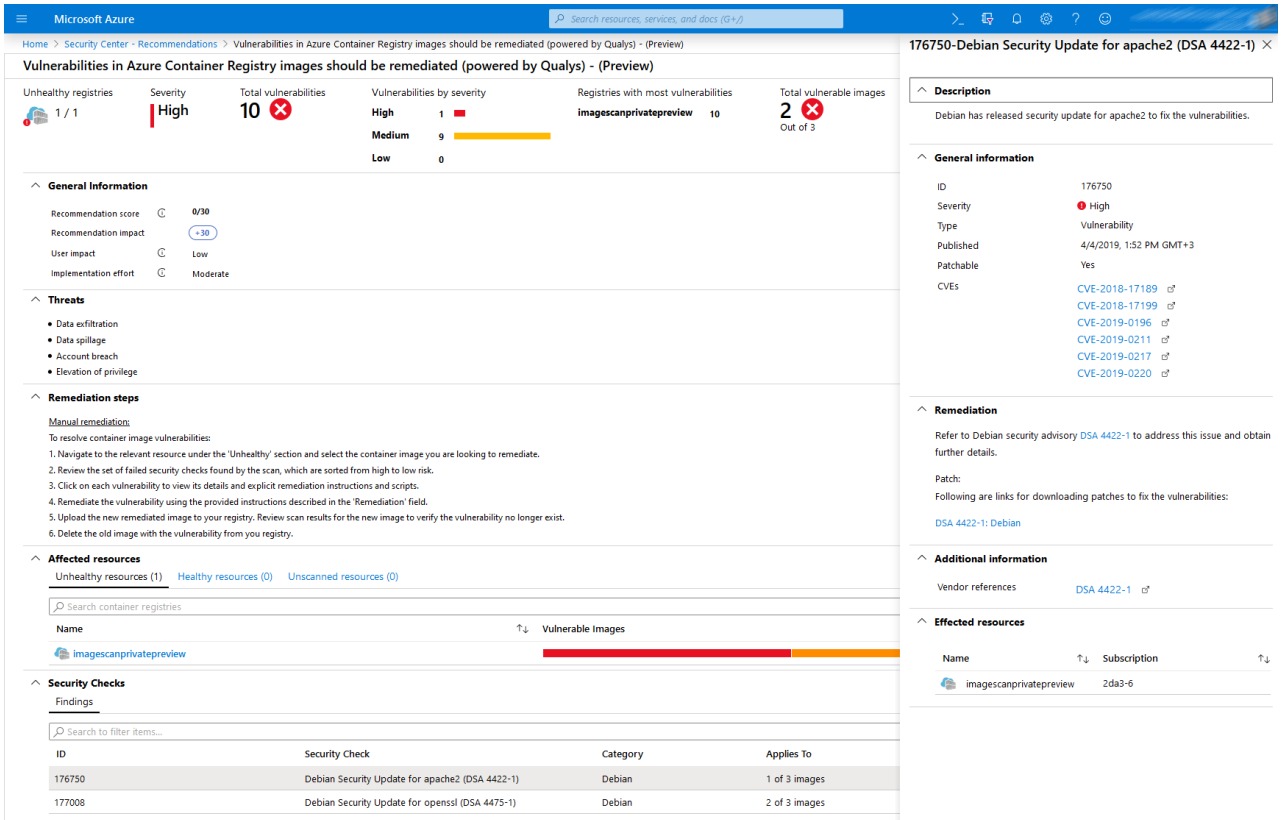
2/18/2020 • 2 minutes to read • Edit Online

Azure Container Registry (ACR) is a managed, private Docker registry service that stores and manages your container images for Azure deployments in a central registry. It's based on the open-source Docker Registry 2.0.

If you're on Azure Security Center's standard tier, you can add the Container Registries bundle. This optional feature brings deeper visibility into the vulnerabilities of the images in your ARM-based registries. Enable or disable the bundle at the subscription level to cover all registries in a subscription. This feature is charged per image, as shown on the pricing page. Enabling the Container Registries bundle, ensures that Security Center is ready to scan images that get pushed to the registry.

Whenever an image is pushed to your registry, Security Center automatically scans that image. To trigger the scan of an image, push it to your repository.

When the scan completes (typically after approximately 10 minutes), findings are available in Security Center recommendations like this:

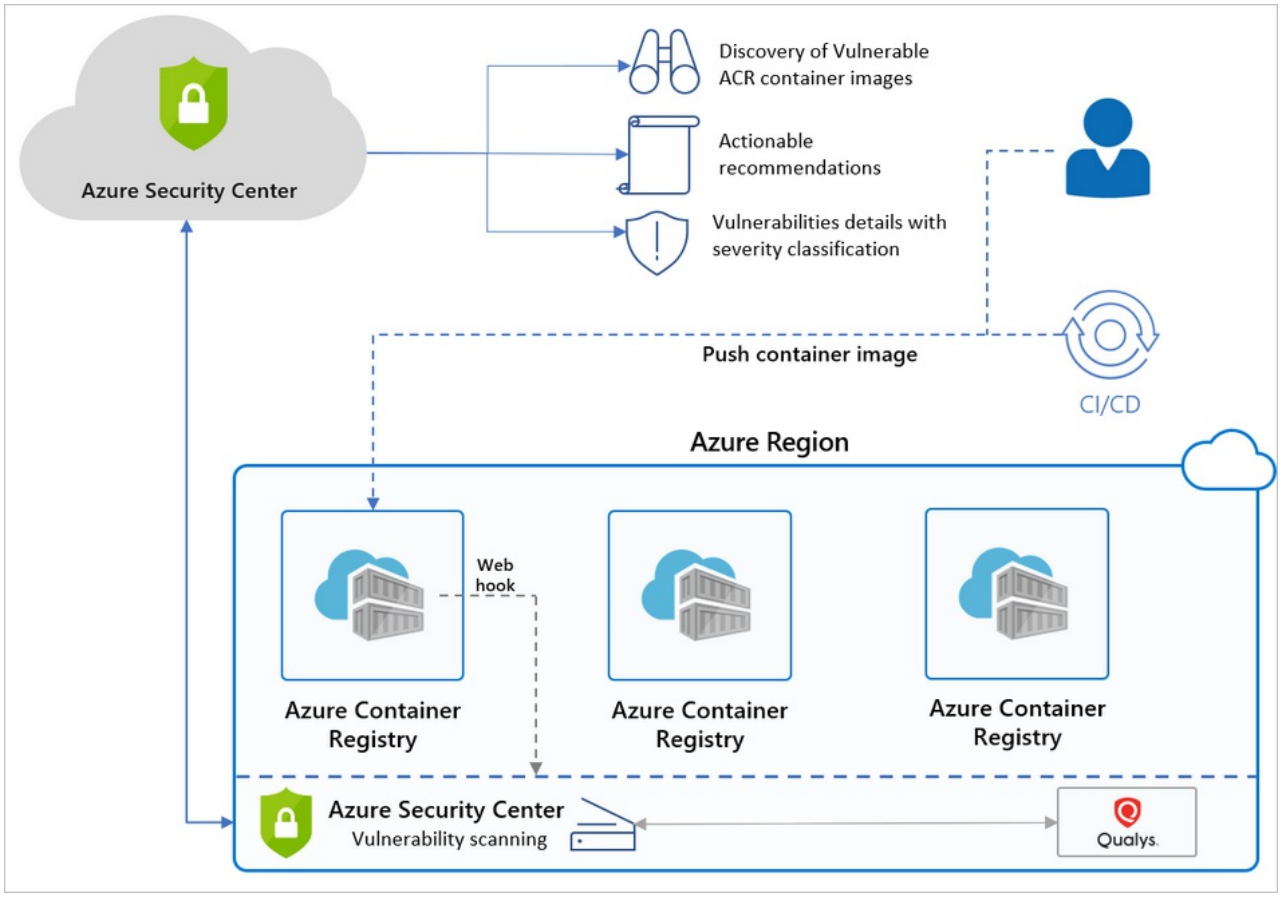


Benefits of integration

Security Center identifies ARM-based ACR registries in your subscription and seamlessly provides:

- **Azure-native vulnerability scanning** for all pushed Linux images. Security Center scans the image using a scanner from the industry-leading vulnerability scanning vendor, Qualys. This native solution is seamlessly integrated by default.
- **Security recommendations** for Linux images with known vulnerabilities. Security Center provides details of each reported vulnerability and a severity classification. Additionally, it gives guidance for how to

remediate the specific vulnerabilities found on each image pushed to registry.



Next steps

To learn more about Security Center's container security features, see:

- Azure Security Center and container security
- Integration with Azure Kubernetes Service
- Virtual Machine protection - Describes Security Center's recommendations

Tutorial: Create a container image for deployment to Azure Container Instances

11/26/2019 • 3 minutes to read • Edit Online

Azure Container Instances enables deployment of Docker containers onto Azure infrastructure without provisioning any virtual machines or adopting a higher-level service. In this tutorial, you package a small Node.js web application into a container image that can be run using Azure Container Instances.

In this article, part one of the series, you:

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

In tutorial parts two and three, you upload your image to Azure Container Registry, and then deploy it to Azure Container Instances.

Before you begin

You must satisfy the following requirements to complete this tutorial:

Azure CLI: You must have Azure CLI version 2.0.29 or later installed on your local computer. Run `az --version` to find the version. If you need to install or upgrade, see Install the Azure CLI.

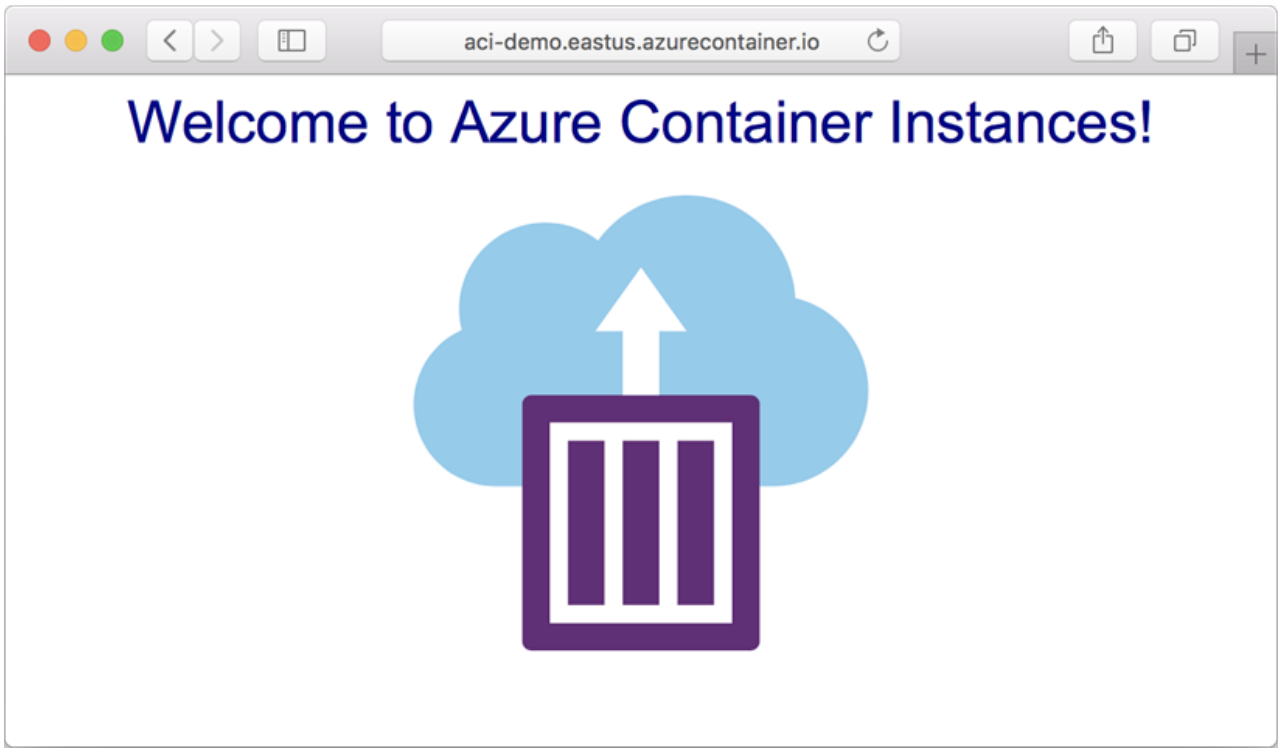
Docker: This tutorial assumes a basic understanding of core Docker concepts like containers, container images, and basic `docker` commands. For a primer on Docker and container basics, see the Docker overview.

Docker: To complete this tutorial, you need Docker installed locally. Docker provides packages that configure the Docker environment on macOS, Windows, and Linux.

> **IMPORTANT**
> Because the Azure Cloud shell does not include the Docker daemon, you *must* install both the Azure CLI and Docker Engine on your *local computer* to complete this tutorial. You cannot use the Azure Cloud Shell for this tutorial.

Get application code

The sample application in this tutorial is a simple web app built in Node.js. The application serves a static HTML page, and looks similar to the following screenshot:



Use Git to clone the sample application's repository:

```
git clone https://github.com/Azure-Samples/aci-helloworld.git
```

You can also download the ZIP archive from GitHub directly.

Build the container image

The Dockerfile in the sample application shows how the container is built. It starts from an official Node.js image based on Alpine Linux, a small distribution that is well suited for use with containers. It then copies the application files into the container, installs dependencies using the Node Package Manager, and finally, starts the application.

```
FROM node:8.9.3-alpine
RUN mkdir -p /usr/src/app
COPY ./app/ /usr/src/app/
WORKDIR /usr/src/app
RUN npm install
CMD node /usr/src/app/index.js
```

Use the docker build command to create the container image and tag it as *aci-tutorial-app*:

```
docker build ./aci-helloworld -t aci-tutorial-app
```

Output from the docker build command is similar to the following (truncated for readability):

```
$ docker build ./aci-helloworld -t aci-tutorial-app
Sending build context to Docker daemon  119.3kB
Step 1/6 : FROM node:8.9.3-alpine
8.9.3-alpine: Pulling from library/node
88286f41530e: Pull complete
84f3a4bf8410: Pull complete
d0d9b2214720: Pull complete
Digest: sha256:c73277ccc763752b42bb2400d1aaecb4e3d32e3a9dbedd0e49885c71bea07354
Status: Downloaded newer image for node:8.9.3-alpine
 ---> 90f5ee24bee2
...
Step 6/6 : CMD node /usr/src/app/index.js
 ---> Running in f4a1ea099eec
 ---> 6edad76d09e9
Removing intermediate container f4a1ea099eec
Successfully built 6edad76d09e9
Successfully tagged aci-tutorial-app:latest
```

Use the docker images command to see the built image:

```
docker images
```

Your newly built image should appear in the list:

```
$ docker images
REPOSITORY          TAG       IMAGE ID        CREATED           SIZE
aci-tutorial-app    latest    5c745774dfa9    39 seconds ago    68.1 MB
```

Run the container locally

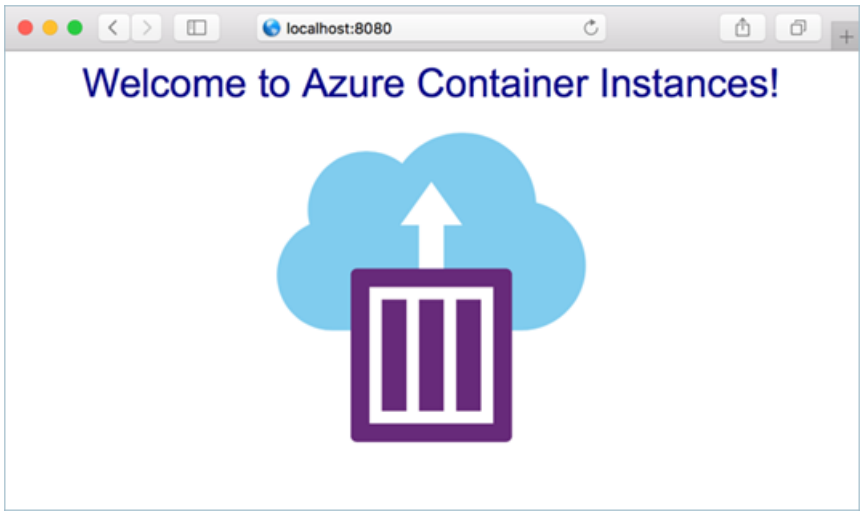
Before you deploy the container to Azure Container Instances, use docker run to run it locally and confirm that it works. The `-d` switch lets the container run in the background, while `-p` allows you to map an arbitrary port on your computer to port 80 in the container.

```
docker run -d -p 8080:80 aci-tutorial-app
```

Output from the `docker run` command displays the running container's ID if the command was successful:

```
$ docker run -d -p 8080:80 aci-tutorial-app
a2e3e4435db58ab0c664ce521854c2e1a1bda88c9cf2fcff46aedf48df86cccf
```

Now, navigate to `http://localhost:8080` in your browser to confirm that the container is running. You should see a web page similar to the following:



Next steps

In this tutorial, you created a container image that can be deployed in Azure Container Instances, and verified that it runs locally. So far, you've done the following:

- Cloned the application source from GitHub
- Created a container image from the application source
- Tested the container locally

Advance to the next tutorial in the series to learn about storing your container image in Azure Container Registry:

Push image to Azure Container Registry

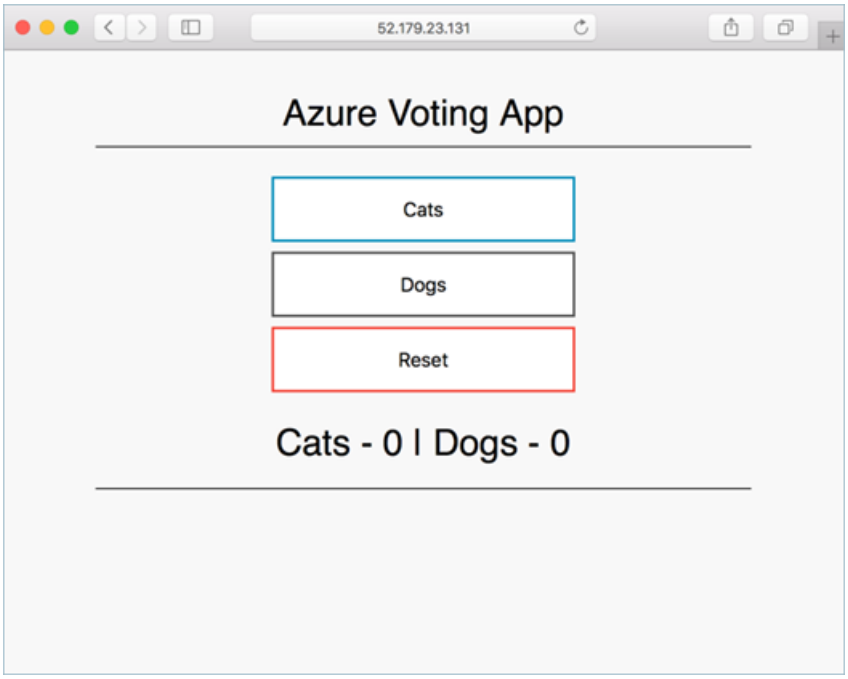
Tutorial: Prepare an application for Azure Kubernetes Service (AKS)

2/25/2020 • 3 minutes to read • Edit Online

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

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

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



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

Before you begin

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

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

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

Get application code

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

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

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

Change into the cloned directory.

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

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

Create container images

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

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

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

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

```
$ docker images

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

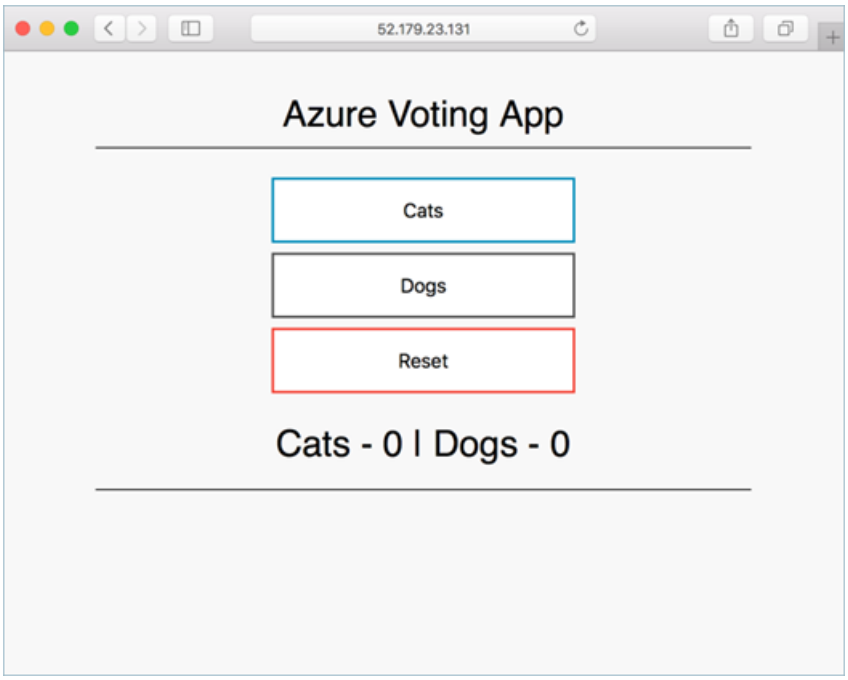
Run the docker ps command to see the running containers:

```
$ docker ps

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

Test application locally

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



Clean up resources

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

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

```
docker-compose down
```

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

Next steps

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

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

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

Push images to Azure Container Registry

Tutorial: Create container images on a Linux Service Fabric cluster

12/19/2019 • 5 minutes to read • Edit Online

This tutorial is part one of a tutorial series that demonstrates how to use containers in a Linux Service Fabric cluster. In this tutorial, a multi-container application is prepared for use with Service Fabric. In subsequent tutorials, these images are used as part of a Service Fabric application. In this tutorial you learn how to:

- Clone application source from GitHub
- Create a container image from the application source
- Deploy an Azure Container Registry (ACR) instance
- Tag a container image for ACR
- Upload the image to ACR

In this tutorial series, you learn how to:

- Create container images for Service Fabric
- Build and Run a Service Fabric Application with Containers
- How failover and scaling are handled in Service Fabric

Prerequisites

- Linux development environment set up for Service Fabric. Follow the instructions here to set up your Linux environment.
- 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 the Azure CLI.
- Additionally, it requires that you have an Azure subscription available. For more information on a free trial version, go here.

Get application code

The sample application used in this tutorial is a voting app. The application consists of a front-end web component and a back-end Redis instance. The components are packaged into container images.

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

```
git clone https://github.com/Azure-Samples/service-fabric-containers.git

cd service-fabric-containers/Linux/container-tutorial/
```

The solution contains two folders and a 'docker-compose.yml' file. The 'azure-vote' folder contains the Python frontend service along with the Dockerfile used to build the image. The 'Voting' directory contains the Service Fabric application package that is deployed to the cluster. These directories contain the necessary assets for this tutorial.

Create container images

Inside the **azure-vote** directory, run the following command to build the image for the front-end web component. This command uses the Dockerfile in this directory to build the image.

```
docker build -t azure-vote-front .
```

> **NOTE**
> If you are getting permission denied then follow this documentation on how to work with docker without sudo.

This command can take some time since all the necessary dependencies need to be pulled from Docker Hub. When completed, use the docker images command to see the created images.

```
docker images
```

Notice that two images have been downloaded or created. The *azure-vote-front* image contains the application. It was derived from a *python* image from Docker Hub.

```
REPOSITORY                  TAG                 IMAGE ID            CREATED              SIZE
azure-vote-front            latest              052c549a75bf        About a minute ago   708MB
tiangolo/uwsgi-nginx-flask  python3.6           590e17342131        5 days ago           707MB
```

Deploy Azure Container Registry

First run the **az login** command to sign in to your Azure account.

```
az login
```

Next, use the **az account** command to choose your subscription to create the Azure Container registry. You have to enter the subscription ID of your Azure subscription in place of <subscription_id>.

```
az account set --subscription <subscription_id>
```

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 *westus* region.

```
az group create --name <myResourceGroup> --location westus
```

Create an Azure Container registry with the **az acr create** command. Replace <acrName> with the name of the container registry you want to create under your subscription. This name must be alphanumeric and unique.

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

Throughout the rest of this tutorial, we use "acrName" as a placeholder for the container registry name that you chose. Please make note of this value.

Sign in to your container registry

Sign in to your ACR instance before pushing images to it. Use the **az acr login** command to complete the

operation. Provide the unique name given to the container registry when it was created.

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

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

Tag container images

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              052c549a75bf        About a minute ago   708MB
tiangolo/uwsgi-nginx-flask    python3.6           590e17342131        5 days ago           707MB
```

To get the loginServer name, run the following command:

```
az acr show --name <acrName> --query loginServer --output table
```

This outputs a table with the following results. This result will be used to tag your **azure-vote-front** image before pushing it to the container registry in the next step.

```
Result
------------------
<acrName>.azurecr.io
```

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

```
docker tag azure-vote-front <acrName>.azurecr.io/azure-vote-front:v1
```

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

Output:

```
REPOSITORY                            TAG                 IMAGE ID            CREATED              SIZE
azure-vote-front                      latest              052c549a75bf        23 minutes ago       708MB
<acrName>.azurecr.io/azure-vote-front  v1                  052c549a75bf        23 minutes ago       708MB
tiangolo/uwsgi-nginx-flask            python3.6           590e17342131        5 days ago           707MB
```

Push images to registry

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

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

```
docker push <acrName>.azurecr.io/azure-vote-front:v1
```

The docker push commands take a couple of minutes to complete.

List images in registry

To return a list of images that have been pushed to your Azure Container registry, use the az acr repository list command. Update the command with the ACR instance name.

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

Output:

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

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

Next steps

In this tutorial, an application was pulled from GitHub and container images were created and pushed to a registry. The following steps were completed:

- Clone application source from GitHub
- Create a container image from the application source
- Deploy an Azure Container Registry (ACR) instance
- Tag a container image for ACR
- Upload the image to ACR

Advance to the next tutorial to learn about packaging containers into a Service Fabric application using Yeoman.

Package and deploy containers as a Service Fabric application

Azure Container Registry logs for diagnostic evaluation and auditing

1/8/2020 • 3 minutes to read • Edit Online

This article explains how to collect log data for an Azure container registry using features of Azure Monitor. Azure Monitor collects resource logs (formerly called *diagnostic logs*) for user-driven events in your registry. Collect and consume this data to meet needs such as:

- Audit registry authentication events to ensure security and compliance
- Provide a complete activity trail on registry artifacts such as pull and pull events so you can diagnose operational issues with your registry

Collecting resource log data using Azure Monitor may incur additional costs. See Azure Monitor pricing.

> **IMPORTANT**
> This feature is currently in preview, and some limitations apply. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

Preview limitations

The following repository-level events for images and other artifacts are currently logged:

- **Push events**
- **Pull events**
- **Untag events**
- **Delete events** (including repository delete events)

Repository-level events that aren't currently logged: Purge events.

Registry resource logs

Resource logs contain information emitted by Azure resources that describe their internal operation. For an Azure container registry, the logs contain authentication and repository-level events stored in the following tables.

- **ContainerRegistryLoginEvents** - Registry authentication events and status, including the incoming identity and IP address
- **ContainerRegistryRepositoryEvents** - Operations such as push and pull for images and other artifacts in registry repositories
- **AzureMetrics** - Container registry metrics such as aggregated push and pull counts.

For operations, log data includes:

- Success or failure status
- Start and end time stamps

In addition to resource logs, Azure provides an activity log, a single subscription-level record of Azure management events such as the creation or deletion of a container registry.

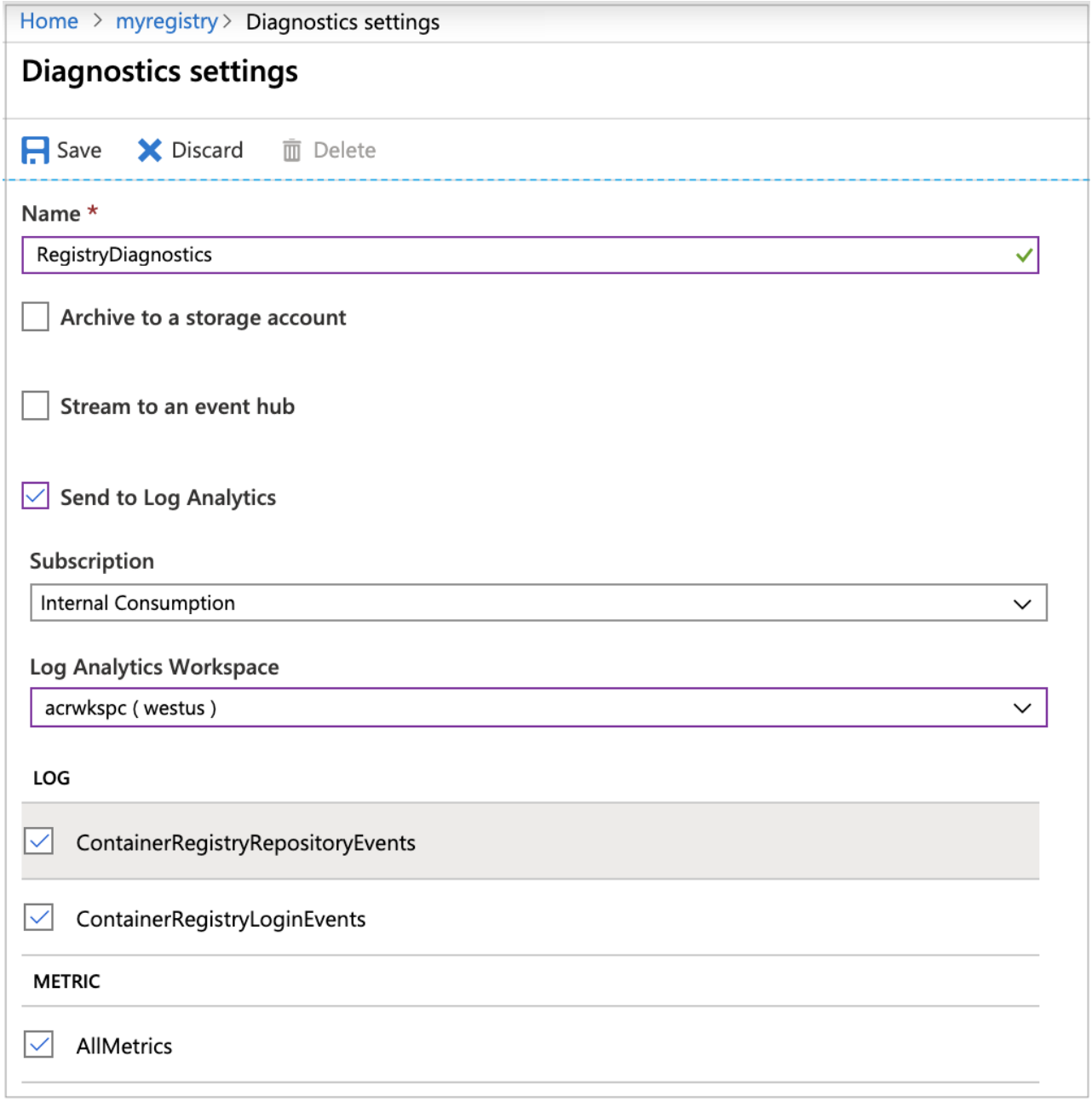
Enable collection of resource logs

Collection of resource logs for a container registry isn't enabled by default. Explicitly enable diagnostic settings for each registry you want to monitor. For options to enable diagnostic settings, see Create diagnostic setting to collect platform logs and metrics in Azure.

For example, to view logs and metrics for a container registry in near real-time in Azure Monitor, collect the resource logs in a Log Analytics workspace. To enable this diagnostic setting using the Azure portal:

1. If you don't already have a workspace, create a workspace using the Azure portal. To minimize latency in data collection, ensure that the workspace is in the **same region** as your container registry.
2. In the portal, select the registry, and select **Monitoring > Diagnostic settings > Add diagnostic setting**.
3. Enter a name for the setting, and select **Send to Log Analytics**.
4. Select the workspace for the registry diagnostic logs.
5. Select the log data you want to collect, and click **Save**.

The following image shows creation of a diagnostic setting for a registry using the portal.



> **TIP**
> Collect only the data that you need, balancing cost and your monitoring needs. For example, if you only need to audit authentication events, select only the **ContainerRegistryLoginEvents** log.

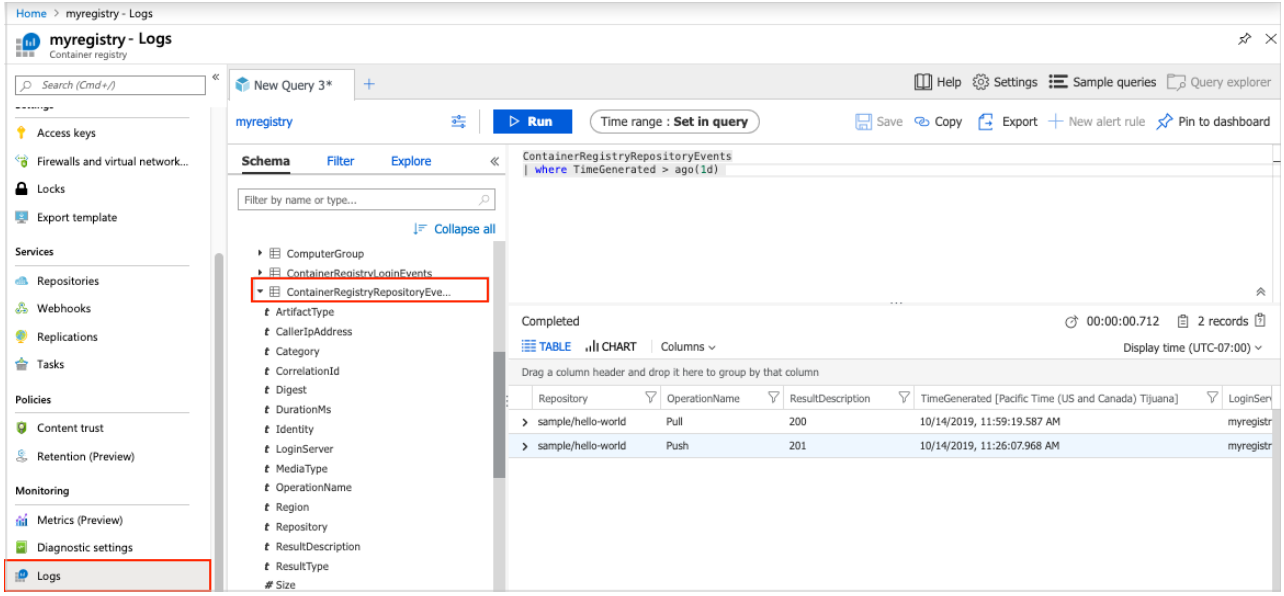
View data in Azure Monitor

After you enable collection of diagnostic logs in Log Analytics, it can take a few minutes for data to appear in Azure Monitor. To view the data in the portal, select the registry, and select **Monitoring > Logs**. Select one of the tables that contains data for the registry.

Run queries to view the data. Several sample queries are provided, or run your own. For example, the following query retrieves the most recent 24 hours of data from the **ContainerRegistryRepositoryEvents** table:

```
ContainerRegistryRepositoryEvents
| where TimeGenerated > ago(1d)
```

The following image shows sample output:



For a tutorial on using Log Analytics in the Azure portal, see Get started with Azure Monitor Log Analytics, or try the Log Analytics Demo environment.

For more information on log queries, see Overview of log queries in Azure Monitor.

Additional query examples

100 most recent registry events

```
ContainerRegistryRepositoryEvents
| union ContainerRegistryLoginEvents
| top 100 by TimeGenerated
| project TimeGenerated, LoginServer , OperationName , Identity , Repository , DurationMs , Region , ResultType
```

Additional log destinations

In addition to sending the logs to Log Analytics, or as an alternative, a common scenario is to select an Azure Storage account as a log destination. To archive logs in Azure Storage, create a storage account before enabling archiving through the diagnostic settings.

You can also stream diagnostic log events to an Azure Event Hub. Event Hubs can ingest millions of events per second, which you can then transform and store using any real-time analytics provider.

Next steps

- Learn more about using Log Analytics and creating log queries.
- See Overview of Azure platform logs to learn about platform logs that are available at different layers of Azure.

Audit compliance of Azure container registries using Azure Policy

2/28/2020 • 2 minutes to read • Edit Online

Azure Policy is a service in Azure that you use to create, assign, and manage policies. These policies enforce different rules and effects over your resources, so those resources stay compliant with your corporate standards and service level agreements.

This article introduces built-in policies (preview) for Azure Container Registry. Use these policies to audit new and existing registries for compliance.

There are no charges for using Azure Policy.

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

Built-in policy definitions

The following built-in policy definitions are specific to Azure Container Registry:

NAME	DESCRIPTION	EFFECT(S)	VERSION	SOURCE
Container Registries should be encrypted with a Customer-Managed Key (CMK)	Audit Container Registries that do not have encryption enabled with Customer-Managed Keys (CMK). For more information on CMK encryption, please visit: https://aka.ms/acr/CMK.	Audit, Disabled	1.0.0-preview	GitHub

NAME	DESCRIPTION	EFFECT(S)	VERSION	SOURCE
Container Registries should not allow unrestricted network access	Audit Container Registries that do not have any Network (IP or VNET) Rules configured and allow all network access by default. Container Registries with at least one IP / Firewall rule or configured virtual network will be deemed compliant. For more information on Container Registry Network rules, please visit: https://aka.ms/acr/vnet.	Audit, Disabled	1.0.0-preview	GitHub

See also the built-in network policy definition: [Preview] Container Registry should use a virtual network service endpoint.

Assign policies

- Assign policies using the Azure portal, Azure CLI, a Resource Manager template, or the Azure Policy SDKs.
- Scope a policy assignment to a resource group, a subscription, or an Azure management group. Container registry policy assignments apply to existing and new container registries within the scope.
- Enable or disable policy enforcement at any time.

> **NOTE**
> After you assign or update a policy, it takes some time for the assignment to be applied to resources in the defined scope. See information about policy evaluation triggers.

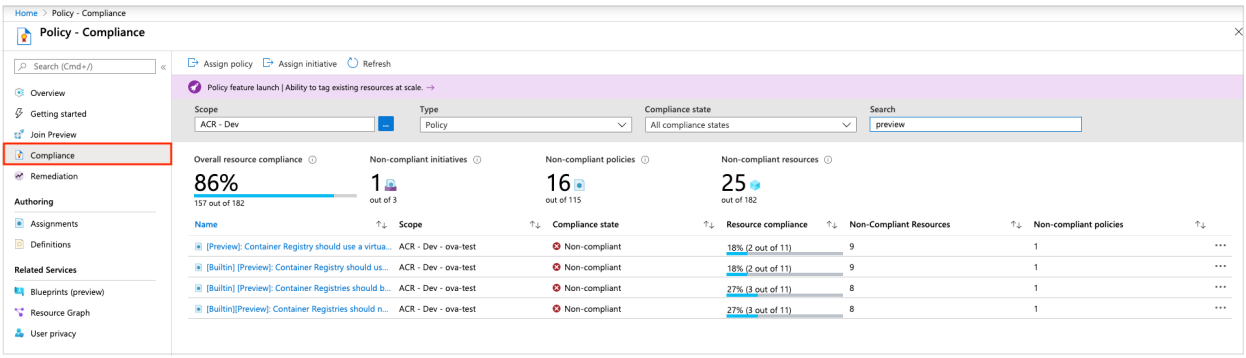
Review policy compliance

Access compliance information generated by your policy assignments using the Azure portal, Azure command-line tools, or the Azure Policy SDKs. For details, see Get compliance data of Azure resources.

When a resource is non-compliant, there are many possible reasons. To determine the reason or to find the change responsible, see Determine non-compliance.

Policy compliance in the portal:

1. Select **All services**, and search for **Policy**.
2. Select **Compliance**.
3. Use the filters to limit compliance states or to search for policies



.

4. Select a policy to review aggregate compliance details and events. If desired, then select a specific registry for resource compliance.

Policy compliance in the Azure CLI

You can also use the Azure CLI to get compliance data. For example, use the az policy assignment list command in the CLI to get the policy IDs of the Azure Container Registry policies that are applied:

```
az policy assignment list --query "[?contains(displayName,'Container Registries')].{name:displayName, ID:id}" --output table
```

Sample output:

```
Name                                                                                ID
----------------------------------------------------------------------------------  ------------------------
-------------------------------------------------------------------------------------------------------------
[Preview]: Container Registries should not allow unrestricted network access
/subscriptions/<subscriptionID>/providers/Microsoft.Authorization/policyAssignments/b4faf132dc344b84ba68a441
[Preview]: Container Registries should be encrypted with a Customer-Managed Key (CMK)
/subscriptions/<subscriptionID>/providers/Microsoft.Authorization/policyAssignments/cce1ed4f38a147ad994ab60a
```

Then run az policy state list to return the JSON-formatted compliance state for all resources under a specific policy ID:

```
az policy state list \
  --resource <policyID>
```

Or run az policy state list to return the JSON-formatted compliance state of a specific registry resource, such as *myregistry*:

```
az policy state list \
 --resource myregistry \
 --namespace Microsoft.ContainerRegistry \
 --resource-type registries \
 --resource-group myresourcegroup
```

Next steps

- Learn more about Azure Policy definitions and effects
- Create a custom policy definition
- Learn more about governance capabilities in Azure

Check the health of an Azure container registry

11/24/2019 • 2 minutes to read • Edit Online

When using an Azure container registry, you might occasionally encounter problems. For example, you might not be able to pull a container image because of an issue with Docker in your local environment. Or, a network issue might prevent you from connecting to the registry.

As a first diagnostic step, run the az acr check-health command to get information about the health of the environment and optionally access to a target registry. This command is available in Azure CLI version 2.0.67 or later. If you need to install or upgrade, see Install Azure CLI.

Run az acr check-health

The follow examples show different ways to run the `az acr check-health` command.

> **NOTE**
> If you run the command in Azure Cloud Shell, the local environment is not checked. However, you can check the access to a target registry.

Check the environment only

To check the local Docker daemon, CLI version, and Helm client configuration, run the command without additional parameters:

```
az acr check-health
```

Check the environment and a target registry

To check access to a registry as well as perform local environment checks, pass the name of a target registry. For example:

```
az acr check-health --name myregistry
```

Error reporting

The command logs information to the standard output. If a problem is detected, it provides an error code and description. For more information about the codes and possible solutions, see the error reference.

By default, the command stops whenever it finds an error. You can also run the command so that it provides output for all health checks, even if errors are found. Add the `--ignore-errors` parameter, as shown in the following examples:

```
# Check environment only
az acr check-health --ignore-errors

# Check environment and target registry
az acr check-health --name myregistry --ignore-errors
```

Sample output:

```
$ az acr check-health --name myregistry --ignore-errors --yes

Docker daemon status: available
Docker version: Docker version 18.09.2, build 6247962
Docker pull of 'mcr.microsoft.com/mcr/hello-world:latest' : OK
ACR CLI version: 2.2.9
Helm version:
Client: &version.Version{SemVer:"v2.14.1", GitCommit:"5270352a09c7e8b6e8c9593002a73535276507c0",
GitTreeState:"clean"}
DNS lookup to myregistry.azurecr.io at IP 40.xxx.xxx.162 : OK
Challenge endpoint https://myregistry.azurecr.io/v2/ : OK
Fetch refresh token for registry 'myregistry.azurecr.io' : OK
Fetch access token for registry 'myregistry.azurecr.io' : OK
```

Next steps

For details about error codes returned by the az acr check-health command, see the Health check error reference.

See the FAQ for frequently asked questions and other known issues about Azure Container Registry.

Frequently asked questions about Azure Container Registry

1/7/2020 • 14 minutes to read • Edit Online

This article addresses frequently asked questions and known issues about Azure Container Registry.

Resource management

Can I create an Azure Container Registry using a Resource Manager template?

Yes. Here is a template that you can use to create a registry.

Is there security vulnerability scanning for images in ACR?

Yes. See the documentation from Azure Security Center, Twistlock and Aqua.

How do I configure Kubernetes with Azure Container Registry?

See the documentation for Kubernetes and steps for Azure Kubernetes Service.

How do I get admin credentials for a container registry?

> **IMPORTANT**
> The admin user account is designed for a single user to access the registry, mainly for testing purposes. We do not recommend sharing the admin account credentials with multiple users. Individual identity is recommended for users and service principals for headless scenarios. See Authentication overview.

Before getting admin credentials, make sure the registry's admin user is enabled.

To get credentials using the Azure CLI:

```
az acr credential show -n myRegistry
```

Using Azure Powershell:

```
Invoke-AzureRmResourceAction -Action listCredentials -ResourceType Microsoft.ContainerRegistry/registries -
ResourceGroupName myResourceGroup -ResourceName myRegistry
```

How do I get admin credentials in a Resource Manager template?

> **IMPORTANT**
> The admin user account is designed for a single user to access the registry, mainly for testing purposes. We do not recommend sharing the admin account credentials with multiple users. Individual identity is recommended for users and service principals for headless scenarios. See Authentication overview.

Before getting admin credentials, make sure the registry's admin user is enabled.

To get the first password:

```
{
    "password": "[listCredentials(resourceId('Microsoft.ContainerRegistry/registries', 'myRegistry'), '2017-
10-01').passwords[0].value]"
}
```

To get the second password:

```
{
    "password": "[listCredentials(resourceId('Microsoft.ContainerRegistry/registries', 'myRegistry'), '2017-
10-01').passwords[1].value]"
}
```

Delete of replication fails with Forbidden status although the replication gets deleted using the Azure CLI or Azure PowerShell

The error is seen when the user has permissions on a registry but doesn't have Reader-level permissions on the subscription. To resolve this issue, assign Reader permissions on the subscription to the user:

```
az role assignment create --role "Reader" --assignee user@contoso.com --scope /subscriptions/<subscription_id>
```

Firewall rules are updated successfully but they do not take effect

It takes some time to propagate firewall rule changes. After you change firewall settings, please wait for a few minutes before verifying this change.

Registry operations

- How do I access Docker Registry HTTP API V2?
- How do I delete all manifests that are not referenced by any tag in a repository?
- Why does the registry quota usage not reduce after deleting images?
- How do I validate storage quota changes?
- How do I authenticate with my registry when running the CLI in a container?
- How to enable TLS 1.2?
- Does Azure Container Registry support Content Trust?
- How do I grant access to pull or push images without permission to manage the registry resource?
- How do I enable automatic image quarantine for a registry

How do I access Docker Registry HTTP API V2?

ACR supports Docker Registry HTTP API V2. The APIs can be accessed at `https://<your registry login server>/v2/`. Example: `https://mycontainerregistry.azurecr.io/v2/`

How do I delete all manifests that are not referenced by any tag in a repository?

If you are on bash:

```
az acr repository show-manifests -n myRegistry --repository myRepository --query "[?tags[0]==null].digest" -o
tsv  | xargs -I% az acr repository delete -n myRegistry -t myRepository@%
```

For Powershell:

```
az acr repository show-manifests -n myRegistry --repository myRepository --query "[?tags[0]==null].digest" -o
tsv | %{ az acr repository delete -n myRegistry -t myRepository@$_ }
```

Note: You can add `-y` in the delete command to skip confirmation.

For more information, see Delete container images in Azure Container Registry.

Why does the registry quota usage not reduce after deleting images?

This situation can happen if the underlying layers are still being referenced by other container images. If you delete an image with no references, the registry usage updates in a few minutes.

How do I validate storage quota changes?

Create an image with a 1GB layer using the following docker file. This ensures that the image has a layer that is not shared by any other image in the registry.

```
FROM alpine
RUN dd if=/dev/urandom of=1GB.bin  bs=32M  count=32
RUN ls -lh 1GB.bin
```

Build and push the image to your registry using the docker CLI.

```
docker build -t myregistry.azurecr.io/1gb:latest .
docker push myregistry.azurecr.io/1gb:latest
```

You should be able to see that the storage usage has increased in the Azure portal, or you can query usage using the CLI.

```
az acr show-usage -n myregistry
```

Delete the image using the Azure CLI or portal and check the updated usage in a few minutes.

```
az acr repository delete -n myregistry --image 1gb
```

How do I authenticate with my registry when running the CLI in a container?

You need to run the Azure CLI container by mounting the Docker socket:

```
docker run -it -v /var/run/docker.sock:/var/run/docker.sock azuresdk/azure-cli-python:dev
```

In the container, install `docker`:

```
apk --update add docker
```

Then authenticate with your registry:

```
az acr login -n MyRegistry
```

How to enable TLS 1.2?

Enable TLS 1.2 by using any recent docker client (version 18.03.0 and above).

> **IMPORTANT**
> Starting January 13, 2020, Azure Container Registry will require all secure connections from servers and applications to use TLS 1.2. Support for TLS 1.0 and 1.1 will be retired.

Does Azure Container Registry support Content Trust?

Yes, you can use trusted images in Azure Container Registry, since the Docker Notary has been integrated and can be enabled. For details, see Content Trust in Azure Container Registry.

Where is the file for the thumbprint located?

Under `~/.docker/trust/tuf/myregistry.azurecr.io/myrepository/metadata`:

- Public keys and certificates of all roles (except delegation roles) are stored in the `root.json`.
- Public keys and certificates of the delegation role are stored in the JSON file of its parent role (for example `targets.json` for the `targets/releases` role).

It is suggested to verify those public keys and certificates after the overall TUF verification done by the Docker and Notary client.

How do I grant access to pull or push images without permission to manage the registry resource?

ACR supports custom roles that provide different levels of permissions. Specifically, `AcrPull` and `AcrPush` roles allow users to pull and/or push images without the permission to manage the registry resource in Azure.

- Azure portal: Your registry -> Access Control (IAM) -> Add (Select `AcrPull` or `AcrPush` for the Role).
- Azure CLI: Find the resource ID of the registry by running the following command:

  ```
  az acr show -n myRegistry
  ```

  Then you can assign the `AcrPull` or `AcrPush` role to a user (the following example uses `AcrPull`):

  ```
  az role assignment create --scope resource_id --role AcrPull --assignee user@example.com
  ```

  Or, assign the role to a service principle identified by its application ID:

  ```
  az role assignment create --scope resource_id --role AcrPull --assignee 00000000-0000-0000-0000-000000000000
  ```

The assignee is then able to authenticate and access images in the registry.

- To authenticate to a registry:

  ```
  az acr login -n myRegistry
  ```

- To list repositories:

```
az acr repository list -n myRegistry
```

To pull an image:

```
docker pull myregistry.azurecr.io/hello-world
```

With the use of only the `AcrPull` or `AcrPush` role, the assignee doesn't have the permission to manage the registry resource in Azure. For example, `az acr list` or `az acr show -n myRegistry` won't show the registry.

How do I enable automatic image quarantine for a registry?

Image quarantine is currently a preview feature of ACR. You can enable the quarantine mode of a registry so that only those images which have successfully passed security scan are visible to normal users. For details, see the ACR GitHub repo.

Diagnostics and health checks

- Check health with `az acr check-health`
- docker pull fails with error: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)
- docker push succeeds but docker pull fails with error: unauthorized: authentication required
- `az acr login` succeeds, but docker commands fails with error: unauthorized: authentication required
- Enable and get the debug logs of the docker daemon
- New user permissions may not be effective immediately after updating
- Authentication information is not given in the correct format on direct REST API calls
- Why does the Azure portal not list all my repositories or tags?
- Why does the Azure portal fail to fetch repositories or tags?
- Why does my pull or push request fail with disallowed operation?
- How do I collect http traces on Windows?

Check health with `az acr check-health`

To troubleshoot common environment and registry issues, see Check the health of an Azure container registry.

docker pull fails with error: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)

- If this error is a transient issue, then retry will succeed.
- If `docker pull` fails continuously, then there could be a problem with the Docker daemon. The problem can generally be mitigated by restarting the Docker daemon.
- If you continue to see this issue after restarting Docker daemon, then the problem could be some network connectivity issues with the machine. To check if general network on the machine is healthy, run the following command to test endpoint connectivity. The minimum `az acr` version that contains this connectivity check command is 2.2.9. Upgrade your Azure CLI if you are using an older version.

  ```
  az acr check-health -n myRegistry
  ```

- You should always have a retry mechanism on all Docker client operations.

Docker pull is slow

Use this tool to test your machine network download speed. If machine network is slow, consider using Azure VM in the same region as your registry. This usually gives you faster network speed.

Docker push is slow

Use this tool to test your machine network upload speed. If machine network is slow, consider using Azure VM in the same region as your registry. This usually gives you faster network speed.

Docker push succeeds but docker pull fails with error: unauthorized: authentication required

This error can happen with the Red Hat version of the Docker daemon, where `--signature-verification` is enabled by default. You can check the Docker daemon options for Red Hat Enterprise Linux (RHEL) or Fedora by running the following command:

```
grep OPTIONS /etc/sysconfig/docker
```

For instance, Fedora 28 Server has the following docker daemon options:

```
OPTIONS='--selinux-enabled --log-driver=journald --live-restore'
```

With `--signature-verification=false` missing, `docker pull` fails with an error similar to:

```
Trying to pull repository myregistry.azurecr.io/myimage ...
unauthorized: authentication required
```

To resolve the error:

1. Add the option `--signature-verification=false` to the Docker daemon configuration file `/etc/sysconfig/docker`. For example:

```
OPTIONS='--selinux-enabled --log-driver=journald --live-restore --signature-verification=false'
```

2. Restart the Docker daemon service by running the following command:

```
sudo systemctl restart docker.service
```

Details of `--signature-verification` can be found by running `man dockerd`.

az acr login succeeds but docker fails with error: unauthorized: authentication required

Make sure you use an all lowercase server URL, for example, `docker push myregistry.azurecr.io/myimage:latest`, even if the registry resource name is uppercase or mixed case, like `myRegistry`.

Enable and get the debug logs of the Docker daemon

Start `dockerd` with the `debug` option. First, create the Docker daemon configuration file (`/etc/docker/daemon.json`) if it doesn't exist, and add the `debug` option:

```
{
    "debug": true
}
```

Then, restart the daemon. For example, with Ubuntu 14.04:

```
sudo service docker restart
```

Details can be found in the Docker documentation.

- The logs may be generated at different locations, depending on your system. For example, for Ubuntu 14.04, it's `/var/log/upstart/docker.log`. See Docker documentation for details.
- For Docker for Windows, the logs are generated under %LOCALAPPDATA%/docker/. However it may not contain all the debug information yet.

  In order to access the full daemon log, you may need some extra steps:

  ```
  docker run --privileged -it --rm -v /var/run/docker.sock:/var/run/docker.sock -v
  /usr/local/bin/docker:/usr/local/bin/docker alpine sh

  docker run --net=host --ipc=host --uts=host --pid=host -it --security-opt=seccomp=unconfined --
  privileged --rm -v /:/host alpine /bin/sh
  chroot /host
  ```

  Now you have access to all the files of the VM running `dockerd`. The log is at `/var/log/docker.log`.

New user permissions may not be effective immediately after updating

When you grant new permissions (new roles) to a service principal, the change might not take effect immediately. There are two possible reasons:

- Azure Active Directory role assignment delay. Normally it's fast, but it could take minutes due to propagation delay.
- Permission delay on ACR token server. This could take up to 10 minutes. To mitigate, you can `docker logout` and then authenticate again with the same user after 1 minute:

  ```
  docker logout myregistry.azurecr.io
  docker login myregistry.azurecr.io
  ```

Currently ACR doesn't support home replication deletion by the users. The workaround is to include the home replication create in the template but skip its creation by adding `"condition": false` as shown below:

```
{
    "name": "[concat(parameters('acrName'), '/', parameters('location'))]",
    "condition": false,
    "type": "Microsoft.ContainerRegistry/registries/replications",
    "apiVersion": "2017-10-01",
    "location": "[parameters('location')]",
    "properties": {},
    "dependsOn": [
        "[concat('Microsoft.ContainerRegistry/registries/', parameters('acrName'))]"
     ]
},
```

Authentication information is not given in the correct format on direct REST API calls

You may encounter an `InvalidAuthenticationInfo` error, especially using the `curl` tool with the option `-L`, `--location` (to follow redirects). For example, fetching the blob using `curl` with `-L` option and basic authentication:

```
curl -L -H "Authorization: basic $credential" https://$registry.azurecr.io/v2/$repository/blobs/$digest
```

may result in the following response:

```
<?xml version="1.0" encoding="utf-8"?>
<Error><Code>InvalidAuthenticationInfo</Code><Message>Authentication information is not given in the correct
format. Check the value of Authorization header.
RequestId:00000000-0000-0000-0000-000000000000
Time:2019-01-01T00:00:00.0000000Z</Message></Error>
```

The root cause is that some `curl` implementations follow redirects with headers from the original request.

To resolve the problem, you need to follow redirects manually without the headers. Print the response headers with the `-D -` option of `curl` and then extract: the `Location` header:

```
redirect_url=$(curl -s -D - -H "Authorization: basic $credential"
https://$registry.azurecr.io/v2/$repository/blobs/$digest | grep "^Location: " | cut -d " " -f2 | tr -d '\r')
curl $redirect_url
```

Why does the Azure portal not list all my repositories or tags?

If you are using the Microsoft Edge/IE browser, you can see at most 100 repositories or tags. If your registry has more than 100 repositories or tags, we recommend that you use either the Firefox or Chrome browser to list them all.

Why does the Azure portal fail to fetch repositories or tags?

The browser might not be able to send the request for fetching repositories or tags to the server. There could be various reasons such as:

- Lack of network connectivity
- Firewall
- Ad blockers
- DNS errors

Please contact your network administrator or check your network configuration and connectivity. Try running `az acr check-health -n yourRegistry` using your Azure CLI to check if your environment is able to connect to the Container Registry. In addition, you could also try an incognito or private session in your browser to avoid any stale browser cache or cookies.

Why does my pull or push request fail with disallowed operation?

Here are some scenarios where operations maybe disallowed:

- Classic registries are no longer supported. Please upgrade to a supported SKUs using az acr update or the Azure portal.
- The image or repository maybe locked so that it can't be deleted or updated. You can use the az acr show repository command to view current attributes.
- Some operations are disallowed if the image is in quarantine. Learn more about quarantine.

How do I collect http traces on Windows?

Prerequisites

- Enable decrypting https in fiddler: https://docs.telerik.com/fiddler/Configure-Fiddler/Tasks/DecryptHTTPS
- Enable Docker to use a proxy through the Docker ui: https://docs.docker.com/docker-for-windows/#proxies
- Be sure to revert when complete. Docker won't work with this enabled and fiddler not running.

Windows containers

Configure Docker proxy to 127.0.0.1:8888

Linux containers

Find the ip of the Docker vm virtual switch:

```
(Get-NetIPAddress -InterfaceAlias "*Docker*" -AddressFamily IPv4).IPAddress
```

Configure the Docker proxy to output of the previous command and the port 8888 (for example 10.0.75.1:8888)

Tasks

- How do I batch cancel runs?
- How do I include the .git folder in az acr build command?
- Does Tasks support GitLab for Source triggers?
- What git repository management service does Tasks support?

How do I batch cancel runs?

The following commands cancel all running tasks in the specified registry.

```
az acr task list-runs -r $myregistry --run-status Running --query '[].runId' -o tsv \
| xargs -I% az acr task cancel-run -r $myregistry --run-id %
```

How do I include the .git folder in az acr build command?

If you pass a local source folder to the `az acr build` command, the `.git` folder is excluded from the uploaded package by default. You can create a `.dockerignore` file with the following setting. It tells the command to restore all files under `.git` in the uploaded package.

```
!.git/**
```

This setting also applies to the `az acr run` command.

Does Tasks support GitLab for Source triggers?

We currently do not support GitLab for Source triggers.

What git repository management service does Tasks support?

GIT SERVICE	SOURCE CONTEXT	MANUAL BUILD	AUTO BUILD THROUGH COMMIT TRIGGER
GitHub	https://github.com/user/myapp-repo.git#mybranch:myfolder	Yes	Yes
Azure Repos	https://dev.azure.com/user/myproject/_git/myapp-repo#mybranch:myfolder	Yes	Yes
GitLab	https://gitlab.com/user/myapp-repo.git#mybranch:myfolder	Yes	No
BitBucket	https://user@bitbucket.org/user/mayapp-repo.git#mybranch:myfolder	Yes	No

Run Error Message Troubleshooting

ERROR MESSAGE	TROUBLESHOOTING GUIDE
No access was configured for the VM, hence no subscriptions were found	This could happen if you are using `az login --identity` in your ACR Task. This is a transient error and occurs when the role assignment of your Managed Identity hasn't propagated. Waiting a few seconds before retrying works.

CI/CD integration

- CircleCI
- GitHub Actions

Next steps

- Learn more about Azure Container Registry.

Automate container image builds and maintenance with ACR Tasks

2/25/2020 • 8 minutes to read • Edit Online

Containers provide new levels of virtualization, isolating application and developer dependencies from infrastructure and operational requirements. What remains, however, is the need to address how this application virtualization is managed and patched over the container lifecycle.

What is ACR Tasks?

ACR Tasks is a suite of features within Azure Container Registry. It provides cloud-based container image building for platforms including Linux, Windows, and ARM, and can automate OS and framework patching for your Docker containers. ACR Tasks not only extends your "inner-loop" development cycle to the cloud with on-demand container image builds, but also enables automated builds triggered by source code updates, updates to a container's base image, or timers. For example, with base image update triggers, you can automate your OS and application framework patching workflow, maintaining secure environments while adhering to the principles of immutable containers.

Task scenarios

ACR Tasks supports several scenarios to build and maintain container images and other artifacts. See the following sections in this article for details.

- **Quick task** - Build and push a single container image to a container registry on-demand, in Azure, without needing a local Docker Engine installation. Think `docker build`, `docker push` in the cloud.
- **Automatically triggered tasks** - Enable one or more *triggers* to build an image:
  - **Trigger on source code update**
  - **Trigger on base image update**
  - **Trigger on a schedule**
- **Multi-step task** - Extend the single image build-and-push capability of ACR Tasks with multi-step, multi-container-based workflows.

Each ACR Task has an associated source code context - the location of a set of source files used to build a container image or other artifact. Example contexts include a Git repository or a local filesystem.

Tasks can also take advantage of run variables, so you can reuse task definitions and standardize tags for images and artifacts.

Quick task

The inner-loop development cycle, the iterative process of writing code, building, and testing your application before committing to source control, is really the beginning of container lifecycle management.

Before you commit your first line of code, ACR Tasks's quick task feature can provide an integrated development experience by offloading your container image builds to Azure. With quick tasks, you can verify your automated build definitions and catch potential problems prior to committing your code.

Using the familiar `docker build` format, the az acr build command in the Azure CLI takes a context (the set of files to build), sends it ACR Tasks and, by default, pushes the built image to its registry upon completion.

For an introduction, see the quickstart to build and run a container image in Azure Container Registry.

ACR Tasks is designed as a container lifecycle primitive. For example, integrate ACR Tasks into your CI/CD solution. By executing az login with a service principal, your CI/CD solution could then issue az acr build commands to kick off image builds.

Learn how to use quick tasks in the first ACR Tasks tutorial, Build container images in the cloud with Azure Container Registry Tasks.

> **TIP**
> If you want to build and push an image directly from source code, without a Dockerfile, Azure Container Registry provides the az acr pack build command (preview). This tool builds and pushes an image from application source code using Cloud Native Buildpacks.

Trigger task on source code update

Trigger a container image build or multi-step task when code is committed, or a pull request is made or updated, to a public or private Git repository in GitHub or Azure DevOps. For example, configure a build task with the Azure CLI command az acr task create by specifying a Git repository and optionally a branch and Dockerfile. When your team updates code in the repository, an ACR Tasks-created webhook triggers a build of the container image defined in the repo.

ACR Tasks supports the following triggers when you set a Git repo as the task's context:

TRIGGER	ENABLED BY DEFAULT
Commit	Yes
Pull request	No

To configure a source code update trigger, you need to provide the task a personal access token (PAT) to set the webhook in the public or private GitHub or Azure DevOps repo.

> **NOTE**
> Currently, ACR Tasks doesn't support commit or pull request triggers in GitHub Enterprise repos.

Learn how to trigger builds on source code commit in the second ACR Tasks tutorial, Automate container image builds with Azure Container Registry Tasks.

Automate OS and framework patching

The power of ACR Tasks to truly enhance your container build workflow comes from its ability to detect an update to a *base image*. A feature of most container images, a base image is a parent image on which one or more application images are based. Base images typically contain the operating system, and sometimes application frameworks.

You can set up an ACR task to track a dependency on a base image when it builds an application image. When the updated base image is pushed to your registry, or a base image is updated in a public repo such as in Docker Hub, ACR Tasks can automatically build any application images based on it. With this automatic detection and rebuilding, ACR Tasks saves you the time and effort normally required to manually track and update each and every application image referencing your updated base image.

Learn more about base image update triggers for ACR Tasks. And learn how to trigger an image build when a base image is pushed to a container registry in the tutorial Automate container image builds when a base image is updated in a Azure container registry

Schedule a task

Optionally schedule a task by setting up one or more *timer triggers* when you create or update the task. Scheduling a task is useful for running container workloads on a defined schedule, or running maintenance operations or tests on images pushed regularly to your registry. For details, see Run an ACR task on a defined schedule.

Multi-step tasks

Multi-step tasks provide step-based task definition and execution for building, testing, and patching container images in the cloud. Task steps defined in a YAML file specify individual build and push operations for container images or other artifacts. They can also define the execution of one or more containers, with each step using the container as its execution environment.

For example, you can create a multi-step task that automates the following:

1. Build a web application image
2. Run the web application container
3. Build a web application test image
4. Run the web application test container, which performs tests against the running application container
5. If the tests pass, build a Helm chart archive package
6. Perform a `helm upgrade` using the new Helm chart archive package

Multi-step tasks enable you to split the building, running, and testing of an image into more composable steps, with inter-step dependency support. With multi-step tasks in ACR Tasks, you have more granular control over image building, testing, and OS and framework patching workflows.

Learn about multi-step tasks in Run multi-step build, test, and patch tasks in ACR Tasks.

Context locations

The following table shows a few examples of supported context locations for ACR Tasks:

CONTEXT LOCATION	DESCRIPTION	EXAMPLE
Local filesystem	Files within a directory on the local filesystem.	`/home/user/projects/myapp`
GitHub master branch	Files within the master (or other default) branch of a public or private GitHub repository.	`https://github.com/gituser/myapp-repo.git`
GitHub branch	Specific branch of a public or private GitHub repo.	`https://github.com/gituser/myapp-repo.git#mybranch`
GitHub subfolder	Files within a subfolder in a public or private GitHub repo. Example shows combination of a branch and subfolder specification.	`https://github.com/gituser/myapp-repo.git#mybranch:myfolder`
GitHub commit	Specific commit in a public or private GitHub repo. Example shows combination of a commit hash (SHA) and subfolder specification.	`https://github.com/gituser/myapp-repo.git#git-commit-hash:myfolder`
Azure DevOps subfolder	Files within a subfolder in a public or private Azure repo. Example shows combination of branch and subfolder specification.	`https://dev.azure.com/user/myproject/_git/my repo#mybranch:myfolder`
Remote tarball	Files in a compressed archive on a remote webserver.	`http://remoteserver/myapp.tar.gz`

NOTE

When using a private Git repo as a context for a task, you need to provide a personal access token (PAT).

Image platforms

By default, ACR Tasks builds images for the Linux OS and the amd64 architecture. Specify the `--platform` tag to build Windows images or Linux images for other architectures. Specify the OS and optionally a supported architecture in OS/architecture format (for example, `--platform Linux/arm`). For ARM architectures, optionally specify a variant in OS/architecture/variant format (for example, `--platform Linux/arm64/v8`):

OS	ARCHITECTURE
Linux	amd64 arm arm64 386
Windows	amd64

View task logs

Each task run generates log output that you can inspect to determine whether the task steps ran successfully. If you use the az acr build, az acr run, or az acr task run command to trigger the task, log output for the task run is streamed to the console and also stored for later retrieval. When a task is automatically triggered, for example by a source code commit or a base image update, task logs are only stored. View the logs for a task run in the Azure portal, or use the az acr task logs command.

By default, data and logs for task runs in a registry are retained for 30 days and then automatically purged. If you want to archive the data for a task run, enable archiving using the az acr task update-run command. The following example enables archiving for the task run *cf11* in registry *myregistry*.

```
az acr task update-run --registry myregistry --run-id cf11 --no-archive false
```

Next steps

When you're ready to automate container image builds and maintenance in the cloud, check out the ACR Tasks tutorial series.

Optionally install the Docker Extension for Visual Studio Code and the Azure Account extension to work with your Azure container registries. Pull and push images to an Azure container registry, or run ACR Tasks, all within Visual Studio Code.

Quickstart: Build and run a container image using Azure Container Registry Tasks

2/4/2020 • 6 minutes to read • Edit Online

In this quickstart, you use Azure Container Registry Tasks commands to quickly build, push, and run a Docker container image natively within Azure, showing how to offload your "inner-loop" development cycle to the cloud. ACR Tasks is a suite of features within Azure Container Registry to help you manage and modify container images across the container lifecycle.

After this quickstart, explore more advanced features of ACR Tasks. ACR Tasks can automate image builds based on code commits or base image updates, or test multiple containers, in parallel, among other scenarios.

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

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by selecting **Cmd**+**Shift**+**V** on macOS.
4. Select **Enter** to run the code.

You can use the Azure Cloud Shell or a local installation of the Azure CLI to complete this quickstart. If you'd like to use it locally, version 2.0.58 or later is recommended. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create a resource group

If you don't already have a container registry, 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
```

Create a container registry

Create a container registry using the az acr create command. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. In the following example, *myContainerRegistry008* is used. Update this to a unique value.

```
az acr create --resource-group myResourceGroup --name myContainerRegistry008 --sku Basic
```

This example creates a *Basic* registry, a cost-optimized option for developers learning about Azure Container Registry. For details on available service tiers, see Container registry SKUs.

Build an image from a Dockerfile

Now use Azure Container Registry to build an image. First, create a working directory and then create a Dockerfile named *Dockerfile* with the following content. This is a simple example to build a Linux container image, but you can create your own standard Dockerfile and build images for other platforms. Command examples in this article are formatted for the bash shell.

```
echo FROM hello-world > Dockerfile
```

Run the az acr build command to build the image. When successfully built, the image is pushed to your registry. The following example pushes the `sample/hello-world:v1` image. The `.` at the end of the command sets the location of the Dockerfile, in this case the current directory.

```
az acr build --image sample/hello-world:v1 \
  --registry myContainerRegistry008 \
  --file Dockerfile .
```

Output from a successful build and push is similar to the following:

```
Packing source code into tar to upload...
Uploading archived source code from '/tmp/build_archive_b0bc1e5d361b44f0833xxxx41b78c24e.tar.gz'...
Sending context (1.856 KiB) to registry: mycontainerregistry008...
Queued a build with ID: ca8
Waiting for agent...
2019/03/18 21:56:57 Using acb_vol_4c7ffa31-c862-4be3-xxxx-ab8e615c55c4 as the home volume
2019/03/18 21:56:57 Setting up Docker configuration...
2019/03/18 21:56:58 Successfully set up Docker configuration
2019/03/18 21:56:58 Logging in to registry: mycontainerregistry008.azurecr.io
2019/03/18 21:56:59 Successfully logged into mycontainerregistry008.azurecr.io
2019/03/18 21:56:59 Executing step ID: build. Working directory: '', Network: ''
2019/03/18 21:56:59 Obtaining source code and scanning for dependencies...
2019/03/18 21:57:00 Successfully obtained source code and scanned for dependencies
2019/03/18 21:57:00 Launching container with name: build
Sending build context to Docker daemon  13.82kB
Step 1/1 : FROM hello-world
latest: Pulling from library/hello-world
Digest: sha256:2557e3c07ed1e38f26e389462d03ed943586fxxxx21577a99efb77324b0fe535
Successfully built fce289e99eb9
Successfully tagged mycontainerregistry008.azurecr.io/sample/hello-world:v1
2019/03/18 21:57:01 Successfully executed container: build
2019/03/18 21:57:01 Executing step ID: push. Working directory: '', Network: ''
2019/03/18 21:57:01 Pushing image: mycontainerregistry008.azurecr.io/sample/hello-world:v1, attempt 1
The push refers to repository [mycontainerregistry008.azurecr.io/sample/hello-world]
af0b15c8625b: Preparing
af0b15c8625b: Layer already exists
v1: digest: sha256:92c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e899a size: 524
2019/03/18 21:57:03 Successfully pushed image: mycontainerregistry008.azurecr.io/sample/hello-world:v1
2019/03/18 21:57:03 Step ID: build marked as successful (elapsed time in seconds: 2.543040)
2019/03/18 21:57:03 Populating digests for step ID: build...
2019/03/18 21:57:05 Successfully populated digests for step ID: build
2019/03/18 21:57:05 Step ID: push marked as successful (elapsed time in seconds: 1.473581)
2019/03/18 21:57:05 The following dependencies were found:
2019/03/18 21:57:05
- image:
    registry: mycontainerregistry008.azurecr.io
    repository: sample/hello-world
    tag: v1
    digest: sha256:92c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e899a
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/hello-world
    tag: v1
    digest: sha256:2557e3c07ed1e38f26e389462d03ed943586f744621577a99efb77324b0fe535
  git: {}

Run ID: ca8 was successful after 10s
```

Run the image

Now quickly run the image you built and pushed to your registry. Here you use az acr run to run the container command. In your container development workflow, this might be a validation step before you deploy the image, or you could include the command in a multi-step YAML file.

The following example uses `$Registry` to specify the registry where you run the command:

```
az acr run --registry myContainerRegistry008 \
  --cmd '$Registry/sample/hello-world:v1' /dev/null
```

The `cmd` parameter in this example runs the container in its default configuration, but `cmd` supports additional `docker run` parameters or even other `docker` commands.

Output is similar to the following:

```
Packing source code into tar to upload...
Uploading archived source code from '/tmp/run_archive_ebf74da7fcb04683867b129e2ccad5e1.tar.gz'...
Sending context (1.855 KiB) to registry: mycontainerre...
Queued a run with ID: cab
Waiting for an agent...
2019/03/19 19:01:53 Using acb_vol_60e9a538-b466-475f-9565-80c5b93eaa15 as the home volume
2019/03/19 19:01:53 Creating Docker network: acb_default_network, driver: 'bridge'
2019/03/19 19:01:53 Successfully set up Docker network: acb_default_network
2019/03/19 19:01:53 Setting up Docker configuration...
2019/03/19 19:01:54 Successfully set up Docker configuration
2019/03/19 19:01:54 Logging in to registry: mycontainerregistry008.azurecr.io
2019/03/19 19:01:55 Successfully logged into mycontainerregistry008.azurecr.io
2019/03/19 19:01:55 Executing step ID: acb_step_0. Working directory: '', Network: 'acb_default_network'
2019/03/19 19:01:55 Launching container with name: acb_step_0

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

2019/03/19 19:01:56 Successfully executed container: acb_step_0
2019/03/19 19:01:56 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 0.843801)

Run ID: cab was successful after 6s
```

Clean up resources

When no longer needed, you can use the az group delete command to remove the resource group, the container registry, and the container images stored there.

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

Next steps

In this quickstart, you used features of ACR Tasks to quickly build, push, and run a Docker container image natively within Azure, without a local Docker installation. Continue to the Azure Container Registry Tasks tutorials to learn about using ACR Tasks to automate image builds and updates.

Azure Container Registry Tasks tutorials

Tutorial: Build and deploy container images in the cloud with Azure Container Registry Tasks

11/24/2019 • 10 minutes to read • Edit Online

ACR Tasks is a suite of features within Azure Container Registry that provides streamlined and efficient Docker container image builds in Azure. In this article, you learn how to use the *quick task* feature of ACR Tasks.

The "inner-loop" development cycle is the iterative process of writing code, building, and testing your application before committing to source control. A quick task extends your inner-loop to the cloud, providing you with build success validation and automatic pushing of successfully built images to your container registry. Your images are built natively in the cloud, close to your registry, enabling faster deployment.

All your Dockerfile expertise is directly transferrable to ACR Tasks. You don't have to change your Dockerfiles to build in the cloud with ACR Tasks, just the command you run.

In this tutorial, part one of a series:

- Get the sample application source code
- Build a container image in Azure
- Deploy a container to Azure Container Instances

In subsequent tutorials, you learn to use ACR Tasks for automated container image builds on code commit and base image update. ACR Tasks can also run multi-step tasks, using a YAML file to define steps to build, push, and optionally test multiple containers.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.

3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by selecting **Cmd**+**Shift**+**V** on macOS.

4. Select **Enter** to run the code.

If you'd like to use the Azure CLI locally, you must have Azure CLI version **2.0.46** or later installed and logged in with az login. Run `az --version` to find the version. If you need to install or upgrade the CLI, see Install Azure CLI.

Prerequisites

GitHub account

Create an account on https://github.com if you don't already have one. This tutorial series uses a GitHub repository to demonstrate automated image builds in ACR Tasks.

Fork sample repository

Next, use the GitHub UI to fork the sample repository into your GitHub account. In this tutorial, you build a container image from the source in the repo, and in the next tutorial, you push a commit to your fork of the repo to kick off an automated task.

Fork this repository: https://github.com/Azure-Samples/acr-build-helloworld-node

Clone your fork

Once you've forked the repo, clone your fork and enter the directory containing your local clone.

Clone the repo with `git`, replace **<your-github-username>** with your GitHub username:

```
git clone https://github.com/<your-github-username>/acr-build-helloworld-node
```

Enter the directory containing the source code:

```
cd acr-build-helloworld-node
```

Bash shell

The commands in this tutorial series are formatted for the Bash shell. If you prefer to use PowerShell, Command Prompt, or another shell, you may need to adjust the line continuation and environment variable format accordingly.

Build in Azure with ACR Tasks

Now that you've pulled the source code down to your machine, follow these steps to create a container registry and build the container image with ACR Tasks.

To make executing the sample commands easier, the tutorials in this series use shell environment variables. Execute the following command to set the `ACR_NAME` variable. Replace **<registry-name>** with a unique name for your new container registry. The registry name must be unique within Azure, contain only lower case letters, and contain 5-50 alphanumeric characters. The other resources you create in the tutorial are based on this name, so you should need to modify only this first variable.

```
ACR_NAME=<registry-name>
```

With the container registry environment variable populated, you should now be able to copy and paste the remainder of the commands in the tutorial without editing any values. Execute the following commands to create a resource group and container registry:

```
RES_GROUP=$ACR_NAME # Resource Group name

az group create --resource-group $RES_GROUP --location eastus
az acr create --resource-group $RES_GROUP --name $ACR_NAME --sku Standard --location eastus
```

Now that you have a registry, use ACR Tasks to build a container image from the sample code. Execute the az acr build command to perform a *quick task*:

```
az acr build --registry $ACR_NAME --image helloacrtasks:v1 .
```

Output from the az acr build command is similar to the following. You can see the upload of the source code (the "context") to Azure, and the details of the `docker build` operation that the ACR task runs in the cloud. Because ACR tasks use `docker build` to build your images, no changes to your Dockerfiles are required to start using ACR Tasks immediately.

```
$ az acr build --registry $ACR_NAME --image helloacrtasks:v1 .
Packing source code into tar file to upload...
Sending build context (4.813 KiB) to ACR...
Queued a build with build ID: da1
Waiting for build agent...
2018/08/22 18:31:42 Using acb_vol_01185991-be5f-42f0-9403-a36bb997ff35 as the home volume
2018/08/22 18:31:42 Setting up Docker configuration...
2018/08/22 18:31:43 Successfully set up Docker configuration
2018/08/22 18:31:43 Logging in to registry: myregistry.azurecr.io
2018/08/22 18:31:55 Successfully logged in
Sending build context to Docker daemon   21.5kB
Step 1/5 : FROM node:9-alpine
9-alpine: Pulling from library/node
Digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
Status: Image is up to date for node:9-alpine
 ---> a56170f59699
Step 2/5 : COPY . /src
 ---> 88087d7e709a
Step 3/5 : RUN cd /src && npm install
 ---> Running in e80e1263ce9a
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN helloworld@1.0.0 No repository field.

up to date in 0.1s
Removing intermediate container e80e1263ce9a
 ---> 26aac291c02e
Step 4/5 : EXPOSE 80
 ---> Running in 318fb4c124ac
Removing intermediate container 318fb4c124ac
 ---> 113e157d0d5a
Step 5/5 : CMD ["node", "/src/server.js"]
 ---> Running in fe7027a11787
Removing intermediate container fe7027a11787
 ---> 20a27b90eb29
Successfully built 20a27b90eb29
Successfully tagged myregistry.azurecr.io/helloacrtasks:v1
2018/08/22 18:32:11 Pushing image: myregistry.azurecr.io/helloacrtasks:v1, attempt 1
The push refers to repository [myregistry.azurecr.io/helloacrtasks]
6428a18b7034: Preparing
c44b9827df52: Preparing
172ed8ca5e43: Preparing
8c9992f4e5dd: Preparing
8dfad2055603: Preparing
c44b9827df52: Pushed
172ed8ca5e43: Pushed
8dfad2055603: Pushed
6428a18b7034: Pushed
8c9992f4e5dd: Pushed
v1: digest: sha256:b038dcaa72b2889f56deaff7fa675f58c7c666041584f706c783a3958c4ac8d1 size: 1366
2018/08/22 18:32:43 Successfully pushed image: myregistry.azurecr.io/helloacrtasks:v1
2018/08/22 18:32:43 Step ID acb_step_0 marked as successful (elapsed time in seconds: 15.648945)
The following dependencies were found:
- image:
    registry: myregistry.azurecr.io
    repository: helloacrtasks
    tag: v1
    digest: sha256:b038dcaa72b2889f56deaff7fa675f58c7c666041584f706c783a3958c4ac8d1
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 9-alpine
    digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
  git: {}

Run ID: da1 was successful after 1m9.970148252s
```

Near the end of the output, ACR Tasks displays the dependencies it's discovered for your image. This enables ACR Tasks to automate image builds on base image updates, such as when a base image is updated with OS or framework patches. You learn about ACR Tasks support for base image updates later in this tutorial series.

Deploy to Azure Container Instances

ACR tasks automatically push successfully built images to your registry by default, allowing you to deploy them from your registry immediately.

In this section, you create an Azure Key Vault and service principal, then deploy the container to Azure Container Instances (ACI) using the service principal's credentials.

Configure registry authentication

All production scenarios should use service principals to access an Azure container registry. Service principals allow you to provide role-based access control to your container images. For example, you can configure a service principal with pull-only access to a registry.

Create a key vault

If you don't already have a vault in Azure Key Vault, create one with the Azure CLI using the following commands.

```
AKV_NAME=$ACR_NAME-vault

az keyvault create --resource-group $RES_GROUP --name $AKV_NAME
```

Create a service principal and store credentials

You now need to create a service principal and store its credentials in your key vault.

Use the az ad sp create-for-rbac command to create the service principal, and az keyvault secret set to store the service principal's **password** in the vault:

```
# Create service principal, store its password in AKV (the registry *password*)
az keyvault secret set \
  --vault-name $AKV_NAME \
  --name $ACR_NAME-pull-pwd \
  --value $(az ad sp create-for-rbac \
               --name $ACR_NAME-pull \
               --scopes $(az acr show --name $ACR_NAME --query id --output tsv) \
               --role acrpull \
               --query password \
               --output tsv)
```

The `--role` argument in the preceding command configures the service principal with the *acrpull* role, which grants it pull-only access to the registry. To grant both push and pull access, change the `--role` argument to *acrpush*.

Next, store the service principal's *appId* in the vault, which is the **username** you pass to Azure Container Registry for authentication:

```
# Store service principal ID in AKV (the registry *username*)
az keyvault secret set \
    --vault-name $AKV_NAME \
    --name $ACR_NAME-pull-usr \
    --value $(az ad sp show --id http://$ACR_NAME-pull --query appId --output tsv)
```

You've created an Azure Key Vault and stored two secrets in it:

- `$ACR_NAME-pull-usr`: The service principal ID, for use as the container registry **username**.
- `$ACR_NAME-pull-pwd`: The service principal password, for use as the container registry **password**.

You can now reference these secrets by name when you or your applications and services pull images from the registry.

Deploy a container with Azure CLI

Now that the service principal credentials are stored as Azure Key Vault secrets, your applications and services can use them to access your private registry.

Execute the following az container create command to deploy a container instance. The command uses the service principal's credentials stored in Azure Key Vault to authenticate to your container registry.

```
az container create \
    --resource-group $RES_GROUP \
    --name acr-tasks \
    --image $ACR_NAME.azurecr.io/helloacrtasks:v1 \
    --registry-login-server $ACR_NAME.azurecr.io \
    --registry-username $(az keyvault secret show --vault-name $AKV_NAME --name $ACR_NAME-pull-usr --query
value -o tsv) \
    --registry-password $(az keyvault secret show --vault-name $AKV_NAME --name $ACR_NAME-pull-pwd --query
value -o tsv) \
    --dns-name-label acr-tasks-$ACR_NAME \
    --query "{FQDN:ipAddress.fqdn}" \
    --output table
```

The `--dns-name-label` value must be unique within Azure, so the preceding command appends your container registry's name to the container's DNS name label. The output from the command displays the container's fully qualified domain name (FQDN), for example:

```
$ az container create \
>     --resource-group $RES_GROUP \
>     --name acr-tasks \
>     --image $ACR_NAME.azurecr.io/helloacrtasks:v1 \
>     --registry-login-server $ACR_NAME.azurecr.io \
>     --registry-username $(az keyvault secret show --vault-name $AKV_NAME --name $ACR_NAME-pull-usr --
query value -o tsv) \
>     --registry-password $(az keyvault secret show --vault-name $AKV_NAME --name $ACR_NAME-pull-pwd --
query value -o tsv) \
>     --dns-name-label acr-tasks-$ACR_NAME \
>     --query "{FQDN:ipAddress.fqdn}" \
>     --output table
FQDN
-----------------------------------------------
acr-tasks-myregistry.eastus.azurecontainer.io
```

Take note of the container's FQDN, you'll use it in the next section.

Verify the deployment

To watch the startup process of the container, use the az container attach command:

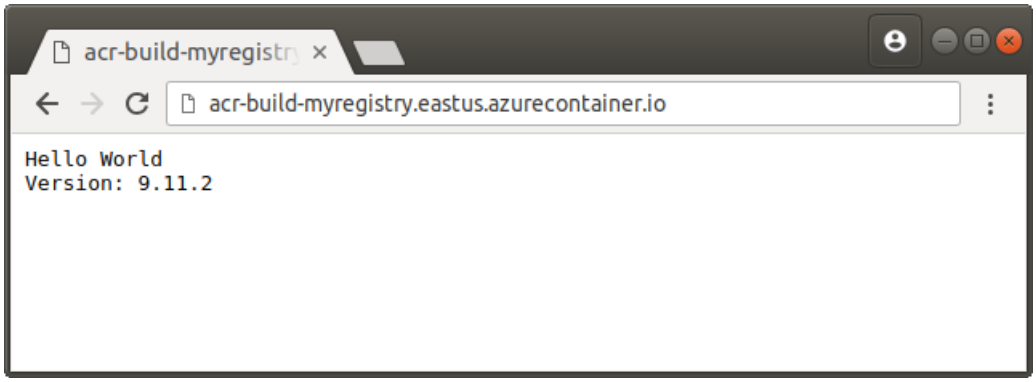
```
az container attach --resource-group $RES_GROUP --name acr-tasks
```

The `az container attach` output first displays the container's status as it pulls the image and starts, then binds your local console's STDOUT and STDERR to that of the container's.

```
$ az container attach --resource-group $RES_GROUP --name acr-tasks
Container 'acr-tasks' is in state 'Running'...
(count: 1) (last timestamp: 2018-08-22 18:39:10+00:00) pulling image
"myregistry.azurecr.io/helloacrtasks:v1"
(count: 1) (last timestamp: 2018-08-22 18:39:15+00:00) Successfully pulled image
"myregistry.azurecr.io/helloacrtasks:v1"
(count: 1) (last timestamp: 2018-08-22 18:39:17+00:00) Created container
(count: 1) (last timestamp: 2018-08-22 18:39:17+00:00) Started container

Start streaming logs:
Server running at http://localhost:80
```

When `Server running at http://localhost:80` appears, navigate to the container's FQDN in your browser to see the running application. The FQDN should have been displayed in the output of the `az container create` command you executed in the previous section.



To detach your console from the container, hit `Control+C`.

Clean up resources

Stop the container instance with the az container delete command:

```
az container delete --resource-group $RES_GROUP --name acr-tasks
```

To remove *all* resources you've created in this tutorial, including the container registry, key vault, and service principal, issue the following commands. These resources are used in the next tutorial in the series, however, so you might want to keep them if you move on directly to the next tutorial.

```
az group delete --resource-group $RES_GROUP
az ad sp delete --id http://$ACR_NAME-pull
```

Next steps

Now that you've tested your inner loop with a quick task, configure a **build task** to trigger container images builds when you commit source code to a Git repository:

Trigger automatic builds with tasks

Tutorial: Automate container image builds in the cloud when you commit source code

12/4/2019 • 8 minutes to read • Edit Online

In addition to a quick task, ACR Tasks supports automated Docker container image builds in the cloud when you commit source code to a Git repository. Supported Git contexts for ACR Tasks include public or private GitHub or Azure repos.

> **NOTE**
> Currently, ACR Tasks doesn't support commit or pull request triggers in GitHub Enterprise repos.

In this tutorial, your ACR task builds and pushes a single container image specified in a Dockerfile when you commit source code to a Git repo. To create a multi-step task that uses a YAML file to define steps to build, push, and optionally test multiple containers on code commit, see Tutorial: Run a multi-step container workflow in the cloud when you commit source code. For an overview of ACR Tasks, see Automate OS and framework patching with ACR Tasks

In this tutorial:

- Create a task
- Test the task
- View task status
- Trigger the task with a code commit

This tutorial assumes you've already completed the steps in the previous tutorial. If you haven't already done so, complete the steps in the Prerequisites section of the previous tutorial before proceeding.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by selecting **Cmd**+**Shift**+**V** on macOS.
4. Select **Enter** to run the code.

If you'd like to use the Azure CLI locally, you must have Azure CLI version **2.0.46** or later installed and logged in with az login. Run `az --version` to find the version. If you need to install or upgrade the CLI, see Install Azure CLI.

Prerequisites

Get sample code

This tutorial assumes you've already completed the steps in the previous tutorial, and have forked and cloned the sample repository. If you haven't already done so, complete the steps in the Prerequisites section of the previous tutorial before proceeding.

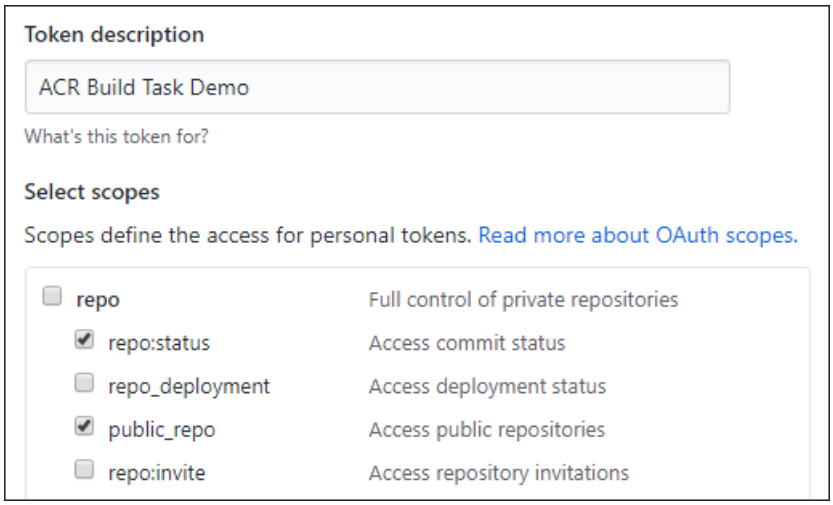
Container registry

You must have an Azure container registry in your Azure subscription to complete this tutorial. If you need a registry, see the previous tutorial, or Quickstart: Create a container registry using the Azure CLI.

Create a GitHub personal access token

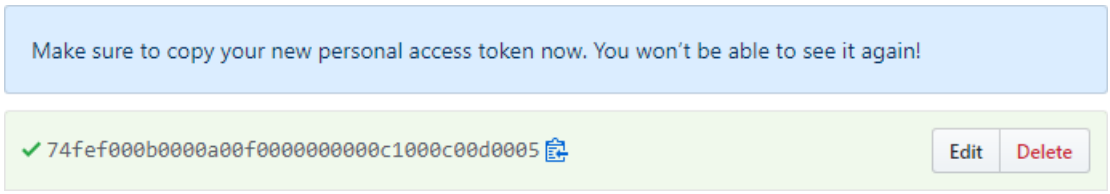
To trigger a task on a commit to a Git repository, ACR Tasks need a personal access token (PAT) to access the repository. If you do not already have a PAT, follow these steps to generate one in GitHub:

1. Navigate to the PAT creation page on GitHub at https://github.com/settings/tokens/new
2. Enter a short **description** for the token, for example, "ACR Tasks Demo"
3. Select scopes for ACR to access the repo. To access a public repo as in this tutorial, under **repo**, enable **repo:status** and **public_repo**



> **NOTE**
> To generate a PAT to access a *private* repo, select the scope for full **repo** control.

4. Select the **Generate token** button (you may be asked to confirm your password)
5. Copy and save the generated token in a **secure location** (you use this token when you define a task in the following section)



Create the build task

Now that you've completed the steps required to enable ACR Tasks to read commit status and create webhooks in a repository, you can create a task that triggers a container image build on commits to the repo.

First, populate these shell environment variables with values appropriate for your environment. This step isn't strictly required, but makes executing the multiline Azure CLI commands in this tutorial a bit easier. If you don't populate these environment variables, you must manually replace each value wherever it appears in the example commands.

```
ACR_NAME=<registry-name>        # The name of your Azure container registry
GIT_USER=<github-username>      # Your GitHub user account name
GIT_PAT=<personal-access-token> # The PAT you generated in the previous section
```

Now, create the task by executing the following az acr task create command:

```
az acr task create \
    --registry $ACR_NAME \
    --name taskhelloworld \
    --image helloworld:{{.Run.ID}} \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git \
    --file Dockerfile \
    --git-access-token $GIT_PAT
```

IMPORTANT

If you previously created tasks during the preview with the `az acr build-task` command, those tasks need to be re-created using the az acr task command.

This task specifies that any time code is committed to the *master* branch in the repository specified by `--context`, ACR Tasks will build the container image from the code in that branch. The Dockerfile specified by `--file` from the repository root is used to build the image. The `--image` argument specifies a parameterized value of `{{.Run.ID}}` for the version portion of the image's tag, ensuring the built image correlates to a specific build, and is tagged uniquely.

Output from a successful az acr task create command is similar to the following:

```
{
  "agentConfiguration": {
    "cpu": 2
  },
  "creationDate": "2018-09-14T22:42:32.972298+00:00",
  "id": "/subscriptions/<Subscription
ID>/resourceGroups/myregistry/providers/Microsoft.ContainerRegistry/registries/myregistry/tasks/taskhelloworl
d",
  "location": "westcentralus",
  "name": "taskhelloworld",
  "platform": {
    "architecture": "amd64",
    "os": "Linux",
    "variant": null
  },
  "provisioningState": "Succeeded",
  "resourceGroup": "myregistry",
  "status": "Enabled",
  "step": {
    "arguments": [],
    "baseImageDependencies": null,
    "contextPath": "https://github.com/gituser/acr-build-helloworld-node",
    "dockerFilePath": "Dockerfile",
    "imageNames": [
      "helloworld:{{.Run.ID}}"
    ],
    "isPushEnabled": true,
    "noCache": false,
    "type": "Docker"
  },
  "tags": null,
  "timeout": 3600,
  "trigger": {
    "baseImageTrigger": {
      "baseImageTriggerType": "Runtime",
      "name": "defaultBaseimageTriggerName",
      "status": "Enabled"
    },
    "sourceTriggers": [
      {
        "name": "defaultSourceTriggerName",
        "sourceRepository": {
          "branch": "master",
          "repositoryUrl": "https://github.com/gituser/acr-build-helloworld-node",
          "sourceControlAuthProperties": null,
          "sourceControlType": "GitHub"
        },
        "sourceTriggerEvents": [
          "commit"
        ],
        "status": "Enabled"
      }
    ]
  },
  "type": "Microsoft.ContainerRegistry/registries/tasks"
}
```


Test the build task

You now have a task that defines your build. To test the build pipeline, trigger a build manually by executing the az acr task run command:

```
az acr task run --registry $ACR_NAME --name taskhelloworld
```

By default, the `az acr task run` command streams the log output to your console when you execute the command.

```
$ az acr task run --registry $ACR_NAME --name taskhelloworld

2018/09/17 22:51:00 Using acb_vol_9ee1f28c-4fd4-43c8-a651-f0ed027bbf0e as the home volume
2018/09/17 22:51:00 Setting up Docker configuration...
2018/09/17 22:51:02 Successfully set up Docker configuration
2018/09/17 22:51:02 Logging in to registry: myregistry.azurecr.io
2018/09/17 22:51:03 Successfully logged in
2018/09/17 22:51:03 Executing step: build
2018/09/17 22:51:03 Obtaining source code and scanning for dependencies...
2018/09/17 22:51:05 Successfully obtained source code and scanned for dependencies
Sending build context to Docker daemon  23.04kB
Step 1/5 : FROM node:9-alpine
9-alpine: Pulling from library/node
Digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
Status: Image is up to date for node:9-alpine
 ---> a56170f59699
Step 2/5 : COPY . /src
 ---> 5f574fcf5816
Step 3/5 : RUN cd /src && npm install
 ---> Running in b1bca3b5f4fc
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN helloworld@1.0.0 No repository field.

up to date in 0.078s
Removing intermediate container b1bca3b5f4fc
 ---> 44457db20dac
Step 4/5 : EXPOSE 80
 ---> Running in 9e6f63ec612f
Removing intermediate container 9e6f63ec612f
 ---> 74c3e8ea0d98
Step 5/5 : CMD ["node", "/src/server.js"]
 ---> Running in 7382eea2a56a
Removing intermediate container 7382eea2a56a
 ---> e33cd684027b
Successfully built e33cd684027b
Successfully tagged myregistry.azurecr.io/helloworld:da2
2018/09/17 22:51:11 Executing step: push
2018/09/17 22:51:11 Pushing image: myregistry.azurecr.io/helloworld:da2, attempt 1
The push refers to repository [myregistry.azurecr.io/helloworld]
4a853682c993: Preparing
[...]
4a853682c993: Pushed
[...]
da2: digest: sha256:c24e62fd848544a5a87f06ea60109dbef9624d03b1124bfe03e1d2c11fd62419 size: 1366
2018/09/17 22:51:21 Successfully pushed image: myregistry.azurecr.io/helloworld:da2
2018/09/17 22:51:21 Step id: build marked as successful (elapsed time in seconds: 7.198937)
2018/09/17 22:51:21 Populating digests for step id: build...
2018/09/17 22:51:22 Successfully populated digests for step id: build
2018/09/17 22:51:22 Step id: push marked as successful (elapsed time in seconds: 10.180456)
The following dependencies were found:
- image:
    registry: myregistry.azurecr.io
    repository: helloworld
    tag: da2
    digest: sha256:c24e62fd848544a5a87f06ea60109dbef9624d03b1124bfe03e1d2c11fd62419
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 9-alpine
    digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
  git:
    git-head-revision: 68cdf2a37cdae0873b8e2f1c4d80ca60541029bf


Run ID: da2 was successful after 27s
```

Trigger a build with a commit

Now that you've tested the task by manually running it, trigger it automatically with a source code change.

First, ensure you're in the directory containing your local clone of the repository:

```
cd acr-build-helloworld-node
```

Next, execute the following commands to create, commit, and push a new file to your fork of the repo on GitHub:

```
echo "Hello World!" > hello.txt
git add hello.txt
git commit -m "Testing ACR Tasks"
git push origin master
```

You may be asked to provide your GitHub credentials when you execute the `git push` command. Provide your GitHub username, and enter the personal access token (PAT) that you created earlier for the password.

```
$ git push origin master
Username for 'https://github.com': <github-username>
Password for 'https://githubuser@github.com': <personal-access-token>
```

Once you've pushed a commit to your repository, the webhook created by ACR Tasks fires and kicks off a build in Azure Container Registry. Display the logs for the currently running task to verify and monitor the build progress:

```
az acr task logs --registry $ACR_NAME
```

Output is similar to the following, showing the currently executing (or last-executed) task:

```
$ az acr task logs --registry $ACR_NAME
Showing logs of the last created run.
Run ID: da4

[...]

Run ID: da4 was successful after 38s
```

List builds

To see a list of the task runs that ACR Tasks has completed for your registry, run the az acr task list-runs command:

```
az acr task list-runs --registry $ACR_NAME --output table
```

Output from the command should appear similar to the following. The runs that ACR Tasks has executed are displayed, and "Git Commit" appears in the TRIGGER column for the most recent task:

```
$ az acr task list-runs --registry $ACR_NAME --output table

RUN ID    TASK             PLATFORM    STATUS     TRIGGER     STARTED               DURATION
--------  --------------  ----------  ---------  ----------  --------------------  ----------
da4       taskhelloworld  Linux       Succeeded  Git Commit  2018-09-17T23:03:45Z  00:00:44
da3       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T22:55:35Z  00:00:35
da2       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T22:50:59Z  00:00:32
da1                       Linux       Succeeded  Manual      2018-09-17T22:29:59Z  00:00:57
```

Next steps

In this tutorial, you learned how to use a task to automatically trigger container image builds in Azure when you commit source code to a Git repository. Move on to the next tutorial to learn how to create tasks that trigger builds when a container image's base image is updated.

Automate builds on base image update

Tutorial: Run a multi-step container workflow in the cloud when you commit source code

11/24/2019 • 14 minutes to read • Edit Online

In addition to a quick task, ACR Tasks supports multi-step, multi-container-based workflows that can automatically trigger when you commit source code to a Git repository.

In this tutorial, you learn how to use example YAML files to define multi-step tasks that build, run, and push one or more container images to a registry when you commit source code. To create a task that only automates a single image build on code commit, see Tutorial: Automate container image builds in the cloud when you commit source code. For an overview of ACR Tasks, see Automate OS and framework patching with ACR Tasks,

In this tutorial:

- Define a multi-step task using a YAML file
- Create a task
- Optionally add credentials to the task to enable access to another registry
- Test the task
- View task status
- Trigger the task with a code commit

This tutorial assumes you've already completed the steps in the previous tutorial. If you haven't already done so, complete the steps in the Prerequisites section of the previous tutorial before proceeding.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by

selecting **Cmd**+**Shift**+**V** on macOS.

4. Select **Enter** to run the code.

If you'd like to use the Azure CLI locally, you must have Azure CLI version **2.0.62** or later installed and logged in with az login. Run `az --version` to find the version. If you need to install or upgrade the CLI, see Install Azure CLI.

Prerequisites

Get sample code

This tutorial assumes you've already completed the steps in the previous tutorial, and have forked and cloned the sample repository. If you haven't already done so, complete the steps in the Prerequisites section of the previous tutorial before proceeding.

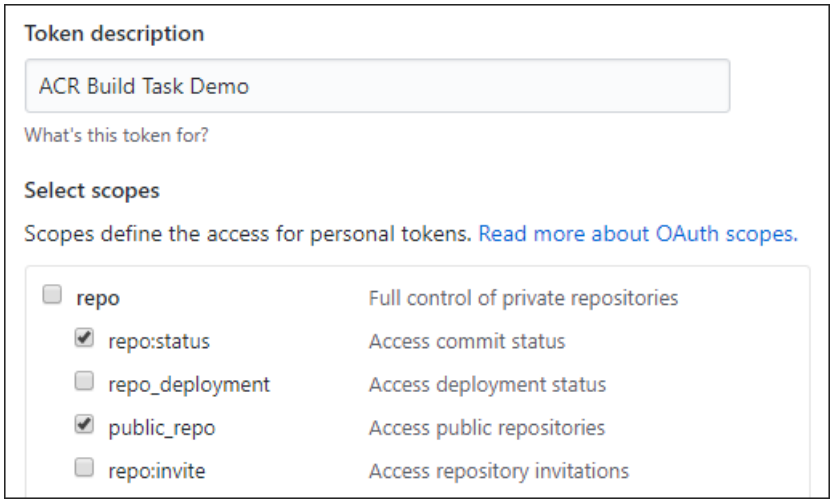
Container registry

You must have an Azure container registry in your Azure subscription to complete this tutorial. If you need a registry, see the previous tutorial, or Quickstart: Create a container registry using the Azure CLI.

Create a GitHub personal access token

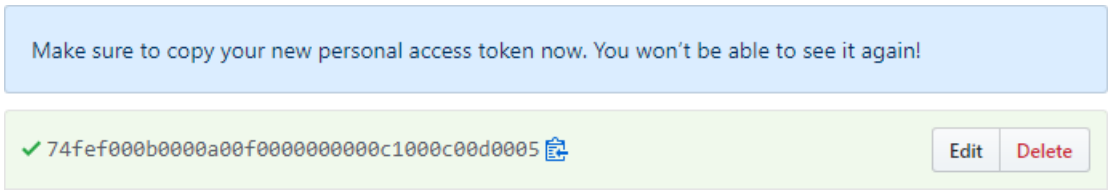
To trigger a task on a commit to a Git repository, ACR Tasks need a personal access token (PAT) to access the repository. If you do not already have a PAT, follow these steps to generate one in GitHub:

1. Navigate to the PAT creation page on GitHub at https://github.com/settings/tokens/new
2. Enter a short **description** for the token, for example, "ACR Tasks Demo"
3. Select scopes for ACR to access the repo. To access a public repo as in this tutorial, under **repo**, enable **repo:status** and **public_repo**



> **NOTE**
> To generate a PAT to access a *private* repo, select the scope for full **repo** control.

4. Select the **Generate token** button (you may be asked to confirm your password)
5. Copy and save the generated token in a **secure location** (you use this token when you define a task in the following section)



Create a multi-step task

Now that you've completed the steps required to enable ACR Tasks to read commit status and create webhooks in a repository, create a multi-step task that triggers building, running, and pushing a container image.

YAML file

You define the steps for a multi-step task in a YAML file. The first example multi-step task for this tutorial is defined in the file `taskmulti.yaml`, which is in the root of the GitHub repo that you cloned:

```
version: v1.0.0
steps:
# Build target image
- build: -t {{.Run.Registry}}/hello-world:{{.Run.ID}} -f Dockerfile .
# Run image
- cmd: -t {{.Run.Registry}}/hello-world:{{.Run.ID}}
  id: test
  detach: true
  ports: ["8080:80"]
- cmd: docker stop test
# Push image
- push:
  - {{.Run.Registry}}/hello-world:{{.Run.ID}}
```

This multi-step task does the following:

1. Runs a `build` step to build an image from the Dockerfile in the working directory. The image targets the `Run.Registry`, the registry where the task is run, and is tagged with a unique ACR Tasks run ID.
2. Runs a `cmd` step to run the image in a temporary container. This example starts a long-running container in the background and returns the container ID, then stops the container. In a real-world scenario, you might include steps to test the running container to ensure it runs correctly.
3. In a `push` step, pushes the image that was built to the run registry.

Task command

First, populate these shell environment variables with values appropriate for your environment. This step isn't strictly required, but makes executing the multiline Azure CLI commands in this tutorial a bit easier. If you don't populate these environment variables, you must manually replace each value wherever it appears in the example commands.

```
ACR_NAME=<registry-name>        # The name of your Azure container registry
GIT_USER=<github-username>      # Your GitHub user account name
GIT_PAT=<personal-access-token> # The PAT you generated in the previous section
```

Now, create the task by executing the following az acr task create command:

```
az acr task create \
    --registry $ACR_NAME \
    --name example1 \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git \
    --file taskmulti.yaml \
    --git-access-token $GIT_PAT
```

This task specifies that any time code is committed to the *master* branch in the repository specified by `--context`, ACR Tasks will run the multi-step task from the code in that branch. The YAML file specified by `--file` from the repository root defines the steps.

Output from a successful az acr task create command is similar to the following:


```
{
  "agentConfiguration": {
    "cpu": 2
  },
  "creationDate": "2019-05-03T03:14:31.763887+00:00",
  "credentials": null,
  "id": "/subscriptions/<Subscription
ID>/resourceGroups/myregistry/providers/Microsoft.ContainerRegistry/registries/myregistry/tasks/taskmulti",
  "location": "westus",
  "name": "example1",
  "platform": {
    "architecture": "amd64",
    "os": "linux",
    "variant": null
  },
  "provisioningState": "Succeeded",
  "resourceGroup": "myresourcegroup",
  "status": "Enabled",
  "step": {
    "baseImageDependencies": null,
    "contextAccessToken": null,
    "contextPath": "https://github.com/gituser/acr-build-helloworld-node.git",
    "taskFilePath": "taskmulti.yaml",
    "type": "FileTask",
    "values": [],
    "valuesFilePath": null
  },
  "tags": null,
  "timeout": 3600,
  "trigger": {
    "baseImageTrigger": {
      "baseImageTriggerType": "Runtime",
      "name": "defaultBaseimageTriggerName",
      "status": "Enabled"
    },
    "sourceTriggers": [
      {
        "name": "defaultSourceTriggerName",
        "sourceRepository": {
          "branch": "master",
          "repositoryUrl": "https://github.com/gituser/acr-build-helloworld-node.git",
          "sourceControlAuthProperties": null,
          "sourceControlType": "Github"
        },
        "sourceTriggerEvents": [
          "commit"
        ],
        "status": "Enabled"
      }
    ]
  },
  "type": "Microsoft.ContainerRegistry/registries/tasks"
}
```


Test the multi-step workflow

To test the multi-step task, trigger it manually by executing the az acr task run command:

```
az acr task run --registry $ACR_NAME --name example1
```

By default, the `az acr task run` command streams the log output to your console when you execute the command. The output shows the progress of running each of the task steps. The output below is condensed to show key steps.

```
Queued a run with ID: cf19
Waiting for an agent...
2019/05/03 03:03:31 Downloading source code...
2019/05/03 03:03:33 Finished downloading source code
2019/05/03 03:03:33 Using acb_vol_cfe6bd55-3076-4215-8091-6a81aec3d1b1 as the home volume
2019/05/03 03:03:33 Creating Docker network: acb_default_network, driver: 'bridge'
2019/05/03 03:03:34 Successfully set up Docker network: acb_default_network
2019/05/03 03:03:34 Setting up Docker configuration...
2019/05/03 03:03:34 Successfully set up Docker configuration
2019/05/03 03:03:34 Logging in to registry: myregistry.azurecr.io
2019/05/03 03:03:35 Successfully logged into myregistry.azurecr.io
2019/05/03 03:03:35 Executing step ID: acb_step_0. Working directory: '', Network: 'acb_default_network'
2019/05/03 03:03:35 Scanning for dependencies...
2019/05/03 03:03:36 Successfully scanned dependencies
2019/05/03 03:03:36 Launching container with name: acb_step_0
Sending build context to Docker daemon  24.06kB

[...]

Successfully built f669bfd170af
Successfully tagged myregistry.azurecr.io/hello-world:cf19
2019/05/03 03:03:43 Successfully executed container: acb_step_0
2019/05/03 03:03:43 Executing step ID: acb_step_1. Working directory: '', Network: 'acb_default_network'
2019/05/03 03:03:43 Launching container with name: acb_step_1
279b1cb6e092b64c8517c5506fcb45494cd5a0bd10a6beca3ba97f25c5d940cd
2019/05/03 03:03:44 Successfully executed container: acb_step_1
2019/05/03 03:03:44 Executing step ID: acb_step_2. Working directory: '', Network: 'acb_default_network'
2019/05/03 03:03:44 Pushing image: myregistry.azurecr.io/hello-world:cf19, attempt 1

[...]

2019/05/03 03:03:46 Successfully pushed image: myregistry.azurecr.io/hello-world:cf19
2019/05/03 03:03:46 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 7.425169)
2019/05/03 03:03:46 Populating digests for step ID: acb_step_0...
2019/05/03 03:03:47 Successfully populated digests for step ID: acb_step_0
2019/05/03 03:03:47 Step ID: acb_step_1 marked as successful (elapsed time in seconds: 0.827129)
2019/05/03 03:03:47 Step ID: acb_step_2 marked as successful (elapsed time in seconds: 2.112113)
2019/05/03 03:03:47 The following dependencies were found:
2019/05/03 03:03:47
- image:
    registry: myregistry.azurecr.io
    repository: hello-world
    tag: cf19
    digest: sha256:6b981a8ca8596e840228c974c929db05c0727d8630465de536be74104693467a
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 9-alpine
    digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
  git:
    git-head-revision: 1a3065388a0238e52865db1c8f3e97492a43444c

Run ID: cf19 was successful after 18s
```

Trigger a build with a commit

Now that you've tested the task by manually running it, trigger it automatically with a source code change.

First, ensure you're in the directory containing your local clone of the repository:

```
cd acr-build-helloworld-node
```

Next, execute the following commands to create, commit, and push a new file to your fork of the repo on GitHub:

```
echo "Hello World!" > hello.txt
git add hello.txt
git commit -m "Testing ACR Tasks"
git push origin master
```

You may be asked to provide your GitHub credentials when you execute the `git push` command. Provide your GitHub username, and enter the personal access token (PAT) that you created earlier for the password.

```
$ git push origin master
Username for 'https://github.com': <github-username>
Password for 'https://githubuser@github.com': <personal-access-token>
```

Once you've pushed a commit to your repository, the webhook created by ACR Tasks fires and kicks off the task in Azure Container Registry. Display the logs for the currently running task to verify and monitor the build progress:

```
az acr task logs --registry $ACR_NAME
```

Output is similar to the following, showing the currently executing (or last-executed) task:

```
$ az acr task logs --registry $ACR_NAME
Showing logs of the last created run.
Run ID: cf1d

[...]

Run ID: cf1d was successful after 37s
```

List builds

To see a list of the task runs that ACR Tasks has completed for your registry, run the az acr task list-runs command:

```
az acr task list-runs --registry $ACR_NAME --output table
```

Output from the command should appear similar to the following. The runs that ACR Tasks has executed are displayed, and "Git Commit" appears in the TRIGGER column for the most recent task:

```
$ az acr task list-runs --registry $ACR_NAME --output table

RUN ID    TASK       PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  ---------  ----------  ---------  ---------  --------------------  ----------
cf1d      example1   linux       Succeeded  Commit     2019-05-03T04:16:44Z  00:00:37
cf1c      example1   linux       Succeeded  Commit     2019-05-03T04:16:44Z  00:00:39
cf1b      example1   linux       Succeeded  Manual     2019-05-03T03:10:30Z  00:00:31
cf1a      example1   linux       Succeeded  Commit     2019-05-03T03:09:32Z  00:00:31
cf19      example1   linux       Succeeded  Manual     2019-05-03T03:03:30Z  00:00:21
```

Create a multi-registry multi-step task

ACR Tasks by default has permissions to push or pull images from the registry where the task runs. You might want to run a multi-step task that targets one or more registries in addition to the run registry. For example, you might need to build images in one registry, and store images with different tags in a second registry that is accessed by a production system. This example shows you how to create such a task and provide credentials for another registry.

If you don't already have a second registry, create one for this example. If you need a registry, see the previous tutorial, or Quickstart: Create a container registry using the Azure CLI.

To create the task, you need the name of the registry login server, which is of the form *mycontainerregistrydate.azurecr.io* (all lowercase). In this example, you use the second registry to store images tagged by build date.

YAML file

The second example multi-step task for this tutorial is defined in the file `taskmulti-multiregistry.yaml`, which is in the root of the GitHub repo that you cloned:

```
version: v1.0.0
steps:
# Build target images
- build: -t {{.Run.Registry}}/hello-world:{{.Run.ID}} -f Dockerfile .
- build: -t {{.Values.regDate}}/hello-world:{{.Run.Date}} -f Dockerfile .
# Run image
- cmd: -t {{.Run.Registry}}/hello-world:{{.Run.ID}}
  id: test
  detach: true
  ports: ["8080:80"]
- cmd: docker stop test
# Push images
- push:
  - {{.Run.Registry}}/hello-world:{{.Run.ID}}
  - {{.Values.regDate}}/hello-world:{{.Run.Date}}
```

This multi-step task does the following:

1. Runs two `build` steps to build images from the Dockerfile in the working directory:
   - The first targets the `Run.Registry`, the registry where the task is run, and is tagged with the ACR Tasks run ID.
   - The second targets the registry identified by the value of `regDate`, which you set when you create the task (or provide through an external `values.yaml` file passed to `az acr task create`). This image is tagged with the run date.
2. Runs a `cmd` step to run one of the built containers. This example starts a long-running container in the background and returns the container ID, then stops the container. In a real-world scenario, you might test a running container to ensure it runs correctly.
3. In a `push` step, pushes the images that were built, the first to the run registry, the second to the registry identified by `regDate`.

Task command

Using the shell environment variables defined previously, create the task by executing the following az acr task create command. Substitute the name of your registry for *mycontainerregistrydate*.

```
az acr task create \
    --registry $ACR_NAME \
    --name example2 \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git \
    --file taskmulti-multiregistry.yaml \
    --git-access-token $GIT_PAT \
    --set regDate=mycontainerregistrydate.azurecr.io
```

Add task credential

To push images to the registry identified by the value of `regDate`, use the az acr task credential add command to add login credentials for that registry to the task.

For this example, we recommend that you create a service principal with access to the registry scoped to the *AcrPush* role. To create the service principal, see this Azure CLI script.

Pass the service principal application ID and password in the following `az acr task credential add` command:

```
az acr task credential add --name example2 \
    --registry $ACR_NAME \
    --login-server mycontainerregistrydate.azurecr.io \
    --username <service-principal-application-id> \
    --password <service-principal-password>
```

The CLI returns the name of the registry login server you added.

Test the multi-step workflow

As in the preceding example, to test the multi-step task, trigger it manually by executing the az acr task run command. To trigger the task with a commit to the Git repository, see the section Trigger a build with a commit.

```
az acr task run --registry $ACR_NAME --name example2
```

By default, the `az acr task run` command streams the log output to your console when you execute the command. As before, the output shows the progress of running each of the task steps. The output is condensed to show key steps.

Output:

```
Queued a run with ID: cf1g
Waiting for an agent...
2019/05/03 04:33:39 Downloading source code...
2019/05/03 04:33:41 Finished downloading source code
2019/05/03 04:33:42 Using acb_vol_4569b017-29fe-42bd-83b2-25c45a8ac807 as the home volume
2019/05/03 04:33:42 Creating Docker network: acb_default_network, driver: 'bridge'
2019/05/03 04:33:43 Successfully set up Docker network: acb_default_network
2019/05/03 04:33:43 Setting up Docker configuration...
2019/05/03 04:33:44 Successfully set up Docker configuration
2019/05/03 04:33:44 Logging in to registry: mycontainerregistry.azurecr.io
2019/05/03 04:33:45 Successfully logged into mycontainerregistry.azurecr.io
2019/05/03 04:33:45 Logging in to registry: mycontainerregistrydate.azurecr.io
2019/05/03 04:33:47 Successfully logged into mycontainerregistrydate.azurecr.io
2019/05/03 04:33:47 Executing step ID: acb_step_0. Working directory: '', Network: 'acb_default_network'
2019/05/03 04:33:47 Scanning for dependencies...
2019/05/03 04:33:47 Successfully scanned dependencies
2019/05/03 04:33:47 Launching container with name: acb_step_0
Sending build context to Docker daemon  25.09kB

[...]

Successfully tagged mycontainerregistry.azurecr.io/hello-world:cf1g
```

```
2019/05/03 04:33:55 Successfully executed container: acb_step_0
2019/05/03 04:33:55 Executing step ID: acb_step_1. Working directory: '', Network: 'acb_default_network'
2019/05/03 04:33:55 Scanning for dependencies...
2019/05/03 04:33:56 Successfully scanned dependencies
2019/05/03 04:33:56 Launching container with name: acb_step_1
Sending build context to Docker daemon  25.09kB

[...]

Successfully tagged mycontainerregistrydate.azurecr.io/hello-world:20190503-043342z
2019/05/03 04:33:57 Successfully executed container: acb_step_1
2019/05/03 04:33:57 Executing step ID: acb_step_2. Working directory: '', Network: 'acb_default_network'
2019/05/03 04:33:57 Launching container with name: acb_step_2
721437ff674051b6be63cbcd2fa8eb085eacbf38d7d632f1a079320133182101
2019/05/03 04:33:58 Successfully executed container: acb_step_2
2019/05/03 04:33:58 Executing step ID: acb_step_3. Working directory: '', Network: 'acb_default_network'
2019/05/03 04:33:58 Launching container with name: acb_step_3
test
2019/05/03 04:34:09 Successfully executed container: acb_step_3
2019/05/03 04:34:09 Executing step ID: acb_step_4. Working directory: '', Network: 'acb_default_network'
2019/05/03 04:34:09 Pushing image: mycontainerregistry.azurecr.io/hello-world:cf1g, attempt 1
The push refers to repository [mycontainerregistry.azurecr.io/hello-world]

[...]

2019/05/03 04:34:12 Successfully pushed image: mycontainerregistry.azurecr.io/hello-world:cf1g
2019/05/03 04:34:12 Pushing image: mycontainerregistrydate.azurecr.io/hello-world:20190503-043342z, attempt 1
The push refers to repository [mycontainerregistrydate.azurecr.io/hello-world]

[...]

2019/05/03 04:34:19 Successfully pushed image: mycontainerregistrydate.azurecr.io/hello-world:20190503-043342z
2019/05/03 04:34:19 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 8.125744)
2019/05/03 04:34:19 Populating digests for step ID: acb_step_0...
2019/05/03 04:34:21 Successfully populated digests for step ID: acb_step_0
2019/05/03 04:34:21 Step ID: acb_step_1 marked as successful (elapsed time in seconds: 2.009281)
2019/05/03 04:34:21 Populating digests for step ID: acb_step_1...
2019/05/03 04:34:23 Successfully populated digests for step ID: acb_step_1
2019/05/03 04:34:23 Step ID: acb_step_2 marked as successful (elapsed time in seconds: 0.795440)
2019/05/03 04:34:23 Step ID: acb_step_3 marked as successful (elapsed time in seconds: 11.446775)
2019/05/03 04:34:23 Step ID: acb_step_4 marked as successful (elapsed time in seconds: 9.734973)
2019/05/03 04:34:23 The following dependencies were found:
2019/05/03 04:34:23
- image:
    registry: mycontainerregistry.azurecr.io
    repository: hello-world
    tag: cf1g
    digest: sha256:75354e9edb995e8661438bad9913deed87a185fddd0193811f916d684b71a5d2
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 9-alpine
    digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
  git:
    git-head-revision: 9d9023473c46a5e2c315681b11eb4552ef0faccc
- image:
    registry: mycontainerregistrydate.azurecr.io
    repository: hello-world
    tag: 20190503-043342z
    digest: sha256:75354e9edb995e8661438bad9913deed87a185fddd0193811f916d684b71a5d2
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 9-alpine
    digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
  git:
    git-head-revision: 9d9023473c46a5e2c315681b11eb4552ef0faccc

Run ID: cf1g was successful after 46s
```

Next steps

In this tutorial, you learned how to create multi-step, multi-container-based tasks that automatically trigger when you commit source code to a Git repository. For advanced features of multi-step tasks, including parallel and dependent step execution, see the ACR Tasks YAML reference. Move on to the next tutorial to learn how to create tasks that trigger builds when a container image's base image is updated.

Automate builds on base image update

Tutorial: Automate container image builds when a base image is updated in an Azure container registry

2/25/2020 • 9 minutes to read • Edit Online

ACR Tasks supports automated container image builds when a container's base image is updated, such as when you patch the OS or application framework in one of your base images.

In this tutorial, you learn how to create an ACR task that triggers a build in the cloud when a container's base image is pushed to the same registry. You can also try a tutorial to create an ACR task that triggers an image build when a base image is pushed to another Azure container registry.

In this tutorial:

- Build the base image
- Create an application image in the same registry to track the base image
- Update the base image to trigger an application image task
- Display the triggered task
- Verify updated application image

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by selecting **Cmd**+**Shift**+**V** on macOS.
4. Select **Enter** to run the code.

If you'd like to use the Azure CLI locally, you must have the Azure CLI version **2.0.46** or later installed. Run

`az --version` to find the version. If you need to install or upgrade the CLI, see Install Azure CLI.

Prerequisites

Complete the previous tutorials

This tutorial assumes you've already completed the steps in the first two tutorials in the series, in which you:

- Create Azure container registry
- Fork sample repository
- Clone sample repository
- Create GitHub personal access token

If you haven't already done so, complete the following tutorials before proceeding:

Build container images in the cloud with Azure Container Registry Tasks

Automate container image builds with Azure Container Registry Tasks

Configure the environment

Populate these shell environment variables with values appropriate for your environment. This step isn't strictly required, but makes executing the multiline Azure CLI commands in this tutorial a bit easier. If you don't populate these environment variables, you must manually replace each value wherever it appears in the example commands.

```
ACR_NAME=<registry-name>        # The name of your Azure container registry
GIT_USER=<github-username>      # Your GitHub user account name
GIT_PAT=<personal-access-token> # The PAT you generated in the second tutorial
```

Base image update scenario

This tutorial walks you through a base image update scenario in which a base image and an application image are maintained in a single registry.

The code sample includes two Dockerfiles: an application image, and an image it specifies as its base. In the following sections, you create an ACR task that automatically triggers a build of the application image when a new version of the base image is pushed to the same container registry.

- Dockerfile-app: A small Node.js web application that renders a static web page displaying the Node.js version on which it's based. The version string is simulated: it displays the contents of an environment variable, `NODE_VERSION`, that's defined in the base image.
- Dockerfile-base: The image that `Dockerfile-app` specifies as its base. It is itself based on a Node image, and includes the `NODE_VERSION` environment variable.

In the following sections, you create a task, update the `NODE_VERSION` value in the base image Dockerfile, then use ACR Tasks to build the base image. When the ACR task pushes the new base image to your registry, it automatically triggers a build of the application image. Optionally, you run the application container image locally to see the different version strings in the built images.

In this tutorial, your ACR task builds and pushes an application container image specified in a Dockerfile. ACR Tasks can also run multi-step tasks, using a YAML file to define steps to build, push, and optionally test multiple containers.

Build the base image

Start by building the base image with an ACR Tasks *quick task*, using az acr build. As discussed in the first tutorial

in the series, this process not only builds the image, but pushes it to your container registry if the build is successful.

```
az acr build --registry $ACR_NAME --image baseimages/node:9-alpine --file Dockerfile-base .
```

Create a task

Next, create a task with az acr task create:

```
az acr task create \
    --registry $ACR_NAME \
    --name taskhelloworld \
    --image helloworld:{{.Run.ID}} \
    --arg REGISTRY_NAME=$ACR_NAME.azurecr.io \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git \
    --file Dockerfile-app \
    --git-access-token $GIT_PAT
```

This task is similar to the task created in the previous tutorial. It instructs ACR Tasks to trigger an image build when commits are pushed to the repository specified by `--context`. While the Dockerfile used to build the image in the previous tutorial specifies a public base image (`FROM node:9-alpine`), the Dockerfile in this task, Dockerfile-app, specifies a base image in the same registry:

```
FROM ${REGISTRY_NAME}/baseimages/node:9-alpine
```

This configuration makes it easy to simulate a framework patch in the base image later in this tutorial.

Build the application container

Use az acr task run to manually trigger the task and build the application image. This step is needed so that the task tracks the application image's dependency on the base image.

```
az acr task run --registry $ACR_NAME --name taskhelloworld
```

Once the task has completed, take note of the **Run ID** (for example, "da6") if you wish to complete the following optional step.

Optional: Run application container locally

If you're working locally (not in the Cloud Shell), and you have Docker installed, run the container to see the application rendered in a web browser before you rebuild its base image. If you're using the Cloud Shell, skip this section (Cloud Shell does not support `az acr login` or `docker run`).

First, authenticate to your container registry with az acr login:

```
az acr login --name $ACR_NAME
```

Now, run the container locally with `docker run`. Replace **<run-id>** with the Run ID found in the output from the previous step (for example, "da6"). This example names the container `myapp` and includes the `--rm` parameter to remove the container when you stop it.

```
docker run -d -p 8080:80 --name myapp --rm $ACR_NAME.azurecr.io/helloworld:<run-id>
```

Navigate to `http://localhost:8080` in your browser, and you should see the Node.js version number rendered in the web page, similar to the following. In a later step, you bump the version by adding an "a" to the version string.

To stop and remove the container, run the following command:

```
docker stop myapp
```

List the builds

Next, list the task runs that ACR Tasks has completed for your registry using the az acr task list-runs command:

```
az acr task list-runs --registry $ACR_NAME --output table
```

If you completed the previous tutorial (and didn't delete the registry), you should see output similar to the following. Take note of the number of task runs, and the latest RUN ID, so you can compare the output after you update the base image in the next section.

```
$ az acr task list-runs --registry $ACR_NAME --output table

RUN ID    TASK            PLATFORM    STATUS     TRIGGER     STARTED               DURATION
--------  --------------  ----------  ---------  ----------  --------------------  ----------
da6       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T23:07:22Z  00:00:38
da5                       Linux       Succeeded  Manual      2018-09-17T23:06:33Z  00:00:31
da4       taskhelloworld  Linux       Succeeded  Git Commit  2018-09-17T23:03:45Z  00:00:44
da3       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T22:55:35Z  00:00:35
da2       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T22:50:59Z  00:00:32
da1                       Linux       Succeeded  Manual      2018-09-17T22:29:59Z  00:00:57
```

Update the base image

Here you simulate a framework patch in the base image. Edit **Dockerfile-base**, and add an "a" after the version number defined in `NODE_VERSION`:

```
ENV NODE_VERSION 9.11.2a
```

Run a quick task to build the modified base image. Take note of the **Run ID** in the output.

```
az acr build --registry $ACR_NAME --image baseimages/node:9-alpine --file Dockerfile-base .
```

Once the build is complete and the ACR task has pushed the new base image to your registry, it triggers a build of the application image. It may take few moments for the task you created earlier to trigger the application image build, as it must detect the newly built and pushed base image.

List updated build

Now that you've updated the base image, list your task runs again to compare to the earlier list. If at first the output doesn't differ, periodically run the command to see the new task run appear in the list.

```
az acr task list-runs --registry $ACR_NAME --output table
```

Output is similar to the following. The TRIGGER for the last-executed build should be "Image Update", indicating that the task was kicked off by your quick task of the base image.

```
$ az acr task list-runs --registry $ACR_NAME --output table

Run ID    TASK            PLATFORM    STATUS     TRIGGER       STARTED               DURATION
--------  --------------  ----------  ---------  ------------  --------------------  ----------
da8       taskhelloworld  Linux       Succeeded  Image Update  2018-09-17T23:11:50Z  00:00:33
da7                       Linux       Succeeded  Manual        2018-09-17T23:11:27Z  00:00:35
da6       taskhelloworld  Linux       Succeeded  Manual        2018-09-17T23:07:22Z  00:00:38
da5                       Linux       Succeeded  Manual        2018-09-17T23:06:33Z  00:00:31
da4       taskhelloworld  Linux       Succeeded  Git Commit    2018-09-17T23:03:45Z  00:00:44
da3       taskhelloworld  Linux       Succeeded  Manual        2018-09-17T22:55:35Z  00:00:35
da2       taskhelloworld  Linux       Succeeded  Manual        2018-09-17T22:50:59Z  00:00:32
da1                       Linux       Succeeded  Manual        2018-09-17T22:29:59Z  00:00:57
```

If you'd like to perform the following optional step of running the newly built container to see the updated version number, take note of the **RUN ID** value for the Image Update-triggered build (in the preceding output, it's "da8").

Optional: Run newly built image

If you're working locally (not in the Cloud Shell), and you have Docker installed, run the new application image once its build has completed. Replace `<run-id>` with the RUN ID you obtained in the previous step. If you're using the Cloud Shell, skip this section (Cloud Shell does not support `docker run`).

```
docker run -d -p 8081:80 --name updatedapp --rm $ACR_NAME.azurecr.io/helloworld:<run-id>
```

Navigate to http://localhost:8081 in your browser, and you should see the updated Node.js version number (with the "a") in the web page:

What's important to note is that you updated your **base** image with a new version number, but the last-built **application** image displays the new version. ACR Tasks picked up your change to the base image, and rebuilt your application image automatically.

To stop and remove the container, run the following command:

```
docker stop updatedapp
```

Next steps

In this tutorial, you learned how to use a task to automatically trigger container image builds when the image's base image has been updated. Now, move on to the next tutorial to learn how to trigger tasks on a defined schedule.

Run a task on a schedule

Tutorial: Automate container image builds when a base image is updated in another private container registry

2/25/2020 • 10 minutes to read • Edit Online

ACR Tasks supports automated image builds when a container's base image is updated, such as when you patch the OS or application framework in one of your base images.

In this tutorial, you learn how to create an ACR task that triggers a build in the cloud when a container's base image is pushed to another Azure container registry. You can also try a tutorial to create an ACR task that triggers an image build when a base image is pushed to the same Azure container registry.

In this tutorial:

- Build the base image in a base registry
- Create an application build task in another registry to track the base image
- Update the base image to trigger an application image task
- Display the triggered task
- Verify updated application image

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl**+**Shift**+**V** on Windows and Linux or by selecting **Cmd**+**Shift**+**V** on macOS.
4. Select **Enter** to run the code.

If you'd like to use the Azure CLI locally, you must have the Azure CLI version **2.0.68** or later installed. Run `az --version` to find the version. If you need to install or upgrade the CLI, see Install Azure CLI.

Prerequisites

Complete the previous tutorials

This tutorial assumes you've already completed the steps in the first two tutorials in the series, in which you:

- Create Azure container registry
- Fork sample repository
- Clone sample repository
- Create GitHub personal access token

If you haven't already done so, complete the following tutorials before proceeding:

Build container images in the cloud with Azure Container Registry Tasks

Automate container image builds with Azure Container Registry Tasks

In addition to the container registry created for the previous tutorials, you need to create a registry to store the base images. If you want to, create the second registry in a different location than the original registry.

Configure the environment

Populate these shell environment variables with values appropriate for your environment. This step isn't strictly required, but makes executing the multiline Azure CLI commands in this tutorial a bit easier. If you don't populate these environment variables, you must manually replace each value wherever it appears in the example commands.

```
BASE_ACR=<base-registry-name>   # The name of your Azure container registry for base images
ACR_NAME=<registry-name>        # The name of your Azure container registry for application images
GIT_USER=<github-username>      # Your GitHub user account name
GIT_PAT=<personal-access-token> # The PAT you generated in the second tutorial
```

Base image update scenario

This tutorial walks you through a base image update scenario. This scenario reflects a development workflow to manage base images in a common, private container registry when creating application images in other registries. The base images could specify common operating systems and frameworks used by a team, or even common service components.

For example, developers who develop application images in their own registries can access a set of base images maintained in the common base registry. The base registry can be in another region or even geo-replicated.

The code sample includes two Dockerfiles: an application image, and an image it specifies as its base. In the following sections, you create an ACR task that automatically triggers a build of the application image when a new version of the base image is pushed to a different Azure container registry.

- Dockerfile-app: A small Node.js web application that renders a static web page displaying the Node.js version on which it's based. The version string is simulated: it displays the contents of an environment variable, `NODE_VERSION`, that's defined in the base image.
- Dockerfile-base: The image that `Dockerfile-app` specifies as its base. It is itself based on a Node image, and includes the `NODE_VERSION` environment variable.

In the following sections, you create a task, update the `NODE_VERSION` value in the base image Dockerfile, then use ACR Tasks to build the base image. When the ACR task pushes the new base image to your registry, it automatically triggers a build of the application image. Optionally, you run the application container image locally

to see the different version strings in the built images.

In this tutorial, your ACR task builds and pushes an application container image specified in a Dockerfile. ACR Tasks can also run multi-step tasks, using a YAML file to define steps to build, push, and optionally test multiple containers.

Build the base image

Start by building the base image with an ACR Tasks *quick task*, using az acr build. As discussed in the first tutorial in the series, this process not only builds the image, but pushes it to your container registry if the build is successful. In this example, the image is pushed to the base image registry.

```
az acr build --registry $BASE_ACR --image baseimages/node:9-alpine --file Dockerfile-base .
```

Create a task to track the private base image

Next, create a task in the application image registry with az acr task create, enabling a managed identity. The managed identity is used in later steps so that the task authenticates with the base image registry.

This example uses a system-assigned identity, but you could create and enable a user-assigned managed identity for certain scenarios. For details, see Cross-registry authentication in an ACR task using an Azure-managed identity.

```
az acr task create \
    --registry $ACR_NAME \
    --name taskhelloworld \
    --image helloworld:{{.Run.ID}} \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git \
    --file Dockerfile-app \
    --git-access-token $GIT_PAT \
    --arg REGISTRY_NAME=$BASE_ACR.azurecr.io \
    --assign-identity
```

This task is similar to the task created in the previous tutorial. It instructs ACR Tasks to trigger an image build when commits are pushed to the repository specified by `--context`. While the Dockerfile used to build the image in the previous tutorial specifies a public base image (`FROM node:9-alpine`), the Dockerfile in this task, Dockerfile-app, specifies a base image in the base image registry:

```
FROM ${REGISTRY_NAME}/baseimages/node:9-alpine
```

This configuration makes it easy to simulate a framework patch in the base image later in this tutorial.

Give identity pull permissions to base registry

To give the task's managed identity permissions to pull images from the base image registry, first run az acr task show to get the service principal ID of the identity. Then run az acr show to get the resource ID of the base registry:

```
# Get service principal ID of the task
principalID=$(az acr task show --name taskhelloworld --registry $ACR_NAME --query identity.principalId --
output tsv)

# Get resource ID of the base registry
baseregID=$(az acr show --name $BASE_ACR --query id --output tsv)
```

Assign the managed identity pull permissions to the registry by running az role assignment create:

```
az role assignment create \
  --assignee $principalID \
  --scope $baseregID --role acrpull
```

Add target registry credentials to the task

Run az acr task credential add to add credentials to the task. Pass the `--use-identity [system]` parameter to indicate that the task's system-assigned managed identity can access the credentials.

```
az acr task credential add \
  --name taskhelloworld \
  --registry $ACR_NAME \
  --login-server $BASE_ACR.azurecr.io \
  --use-identity [system]
```

Manually run the task

Use az acr task run to manually trigger the task and build the application image. This step is needed so that the task tracks the application image's dependency on the base image.

```
az acr task run --registry $ACR_NAME --name taskhelloworld
```

Once the task has completed, take note of the **Run ID** (for example, "da6") if you wish to complete the following optional step.

Optional: Run application container locally

If you're working locally (not in the Cloud Shell), and you have Docker installed, run the container to see the application rendered in a web browser before you rebuild its base image. If you're using the Cloud Shell, skip this section (Cloud Shell does not support `az acr login` or `docker run`).

First, authenticate to your container registry with az acr login:

```
az acr login --name $ACR_NAME
```

Now, run the container locally with `docker run`. Replace **<run-id>** with the Run ID found in the output from the previous step (for example, "da6"). This example names the container `myapp` and includes the `--rm` parameter to remove the container when you stop it.

```
docker run -d -p 8080:80 --name myapp --rm $ACR_NAME.azurecr.io/helloworld:<run-id>
```

Navigate to `http://localhost:8080` in your browser, and you should see the Node.js version number rendered in the web page, similar to the following. In a later step, you bump the version by adding an "a" to the version string.

To stop and remove the container, run the following command:

```
docker stop myapp
```

List the builds

Next, list the task runs that ACR Tasks has completed for your registry using the az acr task list-runs command:

```
az acr task list-runs --registry $ACR_NAME --output table
```

If you completed the previous tutorial (and didn't delete the registry), you should see output similar to the following. Take note of the number of task runs, and the latest RUN ID, so you can compare the output after you update the base image in the next section.

```
$ az acr task list-runs --registry $ACR_NAME --output table

RUN ID    TASK            PLATFORM    STATUS     TRIGGER     STARTED               DURATION
--------  --------------  ----------  ---------  ----------  --------------------  ----------
da6       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T23:07:22Z  00:00:38
da5                       Linux       Succeeded  Manual      2018-09-17T23:06:33Z  00:00:31
da4       taskhelloworld  Linux       Succeeded  Git Commit  2018-09-17T23:03:45Z  00:00:44
da3       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T22:55:35Z  00:00:35
da2       taskhelloworld  Linux       Succeeded  Manual      2018-09-17T22:50:59Z  00:00:32
da1                       Linux       Succeeded  Manual      2018-09-17T22:29:59Z  00:00:57
```

Update the base image

Here you simulate a framework patch in the base image. Edit **Dockerfile-base**, and add an "a" after the version number defined in `NODE_VERSION`:

```
ENV NODE_VERSION 9.11.2a
```

Run a quick task to build the modified base image. Take note of the **Run ID** in the output.

```
az acr build --registry $BASE_ACR --image baseimages/node:9-alpine --file Dockerfile-base .
```

Once the build is complete and the ACR task has pushed the new base image to your registry, it triggers a build of the application image. It may take few moments for the task you created earlier to trigger the application image build, as it must detect the newly built and pushed base image.

List updated build

Now that you've updated the base image, list your task runs again to compare to the earlier list. If at first the output doesn't differ, periodically run the command to see the new task run appear in the list.

```
az acr task list-runs --registry $ACR_NAME --output table
```

Output is similar to the following. The TRIGGER for the last-executed build should be "Image Update", indicating that the task was kicked off by your quick task of the base image.

```
$ az acr task list-runs --registry $ACR_NAME --output table

Run ID    TASK            PLATFORM    STATUS     TRIGGER       STARTED               DURATION
--------  --------------  ----------  ---------  ------------  --------------------  ----------
da8       taskhelloworld  Linux       Succeeded  Image Update  2018-09-17T23:11:50Z  00:00:33
da7                       Linux       Succeeded  Manual        2018-09-17T23:11:27Z  00:00:35
da6       taskhelloworld  Linux       Succeeded  Manual        2018-09-17T23:07:22Z  00:00:38
da5                       Linux       Succeeded  Manual        2018-09-17T23:06:33Z  00:00:31
da4       taskhelloworld  Linux       Succeeded  Git Commit    2018-09-17T23:03:45Z  00:00:44
da3       taskhelloworld  Linux       Succeeded  Manual        2018-09-17T22:55:35Z  00:00:35
da2       taskhelloworld  Linux       Succeeded  Manual        2018-09-17T22:50:59Z  00:00:32
da1                       Linux       Succeeded  Manual        2018-09-17T22:29:59Z  00:00:57
```

If you'd like to perform the following optional step of running the newly built container to see the updated version number, take note of the **RUN ID** value for the Image Update-triggered build (in the preceding output, it's "da8").

Optional: Run newly built image

If you're working locally (not in the Cloud Shell), and you have Docker installed, run the new application image once its build has completed. Replace `<run-id>` with the RUN ID you obtained in the previous step. If you're using the Cloud Shell, skip this section (Cloud Shell does not support `docker run`).

```
docker run -d -p 8081:80 --name updatedapp --rm $ACR_NAME.azurecr.io/helloworld:<run-id>
```

Navigate to http://localhost:8081 in your browser, and you should see the updated Node.js version number (with the "a") in the web page:

What's important to note is that you updated your **base** image with a new version number, but the last-built **application** image displays the new version. ACR Tasks picked up your change to the base image, and rebuilt your application image automatically.

To stop and remove the container, run the following command:

```
docker stop updatedapp
```

Next steps

In this tutorial, you learned how to use a task to automatically trigger container image builds when the image's base image has been updated. Now, move on to the next tutorial to learn how to trigger tasks on a defined schedule.

Run a task on a schedule

Run an ACR task on a defined schedule

2/25/2020 • 6 minutes to read • Edit Online

This tutorial shows you how to run an ACR Task on a schedule. Schedule a task by setting up one or more *timer triggers*. Timer triggers can be used alone, or in combination with other task triggers.

In this tutorial, learn about scheduling tasks and:

- Create a task with a timer trigger
- Manage timer triggers

Scheduling a task is useful for scenarios like the following:

- Run a container workload for scheduled maintenance operations. For example, run a containerized app to remove unneeded images from your registry.
- Run a set of tests on a production image during the workday as part of your live-site monitoring.

You can use the Azure Cloud Shell or a local installation of the Azure CLI to run the examples in this article. If you'd like to use it locally, version 2.0.68 or later is required. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

About scheduling a task

- **Trigger with cron expression** - The timer trigger for a task uses a *cron expression*. The expression is a string with five fields specifying the minute, hour, day, month, and day of week to trigger the task. Frequencies of up to once per minute are supported.

  For example, the expression `"0 12 * * Mon-Fri"` triggers a task at noon UTC on each weekday. See details later in this article.

- **Multiple timer triggers** - Adding multiple timers to a task is allowed, as long as the schedules differ.
  - Specify multiple timer triggers when you create the task, or add them later.
  - Optionally name the triggers for easier management, or ACR Tasks will provide default trigger names.
  - If timer schedules overlap at a time, ACR Tasks triggers the task at the scheduled time for each timer.
- **Other task triggers** - In a timer-triggered task, you can also enable triggers based on source code commit or base image updates. Like other ACR tasks, you can also manually trigger a scheduled task.

Create a task with a timer trigger

When you create a task with the az acr task create command, you can optionally add a timer trigger. Add the `--schedule` parameter and pass a cron expression for the timer.

As a simple example, the following command triggers running the `hello-world` image from Docker Hub every day at 21:00 UTC. The task runs without a source code context.

```
az acr task create \
  --name mytask \
  --registry myregistry \
  --cmd hello-world \
  --schedule "0 21 * * *" \
  --context /dev/null
```

Run the az acr task show command to see that the timer trigger is configured. By default, the base image update trigger is also enabled.

```
$ az acr task show --name mytask --registry registry --output table
NAME      PLATFORM    STATUS    SOURCE REPOSITORY       TRIGGERS
--------  ----------  --------  -------------------     -----------------
mytask    linux       Enabled                           BASE_IMAGE, TIMER
```

Trigger the task manually with az acr task run to ensure that it is set up properly:

```
az acr task run --name mytask --registry myregistry
```

If the container runs successfully, the output is similar to the following:

```
Queued a run with ID: cf2a
Waiting for an agent...
2019/06/28 21:03:36 Using acb_vol_2ca23c46-a9ac-4224-b0c6-9fde44eb42d2 as the home volume
2019/06/28 21:03:36 Creating Docker network: acb_default_network, driver: 'bridge'
[...]
2019/06/28 21:03:38 Launching container with name: acb_step_0

Hello from Docker!
This message shows that your installation appears to be working correctly.
[...]
```

After the scheduled time, run the az acr task list-runs command to verify that the timer triggered the task as expected:

```
az acr task list-runs --name mytask --registry myregistry --output table
```

When the timer is successful, output is similar to the following:

```
RUN ID    TASK     PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  -------- ----------  ---------  ---------  --------------------  ----------
[...]
cf2b      mytask   linux       Succeeded  Timer      2019-06-28T21:00:23Z  00:00:06
cf2a      mytask   linux       Succeeded  Manual     2019-06-28T20:53:23Z  00:00:06
```

Manage timer triggers

Use the az acr task timer commands to manage the timer triggers for an ACR task.

Add or update a timer trigger

After a task is created, optionally add a timer trigger by using the az acr task timer add command. The following example adds a timer trigger name *timer2* to *mytask* created previously. This timer triggers the task every day at 10:30 UTC.

```
az acr task timer add \
  --name mytask \
  --registry myregistry \
  --timer-name timer2 \
  --schedule "30 10 * * *"
```

Update the schedule of an existing trigger, or change its status, by using the az acr task timer update command.

For example, update the trigger named *timer2* to trigger the task at 11:30 UTC:

```
az acr task timer update \
  --name mytask \
  --registry myregistry \
  --timer-name timer2 \
  --schedule "30 11 * * *"
```

List timer triggers

The az acr task timer list command shows the timer triggers set up for a task:

```
az acr task timer list --name mytask --registry myregistry
```

Example output:

```
[
  {
    "name": "timer2",
    "schedule": "30 11 * * *",
    "status": "Enabled"
  },
  {
    "name": "t1",
    "schedule": "0 21 * * *",
    "status": "Enabled"
  }
]
```

Remove a timer trigger

Use the az acr task timer remove command to remove a timer trigger from a task. The following example removes the *timer2* trigger from *mytask*:

```
az acr task timer remove \
  --name mytask \
  --registry myregistry \
  --timer-name timer2
```

Cron expressions

ACR Tasks uses the NCronTab library to interpret cron expressions. Supported expressions in ACR Tasks have five required fields separated by white space:

`{minute} {hour} {day} {month} {day-of-week}`

The time zone used with the cron expressions is Coordinated Universal Time (UTC). Hours are in 24-hour format.

> **NOTE**
> ACR Tasks does not support the `{second}` or `{year}` field in cron expressions. If you copy a cron expression used in another system, be sure to remove those fields, if they are used.

Each field can have one of the following types of values:

TYPE	EXAMPLE	WHEN TRIGGERED
A specific value	`"5 * * * *"`	every hour at 5 minutes past the hour
All values (`*`)	`"* 5 * * *"`	every minute of the hour beginning 5:00 UTC (60 times a day)
A range (`-` operator)	`"0 1-3 * * *"`	3 times per day, at 1:00, 2:00, and 3:00 UTC
A set of values (`,` operator)	`"20,30,40 * * * *"`	3 times per hour, at 20 minutes, 30 minutes, and 40 minutes past the hour
An interval value (`/` operator)	`"*/10 * * * *"`	6 times per hour, at 10 minutes, 20 minutes, and so on, past the hour

To specify months or days you can use numeric values, names, or abbreviations of names:

- For days, the numeric values are 0 to 6 where 0 starts with Sunday.
- Names are in English. For example: `Monday`, `January`.
- Names are case-insensitive.
- Names can be abbreviated. Three letters is the recommended abbreviation length. For example: `Mon`, `Jan`.

Cron examples

EXAMPLE	WHEN TRIGGERED
`"*/5 * * * *"`	once every five minutes
`"0 * * * *"`	once at the top of every hour
`"0 */2 * * *"`	once every two hours
`"0 9-17 * * *"`	once every hour from 9:00 to 17:00 UTC
`"30 9 * * *"`	at 9:30 UTC every day
`"30 9 * * 1-5"`	at 9:30 UTC every weekday
`"30 9 * Jan Mon"`	at 9:30 UTC every Monday in January

Clean up resources

To remove all resources you've created in this tutorial series, including the container registry or registries, container instance, key vault, and service principal, issue the following commands:

```
az group delete --resource-group $RES_GROUP
az ad sp delete --id http://$ACR_NAME-pull
```

Next steps

In this tutorial, you learned how to create Azure Container Registry tasks that are automatically triggered by a

timer.

For an example of using a scheduled task to clean up repositories in a registry, see Automatically purge images from an Azure container registry.

For examples of tasks triggered by source code commits or base image updates, see other articles in the ACR Tasks tutorial series.

ACR Tasks samples

11/24/2019 • 2 minutes to read • Edit Online

This article links to example `task.yaml` files and associated Dockerfiles for several Azure Container Registry Tasks (ACR Tasks) scenarios.

For additional examples, see the Azure samples repo.

Scenarios

- **Build image** - YAML, Dockerfile
- **Run container** - YAML
- **Build and push image** - YAML, Dockerfile
- **Build and run image** - YAML, Dockerfile
- **Build and push multiple images** - YAML, Dockerfile
- **Build and test images in parallel** - YAML, Dockerfile
- **Build and push images to multiple registries** - YAML, Dockerfile

Next steps

Learn more about ACR Tasks:

- Multi-step tasks - ACR Task-based workflows for building, testing, and patching container images in the cloud.
- Task reference - Task step types, their properties, and usage.
- Cmd repo - A collection of containers as commands for ACR Tasks.

Run multi-step build, test, and patch tasks in ACR Tasks

1/14/2020 • 5 minutes to read • Edit Online

Multi-step tasks extend the single image build-and-push capability of ACR Tasks with multi-step, multi-container-based workflows. Use multi-step tasks to build and push several images, in series or in parallel. Then run those images as commands within a single task run. Each step defines a container image build or push operation, and can also define the execution of a container. Each step in a multi-step task uses a container as its execution environment.

> **IMPORTANT**
> If you previously created tasks during the preview with the `az acr build-task` command, those tasks need to be re-created using the az acr task command.

For example, you can run a task with steps that automate the following logic:

1. Build a web application image
2. Run the web application container
3. Build a web application test image
4. Run the web application test container which performs tests against the running application container
5. If the tests pass, build a Helm chart archive package
6. Perform a `helm upgrade` using the new Helm chart archive package

All steps are performed within Azure, offloading the work to Azure's compute resources and freeing you from infrastructure management. Besides your Azure container registry, you pay only for the resources you use. For information on pricing, see the **Container Build** section in Azure Container Registry pricing.

Common task scenarios

Multi-step tasks enable scenarios like the following logic:

- Build, tag, and push one or more container images, in series or in parallel.
- Run and capture unit test and code coverage results.
- Run and capture functional tests. ACR Tasks supports running more than one container, executing a series of requests between them.
- Perform task-based execution, including pre/post steps of a container image build.
- Deploy one or more containers with your favorite deployment engine to your target environment.

Multi-step task definition

A multi-step task in ACR Tasks is defined as a series of steps within a YAML file. Each step can specify dependencies on the successful completion of one or more previous steps. The following task step types are available:

- `build`: Build one or more container images using familiar `docker build` syntax, in series or in parallel.
- `push`: Push built images to a container registry. Private registries like Azure Container Registry are supported, as is the public Docker Hub.

- `cmd`: Run a container, such that it can operate as a function within the context of the running task. You can pass parameters to the container's `[ENTRYPOINT]`, and specify properties like env, detach, and other familiar `docker run` parameters. The `cmd` step type enables unit and functional testing, with concurrent container execution.

The following snippets show how to combine these task step types. Multi-step tasks can be as simple as building a single image from a Dockerfile and pushing to your registry, with a YAML file similar to:

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world:$ID .
  - push: ["$Registry/hello-world:$ID"]
```

Or more complex, such as this fictitious multi-step definition which includes steps for build, test, helm package, and helm deploy (container registry and Helm repository configuration not shown):

```
version: v1.1.0
steps:
  - id: build-web
    build: -t $Registry/hello-world:$ID .
    when: ["-"]
  - id: build-tests
    build -t $Registry/hello-world-tests ./funcTests
    when: ["-"]
  - id: push
    push: ["$Registry/helloworld:$ID"]
    when: ["build-web", "build-tests"]
  - id: hello-world-web
    cmd: $Registry/helloworld:$ID
  - id: funcTests
    cmd: $Registry/helloworld:$ID
    env: ["host=helloworld:80"]
  - cmd: $Registry/functions/helm package --app-version $ID -d ./helm ./helm/helloworld/
  - cmd: $Registry/functions/helm upgrade helloworld ./helm/helloworld/ --reuse-values --set
helloworld.image=$Registry/helloworld:$ID
```

See task examples for multi-step task YAML files and Dockerfiles for several scenarios.

Run a sample task

Tasks support both manual execution, called a "quick run," and automated execution on Git commit or base image update.

To run a task, you first define the task's steps in a YAML file, then execute the Azure CLI command az acr run.

Here's an example Azure CLI command that runs a task using a sample task YAML file. Its steps build and then push an image. Update `\<acrName\>` with the name of your own Azure container registry before running the command.

```
az acr run --registry <acrName> -f build-push-hello-world.yaml https://github.com/Azure-Samples/acr-
tasks.git
```

When you run the task, the output should show the progress of each step defined in the YAML file. In the following output, the steps appear as `acb_step_0` and `acb_step_1`.

```
$ az acr run --registry myregistry -f build-push-hello-world.yaml https://github.com/Azure-Samples/acr-
tasks.git
Sending context to registry: myregistry...
Queued a run with ID: yd14
Waiting for an agent...
2018/09/12 20:08:44 Using acb_vol_0467fe58-f6ab-4dbd-a022-1bb487366941 as the home volume
2018/09/12 20:08:44 Creating Docker network: acb_default_network
2018/09/12 20:08:44 Successfully set up Docker network: acb_default_network
2018/09/12 20:08:44 Setting up Docker configuration...
2018/09/12 20:08:45 Successfully set up Docker configuration
2018/09/12 20:08:45 Logging in to registry: myregistry.azurecr-test.io
2018/09/12 20:08:46 Successfully logged in
2018/09/12 20:08:46 Executing step: acb_step_0
2018/09/12 20:08:46 Obtaining source code and scanning for dependencies...
2018/09/12 20:08:47 Successfully obtained source code and scanned for dependencies
Sending build context to Docker daemon  109.6kB
Step 1/1 : FROM hello-world
 ---> 4ab4c602aa5e
Successfully built 4ab4c602aa5e
Successfully tagged myregistry.azurecr-test.io/hello-world:yd14
2018/09/12 20:08:48 Executing step: acb_step_1
2018/09/12 20:08:48 Pushing image: myregistry.azurecr-test.io/hello-world:yd14, attempt 1
The push refers to repository [myregistry.azurecr-test.io/hello-world]
428c97da766c: Preparing
428c97da766c: Layer already exists
yd14: digest: sha256:1a6fd470b9ce10849be79e99529a88371dff60c60aab424c077007f6979b4812 size: 524
2018/09/12 20:08:55 Successfully pushed image: myregistry.azurecr-test.io/hello-world:yd14
2018/09/12 20:08:55 Step id: acb_step_0 marked as successful (elapsed time in seconds: 2.035049)
2018/09/12 20:08:55 Populating digests for step id: acb_step_0...
2018/09/12 20:08:57 Successfully populated digests for step id: acb_step_0
2018/09/12 20:08:57 Step id: acb_step_1 marked as successful (elapsed time in seconds: 6.832391)
The following dependencies were found:
- image:
    registry: myregistry.azurecr-test.io
    repository: hello-world
    tag: yd14
    digest: sha256:1a6fd470b9ce10849be79e99529a88371dff60c60aab424c077007f6979b4812
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/hello-world
    tag: latest
    digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
  git: {}


Run ID: yd14 was successful after 19s
```

For more information about automated builds on Git commit or base image update, see the Automate image builds and Base image update builds tutorial articles.

Next steps

You can find multi-step task reference and examples here:

- Task reference - Task step types, their properties, and usage.
- Task examples - Example `task.yaml` and Docker files for several scenarios, simple to complex.
- Cmd repo - A collection of containers as commands for ACR tasks.

About base image updates for ACR Tasks

2/25/2020 • 3 minutes to read • Edit Online

This article provides background information about updates to an application's base image and how these updates can trigger an Azure Container Registry task.

What are base images?

Dockerfiles defining most container images specify a parent image from which the image is based, often referred to as its *base image*. Base images typically contain the operating system, for example Alpine Linux or Windows Nano Server, on which the rest of the container's layers are applied. They might also include application frameworks such as Node.js or .NET Core. These base images are themselves typically based on public upstream images. Several of your application images might share a common base image.

A base image is often updated by the image maintainer to include new features or improvements to the OS or framework in the image. Security patches are another common cause for a base image update. When these upstream updates occur, you must also update your base images to include the critical fix. Each application image must then also be rebuilt to include these upstream fixes now included in your base image.

In some cases, such as a private development team, a base image might specify more than OS or framework. For example, a base image could be a shared service component image that needs to be tracked. Members of a team might need to track this base image for testing, or need to regularly update the image when developing application images.

Track base image updates

ACR Tasks includes the ability to automatically build images for you when a container's base image is updated.

ACR Tasks dynamically discovers base image dependencies when it builds a container image. As a result, it can detect when an application image's base image is updated. With one preconfigured build task, ACR Tasks can automatically rebuild every application image that references the base image. With this automatic detection and rebuilding, ACR Tasks saves you the time and effort normally required to manually track and update each and every application image referencing your updated base image.

Base image locations

For image builds from a Dockerfile, an ACR task detects dependencies on base images in the following locations:

- The same Azure container registry where the task runs
- Another private Azure container registry in the same or a different region
- A public repo in Docker Hub
- A public repo in Microsoft Container Registry

If the base image specified in the `FROM` statement resides in one of these locations, the ACR task adds a hook to ensure the image is rebuilt anytime its base is updated.

Additional considerations

- **Base images for application images** - Currently, an ACR task only tracks base image updates for application (*runtime*) images. It doesn't track base image updates for intermediate (*buildtime*) images used in multi-stage Dockerfiles.

- **Enabled by default** - When you create an ACR task with the az acr task create command, by default the task is *enabled* for trigger by a base image update. That is, the `base-image-trigger-enabled` property is set to True. If you want to disable this behavior in a task, update the property to False. For example, run the following az acr task update command:

  ```
  az acr task update --myregistry --name mytask --base-image-trigger-enabled False
  ```

- **Trigger to track dependencies** - To enable an ACR task to determine and track a container image's dependencies -- which include its base image -- you must first trigger the task to build the image **at least once**. For example, trigger the task manually using the az acr task run command.

- **Stable tag for base image** - To trigger a task on base image update, the base image must have a *stable* tag, such as `node:9-alpine`. This tagging is typical for a base image that is updated with OS and framework patches to a latest stable release. If the base image is updated with a new version tag, it does not trigger a task. For more information about image tagging, see the best practices guidance.

- **Other task triggers** - In a task triggered by base image updates, you can also enable triggers based on source code commit or a schedule. A base image update can also trigger a multi-step task.

Next steps

See the following tutorials for scenarios to automate application image builds after a base image is updated:

- Automate container image builds when a base image is updated in the same registry
- Automate container image builds when a base image is updated in a different registry

Use an Azure-managed identity in ACR Tasks

2/9/2020 • 5 minutes to read • Edit Online

Enable a managed identity for Azure resources in an ACR task, so the task can access other Azure resources, without needing to provide or manage credentials. For example, use a managed identity to enable a task step to pull or push container images to another registry.

In this article, you learn how to use the Azure CLI to enable a user-assigned or system-assigned managed identity on an ACR task. You can use the Azure Cloud Shell or a local installation of the Azure CLI. If you'd like to use it locally, version 2.0.68 or later is required. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

For illustration purposes, the example commands in this article use az acr task create to create a basic image build task that enables a managed identity. For sample scenarios to access secured resources from an ACR task using a managed identity, see:

- Cross-registry authentication
- Access external resources with secrets stored in Azure Key Vault

Why use a managed identity?

A managed identity for Azure resources provides selected Azure services with an automatically managed identity in Azure Active Directory. You can configure an ACR task with a managed identity so that the task can access other secured Azure resources, without passing credentials in the task steps.

Managed identities are of two types:

- *User-assigned identities*, which you can assign to multiple resources and persist for as long as you want. User-assigned identities are currently in preview.
- A *system-assigned identity*, which is unique to a specific resource such as an ACR task and lasts for the lifetime of that resource.

You can enable either or both types of identity in an ACR task. Grant the identity access to another resource, just like any security principal. When the task runs, it uses the identity to access the resource in any task steps that require access.

Steps to use a managed identity

Follow these high-level steps to use a managed identity with an ACR task.

1. (Optional) Create a user-assigned identity

If you plan to use a user-assigned identity, use an existing identity, or create the identity using the Azure CLI or other Azure tools. For example, use the az identity create command.

If you plan to use only a system-assigned identity, skip this step. You create a system-assigned identity when you create the ACR task.

2. Enable identity on an ACR task

When you create an ACR task, optionally enable a user-assigned identity, a system-assigned identity, or both. For example, pass the `--assign-identity` parameter when you run the az acr task create command in the Azure CLI.

To enable a system-assigned identity, pass `--assign-identity` with no value or `assign-identity [system]`. The

following example command creates a Linux task from a public GitHub repository which builds the `hello-world` image and enables a system-assigned managed identity:

```
az acr task create \
    --image hello-world:{{.Run.ID}} \
    --name hello-world --registry MyRegistry \
    --context https://github.com/Azure-Samples/acr-build-helloworld-node.git \
    --file Dockerfile \
    --commit-trigger-enabled false \
    --assign-identity
```

To enable a user-assigned identity, pass `--assign-identity` with a value of the *resource ID* of the identity. The following example command creates a Linux task from a public GitHub repository which builds the `hello-world` image and enables a user-assigned managed identity:

```
az acr task create \
    --image hello-world:{{.Run.ID}} \
    --name hello-world --registry MyRegistry \
    --context https://github.com/Azure-Samples/acr-build-helloworld-node.git \
    --file Dockerfile \
    --commit-trigger-enabled false
    --assign-identity <resourceID>
```

You can get the resource ID of the identity by running the az identity show command. The resource ID for the ID *myUserAssignedIdentity* in resource group *myResourceGroup* is of the form:

```
"/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myUserA
ssignedIdentity"
```

3. Grant the identity permissions to access other Azure resources

Depending on the requirements of your task, grant the identity permissions to access other Azure resources. Examples include:

- Assign the managed identity a role with pull, push and pull, or other permissions to a target container registry in Azure. For a complete list of registry roles, see Azure Container Registry roles and permissions.
- Assign the managed identity a role to read secrets in an Azure key vault.

Use the Azure CLI or other Azure tools to manage role-based access to resources. For example, run the az role assignment create command to assign the identity a role to the resource.

The following example assigns a managed identity the permissions to pull from a container registry. The command specifies the *principal ID* of the task identity and the *resource ID* of the target registry.

```
az role assignment create \
  --assignee <principalID> \
  --scope <registryID> \
  --role acrpull
```

4. (Optional) Add credentials to the task

If your task needs credentials to pull or push images to another custom registry, or to access other resources, add credentials to the task. Run the az acr task credential add command to add credentials, and pass the `--use-identity` parameter to indicate that the identity can access the credentials.

For example, to add credentials for a system-assigned identity to authenticate with the Azure container registry

targetregistry, pass `use-identity [system]`:

```
az acr task credential add \
    --name helloworld \
    --registry myregistry \
    --login-server targetregistry.azurecr.io \
    --use-identity [system]
```

To add credentials for a user-assigned identity to authenticate with the registry *targetregistry*, pass `use-identity` with a value of the *client ID* of the identity. For example:

```
az acr task credential add \
    --name helloworld \
    --registry myregistry \
    --login-server targetregistry.azurecr.io \
    --use-identity <clientID>
```

You can get the client ID of the identity by running the az identity show command. The client ID is a GUID of the form `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.

5. Run the task

After configuring a task with a managed identity, run the task. For example, to test one of the tasks created in this article, manually trigger it using the az acr task run command. If you configured additional, automated task triggers, the task runs when automatically triggered.

Next steps

In this article, you learned how to enable and use a user-assigned or system-assigned managed identity on an ACR task. For scenarios to access secured resources from an ACR task using a managed identity, see:

- Cross-registry authentication
- Access external resources with secrets stored in Azure Key Vault

Cross-registry authentication in an ACR task using an Azure-managed identity

1/29/2020 • 7 minutes to read • Edit Online

In an ACR task, you can enable a managed identity for Azure resources. The task can use the identity to access other Azure resources, without needing to provide or manage credentials.

In this article, you learn how to enable a managed identity in a task to pull an image from a registry different from the one used to run the task.

To create the Azure resources, this article requires that you run the Azure CLI version 2.0.68 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Scenario overview

The example task pulls a base image from another Azure container registry to build and push an application image. To pull the base image, you configure the task with a managed identity and assign appropriate permissions to it.

This example shows steps using either a user-assigned or system-assigned managed identity. Your choice of identity depends on your organization's needs.

In a real-world scenario, an organization might maintain a set of base images used by all development teams to build their applications. These base images are stored in a corporate registry, with each development team having only pull rights.

Prerequisites

For this article, you need two Azure container registries:

- You use the first registry to create and execute ACR tasks. In this article, this registry is named *myregistry*.
- The second registry hosts a base image used for the task to build an image. In this article, the second registry is named *mybaseregistry*.

Replace with your own registry names in later steps.

If you don't already have the needed Azure container registries, see Quickstart: Create a private container registry using the Azure CLI. You don't need to push images to the registry yet.

Prepare base registry

First, create a working directory and then create a file named Dockerfile with the following content. This simple example builds a Node.js base image from a public image in Docker Hub.

```
echo FROM node:9-alpine > Dockerfile
```

In the current directory, run the az acr build command to build and push the base image to the base registry. In practice, another team or process in the organization might maintain the base registry.

```
az acr build --image baseimages/node:9-alpine --registry mybaseregistry --file Dockerfile .
```

Define task steps in YAML file

The steps for this example multi-step task are defined in a YAML file. Create a file named `helloworldtask.yaml` in your local working directory and paste the following contents. Update the value of `REGISTRY_NAME` in the build step with the server name of your base registry.

```
version: v1.1.0
steps:
# Replace mybaseregistry with the name of your registry containing the base image
  - build: -t $Registry/hello-world:$ID  https://github.com/Azure-Samples/acr-build-helloworld-node.git -f
Dockerfile-app --build-arg REGISTRY_NAME=mybaseregistry.azurecr.io
  - push: ["$Registry/hello-world:$ID"]
```

The build step uses the `Dockerfile-app` file in the Azure-Samples/acr-build-helloworld-node repo to build an image. The `--build-arg` references the base registry to pull the base image. When successfully built, the image is pushed to the registry used to run the task.

Option 1: Create task with user-assigned identity

The steps in this section create a task and enable a user-assigned identity. If you want to enable a system-assigned identity instead, see Option 2: Create task with system-assigned identity.

Create a user-assigned identity

Create an identity named *myACRTasksId* in your subscription using the az identity create command. You can use the same resource group you used previously to create a container registry, or a different one.

```
az identity create --resource-group myResourceGroup --name myACRTasksId
```

To configure the user-assigned identity in the following steps, use the az identity show command to store the identity's resource ID, principal ID, and client ID in variables.

```
# Get resource ID of the user-assigned identity
resourceID=$(az identity show --resource-group myResourceGroup --name myACRTasksId --query id --output tsv)

# Get principal ID of the task's user-assigned identity
principalID=$(az identity show --resource-group myResourceGroup --name myACRTasksId --query principalId --
output tsv)

# Get client ID of the user-assigned identity
clientID=$(az identity show --resource-group myResourceGroup --name myACRTasksId --query clientId --output
tsv)
```

Create task

Create the task *helloworldtask* by executing the following az acr task create command. The task runs without a source code context, and the command references the file `helloworldtask.yaml` in the working directory. The `--assign-identity` parameter passes the resource ID of the user-assigned identity.

```
az acr task create \
  --registry myregistry \
  --name helloworldtask \
  --context /dev/null \
  --file helloworldtask.yaml \
  --assign-identity $resourceID
```

In the command output, the `identity` section shows the identity of type `UserAssigned` is set in the task:

```
[...]
"identity": {
    "principalId": null,
    "tenantId": null,
    "type": "UserAssigned",
    "userAssignedIdentities": {
      "/subscriptions/xxxxxxxx-d12e-4760-9ab6-
xxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myACRTa
sksId": {
        "clientId": "xxxxxxxx-f17e-4768-bb4e-xxxxxxxxxxxx",
        "principalId": "xxxxxxxx-1335-433d-bb6c-xxxxxxxxxxxx"
      }
[...]
```

Option 2: Create task with system-assigned identity

The steps in this section create a task and enable a system-assigned identity. If you want to enable a user-assigned identity instead, see Option 1: Create task with user-assigned identity.

Create task

Create the task *helloworldtask* by executing the following az acr task create command. The task runs without a source code context, and the command references the file `helloworldtask.yaml` in the working directory. The `--assign-identity` parameter with no value enables the system-assigned identity on the task.

```
az acr task create \
  --registry myregistry \
  --name helloworldtask \
  --context /dev/null \
  --file helloworldtask.yaml \
  --assign-identity
```

In the command output, the `identity` section shows an identity of type `SystemAssigned` is set in the task. The `principalId` is the principal ID of the task identity:

```
[...]
  "identity": {
    "principalId": "xxxxxxxx-2703-42f9-97d0-xxxxxxxxxxxx",
    "tenantId": "xxxxxxxx-86f1-41af-91ab-xxxxxxxxxxxx",
    "type": "SystemAssigned",
    "userAssignedIdentities": null
  },
  "location": "eastus",
[...]
```

Use the az acr task show command to store the principalId in a variable, to use in later commands. Substitute the name of your task and your registry in the following command:

```
principalID=$(az acr task show --name mytask --registry myregistry --query identity.principalId --output tsv)
```

Give identity pull permissions to the base registry

In this section, give the managed identity permissions to pull from the base registry, *mybaseregistry*.

Use the az acr show command to get the resource ID of the base registry and store it in a variable:

```
baseregID=$(az acr show --name mybaseregistry --query id --output tsv)
```

Use the az role assignment create command to assign the identity the `acrpull` role to the base registry. This role has permissions only to pull images from the registry.

```
az role assignment create \
  --assignee $principalID \
  --scope $baseregID \
  --role acrpull
```

Add target registry credentials to task

Now use the az acr task credential add command to enable the task to authenticate with the base registry using the identity's credentials. Run the command corresponding to the type of managed identity you enabled in the task. If you enabled a user-assigned identity, pass `--use-identity` with the client ID of the identity. If you enabled a system-assigned identity, pass `--use-identity [system]`.

```
# Add credentials for user-assigned identity to the task
az acr task credential add \
  --name helloworldtask \
  --registry myregistry \
  --login-server mybaseregistry.azurecr.io \
  --use-identity $clientID

# Add credentials for system-assigned identity to the task
az acr task credential add \
  --name helloworldtask \
  --registry myregistry \
  --login-server mybaseregistry.azurecr.io \
  --use-identity [system]
```

Manually run the task

To verify that the task in which you enabled a managed identity runs successfully, manually trigger the task with the az acr task run command.

```
az acr task run \
  --name helloworldtask \
  --registry myregistry
```

If the task runs successfully, output is similar to:

```
Queued a run with ID: cf10
Waiting for an agent...
2019/06/14 22:47:32 Using acb_vol_dbfbe232-fd76-4ca3-bd4a-687e84cb4ce2 as the home volume
2019/06/14 22:47:39 Creating Docker network: acb_default_network, driver: 'bridge'
2019/06/14 22:47:40 Successfully set up Docker network: acb_default_network
2019/06/14 22:47:40 Setting up Docker configuration...
2019/06/14 22:47:41 Successfully set up Docker configuration
2019/06/14 22:47:41 Logging in to registry: myregistry.azurecr.io
2019/06/14 22:47:42 Successfully logged into myregistry.azurecr.io
2019/06/14 22:47:42 Logging in to registry: mybaseregistry.azurecr.io
2019/06/14 22:47:43 Successfully logged into mybaseregistry.azurecr.io
2019/06/14 22:47:43 Executing step ID: acb_step_0. Timeout(sec): 600, Working directory: '', Network:
'acb_default_network'
2019/06/14 22:47:43 Scanning for dependencies...
2019/06/14 22:47:45 Successfully scanned dependencies
2019/06/14 22:47:45 Launching container with name: acb_step_0
Sending build context to Docker daemon   25.6kB
Step 1/6 : ARG REGISTRY_NAME
Step 2/6 : FROM ${REGISTRY_NAME}/baseimages/node:9-alpine
9-alpine: Pulling from baseimages/node
[...]
Successfully built 41b49a112663
Successfully tagged myregistry.azurecr.io/hello-world:cf10
2019/06/14 22:47:56 Successfully executed container: acb_step_0
2019/06/14 22:47:56 Executing step ID: acb_step_1. Timeout(sec): 600, Working directory: '', Network:
'acb_default_network'
2019/06/14 22:47:56 Pushing image: myregistry.azurecr.io/hello-world:cf10, attempt 1
The push refers to repository [myregistry.azurecr.io/hello-world]
[...]
2019/06/14 22:48:00 Step ID: acb_step_1 marked as successful (elapsed time in seconds: 2.517011)
2019/06/14 22:48:00 The following dependencies were found:
2019/06/14 22:48:00
- image:
    registry: myregistry.azurecr.io
    repository: hello-world
    tag: cf10
    digest: sha256:611cf6e3ae3cb99b23fadcd89fa144e18aa1b1c9171ad4a0da4b62b31b4e38d1
  runtime-dependency:
    registry: mybaseregistry.azurecr.io
    repository: baseimages/node
    tag: 9-alpine
    digest: sha256:e8e92cffd464fce3be9a3eefd1b65dc9cbe2484da31c11e813a4effc6105c00f
  git:
    git-head-revision: 0f988779c97fe0bfc7f2f74b88531617f4421643

Run ID: cf10 was successful after 32s
```

Run the az acr repository show-tags command to verify that the image built and was successfully pushed to *myregistry*:

```
az acr repository show-tags --name myregistry --repository hello-world --output tsv
```

Example output:

```
cf10
```

Next steps

- Learn more about enabling a managed identity in an ACR task.
- See the ACR Tasks YAML reference

External authentication in an ACR task using an Azure-managed identity

1/29/2020 • 7 minutes to read • Edit Online

In an ACR task, you can enable a managed identity for Azure resources. The task can use the identity to access other Azure resources, without needing to provide or manage credentials.

In this article, you learn how to enable a managed identity in a task that accesses secrets stored in an Azure key vault.

To create the Azure resources, this article requires that you run the Azure CLI version 2.0.68 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Scenario overview

The example task reads Docker Hub credentials stored in an Azure key vault. The credentials are for a Docker Hub account with write (push) permissions to a private Docker Hub repository. To read the credentials, you configure the task with a managed identity and assign appropriate permissions to it. The task associated with the identity builds an image, and signs into Docker Hub to push the image to the private repo.

This example shows steps using either a user-assigned or system-assigned managed identity. Your choice of identity depends on your organization's needs.

In a real-world scenario, a company might publish images to a private repo in Docker Hub as part of a build process.

Prerequisites

You need an Azure container registry in which you run the task. In this article, this registry is named *myregistry*. Replace with your own registry name in later steps.

If you don't already have an Azure container registry, see Quickstart: Create a private container registry using the Azure CLI. You don't need to push images to the registry yet.

You also need a private repository in Docker Hub, and a Docker Hub account with permissions to write to the repo. In this example, this repo is named *hubuser/hubrepo*.

Create a key vault and store secrets

First, if you need to, create a resource group named *myResourceGroup* in the *eastus* location with the following az group create command:

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

Use the az keyvault create command to create a key vault. Be sure to specify a unique key vault name.

```
az keyvault create --name mykeyvault --resource-group myResourceGroup --location eastus
```

Store the required Docker Hub credentials in the key vault using the az keyvault secret set command. In these commands, the values are passed in environment variables:

```
# Store Docker Hub user name
az keyvault secret set \
  --name UserName \
  --value $USERNAME \
  --vault-name mykeyvault

# Store Docker Hub password
az keyvault secret set \
  --name Password \
  --value $PASSWORD \
  --vault-name mykeyvault
```

In a real-world scenario, secrets would likely be set and maintained in a separate process.

Define task steps in YAML file

The steps for this example task are defined in a YAML file. Create a file named `dockerhubtask.yaml` in a local working directory and paste the following contents. Be sure to replace the key vault name in the file with the name of your key vault.

```
version: v1.1.0
# Replace mykeyvault with the name of your key vault
secrets:
  - id: username
    keyvault: https://mykeyvault.vault.azure.net/secrets/UserName
  - id: password
    keyvault: https://mykeyvault.vault.azure.net/secrets/Password
steps:
# Log in to Docker Hub
  - cmd: bash echo '{{.Secrets.password}}' | docker login --username '{{.Secrets.username}}' --password-stdin
# Build image
  - build: -t {{.Values.PrivateRepo}}:$ID https://github.com/Azure-Samples/acr-tasks.git -f hello-
world.dockerfile
# Push image to private repo in Docker Hub
  - push:
    - {{.Values.PrivateRepo}}:$ID
```

The task steps do the following:

- Manage secret credentials to authenticate with Docker Hub.
- Authenticate with Docker Hub by passing the secrets to the `docker login` command.
- Build an image using a sample Dockerfile in the Azure-Samples/acr-tasks repo.
- Push the image to the private Docker Hub repository.

Option 1: Create task with user-assigned identity

The steps in this section create a task and enable a user-assigned identity. If you want to enable a system-assigned identity instead, see Option 2: Create task with system-assigned identity.

Create a user-assigned identity

Create an identity named *myACRTasksId* in your subscription using the az identity create command. You can use the same resource group you used previously to create a container registry, or a different one.

```
az identity create --resource-group myResourceGroup --name myACRTasksId
```

To configure the user-assigned identity in the following steps, use the az identity show command to store the identity's resource ID, principal ID, and client ID in variables.

```
# Get resource ID of the user-assigned identity
resourceID=$(az identity show --resource-group myResourceGroup --name myACRTasksId --query id --output tsv)

# Get principal ID of the task's user-assigned identity
principalID=$(az identity show --resource-group myResourceGroup --name myACRTasksId --query principalId --
output tsv)

# Get client ID of the user-assigned identity
clientID=$(az identity show --resource-group myResourceGroup --name myACRTasksId --query clientId --output
tsv)
```

Create task

Create the task *dockerhubtask* by executing the following az acr task create command. The task runs without a source code context, and the command references the file `dockerhubtask.yaml` in the working directory. The `--assign-identity` parameter passes the resource ID of the user-assigned identity.

```
az acr task create \
  --name dockerhubtask \
  --registry myregistry \
  --context /dev/null \
  --file dockerhubtask.yaml \
  --assign-identity $resourceID
```

In the command output, the `identity` section shows the identity of type `UserAssigned` is set in the task:

```
[...]
"identity": {
    "principalId": null,
    "tenantId": null,
    "type": "UserAssigned",
    "userAssignedIdentities": {
      "/subscriptions/xxxxxxxx-d12e-4760-9ab6-
xxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myACRTa
sksId": {
        "clientId": "xxxxxxxx-f17e-4768-bb4e-xxxxxxxxxxxx",
        "principalId": "xxxxxxxx-1335-433d-bb6c-xxxxxxxxxxxx"
      }
[...]
```

Option 2: Create task with system-assigned identity

The steps in this section create a task and enable a system-assigned identity. If you want to enable a user-assigned identity instead, see Option 1: Create task with user-assigned identity.

Create task

Create the task *dockerhubtask* by executing the following az acr task create command. The task runs without a source code context, and the command references the file `dockerhubtask.yaml` in the working directory. The `--assign-identity` parameter with no value enables the system-assigned identity on the task.

```
az acr task create \
  --name dockerhubtask \
  --registry myregistry \
  --context /dev/null \
  --file dockerhubtask.yaml \
  --assign-identity
```

In the command output, the `identity` section shows an identity of type `SystemAssigned` is set in the task. The

`principalId` is the principal ID of the task identity:

```
[...]
  "identity": {
    "principalId": "xxxxxxxx-2703-42f9-97d0-xxxxxxxxxxxx",
    "tenantId": "xxxxxxxx-86f1-41af-91ab-xxxxxxxxxxxx",
    "type": "SystemAssigned",
    "userAssignedIdentities": null
  },
  "location": "eastus",
[...]
```

Use the az acr task show command to store the principalId in a variable, to use in later commands. Substitute the name of your task and your registry in the following command:

```
principalID=$(az acr task show --name mytask --registry myregistry --query identity.principalId --output tsv)
```

Grant identity access to key vault

Run the following az keyvault set-policy command to set an access policy on the key vault. The following example allows the identity to read secrets from the key vault.

```
az keyvault set-policy --name mykeyvault \
  --resource-group myResourceGroup \
  --object-id $principalID \
  --secret-permissions get
```

Manually run the task

To verify that the task in which you enabled a managed identity runs successfully, manually trigger the task with the az acr task run command. The `--set` parameter is used to pass the private repo name to the task. In this example, the placeholder repo name is *hubuser/hubrepo*.

```
az acr task run --name dockerhubtask --registry myregistry --set PrivateRepo=hubuser/hubrepo
```

When the task runs successfully, output shows successful authentication to Docker Hub, and the image is successfully built and pushed to the private repo:

```
Queued a run with ID: cf24
Waiting for an agent...
2019/06/20 18:05:55 Using acb_vol_b1edae11-30de-4f2b-a9c7-7d743e811101 as the home volume
2019/06/20 18:05:58 Creating Docker network: acb_default_network, driver: 'bridge'
2019/06/20 18:05:58 Successfully set up Docker network: acb_default_network
2019/06/20 18:05:58 Setting up Docker configuration...
2019/06/20 18:05:59 Successfully set up Docker configuration
2019/06/20 18:05:59 Logging in to registry: myregistry.azurecr.io
2019/06/20 18:06:00 Successfully logged into myregistry.azurecr.io
2019/06/20 18:06:00 Executing step ID: acb_step_0. Timeout(sec): 600, Working directory: '', Network:
'acb_default_network'
2019/06/20 18:06:00 Launching container with name: acb_step_0
[...]
Login Succeeded
2019/06/20 18:06:02 Successfully executed container: acb_step_0
2019/06/20 18:06:02 Executing step ID: acb_step_1. Timeout(sec): 600, Working directory: '', Network:
'acb_default_network'
2019/06/20 18:06:02 Scanning for dependencies...
2019/06/20 18:06:04 Successfully scanned dependencies
2019/06/20 18:06:04 Launching container with name: acb_step_1
Sending build context to Docker daemon    129kB
[...]
2019/06/20 18:06:07 Successfully pushed image: hubuser/hubrepo:cf24
2019/06/20 18:06:07 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 2.064353)
2019/06/20 18:06:07 Step ID: acb_step_1 marked as successful (elapsed time in seconds: 2.594061)
2019/06/20 18:06:07 Populating digests for step ID: acb_step_1...
2019/06/20 18:06:09 Successfully populated digests for step ID: acb_step_1
2019/06/20 18:06:09 Step ID: acb_step_2 marked as successful (elapsed time in seconds: 2.743923)
2019/06/20 18:06:09 The following dependencies were found:
2019/06/20 18:06:09
- image:
    registry: registry.hub.docker.com
    repository: hubuser/hubrepo
    tag: cf24
    digest: sha256:92c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e899a
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/hello-world
    tag: latest
    digest: sha256:0e11c388b664df8a27a901dce21eb89f11d8292f7fca1b3e3c4321bf7897bffe
  git:
    git-head-revision: b0ffa6043dd893a4c75644c5fed384c82ebb5f9e

Run ID: cf24 was successful after 15s
```

To confirm the image is pushed, check for the tag (`cf24` in this example) in the private Docker Hub repo.

Next steps

- Learn more about enabling a managed identity in an ACR task.
- See the ACR Tasks YAML reference

Build and push an image from an app using a Cloud Native Buildpack

11/24/2019 • 3 minutes to read • Edit Online

The Azure CLI command `az acr pack build` uses the `pack` CLI tool, from Buildpacks, to build an app and push its image into an Azure container registry. This feature provides an option to quickly build a container image from your application source code in Node.js, Java, and other languages without having to define a Dockerfile.

You can use the Azure Cloud Shell or a local installation of the Azure CLI to run the examples in this article. If you'd like to use it locally, version 2.0.70 or later is required. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

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

Use the build command

To build and push a container image using Cloud Native Buildpacks, run the az acr pack build command. Whereas the az acr build command builds and pushes an image from a Dockerfile source and related code, with `az acr pack build` you specify an application source tree directly.

At a minimum, specify the following when you run `az acr pack build`:

- An Azure container registry where you run the command
- An image name and tag for the resulting image
- One of the supported context locations for ACR Tasks, such as a local directory, a GitHub repo, or a remote tarball
- The name of a Buildpack builder image suitable for your application. Azure Container Registry caches builder images such as `cloudfoundry/cnb:0.0.34-cflinuxfs3` for faster builds.

`az acr pack build` supports other features of ACR Tasks commands including run variables and task run logs that are streamed and also saved for later retrieval.

Example: Build Node.js image with Cloud Foundry builder

The following example builds a container image from a Node.js app in the Azure-Samples/nodejs-docs-hello-world repo, using the `cloudfoundry/cnb:0.0.34-cflinuxfs3` builder. This builder is cached by Azure Container Registry, so a `--pull` parameter isn't required:

```
az acr pack build \
    --registry myregistry \
    --image {{.Run.Registry}}/node-app:1.0 \
    --builder cloudfoundry/cnb:0.0.34-cflinuxfs3 \
    https://github.com/Azure-Samples/nodejs-docs-hello-world.git
```

This example builds the `node-app` image with the `1.0` tag and pushes it to the *myregistry* container registry. In this example, the target registry name is explicitly prepended to the image name. If not specified, the registry login

server name is automatically prepended to the image name.

Command output shows the progress of building and pushing the image.

After the image is successfully built, you can run it with Docker, if you have it installed. First sign into your registry:

```
az acr login --name myregistry
```

Run the image:

```
docker run --rm -p 1337:1337 myregistry.azurecr.io/node-app:1.0
```

Browse to `localhost:1337` in your favorite browser to see the sample web app. Press `[Ctrl]+[C]` to stop the container.

Example: Build Java image with Heroku builder

The following example builds a container image from the Java app in the buildpack/sample-java-app repo, using the `heroku/buildpacks:18` builder. The `--pull` parameter specifies that the command should pull the latest builder image.

```
az acr pack build \
    --registry myregistry \
    --image java-app:{{.Run.ID}} \
    --pull --builder heroku/buildpacks:18 \
    https://github.com/buildpack/sample-java-app.git
```

This example builds the `java-app` image tagged with the run ID of the command and pushes it to the *myregistry* container registry.

Command output shows the progress of building and pushing the image.

After the image is successfully built, you can run it with Docker, if you have it installed. First sign into your registry:

```
az acr login --name myregistry
```

Run the image, substituting your image tag for *runid*:

```
docker run --rm -p 8080:8080 myregistry.azurecr.io/java-app:runid
```

Browse to `localhost:8080` in your favorite browser to see the sample web app. Press `[Ctrl]+[C]` to stop the container.

Next steps

After you build and push a container image with `az acr pack build`, you can deploy it like any image to a target of your choice. Azure deployment options include running it in App Service or Azure Kubernetes Service, among others.

For more information about ACR Tasks features, see Automate container image builds and maintenance with ACR Tasks.

ACR Tasks reference: YAML

1/14/2020 • 17 minutes to read • Edit Online

Multi-step task definition in ACR Tasks provides a container-centric compute primitive focused on building, testing, and patching containers. This article covers the commands, parameters, properties, and syntax for the YAML files that define your multi-step tasks.

This article contains reference for creating multi-step task YAML files for ACR Tasks. If you'd like an introduction to ACR Tasks, see the ACR Tasks overview.

acr-task.yaml file format

ACR Tasks supports multi-step task declaration in standard YAML syntax. You define a task's steps in a YAML file. You can then run the task manually by passing the file to the az acr run command. Or, use the file to create a task with az acr task create that's triggered automatically on a Git commit or base image update. Although this article refers to `acr-task.yaml` as the file containing the steps, ACR Tasks supports any valid filename with a supported extension.

The top-level `acr-task.yaml` primitives are **task properties**, **step types**, and **step properties**:

- Task properties apply to all steps throughout task execution. There are several global task properties, including:
  - `version`
  - `stepTimeout`
  - `workingDirectory`
- Task step types represent the types of actions that can be performed in a task. There are three step types:
  - `build`
  - `push`
  - `cmd`
- Task step properties are parameters that apply to an individual step. There are several step properties, including:
  - `startDelay`
  - `timeout`
  - `when`
  - ...and many more.

The base format of an `acr-task.yaml` file, including some common step properties, follows. While not an exhaustive representation of all available step properties or step type usage, it provides a quick overview of the basic file format.

```
version: # acr-task.yaml format version.
stepTimeout: # Seconds each step may take.
steps: # A collection of image or container actions.
  - build: # Equivalent to "docker build," but in a multi-tenant environment
  - push: # Push a newly built or retagged image to a registry.
    when: # Step property that defines either parallel or dependent step execution.
  - cmd: # Executes a container, supports specifying an [ENTRYPOINT] and parameters.
    startDelay: # Step property that specifies the number of seconds to wait before starting execution.
```

Supported task filename extensions

ACR Tasks has reserved several filename extensions, including `.yaml`, that it will process as a task file. Any extension *not* in the following list is considered by ACR Tasks to be a Dockerfile: .yaml, .yml, .toml, .json, .sh, .bash, .zsh, .ps1, .ps, .cmd, .bat, .ts, .js, .php, .py, .rb, .lua

YAML is the only file format currently supported by ACR Tasks. The other filename extensions are reserved for possible future support.

Run the sample tasks

There are several sample task files referenced in the following sections of this article. The sample tasks are in a public GitHub repository, Azure-Samples/acr-tasks. You can run them with the Azure CLI command az acr run. The sample commands are similar to:

```
az acr run -f build-push-hello-world.yaml https://github.com/Azure-Samples/acr-tasks.git
```

The formatting of the sample commands assumes you've configured a default registry in the Azure CLI, so they omit the `--registry` parameter. To configure a default registry, use the az configure command with the `--defaults` parameter, which accepts an `acr=REGISTRY_NAME` value.

For example, to configure the Azure CLI with a default registry named "myregistry":

```
az configure --defaults acr=myregistry
```

Task properties

Task properties typically appear at the top of an `acr-task.yaml` file, and are global properties that apply throughout the full execution of the task steps. Some of these global properties can be overridden within an individual step.

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	OVERRIDE SUPPORTED	DEFAULT VALUE
`version`	string	Yes	The version of the `acr-task.yaml` file as parsed by the ACR Tasks service. While ACR Tasks strives to maintain backward compatibility, this value allows ACR Tasks to maintain compatibility within a defined version. If unspecified, defaults to the latest version.	No	None

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	OVERRIDE SUPPORTED	DEFAULT VALUE
`stepTimeout`	int (seconds)	Yes	The maximum number of seconds a step can run. If the property is specified on a task, it sets the default `timeout` property of all the steps. If the `timeout` property is specified on a step, it overrides the property provided by the task.	Yes	600 (10 minutes)
`workingDirectory`	string	Yes	The working directory of the container during runtime. If the property is specified on a task, it sets the default `workingDirectory` property of all the steps. If specified on a step, it overrides the property provided by the task.	Yes	`/workspace`
`env`	[string, string, ...]	Yes	Array of strings in `key=value` format that define the environment variables for the task. If the property is specified on a task, it sets the default `env` property of all the steps. If specified on a step, it overrides any environment variables inherited from the task.	None	
`secrets`	[secret, secret, ...]	Yes	Array of secret objects.	None	

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	OVERRIDE SUPPORTED	DEFAULT VALUE
`networks`	[network, network, ...]	Yes	Array of network objects.	None	

secret

The secret object has the following properties.

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`id`	string	No	The identifier of the secret.	None
`keyvault`	string	Yes	The Azure Key Vault Secret URL.	None
`clientID`	string	Yes	The client ID of the user-assigned managed identity for Azure resources.	None

network

The network object has the following properties.

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`name`	string	No	The name of the network.	None
`driver`	string	Yes	The driver to manage the network.	None
`ipv6`	bool	Yes	Whether IPv6 networking is enabled.	`false`
`skipCreation`	bool	Yes	Whether to skip network creation.	`false`
`isDefault`	bool	Yes	Whether the network is a default network provided with Azure Container Registry	`false`

Task step types

ACR Tasks supports three step types. Each step type supports several properties, detailed in the section for each step type.

STEP TYPE	DESCRIPTION
`build`	Builds a container image using familiar `docker build` syntax.

STEP TYPE	DESCRIPTION
`push`	Executes a `docker push` of newly built or retagged images to a container registry. Azure Container Registry, other private registries, and the public Docker Hub are supported.
`cmd`	Runs a container as a command, with parameters passed to the container's `[ENTRYPOINT]`. The `cmd` step type supports parameters like `env`, `detach`, and other familiar `docker run` command options, enabling unit and functional testing with concurrent container execution.

build

Build a container image. The `build` step type represents a multi-tenant, secure means of running `docker build` in the cloud as a first-class primitive.

Syntax: build

```
version: v1.1.0
steps:
  - [build]: -t [imageName]:[tag] -f [Dockerfile] [context]
    [property]: [value]
```

The `build` step type supports the parameters in the following table. The `build` step type also supports all build options of the docker build command, such as `--build-arg` to set build-time variables.

PARAMETER	DESCRIPTION	OPTIONAL
`-t` | `--image`	Defines the fully qualified `image:tag` of the built image. As images may be used for inner task validations, such as functional tests, not all images require `push` to a registry. However, to instance an image within a Task execution, the image does need a name to reference. Unlike `az acr build`, running ACR Tasks doesn't provide default push behavior. With ACR Tasks, the default scenario assumes the ability to build, validate, then push an image. See push for how to optionally push built images.	Yes
`-f` | `--file`	Specifies the Dockerfile passed to `docker build`. If not specified, the default Dockerfile in the root of the context is assumed. To specify a Dockerfile, pass the filename relative to the root of the context.	Yes

PARAMETER	DESCRIPTION	OPTIONAL
`context`	The root directory passed to `docker build`. The root directory of each task is set to a shared workingDirectory, and includes the root of the associated Git cloned directory.	No

Properties: build

The `build` step type supports the following properties. Find details of these properties in the Task step properties section of this article.

`detach`	bool	Optional
`disableWorkingDirectoryOverride`	bool	Optional
`entryPoint`	string	Optional
`env`	[string, string, ...]	Optional
`expose`	[string, string, ...]	Optional
`id`	string	Optional
`ignoreErrors`	bool	Optional
`isolation`	string	Optional
`keep`	bool	Optional
`network`	object	Optional
`ports`	[string, string, ...]	Optional
`pull`	bool	Optional
`repeat`	int	Optional
`retries`	int	Optional
`retryDelay`	int (seconds)	Optional
`secret`	object	Optional
`startDelay`	int (seconds)	Optional
`timeout`	int (seconds)	Optional
`when`	[string, string, ...]	Optional

`workingDirectory`	string	Optional

Examples: build

Build image - context in root

```
az acr run -f build-hello-world.yaml https://github.com/AzureCR/acr-tasks-sample.git
```

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world -f hello-world.dockerfile .
```

Build image - context in subdirectory

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world -f hello-world.dockerfile ./subDirectory
```

push

Push one or more built or retagged images to a container registry. Supports pushing to private registries like Azure Container Registry, or to the public Docker Hub.

Syntax: push

The `push` step type supports a collection of images. YAML collection syntax supports inline and nested formats. Pushing a single image is typically represented using inline syntax:

```
version: v1.1.0
steps:
  # Inline YAML collection syntax
  - push: ["$Registry/hello-world:$ID"]
```

For increased readability, use nested syntax when pushing multiple images:

```
version: v1.1.0
steps:
  # Nested YAML collection syntax
  - push:
    - $Registry/hello-world:$ID
    - $Registry/hello-world:latest
```

Properties: push

The `push` step type supports the following properties. Find details of these properties in the Task step properties section of this article.

`env`	[string, string, ...]	Optional
`id`	string	Optional
`ignoreErrors`	bool	Optional

`startDelay`	int (seconds)	Optional
`timeout`	int (seconds)	Optional
`when`	[string, string, ...]	Optional

Examples: push

Push multiple images

```
az acr run -f build-push-hello-world.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world:$ID -f hello-world.dockerfile .
  - push:
    - $Registry/hello-world:$ID
```

Build, push, and run

```
az acr run -f build-run-hello-world.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world:$ID -f hello-world.dockerfile .
  - push:
    - $Registry/hello-world:$ID
  - cmd: $Registry/hello-world:$ID
```

cmd

The `cmd` step type runs a container.

Syntax: cmd

```
version: v1.1.0
steps:
  - [cmd]: [containerImage]:[tag (optional)] [cmdParameters to the image]
```

Properties: cmd

The `cmd` step type supports the following properties:

`detach`	bool	Optional
`disableWorkingDirectoryOverride`	bool	Optional
`entryPoint`	string	Optional
`env`	[string, string, ...]	Optional

`expose`	[string, string, ...]	Optional
`id`	string	Optional
`ignoreErrors`	bool	Optional
`isolation`	string	Optional
`keep`	bool	Optional
`network`	object	Optional
`ports`	[string, string, ...]	Optional
`pull`	bool	Optional
`repeat`	int	Optional
`retries`	int	Optional
`retryDelay`	int (seconds)	Optional
`secret`	object	Optional
`startDelay`	int (seconds)	Optional
`timeout`	int (seconds)	Optional
`when`	[string, string, ...]	Optional
`workingDirectory`	string	Optional

You can find details of these properties in the Task step properties section of this article.

Examples: cmd

Run hello-world image

This command executes the `hello-world.yaml` task file, which references the hello-world image on Docker Hub.

```
az acr run -f hello-world.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  - cmd: hello-world
```

Run bash image and echo "hello world"

This command executes the `bash-echo.yaml` task file, which references the bash image on Docker Hub.

```
az acr run -f bash-echo.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  - cmd: bash echo hello world
```

Run specific bash image tag

To run a specific image version, specify the tag in the `cmd`.

This command executes the `bash-echo-3.yaml` task file, which references the bash:3.0 image on Docker Hub.

```
az acr run -f bash-echo-3.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  - cmd: bash:3.0 echo hello world
```

Run custom images

The `cmd` step type references images using the standard `docker run` format. Images not prefaced with a registry are assumed to originate from docker.io. The previous example could equally be represented as:

```
version: v1.1.0
steps:
  - cmd: docker.io/bash:3.0 echo hello world
```

By using the standard `docker run` image reference convention, `cmd` can run images from any private registry or the public Docker Hub. If you're referencing images in the same registry in which ACR Task is executing, you don't need to specify any registry credentials.

- Run an image that's from an Azure container registry. The following example assumes you have a registry named `myregistry`, and a custom image `myimage:mytag`.

  ```
  version: v1.1.0
  steps:
      - cmd: myregistry.azurecr.io/myimage:mytag
  ```

- Generalize the registry reference with a Run variable or alias

  Instead of hard-coding your registry name in an `acr-task.yaml` file, you can make it more portable by using a Run variable or alias. The `Run.Registry` variable or `$Registry` alias expands at runtime to the name of the registry in which the task is executing.

  For example, to generalize the preceding task so that it works in any Azure container registry, reference the $Registry variable in the image name:

  ```
  version: v1.1.0
  steps:
    - cmd: $Registry/myimage:mytag
  ```

Task step properties

Each step type supports several properties appropriate for its type. The following table defines all of the available step properties. Not all step types support all properties. To see which of these properties are available for each step type, see the cmd, build, and push step type reference sections.

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`detach`	bool	Yes	Whether the container should be detached when running.	`false`
`disableWorkingDirectoryOverride`	bool	Yes	Whether to disable `workingDirectory` override functionality. Use this in combination with `workingDirectory` to have complete control over the container's working directory.	`false`
`entryPoint`	string	Yes	Overrides the `[ENTRYPOINT]` of a step's container.	None
`env`	[string, string, ...]	Yes	Array of strings in `key=value` format that define the environment variables for the step.	None
`expose`	[string, string, ...]	Yes	Array of ports that are exposed from the container.	None
`id`	string	Yes	Uniquely identifies the step within the task. Other steps in the task can reference a step's `id`, such as for dependency checking with `when`. The `id` is also the running container's name. Processes running in other containers in the task can refer to the `id` as its DNS host name, or for accessing it with docker logs [id], for example.	`acb_step_%d`, where `%d` is the 0-based index of the step top-down in the YAML file
`ignoreErrors`	bool	Yes	Whether to mark the step as successful regardless of whether an error occurred during container execution.	`false`

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`isolation`	string	Yes	The isolation level of the container.	`default`
`keep`	bool	Yes	Whether the step's container should be kept after execution.	`false`
`network`	object	Yes	Identifies a network in which the container runs.	None
`ports`	[string, string, ...]	Yes	Array of ports that are published from the container to the host.	None
`pull`	bool	Yes	Whether to force a pull of the container before executing it to prevent any caching behavior.	`false`
`privileged`	bool	Yes	Whether to run the container in privileged mode.	`false`
`repeat`	int	Yes	The number of retries to repeat the execution of a container.	0
`retries`	int	Yes	The number of retries to attempt if a container fails its execution. A retry is only attempted if a container's exit code is non-zero.	0
`retryDelay`	int (seconds)	Yes	The delay in seconds between retries of a container's execution.	0
`secret`	object	Yes	Identifies an Azure Key Vault secret or managed identity for Azure resources.	None
`startDelay`	int (seconds)	Yes	Number of seconds to delay a container's execution.	0
`timeout`	int (seconds)	Yes	Maximum number of seconds a step may execute before being terminated.	600

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`when`	[string, string, ...]	Yes	Configures a step's dependency on one or more other steps within the task.	None
`user`	string	Yes	The user name or UID of a container	None
`workingDirectory`	string	Yes	Sets the working directory for a step. By default, ACR Tasks creates a root directory as the working directory. However, if your build has several steps, earlier steps can share artifacts with later steps by specifying the same working directory.	`/workspace`

Examples: Task step properties

Example: id

Build two images, instancing a functional test image. Each step is identified by a unique `id` which other steps in the task reference in their `when` property.

```
az acr run -f when-parallel-dependent.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  # build website and func-test images, concurrently
  - id: build-hello-world
    build: -t $Registry/hello-world:$ID -f hello-world.dockerfile .
    when: ["-"]
  - id: build-hello-world-test
    build: -t hello-world-test -f hello-world.dockerfile .
    when: ["-"]
  # run built images to be tested
  - id: hello-world
    cmd: $Registry/hello-world:$ID
    when: ["build-hello-world"]
  - id: func-tests
    cmd: hello-world-test
    env:
      - TEST_TARGET_URL=hello-world
    when: ["hello-world"]
  # push hello-world if func-tests are successful
  - push: ["$Registry/hello-world:$ID"]
    when: ["func-tests"]
```

Example: when

The `when` property specifies a step's dependency on other steps within the task. It supports two parameter values:

- `when: ["-"]` - Indicates no dependency on other steps. A step specifying `when: ["-"]` will begin execution immediately, and enables concurrent step execution.

- `when: ["id1", "id2"]` - Indicates the step is dependent upon steps with `id` "id1" and `id` "id2". This step won't be executed until both "id1" and "id2" steps complete.

If `when` isn't specified in a step, that step is dependent on completion of the previous step in the `acr-task.yaml` file.

Sequential step execution without `when`:

```
az acr run -f when-sequential-default.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
    - cmd: bash echo one
    - cmd: bash echo two
    - cmd: bash echo three
```

Sequential step execution with `when`:

```
az acr run -f when-sequential-id.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
    - id: step1
      cmd: bash echo one
    - id: step2
      cmd: bash echo two
      when: ["step1"]
    - id: step3
      cmd: bash echo three
      when: ["step2"]
```

Parallel images build:

```
az acr run -f when-parallel.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  # build website and func-test images, concurrently
  - id: build-hello-world
    build: -t $Registry/hello-world:$ID -f hello-world.dockerfile .
    when: ["-"]
  - id: build-hello-world-test
    build: -t hello-world-test -f hello-world.dockerfile .
    when: ["-"]
```

Parallel image build and dependent testing:

```
az acr run -f when-parallel-dependent.yaml https://github.com/Azure-Samples/acr-tasks.git
```

```
version: v1.1.0
steps:
  # build website and func-test images, concurrently
  - id: build-hello-world
    build: -t $Registry/hello-world:$ID -f hello-world.dockerfile .
    when: ["-"]
  - id: build-hello-world-test
    build: -t hello-world-test -f hello-world.dockerfile .
    when: ["-"]
  # run built images to be tested
  - id: hello-world
    cmd: $Registry/hello-world:$ID
    when: ["build-hello-world"]
  - id: func-tests
    cmd: hello-world-test
    env:
      - TEST_TARGET_URL=hello-world
    when: ["hello-world"]
  # push hello-world if func-tests are successful
  - push: ["$Registry/hello-world:$ID"]
    when: ["func-tests"]
```

Run variables

ACR Tasks includes a default set of variables that are available to task steps when they execute. These variables can be accessed by using the format `{{.Run.VariableName}}`, where `VariableName` is one of the following:

- `Run.ID`
- `Run.SharedVolume`
- `Run.Registry`
- `Run.RegistryName`
- `Run.Date`
- `Run.OS`
- `Run.Architecture`
- `Run.Commit`
- `Run.Branch`
- `Run.TaskName`

The variable names are generally self-explanatory. Details follows for commonly used variables. As of YAML version `v1.1.0`, you can use an abbreviated, predefined task alias in place of most run variables. For example, in place of `{{.Run.Registry}}`, use the `$Registry` alias.

Run.ID

Each Run, through `az acr run`, or trigger based execution of tasks created through `az acr task create`, has a unique ID. The ID represents the Run currently being executed.

Typically used for a uniquely tagging an image:

```
version: v1.1.0
steps:
    - build: -t $Registry/hello-world:$ID .
```

Run.Registry

The fully qualified server name of the registry. Typically used to generically reference the registry where the task is being run.

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world:$ID .
```

Run.RegistryName

The name of the container registry. Typically used in task steps that don't require a fully qualified server name, for example, `cmd` steps that run Azure CLI commands on registries.

```
version 1.1.0
steps:
# List repositories in registry
- cmd: az login --identity
- cmd: az acr repository list --name $RegistryName
```

Run.Date

The current UTC time the run began.

Run.Commit

For a task triggered by a commit to a GitHub repository, the commit identifier.

Run.Branch

For a task triggered by a commit to a GitHub repository, the branch name.

Aliases

As of `v1.1.0`, ACR Tasks supports aliases that are available to task steps when they execute. Aliases are similar in concept to aliases (command shortcuts) supported in bash and some other command shells.

With an alias, you can launch any command or group of commands (including options and filenames) by entering a single word.

ACR Tasks supports several predefined aliases and also custom aliases you create.

Predefined aliases

The following task aliases are available to use in place of run variables:

ALIAS	RUN VARIABLE
`ID`	`Run.ID`
`SharedVolume`	`Run.SharedVolume`
`Registry`	`Run.Registry`
`RegistryName`	`Run.RegistryName`
`Date`	`Run.Date`
`OS`	`Run.OS`
`Architecture`	`Run.Architecture`
`Commit`	`Run.Commit`

ALIAS	RUN VARIABLE
`Branch`	`Run.Branch`

In task steps, precede an alias with the `$` directive, as in this example:

```
version: v1.1.0
steps:
  - build: -t $Registry/hello-world:$ID -f hello-world.dockerfile .
```

Image aliases

Each of the following aliases points to a stable image in Microsoft Container Registry (MCR). You can refer to each of them in the `cmd` section of a Task file without using a directive.

ALIAS	IMAGE
`acr`	`mcr.microsoft.com/acr/acr-cli:0.1`
`az`	`mcr.microsoft.com/acr/azure-cli:a80af84`
`bash`	`mcr.microsoft.com/acr/bash:a80af84`
`curl`	`mcr.microsoft.com/acr/curl:a80af84`

The following example task uses several aliases to purge image tags older than 7 days in the repo `samples/hello-world` in the run registry:

```
version: v1.1.0
steps:
  - cmd: acr tag list --registry $RegistryName --repository samples/hello-world
  - cmd: acr purge --registry $RegistryName --filter samples/hello-world:.* --ago 7d
```

Custom alias

Define a custom alias in your YAML file and use it as shown in the following example. An alias can contain only alphanumeric characters. The default directive to expand an alias is the `$` character.

```
version: v1.1.0
alias:
  values:
    repo: myrepo
steps:
  - build: -t $Registry/$repo/hello-world:$ID -f Dockerfile .
```

You can link to a remote or local YAML file for custom alias definitions. The following example links to a YAML file in Azure blob storage:

```
version: v1.1.0
alias:
  src:  # link to local or remote custom alias files
    - 'https://link/to/blob/remoteAliases.yml?readSasToken'
[...]
```

Next steps

For an overview of multi-step tasks, see the Run multi-step build, test, and patch tasks in ACR Tasks.

For single-step builds, see the ACR Tasks overview.

Azure Container Registry webhook reference

11/24/2019 • 5 minutes to read • Edit Online

You can configure webhooks for your container registry that generate events when certain actions are performed against it. For example, enable webhooks that are triggered when a container image or Helm chart is pushed to a registry, or deleted. When a webhook is triggered, Azure Container Registry issues an HTTP or HTTPS request containing information about the event to an endpoint you specify. Your endpoint can then process the webhook and act accordingly.

The following sections detail the schema of webhook requests generated by supported events. The event sections contain the payload schema for the event type, an example request payload, and one or more example commands that would trigger the webhook.

For information about configuring webhooks for your Azure container registry, see Using Azure Container Registry webhooks.

Webhook requests

HTTP request

A triggered webhook makes an HTTP `POST` request to the URL endpoint you specified when you configured the webhook.

HTTP headers

Webhook requests include a `Content-Type` of `application/json` if you have not specified a `Content-Type` custom header for your webhook.

No other headers are added to the request beyond those custom headers you might have specified for the webhook.

Push event

Webhook triggered when a container image is pushed to a repository.

Push event payload

ELEMENT	TYPE	DESCRIPTION
`id`	String	The ID of the webhook event.
`timestamp`	DateTime	The time at which the webhook event was triggered.
`action`	String	The action that triggered the webhook event.
target	Complex Type	The target of the event that triggered the webhook event.
request	Complex Type	The request that generated the webhook event.

target

ELEMENT	TYPE	DESCRIPTION
`mediaType`	String	The MIME type of the referenced object.
`size`	Int32	The number of bytes of the content. Same as Length field.
`digest`	String	The digest of the content, as defined by the Registry V2 HTTP API Specification.
`length`	Int32	The number of bytes of the content. Same as Size field.
`repository`	String	The repository name.
`tag`	String	The image tag name.

request

ELEMENT	TYPE	DESCRIPTION
`id`	String	The ID of the request that initiated the event.
`host`	String	The externally accessible hostname of the registry instance, as specified by the HTTP host header on incoming requests.
`method`	String	The request method that generated the event.
`useragent`	String	The user agent header of the request.

Payload example: image push event

```
{
  "id": "cb8c3971-9adc-488b-xxxx-43cbb4974ff5",
  "timestamp": "2017-11-17T16:52:01.343145347Z",
  "action": "push",
  "target": {
    "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
    "size": 524,
    "digest": "sha256:xxxxd5c8786bb9e621a45ece0dbxxxx1cdc624ad20da9fe62e9d25490f33xxxx",
    "length": 524,
    "repository": "hello-world",
    "tag": "v1"
  },
  "request": {
    "id": "3cbb6949-7549-4fa1-xxxx-a6d5451dffc7",
    "host": "myregistry.azurecr.io",
    "method": "PUT",
    "useragent": "docker/17.09.0-ce go/go1.8.3 git-commit/afdb6d4 kernel/4.10.0-27-generic os/linux arch/amd64
UpstreamClient(Docker-Client/17.09.0-ce \\(linux\\))"
  }
}
```

Example Docker CLI command that triggers the image **push** event webhook:

```
docker push myregistry.azurecr.io/hello-world:v1
```

Chart push event

Webhook triggered when a Helm chart is pushed to a repository.

Chart push event payload

ELEMENT	TYPE	DESCRIPTION
`id`	String	The ID of the webhook event.
`timestamp`	DateTime	The time at which the webhook event was triggered.
`action`	String	The action that triggered the webhook event.
target	Complex Type	The target of the event that triggered the webhook event.

target

ELEMENT	TYPE	DESCRIPTION
`mediaType`	String	The MIME type of the referenced object.
`size`	Int32	The number of bytes of the content.
`digest`	String	The digest of the content, as defined by the Registry V2 HTTP API Specification.
`repository`	String	The repository name.
`tag`	String	The chart tag name.
`name`	String	The chart name.
`version`	String	The chart version.

Payload example: chart push event

```
{
  "id": "6356e9e0-627f-4fed-xxxx-d9059b5143ac",
  "timestamp": "2019-03-05T23:45:31.2614267Z",
  "action": "chart_push",
  "target": {
    "mediaType": "application/vnd.acr.helm.chart",
    "size": 25265,
    "digest": "sha256:xxxx8075264b5ba7c14c23672xxxx52ae6a3ebac1c47916e4efe19cd624dxxxx",
    "repository": "repo",
    "tag": "wordpress-5.4.0.tgz",
    "name": "wordpress",
    "version": "5.4.0.tgz"
  }
}
```

Example Azure CLI command that triggers the **chart_push** event webhook:

```
az acr helm push wordpress-5.4.0.tgz --name MyRegistry
```

Delete event

Webhook triggered when an image repository or manifest is deleted. Not triggered when a tag is deleted.

Delete event payload

ELEMENT	TYPE	DESCRIPTION
`id`	String	The ID of the webhook event.
`timestamp`	DateTime	The time at which the webhook event was triggered.
`action`	String	The action that triggered the webhook event.
target	Complex Type	The target of the event that triggered the webhook event.
request	Complex Type	The request that generated the webhook event.

target

ELEMENT	TYPE	DESCRIPTION
`mediaType`	String	The MIME type of the referenced object.
`digest`	String	The digest of the content, as defined by the Registry V2 HTTP API Specification.
`repository`	String	The repository name.

request

ELEMENT	TYPE	DESCRIPTION
`id`	String	The ID of the request that initiated the event.
`host`	String	The externally accessible hostname of the registry instance, as specified by the HTTP host header on incoming requests.
`method`	String	The request method that generated the event.
`useragent`	String	The user agent header of the request.

Payload example: image delete event

```
{
    "id": "afc359ce-df7f-4e32-xxxx-1ff8aa80927b",
    "timestamp": "2017-11-17T16:54:53.657764628Z",
    "action": "delete",
    "target": {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "digest": "sha256:xxxxd5c8786bb9e621a45ece0dbxxxx1cdc624ad20da9fe62e9d25490f33xxxx",
      "repository": "hello-world"
    },
    "request": {
      "id": "3d78b540-ab61-4f75-xxxx-7ca9ecf559b3",
      "host": "myregistry.azurecr.io",
      "method": "DELETE",
      "useragent": "python-requests/2.18.4"
    }
  }
```

Example Azure CLI commands that trigger a **delete** event webhook:

```
# Delete repository
az acr repository delete --name MyRegistry --repository MyRepository

# Delete image
az acr repository delete --name MyRegistry --image MyRepository:MyTag
```

Chart delete event

Webhook triggered when a Helm chart or repository is deleted.

Chart delete event payload

ELEMENT	TYPE	DESCRIPTION
`id`	String	The ID of the webhook event.
`timestamp`	DateTime	The time at which the webhook event was triggered.
`action`	String	The action that triggered the webhook event.

ELEMENT	TYPE	DESCRIPTION
target	Complex Type	The target of the event that triggered the webhook event.

target

ELEMENT	TYPE	DESCRIPTION
`mediaType`	String	The MIME type of the referenced object.
`size`	Int32	The number of bytes of the content.
`digest`	String	The digest of the content, as defined by the Registry V2 HTTP API Specification.
`repository`	String	The repository name.
`tag`	String	The chart tag name.
`name`	String	The chart name.
`version`	String	The chart version.

Payload example: chart delete event

```
{
  "id": "338a3ef7-ad68-4128-xxxx-fdd3af8e8f67",
  "timestamp": "2019-03-06T00:10:48.1270754Z",
  "action": "chart_delete",
  "target": {
    "mediaType": "application/vnd.acr.helm.chart",
    "size": 25265,
    "digest": "sha256:xxxx8075264b5ba7c14c23672xxxx52ae6a3ebac1c47916e4efe19cd624dxxxx",
    "repository": "repo",
    "tag": "wordpress-5.4.0.tgz",
    "name": "wordpress",
    "version": "5.4.0.tgz"
  }
}
```

Example Azure CLI command that triggers the **chart_delete** event webhook:

```
az acr helm delete wordpress --version 5.4.0 --name MyRegistry
```

Next steps

Using Azure Container Registry webhooks

Azure Event Grid event schema for Container Registry

3/15/2019 • 3 minutes to read • Edit Online

This article provides the properties and schema for Container Registry events. For an introduction to event schemas, see Azure Event Grid event schema.

Available event types

Azure Container Registry emits the following event types:

EVENT TYPE	DESCRIPTION
Microsoft.ContainerRegistry.ImagePushed	Raised when an image is pushed.
Microsoft.ContainerRegistry.ImageDeleted	Raised when an image is deleted.
Microsoft.ContainerRegistry.ChartPushed	Raised when a Helm chart is pushed.
Microsoft.ContainerRegistry.ChartDeleted	Raised when a Helm chart is deleted.

Example event

The following example shows the schema of an image pushed event:

```
[{
  "id": "831e1650-001e-001b-66ab-eeb76e069631",
  "topic": "/subscriptions/<subscription-id>/resourceGroups/<resource-group-
name>/providers/Microsoft.ContainerRegistry/registries/<name>",
  "subject": "aci-helloworld:v1",
  "eventType": "ImagePushed",
  "eventTime": "2018-04-25T21:39:47.6549614Z",
  "data": {
    "id": "31c51664-e5bd-416a-a5df-e5206bc47ed0",
    "timestamp": "2018-04-25T21:39:47.276585742Z",
    "action": "push",
    "target": {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 3023,
      "digest": "sha256:213bbc182920ab41e18edc2001e06abcca6735d87782d9cef68abd83941cf0e5",
      "length": 3023,
      "repository": "aci-helloworld",
      "tag": "v1"
    },
    "request": {
      "id": "7c66f28b-de19-40a4-821c-6f5f6c0003a4",
      "host": "demo.azurecr.io",
      "method": "PUT",
      "useragent": "docker/18.03.0-ce go/go1.9.4 git-commit/0520e24 os/windows arch/amd64
UpstreamClient(Docker-Client/18.03.0-ce \\\\(windows\\\\))"
    }
  },
  "dataVersion": "1.0",
  "metadataVersion": "1"
}]
```

The schema for an image deleted event is similar:

```
[{
  "id": "f06e3921-301f-42ec-b368-212f7d5354bd",
  "topic": "/subscriptions/<subscription-id>/resourceGroups/<resource-group-
name>/providers/Microsoft.ContainerRegistry/registries/<name>",
  "subject": "aci-helloworld",
  "eventType": "ImageDeleted",
  "eventTime": "2018-04-26T17:56:01.8211268Z",
  "data": {
    "id": "f06e3921-301f-42ec-b368-212f7d5354bd",
    "timestamp": "2018-04-26T17:56:00.996603117Z",
    "action": "delete",
    "target": {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "digest": "sha256:213bbc182920ab41e18edc2001e06abcca6735d87782d9cef68abd83941cf0e5",
      "repository": "aci-helloworld"
    },
    "request": {
      "id": "aeda5b99-4197-409f-b8a8-ff539edb7de2",
      "host": "demo.azurecr.io",
      "method": "DELETE",
      "useragent": "python-requests/2.18.4"
    }
  },
  "dataVersion": "1.0",
  "metadataVersion": "1"
}]
```

The schema for a chart pushed event is similar to the schema for an imaged pushed event, but it doesn't include a request object:

```
[{
  "id": "ea3a9c28-5b17-40f6-a500-3f02b6829277",
  "topic": "/subscriptions/<subscription-id>/resourceGroups/<resource-group-
name>/providers/Microsoft.ContainerRegistry/registries/<name>",
  "subject": "mychart:1.0.0",
  "eventType": "Microsoft.ContainerRegistry.ChartPushed",
  "eventTime": "2019-03-12T22:16:31.5164086Z",
  "data": {
    "id":"ea3a9c28-5b17-40f6-a500-3f02b682927",
    "timestamp":"2019-03-12T22:16:31.0087496+00:00",
    "action":"chart_push",
    "target":{
      "mediaType":"application/vnd.acr.helm.chart",
      "size":25265,
      "digest":"sha256:7f060075264b5ba7c14c23672698152ae6a3ebac1c47916e4efe19cd624d5fab",
      "repository":"repo",
      "tag":"mychart-1.0.0.tgz",
      "name":"mychart",
      "version":"1.0.0"
    }
  },
  "dataVersion": "1.0",
  "metadataVersion": "1"
}]
```

The schema for a chart deleted event is similar to the schema for an imaged deleted event, but it doesn't include a request object:

```
[{
  "id": "39136b3a-1a7e-416f-a09e-5c85d5402fca",
  "topic": "/subscriptions/<subscription-id>/resourceGroups/<resource-group-
name>/providers/Microsoft.ContainerRegistry/registries/<name>",
  "subject": "mychart:1.0.0",
  "eventType": "Microsoft.ContainerRegistry.ChartDeleted",
  "eventTime": "019-03-12T22:42:08.7034064Z",
  "data": {
    "id":"ea3a9c28-5b17-40f6-a500-3f02b682927",
    "timestamp":"2019-03-12T22:42:08.3783775+00:00",
    "action":"chart_delete",
    "target":{
      "mediaType":"application/vnd.acr.helm.chart",
      "size":25265,
      "digest":"sha256:7f060075264b5ba7c14c23672698152ae6a3ebac1c47916e4efe19cd624d5fab",
      "repository":"repo",
      "tag":"mychart-1.0.0.tgz",
      "name":"mychart",
      "version":"1.0.0"
    }
  },
  "dataVersion": "1.0",
  "metadataVersion": "1"
}]
```

Event properties

An event has the following top-level data:

PROPERTY	TYPE	DESCRIPTION

PROPERTY	TYPE	DESCRIPTION
topic	string	Full resource path to the event source. This field is not writeable. Event Grid provides this value.
subject	string	Publisher-defined path to the event subject.
eventType	string	One of the registered event types for this event source.
eventTime	string	The time the event is generated based on the provider's UTC time.
id	string	Unique identifier for the event.
data	object	Blob storage event data.
dataVersion	string	The schema version of the data object. The publisher defines the schema version.
metadataVersion	string	The schema version of the event metadata. Event Grid defines the schema of the top-level properties. Event Grid provides this value.

The data object has the following properties:

PROPERTY	TYPE	DESCRIPTION
id	string	The event ID.
timestamp	string	The time at which the event occurred.
action	string	The action that encompasses the provided event.
target	object	The target of the event.
request	object	The request that generated the event.

The target object has the following properties:

PROPERTY	TYPE	DESCRIPTION
mediaType	string	The MIME type of the referenced object.
size	integer	The number of bytes of the content. Same as Length field.
digest	string	The digest of the content, as defined by the Registry V2 HTTP API Specification.

PROPERTY	TYPE	DESCRIPTION
length	integer	The number of bytes of the content. Same as Size field.
repository	string	The repository name.
tag	string	The tag name.
name	string	The chart name.
version	string	The chart version.

The request object has the following properties:

PROPERTY	TYPE	DESCRIPTION
id	string	The ID of the request that initiated the event.
addr	string	The IP or hostname and possibly port of the client connection that initiated the event. This value is the RemoteAddr from the standard http request.
host	string	The externally accessible hostname of the registry instance, as specified by the http host header on incoming requests.
method	string	The request method that generated the event.
useragent	string	The user agent header of the request.

Next steps

- For an introduction to Azure Event Grid, see What is Event Grid?
- For more information about creating an Azure Event Grid subscription, see Event Grid subscription schema.

Health check error reference

11/24/2019 • 4 minutes to read • Edit Online

Following are details about error codes returned by the az acr check-health command. For each error, possible solutions are listed.

DOCKER_COMMAND_ERROR

This error means that Docker client for CLI could not be found. As a result, the following additional checks are not run: finding Docker version, evaluating Docker daemon status, and running a Docker pull command.

Potential solutions: Install Docker client; add Docker path to the system variables.

DOCKER_DAEMON_ERROR

This error means that the Docker daemon status is unavailable, or that it could not be reached using the CLI. As a result, Docker operations (such as `docker login` and `docker pull`) are unavailable through the CLI.

Potential solutions: Restart Docker daemon, or validate that it is properly installed.

DOCKER_VERSION_ERROR

This error means that CLI was not able to run the command `docker --version`.

Potential solutions: Try running the command manually, make sure you have the latest CLI version, and investigate the error message.

DOCKER_PULL_ERROR

This error means that the CLI was not able to pull a sample image to your environment.

Potential solutions: Validate that all components necessary to pull an image are running properly.

HELM_COMMAND_ERROR

This error means that Helm client could not be found by the CLI, which precludes other Helm operations.

Potential solutions: Verify that Helm client is installed, and that its path is added to the system environment variables.

HELM_VERSION_ERROR

This error means that the CLI was unable to determine the Helm version installed. This can happen if the Azure CLI version (or if the Helm version) being used is obsolete.

Potential solutions: Update to the latest Azure CLI version or to the recommended Helm version; run the command manually and investigate the error message.

CONNECTIVITY_DNS_ERROR

This error means that the DNS for the given registry login server was pinged but did not respond, which means it is unavailable. This can indicate some connectivity issues. Alternatively, the registry might not exist, the user might

not have the permissions on the registry (to retrieve its login server properly), or the target registry is in a different cloud than the one used in the Azure CLI.

Potential solutions: Validate connectivity; verify spelling of the registry, and that registry exists; verify that the user has the right permissions on it and that the registry's cloud is the same that is used in the Azure CLI.

CONNECTIVITY_FORBIDDEN_ERROR

This error means that the challenge endpoint for the given registry responded with a 403 Forbidden HTTP status. This error means that users don't have access to the registry, most likely because of a virtual network configuration. To see the currently configured firewall rules, run `az acr show --query networkRuleSet --name <registry>`.

Potential solutions: Remove virtual network rules, or add the current client IP address to the allowed list.

CONNECTIVITY_CHALLENGE_ERROR

This error means that the challenge endpoint of the target registry did not issue a challenge.

Potential solutions: Try again after some time. If the error persists, open an issue at https://aka.ms/acr/issues.

CONNECTIVITY_AAD_LOGIN_ERROR

This error means that the challenge endpoint of the target registry issued a challenge, but the registry does not support Azure Active Directory authentication.

Potential solutions: Try a different way to authenticate, for example, with admin credentials. If users need to authenticate using Azure Active Directory, open an issue at https://aka.ms/acr/issues.

CONNECTIVITY_REFRESH_TOKEN_ERROR

This error means that the registry login server did not respond with a refresh token, so access to the target registry was denied. This error can occur if the user does not have the right permissions on the registry or if the user credentials for the Azure CLI are stale.

Potential solutions: Verify if the user has the right permissions on the registry; run `az login` to refresh permissions, tokens, and credentials.

CONNECTIVITY_ACCESS_TOKEN_ERROR

This error means that the registry login server did not respond with an access token, so that the access to the target registry was denied. This error can occur if the user does not have the right permissions on the registry or if the user credentials for the Azure CLI are stale.

Potential solutions: Verify if the user has the right permissions on the registry; run `az login` to refresh permissions, tokens, and credentials.

CONNECTIVITY_SSL_ERROR

This error means that the client was unable to establish a secure connection to the container registry. This error generally occurs if you're running or using a proxy server.

Potential solutions: More information on working behind a proxy can be found here.

LOGIN_SERVER_ERROR

This error means that the CLI was unable to find the login server of the given registry, and no default suffix was found for the current cloud. This error can occur if the registry does not exist, if the user does not have the right permissions on the registry, if the registry's cloud and the current Azure CLI cloud do not match, or if the Azure CLI version is obsolete.

Potential solutions: Verify that the spelling is correct and that the registry exists; verify that user has the right permissions on the registry, and that the clouds of the registry and the CLI environment match; update Azure CLI to the latest version.

Next steps

For options to check the health of a registry, see Check the health of an Azure container registry.

See the FAQ for frequently asked questions and other known issues about Azure Container Registry.