Contents

- Azure Container Registry documentation
- Container registries
  - Overview
    - About Container Registry
  - Quickstarts
    - Create container registry - CLI
    - Create container registry - Portal
    - Create container registry - PowerShell
    - Create container registry - ARM template
    - Create container registry - Bicep
    - Send events to Event Grid - CLI
    - Manage content with client libraries
    - Build and push Java images - CLI
  - Tutorials
    - Geo-replicate a registry
      - 1 - Prepare container registry
      - 2 - Deploy web application
      - 3 - Update web application
  - Samples
    - Azure Resource Graph queries
  - Concepts
    - Security
      - Security controls by Azure Policy
      - Security baseline
    - Container registry tiers and limits
    - Registries, repositories, and images
    - Image storage
    - Content formats
    - Tag and version images

- Geo-replication
- Zone redundancy
- Registry best practices
- Multi-arch images
- Manage public images
- How-to guides
  - Registry operations
    - Push and pull an image
    - Push and pull an OCI artifact
    - Push and pull a Helm chart
    - Push and pull supply chain artifacts (preview)
    - Sign and verify containers with notation (preview)
    - View repositories
    - Import container images
    - Transfer artifacts (preview)
      - Overview and prerequisites
      - ACR Transfer with Az CLI
      - ACR Transfer with ARM templates
      - Troubleshooting
    - Protect registry data
      - Lock container images
      - Disable export
    - Delete container images
      - Delete image data - CLI
      - Retention policy for untagged manifests (preview)
      - Automatically purge tags and manifests (preview)
    - Use ACR webhooks
    - Move registry to different region
  - Security and authentication
    - Restrict access using private endpoint
    - Configure public registry access
    - Restrict access using service endpoint (preview)

- Allow access by trusted services
- Encrypt with customer-managed key
- Access behind a firewall
- Authentication
  - Authentication overview
  - Authenticate with service principal
  - Authenticate with managed identity
  - Authenticate with token (preview)
  - Authenticate from Azure Container Instances
  - Authenticate from Kubernetes
    - Kubernetes scenarios
    - Attach registry to Azure Kubernetes Service (AKS)
    - Cross-tenant authentication from AKS
    - Authenticate using Kubernetes pull secret
  - Enable unauthenticated pull access
- Azure role-based access control
- Content trust
- Registries and other Azure services
- Azure Container Instances
- Azure Kubernetes Service (AKS)
- Service Fabric
- Monitor
- Monitor Container Registry
- Audit compliance using Azure Policy
- Scan container images
  - Scan with Microsoft Defender for Cloud
  - Scan with GitHub Actions

Tasks

- Overview
  - About ACR Tasks
- Quickstarts
  - Build, push, and run image - CLI

- Tutorials
  - Automate container image builds
    - 1 - Build from source context
    - 2a - Build on code commit
    - 2b - Multi-step task on code commit
    - 3a - Build on base image update
    - 3b - Build on private base image update
    - 4 - Build on a schedule
- Samples
  - YAML and Dockerfiles
- Concepts
  - Multi-step tasks
  - Base image updates
- How-to guides
  - Access secured resources with managed identities
    - Enable managed identity on a task
    - Cross-registry authentication
    - External authentication using key vault
  - Quick task run with Resource Manager template
  - Run task on dedicated agent pool (preview)
  - Consume and maintain public content
  - Build image with Buildpacks (preview)
  - View task logs
Connected registry
- Overview
  - About connected registry
  - Use with IoT Edge
- Quickstarts
  - Create connected registry - CLI
  - Create connected registry - Portal
  - Deploy connected registry to IoT Edge
- Tutorials

- Deploy connected registry to nested IoT Edge
- Concepts
  - Access to connected registry
- How-to guides
  - Pull image from connected registry

Reference
- Azure CLI
- REST
- PowerShell
- .NET
- Python
- Java
- Node.js
- Resource Manager template
- Tasks YAML
- Webhook schema
- Event Grid schema
- Health check errors
- Azure Policy built-ins
- Monitoring data

Troubleshoot
- Troubleshoot login issues
- Troubleshoot network issues
- Troubleshoot performance issues
- Detect common issues
- Technical Support policies
- FAQ

Resources
- Region availability
- Pricing
- Roadmap
- Provide product feedback

Stack Overflow

Videos

Introduction to Container registries in Azure

8/22/2022 • 3 minutes to read • Edit Online

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

Learn 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 the Azure CLI, the 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 service tiers** - Create one or more container registries in your Azure subscription. Registries are available in three tiers: 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 Azure role-based access control (Azure RBAC) to assign users or systems fine-grained permissions to a registry.

Security features of the Premium service tier include content trust for image tag signing, and firewalls and virtual networks (preview) to restrict access to the registry. Microsoft Defender for Cloud 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

8/22/2022 • 4 minutes to read • Edit Online

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts. In this quickstart, you create an Azure container registry instance with 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 service tiers.

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 `<registry-name>` is a placeholder for the container registry name, and `<login-server>` is a placeholder for the registry's login server name.

> **TIP**
> In this quickstart, you create a *Basic* registry, which is a cost-optimized option for developers learning about Azure Container Registry. Choose other tiers for increased storage and image throughput, and capabilities such as connection using a private endpoint. For details on available service tiers (SKUs), see Container registry service tiers.

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. Specify only the registry resource name when logging in with the Azure CLI. Don't use the fully qualified login server name.

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

Example:

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

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 public image. For this example, pull the `hello-world` image from Microsoft Container Registry.

```
docker pull mcr.microsoft.com/hello-world
```

Before you can push an image to your registry, you must tag it with the fully qualified name of your registry

login server. The login server name is in the format *<registry-name>.azurecr.io* (must be all lowercase), for example, *mycontainerregistry.azurecr.io*.

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

```
docker tag mcr.microsoft.com/hello-world <login-server>/hello-world:v1
```

Example:

```
docker tag mcr.microsoft.com/hello-world mycontainerregistry.azurecr.io/hello-world:v1
```

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

```
docker push <login-server>/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 <login-server>/hello-world:v1
```

List container images

The following example lists the repositories in your registry:

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

Output:

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

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

```
az acr repository show-tags --name <registry-name> --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 <login-server>/hello-world:v1
```

Example output:

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

Azure Container Registry Tasks tutorials

Quickstart: Create an Azure container registry using the Azure portal

8/22/2022 • 4 minutes to read • Edit Online

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts. In this quickstart, you create an Azure container registry instance 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.

- Azure CLI
- Azure PowerShell

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**.

In the **Basics** tab, enter values for **Resource group** and **Registry name**. 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'.

Accept default values for the remaining settings. Then select **Review + create**. After reviewing the settings, select **Create**.

> **TIP**
> In this quickstart, you create a *Basic* registry, which is a cost-optimized option for developers learning about Azure Container Registry. Choose other tiers for increased storage and image throughput, and capabilities such as connection using a private endpoint. For details on available service tiers (SKUs), see Container registry service tiers.

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

Take note of the registry name and the value of the **Login server**, which is a fully qualified name ending with `azurecr.io` in the Azure cloud. You use these values in the following steps when you push and pull images with Docker.

Log in to registry

- Azure CLI
- Azure PowerShell

Before pushing and pulling container images, you must log in to the registry instance. Sign into the Azure CLI on your local machine, then run the az acr login command. Specify only the registry resource name when logging in with the Azure CLI. Don't use the fully qualified login server name.

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

Example:

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

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 public image. For this example, pull the `hello-world` image from Microsoft Container Registry.

```
docker pull mcr.microsoft.com/hello-world
```

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

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

```
docker tag mcr.microsoft.com/hello-world <login-server>/hello-world:v1
```

Example:

```
docker tag mcr.microsoft.com/hello-world mycontainerregistry.azurecr.io/hello-world:v1
```

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

```
docker push <login-server>/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 <login-server>/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 **hello-world** repository you created with `docker push`.

By selecting the **hello-world** repository, you 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 <login-server>/hello-world:v1
```

Example output:

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

Azure Container Registry Tasks tutorials

Quickstart: Create a private container registry using Azure PowerShell

8/22/2022 • 3 minutes to read • Edit Online

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts. In this quickstart, you create an Azure container registry instance with Azure 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 uses the Azure Az PowerShell module, which is the recommended PowerShell module for interacting with Azure. To get started with the Az PowerShell module, see Install Azure PowerShell. To learn how to migrate to the Az PowerShell module, see Migrate Azure PowerShell from AzureRM to Az.

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
```

> **TIP**
> In this quickstart, you create a *Basic* registry, which is a cost-optimized option for developers learning about Azure Container Registry. Choose other tiers for increased storage and image throughput, and capabilities such as connection using a private endpoint. For details on available service tiers (SKUs), see Container registry service tiers.

Log in to registry

Before pushing and pulling container images, you must log in to your registry with the Connect-AzContainerRegistry cmdlet. The following example uses the same credentials you logged in with when authenticating to Azure with the `Connect-AzAccount` cmdlet.

> **NOTE**
> In the following example, the value of `$registry.Name` is the resource name, not the fully qualified registry name.

```
Connect-AzContainerRegistry -Name $registry.Name
```

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 public image. For this example, pull the `hello-world` image from Microsoft Container Registry.

```
docker pull mcr.microsoft.com/hello-world
```

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

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

```
docker tag mcr.microsoft.com/hello-world <login-server>/hello-world:v1
```

Example:

```
docker tag mcr.microsoft.com/hello-world mycontainerregistry.azurecr.io/hello-world:v1
```

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

```
docker push <login-server>/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 <login-server>/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 <login-server>/hello-world:v1
```

Example output:

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

Azure Container Registry Tasks tutorials

Quickstart: Create a geo-replicated container registry by using an ARM template

8/22/2022 • 3 minutes to read • Edit Online

This quickstart shows how to create an Azure Container Registry instance by using an Azure Resource Manager template (ARM template). The template sets up a geo-replicated registry, which automatically synchronizes registry content across more than one Azure region. Geo-replication enables network-close access to images from regional deployments, while providing a single management experience. It's a feature of the Premium registry service tier.

An ARM template is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

The registry with replications does not support the ARM/Bicep template Complete mode deployments.

If your environment meets the prerequisites and you're familiar with using ARM templates, select the **Deploy to Azure** button. The template will open in the Azure portal.

Deploy to Azure

Prerequisites

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

Review the template

The template used in this quickstart is from Azure Quickstart Templates. The template sets up a registry and an additional regional replica.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "metadata": {
    "_generator": {
      "name": "bicep",
      "version": "0.5.6.12127",
      "templateHash": "12610175857982700190"
    }
  },
  "parameters": {
    "acrName": {
      "type": "string",
      "defaultValue": "[format('acr{0}', uniqueString(resourceGroup().id))]",
      "maxLength": 50,
      "minLength": 5,
      "metadata": {
        "description": "Globally unique name of your Azure Container Registry"
      }
    },
    "acrAdminUserEnabled": {
      "type": "bool",
      "defaultValue": false,
      "metadata": {
        "description": "Enable admin user that has push / pull permission to the registry."
```

```
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for registry home replica."
      }
    },
    "acrSku": {
      "type": "string",
      "defaultValue": "Premium",
      "allowedValues": [
        "Premium"
      ],
      "metadata": {
        "description": "Tier of your Azure Container Registry. Geo-replication requires Premium SKU."
      }
    },
    "acrReplicaLocation": {
      "type": "string",
      "metadata": {
        "description": "Short name for registry replica location."
      }
    }
  },
  "resources": [
    {
      "type": "Microsoft.ContainerRegistry/registries",
      "apiVersion": "2019-12-01-preview",
      "name": "[parameters('acrName')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[parameters('acrSku')]"
      },
      "tags": {
        "displayName": "Container Registry",
        "container.registry": "[parameters('acrName')]"
      },
      "properties": {
        "adminUserEnabled": "[parameters('acrAdminUserEnabled')]"
      }
    },
    {
      "type": "Microsoft.ContainerRegistry/registries/replications",
      "apiVersion": "2019-12-01-preview",
      "name": "[format('{0}/{1}', parameters('acrName'), parameters('acrReplicaLocation'))]",
      "location": "[parameters('acrReplicaLocation')]",
      "properties": {},
      "dependsOn": [
        "[resourceId('Microsoft.ContainerRegistry/registries', parameters('acrName'))]"
      ]
    }
  ],
  "outputs": {
    "acrLoginServer": {
      "type": "string",
      "value": "[reference(resourceId('Microsoft.ContainerRegistry/registries',
parameters('acrName'))).loginServer]"
    }
  }
}
```

The following resources are defined in the template:

- **Microsoft.ContainerRegistry/registries**: create an Azure container registry
- **Microsoft.ContainerRegistry/registries/replications**: create a container registry replica

More Azure Container Registry template samples can be found in the quickstart template gallery.

Deploy the template

1. Select the following image to sign in to Azure and open a template.

   Deploy to Azure

2. Select or enter the following values.

   - **Subscription**: select an Azure subscription.
   - **Resource group**: select **Create new**, enter a unique name for the resource group, and then select **OK**.
   - **Region**: select a location for the resource group. Example: **Central US**.
   - **Acr Name**: accept the generated name for the registry, or enter a name. It must be globally unique.
   - **Acr Admin User Enabled**: accept the default value.
   - **Location**: accept the generated location for the registry's home replica, or enter a location such as **Central US**.
   - **Acr Sku**: accept the default value.
   - **Acr Replica Location**: enter a location for the registry replica, using the region's short name. It must be different from the home registry location. Example: **westeurope**.

3. Select **Review + Create**, then review the terms and conditions. If you agree, select **Create**.

4. After the registry has been created successfully, you get a notification:

The Azure portal is used to deploy the template. In addition to the Azure portal, you can use the Azure PowerShell, Azure CLI, and REST API. To learn other deployment methods, see Deploy templates.

Review deployed resources

Use the Azure portal or a tool such as the Azure CLI to review the properties of the container registry.

1. In the portal, search for Container Registries, and select the container registry you created.

2. On the **Overview** page, note the **Login server** of the registry. Use this URI when you use Docker to tag and push images to your registry. For information, see Push your first image using the Docker CLI.

3. On the **Replications** page, confirm the locations of the home replica and the replica added through the template. If desired, add more replicas on this page.

Clean up resources

When you no longer need them, delete the resource group, the registry, and the registry replica. To do so, go to the Azure portal, select the resource group that contains the registry, and then select **Delete resource group**.

Next steps

In this quickstart, you created an Azure Container Registry with an ARM template, and configured a registry replica in another location. Continue to the Azure Container Registry tutorials for a deeper look at ACR.

Azure Container Registry tutorials

For a step-by-step tutorial that guides you through the process of creating a template, see:

Tutorial: Create and deploy your first ARM template

Quickstart: Create a container registry by using a Bicep file

8/22/2022 • 2 minutes to read • Edit Online

This quickstart shows how to create an Azure Container Registry instance by using a Bicep file.

Bicep is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It provides concise syntax, reliable type safety, and support for code reuse. Bicep offers the best authoring experience for your infrastructure-as-code solutions in Azure.

Prerequisites

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

Review the Bicep file

Use Visual studio code or your favorite editor to create a file with the following content and name it **main.bicep**:

```
@minLength(5)
@maxLength(50)
@description('Provide a globally unique name of your Azure Container Registry')
param acrName string = 'acr${uniqueString(resourceGroup().id)}'

@description('Provide a location for the registry.')
param location string = resourceGroup().location

@description('Provide a tier of your Azure Container Registry.')
param acrSku string = 'Basic'

resource acrResource 'Microsoft.ContainerRegistry/registries@2021-06-01-preview' = {
  name: acrName
  location: location
  sku: {
    name: acrSku
  }
  properties: {
    adminUserEnabled: false
  }
}

@description('Output the login server property for later use')
output loginServer string = acrResource.properties.loginServer
```

The following resource is defined in the Bicep file:

- **Microsoft.ContainerRegistry/registries**: create an Azure container registry

More Azure Container Registry template samples can be found in the quickstart template gallery.

Deploy the Bicep file

To deploy the file you've created, open PowerShell or Azure CLI. If you want to use the integrated Visual Studio Code terminal, select the `ctrl` + `` ` `` key combination. Change the current directory to where the Bicep file is

located.

- CLI
- PowerShell

```
az group create --name myContainerRegRG --location centralus

az deployment group create --resource-group myContainerRegRG --template-file main.bicep --parameters
acrName={your-unique-name}
```

> **NOTE**
>
> Replace **{your-unique-name}**, including the curly braces, with a unique container registry name.

When the deployment finishes, you should see a message indicating the deployment succeeded.

Review deployed resources

Use the Azure portal or a tool such as the Azure CLI to review the properties of the container registry.

1. In the portal, search for **Container Registries**, and select the container registry you created.
2. On the **Overview** page, note the **Login server** of the registry. Use this URI when you use Docker to tag and push images to your registry. For information, see Push your first image using the Docker CLI.

Clean up resources

When you no longer need the resource, delete the resource group, and the registry. To do so, go to the Azure portal, select the resource group that contains the registry, and then select **Delete resource group**.

Next steps

In this quickstart, you created an Azure Container Registry with a Bicep file. Continue to the Azure Container Registry tutorials for a deeper look at ACR.

Azure Container Registry tutorials

For a step-by-step tutorial that guides you through the process of creating a Bicep file, see:

Quickstart: Create Bicep files with Visual Studio Code

Quickstart: Send events from private container registry to Event Grid

8/22/2022 • 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 an Azure free account before you begin.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.

- Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

- 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 deployment group 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 the Event Grid resource provider

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

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

It might 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/Mic
rosoft.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.

> **NOTE**
> The Dockerfile used in the following example depends on a public base container image from Docker Hub. To improve reliability when using public content, import and manage the image in a private Azure container registry, and update your Dockerfile to use your privately managed base image. Learn more about working with public images.

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

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.

```
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:

```
[
  "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:

```
This operation will delete the manifest
'sha256:f15fa9d0a69081ba93eee308b0e475a54fac9c682196721e294b2bc20ab23a1b' and all the following images:
'myimage:v1'.
Are you sure you want to continue? (y/n):
```

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

Quickstart: Use the Azure Container Registry client libraries

8/22/2022 • 10 minutes to read • Edit Online

Use this article to get started with the client library for Azure Container Registry. Follow these steps to try out example code for data-plane operations on images and artifacts.

Use the client library for Azure Container Registry to:

- List images or artifacts in a registry
- Obtain metadata for images and artifacts, repositories, and tags
- Set read/write/delete properties on registry items
- Delete images and artifacts, repositories, and tags

Azure Container Registry also has a management library for control-plane operations including registry creation and updates.

Prerequisites

- You need an Azure subscription and an Azure container registry to use this library.

  To create a new Azure container registry, you can use the Azure portal, Azure PowerShell, or the Azure CLI. Here's an example using the Azure CLI:

  ```
  az acr create --name MyContainerRegistry --resource-group MyResourceGroup \
      --location westus --sku Basic
  ```

- Push one or more container images to your registry. For steps, see Push your first image to your Azure container registry using the Docker CLI.

Key concepts

- An Azure container registry stores *container images* and OCI artifacts.
- An image or artifact consists of a *manifest* and *layers*.
- A manifest describes the layers that make up the image or artifact. It is uniquely identified by its *digest*.
- An image or artifact can also be *tagged* to give it a human-readable alias. An image or artifact can have zero or more tags associated with it, and each tag uniquely identifies the image.
- A collection of images or artifacts that share the same name, but have different tags, is a *repository*.

For more information, see About registries, repositories, and artifacts.

Get started

Source code | Package (NuGet) | API reference | Samples

To develop .NET application code that can connect to an Azure Container Registry instance, you will need the `Azure.Containers.ContainerRegistry` library.

Install the package

Install the Azure Container Registry client library for .NET with NuGet:

```
dotnet add package Azure.Containers.ContainerRegistry --prerelease
```

Authenticate the client

For your application to connect to your registry, you'll need to create a `ContainerRegistryClient` that can authenticate with it. Use the Azure Identity library to add Azure Active Directory support for authenticating Azure SDK clients with their corresponding Azure services.

When you're developing and debugging your application locally, you can use your own user to authenticate with your registry. One way to accomplish this is to authenticate your user with the Azure CLI and run your application from this environment. If your application is using a client that has been constructed to authenticate with `DefaultAzureCredential`, it will correctly authenticate with the registry at the specified endpoint.

```
// Create a ContainerRegistryClient that will authenticate to your registry through Azure Active Directory
Uri endpoint = new Uri("https://myregistry.azurecr.io");
ContainerRegistryClient client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential(),
    new ContainerRegistryClientOptions()
    {
        Audience = ContainerRegistryAudience.AzureResourceManagerPublicCloud
    });
```

See the Azure Identity README for more approaches to authenticating with `DefaultAzureCredential`, both locally and in deployment environments. To connect to registries in non-public Azure clouds, see the API reference.

For more information on using Azure AD with Azure Container Registry, see the authentication overview.

Examples

Each sample assumes there is a `REGISTRY_ENDPOINT` environment variable set to a string containing the `https://` prefix and the name of the login server, for example "https://myregistry.azurecr.io".

The following samples use asynchronous APIs that return a task. Synchronous APIs are also available.

List repositories asynchronously

Iterate through the collection of repositories in the registry.

```
// Get the service endpoint from the environment
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("REGISTRY_ENDPOINT"));

// Create a new ContainerRegistryClient
ContainerRegistryClient client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential(),
    new ContainerRegistryClientOptions()
    {
        Audience = ContainerRegistryAudience.AzureResourceManagerPublicCloud
    });

// Get the collection of repository names from the registry
AsyncPageable<string> repositories = client.GetRepositoryNamesAsync();
await foreach (string repository in repositories)
{
    Console.WriteLine(repository);
}
```

Set artifact properties asynchronously

```
// Get the service endpoint from the environment
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("REGISTRY_ENDPOINT"));

// Create a new ContainerRegistryClient
ContainerRegistryClient client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential(),
    new ContainerRegistryClientOptions()
    {
        Audience = ContainerRegistryAudience.AzureResourceManagerPublicCloud
    });

// Get the collection of repository names from the registry
AsyncPageable<string> repositories = client.GetRepositoryNamesAsync();
await foreach (string repository in repositories)
{
    Console.WriteLine(repository);
}
```

Delete images asynchronously

```
using System.Linq;
using Azure.Containers.ContainerRegistry;
using Azure.Identity;

// Get the service endpoint from the environment
Uri endpoint = new Uri(Environment.GetEnvironmentVariable("REGISTRY_ENDPOINT"));

// Create a new ContainerRegistryClient
ContainerRegistryClient client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential(),
    new ContainerRegistryClientOptions()
    {
        Audience = ContainerRegistryAudience.AzureResourceManagerPublicCloud
    });

// Iterate through repositories
AsyncPageable<string> repositoryNames = client.GetRepositoryNamesAsync();
await foreach (string repositoryName in repositoryNames)
{
    ContainerRepository repository = client.GetRepository(repositoryName);

    // Obtain the images ordered from newest to oldest
    AsyncPageable<ArtifactManifestProperties> imageManifests =
        repository.GetManifestPropertiesCollectionAsync(orderBy:
ArtifactManifestOrderBy.LastUpdatedOnDescending);

    // Delete images older than the first three.
    await foreach (ArtifactManifestProperties imageManifest in imageManifests.Skip(3))
    {
        RegistryArtifact image = repository.GetArtifact(imageManifest.Digest);
        Console.WriteLine($"Deleting image with digest {imageManifest.Digest}.");
        Console.WriteLine($"   Deleting the following tags from the image: ");
        foreach (var tagName in imageManifest.Tags)
        {
            Console.WriteLine($"        {imageManifest.RepositoryName}:{tagName}");
            await image.DeleteTagAsync(tagName);
        }
        await image.DeleteAsync();
    }
}
```

Get started

Source code | Package (Maven) | API reference | Samples

Currently supported environments

- Java Development Kit (JDK), version 8 or later.

Include the package

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-containers-containerregistry</artifactId>
  <version>1.0.0-beta.3</version>
</dependency>
```

Authenticate the client

The Azure Identity library provides Azure Active Directory support for authentication.

The following samples assume you have a registry endpoint string containing the `https://` prefix and the name of the login server, for example "https://myregistry.azurecr.io".

```
DefaultAzureCredential credential = new DefaultAzureCredentialBuilder().build();
ContainerRegistryClient client = new ContainerRegistryClientBuilder()
    .endpoint(endpoint)
    .credential(credential)
    .buildClient();
```

```
DefaultAzureCredential credential = new DefaultAzureCredentialBuilder().build();
ContainerRegistryAsyncClient client = new ContainerRegistryClientBuilder()
    .endpoint(endpoint)
    .credential(credential)
    .buildAsyncClient();
```

For more information on using Azure AD with Azure Container Registry, see the authentication overview.

Examples

Each sample assumes there is a registry endpoint string containing the `https://` prefix and the name of the login server, for example "https://myregistry.azurecr.io".

List repository names

Iterate through the collection of repositories in the registry.

```
DefaultAzureCredential credential = new DefaultAzureCredentialBuilder().build();
ContainerRegistryClient client = new ContainerRegistryClientBuilder()
    .endpoint(endpoint)
    .credential(credential)
    .buildClient();

client.listRepositoryNames().forEach(repository -> System.out.println(repository));
```

Set artifact properties

```
TokenCredential defaultCredential = new DefaultAzureCredentialBuilder().build();

ContainerRegistryClient client = new ContainerRegistryClientBuilder()
    .endpoint(endpoint)
    .credential(defaultCredential)
    .buildClient();

RegistryArtifact image = client.getArtifact(repositoryName, digest);

image.updateTagProperties(
    tag,
    new ArtifactTagProperties()
        .setWriteEnabled(false)
        .setDeleteEnabled(false));
```

Delete images

```
TokenCredential defaultCredential = new DefaultAzureCredentialBuilder().build();

ContainerRegistryClient client = new ContainerRegistryClientBuilder()
    .endpoint(endpoint)
    .credential(defaultCredential)
    .buildClient();

final int imagesCountToKeep = 3;
for (String repositoryName : client.listRepositoryNames()) {
    final ContainerRepository repository = client.getRepository(repositoryName);

    // Obtain the images ordered from newest to oldest
    PagedIterable<ArtifactManifestProperties> imageManifests =
        repository.listManifestProperties(
            ArtifactManifestOrderBy.LAST_UPDATED_ON_DESCENDING,
            Context.NONE);

    imageManifests.stream().skip(imagesCountToKeep)
        .forEach(imageManifest -> {
            System.out.printf(String.format("Deleting image with digest %s.%n", imageManifest.getDigest()));
            System.out.printf("    This image has the following tags: ");

            for (String tagName : imageManifest.getTags()) {
                System.out.printf("        %s:%s", imageManifest.getRepositoryName(), tagName);
            }

            repository.getArtifact(imageManifest.getDigest()).delete();
        });
}
```

Get started

Source code | Package (npm) | API reference | Samples

Currently supported environments

- LTS versions of Node.js

See our support policy for more details.

Install the `@azure/container-registry` package

Install the Container Registry client library for JavaScript with `npm`:

```
npm install @azure/container-registry
```

Authenticate the client

The Azure Identity library provides Azure Active Directory support for authentication.

```
const { ContainerRegistryClient } = require("@azure/container-registry");
const { DefaultAzureCredential } = require("@azure/identity");

const endpoint = process.env.CONTAINER_REGISTRY_ENDPOINT;
// Create a ContainerRegistryClient that will authenticate through Active Directory
const client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential());
```

For more information on using Azure AD with Azure Container Registry, see the authentication overview.

Examples

Each sample assumes there is a `CONTAINER_REGISTRY_ENDPOINT` environment variable set to a string containing the `https://` prefix and the name of the login server, for example "https://myregistry.azurecr.io".

List repositories asynchronously

Iterate through the collection of repositories in the registry.

```
const { ContainerRegistryClient } = require("@azure/container-registry");
const { DefaultAzureCredential } = require("@azure/identity");

async function main() {
  // endpoint should be in the form of "https://myregistryname.azurecr.io"
  // where "myregistryname" is the actual name of your registry
  const endpoint = process.env.CONTAINER_REGISTRY_ENDPOINT || "<endpoint>";
  const client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential());

  console.log("Listing repositories");
  const iterator = client.listRepositoryNames();
  for await (const repository of iterator) {
    console.log(`  repository: ${repository}`);
  }
}

main().catch((err) => {
  console.error("The sample encountered an error:", err);
});
```

Set artifact properties asynchronously

```
const { ContainerRegistryClient } = require("@azure/container-registry");
const { DefaultAzureCredential } = require("@azure/identity");

async function main() {
  // Get the service endpoint from the environment
  const endpoint = process.env.CONTAINER_REGISTRY_ENDPOINT || "<endpoint>";

  // Create a new ContainerRegistryClient and RegistryArtifact to access image operations
  const client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential());
  const image = client.getArtifact("library/hello-world", "v1");

  // Set permissions on the image's "latest" tag
  await image.updateTagProperties("latest", { canWrite: false, canDelete: false });
}

main().catch((err) => {
  console.error("The sample encountered an error:", err);
});
```

Delete images asynchronously

```
const { ContainerRegistryClient } = require("@azure/container-registry");
const { DefaultAzureCredential } = require("@azure/identity");

async function main() {
  // Get the service endpoint from the environment
  const endpoint = process.env.CONTAINER_REGISTRY_ENDPOINT || "<endpoint>";
  // Create a new ContainerRegistryClient
  const client = new ContainerRegistryClient(endpoint, new DefaultAzureCredential());

  // Iterate through repositories
  const repositoryNames = client.listRepositoryNames();
  for await (const repositoryName of repositoryNames) {
    const repository = client.getRepository(repositoryName);
    // Obtain the images ordered from newest to oldest by passing the `orderBy` option
    const imageManifests = repository.listManifestProperties({
      orderBy: "LastUpdatedOnDescending"
    });
    const imagesToKeep = 3;
    let imageCount = 0;
    // Delete images older than the first three.
    for await (const manifest of imageManifests) {
      imageCount++;
      if (imageCount > imagesToKeep) {
        const image = repository.getArtifact(manifest.digest);
        console.log(`Deleting image with digest ${manifest.digest}`);
        console.log(`  Deleting the following tags from the image:`);
        for (const tagName of manifest.tags) {
          console.log(`    ${manifest.repositoryName}:${tagName}`);
          image.deleteTag(tagName);
        }
        await image.delete();
      }
    }
  }
}

main().catch((err) => {
  console.error("The sample encountered an error:", err);
});
```

Get started

Source code | Package (Pypi) | API reference | Samples

Install the package

Install the Azure Container Registry client library for Python with pip:

```
pip install --pre azure-containerregistry
```

Authenticate the client

The Azure Identity library provides Azure Active Directory support for authentication. The `DefaultAzureCredential` assumes the `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, and `AZURE_CLIENT_SECRET` environment variables are set. For more information, see Azure Identity environment variables.

```
# Create a ContainerRegistryClient that will authenticate through Active Directory
from azure.containerregistry import ContainerRegistryClient
from azure.identity import DefaultAzureCredential

account_url = "https://mycontainerregistry.azurecr.io"
client = ContainerRegistryClient(account_url, DefaultAzureCredential())
```

Examples

Each sample assumes there is a `CONTAINERREGISTRY_ENDPOINT` environment variable set to a string containing the `https://` prefix and the name of the login server, for example "https://myregistry.azurecr.io".

List tags asynchronously

This sample assumes the registry has a repository `hello-world`.

```
import asyncio
from dotenv import find_dotenv, load_dotenv
import os

from azure.containerregistry.aio import ContainerRegistryClient
from azure.identity.aio import DefaultAzureCredential


class ListTagsAsync(object):
    def __init__(self):
        load_dotenv(find_dotenv())

    async def list_tags(self):
        # Create a new ContainerRegistryClient
        audience = "https://management.azure.com"
        account_url = os.environ["CONTAINERREGISTRY_ENDPOINT"]
        credential = DefaultAzureCredential()
        client = ContainerRegistryClient(account_url, credential, audience=audience)

        manifest = await client.get_manifest_properties("library/hello-world", "latest")
        print(manifest.repository_name + ": ")
        for tag in manifest.tags:
            print(tag + "\n")
```

Set artifact properties asynchronously

This sample assumes the registry has a repository `hello-world` with image tagged `v1`.

```
import asyncio
from dotenv import find_dotenv, load_dotenv
import os

from azure.containerregistry.aio import ContainerRegistryClient
from azure.identity.aio import DefaultAzureCredential


class SetImagePropertiesAsync(object):
    def __init__(self):
        load_dotenv(find_dotenv())

    async def set_image_properties(self):
        # Create a new ContainerRegistryClient
        account_url = os.environ["CONTAINERREGISTRY_ENDPOINT"]
        audience = "https://management.azure.com"
        credential = DefaultAzureCredential()
        client = ContainerRegistryClient(account_url, credential, audience=audience)

        # [START update_manifest_properties]
        # Set permissions on the v1 image's "latest" tag
        await client.update_manifest_properties(
            "library/hello-world",
            "latest",
            can_write=False,
            can_delete=False
        )
        # [END update_manifest_properties]
        # After this update, if someone were to push an update to "myacr.azurecr.io\hello-world:v1", it
would fail.
        # It's worth noting that if this image also had another tag, such as "latest", and that tag did not
have
        # permissions set to prevent reads or deletes, the image could still be overwritten. For example,
        # if someone were to push an update to "myacr.azurecr.io\hello-world:latest"
        # (which references the same image), it would succeed.
```

Delete images asynchronously

```
import asyncio
from dotenv import find_dotenv, load_dotenv
import os

from azure.containerregistry import ManifestOrder
from azure.containerregistry.aio import ContainerRegistryClient
from azure.identity.aio import DefaultAzureCredential


class DeleteImagesAsync(object):
    def __init__(self):
        load_dotenv(find_dotenv())

    async def delete_images(self):
        # [START list_repository_names]
        audience = "https://management.azure.com"
        account_url = os.environ["CONTAINERREGISTRY_ENDPOINT"]
        credential = DefaultAzureCredential()
        client = ContainerRegistryClient(account_url, credential, audience=audience)

        async with client:
            async for repository in client.list_repository_names():
                print(repository)
                # [END list_repository_names]

                # [START list_manifest_properties]
                # Keep the three most recent images, delete everything else
                manifest_count = 0
                async for manifest in client.list_manifest_properties(repository,
order_by=ManifestOrder.LAST_UPDATE_TIME_DESCENDING):
                    manifest_count += 1
                    if manifest_count > 3:
                        await client.delete_manifest(repository, manifest.digest)
                # [END list_manifest_properties]
```

Clean up resources

If you want to clean up and remove an Azure container registry, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

Next steps

In this quickstart, you learned about using the Azure Container Registry client library to perform operations on images and artifacts in your container registry.

- For more information, see the API reference documentation:
  - .NET
  - Java
  - JavaScript
  - Python
- Learn about the Azure Container Registry REST API.

Quickstart: Build and push container images of the Java Spring Boot app to Azure Container Registry

8/22/2022 • 4 minutes to read • Edit Online

You can use this Quickstart to build container images of Java Spring Boot app and push it to Azure Container Registry using Maven and Jib. Maven and Jib are one way of using developer tooling to interact with an Azure container registry.

Prerequisites

- An Azure subscription; Sign up for a free Azure account or activate MSDN subscriber benefits if you don't already have an Azure subscription.
- A supported Java Development Kit (JDK); For more information on available JDKs when developing on Azure, see Java support on Azure and Azure Stack.
- The Azure CLI.
- The Apache's Maven build tool (Version 3 or above).
- A Git client.
- A Docker client.
- The ACR Docker credential helper.

Create and build a Spring Boot application on Docker

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

1. From the command prompt, use the following command to clone the Spring Boot on Docker Getting Started sample project.

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

2. Change directory to the complete project.

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

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

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

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

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

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

Create an Azure Container Registry using the Azure CLI

Next, you'll create an Azure resource group and your ACR using the following steps:

1. Log in to your Azure account by using the following command:

```
az login
```

2. Specify the Azure subscription to use:

```
az account set -s <subscription ID>
```

3. Create a resource group for the Azure resources used in this tutorial. In the following command, be sure to replace the placeholders with your own resource name and a location such as `eastus`.

```
az group create \
    --name=<your resource group name> \
    --location=<location>
```

4. Create a private Azure container registry in the resource group, using the following command. Be sure to replace the placeholders with actual values. The tutorial pushes the sample app as a Docker image to this registry in later steps.

```
az acr create \
    --resource-group <your resource group name> \
    --location <location> \
    --name <your registry name> \
    --sku Basic
```

Push your app to the container registry via Jib

Finally, you'll update your project configuration and use the command prompt to build and deploy your image.

> **NOTE**
> To log in the Azure container registry that you just created, you will need to have the Docker daemon running. To install Docker on your machine, here is the official Docker documentation.

1. Log in to your Azure Container Registry from the Azure CLI using the following command. Be sure to replace the placeholder with your own registry name.

```
az config set defaults.acr=<your registry name>
az acr login
```

   The `az config` command sets the default registry name to use with `az acr` commands.

2. Navigate to the completed project directory for your Spring Boot application (for example, "*C:\SpringBoot\gs-spring-boot-docker\complete*" or "*/users/robert/SpringBoot/gs-spring-boot-docker/complete*"), and open the *pom.xml* file with a text editor.

3. Update the `<properties>` collection in the *pom.xml* file with the following XML. Replace the placeholder with your registry name, and add a `<jib-maven-plugin.version>` property with value `2.2.0`, or a newer version of the jib-maven-plugin.

```
<properties>
   <docker.image.prefix><your registry name>.azurecr.io</docker.image.prefix>
   <java.version>1.8</java.version>
   <jib-maven-plugin.version>2.2.0</jib-maven-plugin.version>
</properties>
```

4. Update the `<plugins>` collection in the *pom.xml* file so that the `<plugin>` element contains and an entry for the `jib-maven-plugin`, as shown in the following example. Note that we are using a base image from the Microsoft Container Registry (MCR): `mcr.microsoft.com/java/jdk:8-zulu-alpine`, which contains an officially supported JDK for Azure. For other MCR base images with officially supported JDKs, see Java SE JDK, Java SE JRE, Java SE Headless JRE, and Java SE JDK and Maven.

```
<plugin>
  <artifactId>jib-maven-plugin</artifactId>
  <groupId>com.google.cloud.tools</groupId>
  <version>${jib-maven-plugin.version}</version>
  <configuration>
     <from>
         <image>mcr.microsoft.com/java/jdk:8-zulu-alpine</image>
     </from>
     <to>
         <image>${docker.image.prefix}/${project.artifactId}</image>
     </to>
  </configuration>
</plugin>
```

5. Navigate to the complete project directory for your Spring Boot application and run the following command to build the image and push the image to the registry:

```
az acr login && mvn compile jib:build
```

NOTE

For security reasons, the credential created by `az acr login` is valid for 1 hour only. If you receive a *401 Unauthorized* error, you can run the `az acr login -n <your registry name>` command again to reauthenticate.

Verify your container image

Congratulations! Now you have your containerized Java App build on Azure supported JDK pushed to your ACR. You can now test the image by deploying it to Azure App Service, or pulling it to local with command (replacing the placeholder):

```
docker pull <your registry name>.azurecr.io/gs-spring-boot-docker
```

Next steps

For other versions of the official Microsoft-supported Java base images, see:

- Java SE JDK
- Java SE JRE
- Java SE Headless JRE
- Java SE JDK and Maven

To learn more about Spring and Azure, continue to the Spring on Azure documentation center.

Spring on Azure

Additional Resources

For more information, see the following resources:

- Azure for Java Developers
- Working with Azure DevOps and Java
- Spring Boot on Docker Getting Started
- Spring Initializr
- Deploy a Spring Boot Application to the Azure App Service
- Using a custom Docker image for Azure Web App on Linux

Tutorial: Prepare a geo-replicated Azure container registry

8/22/2022 • 6 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

For this tutorial, you need an Azure container registry in the Premium service tier. To create a new Azure container registry, follow the steps in this section.

> **TIP**
> If you previously created a registry and need to upgrade, see Changing tiers.

Sign in to the Azure portal.

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

Configure your new registry with the following settings. In the **Basics** tab:

- **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`
- **SKU**: `Premium` (required for geo-replication)

Select **Review + create** and then **Create** to create the registry 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	...

Enable admin account

In subsequent tutorials, you deploy a container image from the registry directly to Web App for Containers. To enable this capability, you must also enable the registry's admin account.

Navigate to your new container registry in the Azure portal and select **Access keys** under **Settings**. Under **Admin user**, select **Enable**.

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 registry 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 ASP.NET Core runtime image, copies the application files into the container, installs dependencies, compiles the output using the official .NET Core SDK image, and finally, builds an optimized aspnetcore image.

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

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 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 mcr.microsoft.com/dotnet/core/sdk:2.2 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 mcr.microsoft.com/dotnet/core/aspnet:2.2 AS base
2.2: Pulling from mcr.microsoft.com/dotnet/core/aspnet
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

8/22/2022 • 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 second 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

8/22/2022 • 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 Resource Graph sample queries for Azure Container Registry

8/22/2022 • 2 minutes to read • Edit Online

This page is a collection of Azure Resource Graph sample queries for Azure Container Registry. For a complete list of Azure Resource Graph samples, see Resource Graph samples by Category and Resource Graph samples by Table.

Sample queries

List Container Registry vulnerability assessment results

Returns all the all the vulnerabilities found on container images. Microsoft Defender for Containers has to be enabled in order to view these security findings.

```
SecurityResources
| where type == 'microsoft.security/assessments'
| where properties.displayName contains 'Container registry images should have vulnerability findings
resolved'
| summarize by assessmentKey=name //the ID of the assessment
| join kind=inner (
 securityresources
 | where type == 'microsoft.security/assessments/subassessments'
 | extend assessmentKey = extract('.*assessments/(.+?)/.*',1,  id)
) on assessmentKey
| project assessmentKey, subassessmentKey=name, id, parse_json(properties), resourceGroup, subscriptionId,
tenantId
| extend description = properties.description,
 displayName = properties.displayName,
 resourceId = properties.resourceDetails.id,
 resourceSource = properties.resourceDetails.source,
 category = properties.category,
 severity = properties.status.severity,
 code = properties.status.code,
 timeGenerated = properties.timeGenerated,
 remediation = properties.remediation,
 impact = properties.impact,
 vulnId = properties.id,
 additionalData = properties.additionalData
```

- Azure CLI
- Azure PowerShell
- Portal

```
az graph query -q "SecurityResources | where type == 'microsoft.security/assessments' | where
properties.displayName contains 'Container registry images should have vulnerability findings resolved' |
summarize by assessmentKey=name //the ID of the assessment | join kind=inner ( securityresources | where
type == 'microsoft.security/assessments/subassessments' | extend assessmentKey =
extract('.*assessments/(.+?)/.*',1, id) ) on assessmentKey | project assessmentKey, subassessmentKey=name,
id, parse_json(properties), resourceGroup, subscriptionId, tenantId | extend description =
properties.description, displayName = properties.displayName, resourceId = properties.resourceDetails.id,
resourceSource = properties.resourceDetails.source, category = properties.category, severity =
properties.status.severity, code = properties.status.code, timeGenerated = properties.timeGenerated,
remediation = properties.remediation, impact = properties.impact, vulnId = properties.id, additionalData =
properties.additionalData"
```

Next steps

- Learn more about the query language.
- Learn more about how to explore resources.
- See samples of Starter language queries.
- See samples of Advanced language queries.

Azure Policy Regulatory Compliance controls for Azure Container Registry

8/22/2022 • 11 minutes to read • Edit Online

Regulatory Compliance in Azure Policy provides Microsoft created and managed initiative definitions, known as *built-ins*, for the **compliance domains** and **security controls** related to different compliance standards. This page lists the **compliance domains** and **security controls** for Azure Container Registry. You can assign the built-ins for a **security control** individually to help make your Azure resources compliant with the specific standard.

The title of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Policy Version** column to view the source on the Azure Policy GitHub repo.

> **IMPORTANT**
>
> Each control is associated with one or more Azure Policy definitions. These policies might help you assess compliance with the control. However, there often isn't a one-to-one or complete match between a control and one or more policies. As such, **Compliant** in Azure Policy refers only to the policies themselves. This doesn't ensure that you're fully compliant with all requirements of a control. In addition, the compliance standard includes controls that aren't addressed by any Azure Policy definitions at this time. Therefore, compliance in Azure Policy is only a partial view of your overall compliance status. The associations between controls and Azure Policy Regulatory Compliance definitions for these compliance standards can change over time.

Azure Security Benchmark

The Azure Security Benchmark provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Azure Security Benchmark, see the Azure Security Benchmark mapping files.

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - Azure Security Benchmark.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Network Security	NS-2	Secure cloud services with network controls	Container registries should not allow unrestricted network access	2.0.0
Network Security	NS-2	Secure cloud services with network controls	Container registries should use private link	1.0.1
Data Protection	DP-5	Use customer-managed key option in data at rest encryption when required	Container registries should be encrypted with a customer-managed key	1.1.2

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Posture and Vulnerability Management	PV-6	Rapidly and automatically remediate vulnerabilities	Container registry images should have vulnerability findings resolved	2.0.1
DevOps Security	DS-6	Enforce security of workload throughout DevOps lifecycle	Container registry images should have vulnerability findings resolved	2.0.1

Azure Security Benchmark v1

The Azure Security Benchmark provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Azure Security Benchmark, see the Azure Security Benchmark mapping files.

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - Azure Security Benchmark.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Network Security	1.1	Protect resources using Network Security Groups or Azure Firewall on your Virtual Network	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview

CMMC Level 3

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - CMMC Level 3. For more information about this compliance standard, see Cybersecurity Maturity Model Certification (CMMC).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC.1.001	Limit information system access to authorized users, processes acting on behalf of authorized users, and devices (including other information systems).	Container registries should not allow unrestricted network access	2.0.0
Access Control	AC.1.002	Limit information system access to the types of transactions and functions that authorized users are permitted to execute.	Container registries should not allow unrestricted network access	2.0.0

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Access Control	AC.2.016	Control the flow of CUI in accordance with approved authorizations.	Container registries should not allow unrestricted network access	2.0.0
Configuration Management	CM.3.068	Restrict, disable, or prevent the use of nonessential programs, functions, ports, protocols, and services.	Container registries should not allow unrestricted network access	2.0.0
Risk Assessment	RM.2.143	Remediate vulnerabilities in accordance with risk assessments.	Container registry images should have vulnerability findings resolved	2.0.1
System and Communications Protection	SC.1.175	Monitor, control, and protect communications (i.e., information transmitted or received by organizational systems) at the external boundaries and key internal boundaries of organizational systems.	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC.3.177	Employ FIPS-validated cryptography when used to protect the confidentiality of CUI.	Container registries should be encrypted with a customer-managed key	1.1.2
System and Communications Protection	SC.3.183	Deny network communications traffic by default and allow network communications traffic by exception (i.e., deny all, permit by exception).	Container registries should not allow unrestricted network access	2.0.0

FedRAMP High

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - FedRAMP High. For more information about this compliance standard, see FedRAMP High.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Access Control	AC-4	Information Flow Enforcement	Container registries should not allow unrestricted network access	2.0.0
Access Control	AC-4	Information Flow Enforcement	Container registries should use private link	1.0.1
Access Control	AC-17	Remote Access	Container registries should use private link	1.0.1
Access Control	AC-17 (1)	Automated Monitoring / Control	Container registries should use private link	1.0.1
Risk Assessment	RA-5	Vulnerability Scanning	Container registry images should have vulnerability findings resolved	2.0.1
System and Communications Protection	SC-7	Boundary Protection	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC-7	Boundary Protection	Container registries should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC-7 (3)	Access Points	Container registries should use private link	1.0.1
System and Communications Protection	SC-12	Cryptographic Key Establishment and Management	Container registries should be encrypted with a customer-managed key	1.1.2

FedRAMP Moderate

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - FedRAMP Moderate. For more information about this compliance standard, see FedRAMP Moderate.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC-4	Information Flow Enforcement	Container registries should not allow unrestricted network access	2.0.0
Access Control	AC-4	Information Flow Enforcement	Container registries should use private link	1.0.1
Access Control	AC-17	Remote Access	Container registries should use private link	1.0.1
Access Control	AC-17 (1)	Automated Monitoring / Control	Container registries should use private link	1.0.1
Risk Assessment	RA-5	Vulnerability Scanning	Container registry images should have vulnerability findings resolved	2.0.1
System and Communications Protection	SC-7	Boundary Protection	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC-7	Boundary Protection	Container registries should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC-7 (3)	Access Points	Container registries should use private link	1.0.1
System and Communications Protection	SC-12	Cryptographic Key Establishment and Management	Container registries should be encrypted with a customer-managed key	1.1.2

HIPAA HITRUST 9.2

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - HIPAA HITRUST 9.2. For more information about this compliance standard, see HIPAA HITRUST 9.2.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Segregation in Networks	0805.01m1Organizational.12 - 01.m	The organization's security gateways (e.g. firewalls) enforce security policies and are configured to filter traffic between domains, block unauthorized access, and are used to maintain segregation between internal wired, internal wireless, and external network segments (e.g., the Internet) including DMZs and enforce access control policies for each of the domains.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview
Segregation in Networks	0806.01m2Organizational.12356 - 01.m	The organizations network is logically and physically segmented with a defined security perimeter and a graduated set of controls, including subnetworks for publicly accessible system components that are logically separated from the internal network, based on organizational requirements; and traffic is controlled based on functionality required and classification of the data/systems based on a risk assessment and their respective security requirements.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview
Segregation in Networks	0894.01m2Organizational.7 - 01.m	Networks are segregated from production-level networks when migrating physical servers, applications or data to virtualized servers.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Network Controls	0868.09m3Organizational.18 - 09.m	The organization builds a firewall configuration to restrict inbound and outbound traffic to that which is necessary for the covered data environment.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview
Network Controls	0869.09m3Organizational.19 - 09.m	The router configuration files are secured and synchronized.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview
Network Controls	0870.09m3Organizational.20 - 09.m	Access to all proxies is denied, except for those hosts, ports, and services that are explicitly required.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview
Network Controls	0871.09m3Organizational.22 - 09.m	Authoritative DNS servers are segregated into internal and external roles.	[Preview]: Container Registry should use a virtual network service endpoint	1.0.0-preview

New Zealand ISM Restricted

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - New Zealand ISM Restricted. For more information about this compliance standard, see New Zealand ISM Restricted.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Information security monitoring	ISM-4	6.2.6 Resolving vulnerabilities	Container registry images should have vulnerability findings resolved	2.0.1
Infrastructure	INF-9	10.8.35 Security Architecture	Container registries should use private link	1.0.1
Cryptography	CR-3	17.1.46 Reducing storage and physical transfer requirements	Container registries should be encrypted with a customer-managed key	1.1.2
Gateway security	GS-3	19.1.12 Configuration of Gateways	Container registries should not allow unrestricted network access	2.0.0

NIST SP 800-53 Rev. 5

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - NIST SP 800-53 Rev. 5. For more information about this compliance standard, see NIST SP 800-53 Rev. 5.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC-4	Information Flow Enforcement	Container registries should not allow unrestricted network access	2.0.0
Access Control	AC-4	Information Flow Enforcement	Container registries should use private link	1.0.1
Access Control	AC-17	Remote Access	Container registries should use private link	1.0.1
Access Control	AC-17 (1)	Monitoring and Control	Container registries should use private link	1.0.1
Risk Assessment	RA-5	Vulnerability Monitoring and Scanning	Container registry images should have vulnerability findings resolved	2.0.1
System and Communications Protection	SC-7	Boundary Protection	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC-7	Boundary Protection	Container registries should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Container registries should not allow unrestricted network access	2.0.0
System and Communications Protection	SC-7 (3)	Access Points	Container registries should use private link	1.0.1
System and Communications Protection	SC-12	Cryptographic Key Establishment and Management	Container registries should be encrypted with a customer-managed key	1.1.2

NZ ISM Restricted v3.5

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - NZ ISM Restricted v3.5. For more information about this compliance

standard, see NZ ISM Restricted v3.5.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Cryptography	NZISM Security Benchmark CR-3	17.1.53 Reducing storage and physical transfer requirements	Container registries should be encrypted with a customer-managed key	1.1.2
Gateway security	NZISM Security Benchmark GS-3	19.1.12 Configuration of Gateways	Container registries should not allow unrestricted network access	2.0.0
Infrastructure	NZISM Security Benchmark INF-9	10.8.35 Security Architecture	Container registries should use private link	1.0.1
Information security monitoring	NZISM Security Benchmark ISM-4	6.2.6 Resolving vulnerabilities	Container registry images should have vulnerability findings resolved	2.0.1

Reserve Bank of India - IT Framework for NBFC

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - Reserve Bank of India - IT Framework for NBFC. For more information about this compliance standard, see Reserve Bank of India - IT Framework for NBFC.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
IT Governance	RBI IT Framework 1	IT Governance-1	Container registry images should have vulnerability findings resolved	2.0.1
Information and Cyber Security	RBI IT Framework 3.3	Vulnerability Management-3.3	Container registry images should have vulnerability findings resolved	2.0.1

RMIT Malaysia

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - RMIT Malaysia. For more information about this compliance standard, see RMIT Malaysia.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Network Resilience	RMiT 10.33	Network Resilience - 10.33	Configure Container registries to disable public network access	1.0.0

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Network Resilience	RMiT 10.33	Network Resilience - 10.33	Container registries should not allow unrestricted network access	2.0.0
Network Resilience	RMiT 10.33	Network Resilience - 10.33	Public network access should be disabled for Container registries	1.0.0
Cloud Services	RMiT 10.53	Cloud Services - 10.53	Container registries should be encrypted with a customer-managed key	1.1.2
Data Loss Prevention (DLP)	RMiT 11.15	Data Loss Prevention (DLP) - 11.15	Configure Container registries to disable public network access	1.0.0
Cybersecurity Operations	RMiT 11.8	Cybersecurity Operations - 11.8	Container registry images should have vulnerability findings resolved	2.0.1

Next steps

- Learn more about Azure Policy Regulatory Compliance.
- See the built-ins on the Azure Policy GitHub repo.

Azure Container Registry service tiers

8/22/2022 • 6 minutes to read • Edit Online

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

TIER	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, private link with private endpoints to restrict access to the registry.

The Basic, Standard, and Premium tiers all provide the same programmatic capabilities. They also all benefit from image storage managed entirely by Azure. Choosing a higher-level tier 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.

Service tier features and limits

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

RESOURCE	BASIC	STANDARD	PREMIUM
Included storage[1] (GiB)	10	100	500
Storage limit (TiB)	20	20	20
Maximum image layer size (GiB)	200	200	200
Maximum manifest size (MiB)	4	4	4
ReadOps per minute[2, 3]	1,000	3,000	10,000

RESOURCE	BASIC	STANDARD	PREMIUM
WriteOps per minute[2, 4]	100	500	2,000
Download bandwidth[2] (Mbps)	30	60	100
Upload bandwidth [2] (Mbps)	10	20	50
Webhooks	2	10	500
Geo-replication	N/A	N/A	Supported
Availability zones	N/A	N/A	Supported
Content trust	N/A	N/A	Supported
Private link with private endpoints	N/A	N/A	Supported
• Private endpoints	N/A	N/A	200
Public IP network rules	N/A	N/A	100
Service endpoint VNet access	N/A	N/A	Preview
• Virtual network rules	N/A	N/A	100
Customer-managed keys	N/A	N/A	Supported
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[5]	N/A	N/A	500

[1] Storage included in the daily rate for each tier. Additional storage may be used, up to the registry storage limit, at an additional daily rate per GiB. For rate information, see Azure Container Registry pricing. If you need storage beyond the registry storage limit, please contact Azure Support.

[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.

[5] Individual *actions* of `content/delete`, `content/read`, `content/write`, `metadata/read`, `metadata/write`

corresponds to the limit of Repositories per scope map.

Registry throughput and throttling

Throughput

When generating a high rate of registry operations, use the service tier's limits for read and write operations and bandwidth as a guide for expected maximum throughput. These limits affect data-plane operations including listing, deleting, pushing, and pulling images and other artifacts.

To estimate the throughput of image pulls and pushes specifically, consider the registry limits and these factors:

- Number and size of image layers
- Reuse of layers or base images across images
- additional API calls that might be required for each pull or push

For details, see documentation for the Docker HTTP API V2.

When evaluating or troubleshooting registry throughput, also consider the configuration of your client environment:

- your Docker daemon configuration for concurrent operations
- your network connection to the registry's data endpoint (or endpoints, if your registry is geo-replicated).

If you experience issues with throughput to your registry, see Troubleshoot registry performance.

Example

Pushing a single 133 MB `nginx:latest` image to an Azure container registry requires multiple read and write operations for the image's five layers:

- Read operations to read the image manifest, if it exists in the registry
- Write operations to write the configuration blob of the image
- Write operations to write the image manifest

Throttling

You may experience throttling of pull or push operations when the registry determines the rate of requests exceeds the limits allowed for the registry's service tier. You may see an HTTP 429 error similar to `Too many requests`.

Throttling could occur temporarily when you generate a burst of image pull or push operations in a very short period, even when the average rate of read and write operations is within registry limits. You may need to implement retry logic with some backoff in your code or reduce the maximum rate of requests to the registry.

Show registry usage

Use the az acr show-usage command in the Azure CLI, Get-AzContainerRegistryUsage in Azure PowerShell, or the List Usages REST API, to get a snapshot of your registry's current consumption of storage and other resources, compared with the limits for that registry's service tier. Storage usage also appears on the registry's **Overview** page in the portal.

Usage information helps you make decisions about changing the service tier when your registry nears a limit. This information also helps you manage consumption.

> **NOTE**
>
> The registry's storage usage should only be used as a guide and may not reflect recent registry operations. Monitor the registry's StorageUsed metric for up-to-date data.

Depending on your registry's service tier, usage information includes some or all of the following, along with the limit in that tier:

- Storage consumed in bytes[1]
- Number of webhooks
- Number of geo-replications (includes the home replica)
- Number of private endpoints
- Number of IP access rules
- Number of virtual network rules

[1]In a geo-replicated registry, storage usage is shown for the home region. Multiply by the number of replications for total storage consumed.

Changing tiers

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

There is no registry downtime or impact on registry operations when you move between service tiers.

Azure CLI

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

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

Azure PowerShell

To move between service tiers in Azure PowerShell, use the Update-AzContainerRegistry cmdlet. For example, to switch to Premium:

```
Update-AzContainerRegistry -ResourceGroupName myResourceGroup -Name myContainerRegistry -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 service tiers, 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 artifacts

8/22/2022 • 5 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 and related artifacts. Docker Hub is an example of a public container registry that serves as a general catalog of Docker container images. Azure Container Registry provides users with direct control of their container content, with integrated authentication, geo-replication supporting global distribution and reliability for network-close deployments, virtual network configuration with Private Link, tag locking, and many other enhanced features.

In addition to Docker-compatible container images, Azure Container Registry supports a range of content artifacts including Helm charts and Open Container Initiative (OCI) image formats.

Repository

A *repository* is a collection of container images or other artifacts in a registry that have 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 identify related repositories and artifact ownership in your organization by using forward slash-delimited names. However, the registry manages all repositories independently, not as a hierarchy. 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*

Repository names can only include lowercase alphanumeric characters, periods, dashes, underscores, and forward slashes.

For complete repository naming rules, see the Open Container Initiative Distribution Specification.

Artifact

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. The tag `latest` is used by default if you don't provide one in your Docker commands.

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.

For tag naming rules, see the Docker documentation.

Layer

Container images and artifacts are made up of one or more *layers*. Different artifact types define layers differently. For example, in a Docker container image, each layer corresponds to a line in the Dockerfile that defines the image:

Artifacts in a registry share common layers, increasing storage efficiency. For example, several images in different repositories might have a common ASP.NET Core base layer, but only one copy of that layer is stored in the registry. Layer sharing also optimizes layer distribution to nodes, with multiple artifacts sharing common layers. If an image already on a node includes the ASP.NET Core 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 content is pushed, uniquely identifies the artifacts and specifies the layers.

A basic manifest for a Linux `hello-world` image looks similar to the following:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
    "mediaType": "application/vnd.docker.container.image.v1+json",
    "size": 1510,
    "digest": "sha256:fbf289e99eb9bca977dae136fbe2a82b6b7d4c372474c9235adc1741675f587e"
  },
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 977,
      "digest": "sha256:2c930d010525941c1d56ec53b97bd057a67ae1865eebf042686d2a2d18271ced"
    }
  ]
}
```

You can list the manifests for a repository with the Azure CLI command az acr manifest list-metadata:

```
az acr manifest list-metadata --name <repositoryName> --registry <acrName>
```

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

```
az acr manifest list-metadata --name acr-helloworld --registry myregistry
```

```
[
  {
    "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 artifact's layer data is identical to that of another artifact. 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 artifact 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 pushed later to the registry.

> **IMPORTANT**
> If you repeatedly push modified artifacts with identical tags, you might create "orphans"--artifacts 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.

Addressing an artifact

To address a registry artifact for push and pull operations with Docker or other client tools, combine the fully qualified registry name, repository name (including namespace path if applicable), and an artifact tag or manifest digest. See previous sections for explanations of these terms.

Address by tag: `[loginServerUrl]/[repository][:tag]`

Address by digest: `[loginServerUrl]/[repository@sha256][:digest]`

When using Docker or other client tools to pull or push artifacts to an Azure container registry, use the registry's fully qualified URL, also called the *login server* name. In the Azure cloud, the fully qualified URL of an Azure container registry is in the format `myregistry.azurecr.io` (all lowercase).

> **NOTE**
> - You can't specify a port number in the registry login server URL, such as `myregistry.azurecr.io:443`.
> - The tag `latest` is used by default if you don't provide a tag in your command.

Push by tag

Examples:

```
docker push myregistry.azurecr.io/samples/myimage:20210106
```

```
docker push myregistry.azurecr.io/marketing/email-sender
```

Pull by tag

Example:

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

Pull by manifest digest

Example:

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

Next steps

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

Learn how to push and pull images from Azure Container Registry.

Container image storage in Azure Container Registry

8/22/2022 • 2 minutes to read • Edit Online

Every Basic, Standard, and Premium Azure container registry benefits from advanced Azure storage features including encryption-at-rest. The following sections describe the features and limits of image storage in Azure Container Registry (ACR).

Encryption-at-rest

All container images and other artifacts 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. Optionally apply an extra encryption layer with a customer-managed key.

Regional storage

Azure Container Registry stores data in the region where the registry is created, to help customers meet data residency and compliance requirements. In all regions except Brazil South and Southeast Asia, Azure may also store registry data in a paired region in the same geography. In the Brazil South and Southeast Asia regions, registry data is always confined to the region, to accommodate data residency requirements for those regions.

If a regional outage occurs, the registry data may become unavailable and is not automatically recovered. Customers who wish to have their registry data stored in multiple regions for better performance across different geographies or who wish to have resiliency in the event of a regional outage should enable geo-replication.

Geo-replication

For scenarios requiring 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 regional failure. Geo-replication provides other benefits, too, like network-close image storage for faster pushes and pulls in distributed development or deployment scenarios.

Zone redundancy

To help create a resilient and high-availability Azure container registry, optionally enable zone redundancy in select Azure regions. A feature of the Premium service tier, zone redundancy uses Azure availability zones to replicate your registry to a minimum of three separate zones in each enabled region. Combine geo-replication and zone redundancy to enhance both the reliability and performance of a registry.

Scalable storage

Azure Container Registry allows you to create as many repositories, images, layers, or tags as you need, up to the registry storage limit.

High numbers of repositories and tags can affect the performance of your registry. Periodically delete unused repositories, tags, and images as part of your registry maintenance routine, and optionally set a retention policy for untagged manifests. Deleted registry resources such as 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 Basic, Standard, and Premium container registries, see Azure Container Registry service tiers.

Content formats supported in Azure Container Registry

8/22/2022 • 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 multi-architecture images under a single `image:tag` reference

OCI images

Azure Container Registry supports images that meet the Open Container Initiative (OCI) Image Format Specification, including the optional image index 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 examples, see:

- Push and pull an OCI artifact using an Azure container registry
- Push and pull Helm charts to 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 3 is recommended. See Push and pull Helm charts to an Azure container registry.

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

8/22/2022 • 4 minutes to read • Edit Online

When pushing 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

8/22/2022 • 8 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, and tag names can be used across multiple regions
- Improve performance and reliability of regional deployments with network-close registry access
- Reduce data transfer costs by pulling image layers from a local, replicated registry in the same or nearby region as your container host
- Single management of a registry across multiple regions
- Registry resilience if a regional outage occurs

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.
- If you want to move a registry to a different Azure region, instead of geo-replicating the registry, see Manually move a container registry to another region.

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 use the same image URL: `contoso.azurecr.io/public/products/web:1.2`
- Push to a single registry, while ACR automatically manages the geo-replication. ACR only replicates unique layers, reducing data transfer across regions.
- Configure regional webhooks to notify you of events in specific replicas.
- Provide a highly available registry that is resilient to regional outages.

Azure Container Registry also supports availability zones to create a resilient and high availability Azure container registry within an Azure region. The combination of availability zones for redundancy within a region, and geo-replication across multiple regions, enhances both the reliability and performance of a registry.

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. 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 that is closest to you in terms of network latency.
- 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 registry, 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.
- To serve blobs representing content layers, Azure Container Registry uses data endpoints. You can enable dedicated data endpoints for your registry in each of your registry's geo-replicated regions. These endpoints allow configuration of tightly scoped firewall access rules. For troubleshooting purposes, you can optionally disable routing to a replication while maintaining replicated data.
- If you configure a private link for your registry using private endpoints in a virtual network, dedicated data endpoints in each of the geo-replicated regions are enabled by default.

Considerations for high availability

- For high availability and resiliency, we recommend creating a registry in a region that supports enabling zone redundancy. Enabling zone redundancy in each replica region is also recommended.
- If an outage occurs in the registry's home region (the region where it was created) or one of its replica regions, a geo-replicated registry remains available for data plane operations such as pushing or pulling container images.
- If the registry's home region becomes unavailable, you may be unable to carry out registry management operations including configuring network rules, enabling availability zones, and managing replicas.
- To plan for high availablity of a geo-replicated registry encrypted with a customer-managed key stored in an Azure key vault, review the guidance for key vault failover and redundancy.

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.

To use the Azure CLI to delete a replica of *myregistry* in the East US region:

```
az acr replication delete --name eastus --registry myregistry
```

Geo-replication pricing

Geo-replication is a feature of the Premium service tier 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.

Temporarily disable routing to replication

To troubleshoot operations with a geo-replicated registry, you might want to temporarily disable Traffic Manager routing to one or more replications. Starting in Azure CLI version 2.8, you can configure a `--region-endpoint-enabled` option (preview) when you create or update a replicated region. When you set a replication's `--region-endpoint-enabled` option to `false`, Traffic Manager no longer routes docker push or pull requests to that region. By default, routing to all replications is enabled, and data synchronization across all replications takes place whether routing is enabled or disabled.

To disable routing to an existing replication, first run az acr replication list to list the replications in the registry. Then, run az acr replication update and set `--region-endpoint-enabled false` for a specific replication. For example, to configure the setting for the *westus* replication in *myregistry*:

```
# Show names of existing replications
az acr replication list --registry --output table

# Disable routing to replication
az acr replication update --name westus \
  --registry myregistry --resource-group MyResourceGroup \
  --region-endpoint-enabled false
```

To restore routing to a replication:

```
az acr replication update --name westus \
  --registry myregistry --resource-group MyResourceGroup \
  --region-endpoint-enabled true
```

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

Enable zone redundancy in Azure Container Registry for resiliency and high availability

8/22/2022 • 6 minutes to read • Edit Online

In addition to geo-replication, which replicates registry data across one or more Azure regions to provide availability and reduce latency for regional operations, Azure Container Registry supports optional *zone redundancy*. Zone redundancy provides resiliency and high availability to a registry or replication resource (replica) in a specific region.

This article shows how to set up a zone-redundant container registry or replica by using the Azure CLI, Azure portal, or Azure Resource Manager template.

Zone redundancy is a feature of the Premium container registry service tier. For information about registry service tiers and limits, see Azure Container Registry service tiers.

Regional Support

- ACR Availability Zones are supported in the following regions:

AMERICAS	EUROPE	AFRICA	ASIA PACIFIC
Brazil South Canada Central Central US East US East US 2 South Central US US Government Virginia West US 2 West US 3	France Central Germany West Central North Europe Norway East West Europe UK South	South Africa North	Australia East Central India Japan East Korea Central Southeast Asia East Asia

- Region conversions to availability zones aren't currently supported. To enable availability zone support in a region, the registry must either be created in the desired region, with availability zone support enabled, or a replicated region must be added with availability zone support enabled.
- A registry with an AZ-enabled stamp creates a home region replication with an AZ-enabled stamp by default. The AZ stamp can't be disabled once it's enabled.
- The home region replication represents the home region registry. It helps to view and manage the availability zone properties and can't be deleted.
- The availability zone is per region, once the replications are created, their states cannot be changed, except by deleting and re-creating the replications.
- Zone redundancy can't be disabled in a region.
- ACR Tasks doesn't yet support availability zones.

About zone redundancy

Use Azure availability zones to create a resilient and high availability Azure container registry within an Azure region. For example, organizations can set up a zone-redundant Azure container registry with other supported Azure resources to meet data residency or other compliance requirements, while providing high availability

within a region.

Azure Container Registry also supports geo-replication, which replicates the service across multiple regions, enabling redundancy and locality to compute resources in other locations. The combination of availability zones for redundancy within a region, and geo-replication across multiple regions, enhances both the reliability and performance of a registry.

Availability zones are unique physical locations within an Azure region. To ensure resiliency, there's a minimum of three separate zones in all enabled regions. Each zone has one or more datacenters equipped with independent power, cooling, and networking. When configured for zone redundancy, a registry (or a registry replica in a different region) is replicated across all availability zones in the region, keeping it available if there are datacenter failures.

Create a zone-redundant registry - CLI

To use the Azure CLI to enable zone redundancy, you need Azure CLI version 2.17.0 or later, or Azure Cloud Shell. If you need to install or upgrade, see Install Azure CLI.

Create a resource group

If needed, run the az group create command to create a resource group for the registry.

```
az group create --name <resource-group-name> --location <location>
```

Create zone-enabled registry

Run the az acr create command to create a zone-redundant registry in the Premium service tier. Choose a region that supports availability zones for Azure Container Registry. In the following example, zone redundancy is enabled in the *eastus* region. See the `az acr create` command help for more registry options.

```
az acr create \
  --resource-group <resource-group-name> \
  --name <container-registry-name> \
  --location eastus \
  --zone-redundancy enabled \
  --sku Premium
```

In the command output, note the `zoneRedundancy` property for the registry. When enabled, the registry is zone redundant:

```
{
 [...]
"zoneRedundancy": "Enabled",
}
```

Create zone-redundant replication

Run the az acr replication create command to create a zone-redundant registry replica in a region that supports availability zones for Azure Container Registry, such as *westus2*.

```
az acr replication create \
  --location westus2 \
  --resource-group <resource-group-name> \
  --registry <container-registry-name> \
  --zone-redundancy enabled
```

In the command output, note the `zoneRedundancy` property for the replica. When enabled, the replica is zone

redundant:

```
{
 [...]
"zoneRedundancy": "Enabled",
}
```

Create a zone-redundant registry - portal

1. Sign in to the Azure portal at https://portal.azure.com.
2. Select **Create a resource** > **Containers** > **Container Registry**.
3. In the **Basics** tab, select or create a resource group, and enter a unique registry name.
4. In **Location**, select a region that supports zone redundancy for Azure Container Registry, such as *East US*.
5. In **SKU**, select **Premium**.
6. In **Availability zones**, select **Enabled**.
7. Optionally, configure additional registry settings, and then select **Review + create**.
8. Select **Create** to deploy the registry instance.

To create a zone-redundant replication:

1. Navigate to your Premium tier container registry, and select **Replications**.

2. On the map that appears, select a green hexagon in a region that supports zone redundancy for Azure Container Registry, such as **West US 2**. Or select **+ Add**.

3. In the **Create replication** window, confirm the **Location**. In **Availability zones**, select **Enabled**, and then select **Create**.

Create a zone-redundant registry - template

Create a resource group

If needed, run the az group create command to create a resource group for the registry in a region that supports availability zones for Azure Container Registry, such as *eastus*. This region is used by the template to set the registry location.

```
az group create --name <resource-group-name> --location eastus
```

Deploy the template

You can use the following Resource Manager template to create a zone-redundant, geo-replicated registry. The template by default enables zone redundancy in the registry and a regional replica.

Copy the following contents to a new file and save it using a filename such as `registryZone.json`.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      "acrName": {
        "type": "string",
        "defaultValue": "[concat('acr', uniqueString(resourceGroup().id))]",
        "minLength": 5,
        "maxLength": 50,
        "metadata": {
          "description": "Globally unique name of your Azure Container Registry"
        }
      },
      "location": {
        "type": "string",
        "defaultValue": "[resourceGroup().location]",
        "metadata": {
          "description": "Location for registry home replica."
        }
      },
      "acrSku": {
        "type": "string",
```

```
        "defaultValue": "Premium",
        "allowedValues": [
          "Premium"
        ],
        "metadata": {
          "description": "Tier of your Azure Container Registry. Geo-replication and zone redundancy require
Premium SKU."
        }
      },
      "acrZoneRedundancy": {
        "type": "string",
        "defaultValue": "Enabled",
        "metadata": {
          "description": "Enable zone redundancy of registry's home replica. Requires registry location to
support availability zones."
        }
      },
      "acrReplicaLocation": {
        "type": "string",
        "metadata": {
          "description": "Short name for registry replica location."
        }
      },
      "acrReplicaZoneRedundancy": {
        "type": "string",
        "defaultValue": "Enabled",
        "metadata": {
          "description": "Enable zone redundancy of registry replica. Requires replica location to support
availability zones."
        }
      }
    },
    "resources": [
      {
        "comments": "Container registry for storing docker images",
        "type": "Microsoft.ContainerRegistry/registries",
        "apiVersion": "2020-11-01-preview",
        "name": "[parameters('acrName')]",
        "location": "[parameters('location')]",
        "sku": {
          "name": "[parameters('acrSku')]",
          "tier": "[parameters('acrSku')]"
        },
        "tags": {
          "displayName": "Container Registry",
          "container.registry": "[parameters('acrName')]"
        },
        "properties": {
          "adminUserEnabled": "[parameters('acrAdminUserEnabled')]",
          "zoneRedundancy": "[parameters('acrZoneRedundancy')]"
        }
      },
      {
        "type": "Microsoft.ContainerRegistry/registries/replications",
        "apiVersion": "2020-11-01-preview",
        "name": "[concat(parameters('acrName'), '/', parameters('acrReplicaLocation'))]",
        "location": "[parameters('acrReplicaLocation')]",
          "dependsOn": [
          "[resourceId('Microsoft.ContainerRegistry/registries/', parameters('acrName'))]"
        ],
        "properties": {
          "zoneRedundancy": "[parameters('acrReplicaZoneRedundancy')]"
        }
      }
    ],
    "outputs": {
      "acrLoginServer": {
        "value": "
[reference(resourceId('Microsoft.ContainerRegistry/registries', parameters('acrName')), '2019-12-01-
```

```
[reference(resourceId('Microsoft.ContainerRegistry/registries',parameters('acrName')), 2019-12-01-
preview').loginServer]",
        "type": "string"
      }
    }
  }
```

Run the following az deployment group create command to create the registry using the preceding template file. Where indicated, provide:

- a unique registry name, or deploy the template without parameters and it will create a unique name for you
- a location for the replica that supports availability zones, such as *westus2*

```
az deployment group create \
  --resource-group <resource-group-name> \
  --template-file registryZone.json \
  --parameters acrName=<registry-name> acrReplicaLocation=<replica-location>
```

In the command output, note the `zoneRedundancy` property for the registry and the replica. When enabled, each resource is zone redundant:

```
{
 [...]
"zoneRedundancy": "Enabled",
}
```

Next steps

- Learn more about regions that support availability zones.
- Learn more about building for reliability in Azure.

Best practices for Azure Container Registry

8/22/2022 • 5 minutes to read • Edit Online

By following these best practices, you can help maximize the performance and cost-effective use of your private registry in Azure to store and deploy container images and other artifacts.

For background on registry concepts, see About registries, repositories, and images. 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. You can also configure regional webhooks to notify you of events in specific replicas such as when images are pushed.

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

Maximize pull performance

In addition to placing images close to your deployments, characteristics of your images themselves can impact pull performance.

- **Image size** - Minimize the sizes of your images by removing unnecessary layers or reducing the size of layers. One way to reduce image size is to use the multi-stage Docker build approach to include only the necessary runtime components.

  Also check whether your image can include a lighter base OS image. And if you use a deployment environment such as Azure Container Instances that caches certain base images, check whether you can swap an image layer for one of the cached images.

- **Number of layers** - Balance the number of layers used. If you have too few, you don't benefit from layer reuse and caching on the host. Too many, and your deployment environment spends more time pulling and decompressing. Five to 10 layers is optimal.

Also choose a service tier of Azure Container Registry that meets your performance needs. The Premium tier provides the greatest bandwidth and highest rate of concurrent read and write operations when you have high-volume deployments.

Repository namespaces

By using 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. However, the registry manages all repositories independently, not as a hierarchy.

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 and authorization

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 these and other Azure Container Registry authentication scenarios, see Authenticate with an Azure container registry.

Azure Container Registry supports security practices in your organization to distribute duties and privileges to different identities. Using role-based access control, assign appropriate permissions to different users, service principals, or other identities that perform different registry operations. For example, assign push permissions to a service principal used in a build pipeline and assign pull permissions to a different identity used for deployment. Create tokens for fine-grained, time-limited access to specific repositories.

Manage registry size

The storage constraints of each container registry service tier are intended to align with a typical scenario: **Basic** for getting started, **Standard** for most 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 consumption of storage and other resources in your registry:

```
az acr show-usage --resource-group myResourceGroup --name myregistry --output table
```

Sample output:

```
NAME                        LIMIT         CURRENT VALUE    UNIT
--------------------------  ------------  ---------------  ------
Size                        536870912000  215629144        Bytes
Webhooks                    500           1                Count
Geo-replications            -1            3                Count
IPRules                     100           1                Count
VNetRules                   100           0                Count
PrivateEndpointConnections  10            0                Count
```

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

> **NOTE**
>
> In a geo-replicated registry, storage usage is shown for the home region. Multiply by the number of replications for total registry storage consumed.

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. You can also set a retention policy for untagged manifests.

Next steps

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

For recommendations to improve the security posture of your container registries, see Azure Security Baseline for Azure Container Registry.

Multi-architecture images in your Azure container registry

8/22/2022 • 3 minutes to read • Edit Online

This article introduces *multi-architecture* (*multi-arch*) images and how you can use Azure Container Registry features to help create, store, and use them.

A multi-arch image is a type of container image that may combine variants for different architectures, and sometimes for different operating systems. When running an image with multi-architecture support, container clients will automatically select an image variant that matches your OS and architecture.

Manifests and manifest lists

Multi-arch images are based on image manifests and manifest lists.

Manifest

Each container image is represented by a manifest. A manifest is a JSON file that uniquely identifies the image, referencing its layers and their corresponding sizes.

A basic manifest for a Linux `hello-world` image looks similar to the following:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
      "mediaType": "application/vnd.docker.container.image.v1+json",
      "size": 1510,
      "digest": "sha256:fbf289e99eb9bca977dae136fbe2a82b6b7d4c372474c9235adc1741675f587e"
    },
  "layers": [
      {
        "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
        "size": 977,
        "digest": "sha256:2c930d010525941c1d56ec53b97bd057a67ae1865eebf042686d2a2d18271ced"
      }
    ]
}
```

You can view a manifest in Azure Container Registry using the Azure portal or tools such as the az acr manifest list-metadata command in the Azure CLI.

Manifest list

A *manifest list* for a multi-arch image (known more generally as an image index for OCI images) is a collection (index) of images, and you create one by specifying one or more image names. It includes details about each of the images such as the supported OS and architecture, size, and manifest digest. The manifest list can be used in the same way as an image name in `docker pull` and `docker run` commands.

The `docker` CLI manages manifests and manifest lists using the docker manifest command.

> **NOTE**
> Currently, the `docker manifest` command and subcommands are experimental. See the Docker documentation for details about using experimental commands.

You can view a manifest list using the `docker manifest inspect` command. The following is the output for the multi-arch image `mcr.microsoft.com/mcr/hello-world:latest`, which has three manifests: two for Linux OS architectures and one for a Windows architecture.

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
  "manifests": [
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 524,
      "digest": "sha256:83c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e899a",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 525,
      "digest": "sha256:873612c5503f3f1674f315c67089dee577d8cc6afc18565e0b4183ae355fb343",
      "platform": {
        "architecture": "arm64",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 1124,
      "digest": "sha256:b791ad98d505abb8c9618868fc43c74aa94d08f1d7afe37d19647c0030905cae",
      "platform": {
        "architecture": "amd64",
        "os": "windows",
        "os.version": "10.0.17763.1697"
      }
    }
  ]
}
```

When a multi-arch manifest list is stored in Azure Container Registry, you can also view the manifest list using the Azure portal or with tools such as the az acr manifest list-metadata command.

Import a multi-arch image

An existing multi-arch image can be imported to an Azure container registry using the az acr import command. The image import syntax is the same as with a single-architecture image. Like import of a single-architecture image, import of a multi-arch image doesn't use Docker commands.

For details, see Import container images to a container registry.

Push a multi-arch image

When you have build workflows to create container images for different architectures, follow these steps to push a multi-arch image to your Azure container registry.

1. Tag and push each architecture-specific image to your container registry. The following example assumes two Linux architectures: arm64 and amd64.

```
docker tag myimage:arm64 \
  myregistry.azurecr.io/multi-arch-samples/myimage:arm64

docker push myregistry.azurecr.io/multi-arch-samples/myimage:arm64

docker tag myimage:amd64 \
  myregistry.azurecr.io/multi-arch-samples/myimage:amd64

docker push myregistry.azurecr.io/multi-arch-samples/myimage:amd64
```

2. Run `docker manifest create` to create a manifest list to combine the preceding images into a multi-arch image.

```
docker manifest create myregistry.azurecr.io/multi-arch-samples/myimage:multi \
 myregistry.azurecr.io/multi-arch-samples/myimage:arm64 \
 myregistry.azurecr.io/multi-arch-samples/myimage:amd64
```

3. Push the manifest to your container registry using `docker manifest push`:

```
docker manifest push myregistry.azurecr.io/multi-arch-samples/myimage:multi
```

4. Use the `docker manifest inspect` command to view the manifest list. An example of command output is shown in a preceding section.

After you push the multi-arch manifest to your registry, work with the multi-arch image the same way that you do with a single-architecture image. For example, pull the image using `docker pull`, and use az acr repository commands to view tags, manifests, and other properties of the image.

Build and push a multi-arch image

Using features of ACR Tasks, you can build and push a multi-arch image to your Azure container registry. For example, define a multi-step task in a YAML file that builds a Linux multi-arch image.

The following example assumes that you have separate Dockerfiles for two architectures, arm64 and amd64. It builds and pushes the architecture-specific images, then creates and pushes a multi-arch manifest that has the `latest` tag:

```
version: v1.1.0

steps:
- build: -t {{.Run.Registry}}/multi-arch-samples/myimage:{{.Run.ID}}-amd64 -f dockerfile.arm64 .
- build: -t {{.Run.Registry}}/multi-arch-samples/myyimage:{{.Run.ID}}-arm64 -f dockerfile.amd64 .
- push:
    - {{.Run.Registry}}/multi-arch-samples/myimage:{{.Run.ID}}-arm64
    - {{.Run.Registry}}/multi-arch-samples/myimage:{{.Run.ID}}-amd64
- cmd: >
    docker manifest create
    {{.Run.Registry}}/multi-arch-samples/myimage:latest
    {{.Run.Registry}}/multi-arch-samples/myimage:{{.Run.ID}}-arm64
    {{.Run.Registry}}/multi-arch-samples/myimage:{{.Run.ID}}-amd64
- cmd: docker manifest push --purge {{.Run.Registry}}/multi-arch-samples/myimage:latest
- cmd: docker manifest inspect {{.Run.Registry}}/multi-arch-samples/myimage:latest
```

Next steps

- Use Azure Pipelines to build container images for different architectures.
- Learn about building multi-platform images using the experimental Docker buildx plug-in.

Manage public content with Azure Container Registry

8/22/2022 • 5 minutes to read • Edit Online

This article is an overview of practices and workflows to use a local registry such as an Azure container registry to maintain copies of public content, such as container images in Docker Hub.

Risks with public content

Your environment may have dependencies on public content such as public container images, Helm charts, Open Policy Agent (OPA) policies, or other artifacts. For example, you might run nginx for service routing or `docker build FROM alpine` by pulling images directly from Docker Hub or another public registry.

Without proper controls, having dependencies on public registry content can introduce risks to your image development and deployment workflows. To mitigate the risks, keep local copies of public content when possible. For details, see the Open Container Initiative blog.

Authenticate with Docker Hub

As a first step, if you currently pull public images from Docker Hub as part of a build or deployment workflow, we recommend that you authenticate using a Docker Hub account instead of making an anonymous pull request.

When making frequent anonymous pull requests you might see Docker errors similar to `ERROR: toomanyrequests: Too Many Requests.` or `You have reached your pull rate limit.` Authenticate to Docker Hub to prevent these errors.

> **NOTE**
>
> Effective November 2, 2020, download rate limits apply to anonymous and authenticated requests to Docker Hub from Docker Free Plan accounts and are enforced by IP address and Docker ID, respectively.
>
> When estimating your number of pull requests, take into account that when using cloud provider services or working behind a corporate NAT, multiple users will be presented to Docker Hub in aggregate as a subset of IP addresses. Adding Docker paid account authentication to requests made to Docker Hub will avoid potential service disruptions due to rate-limit throttling.
>
> For details, see Docker pricing and subscriptions and the Docker Terms of Service.

Docker Hub access token

Docker Hub supports personal access tokens as alternatives to a Docker password when authenticating to Docker Hub. Tokens are recommended for automated services that pull images from Docker Hub. You can generate multiple tokens for different users or services, and revoke tokens when no longer needed.

To authenticate with `docker login` using a token, omit the password on the command line. When prompted for a password, enter the token instead. If you enabled two-factor authentication for your Docker Hub account, you must use a personal access token when logging in from the Docker CLI.

Authenticate from Azure services

Several Azure services including App Service and Azure Container Instances support pulling images from public registries such as Docker Hub for container deployments. If you need to deploy an image from Docker Hub, we

recommend that you configure settings to authenticate using a Docker Hub account. Examples:

App Service

- **Image source**: Docker Hub
- **Repository access**: Private
- **Login**: <Docker Hub username>
- **Password**: <Docker Hub token>

For details, see Docker Hub authenticated pulls on App Service.

Azure Container Instances

- **Image source**: Docker Hub or other registry
- **Image type**: Private
- **Image registry login server**: docker.io
- **Image registry user name**: <Docker Hub username>
- **Image registry password**: <Docker Hub token>
- **Image**: docker.io/<repo name>:<tag>

Import images to an Azure container registry

To begin managing copies of public images, you can create an Azure container registry if you don't already have one. Create a registry using the Azure CLI, Azure portal, Azure PowerShell, or other tools.

- Azure CLI
- Azure PowerShell

As a recommended one-time step, import base images and other public content to your Azure container registry. The az acr import command in the Azure CLI supports image import from public registries such as Docker Hub and Microsoft Container Registry and from other private container registries.

`az acr import` doesn't require a local Docker installation. You can run it with a local installation of the Azure CLI or directly in Azure Cloud Shell. It supports images of any OS type, multi-architecture images, or OCI artifacts such as Helm charts.

Depending on your organization's needs, you can import to a dedicated registry or a repository in a shared registry.

```
az acr import \
  --name myregistry \
  --source docker.io/library/hello-world:latest \
  --image hello-world:latest \
  --username <Docker Hub username> \
  --password <Docker Hub token>
```

Update image references

Developers of application images should ensure that their code references local content under their control.

- Update image references to use the private registry. For example, update a `FROM baseimage:v1` statement in a Dockerfile to `FROM myregistry.azurecr.io/mybaseimage:v1`
- Configure credentials or an authentication mechanism to use the private registry. The exact mechanism depends on the tools you use to access the registry and how you manage user access.
  - If you use a Kubernetes cluster or Azure Kubernetes Service to access the registry, see the

authentication scenarios.

- Learn more about options to authenticate with an Azure container registry.

Automate application image updates

Expanding on image import, set up an Azure Container Registry task to automate application image builds when base images are updated. An automated build task can track both base image updates and source code updates.

For a detailed example, see How to consume and maintain public content with Azure Container Registry Tasks.

> **NOTE**
> A single preconfigured task can automatically rebuild every application image that references a dependent base image.

Next steps

- Learn more about ACR Tasks to build, run, push, and patch container images in Azure.
- See How to consume and maintain public content with Azure Container Registry Tasks for an automated gating workflow to update base images to your environment.
- See the ACR Tasks tutorials for more examples to automate image builds and updates.

Push your first image to your Azure container registry using the Docker CLI

8/22/2022 • 4 minutes to read • Edit Online

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

In the following steps, you download a public Nginx image, 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, the Azure CLI, or Azure PowerShell.
- **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.

- Azure CLI
- Azure PowerShell

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*, log into the Azure CLI and then authenticate to your registry:

```
az login
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.

> **NOTE**
>
> - You might want to use Visual Studio Code with Docker extension for a faster and more convenient login.

> **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 a public Nginx image

First, pull a public Nginx image to your local computer. This example pulls an image from Microsoft Container Registry.

```
docker pull mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
```

Run the container locally

Execute the 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 mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
```

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 mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine 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
```

- Azure CLI
- Azure PowerShell

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
```

Recommendations

Here you can find more information on the authentication options.

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

8/22/2022 • 4 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.

Linux or macOS

```
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
```

Windows

```
.\oras.exe push myregistry.azurecr.io/samples/artifact:1.0 ^
    --manifest-config NUL: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
```

Example: Build Docker image from OCI artifact

Source code and binaries to build a container image can be stored as OCI artifacts in an Azure container registry. You can reference a source artifact as the build context for an ACR task. This example shows how to store a Dockerfile as an OCI artifact and then reference the artifact to build a container image.

For example, create a one-line Dockerfile:

```
echo "FROM mcr.microsoft.com/hello-world" > hello-world.dockerfile
```

Log in to the destination container registry.

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

Create and push a new OCI artifact to the destination registry by using the `oras push` command. This example sets the default media type for the artifact.

```
oras push myregistry.azurecr.io/dockerfile:1.0 hello-world.dockerfile
```

Run the az acr build command to build the hello-world image using the new artifact as build context:

```
az acr build --registry myregistry --image builds/hello-world:v1 \
  --file hello-world.dockerfile \
  oci://myregistry.azurecr.io/dockerfile: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

8/22/2022 • 8 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 repositories in an Azure container registry, using Helm 3 commands and storing charts as OCI artifacts. 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. You can also store an existing Helm chart from another Helm repo.

> **IMPORTANT**
> This article has been updated with Helm 3 commands. Helm 3.7 includes changes to Helm CLI commands and OCI support introduced in earlier versions of Helm 3. By design `helm` moves forward with version. We recommend to use 3.7.2 or later.

Helm 3 or Helm 2?

To store, manage, and install Helm charts, you use commands in the Helm CLI. Major Helm releases include Helm 3 and Helm 2. For details on the version differences, see the version FAQ.

Helm 3 should be used to host Helm charts in Azure Container Registry. With Helm 3, you:

- Can store and manage Helm charts in repositories in an Azure container registry
- Store Helm charts in your registry as OCI artifacts. Azure Container Registry provides GA support for OCI artifacts, including Helm charts.
- Authenticate with your registry using the `helm registry login` or `az acr login` command.
- Use `helm` commands to push, pull, and manage Helm charts in a registry
- Use `helm install` to install charts to a Kubernetes cluster from the registry.

Feature support

Azure Container Registry supports specific Helm chart management features depending on whether you are using Helm 3 (current) or Helm 2 (deprecated).

FEATURE	HELM 2	HELM 3
Manage charts using `az acr helm` commands	✔	
Store charts as OCI artifacts		✔
Manage charts using `az acr repository` commands and the **Repositories** blade in Azure portal		✔

> **NOTE**
> As of Helm 3, az acr helm commands for use with the Helm 2 client are being deprecated. A minimum of 3 months' notice will be provided in advance of command removal.

Chart version compatibility

The following Helm chart versions can be stored in Azure Container Registry and are installable by the Helm 2 and Helm 3 clients.

VERSION	HELM 2	HELM 3
apiVersion v1	✔	✔
apiVersion v2		✔

Migrate from Helm 2 to Helm 3

If you've previously stored and deployed charts using Helm 2 and Azure Container Registry, we recommend migrating to Helm 3. See:

- Migrating Helm 2 to 3 in the Helm documentation.
- Migrate your registry to store Helm OCI artifacts, later in this article

Prerequisites

The following resources are needed for the scenario in this article:

- **An Azure container registry** in your Azure subscription. If needed, create a registry using the Azure portal or the Azure CLI.
- **Helm client version 3.7 or later** - Run `helm version` to find your current version. For more information on how to install and upgrade Helm, see Installing Helm. If you upgrade from an earlier version of Helm 3, review the release notes.
- **A Kubernetes cluster** where you will install a Helm chart. If needed, create an AKS cluster [using the Azure CLI][./learn/quick-kubernetes-deploy-cli], [using Azure PowerShell][./learn/quick-kubernetes-deploy-powershell], or [using the Azure portal][./learn/quick-kubernetes-deploy-portal].
- **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.

Set up Helm client

Use the `helm version` command to verify that you have installed Helm 3:

```
helm version
```

> **NOTE**
> The version indicated must be at least 3.8.0, as OCI support in earlier versions was experimental.

Set the following environment variables for the target registry. The ACR_NAME is the registry resource name. If the ACR registry url is myregistry.azurecr.io, set the ACR_NAME to myregistry

```
ACR_NAME=<container-registry-name>
```

Create a sample chart

Create a test chart using the following commands:

```
mkdir helmtest

cd helmtest
helm create hello-world
```

As a basic example, change directory to the `templates` folder and first delete the contents there:

```
cd hello-world/templates
rm -rf *
```

In the `templates` folder, create a file called `configmap.yaml`, by running the following command:

```
cat <<EOF > configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: hello-world-configmap
data:
  myvalue: "Hello World"
EOF
```

For more about creating and running this example, see Getting Started in the Helm Docs.

Save chart to local archive

Change directory to the `hello-world` subdirectory. Then, run `helm package` to save the chart to a local archive.

In the following example, the chart is saved with the name and version in `Chart.yaml`.

```
cd ..
helm package .
```

Output is similar to:

```
Successfully packaged chart and saved it to: /my/path/hello-world-0.1.0.tgz
```

Authenticate with the registry

Run `helm registry login` to authenticate with the registry. You may pass registry credentials appropriate for your scenario, such as service principal credentials, user identity, or a repository-scoped token.

- Authenticate with an Azure Active Directory service principal with pull and push permissions (AcrPush role) to the registry.

```
SERVICE_PRINCIPAL_NAME=<acr-helm-sp>
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)
PASSWORD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME \
          --scopes $(az acr show --name $ACR_NAME --query id --output tsv) \
           --role acrpush \
          --query "password" --output tsv)
USER_NAME=$(az ad sp list --display-name $SERVICE_PRINCIPAL_NAME --query "[].appId" --output tsv)
```

- Authenticate with your individual Azure AD identity to push and pull Helm charts using an AD token.

```
USER_NAME="00000000-0000-0000-0000-000000000000"
PASSWORD=$(az acr login --name $ACR_NAME --expose-token --output tsv --query accessToken)
```

- Authenticate with a repository scoped token (Preview).

```
USER_NAME="helm-token"
PASSWORD=$(az acr token create -n $USER_NAME \
                  -r $ACR_NAME \
                  --scope-map _repositories_admin \
                  --only-show-errors \
                  --query "credentials.passwords[0].value" -o tsv)
```

- Then supply the credentials to `helm registry login`.

```
helm registry login $ACR_NAME.azurecr.io \
  --username $USER_NAME \
  --password $PASSWORD
```

Push chart to registry as OCI artifact

Run the `helm push` command in the Helm 3 CLI to push the chart archive to the fully qualified target repository. Separate the words in the chart names and use only lower case letters and numbers. In the following example, the target repository namespace is `helm/hello-world`, and the chart is tagged `0.1.0`:

```
helm push hello-world-0.1.0.tgz oci://$ACR_NAME.azurecr.io/helm
```

After a successful push, output is similar to:

```
Pushed: <registry>.azurecr.io/helm/hello-world:0.1.0
digest: sha256:5899db028dcf96aeaabdadfa5899db02589b2899b025899b059db02
```

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 $ACR_NAME \
  --repository helm/hello-world
```

Output is similar to:

```
{
  "changeableAttributes": {
    "deleteEnabled": true,
    "listEnabled": true,
    "readEnabled": true,
    "writeEnabled": true
  },
  "createdTime": "2021-10-05T12:11:37.6701689Z",
  "imageName": "helm/hello-world",
  "lastUpdateTime": "2021-10-05T12:11:37.7637082Z",
  "manifestCount": 1,
  "registry": "mycontainerregistry.azurecr.io",
  "tagCount": 1
}
```

Run the az acr manifest list-metadata command to see details of the chart stored in the repository. For example:

```
az acr manifest list-metadata \
  --registry $ACR_NAME \
  --name helm/hello-world
```

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": "2021-10-05T12:11:37.7167893Z",
    "digest": "sha256:0c03b71c225c3ddff53660258ea16ca7412b53b1f6811bf769d8c85a1f0663ee",
    "imageSize": 3301,
    "lastUpdateTime": "2021-10-05T12:11:37.7167893Z",
    "mediaType": "application/vnd.oci.image.manifest.v1+json",
    "tags": [
      "0.1.0"
    ]
```

Install Helm chart

Run `helm install` to install the Helm chart you pushed to the registry. The chart tag is passed using the `--version` parameter. Specify a release name such as *myhelmtest*, or pass the `--generate-name` parameter. For example:

```
helm install myhelmtest oci://$ACR_NAME.azurecr.io/helm/hello-world --version 0.1.0
```

Output after successful chart installation is similar to:

```
NAME: myhelmtest
LAST DEPLOYED: Tue Oct  4 16:59:51 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

To verify the installation, run the `helm get manifest` command.

```
helm get manifest myhelmtest
```

The command returns the YAML data in your `configmap.yaml` template file.

Run `helm uninstall` to uninstall the chart release on your cluster:

```
helm uninstall myhelmtest
```

Pull chart to local archive

You can optionally pull a chart from the container registry to a local archive using `helm pull`. The chart tag is passed using the `--version` parameter. If a local archive exists at the current path, this command overwrites it.

```
helm pull oci://$ACR_NAME.azurecr.io/helm/hello-world --version 0.1.0
```

Delete chart from the registry

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

```
az acr repository delete --name $ACR_NAME --image helm/hello-world:0.1.0
```

Migrate your registry to store Helm OCI artifacts

If you previously set up your Azure container registry as a chart repository using Helm 2 and the `az acr helm` commands, we recommend that you upgrade to the Helm 3 client. Then, follow these steps to store the charts as OCI artifacts in your registry.

> **IMPORTANT**
>
> - After you complete migration from a Helm 2-style (index.yaml-based) chart repository to OCI artifact repositories, use the Helm CLI and `az acr repository` commands to manage the charts. See previous sections in this article.
> - The Helm OCI artifact repositories are not discoverable using Helm commands such as `helm search` and `helm repo list`. For more information about Helm commands used to store charts as OCI artifacts, see the Helm documentation.

Enable OCI support

Ensure that you are using the Helm 3 client:

```
helm version
```

Enable OCI support in the Helm 3 client. Currently, this support is experimental and subject to change.

```
export HELM_EXPERIMENTAL_OCI=1
```

List current charts

List the charts currently stored in the registry, here named *myregistry*:

```
helm search repo myregistry
```

Output shows the charts and chart versions:

```
NAME                            CHART VERSION   APP VERSION     DESCRIPTION
myregistry/ingress-nginx        3.20.1          0.43.0          Ingress controller for Kubernetes...
myregistry/wordpress            9.0.3           5.3.2           Web publishing platform for building...
[...]
```

Pull chart archives locally

For each chart in the repo, pull the chart archive locally, and take note of the filename:

```
helm pull myregisry/ingress-nginx
ls *.tgz
```

A local chart archive such as `ingress-nginx-3.20.1.tgz` is created.

Push charts as OCI artifacts to registry

Login to the registry:

```
az acr login --name $ACR_NAME
```

Push each chart archive to the registry. Example:

```
helm push ingress-nginx-3.20.1.tgz oci://$ACR_NAME.azurecr.io/helm
```

After pushing a chart, confirm it is stored in the registry:

```
az acr repository list --name $ACR_NAME
```

After pushing all of the charts, optionally remove the Helm 2-style chart repository from the registry. Doing so reduces storage in your registry:

```
helm repo remove $ACR_NAME
```

Next steps

- For more information on how to create and deploy Helm charts, see Developing Helm charts.
- Learn more about installing applications with Helm in Azure Kubernetes Service (AKS).
- Helm charts can be used as part of the container build process. For more information, see Use Azure Container Registry Tasks.

Push and pull supply chain artifacts using Azure Registry (Preview)

8/22/2022 • 7 minutes to read • Edit Online

Use an Azure container registry to store and manage a graph of artifacts, including signatures, software bill of materials (SBoM), security scan results or other types.

To demonstrate this capability, this article shows how to use the OCI Registry as Storage (ORAS) tool to push and pull a graph of artifacts to an Azure container registry.

ORAS Artifacts support is a preview feature and subject to limitations. It requires zone redundancy, which is available in the Premium service tier. For information about registry service tiers and limits, see Azure Container Registry service tiers.

Prerequisites

- **ORAS CLI** - The ORAS CLI enables push, discover, pull of artifacts to an ORAS Artifacts enabled registry.
- **Azure CLI** - To create an identity, list and delete repositories, you need a local installation of the Azure CLI. Version 2.29.1 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 complete the walkthrough, a container image is referenced. You can use Docker installed locally to build and push a container image, or reference an existing container image. Docker provides packages that easily configure Docker on any macOS, Windows, or Linux system.

Preview limitations

ORAS Artifacts support is not available in the government or China clouds, but available in all other regions.

ORAS installation

Download and install a preview ORAS release for your operating system. See ORAS Install instructions for how to extract and install the file for your operating system, referencing an Alpha.1 preview build from the ORAS GitHub repo

Configure a registry

Configure environment variables to easily copy/paste commands into your shell. The commands can be run in the Azure Cloud Shell

```
ACR_NAME=myregistry
REGISTRY=$ACR_NAME.azurecr.io
REPO=net-monitor
TAG=v1
IMAGE=$REGISTRY/${REPO}:$TAG
```

Create a resource group

If needed, run the az group create command to create a resource group for the registry.

```
az group create --name $ACR_NAME --location southcentralus
```

Create ORAS Artifact enabled registry

Preview support for ORAS Artifacts requires Zone Redundancy, which requires a Premium service tier, in the South Central US region. Run the az acr create command to create an ORAS Artifacts enabled registry. See the `az acr create` command help for more registry options.

```
az acr create \
  --resource-group $ACR_NAME \
  --name $ACR_NAME \
  --zone-redundancy enabled \
  --sku Premium \
  --output jsonc
```

In the command output, note the `zoneRedundancy` property for the registry. When enabled, the registry is zone redundant, and ORAS Artifact enabled:

```
{
  [...]
  "zoneRedundancy": "Enabled",
}
```

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.

```
az login
az acr login --name $ACR_NAME
```

> **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.

Sign in with ORAS

This section shows options to sign into the registry. Choose the method appropriate for your environment.

Run `oras login` to authenticate with the registry. You may pass registry credentials appropriate for your scenario, such as service principal credentials, user identity, or a repository-scoped token (preview).

- Authenticate with your individual Azure AD identity to use an AD token.

```
USER_NAME="00000000-0000-0000-0000-000000000000"
PASSWORD=$(az acr login --name $ACR_NAME --expose-token --output tsv --query accessToken)
```

- Authenticate with a repository scoped token (Preview) to use non-AD based tokens.

```
USER_NAME="oras-token"
PASSWORD=$(az acr token create -n $USER_NAME \
                  -r $ACR_NAME \
                  --repository $REPO content/write \
                  --only-show-errors \
                  --query "credentials.passwords[0].value" -o tsv)
```

- Authenticate with an Azure Active Directory service principal with pull and push permissions (AcrPush role) to the registry.

```
SERVICE_PRINCIPAL_NAME="oras-sp"
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)
PASSWORD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME \
          --scopes $(az acr show --name $ACR_NAME --query id --output tsv) \
           --role acrpush \
          --query "password" --output tsv)
USER_NAME=$(az ad sp list --display-name $SERVICE_PRINCIPAL_NAME --query "[].appId" --output tsv)
```

Sign in with ORAS

Supply the credentials to `oras login`.

```
oras login $REGISTRY \
  --username $USER_NAME \
  --password $PASSWORD
```

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

Push a container image

This example associates a graph of artifacts to a container image. Build and push a container image, or reference an existing image in the registry.

```
docker build -t $IMAGE https://github.com/wabbit-networks/net-monitor.git#main
docker push $IMAGE
```

Create a sample signature to the container image

```
echo '{"artifact": "'${IMAGE}'", "signature": "pat hancock"}' > signature.json
```

Push a signature to the registry, as a reference to the container image

The ORAS command pushes the signature to a repository, referencing another artifact through the `subject` parameter. The `--artifact-type` provides for differentiating artifacts, similar to file extensions that enable different file types. One or more files can be pushed by specifying `file:mediaType`

```
oras push $REGISTRY/$REPO \
    --artifact-type 'signature/example' \
    --subject $IMAGE \
    ./signature.json:application/json
```

For more information on oras push, see ORAS documentation.

Push a multi-file artifact as a reference

Create some documentation around an artifact

```
echo 'Readme Content' > readme.md
echo 'Detailed Content' > readme-details.md
```

Push the multi-file artifact as a reference

```
oras push $REGISTRY/$REPO \
    --artifact-type 'readme/example' \
    --subject $IMAGE \
    ./readme.md:application/markdown \
    ./readme-details.md:application/markdown
```

Discovering artifact references

The ORAS Artifacts Specification defines a [referrers API][oras-artifacts-referrers] for discovering references to a `subject` artifact. The `oras discover` command can show the list of references to the container image.

Using `oras discover`, view the graph of artifacts now stored in the registry

```
oras discover -o tree $IMAGE
```

The output shows the beginning of a graph of artifacts, where the signature and docs are viewed as children of the container image

```
myregistry.azurecr.io/net-monitor:v1
├── signature/example
│   └── sha256:555ea91f39e7fb30c06f3b7aa483663f067f2950dcb...
└── readme/example
    └── sha256:1a118663d1085e229ff1b2d4d89b5f6d67911f22e55...
```

Creating a deep graphs of artifacts

The ORAS Artifacts specification enables deep graphs, enabling signed software bill of materials (SBoM) and

other artifact types.

Create a sample SBoM

```
echo '{"version": "0.0.0.0", "artifact": "'${IMAGE}'", "contents": "good"}' > sbom.json
```

Push a sample SBoM to the registry

```
oras push $REGISTRY/$REPO \
  --artifact-type 'sbom/example' \
  --subject $IMAGE \
  ./sbom.json:application/json
```

Sign the SBoM

Artifacts that are pushed as references, typically do not have tags as they are considered part of the subject artifact. To push a signature to an artifact that is a child of another artifact, use the `oras discover` with `--artifact-type` filtering to find the digest.

```
SBOM_DIGEST=$(oras discover -o json \
                --artifact-type sbom/example \
                $IMAGE | jq -r ".references[0].digest")
```

Create a signature of an SBoM

```
echo '{"artifact": "'$REGISTRY/${REPO}@$SBOM_DIGEST'", "signature": "pat hancock"}' > sbom-signature.json
```

Push the SBoM signature

```
oras push $REGISTRY/$REPO \
  --artifact-type 'signature/example' \
  --subject $REGISTRY/$REPO@$SBOM_DIGEST \
  ./sbom-signature.json:application/json
```

View the graph

```
oras discover -o tree $IMAGE
```

Generates the following output:

```
myregistry.azurecr.io/net-monitor:v1
├── signature/example
│   └── sha256:555ea91f39e7fb30c06f3b7aa483663f067f2950dcb...
├── readme/example
│   └── sha256:1a118663d1085e229ff1b2d4d89b5f6d67911f22e55...
└── sbom/example
    └── sha256:4280eef9adb632b42cf200e7cd5a822a456a558e4f3142da6b...
        └── signature/example
            └── sha256:a31ab875d37eee1cca68dbb14b2009979d05594d44a075bdd7...
```

Pull a referenced artifact

To pull a referenced type, the digest of reference is discovered with the `oras discover` command

```
DOC_DIGEST=$(oras discover -o json \
              --artifact-type 'readme/example' \
              $IMAGE | jq -r ".references[0].digest")
```

Create a clean directory for downloading

```
mkdir ./download
```

Pull the docs into the download directory

```
oras pull -a -o ./download $REGISTRY/$REPO@$DOC_DIGEST
```

View the docs

```
ls ./download
```

View the repository and tag listing

ORAS Artifacts enables artifact graphs to be pushed, discovered, pulled and copied without having to assign tags. This enables a tag listing to focus on the artifacts users think about, as opposed to the signatures and SBoMs that are associated with the container images, helm charts and other artifacts.

View a list of tags

```
az acr repository show-tags \
  -n $ACR_NAME \
  --repository $REPO \
  -o jsonc
```

View a list of manifests

A repository can have a list of manifests that are both tagged and untagged

```
az acr manifest list-metadata \
  --name $REPO \
  --repository $ACR_NAME \
  --output jsonc
```

Note the container image manifests have `"tags":`

```
{
  "architecture": "amd64",
  "changeableAttributes": {
    "deleteEnabled": true,
    "listEnabled": true,
    "readEnabled": true,
    "writeEnabled": true
  },
  "configMediaType": "application/vnd.docker.container.image.v1+json",
  "createdTime": "2021-11-12T00:18:54.5123449Z",
  "digest": "sha256:a0fc570a245b09ed752c42d600ee3bb5b4f77bbd70d8898780b7ab4...",
  "imageSize": 2814446,
  "lastUpdateTime": "2021-11-12T00:18:54.5123449Z",
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "os": "linux",
  "tags": [
    "v1"
  ]
}
```

The signature is untagged, but tracked as a `oras.artifact.manifest` reference to the container image

```
{
  "changeableAttributes": {
    "deleteEnabled": true,
    "listEnabled": true,
    "readEnabled": true,
    "writeEnabled": true
  },
  "createdTime": "2021-11-12T00:19:10.987156Z",
  "digest": "sha256:555ea91f39e7fb30c06f3b7aa483663f067f2950dcbcc0b0d...",
  "imageSize": 85,
  "lastUpdateTime": "2021-11-12T00:19:10.987156Z",
  "mediaType": "application/vnd.cncf.oras.artifact.manifest.v1+json"
}
```

Delete all artifacts in the graph

Support for the ORAS Artifacts specification enables deleting the graph of artifacts associated with the root artifact. Use the az acr repository delete command to delete the signature, SBoM and the signature of the SBoM.

```
az acr repository delete \
  -n $ACR_NAME \
  -t ${REPO}:$TAG -y
```

View the remaining manifests

```
az acr manifest list-metadata \
  --name $REPO \
  --registry $ACR_NAME \
  --detail -o jsonc
```

Next steps

- Learn more about the ORAS CLI
- Learn more about ORAS Artifacts for how to push, discover, pull, copy a graph of supply chain artifacts

Build, sign, and verify container images using Notary and Azure Key Vault (Preview)

8/22/2022 • 4 minutes to read • Edit Online

The Azure Key Vault (AKV) is used to store a signing key that can be utilized by **notation** with the notation AKV plugin (azure-kv) to sign and verify container images and other artifacts. The Azure Container Registry (ACR) allows you to attach these signatures using the **az** or **oras** CLI commands.

The signed containers enable users to assure deployments are built from a trusted entity and verify artifact hasn't been tampered with since their creation. The signed artifact ensures integrity and authenticity before the user pulls an artifact into any environment and avoid attacks.

In this tutorial:

- Store a signing certificate in Azure Key Vault
- Sign a container image with notation
- Verify a container image signature with notation

Prerequisites

- Install, create and sign in to ORAS artifact enabled registry
- Create or use an Azure Key Vault
- This tutorial can be run in the Azure Cloud Shell

Install the notation CLI and AKV plugin

> **NOTE**
> The tutorial uses early released versions of notation and notation plugins.

1. Install notation with plugin support from the release version

```
# Download, extract and install
curl -Lo notation.tar.gz https://github.com/notaryproject/notation/releases/download/v0.9.0-alpha.1/notation_0.9.0-alpha.1_linux_amd64.tar.gz
tar xvzf notation.tar.gz

# Copy the notation cli to the desired bin directory in your PATH
cp ./notation /usr/local/bin
```

2. Install the notation Azure Key Vault plugin for remote signing and verification

> **NOTE**
> The plugin directory varies depending upon the operating system being used. The directory path below assumes Ubuntu. Please read the notation config article for more information.

```
# Create a directory for the plugin
mkdir -p ~/.config/notation/plugins/azure-kv

# Download the plugin
curl -Lo notation-azure-kv.tar.gz \
    https://github.com/Azure/notation-azure-kv/releases/download/v0.3.0-alpha.1/notation-azure-
kv_0.3.0-alpha.1_Linux_amd64.tar.gz

# Extract to the plugin directory
tar xvzf notation-azure-kv.tar.gz -C ~/.config/notation/plugins/azure-kv notation-azure-kv
```

3. List the available plugins and verify that the plugin is available

```
notation plugin ls
```

Configure environment variables

> **NOTE**
> For easy execution of commands in the tutorial, provide values for the Azure resources to match the existing ACR and AKV resources.

1. Configure AKV resource names

```
# Name of the existing Azure Key Vault used to store the signing keys
AKV_NAME=<your-unique-keyvault-name>
# New desired key name used to sign and verify
KEY_NAME=wabbit-networks-io
KEY_SUBJECT_NAME=wabbit-networks.io
CERT_PATH=./${KEY_NAME}.pem
```

2. Configure ACR and image resource names

```
# Name of the existing registry example: myregistry.azurecr.io
ACR_NAME=myregistry
# Existing full domain of the ACR
REGISTRY=$ACR_NAME.azurecr.io
# Container name inside ACR where image will be stored
REPO=net-monitor
TAG=v1
IMAGE=$REGISTRY/${REPO}:$TAG
# Source code directory containing Dockerfile to build
IMAGE_SOURCE=https://github.com/wabbit-networks/net-monitor.git#main
```

Store the signing certificate in AKV

If you have an existing certificate, upload it to AKV. For more information on how to use your own signing key, see the signing certificate requirements. Otherwise create an x509 self-signed certificate storing it in AKV for remote signing using the steps below.

Create a self-signed certificate (Azure CLI)

1. Create a certificate policy file

   Once the certificate policy file is executed as below, it creates a valid signing certificate compatible with **notation** in AKV. The EKU listed is for code-signing, but isn't required for notation to sign artifacts.

```
cat <<EOF > ./my_policy.json
{
    "issuerParameters": {
    "certificateTransparency": null,
    "name": "Self"
    },
    "x509CertificateProperties": {
    "ekus": [
        "1.3.6.1.5.5.7.3.3"
    ],
    "subject": "CN=${KEY_SUBJECT_NAME}",
    "validityInMonths": 12
    }
}
EOF
```

2. Create the certificate

```
az keyvault certificate create -n $KEY_NAME --vault-name $AKV_NAME -p @my_policy.json
```

3. Get the Key ID for the certificate

```
KEY_ID=$(az keyvault certificate show -n $KEY_NAME --vault-name $AKV_NAME --query 'kid' -o tsv)
```

4. Download public certificate

```
CERT_ID=$(az keyvault certificate show -n $KEY_NAME --vault-name $AKV_NAME --query 'id' -o tsv)
az keyvault certificate download --file $CERT_PATH --id $CERT_ID --encoding PEM
```

5. Add the Key ID to the keys and certs

```
notation key add --name $KEY_NAME --plugin azure-kv --id $KEY_ID
notation cert add --name $KEY_NAME $CERT_PATH
```

6. List the keys and certs to confirm

```
notation key ls
notation cert ls
```

Build and sign a container image

1. Build and push a new image with ACR Tasks

```
az acr build -r $ACR_NAME -t $IMAGE $IMAGE_SOURCE
```

2. Authenticate with your individual Azure AD identity to use an ACR token

```
export USER_NAME="00000000-0000-0000-0000-000000000000"
export PASSWORD=$(az acr login --name $ACR_NAME --expose-token --output tsv --query accessToken)
export NOTATION_PASSWORD=$PASSWORD
```

3. Sign the container image

```
notation sign --key $KEY_NAME $IMAGE
```

View the graph of artifacts with the ORAS CLI

ACR support for ORAS artifacts enables a linked graph of supply chain artifacts that can be viewed through the ORAS CLI or the Azure CLI

1. Signed images can be view with the ORAS CLI

```
oras login -u $USER_NAME -p $PASSWORD $REGISTRY
oras discover -o tree $IMAGE
```

View the graph of artifacts with the Azure CLI

1. List the manifest details for the container image

```
az acr manifest show-metadata $IMAGE -o jsonc
```

2. Generates a result, showing the `digest` representing the notary v2 signature.

```
{
  "changeableAttributes": {
    "deleteEnabled": true,
    "listEnabled": true,
    "readEnabled": true,
    "writeEnabled": true
  },
  "createdTime": "2022-05-13T23:15:54.3478293Z",
  "digest": "sha256:effba96d9b7092a0de4fa6710f6e73bf8c838e4fbd536e95de94915777b18613",
  "lastUpdateTime": "2022-05-13T23:15:54.3478293Z",
  "name": "v1",
  "quarantineState": "Passed",
  "signed": false
}
```

Verify the container image

1. The notation command can also help to ensure the container image hasn't been tampered with since build time by comparing the `sha` with what is in the registry.

```
notation verify $IMAGE
sha256:effba96d9b7092a0de4fa6710f6e73bf8c838e4fbd536e95de94915777b18613
```

The sha256 result is a successful verification of the image using the trusted certificate.

2. We can add a different local signing certificate to show how multiple certificates and verification failures work.

```
notation cert generate-test -n localcert --trust true
notation verify $IMAGE
sha256:effba96d9b7092a0de4fa6710f6e73bf8c838e4fbd536e95de94915777b18613
```

We can see that verification still passes because `notation verify` will implicitly pass with *any* certificate in its

trust store. To get a verification failure, we'll remove the certificate utilized to sign the image.

```
notation cert rm $KEY_NAME
notation verify $IMAGE
2022/06/10 11:24:30 verification failure: x509: certificate signed by unknown authority
```

Next steps

Enforce policy to only deploy signed container images to Azure Kubernetes Service (AKS) utilizing **ratify** and **gatekeeper**.

View container registry repositories in the Azure portal

8/22/2022 • 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

8/22/2022 • 10 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 and other artifacts from an existing registry:

- Import images from a public registry
- Import images or OCI artifacts including Helm 3 charts from another Azure container registry, in the same or a different Azure subscription or tenant
- 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.
- Access to the target registry doesn't have to use the registry's public endpoint.

Limitations

- The maximum number of manifests for an imported image is 50.
- The maximum layer size for an image imported from a public registry is 2 GiB.

- Azure CLI
- Azure PowerShell

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 service tier required), you can serve multiple regions with identical image and tag names from a single registry.

> **IMPORTANT**
> Changes to image import between two Azure container registries have been introduced as of January 2021:
>
> - Import to or from a network-restricted Azure container registry requires the restricted registry to **allow access by trusted services** to bypass the network. By default, the setting is enabled, allowing import. If the setting isn't enabled in a newly created registry with a private endpoint or with registry firewall rules, import will fail.
> - In an existing network-restricted Azure container registry that is used as an import source or target, enabling this network security feature is optional but recommended.

Prerequisites

- Azure CLI
- Azure PowerShell

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, or a custom role that allows the importImage action). See Azure Container Registry roles and permissions.

Import from a public registry

> **IMPORTANT**
> To import from a public registry to a network-restricted Azure container registry requires the restricted registry to **allow access by trusted services** to bypass the network.By default, the setting is enabled, allowing import. If the setting isn't enabled in a newly created registry with a private endpoint or with registry firewall rules, import will fail.

Import from Docker Hub

- Azure CLI
- Azure PowerShell

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 manifest list-metadata command:

```
az acr manifest list-metadata \
  --name hello-world \
  --registry myregistry
```

To import an artifact by digest without adding a tag:

```
az acr import \
   --name myregistry \
   --source docker.io/library/hello-world@sha256:abc123 \
   --repository hello-world
```

If you have a Docker Hub account, we recommend that you use the credentials when importing an image from Docker Hub. Pass the Docker Hub user name and the password or a personal access token as parameters to `az acr import`. The following example imports a public image from the `tensorflow` repository in Docker Hub, using Docker Hub credentials:

```
az acr import \
  --name myregistry \
  --source docker.io/tensorflow/tensorflow:latest-gpu \
  --image tensorflow:latest-gpu
  --username <Docker Hub user name>
  --password <Docker Hub token>
```

Import from Microsoft Container Registry

For example, import the `ltsc2019` Windows Server Core image from the `windows` repository in Microsoft Container Registry.

- Azure CLI
- Azure PowerShell

```
az acr import \
--name myregistry \
--source mcr.microsoft.com/windows/servercore:ltsc2019 \
--image servercore:ltsc2019
```

Import from an Azure container registry in the same AD tenant

You can import an image from an Azure container registry in the same AD tenant using integrated Azure Active Directory permissions.

- Your identity must have Azure Active Directory permissions to read from the source registry (Reader role) and to import to the target registry (Contributor role, or a custom role that allows the importImage action).
- The registry can be in the same or a different Azure subscription in the same Active Directory tenant.
- Public access to the source registry may be disabled. If public access is disabled, specify the source registry by resource ID instead of by registry login server name.
- If the source registry and/or the target registry has a private endpoint or registry firewall rules are applied, ensure that the restricted registry allows trusted services to access the network.

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.

- Azure CLI
- Azure PowerShell

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

The following example imports the `aci-helloworld:latest` image to *myregistry* from a source registry *mysourceregistry* in which access to the registry's public endpoint is disabled. Supply the resource ID of the source registry with the `--registry` parameter. Notice that the `--source` parameter specifies only the source repository and tag, not the registry login server name.

```
az acr import \
  --name myregistry \
  --source aci-helloworld:latest \
  --image aci-helloworld:latest \
  --registry /subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/sourceResourceGroup/providers/Microsoft.ContainerRegistry/registries/mysourcereg
istry
```

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

- Azure CLI
- Azure PowerShell

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 tag, not the registry login server name.

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

Import from a registry using service principal credentials

To import from a registry that you can't access using integrated Active Directory permissions, you can use service principal credentials (if available) to the source registry. 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.

- Azure CLI
- Azure PowerShell

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

Import from an Azure container registry in a different AD tenant

To import from an Azure container registry in a different Azure Active Directory tenant, specify the source registry by login server name, and provide credentials that enable pull access to the registry.

Cross-tenant import with username and password

For example, use a repository-scoped token and password, or the appID and password of an Active Directory service principal that has ACRPull access to the source registry.

- Azure CLI
- Azure PowerShell

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

Cross-tenant import with access token

To access the source registry using an identity in the source tenant that has registry permissions, you can get an access token:

- Azure CLI
- Azure PowerShell

```
# Login to Azure CLI with the identity, for example a user-assigned managed identity
az login --identity --username <identity_ID>

# Get access token returned by `az account get-access-token`
az account get-access-token
```

In the target tenant, pass the access token as a password to the `az acr import` command. The source registry is specified by login server name. Notice that no username is needed in this command:

```
az acr import \
  --name myregistry \
  --source sourceregistry.azurecr.io/sourcerrepo:tag \
  --image targetimage:tag \
  --password <access-token>
```

Import from a non-Azure private container registry

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

- Azure CLI
- Azure PowerShell

```
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.

- Azure CLI
- Azure PowerShell

- For additional image import options, see the az acr import command reference.
- Image import can help you move content to a container registry in a different Azure region, subscription, or Azure AD tenant. For more information, see Manually move a container registry to another region.
- Learn how to disable artifact export from a network-restricted container registry.

Transfer artifacts to another registry

8/22/2022 • 5 minutes to read • Edit Online

This article shows how to transfer collections of images or other registry artifacts from one Azure container registry to another registry. The source and target registries can be in the same or different subscriptions, Active Directory tenants, Azure clouds, or physically disconnected clouds.

To transfer artifacts, you create a *transfer pipeline* that replicates artifacts between two registries by using blob storage:

- Artifacts from a source registry are exported to a blob in a source storage account
- The blob is copied from the source storage account to a target storage account
- The blob in the target storage account gets imported as artifacts in the target registry. You can set up the import pipeline to trigger whenever the artifact blob updates in the target storage.

In this article, you create the prerequisite resources to create and run the transfer pipeline. The Azure CLI is used to provision the associated resources such as storage secrets. Azure CLI version 2.2.0 or later is recommended. If you need to install or upgrade the CLI, see Install Azure CLI.

This feature is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry tiers.

> **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).

Consider your use-case

Transfer is ideal for copying content between two Azure container registries in physically disconnected clouds, mediated by storage accounts in each cloud. If instead you want to copy images from container registries in connected clouds including Docker Hub and other cloud vendors, image import is recommended.

Prerequisites

- **Container registries** - You need an existing source registry with artifacts to transfer, and a target registry. ACR transfer is intended for movement across physically disconnected clouds. For testing, the source and target registries can be in the same or a different Azure subscription, Active Directory tenant, or cloud.

  If you need to create a registry, see Quickstart: Create a private container registry using the Azure CLI.

- **Storage accounts** - Create source and target storage accounts in a subscription and location of your choice. For testing purposes, you can use the same subscription or subscriptions as your source and target registries. For cross-cloud scenarios, typically you create a separate storage account in each cloud.

  If needed, create the storage accounts with the Azure CLI or other tools.

  Create a blob container for artifact transfer in each account. For example, create a container named *transfer*.

- **Key vaults** - Key vaults are needed to store SAS token secrets used to access source and target storage

accounts. Create the source and target key vaults in the same Azure subscription or subscriptions as your source and target registries. For demonstration purposes, the templates and commands used in this article also assume that the source and target key vaults are located in the same resource groups as the source and target registries, respectively. This use of common resource groups isn't required, but it simplifies the templates and commands used in this article.

If needed, create key vaults with the Azure CLI or other tools.

- **Environment variables** - For example commands in this article, set the following environment variables for the source and target environments. All examples are formatted for the Bash shell.

```
SOURCE_RG="<source-resource-group>"
TARGET_RG="<target-resource-group>"
SOURCE_KV="<source-key-vault>"
TARGET_KV="<target-key-vault>"
SOURCE_SA="<source-storage-account>"
TARGET_SA="<target-storage-account>"
```

Scenario overview

You create the following three pipeline resources for image transfer between registries. All are created using PUT operations. These resources operate on your *source* and *target* registries and storage accounts.

Storage authentication uses SAS tokens, managed as secrets in key vaults. The pipelines use managed identities to read the secrets in the vaults.

- **ExportPipeline** - Long-lasting resource that contains high-level information about the *source* registry and storage account. This information includes the source storage blob container URI and the key vault managing the source SAS token.
- **ImportPipeline** - Long-lasting resource that contains high-level information about the *target* registry and storage account. This information includes the target storage blob container URI and the key vault managing the target SAS token. An import trigger is enabled by default, so the pipeline runs automatically when an artifact blob lands in the target storage container.
- **PipelineRun** - Resource used to invoke either an ExportPipeline or ImportPipeline resource.
  - You run the ExportPipeline manually by creating a PipelineRun resource and specify the artifacts to export.
  - If an import trigger is enabled, the ImportPipeline runs automatically. It can also be run manually using a PipelineRun.
  - Currently a maximum of **50 artifacts** can be transferred with each PipelineRun.

Things to know

- The ExportPipeline and ImportPipeline will typically be in different Active Directory tenants associated with the source and destination clouds. This scenario requires separate managed identities and key vaults for the export and import resources. For testing purposes, these resources can be placed in the same cloud, sharing identities.
- By default, the ExportPipeline and ImportPipeline templates each enable a system-assigned managed identity to access key vault secrets. The ExportPipeline and ImportPipeline templates also support a user-assigned identity that you provide.

Create and store SAS keys

Transfer uses shared access signature (SAS) tokens to access the storage accounts in the source and target environments. Generate and store tokens as described in the following sections.

> **IMPORTANT**
> While ACR Transfer will work with a manually generated SAS token stored in a Keyvault Secret, for production workloads we *strongly* recommend using Keyvault Managed Storage SAS Definition Secrets instead.

Generate SAS token for export

Run the az storage container generate-sas command to generate a SAS token for the container in the source storage account, used for artifact export.

Recommended token permissions: Read, Write, List, Add.

In the following example, command output is assigned to the EXPORT_SAS environment variable, prefixed with the '?' character. Update the `--expiry` value for your environment:

```
EXPORT_SAS=?$(az storage container generate-sas \
  --name transfer \
  --account-name $SOURCE_SA \
  --expiry 2021-01-01 \
  --permissions alrw \
  --https-only \
  --output tsv)
```

Store SAS token for export

Store the SAS token in your source Azure key vault using az keyvault secret set:

```
az keyvault secret set \
  --name acrexportsas \
  --value $EXPORT_SAS \
  --vault-name $SOURCE_KV
```

Generate SAS token for import

Run the az storage container generate-sas command to generate a SAS token for the container in the target storage account, used for artifact import.

Recommended token permissions: Read, Delete, List

In the following example, command output is assigned to the IMPORT_SAS environment variable, prefixed with the '?' character. Update the `--expiry` value for your environment:

```
IMPORT_SAS=?$(az storage container generate-sas \
  --name transfer \
  --account-name $TARGET_SA \
  --expiry 2021-01-01 \
  --permissions dlr \
  --https-only \
  --output tsv)
```

Store SAS token for import

Store the SAS token in your target Azure key vault using az keyvault secret set:

```
az keyvault secret set \
  --name acrimportsas \
  --value $IMPORT_SAS \
  --vault-name $TARGET_KV
```

Next steps

- Follow one of the below tutorials to create your ACR Transfer resources. For most non-automated use-cases, we recommend using the Az CLI Extension.
  - ACR Transfer with Az CLI
  - ACR Transfer with ARM templates

ACR Transfer with Az CLI

8/22/2022 • 6 minutes to read • Edit Online

This article shows how to use the ACR Transfer feature with the acrtransfer Az CLI extension.

Complete prerequisites

Please complete the prerequisites outlined here prior to attempting the actions in this article. This means that:

- You have an existing Premium SKU Registry in both clouds.
- You have an existing Storage Account Container in both clouds.
- You have an existing Keyvault with a secret containing a valid SAS token with the necessary permissions in both clouds.
- You have a recent version of Az CLI installed in both clouds.

Install the Az CLI extension

In AzureCloud, you can install the extension with the following command:

```
az extension add --name acrtransfer
```

In AzureCloud and other clouds, you can install the blob directly from a public storage account container. The blob is hosted in the `acrtransferext` storage account, `dist` container, `acrtransfer-1.0.0-py2.py3-none-any.wh` blob. You may need to change the storage URI suffix depending on which cloud you are in. The following will install in AzureCloud:

```
az extension add --source https://acrtransferext.blob.core.windows.net/dist/acrtransfer-1.0.0-py2.py3-none-any.whl
```

Create ExportPipeline with the acrtransfer Az CLI extension

Create an ExportPipeline resource for your AzureCloud container registry using the acrtransfer Az CLI extension.

Create an export pipeline with no options and a system-assigned identity:

```
az acr export-pipeline create \
--resource-group $MyRG \
--registry $MyReg \
--name $MyPipeline \
--secret-uri https://$MyKV.vault.azure.net/secrets/$MySecret \
--storage-container-uri https://$MyStorage.blob.core.windows.net/$MyContainer
```

Create an export pipeline with all possible options and a user-assigned identity:

```
az acr export-pipeline create \
--resource-group $MyRG \
--registry $MyReg \
--name $MyPipeline \
--secret-uri https://$MyKV.vault.azure.net/secrets/$MySecret \
--storage-container-uri https://$MyStorage.blob.core.windows.net/$MyContainer \
--options OverwriteBlobs ContinueOnErrors \
--assign-identity
/subscriptions/$MySubID/resourceGroups/$MyRG/providers/Microsoft.ManagedIdentity/userAssignedIdentities/$MyI
dentity
```

Export options

The `options` property for the export pipelines supports optional boolean values. The following values are recommended:

PARAMETER	VALUE
options	OverwriteBlobs - Overwrite existing target blobs ContinueOnErrors - Continue export of remaining artifacts in the source registry if one artifact export fails.

Give the ExportPipeline identity keyvault policy access

If you created your pipeline with a user-assigned identity, simply give this user-assigned identity `secret get` access policy permissions on the keyvault.

If you created your pipeline with a system-assigned identity, you will first need to retrieve the principalId that the system has assigned to your pipeline resource.

Run the following command to retrieve your pipeline resource:

```
az acr export-pipeline show --resource-group $MyRG --registry $MyReg --name $MyPipeline
```

From this output, you will want to copy the value in the `principalId` field.

Then, you will run the following command to give this principal the appropriate `secret get` access policy permissions on your keyvault.

```
az keyvault set-policy --name $MyKeyvault --secret-permissions get --object-id $MyPrincipalID
```

Create ImportPipeline with the acrtransfer Az CLI extension

Create an ImportPipeline resource in your target container registry using the acrtransfer Az CLI extension. By default, the pipeline is enabled to create an Import PipelineRun automatically when the attached storage account container receives a new artifact blob.

Create an import pipeline with no options and a system-assigned identity:

```
az acr import-pipeline create \
--resource-group $MyRG \
--registry $MyReg \
--name $MyPipeline \
--secret-uri https://$MyKV.vault.azure.net/secrets/$MySecret \
--storage-container-uri https://$MyStorage.blob.core.windows.net/$MyContainer
```

Create an import pipeline with all possible options, source-trigger disabled, and a user-assigned identity:

```
az acr import-pipeline create \
--resource-group $MyRG \
--registry $MyReg \
--name $MyPipeline \
--secret-uri https://$MyKV.vault.azure.net/secrets/$MySecret \
--storage-container-uri https://$MyStorage.blob.core.windows.net/$MyContainer \
--options DeleteSourceBlobOnSuccess OverwriteTags ContinueOnErrors \
--assign-identity
/subscriptions/$MySubID/resourceGroups/$MyRG/providers/Microsoft.ManagedIdentity/userAssignedIdentities/$MyI
dentity \
--source-trigger-enabled False
```

Import options

The `options` property for the import pipeline supports optional boolean values. The following values are recommended:

PARAMETER	VALUE
options	OverwriteTags - Overwrite existing target tags DeleteSourceBlobOnSuccess - Delete the source storage blob after successful import to the target registry ContinueOnErrors - Continue import of remaining artifacts in the target registry if one artifact import fails.

Give the ImportPipeline identity keyvault policy access

If you created your pipeline with a user-assigned identity, simply give this user-assigned identity `secret get` access policy permissions on the keyvault.

If you created your pipeline with a system-assigned identity, you will first need to retrieve the principalId that the system has assigned to your pipeline resource.

Run the following command to retrieve your pipeline resource:

```
az acr import-pipeline show --resource-group $MyRG --registry $MyReg --name $MyPipeline
```

From this output, you will want to copy the value in the `principalId` field.

Then, you will run the following command to give this principal the appropriate `secret get` access policy on your keyvault.

```
az keyvault set-policy --name $MyKeyvault --secret-permissions get --object-id $MyPrincipalID
```

Create PipelineRun for export with the acrtransfer Az CLI extension

Create a PipelineRun resource for your container registry using the acrtransfer Az CLI extension. This resource runs the ExportPipeline resource you created previously and exports specified artifacts from your container registry as a blob to your storage account container.

Create an export pipeline-run:

```
az acr pipeline-run create \
--resource-group $MyRG \
--registry $MyReg \
--pipeline $MyPipeline \
--name $MyPipelineRun \
--pipeline-type export \
--storage-blob $MyBlob \
--artifacts hello-world:latest hello-
world@sha256:90659bf80b44ce6be8234e6ff90a1ac34acbeb826903b02cfa0da11c82cbc042 \
--force-redeploy
```

If redeploying a PipelineRun resource with identical properties, you must use the --force-redeploy flag.

It can take several minutes for artifacts to export. When deployment completes successfully, verify artifact export by listing the exported blob in the container of the source storage account. For example, run the az storage blob list command:

```
az storage blob list --account-name $MyStorageAccount --container $MyContainer --output table
```

Transfer blob across domain

In most use-cases, you will now use a Cross Domain Solution or other method to transfer your blob from the storage account in your source domain (the storage account associated with your export pipeline) to the storage account in your target domain (the storage account associated with your import pipeline). At this point, we will assume that the blob has arrived in the target domain storage account associated with your import pipeline.

Trigger ImportPipeline resource

If you did not use the `--source-trigger-enabled False` parameter when creating your import pipeline, the pipeline will be triggered within 15 minutes after the blob arrives in the storage account container. It can take several minutes for artifacts to import. When the import completes successfully, verify artifact import by listing the tags on the repository you are importing in the target container registry. For example, run az acr repository show-tags:

```
az acr repository show-tags --name $MyRegistry --repository $MyRepository
```

> **NOTE**
> Source Trigger will only import blobs that have a Last Modified time within the last 60 days. If you intend to use Source Trigger to import blobs older than that, please refresh the Last Modified time of the blobs by add blob metadata to them or else import them with manually created pipeline runs.

If you did use the the `--source-trigger-enabled False` parameter when creating your ImportPipeline, you will need to create a PipelineRun manually, as shown in the following section.

Create PipelineRun for import with the acrtransfer Az CLI extension

Create a PipelineRun resource for your container registry using the acrtransfer Az CLI extension. This resource runs the ImportPipeline resource you created previously and imports specified blobs from your storage account into your container registry.

Create an import pipeline-run:

```
az acr pipeline-run create \
--resource-group $MyRG \
--registry $MyReg \
--pipeline $MyPipeline \
--name $MyPipelineRun \
--pipeline-type import \
--storage-blob $MyBlob \
--force-redeploy
```

If redeploying a PipelineRun resource with identical properties, you must use the --force-redeploy flag.

It can take several minutes for artifacts to import. When the import completes successfully, verify artifact import by listing the repositories in the target container registry. For example, run az acr repository show-tags:

```
az acr repository show-tags --name $MyRegistry --repository $MyRepository
```

Delete ACR Transfer resources

Delete an ExportPipeline:

```
az acr export-pipeline delete --resource-group $MyRG --registry $MyReg --name $MyPipeline
```

Delete an ImportPipeline:

```
az acr import-pipeline delete --resource-group $MyRG --registry $MyReg --name $MyPipeline
```

Delete a PipelineRun resource. Note that this does not reverse the action taken by the PipelineRun. This is more like deleting the log of the PipelineRun.

```
az acr pipeline-run delete --resource-group $MyRG --registry $MyReg --name $MyPipelineRun
```

ACR Transfer troubleshooting

View ACR Transfer Troubleshooting for troubleshooting guidance.

Next steps

- Learn how to block creation of export pipelines from a network-restricted container registry.

ACR Transfer with ARM templates

8/22/2022 • 8 minutes to read • Edit Online

Complete Prerequisites

Please complete the prerequisites outlined here prior to attempting the actions in this article. This means that:

- You have an existing Premium SKU Registry in both clouds.
- You have an existing Storage Account Container in both clouds.
- You have an existing Keyvault with a secret containing a valid SAS token with the necessary permissions in both clouds.
- You have a recent version of Az CLI installed in both clouds.

Consider using the Az CLI extension

For most non-automated use-cases, we recommend using the Az CLI Extension if possible. You can view documentation for the Az CLI Extension here.

Create ExportPipeline with Resource Manager

Create an ExportPipeline resource for your source container registry using Azure Resource Manager template deployment.

Copy ExportPipeline Resource Manager template files to a local folder.

Enter the following parameter values in the file `azuredeploy.parameters.json`:

PARAMETER	VALUE
registryName	Name of your source container registry
exportPipelineName	Name you choose for the export pipeline
targetUri	URI of the storage container in your source environment (the target of the export pipeline). Example: `https://sourcestorage.blob.core.windows.net/transfer`
keyVaultName	Name of the source key vault
sasTokenSecretName	Name of the SAS token secret in the source key vault Example: acrexportsas

Export options

The `options` property for the export pipelines supports optional boolean values. The following values are recommended:

PARAMETER	VALUE
options	OverwriteBlobs - Overwrite existing target blobs ContinueOnErrors - Continue export of remaining artifacts in the source registry if one artifact export fails.

Create the resource

Run az deployment group create to create a resource named *exportPipeline* as shown in the following examples. By default, with the first option, the example template enables a system-assigned identity in the ExportPipeline resource.

With the second option, you can provide the resource with a user-assigned identity. (Creation of the user-assigned identity not shown.)

With either option, the template configures the identity to access the SAS token in the export key vault.

Option 1: Create resource and enable system-assigned identity

```
az deployment group create \
  --resource-group $SOURCE_RG \
  --template-file azuredeploy.json \
  --name exportPipeline \
  --parameters azuredeploy.parameters.json
```

Option 2: Create resource and provide user-assigned identity

In this command, provide the resource ID of the user-assigned identity as an additional parameter.

```
az deployment group create \
  --resource-group $SOURCE_RG \
  --template-file azuredeploy.json \
  --name exportPipeline \
  --parameters azuredeploy.parameters.json \
  --parameters userAssignedIdentity="/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myUse
rAssignedIdentity"
```

In the command output, take note of the resource ID (`id`) of the pipeline. You can store this value in an environment variable for later use by running the az deployment group show. For example:

```
EXPORT_RES_ID=$(az deployment group show \
  --resource-group $SOURCE_RG \
  --name exportPipeline \
  --query 'properties.outputResources[1].id' \
  --output tsv)
```

Create ImportPipeline with Resource Manager

Create an ImportPipeline resource in your target container registry using Azure Resource Manager template deployment. By default, the pipeline is enabled to import automatically when the storage account in the target environment has an artifact blob.

Copy ImportPipeline Resource Manager template files to a local folder.

Enter the following parameter values in the file `azuredeploy.parameters.json`:

PARAMETER	VALUE
registryName	Name of your target container registry
importPipelineName	Name you choose for the import pipeline
sourceUri	URI of the storage container in your target environment (the source for the import pipeline). Example: `https://targetstorage.blob.core.windows.net/transfer`
keyVaultName	Name of the target key vault
sasTokenSecretName	Name of the SAS token secret in the target key vault Example: acr importsas

Import options

The `options` property for the import pipeline supports optional boolean values. The following values are recommended:

PARAMETER	VALUE
options	OverwriteTags - Overwrite existing target tags DeleteSourceBlobOnSuccess - Delete the source storage blob after successful import to the target registry ContinueOnErrors - Continue import of remaining artifacts in the target registry if one artifact import fails.

Create the resource

Run az deployment group create to create a resource named *importPipeline* as shown in the following examples. By default, with the first option, the example template enables a system-assigned identity in the ImportPipeline resource.

With the second option, you can provide the resource with a user-assigned identity. (Creation of the user-assigned identity not shown.)

With either option, the template configures the identity to access the SAS token in the import key vault.

Option 1: Create resource and enable system-assigned identity

```
az deployment group create \
  --resource-group $TARGET_RG \
  --template-file azuredeploy.json \
  --name importPipeline \
  --parameters azuredeploy.parameters.json
```

Option 2: Create resource and provide user-assigned identity

In this command, provide the resource ID of the user-assigned identity as an additional parameter.

```
az deployment group create \
  --resource-group $TARGET_RG \
  --template-file azuredeploy.json \
  --name importPipeline \
  --parameters azuredeploy.parameters.json \
  --parameters userAssignedIdentity="/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myUse
rAssignedIdentity"
```

If you plan to run the import manually, take note of the resource ID (`id`) of the pipeline. You can store this value in an environment variable for later use by running the az deployment group show command. For example:

```
IMPORT_RES_ID=$(az deployment group show \
  --resource-group $TARGET_RG \
  --name importPipeline \
  --query 'properties.outputResources[1].id' \
  --output tsv)
```

Create PipelineRun for export with Resource Manager

Create a PipelineRun resource for your source container registry using Azure Resource Manager template deployment. This resource runs the ExportPipeline resource you created previously, and exports specified artifacts from your container registry as a blob to your source storage account.

Copy PipelineRun Resource Manager template files to a local folder.

Enter the following parameter values in the file `azuredeploy.parameters.json`:

PARAMETER	VALUE
registryName	Name of your source container registry
pipelineRunName	Name you choose for the run
pipelineResourceId	Resource ID of the export pipeline. Example: `/subscriptions/<subscriptionID>/resourceGroups/<resourceGroupName>/providers/Microsoft.ContainerRegi`
targetName	Name you choose for the artifacts blob exported to your source storage account, such as *myblob*
artifacts	Array of source artifacts to transfer, as tags or manifest digests Example: `[samples/hello-world:v1", "samples/nginx:v1" , "myrepository@sha256:0a2e01852872..."]`

If redeploying a PipelineRun resource with identical properties, you must also use the forceUpdateTag property.

Run az deployment group create to create the PipelineRun resource. The following example names the deployment *exportPipelineRun*.

```
az deployment group create \
  --resource-group $SOURCE_RG \
  --template-file azuredeploy.json \
  --name exportPipelineRun \
  --parameters azuredeploy.parameters.json
```

For later use, store the resource ID of the pipeline run in an environment variable:

```
EXPORT_RUN_RES_ID=$(az deployment group show \
  --resource-group $SOURCE_RG \
  --name exportPipelineRun \
  --query 'properties.outputResources[0].id' \
  --output tsv)
```

It can take several minutes for artifacts to export. When deployment completes successfully, verify artifact export by listing the exported blob in the *transfer* container of the source storage account. For example, run the az storage blob list command:

```
az storage blob list \
  --account-name $SOURCE_SA \
  --container transfer \
  --output table
```

Transfer blob (optional)

Use the AzCopy tool or other methods to transfer blob data from the source storage account to the target storage account.

For example, the following `azcopy copy` command copies myblob from the *transfer* container in the source account to the *transfer* container in the target account. If the blob exists in the target account, it's overwritten. Authentication uses SAS tokens with appropriate permissions for the source and target containers. (Steps to create tokens are not shown.)

```
azcopy copy \
  'https://<source-storage-account-name>.blob.core.windows.net/transfer/myblob'$SOURCE_SAS \
  'https://<destination-storage-account-name>.blob.core.windows.net/transfer/myblob'$TARGET_SAS \
  --overwrite true
```

Trigger ImportPipeline resource

If you enabled the `sourceTriggerStatus` parameter of the ImportPipeline (the default value), the pipeline is triggered after the blob is copied to the target storage account. It can take several minutes for artifacts to import. When the import completes successfully, verify artifact import by listing the repositories in the target container registry. For example, run az acr repository list:

```
az acr repository list --name <target-registry-name>
```

> **NOTE**
> Source Trigger will only import blobs that have a Last Modified time within the last 60 days. If you intend to use Source Trigger to import blobs older than that, please refresh the Last Modified time of the blobs by add blob metadata to them or else import them with manually created pipeline runs.

If you didn't enable the `sourceTriggerStatus` parameter of the import pipeline, run the ImportPipeline resource manually, as shown in the following section.

Create PipelineRun for import with Resource Manager (optional)

You can also use a PipelineRun resource to trigger an ImportPipeline for artifact import to your target container registry.

Copy PipelineRun Resource Manager template files to a local folder.

Enter the following parameter values in the file `azuredeploy.parameters.json`:

PARAMETER	VALUE
registryName	Name of your target container registry
pipelineRunName	Name you choose for the run
pipelineResourceId	Resource ID of the import pipeline. Example: `/subscriptions/<subscriptionID>/resourceGroups/<resourceGroupName>/providers/Microsoft.ContainerRegi`
sourceName	Name of the existing blob for exported artifacts in your storage account, such as *myblob*

If redeploying a PipelineRun resource with identical properties, you must also use the forceUpdateTag property.

Run az deployment group create to run the resource.

```
az deployment group create \
  --resource-group $TARGET_RG \
  --name importPipelineRun \
  --template-file azuredeploy.json \
  --parameters azuredeploy.parameters.json
```

For later use, store the resource ID of the pipeline run in an environment variable:

```
IMPORT_RUN_RES_ID=$(az deployment group show \
  --resource-group $TARGET_RG \
  --name importPipelineRun \
  --query 'properties.outputResources[0].id' \
  --output tsv)
```

When deployment completes successfully, verify artifact import by listing the repositories in the target container registry. For example, run az acr repository list:

```
az acr repository list --name <target-registry-name>
```

Redeploy PipelineRun resource

If redeploying a PipelineRun resource with *identical properties*, you must leverage the **forceUpdateTag** property. This property indicates that the PipelineRun resource should be recreated even if the configuration has not changed. Please ensure forceUpdateTag is different each time you redeploy the PipelineRun resource. The example below recreates a PipelineRun for export. The current datetime is used to set forceUpdateTag, thereby ensuring this property is always unique.

```
CURRENT_DATETIME=`date +"%Y-%m-%d:%T"`
```

```
az deployment group create \
  --resource-group $SOURCE_RG \
  --template-file azuredeploy.json \
  --name exportPipelineRun \
  --parameters azuredeploy.parameters.json \
  --parameters forceUpdateTag=$CURRENT_DATETIME
```

Delete pipeline resources

The following example commands use az resource delete to delete the pipeline resources created in this article. The resource IDs were previously stored in environment variables.

```
# Delete export resources
az resource delete \
--resource-group $SOURCE_RG \
--ids $EXPORT_RES_ID $EXPORT_RUN_RES_ID \
--api-version 2019-12-01-preview

# Delete import resources
az resource delete \
--resource-group $TARGET_RG \
--ids $IMPORT_RES_ID $IMPORT_RUN_RES_ID \
--api-version 2019-12-01-preview
```

ACR Transfer troubleshooting

View ACR Transfer Troubleshooting for troubleshooting guidance.

Next steps

- Learn how to block creation of export pipelines from a network-restricted container registry.

ACR Transfer troubleshooting

8/22/2022 • 3 minutes to read • Edit Online

- **Template deployment failures or errors**
  - If a pipeline run fails, look at the `pipelineRunErrorMessage` property of the run resource.
  - For common template deployment errors, see Troubleshoot ARM template deployments
- **Problems accessing storage**
  - If you see a `403 Forbidden` error from storage, you likely have a problem with your SAS token.
  - The SAS token might not currently be valid. The SAS token might be expired or the storage account keys might have changed since the SAS token was created. Verify that the SAS token is valid by attempting to use the SAS token to authenticate for access to the storage account container. For example, put an existing blob endpoint followed by the SAS token in the address bar of a new Microsoft Edge InPrivate window or upload a blob to the container with the SAS token by using `az storage blob upload`.
  - The SAS token might not have sufficient Allowed Resource Types. Verify that the SAS token has been given permissions to Service, Container, and Object under Allowed Resource Types (`srt=sco` in the SAS token).
  - The SAS token might not have sufficient permissions. For export pipelines, the required SAS token permissions are Read, Write, List, and Add. For import pipelines, the required SAS token permissions are Read, Delete, and List. (The Delete permission is required only if the import pipeline has the `DeleteSourceBlobOnSuccess` option enabled.)
  - The SAS token might not be configured to work with HTTPS only. Verify that the SAS token is configured to work with HTTPS only (`spr=https` in the SAS token).
- **Problems with export or import of storage blobs**
  - SAS token may be invalid, or may have insufficient permissions for the specified export or import run. See Problems accessing storage.
  - Existing storage blob in source storage account might not be overwritten during multiple export runs. Confirm that the OverwriteBlob option is set in the export run and the SAS token has sufficient permissions.
  - Storage blob in target storage account might not be deleted after successful import run. Confirm that the DeleteBlobOnSuccess option is set in the import run and the SAS token has sufficient permissions.
  - Storage blob not created or deleted. Confirm that container specified in export or import run exists, or specified storage blob exists for manual import run.
- **Problems with Source Trigger Imports**
  - The SAS token must have the List permission for Source Trigger imports to work.
  - Source Trigger imports will only fire if the Storage Blob has a Last Modified time within the last 60 days.
  - The Storage Blob must have a valid ContentMD5 property in order to be imported by the Source Trigger feature.
  - The Storage Blob must have the "category":"acr-transfer-blob" blob metadata in order to be imported by the Source Trigger feature. This metadata is added automatically during an Export Pipeline Run, but may be stripped when moved from storage account to storage account depending on the method of copy.
- **AzCopy issues**

  - See Troubleshoot AzCopy issues.
- **Artifacts transfer problems**
  - Not all artifacts, or none, are transferred. Confirm spelling of artifacts in export run, and name of blob in export and import runs. Confirm you are transferring a maximum of 50 artifacts.
  - Pipeline run might not have completed. An export or import run can take some time.
  - For other pipeline issues, provide the deployment correlation ID of the export run or import run to the Azure Container Registry team.
- **Problems pulling the image in a physically isolated environment**
  - If you see errors regarding foreign layers or attempts to resolve mcr.microsoft.com when attempting to pull an image in a physically isolated environment, your image manifest likely has non-distributable layers. Due to the nature of a physically isolated environment, these images will often fail to pull. You can confirm that this is the case by checking the image manifest for any references to external registries. If this is the case, you will need to push the non-distributable layers to your public cloud ACR prior to deploying an export pipeline-run for that image. For guidance on how to do this, see How do I push non-distributable layers to a registry?

Lock a container image in an Azure container registry

8/22/2022 • 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 myimage:tag \
    --output jsonc
```

Lock an image by tag

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

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

Lock an image by manifest digest

To lock a *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 manifest list-metadata command.)

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

Lock a repository

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

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

Protect an image or repository from deletion

Protect an image from deletion

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

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

Protect a repository from deletion

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

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

Prevent read operations on an image or repository

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

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

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

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

Unlock an image or repository

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

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

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

```
az acr repository update \
    --name myregistry --repository myrepo \
    --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.

Disable export of artifacts from an Azure container registry

8/22/2022 • 3 minutes to read • Edit Online

To prevent registry users in an organization from maliciously or accidentally leaking artifacts outside a virtual network, you can configure the registry's *export policy* to disable exports.

Export policy is a property introduced in API version **2021-06-01-preview** for Premium container registries. The `exportPolicy` property, when its status is set to `disabled`, blocks export of artifacts from a network-restricted registry when a user attempts to:

- Import the registry's artifacts to another Azure container registry
- Create a registry export pipeline to transfer artifacts to another container registry

> **NOTE**
> Disabling export of artifacts does not prevent authorized users' access to the registry within the virtual network to pull artifacts or perform other data-plane operations. To audit this use, we recommend that you configure diagnostic settings to monitor registry operations.

Prerequisites

- A Premium container registry configured with a private endpoint.
- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

Other requirements to disable exports

- **Disable public network access** - To disable export of artifacts, public access to the registry must also be disabled (the registry's `publicNetworkAccess` property must be set to `disabled`). You can disable public network access to the registry before disabling export or disable it at the same time.

  By disabling access to the registry's public endpoint, you ensure that registry operations are permitted only within the virtual network. Public access to the registry to pull artifacts and perform other

operations is prohibited.

- **Remove export pipelines** - Before setting the registry's `exportPolicy` status to `disabled`, delete any existing export pipelines configured on the registry. If a pipeline is configured, you can't change the `exportPolicy` status.

Disable exportPolicy for an existing registry

When you create a registry, the `exportPolicy` status is set to `enabled` by default, which permits artifacts to be exported. You can update the status to `disabled` using an ARM template or the `az resource update` command.

ARM template

Include the following JSON to update the `exportPolicy` status and set the `publicNetworkAccess` property to `disabled`. Learn more about deploying resources with ARM templates.

```
{
[...]
"resources": [
    {
    "type": "Microsoft.ContainerRegistry/registries",
    "apiVersion": "2021-06-01-preview",
    "name": "myregistry",
    [...]
    "properties": {
      "publicNetworkAccess": "disabled",
      "policies": {
        "exportPolicy": {
          "status": "disabled"
         }
      }
      }
    }
]
[...]
}
```

Azure CLI

Run az resource update to set the `exportPolicy` status in an existing registry to `disabled`. Substitute the names of your registry and resource group.

As shown in this example, when disabling the `exportPolicy` property, also set the `publicNetworkAccess` property to `disabled`.

```
az resource update --resource-group myResourceGroup \
    --name myregistry \
    --resource-type "Microsoft.ContainerRegistry/registries" \
    --api-version "2021-06-01-preview" \
    --set "properties.policies.exportPolicy.status=disabled" \
    --set "properties.publicNetworkAccess=disabled"
```

The output shows that the export policy status is disabled.

```
{
  "id": "/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxxx/resourceGroups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/myregistry",
  "identity": null,
  "kind": null,
  "location": "centralus",
  "managedBy": null,
  "name": "myregistry",
  "plan": null,
  "properties": {
    [...]
    "policies": {
      "exportPolicy": {
        "status": "disabled"
      },
      "quarantinePolicy": {
        "status": "disabled"
      },
      "retentionPolicy": {
        "days": 7,
        "lastUpdatedTime": "2021-07-20T23:20:30.9985256+00:00",
        "status": "disabled"
      },
      "trustPolicy": {
        "status": "disabled",
        "type": "Notary"
      },
    "privateEndpointConnections": [],
    "provisioningState": "Succeeded",
    "publicNetworkAccess": "Disabled",
    "zoneRedundancy": "Disabled"
[...]
}
```

Enable exportPolicy

After disabling the `exportPolicy` status in a registry, you can re-enable it at any time using an ARM template or the `az resource update` command.

ARM template

Include the following JSON to update the `exportPolicy` status to `enabled`. Learn more about deploying resources with ARM templates

```
{
[...]
"resources": [
    {
    "type": "Microsoft.ContainerRegistry/registries",
    "apiVersion": "2021-06-01-preview",
    "name": "myregistry",
    [...]
    "properties": {
     "policies": {
        "exportPolicy": {
          "status": "enabled"
         }
      }
      }
    }
]
[...]
}
```

Azure CLI

Run az resource update to set the `exportPolicy` status to `enabled`. Substitute the names of your registry and resource group.

```
az resource update --resource-group myResourceGroup \
    --name myregistry \
    --resource-type "Microsoft.ContainerRegistry/registries" \
    --api-version "2021-06-01-preview" \
    --set "properties.policies.exportPolicy.status=enabled"
```

Next steps

- Learn about Azure Container Registry roles and permissions.
- If you want to prevent accidental deletion of registry artifacts, see Lock container images.
- Learn about built-in Azure policies to secure your Azure container registry

Delete container images in Azure Container Registry

8/22/2022 • 6 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.

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

> **NOTE**
> After you delete image data, Azure Container Registry stops billing you immediately for the associated storage. However, the registry recovers the associated storage space using an asynchronous process. It takes some time before the registry cleans up layers and shows the updated storage usage.

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):
```

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 manifest list-metadata --name acr-helloworld --registry myregistry
```

```
[
  {
    "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):
```

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 digests 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 manifest list-metadata --name <repositoryName> --registry <acrName> <repositoryName> \
    --orderby time_asc -o tsv --query "[?lastUpdateTime < '2019-04-05'].[digest, lastUpdateTime]"
```

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 manifest list-metadata --name $REPOSITORY --registry $REGISTRY \
    --orderby time_asc --query "[?lastUpdateTime < '$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 manifest list-metadata --name $REPOSITORY --registry $REGISTRY \
   --orderby time_asc --query "[?lastUpdateTime < '$TIMESTAMP'].[digest, lastUpdateTime]" -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 manifest list-metadata --name acr-helloworld --registry myregistry
   ```

   ```
   [
     {
       "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 manifest list-metadata --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**.

Automatically purge tags and manifests

Azure Container Registry provides the following automated methods to remove tags and manifests, and their associated unique layer data:

- Create an ACR task that runs the `acr purge` container command to delete all tags that are older than a certain duration or match a specified name filter. Optionally configure `acr purge` to delete untagged manifests.

  The `acr purge` container command is currently in preview. 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.

  The retention policy is currently a preview feature of **Premium** container registries. The retention policy only applies to untagged manifests created after the policy takes effect.

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

8/22/2022 • 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.

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.

A retention policy for untagged manifests is currently a preview feature of **Premium** container registries. For information about registry service tiers, see Azure Container Registry service tiers.

> **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.

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.

If the `delete-enabled` attribute of an untagged manifest is set to `false`, the manifest is locked and is not deleted by the policy.

> **IMPORTANT**
>
> The retention policy applies only to untagged manifests with timestamps *after* the policy is enabled. Untagged manifests in the registry with earlier timestamps aren't subject to the policy. For other options to delete image data, see examples in Delete container images in Azure Container Registry.

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 manifest list-metadata command. If the test image was the only one in the repository, the repository itself is deleted.

Manage a retention policy

To show 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.

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

8/22/2022 • 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:0.5`), built from source code in the acr-cli repo in GitHub. `acr purge` is currently in preview.

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

> **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 name *regular expression* and a tag name *regular expression* to filter images in the registry. Examples: `--filter "hello-world:.*"` matches all tags in the `hello-world` repository, `--filter "hello-world:^1.*"` matches tags beginning with `1` in the `hello-world` repository, and `--filter ".*/cache:.*"` matches all tags in the repositories ending in `/cache`. You can also pass multiple `--filter` parameters.

- `--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 all 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.
- `--keep` - Specifies that the latest x number of to-be-deleted tags are retained.
- `--concurrency` - Specifies a number of purge tasks to process concurrently. A default value is used if this parameter is not provided.

> **NOTE**
> The `--untagged` filter doesn't respond to the `--ago` filter. 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 and all untagged manifests. 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@sha256:81b6f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e7
88b
myregistry.azurecr.io/samples/devimage1@sha256:3ded859790e68bd02791a972ab0bae727231dc8746f233a7949e40f8ea90c
8b3
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@sha256:8d2527cde610e1715ad095cb12bc7ed169b60c495e5428eefdf336b7cb7c0
371
myregistry.azurecr.io/samples/devimage2@sha256:ca86b078f89607bc03ded859790e68bd02791a972ab0bae727231dc8746f2
33a

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

8/22/2022 • 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.

The endpoint for a webhook must be publicly accessible from the registry. You can configure registry webhook requests to authenticate to a secured endpoint.

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 service tiers 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.

VALUE	DESCRIPTION
Status	The status for the webhook after it's created. It's enabled by default.
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

Manually move a container registry to another region

8/22/2022 • 5 minutes to read • Edit Online

You might need to move an Azure container registry from one Azure region to another. For example, you may run a development pipeline or host a new deployment target in a different region, and want to provide a nearby registry.

While Azure Resource Mover can't currently automate a move for an Azure container registry, you can manually move a container registry to a different region:

- Export registry settings to a Resource Manager template
- Use the template to deploy a registry in a different Azure region
- Import registry content from the source registry to the target registry

> **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 service tier required), you can serve multiple regions with identical image and tag names from a single registry.

Prerequisites

Azure CLI

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

Considerations

- Use steps in this article to move the registry to a different region in the same subscription. More configuration may be needed to move a registry to a different Azure subscription in the same Active Directory tenant.
- Exporting and using a Resource Manager template can help re-create many registry settings. You can edit the

template to configure more settings, or update the target registry after creation.

- Currently, Azure Container Registry doesn't support a registry move to a different Active Directory tenant. This limitation applies to both registries encrypted with a customer-managed key and unencrypted registries.
- If you are unable to move a registry is outlined in this article, create a new registry, manually recreate settings, and Import registry content in the target registry.
- You can find the steps to move resources of registry to a new resource group in the same subscription or move resources to a new subscription.

Export template from source registry

Use the Azure portal, Azure CLI, Azure PowerShell, or other Azure tools to export a Resource Manager template. To use the Azure portal:

1. In the Azure portal, navigate to your source registry.
2. In the menu, under **Automation**, select **Export template** > **Download**.

Redeploy target registry in new region

Modify template

Inspect the registry properties in the template JSON file you downloaded, and make necessary changes. At a minimum:

- Change the registry name's `defaultValue` to the desired name of the target registry
- Update the `location` to the desired Azure region for the target registry

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "registries_myregistry_name": {
            "defaultValue": "myregistry",
            "type": "String"
        }
    },
    "variables": {},
    "resources": [
        {
            "type": "Microsoft.ContainerRegistry/registries",
            "apiVersion": "2020-11-01-preview",
            "name": "[parameters('myregistry_name')]",
            "location": "centralus",
[...]
```

For more information, see Use exported template from the Azure portal and the template reference.

> **IMPORTANT**
>
> If you want to encrypt the target registry using a customer-managed key, make sure to update the template with settings for the required managed identity, key vault, and key. You can only enable the customer-managed key when you deploy the registry.
>
> For more information, see Encrypt registry using customer-managed key.

Create resource group

Create a resource group for the target registry using the az group create. The following example creates a resource group named *myResourceGroup* in the *eastus* location.

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

Deploy target registry in new region

Use the az deployment group create command to deploy the target registry, using the template:

```
az deployment group create --resource-group myResourceGroup \
   --template-file template.json --name mydeployment
```

> **NOTE**
>
> If you see errors during deployment, you might need to update certain configurations in the template file and retry the command.

Import registry content in target registry

After creating the registry in the target region, use the az acr import command, or the equivalent PowerShell command `Import-AzContainerImage`, to import images and other artifacts you want to preserve from the source registry to the target registry. For command examples, see Import container images to a container registry.

- Use the Azure CLI commands az acr repository list and az acr repository show-tags, or Azure PowerShell equivalents, to help enumerate the contents of your source registry.
- Run the import command for individual artifacts, or script it to run over a list of artifacts.

The following sample Azure CLI script enumerates the source repositories and tags and then imports the artifacts to a target registry in the same Azure subscription. Modify as needed to import specific repositories or tags. To import from a registry in a different subscription or tenant, see examples in Import container images to a container registry.

```
#!/bin/bash
# Modify registry names for your environment
SOURCE_REG=myregistry
TARGET_REG=targetregistry

# Get list of source repositories
REPO_LIST=$(az acr repository list \
    --name $SOURCE_REG --output tsv)

# Enumerate tags and import to target registry
for repo in $REPO_LIST; do
    TAGS_LIST=$(az acr repository show-tags --name $SOURCE_REG --repository $repo --output tsv);
    for tag in $TAGS_LIST; do
        echo "Importing $repo:$tag";
        az acr import --name $TARGET_REG --source $SOURCE_REG.azurecr.io/$repo":"$tag;
    done
done
```

Verify target registry

Confirm the following information in your target registry:

- Registry settings such as the registry name, service tier, public access, and replications
- Repositories and tags for content that you want to preserve.

Additional configuration

- If needed, manually configure settings in the target registry such as private endpoints, IP access rules, and managed identities.
- Update development and deployment systems to use the target registry instead of the source registry.
- Update any client firewall rules to allow access to the target registry.

Delete original registry

After you have successfully deployed the target registry, migrated content, and verified registry settings, you may delete the source registry.

Next steps

- Learn more about importing container images to an Azure container registry from a public registry or another private registry.
- See the Resource Manager template reference for Azure Container Registry.

Connect privately to an Azure container registry using Azure Private Link

8/22/2022 • 15 minutes to read • Edit Online

Limit access to a registry by assigning virtual network private IP addresses to the registry endpoints and using Azure Private Link. Network traffic between the clients on the virtual network and the registry's private endpoints traverses the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. Private Link also enables private registry access from on-premises through Azure ExpressRoute private peering or a VPN gateway.

You can configure DNS settings for the registry's private endpoints, so that the settings resolve to the registry's allocated private IP address. With DNS configuration, clients and services in the network can continue to access the registry at the registry's fully qualified domain name, such as *myregistry.azurecr.io*.

This article shows how to configure a private endpoint for your registry using the Azure portal (recommended) or the Azure CLI. This feature is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry tiers.

> **IMPORTANT**
>
> Some functionality may be unavailable or require more configuration in a container registry that restricts access to private endpoints, selected subnets, or IP addresses.
>
> - When public network access to a registry is disabled, registry access by certain trusted services including Azure Security Center requires enabling a network setting to bypass the network rules.
> - Once the public network access is disabled, Instances of certain Azure services including Azure DevOps Services are currently unable to access the container registry.
> - Private endpoints are not currently supported with both self-hosted agents and managed agents.
> - If the registry has an approved private endpoint and public network access is disabled, repositories and tags can't be listed outside the virtual network using the Azure portal, Azure CLI, or other tools.

> **NOTE**
>
> Starting October 2021, new container registries allow a maximum of 200 private endpoints. Registries created earlier allow a maximum of 10 private endpoints. Use the az acr show-usage command to see the limit for your registry.

Prerequisites

- A virtual network and subnet in which to set up the private endpoint. If needed, create a new virtual network and subnet.
- For testing, it's recommended to set up a VM in the virtual network. For steps to create a test virtual machine to access your registry, see Create a Docker-enabled virtual machine.
- To use the Azure CLI steps in this article, Azure CLI version 2.6.0 or later is recommended. If you need to install or upgrade, see Install Azure CLI. Or run in Azure Cloud Shell.
- If you don't already have a container registry, create one (Premium tier required) and import a sample public image such as `mcr.microsoft.com/hello-world` from Microsoft Container Registry. For example, use the Azure portal or the Azure CLI to create a registry.

Register container registry resource provider

To configure registry access using a private link in a different Azure subscription or tenant, you need to register the resource provider for Azure Container Registry in that subscription. Use the Azure portal, Azure CLI, or other tools.

Example:

```
az account set --subscription <Name or ID of subscription of private link>

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

Set up private endpoint - portal (recommended)

Set up a private endpoint when you create a registry, or add a private endpoint to an existing registry.

Create a private endpoint - new registry

1. When creating a registry in the portal, on the **Basics** tab, in **SKU**, select **Premium**.
2. Select the **Networking** tab.
3. In **Network connectivity**, select **Private endpoint** > **+ Add**.
4. Enter or select the following information:

SETTING	VALUE
Subscription	Select your subscription.
Resource group	Enter the name of an existing group or create a new one.
Name	Enter a unique name.
Registry subresource	Select **registry**
Networking	
Virtual network	Select the virtual network for the private endpoint. Example: *myDockerVMVNET*.
Subnet	Select the subnet for the private endpoint. Example: *myDockerVMSubnet*.
Private DNS integration	
Integrate with private DNS zone	Select **Yes**.
Private DNS Zone	Select *(New) privatelink.azurecr.io*

5. Configure the remaining registry settings, and then select **Review + create**.

Your private link is now configured and ready for use.

Create a private endpoint - existing registry

1. In the portal, navigate to your container registry.

2. Under **Settings**, select **Networking**.

3. On the **Private endpoints** tab, select **+ Private endpoint**.

4. In the **Basics** tab, enter or select the following information:

SETTING	VALUE
Project details	
Subscription	Select your subscription.
Resource group	Enter the name of an existing group or create a new one.

SETTING	VALUE
Instance details	
Name	Enter a name.
Region	Select a region.

5. Select **Next: Resource**.
6. Enter or select the following information:

SETTING	VALUE
Connection method	For this example, select **Connect to an Azure resource in my directory**.
Subscription	Select your subscription.
Resource type	Select **Microsoft.ContainerRegistry/registries**.
Resource	Select the name of your registry
Target subresource	Select **registry**

7. Select **Next: Configuration**.
8. Enter or select the information:

SETTING	VALUE
Networking	
Virtual network	Select the virtual network for the private endpoint
Subnet	Select the subnet for the private endpoint
Private DNS Integration	
Integrate with private DNS zone	Select **Yes**.
Private DNS Zone	Select *(New) privatelink.azurecr.io*

9. Select **Review + create**. You're taken to the **Review + create** page where Azure validates your configuration.
10. When you see the **Validation passed** message, select **Create**.

Confirm endpoint configuration

After the private endpoint is created, DNS settings in the private zone appear with the **Private endpoints** settings in the portal:

1. In the portal, navigate to your container registry and select **Settings** > **Networking**.
2. On the **Private endpoints** tab, select the private endpoint you created.
3. Select **DNS configuration**.
4. Review the link settings and custom DNS settings.

Set up private endpoint - CLI

The Azure CLI examples in this article use the following environment variables. You'll need the names of an existing container registry, virtual network, and subnet to set up a private endpoint. Substitute values appropriate for your environment. All examples are formatted for the Bash shell:

```
REGISTRY_NAME=<container-registry-name>
REGISTRY_LOCATION=<container-registry-location> # Azure region such as westeurope where registry created
RESOURCE_GROUP=<resource-group-name> # Resource group for your existing virtual network and subnet
NETWORK_NAME=<virtual-network-name>
SUBNET_NAME=<subnet-name>
```

Disable network policies in subnet

Disable network policies such as network security groups in the subnet for the private endpoint. Update your subnet configuration with az network vnet subnet update:

```
az network vnet subnet update \
 --name $SUBNET_NAME \
 --vnet-name $NETWORK_NAME \
 --resource-group $RESOURCE_GROUP \
 --disable-private-endpoint-network-policies
```

Configure the private DNS zone

Create a private Azure DNS zone for the private Azure container registry domain. In later steps, you create DNS records for your registry domain in this DNS zone. For more information, see DNS configuration options, later in this article.

To use a private zone to override the default DNS resolution for your Azure container registry, the zone must be

named **privatelink.azurecr.io**. Run the following az network private-dns zone create command to create the private zone:

```
az network private-dns zone create \
  --resource-group $RESOURCE_GROUP \
  --name "privatelink.azurecr.io"
```

Create an association link

Run az network private-dns link vnet create to associate your private zone with the virtual network. This example creates a link called *myDNSLink*.

```
az network private-dns link vnet create \
  --resource-group $RESOURCE_GROUP \
  --zone-name "privatelink.azurecr.io" \
  --name MyDNSLink \
  --virtual-network $NETWORK_NAME \
  --registration-enabled false
```

Create a private registry endpoint

In this section, create the registry's private endpoint in the virtual network. First, get the resource ID of your registry:

```
REGISTRY_ID=$(az acr show --name $REGISTRY_NAME \
  --query 'id' --output tsv)
```

Run the az network private-endpoint create command to create the registry's private endpoint.

The following example creates the endpoint *myPrivateEndpoint* and service connection *myConnection*. To specify a container registry resource for the endpoint, pass `--group-ids registry`:

```
az network private-endpoint create \
    --name myPrivateEndpoint \
    --resource-group $RESOURCE_GROUP \
    --vnet-name $NETWORK_NAME \
    --subnet $SUBNET_NAME \
    --private-connection-resource-id $REGISTRY_ID \
    --group-ids registry \
    --connection-name myConnection
```

Get endpoint IP configuration

To configure DNS records, get the IP configuration of the private endpoint. Associated with the private endpoint's network interface in this example are two private IP addresses for the container registry: one for the registry itself, and one for the registry's data endpoint. If your registry is geo-replicated, an additional IP address is associated with each replica.

First, run az network private-endpoint show to query the private endpoint for the network interface ID:

```
NETWORK_INTERFACE_ID=$(az network private-endpoint show \
  --name myPrivateEndpoint \
  --resource-group $RESOURCE_GROUP \
  --query 'networkInterfaces[0].id' \
  --output tsv)
```

The following az network nic show commands get the private IP addresses and FQDNs for the container registry and the registry's data endpoint:

```
REGISTRY_PRIVATE_IP=$(az network nic show \
  --ids $NETWORK_INTERFACE_ID \
  --query "ipConfigurations[?
privateLinkConnectionProperties.requiredMemberName=='registry'].privateIpAddress" \
  --output tsv)

DATA_ENDPOINT_PRIVATE_IP=$(az network nic show \
  --ids $NETWORK_INTERFACE_ID \
  --query "ipConfigurations[?
privateLinkConnectionProperties.requiredMemberName=='registry_data_$REGISTRY_LOCATION'].privateIpAddress" \
  --output tsv)

# An FQDN is associated with each IP address in the IP configurations

REGISTRY_FQDN=$(az network nic show \
  --ids $NETWORK_INTERFACE_ID \
  --query "ipConfigurations[?
privateLinkConnectionProperties.requiredMemberName=='registry'].privateLinkConnectionProperties.fqdns" \
  --output tsv)

DATA_ENDPOINT_FQDN=$(az network nic show \
  --ids $NETWORK_INTERFACE_ID \
  --query "ipConfigurations[?
privateLinkConnectionProperties.requiredMemberName=='registry_data_$REGISTRY_LOCATION'].privateLinkConnectio
nProperties.fqdns" \
  --output tsv)
```

Additional endpoints for geo-replicas

If your registry is geo-replicated, query for the additional data endpoint for each registry replica. For example, in the *eastus* region:

```
REPLICA_LOCATION=eastus
GEO_REPLICA_DATA_ENDPOINT_PRIVATE_IP=$(az network nic show \
  --ids $NETWORK_INTERFACE_ID \
  --query "ipConfigurations[?
privateLinkConnectionProperties.requiredMemberName=='registry_data_$REPLICA_LOCATION'].privateIpAddress" \
  --output tsv)

GEO_REPLICA_DATA_ENDPOINT_FQDN=$(az network nic show \
  --ids $NETWORK_INTERFACE_ID \
  --query "ipConfigurations[?
privateLinkConnectionProperties.requiredMemberName=='registry_data_$REPLICA_LOCATION'].privateLinkConnection
Properties.fqdns" \
  --output tsv)
```

Create DNS records in the private zone

The following commands create DNS records in the private zone for the registry endpoint and its data endpoint. For example, if you have a registry named *myregistry* in the *westeurope* region, the endpoint names are `myregistry.azurecr.io` and `myregistry.westeurope.data.azurecr.io`.

First run az network private-dns record-set a create to create empty A-record sets for the registry endpoint and data endpoint:

```
az network private-dns record-set a create \
  --name $REGISTRY_NAME \
  --zone-name privatelink.azurecr.io \
  --resource-group $RESOURCE_GROUP

# Specify registry region in data endpoint name
az network private-dns record-set a create \
  --name ${REGISTRY_NAME}.${REGISTRY_LOCATION}.data \
  --zone-name privatelink.azurecr.io \
  --resource-group $RESOURCE_GROUP
```

Run the az network private-dns record-set a add-record command to create the A-records for the registry endpoint and data endpoint:

```
az network private-dns record-set a add-record \
  --record-set-name $REGISTRY_NAME \
  --zone-name privatelink.azurecr.io \
  --resource-group $RESOURCE_GROUP \
  --ipv4-address $REGISTRY_PRIVATE_IP

# Specify registry region in data endpoint name
az network private-dns record-set a add-record \
  --record-set-name ${REGISTRY_NAME}.${REGISTRY_LOCATION}.data \
  --zone-name privatelink.azurecr.io \
  --resource-group $RESOURCE_GROUP \
  --ipv4-address $DATA_ENDPOINT_PRIVATE_IP
```

Additional records for geo-replicas

If your registry is geo-replicated, create additional DNS settings for each replica. Continuing the example in the *eastus* region:

```
az network private-dns record-set a create \
  --name ${REGISTRY_NAME}.${REPLICA_LOCATION}.data \
  --zone-name privatelink.azurecr.io \
  --resource-group $RESOURCE_GROUP

az network private-dns record-set a add-record \
  --record-set-name ${REGISTRY_NAME}.${REPLICA_LOCATION}.data \
  --zone-name privatelink.azurecr.io \
  --resource-group $RESOURCE_GROUP \
  --ipv4-address $GEO_REPLICA_DATA_ENDPOINT_PRIVATE_IP
```

The private link is now configured and ready for use.

Disable public access

For many scenarios, disable registry access from public networks. This configuration prevents clients outside the virtual network from reaching the registry endpoints.

Disable public access - portal

1. In the portal, navigate to your container registry and select **Settings** > **Networking**.
2. On the **Public access** tab, in **Allow public network access**, select **Disabled**. Then select **Save**.

Disable public access - CLI

> **NOTE**
>
> If the public access is disabled, the `az acr build` commands will no longer work.

To disable public access using the Azure CLI, run az acr update and set `--public-network-enabled` to `false`.

```
az acr update --name $REGISTRY_NAME --public-network-enabled false
```

Execute the `az acr build` with private endpoint and private registry

Consider the following options to execute the `az acr build` successfully.

> **NOTE**
> Once you disable public network access here, then `az acr build` commands will no longer work.

1. Assign a dedicated agent pool.
2. If agent pool is not available in the region, add the regional Azure Container Registry Service Tag IPv4 to the firewall access rules.
3. Create an ACR task with a managed identity, and enable trusted services to access network restricted ACR.

Disable access to a container registry using a service endpoint

> **IMPORTANT**
> The container registry does not support enabling both private link and service endpoint features configured from a virtual network.

Once the registry has public access disabled and private link configured, you can disable the service endpoint access to a container registry from a virtual network by removing virtual network rules.

- Run `az acr network-rule list` command to list the existing network rules.
- Run `az acr network-rule remove` command to remove the network rule.

Validate private link connection

You should validate that the resources within the subnet of the private endpoint connect to your registry over a private IP address, and have the correct private DNS zone integration.

To validate the private link connection, connect to the virtual machine you set up in the virtual network.

Run a utility such as `nslookup` or `dig` to look up the IP address of your registry over the private link. For example:

```
dig $REGISTRY_NAME.azurecr.io
```

Example output shows the registry's IP address in the address space of the subnet:

```
[...]
; <<>> DiG 9.11.3-1ubuntu1.13-Ubuntu <<>> myregistry.azurecr.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52155
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;myregistry.azurecr.io.         IN      A

;; ANSWER SECTION:
myregistry.azurecr.io.  1783    IN      CNAME   myregistry.privatelink.azurecr.io.
myregistry.privatelink.azurecr.io. 10 IN A      10.0.0.7

[...]
```

Compare this result with the public IP address in `dig` output for the same registry over a public endpoint:

```
[...]
;; ANSWER SECTION:
myregistry.azurecr.io. 2881 IN CNAME myregistry.privatelink.azurecr.io.
myregistry.privatelink.azurecr.io. 2881 IN CNAME xxxx.xx.azcr.io.
xxxx.xx.azcr.io. 300 IN CNAME xxxx-xxx-reg.trafficmanager.net.
xxxx-xxx-reg.trafficmanager.net. 300 IN CNAME xxxx.westeurope.cloudapp.azure.com.
xxxx.westeurope.cloudapp.azure.com. 10 IN A 20.45.122.144

[...]
```

Registry operations over private link

Also verify that you can perform registry operations from the virtual machine in the network. Make an SSH connection to your virtual machine, and run az acr login to login to your registry. Depending on your VM configuration, you might need to prefix the following commands with `sudo`.

```
az acr login --name $REGISTRY_NAME
```

Perform registry operations such as `docker pull` to pull a sample image from the registry. Replace `hello-world:v1` with an image and tag appropriate for your registry, prefixed with the registry login server name (all lowercase):

```
docker pull myregistry.azurecr.io/hello-world:v1
```

Docker successfully pulls the image to the VM.

Manage private endpoint connections

Manage a registry's private endpoint connections using the Azure portal, or by using commands in the az acr private-endpoint-connection command group. Operations include approve, delete, list, reject, or show details of a registry's private endpoint connections.

For example, to list the private endpoint connections of a registry, run the az acr private-endpoint-connection list command. For example:

```
az acr private-endpoint-connection list \
  --registry-name $REGISTRY_NAME
```

When you set up a private endpoint connection using the steps in this article, the registry automatically accepts connections from clients and services that have Azure RBAC permissions on the registry. You can set up the endpoint to require manual approval of connections. For information about how to approve and reject private endpoint connections, see Manage a Private Endpoint Connection.

> **IMPORTANT**
> Currently, if you delete a private endpoint from a registry, you might also need to delete the virtual network's link to the private zone. If the link isn't deleted, you may see an error similar to `unresolvable host`.

DNS configuration options

The private endpoint in this example integrates with a private DNS zone associated with a basic virtual network. This setup uses the Azure-provided DNS service directly to resolve the registry's public FQDN to its private IP addresses in the virtual network.

Private link supports additional DNS configuration scenarios that use the private zone, including with custom DNS solutions. For example, you might have a custom DNS solution deployed in the virtual network, or on-premises in a network you connect to the virtual network using a VPN gateway or Azure ExpressRoute.

To resolve the registry's public FQDN to the private IP address in these scenarios, you need to configure a server-level forwarder to the Azure DNS service (168.63.129.16). Exact configuration options and steps depend on your existing networks and DNS. For examples, see Azure Private Endpoint DNS configuration.

> **IMPORTANT**
> If for high availability you created private endpoints in several regions, we recommend that you use a separate resource group in each region and place the virtual network and the associated private DNS zone in it. This configuration also prevents unpredictable DNS resolution caused by sharing the same private DNS zone.

Manually configure DNS records

For some scenarios, you may need to manually configure DNS records in a private zone instead of using the Azure-provided private zone. Be sure to create records for each of the following endpoints: the registry endpoint, the registry's data endpoint, and the data endpoint for any additional regional replica. If all records aren't configured, the registry may be unreachable.

> **IMPORTANT**
> If you later add a new replica, you need to manually add a new DNS record for the data endpoint in that region. For example, if you create a replica of *myregistry* in the northeurope location, add a record for `myregistry.northeurope.data.azurecr.io`.

The FQDNs and private IP addresses you need to create DNS records are associated with the private endpoint's network interface. You can obtain this information using the Azure portal or Azure CLI.

- In the portal, navigate to your private endpoint, and select **DNS configuration**.
- Using the Azure CLI, run the az network nic show command. For example commands, see Get endpoint IP configuration, earlier in this article.

After creating DNS records, make sure that the registry FQDNs resolve properly to their respective private IP addresses.

Clean up resources

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

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 $RESOURCE_GROUP
```

Next steps

- To learn more about Private Link, see the Azure Private Link documentation.
- To verify DNS settings in the virtual network that route to a private endpoint, run the az acr check-health command with the `--vnet` parameter. For more information, see Check the health of an Azure container registry
- If you need to set up registry access rules from behind a client firewall, see Configure rules to access an Azure container registry behind a firewall.
- Troubleshoot Azure Private Endpoint connectivity problems

Configure public IP network rules

8/22/2022 • 5 minutes to read • Edit Online

An Azure container registry by default accepts connections over the internet from hosts on any network. This article shows how to configure your container registry to allow access from only specific public IP addresses or address ranges. Equivalent steps using the Azure CLI and Azure portal are provided.

IP network rules are configured on the public registry endpoint. IP network rules do not apply to private endpoints configured with Private Link

Configuring IP access rules is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry tiers.

Each registry supports a maximum of 100 IP access rules.

> **IMPORTANT**
>
> Some functionality may be unavailable or require more configuration in a container registry that restricts access to private endpoints, selected subnets, or IP addresses.
>
> - When public network access to a registry is disabled, registry access by certain trusted services including Azure Security Center requires enabling a network setting to bypass the network rules.
> - Once the public network access is disabled, Instances of certain Azure services including Azure DevOps Services are currently unable to access the container registry.
> - Private endpoints are not currently supported with both self-hosted agents and managed agents.
> - If the registry has an approved private endpoint and public network access is disabled, repositories and tags can't be listed outside the virtual network using the Azure portal, Azure CLI, or other tools.

Access from selected public network - CLI

Change default network access to registry

To limit access to a selected public network, first 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 a public IP address or range. For example, substitute the container registry's name and the public IP address of a VM in a virtual network.

```
az acr network-rule add \
  --name mycontainerregistry \
  --ip-address <public-IP-address>
```

> **NOTE**
>
> After adding a rule, it takes a few minutes for the rule to take effect.

Access from selected public network - portal

1. In the portal, navigate to your container registry.
2. Under **Settings**, select **Networking**.
3. On the **Public access** tab, select to allow public access from **Selected networks**.
4. Under **Firewall**, enter a public IP address, such as the public IP address of a VM in a virtual network. Or, enter an address range in CIDR notation that contains the VM's IP address.
5. Select **Save**.

> **NOTE**
>
> After adding a rule, it takes a few minutes for the rule to take effect.

> **TIP**
>
> Optionally, enable registry access from a local client computer or IP address range. To allow this access, you need the computer's public IPv4 address. You can find this address by searching "what is my IP address" in an internet browser. The current client IPv4 address also appears automatically when you configure firewall settings on the **Networking** page in the portal.

Disable public network access

Optionally, disable the public endpoint on the registry. Disabling the public endpoint overrides all firewall configurations. For example, you might want to disable public access to a registry secured in a virtual network using Private Link.

> **NOTE**
>
> If the registry is set up in a virtual network with a service endpoint, disabling access to the registry's public endpoint also disables access to the registry within the virtual network.

Disable public access - CLI

To disable public access using the Azure CLI, run az acr update and set `--public-network-enabled` to `false`. The `public-network-enabled` argument requires Azure CLI 2.6.0 or later.

```
az acr update --name myContainerRegistry --public-network-enabled false
```

Disable public access - portal

1. In the portal, navigate to your container registry and select **Settings** > **Networking**.
2. On the **Public access** tab, in **Allow public network access**, select **Disabled**. Then select **Save**.

Restore public network access

To re-enable the public endpoint, update the networking settings to allow public access. Enabling the public endpoint overrides all firewall configurations.

Restore public access - CLI

Run az acr update and set `--public-network-enabled` to `true`.

> **NOTE**
>
> The `public-network-enabled` argument requires Azure CLI 2.6.0 or later.

```
az acr update --name myContainerRegistry --public-network-enabled true
```

Restore public access - portal

1. In the portal, navigate to your container registry and select **Settings** > **Networking**.
2. On the **Public access** tab, in **Allow public network access**, select **All networks**. Then select **Save**.

Troubleshoot

Access behind HTTPS proxy

If a public network rule is set, or public access to the registry is denied, attempts to login to the registry from a disallowed public network will fail. Client access from behind an HTTPS proxy will also fail if an access rule for the proxy is not set. You will see an error message similar to `Error response from daemon: login attempt failed with status: 403 Forbidden` or `Looks like you don't have access to registry`.

These errors can also occur if you use an HTTPS proxy that is allowed by a network access rule, but the proxy isn't properly configured in the client environment. Check that both your Docker client and the Docker daemon are configured for proxy behavior. For details, see HTTP/HTTPS proxy in the Docker documentation.

Access from Azure Pipelines

If you use Azure Pipelines with an Azure container registry that limits access to specific IP addresses, the pipeline may be unable to access the registry, because the outbound IP address from the pipeline is not fixed. By default, the pipeline runs jobs using a Microsoft-hosted agent on a virtual machine pool with a changing set of IP addresses.

One workaround is to change the agent used to run the pipeline from Microsoft-hosted to self-hosted. With a self-hosted agent running on a Windows or Linux machine that you manage, you control the outbound IP address of the pipeline, and you can add this address in a registry IP access rule.

Access from AKS

If you use Azure Kubernetes Service (AKS) with an Azure container registry that limits access to specific IP addresses, you can't configure a fixed AKS IP address by default. The egress IP address from the AKS cluster is randomly assigned.

To allow the AKS cluster to access the registry, you have these options:

- If you use the Azure Basic Load Balancer, set up a static IP address for the AKS cluster.
- If you use the Azure Standard Load Balancer, see guidance to control egress traffic from the cluster.

Next steps

- To restrict access to a registry using a private endpoint in a virtual network, see Configure Azure Private Link for an Azure container registry.
- If you need to set up registry access rules from behind a client firewall, see Configure rules to access an Azure container registry behind a firewall.
- For more troubleshooting guidance, see Troubleshoot network issues with registry.

Restrict access to a container registry using a service endpoint in an Azure virtual network

8/22/2022 • 7 minutes to read • Edit Online

Azure Virtual Network provides secure, private networking for your Azure and on-premises resources. A service endpoint allows you to secure your container registry's public IP address to only your virtual network. 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.

This article shows how to configure a container registry service endpoint (preview) in a virtual network.

Each registry supports a maximum of 100 virtual network rules.

> **IMPORTANT**
>
> Azure Container Registry now supports Azure Private Link, enabling private endpoints from a virtual network to be placed on a registry. Private endpoints are accessible from within the virtual network, using private IP addresses. We recommend using private endpoints instead of service endpoints in most network scenarios. The container registry does not support enabling both private link and service endpoint features configured from a virtual network. So, we recommend running the list and removing the network rules as required.

Configuring a registry service endpoint is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry service tiers.

Preview limitations

- Future development of service endpoints for Azure Container Registry isn't currently planned. We recommend using private endpoints instead.
- You can't use the Azure portal to configure service endpoints on a registry.
- Only an Azure Kubernetes Service cluster or Azure virtual machine can be used as a host to access a container registry using a service endpoint. *Other Azure services including Azure Container Instances aren't supported.*
- Service endpoints for Azure Container Registry aren't supported in the Azure US Government cloud or Azure China cloud.

> **IMPORTANT**
>
> Some functionality may be unavailable or require more configuration in a container registry that restricts access to private endpoints, selected subnets, or IP addresses.
>
> - When public network access to a registry is disabled, registry access by certain trusted services including Azure Security Center requires enabling a network setting to bypass the network rules.
> - Once the public network access is disabled, Instances of certain Azure services including Azure DevOps Services are currently unable to access the container registry.
> - Private endpoints are not currently supported with both self-hosted agents and managed agents.
> - If the registry has an approved private endpoint and public network access is disabled, repositories and tags can't be listed outside the virtual network using the Azure portal, Azure CLI, or other tools.

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 tier 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 service endpoint in a different Azure subscription, 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
```

Create a Docker-enabled virtual machine

For test purposes, 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 assumes you've set environment variables for the resource group name and registry location:

```
az group create --name $RESOURCE_GROUP --location $REGISTRY_LOCATION
```

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

```
VM_NAME=myDockerVM

az vm create \
  --resource-group $RESOURCE_GROUP \
  --name $VM_NAME \
  --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 commands to install Docker on the Ubuntu VM:

```
sudo apt-get update
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 example:

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

Exit the SSH connection.

Configure network access for registry

In this section, configure your container registry to allow access from a subnet in an Azure virtual network. Steps are provided using the Azure 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 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/subne
ts/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>
```

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 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.

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/subne
ts/myDockerVMSubnet
```

Allow access

Substitute the name of your registry in the following az acr update command:

```
az acr update --name myContainerRegistry --default-action Allow
```

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
```

Next steps

- To restrict access to a registry using a private endpoint in a virtual network, see Configure Azure Private Link for an Azure container registry.
- If you need to set up registry access rules from behind a client firewall, see Configure rules to access an Azure container registry behind a firewall.

Allow trusted services to securely access a network-restricted container registry

8/22/2022 • 4 minutes to read • Edit Online

Azure Container Registry can allow select trusted Azure services to access a registry that's configured with network access rules. When trusted services are allowed, a trusted service instance can securely bypass the registry's network rules and perform operations such as pull or push images. This article explains how to enable and use trusted services with a network-restricted Azure container registry.

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

Limitations

- Certain registry access scenarios with trusted services require a managed identity for Azure resources. Except where noted that a user-assigned managed identity is supported, only a system-assigned identity may be used.
- Allowing trusted services doesn't apply to a container registry configured with a service endpoint. The feature only affects registries that are restricted with a private endpoint or that have public IP access rules applied.

About trusted services

Azure Container Registry has a layered security model, supporting multiple network configurations that restrict access to a registry, including:

- Private endpoint with Azure Private Link. When configured, a registry's private endpoint is accessible only to resources within the virtual network, using private IP addresses.
- Registry firewall rules, which allow access to the registry's public endpoint only from specific public IP addresses or address ranges. You can also configure the firewall to block all access to the public endpoint when using private endpoints.

When deployed in a virtual network or configured with firewall rules, a registry denies access to users or services from outside those sources.

Several multi-tenant Azure services operate from networks that can't be included in these registry network settings, preventing them from performing operations such as pull or push images to the registry. By designating certain service instances as "trusted", a registry owner can allow select Azure resources to securely bypass the registry's network settings to perform registry operations.

Trusted services

Instances of the following services can access a network-restricted container registry if the registry's **allow trusted services** setting is enabled (the default). More services will be added over time.

Where indicated, access by the trusted service requires additional configuration of a managed identity in a service instance, assignment of an RBAC role, and authentication with the registry. For example steps, see Trusted services workflow, later in this article.

TRUSTED SERVICE	SUPPORTED USAGE SCENARIOS	CONFIGURE MANAGED IDENTITY WITH RBAC ROLE
Azure Container Instances	Deploy to Azure Container Instances from Azure Container Registry using a managed identity	Yes, either system-assigned or user-assigned identity
Microsoft Defender for Cloud	Vulnerability scanning by Microsoft Defender for container registries	No
ACR Tasks	Access the parent registry or a different registry from an ACR Task	Yes
Machine Learning	Deploy or train a model in a Machine Learning workspace using a custom Docker container image	Yes
Azure Container Registry	Import images to or from a network-restricted Azure container registry	No

> **NOTE**
> Curently, enabling the allow trusted services setting doesn't apply to App Service.

Allow trusted services - CLI

By default, the allow trusted services setting is enabled in a new Azure container registry. Disable or enable the setting by running the az acr update command.

To disable:

```
az acr update --name myregistry --allow-trusted-services false
```

To enable the setting in an existing registry or a registry where it's already disabled:

```
az acr update --name myregistry --allow-trusted-services true
```

Allow trusted services - portal

By default, the allow trusted services setting is enabled in a new Azure container registry.

To disable or re-enable the setting in the portal:

1. In the portal, navigate to your container registry.
2. Under **Settings**, select **Networking**.
3. In **Allow public network access**, select **Selected networks** or **Disabled**.
4. Do one of the following:
   - To disable access by trusted services, under **Firewall exception**, uncheck **Allow trusted Microsoft services to access this container registry**.
   - To allow trusted services, under **Firewall exception**, check **Allow trusted Microsoft services to access this container registry**.
5. Select **Save**.

Trusted services workflow

Here's a typical workflow to enable an instance of a trusted service to access a network-restricted container registry. This workflow is needed when a service instance's managed identity is used to bypass the registry's network rules.

1. Enable a managed identity in an instance of one of the trusted services for Azure Container Registry.
2. Assign the identity an Azure role to your registry. For example, assign the ACRPull role to pull container images.
3. In the network-restricted registry, configure the setting to allow access by trusted services.
4. Use the identity's credentials to authenticate with the network-restricted registry.
5. Pull images from the registry, or perform other operations allowed by the role.

Example: ACR Tasks

The following example demonstrates using ACR Tasks as a trusted service. See Cross-registry authentication in an ACR task using an Azure-managed identity for task details.

1. Create or update an Azure container registry. Create an ACR task.
   - Enable a system-assigned managed identity when creating the task.
   - Disable default auth mode (`--auth-mode None`) of the task.
2. Assign the task identity an Azure role to access the registry. For example, assign the AcrPush role, which has permissions to pull and push images.
3. Add managed identity credentials for the registry to the task.
4. To confirm that the task bypasses network restrictions, disable public access in the registry.
5. Run the task. If the registry and task are configured properly, the task runs successfully, because the registry allows access.

To test disabling access by trusted services:

1. Disable the setting to allow access by trusted services.
2. Run the task again. In this case, the task run fails, because the registry no longer allows access by the task.

Next steps

- To restrict access to a registry using a private endpoint in a virtual network, see Configure Azure Private Link for an Azure container registry.
- To set up registry firewall rules, see Configure public IP network rules.

Encrypt registry using a customer-managed key

8/22/2022 • 15 minutes to read • Edit Online

When you store images and other artifacts in an Azure container registry, Azure automatically encrypts the registry content at rest with service-managed keys. You can supplement default encryption with an additional encryption layer using a key that you create and manage in Azure Key Vault (a customer-managed key). This article walks you through the steps using the Azure CLI, the Azure portal, or a Resource Manager template.

Server-side encryption with customer-managed keys is supported through integration with Azure Key Vault:

- You can create your own encryption keys and store them in a key vault, or use Azure Key Vault APIs to generate keys.
- With Azure Key Vault, you can also audit key usage.
- Azure Container Registry supports automatic rotation of registry encryption keys when a new key version is available in Azure Key Vault. You can also manually rotate registry encryption keys.

This feature is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry service tiers.

Things to know

- You can currently enable a customer-managed key only when you create a registry. When enabling the key, you configure a *user-assigned* managed identity to access the key vault. Later, you can enable the registry's system-managed identity for key vault access if needed.
- After enabling encryption with a customer-managed key on a registry, you can't disable the encryption.
- Azure Container Registry supports only RSA or RSA-HSM keys. Elliptic curve keys aren't currently supported.
- Content trust is currently not supported in a registry encrypted with a customer-managed key.
- In a registry encrypted with a customer-managed key, run logs for ACR Tasks are currently retained for only 24 hours. If you need to retain logs for a longer period, see guidance to export and store task run logs.

Automatic or manual update of key versions

An important consideration for the security of a registry encrypted with a customer-managed key is how frequently you update (rotate) the encryption key. Your organization might have compliance policies that require regularly updating key versions stored in Azure Key Vault when used as customer-managed keys.

When you configure registry encryption with a customer-managed key, you have two options for updating the key version used for encryption:

- **Automatically update the key version** - To automatically update a customer-managed key when a new version is available in Azure Key Vault, omit the key version when you enable registry encryption with a customer-managed key. When a registry is encrypted with a non-versioned key, Azure Container Registry regularly checks the key vault for a new key version and updates the customer-managed key within 1 hour. Azure Container Registry automatically uses the latest version of the key.

- **Manually update the key version** - To use a specific version of a key for registry encryption, specify that key version when you enable registry encryption with a customer-managed key. When a registry is encrypted with a specific key version, Azure Container Registry uses that version for encryption until you manually rotate the customer-managed key.

For details, see Choose key ID with or without key version and Update key version, later in this article.

Prerequisites

To use the Azure CLI steps in this article, you need Azure CLI version 2.2.0 or later, or Azure Cloud Shell. If you need to install or upgrade, see Install Azure CLI.

Enable customer-managed key - CLI

Create a resource group

If needed, run the az group create command to create a resource group for creating the key vault, container registry, and other required resources.

```
az group create --name <resource-group-name> --location <location>
```

Create a user-assigned managed identity

Create a user-assigned managed identity for Azure resources with the az identity create command. This identity will be used by your registry to access the Key Vault service.

```
az identity create \
  --resource-group <resource-group-name> \
  --name <managed-identity-name>
```

In the command output, take note of the following values: `id` and `principalId`. You need these values in later steps to configure registry access to the key vault.

```
{
  "clientId": "xxxx2bac-xxxx-xxxx-xxxx-192cxxxx6273",
  "clientSecretUrl": "https://control-eastus.identity.azure.net/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourcegroups/myresourcegroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myide
ntityname/credentials?tid=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx&oid=xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx&aid=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "id": "/subscriptions/xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx/resourcegroups/myresourcegroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myres
ourcegroup",
  "location": "eastus",
  "name": "myidentityname",
  "principalId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "resourceGroup": "myresourcegroup",
  "tags": {},
  "tenantId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "type": "Microsoft.ManagedIdentity/userAssignedIdentities"
}
```

For convenience, store these values in environment variables:

```
identityID=$(az identity show --resource-group <resource-group-name> --name <managed-identity-name> --query
'id' --output tsv)

identityPrincipalID=$(az identity show --resource-group <resource-group-name> --name <managed-identity-name>
--query 'principalId' --output tsv)
```

Create a key vault

Create a key vault with az keyvault create to store a customer-managed key for registry encryption.

By default, the **soft delete** setting is automatically enabled in a new key vault. To prevent data loss caused by accidental key or key vault deletions, also enable the **purge protection** setting.

```
az keyvault create --name <key-vault-name> \
  --resource-group <resource-group-name> \
  --enable-purge-protection
```

For use in later steps, get the resource ID of the key vault:

```
keyvaultID=$(az keyvault show --resource-group <resource-group-name> --name <key-vault-name> --query 'id' --
output tsv)
```

Enable key vault access by trusted services

If the key vault is protected with a firewall or virtual network (private endpoint), enable the network setting to allow access by trusted Azure services.

For more information, see Configure Azure Key Vault networking settings.

Enable key vault access by managed identity

Enable key vault access policy

One option is to configure a policy for the key vault so that the identity can access it. In the following az keyvault set-policy command, you pass the principal ID of the managed identity that you created, stored previously in an environment variable. Set key permissions to **get**, **unwrapKey**, and **wrapKey**.

```
az keyvault set-policy \
  --resource-group <resource-group-name> \
  --name <key-vault-name> \
  --object-id $identityPrincipalID \
  --key-permissions get unwrapKey wrapKey
```

Assign RBAC role

Alternatively, use Azure RBAC for Key Vault to assign permissions to the identity to access the key vault. For example, assign the Key Vault Crypto Service Encryption role to the identity using the az role assignment create command:

```
az role assignment create --assignee $identityPrincipalID \
  --role "Key Vault Crypto Service Encryption User" \
  --scope $keyvaultID
```

Create key and get key ID

Run the az keyvault key create command to create a key in the key vault.

```
az keyvault key create \
  --name <key-name> \
  --vault-name <key-vault-name>
```

In the command output, take note of the key's ID, `kid`. You use this ID in the next step:

```
[...]
  "key": {
    "crv": null,
    "d": null,
    "dp": null,
    "dq": null,
    "e": "AQAB",
    "k": null,
    "keyOps": [
      "encrypt",
      "decrypt",
      "sign",
      "verify",
      "wrapKey",
      "unwrapKey"
    ],
    "kid": "https://mykeyvault.vault.azure.net/keys/mykey/<version>",
    "kty": "RSA",
[...]
```

Choose key ID with or without key version

For convenience, store the format you choose for the key ID in the $keyID environment variable. You can use a key ID with a version or a key without a version.

Manual key rotation - key ID with version

When used to encrypt a registry with a customer-managed key, this key allows only manual key rotation in Azure Container Registry.

This example stores the key's `kid` property:

```
keyID=$(az keyvault key show \
  --name <keyname> \
  --vault-name <key-vault-name> \
  --query 'key.kid' --output tsv)
```

Automatic key rotation - key ID omitting version

When used to encrypt a registry with a customer-managed key, this key enables automatic key rotation when a new key version is detected in Azure Key Vault.

This example removes the version from the key's `kid` property:

```
keyID=$(az keyvault key show \
  --name <keyname> \
  --vault-name <key-vault-name> \
  --query 'key.kid' --output tsv)

keyID=$(echo $keyID | sed -e "s/\/[^/]*$//")
```

Create a registry with customer-managed key

Run the az acr create command to create a registry in the Premium service tier and enable the customer-managed key. Pass the managed identity ID and the key ID, stored previously in environment variables:

```
az acr create \
  --resource-group <resource-group-name> \
  --name <container-registry-name> \
  --identity $identityID \
  --key-encryption-key $keyID \
  --sku Premium
```

Show encryption status

To show whether registry encryption with a customer-managed key is enabled, run the az acr encryption show command:

```
az acr encryption show --name <registry-name>
```

Depending on the key used to encrypt the registry, output is similar to:

```
{
  "keyVaultProperties": {
    "identity": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
    "keyIdentifier": "https://myvault.vault.azure.net/keys/myresourcegroup/abcdefg123456789...",
    "keyRotationEnabled": true,
    "lastKeyRotationTimestamp": xxxxxxxx
    "versionedKeyIdentifier": "https://myvault.vault.azure.net/keys/myresourcegroup/abcdefg123456789...",
  },
  "status": "enabled"
}
```

Enable customer-managed key - portal

Create a managed identity

Create a user-assigned managed identity for Azure resources in the Azure portal. For steps, see Create a user-assigned identity.

You use the identity's name in later steps.

Create a key vault

For steps to create a key vault, see Quickstart: Create a key vault using the Azure portal.

When creating a key vault for a customer-managed key, in the **Basics** tab, enable the **Purge protection** setting. This setting helps prevent data loss caused by accidental key or key vault deletions.

Enable key vault access by trusted services

If the key vault is protected with a firewall or virtual network (private endpoint), enable the network setting to allow access by trusted Azure services.

For more information, see Configure Azure Key Vault networking settings.

Enable key vault access by managed identity

Enable key vault access policy

One option is to configure a policy for the key vault so that the identity can access it.

1. Navigate to your key vault.
2. Select **Settings** > **Access policies** > **+Add Access Policy**.
3. Select **Key permissions**, and select **Get**, **Unwrap Key**, and **Wrap Key**.
4. In **Select principal**, select the resource name of your user-assigned managed identity.

5. Select **Add**, then select **Save**.

Assign RBAC role

Alternatively, assign the Key Vault Crypto Service Encryption User role to the user-assigned managed identity at the key vault scope.

For detailed steps, see Assign Azure roles using the Azure portal.

Create key (optional)

Optionally create a key in the key vault for use to encrypt the registry. Follow these steps if you want to select a specific key version as a customer-managed key. You may also need to create a key before creating the registry if key vault access is restricted to a private endpoint or selected networks.

1. Navigate to your key vault.
2. Select **Settings** > **Keys**.
3. Select **+Generate/Import** and enter a unique name for the key.
4. Accept the remaining default values and select **Create**.
5. After creation, select the key and then select the current version. Copy the **Key identifier** for the key version.

Create Azure container registry

1. Select **Create a resource** > **Containers** > **Container Registry**.
2. In the **Basics** tab, select or create a resource group, and enter a registry name. In **SKU**, select **Premium**.
3. In the **Encryption** tab, in **Customer-managed key**, select **Enabled**.
4. In **Identity**, select the managed identity you created.
5. In **Encryption**, choose either of the following:
   - Select **Select from Key Vault**, and select an existing key vault and key, or **Create new**. The key you select is non-versioned and enables automatic key rotation.
   - Select **Enter key URI**, and provide the identifier of an existing key. You can provide either a versioned key URI (for a key that must be rotated manually) or a non-versioned key URI (which enables automatic key rotation). See the previous section for steps to create a key.
6. In the **Encryption** tab, select **Review + create**.
7. Select **Create** to deploy the registry instance.

To see the encryption status of your registry in the portal, navigate to your registry. Under **Settings**, select **Encryption**.

Enable customer-managed key - template

You can also use a Resource Manager template to create a registry and enable encryption with a customer-managed key.

The following template creates a new container registry and a user-assigned managed identity. Copy the following contents to a new file and save it using a filename such as `CMKtemplate.json`.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "vault_name": {
      "defaultValue": "",
      "type": "String"
    },
    "registry_name": {
      "defaultValue": "",
      "type": "String"
    },
    "identity_name": {
      "defaultValue": "",
      "type": "String"
    },
    "kek_id": {
      "type": "String"
    }
  },
  "variables": {},
  "resources": [
    {
      "type": "Microsoft.ContainerRegistry/registries",
      "apiVersion": "2019-12-01-preview",
      "name": "[parameters('registry_name')]",
```

```
      "location": "[resourceGroup().location]",
      "sku": {
        "name": "Premium",
        "tier": "Premium"
      },
      "identity": {
        "type": "UserAssigned",
        "userAssignedIdentities": {
          "[resourceID('Microsoft.ManagedIdentity/userAssignedIdentities', parameters('identity_name'))]":
{}
        }
      },
      "dependsOn": [
        "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', parameters('identity_name'))]"
      ],
      "properties": {
        "adminUserEnabled": false,
        "encryption": {
          "status": "enabled",
          "keyVaultProperties": {
            "identity": "[reference(resourceId('Microsoft.ManagedIdentity/userAssignedIdentities',
parameters('identity_name')), '2018-11-30').clientId]",
            "KeyIdentifier": "[parameters('kek_id')]"
          }
        },
        "networkRuleSet": {
          "defaultAction": "Allow",
          "virtualNetworkRules": [],
          "ipRules": []
        },
        "policies": {
          "quarantinePolicy": {
            "status": "disabled"
          },
          "trustPolicy": {
            "type": "Notary",
            "status": "disabled"
          },
          "retentionPolicy": {
            "days": 7,
            "status": "disabled"
          }
        }
      }
    },
    {
      "type": "Microsoft.KeyVault/vaults/accessPolicies",
      "apiVersion": "2018-02-14",
      "name": "[concat(parameters('vault_name'), '/add')]",
      "dependsOn": [
        "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', parameters('identity_name'))]"
      ],
      "properties": {
        "accessPolicies": [
          {
            "tenantId": "[subscription().tenantId]",
            "objectId": "[reference(resourceId('Microsoft.ManagedIdentity/userAssignedIdentities',
parameters('identity_name')), '2018-11-30').principalId]",
            "permissions": {
              "keys": [
                "get",
                "unwrapKey",
                "wrapKey"
              ]
            }
          }
        ]
      }
    },
```

```
    },
    {
      "type": "Microsoft.ManagedIdentity/userAssignedIdentities",
      "apiVersion": "2018-11-30",
      "name": "[parameters('identity_name')]",
      "location": "[resourceGroup().location]"
    }
  ]
}
```

Follow the steps in the previous sections to create the following resources:

- Key vault, identified by name
- Key vault key, identified by key ID

Run the following az deployment group create command to create the registry using the preceding template file. Where indicated, provide a new registry name and managed identity name, as well as the key vault name and key ID you created.

```
az deployment group create \
  --resource-group <resource-group-name> \
  --template-file CMKtemplate.json \
  --parameters \
    registry_name=<registry-name> \
    identity_name=<managed-identity> \
    vault_name=<key-vault-name> \
    kek_id=<key-vault-key-id>
```

Show encryption status

To show the status of registry encryption, run the az acr encryption show command:

```
az acr encryption show --name <registry-name>
```

Use the registry

After enabling a customer-managed key in a registry, you can perform the same registry operations that you perform in a registry that's not encrypted with a customer-managed key. For example, you can authenticate with the registry and push Docker images. See example commands in Push and pull an image.

Rotate key

Update the key version in Azure Key Vault, or create a new key, and then update the registry to encrypt data using the key. You can perform these steps using the Azure CLI or in the portal.

When rotating a key, typically you specify the same identity used when creating the registry. Optionally, configure a new user-assigned identity for key access, or enable and specify the registry's system-assigned identity.

> **NOTE**
>
> - To enable the registry's system-assigned identity in the portal, select **Settings** > **Identity** and set the system-assigned identity's status to **On**.
> - Ensure that the required key vault access is set for the identity you configure for key access.

Update key version

A common scenario is to update the version of the key used as a customer-managed key. Depending on how the

registry encryption is configured, the customer-managed key in Azure Container Registry is automatically updated, or must be manually updated.

Azure CLI

Use az keyvault key commands to create or manage your key vault keys. To create a new key version, run the az keyvault key create command:

```
# Create new version of existing key
az keyvault key create \
  --name <key-name> \
  --vault-name <key-vault-name>
```

The next step depends on how the registry encryption is configured:

- If the registry is configured to detect key version updates, the customer-managed key is updated automatically within 1 hour.
- If the registry is configured to require manual updating for a new key version, run the az acr encryption rotate-key command, passing the new key ID and the identity you want to configure:

To update the customer-managed key version manually:

```
# Rotate key and use user-assigned identity
az acr encryption rotate-key \
  --name <registry-name> \
  --key-encryption-key <new-key-id> \
  --identity <principal-id-user-assigned-identity>

# Rotate key and use system-assigned identity
az acr encryption rotate-key \
  --name <registry-name> \
  --key-encryption-key <new-key-id> \
  --identity [system]
```

> **TIP**
> When you run `az acr encryption rotate-key`, you can pass either a versioned key ID or a non-versioned key ID. If you use a non-versioned key ID, the registry is then configured to automatically detect later key version updates.

Portal

Use the registry's **Encryption** settings to update the key vault, key, or identity settings used for the customer-managed key.

For example, to configure a new key:

1. In the portal, navigate to your registry.
2. Under **Settings**, select **Encryption** > **Change key**.

3. In **Encryption**, choose one of the following:
   - Select **Select from Key Vault**, and select an existing key vault and key, or **Create new**. The key you select is non-versioned and enables automatic key rotation.
   - Select **Enter key URI**, and provide a key identifier directly. You can provide either a versioned key URI (for a key that must be rotated manually) or a non-versioned key URI (which enables automatic key rotation).
4. Complete the key selection and select **Save**.

Revoke key

Revoke the customer-managed encryption key by changing the access policy or permissions on the key vault or by deleting the key. For example, use the az keyvault delete-policy command to change the access policy of the managed identity used by your registry:

```
az keyvault delete-policy \
  --resource-group <resource-group-name> \
  --name <key-vault-name> \
  --object-id $identityPrincipalID
```

Revoking the key effectively blocks access to all registry data, since the registry can't access the encryption key. If access to the key is enabled or the deleted key is restored, your registry will pick the key so you can again access the encrypted registry data.

Troubleshoot

Removing managed identity

If you try to remove a user-assigned or system-assigned managed identity from a registry that is used to configure encryption, you might see an error message similar to:

```
Azure resource
'/subscriptions/xxxx/resourcegroups/myGroup/providers/Microsoft.ContainerRegistry/registries/myRegistry'
does not have access to identity 'xxxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx' Try forcibly adding the identity to
the registry <registry name>. For more information on bring your own key, please visit
'https://aka.ms/acr/cmk'.
```

You will also be unable to change (rotate) the encryption key. The resolution steps depend on the type of identity used for encryption.

User-assigned identity

If this issue occurs with a user-assigned identity, first reassign the identity using the az acr identity assign command. Pass the identity's resource ID, or use the identity's name when it is in the same resource group as the registry. For example:

```
az acr identity assign -n myRegistry \
    --identities
"/subscriptions/mysubscription/resourcegroups/myresourcegroup/providers/Microsoft.ManagedIdentity/userAssign
edIdentities/myidentity"
```

Then, after changing the key and assigning a different identity, you can remove the original user-assigned identity.

System-assigned identity

If this issue occurs with a system-assigned identity, please create an Azure support ticket for assistance to restore the identity.

Enabling key vault firewall

If you enable a key vault firewall or virtual network after creating an encrypted registry, you might see HTTP 403 or other errors with image import or automated key rotation. To correct this problem, reconfigure the managed identity and key you used initially for encryption. See steps in Rotate key.

If the problem persists, please contact Azure Support.

Accidental deletion of key vault or key

Deletion of the key vault, or the key, used to encrypt a registry with a customer-managed key will make the registry's content inaccessible. If soft delete is enabled in the key vault (the default option), you can recover a deleted vault or key vault object and resume registry operations.

For key vault deletion and recovery scenarios, see Azure Key Vault recovery management with soft delete and purge protection.

Next steps

- Learn more about encryption at rest in Azure.
- Learn more about access policies and how to secure access to a key vault.

Configure rules to access an Azure container registry behind a firewall

8/22/2022 • 5 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 to a container registry only within an Azure virtual network, see Configure Azure Private Link for an Azure container registry.

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. For clients that access a registry from behind a firewall, you need to configure access rules for both endpoints. Both endpoints are reached over port 443.

- **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. Example: `myregistry.azurecr.io`
- **Storage (data) 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 the data endpoint in a specific region or in multiple replicated regions.

Allow access to REST and data endpoints

- **REST endpoint** - Allow access to the fully qualified registry login server name, `<registry-name>.azurecr.io`, or an associated IP address range
- **Storage (data) endpoint** - Allow access to all Azure blob storage accounts using the wildcard `*.blob.core.windows.net`, or an associated IP address range.

> **NOTE**
>
> Azure Container Registry is introducing dedicated data endpoints, allowing you to tightly scope client firewall rules for your registry storage. Optionally enable data endpoints in all regions where the registry is located or replicated, using the form `<registry-name>.<region>.data.azurecr.io`.

About Registry FQDN's

Registry has two FQDN's, the **login url** and the **data endpoint**.

- Both the **login url** and the **data endpoint** are accessible from within the virtual network, using private IP's by enabling a private link.
- A registry that does not use data endpoints would have to access the data from an endpoint of the form

`*.blob.core.windows.net` and does not provide the isolation required when configuring firewall rules.

- A registry with a private link enabled gets the dedicated data endpoint automatically.
- A dedicated data endpoint is created per region for a registry.
- Login url remains the same irrespective of whether data endpoint is enabled or disabled.

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 or you use Azure Firewall, 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*].

Enable dedicated data endpoints

> **WARNING**
> If you previously configured client firewall access to the existing `*.blob.core.windows.net` endpoints, switching to dedicated data endpoints will impact client connectivity, causing pull failures. To ensure clients have consistent access, add the new data endpoint rules to the client firewall rules. Once completed, enable dedicated data endpoints for your registries using the Azure CLI or other tools.

Dedicated data endpoints is an optional feature of the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry service tiers.

You can enable dedicated data endpoints using the Azure portal or the Azure CLI. The data endpoints follow a regional pattern, `<registry-name>.<region>.data.azurecr.io`. In a geo-replicated registry, enabling data endpoints enables endpoints in all replica regions.

Portal

To enable data endpoints using the portal:

1. Navigate to your container registry.
2. Select **Networking** > **Public access**.
3. Select the **Enable dedicated data endpoint** checkbox.
4. Select **Save**.

The data endpoint or endpoints appear in the portal.

Azure CLI

To enable data endpoints using the Azure CLI, use Azure CLI version 2.4.0 or higher. If you need to install or upgrade, see Install Azure CLI.

The following az acr update command enables dedicated data endpoints on a registry *myregistry*.

```
az acr update --name myregistry --data-endpoint-enabled
```

To view the data endpoints, use the az acr show-endpoints command:

```
az acr show-endpoints --name myregistry
```

Output for demonstration purposes shows two regional endpoints

```
{
    "loginServer": "myregistry.azurecr.io",
    "dataEndpoints": [
        {
            "region": "eastus",
            "endpoint": "myregistry.eastus.data.azurecr.io",
        },
        {
            "region": "westus",
            "endpoint": "myregistry.westus.data.azurecr.io",
        }
    ]
}
```

After you set up dedicated data endpoints for your registry, you can enable client firewall access rules for the data endpoints. Enable data endpoint access rules for all required registry regions.

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
- Learn more about setting up Private Link for a container registry
- Learn more about dedicated data endpoints for Azure Container Registry

Authenticate with an Azure container registry

8/22/2022 • 8 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:

- Authenticate to a registry directly via individual login
- Applications and container orchestrators can perform unattended, or "headless," authentication by using an Azure Active Directory (Azure AD) service principal

If you use a container registry with Azure Kubernetes Service (AKS) or another Kubernetes cluster, see Scenarios to authenticate with Azure Container Registry from Kubernetes.

Authentication options

The following table lists available authentication methods and typical scenarios. See linked content for details.

METHOD	HOW TO AUTHENTICATE	SCENARIOS	AZURE ROLE-BASED ACCESS CONTROL (AZURE RBAC)	LIMITATIONS
Individual AD identity	`az acr login` in Azure CLI `Connect-AzContainerRegistry` in Azure PowerShell	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 `Connect-AzContainerRegistry` in Azure PowerShell 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
Managed identity for Azure resources	`docker login` `az acr login` in Azure CLI `Connect-AzContainerRegistry` in Azure PowerShell	Unattended push from Azure CI/CD pipeline Unattended pull to Azure services	Yes	Use only from select Azure services that support managed identities for Azure resources

METHOD	HOW TO AUTHENTICATE	SCENARIOS	AZURE ROLE-BASED ACCESS CONTROL (AZURE RBAC)	LIMITATIONS
AKS cluster managed identity	Attach registry when AKS cluster created or updated	Unattended pull to AKS cluster in the same or a different subscription	No, pull access only	Only available with AKS cluster Can't be used for cross-tenant authentication
AKS cluster service principal	Enable when AKS cluster created or updated	Unattended pull to AKS cluster from registry in another AD tenant	No, pull access only	Only available with AKS cluster
Admin user	`docker login`	Interactive push/pull by individual developer or tester Portal deployment of image from registry to Azure App Service or Azure Container Instances	No, always pull and push access	Single account per registry, not recommended for multiple users
Repository-scoped access token	`docker login` `az acr login` in Azure CLI `Connect-AzContainerRegistry` in Azure PowerShell Kubernetes pull secret	Interactive push/pull to repository by individual developer or tester Unattended pull from repository by individual system or external device	Yes	Not currently integrated with AD identity

Individual login with Azure AD

- Azure CLI
- Azure PowerShell

When working with your registry directly, such as pulling images to and pushing images from a development workstation to a registry you created, authenticate by using your individual Azure identity. Sign in to the Azure CLI with az login, and then run the az acr login command:

```
az login
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, the Docker CLI and Docker daemon 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 Azure role-based access control (Azure RBAC). For some scenarios, you may want to log in to a registry with your own individual identity in Azure AD, or configure other Azure users with specific Azure roles and permissions. 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.

az acr login with --expose-token

In some cases, you need to authenticate with `az acr login` when the Docker daemon isn't running in your environment. For example, you might need to run `az acr login` in a script in Azure Cloud Shell, which provides the Docker CLI but doesn't run the Docker daemon.

For this scenario, run `az acr login` first with the `--expose-token` parameter. This option exposes an access token instead of logging in through the Docker CLI.

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

Output displays the access token, abbreviated here:

```
{
  "accessToken": "eyJhbGciOiJSUzI1NiIs[...]24V7wA",
  "loginServer": "myregistry.azurecr.io"
}
```

For registry authentication, we recommend that you store the token credential in a safe location and follow recommended practices to manage docker login credentials. For example, store the token value in an environment variable:

```
TOKEN=$(az acr login --name <acrName> --expose-token --output tsv --query accessToken)
```

Then, run `docker login`, passing `00000000-0000-0000-0000-000000000000` as the username and using the access token as password:

```
docker login myregistry.azurecr.io --username 00000000-0000-0000-0000-000000000000 --password $TOKEN
```

Likewise, you can use the token returned by `az acr login` with the `helm registry login` command to authenticate with the registry:

```
echo $TOKEN | helm registry login myregistry.azurecr.io \
            --username 00000000-0000-0000-0000-000000000000 \
            --password-stdin
```

Service principal

If you assign a service principal to your registry, your application or service can use it for headless authentication. Service principals allow Azure role-based access control (Azure RBAC) 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.

ACR authentication token gets created upon login to the ACR, and is refreshed upon subsequent operations. The time to live for that token is 3 hours.

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, Azure PowerShell, or other Azure tools. The admin account has full permissions to the registry.

The admin account is currently required for some scenarios to deploy an image from a container registry to certain Azure services. For example, the admin account is needed when you use the Azure portal to deploy a container image from a registry directly to Azure Container Instances or Azure Web Apps for Containers.

> **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. New passwords created for admin accounts are available immediately. Regenerating passwords for admin accounts will take 60 seconds to replicate and be available. 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 recommended practices to manage login credentials, see the docker login command reference.

- Azure CLI
- Azure PowerShell

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
- Push your first image using Azure PowerShell

Azure Container Registry authentication with service principals

8/22/2022 • 8 minutes to read • Edit Online

You can use an Azure Active Directory (Azure AD) service principal to provide push, pull, or other 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, an 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 Container Instances, App Service, Batch, Service Fabric, and others.

  > **TIP**
  >
  > A service principal is recommended in several Kubernetes scenarios to pull images from an Azure container registry. With Azure Kubernetes Service (AKS), you can also use an automated mechanism to authenticate with a target registry by enabling the cluster's managed identity.

  - *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
# This script requires Azure CLI version 2.25.0 or later. Check version with `az --version`.

# 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=$containerRegistry
SERVICE_PRINCIPAL_NAME=$servicePrincipal

# Obtain the full registry ID
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query "id" --output tsv)
# echo $registryId

# 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
PASSWORD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role acrpull
--query "password" --output tsv)
USER_NAME=$(az ad sp list --display-name $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: $USER_NAME"
echo "Service principal password: $PASSWORD"
```

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=$containerRegistry
SERVICE_PRINCIPAL_ID=$servicePrincipal

# 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's **application (client) ID**
- **Password** - service principal's **password (client secret)**

Each value has the format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.

> **TIP**
> You can regenerate the password (client secret) of a service principal by running the az ad sp credential reset 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 recommended 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.

Create service principal for cross-tenant scenarios

A service principal can also be used in Azure scenarios that require pulling images from a container registry in one Azure Active Directory (tenant) to a service or app in another. For example, an organization might run an app in Tenant A that needs to pull an image from a shared container registry in Tenant B.

To create a service principal that can authenticate with a container registry in a cross-tenant scenario:

- Create a multitenant app (service principal) in Tenant A
- Provision the app in Tenant B
- Grant the service principal permissions to pull from the registry in Tenant B
- Update the service or app in Tenant A to authenticate using the new service principal

For example steps, see Pull images from a container registry to an AKS cluster in a different AD tenant.

Service principal renewal

The service principal is created with one-year validity. You have options to extend the validity further than one year, or can provide expiry date of your choice using the `az ad sp credential reset` command.

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

8/22/2022 • 12 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. Or, set up an Azure Kubernetes Service cluster to use its managed identity to pull container images from Azure Container Registry for pod deployments.

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

- Azure CLI
- Azure PowerShell

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?

If you're not familiar with the managed identities for Azure resources feature, see this overview.

After you set up selected Azure resources 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. Choose how to authenticate using the managed identity, depending on your scenario. To use the identity to access an Azure container registry from a virtual machine, you authenticate with Azure Resource Manager.

Create a container registry

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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 update
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 mcr.microsoft.com/hello-world
```

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

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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-
xxxxxxxxxxxxx/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myAC
RId
```

Configure the VM with the identity

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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

- Azure CLI
- Azure PowerShell

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.
- Learn how to use a system-assigned or user-assigned managed identity with App Service and Azure Container Registry.
- Learn how to deploy a container image from Azure Container Registry using a managed identity.

Create a token with repository-scoped permissions

8/22/2022 • 13 minutes to read • Edit Online

This article describes how to create tokens and scope maps to manage access to specific repositories 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.

This feature is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry service tiers.

> **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

- You can't currently assign repository-scoped permissions to an Azure Active Directory identity, such as a service principal or managed identity.
- You can't create a scope map in a registry enabled for anonymous pull access.

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 when creating tokens. The permissions of system-defined scope maps apply to all repositories in your registry.The individual *actions* corresponds to the limit of Repositories per scope map.

The following image shows the relationship between tokens and scope maps.

Prerequisites

- **Azure CLI** - Azure CLI commands command examples in this article require Azure CLI version 2.17.0 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 \
  --output json
```

The output shows details about the token. By default, two passwords are generated that don't expire, but you can optionally set an expiration date. 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/toke
ns/MyToken",
  "name": "MyToken",
  "objectId": null,
  "provisioningState": "Succeeded",
  "resourceGroup": "myresourcegroup",
  "scopeMapId": "/subscriptions/xxxxxxxx-adbd-4cb4-c864-
xxxxxxxxxxxx/resourceGroups/myresourcegroup/providers/Microsoft.ContainerRegistry/registries/myregistry/scop
eMaps/MyToken-scope-map",
  "status": "enabled",
  "type": "Microsoft.ContainerRegistry/registries/tokens"
```


> **NOTE**
>
> To regenerate token passwords and expiration periods, see Regenerate token passwords later in this article.

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. By default, two passwords are generated. 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.

> **NOTE**
>
> To regenerate token passwords and expiration periods, see Regenerate token passwords later in this article.

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 **Repository permissions**, 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

To use a token created in the portal, you must generate a password. You can generate one or two passwords, and set an expiration date for each one. New passwords created for tokens are available immediately. Regenerating new passwords for tokens will take 60 seconds to replicate and be available.

1. In the portal, navigate to your container registry.

2. Under **Repository permissions**, 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**. It's recommended to set an expiration date.

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 Example: `az acr repository delete --name myregistry --repository myrepo --username MyToken --password xxxxxxxxxx`

`content/read`	`docker login`
	
	`az acr login` in Azure CLI
	
	Example:
	`az acr login --name myregistry --username MyToken --password xxxxxxxxxx`

`content/write`	`docker login` `az acr login` in Azure CLI

`metadata/read`	`az acr repository show`
	
	`az acr repository show-tags`
	
	`az acr manifest list-metadata` 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 public `hello-world` and `nginx` images from Microsoft Container Registry, and tag them for your registry and repository.

```
docker pull mcr.microsoft.com/hello-world
docker pull mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
docker tag mcr.microsoft.com/hello-world myregistry.azurecr.io/samples/hello-world:v1
docker tag mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine myregistry.azurecr.io/samples/nginx:v1
```

Authenticate using token

Run `docker login` or `az acr login` to authenticate with the registry to push or pull images. 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/nginx` 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/nginx:v1
```

Update token 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/ngnx` 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-repository samples/nginx content/write content/read \
  --remove-repository samples/hello-world content/write
```

In the Azure portal:

1. Navigate to your container registry.
2. Under **Repository permissions**, select **Scope maps (Preview)**, and select the scope map to update.
3. Under **Repositories**, enter `samples/nginx`, 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/nginx: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/nginx:v1
docker pull myregistry.azurecr.io/samples/hello-world:v1
```

Delete images

Update the scope map by adding the `content/delete` action to the `nginx` 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-repository samples/nginx content/delete
```

To update the scope map using the portal, see the previous section.

Use the following az acr repository delete command to delete the `samples/nginx` repository. To delete images or repositories, 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/nginx \
  --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-repository samples/hello-world metadata/read
```

To update the scope map using the portal, see the previous section.

To read metadata in the `samples/hello-world` repository, run the az acr manifest list-metadata or az acr repository show-tags command.

To read metadata, 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 consists of the three system-defined scope maps and other scope maps generated by you. Tokens can be configured with any of these scope maps.

```
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
```

Regenerate token passwords

If you didn't generate a token password, or you want to generate new passwords, run the az acr token credential generate command.Regenerating new passwords for tokens will take 60 seconds to replicate and be available.

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 --expiration-in-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.
- Learn about connected registries and using tokens for access.

Authenticate with Azure Container Registry from Azure Container Instances

8/22/2022 • 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
# This script requires Azure CLI version 2.25.0 or later. Check version with `az --version`.

# 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=$containerRegistry
SERVICE_PRINCIPAL_NAME=$servicePrincipal

# Obtain the full registry ID
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query "id" --output tsv)
# echo $registryId

# 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
PASSWORD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role acrpull
--query "password" --output tsv)
USER_NAME=$(az ad sp list --display-name $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: $USER_NAME"
echo "Service principal password: $PASSWORD"
```

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=$containerRegistry
SERVICE_PRINCIPAL_ID=$servicePrincipal

# 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>
```

> **NOTE**
>
> We recommend running the commands in the most recent version of the Azure Cloud Shell. Set `export MSYS_NO_PATHCONV=1` for running on-perm bash environment.

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)

Scenarios to authenticate with Azure Container Registry from Kubernetes

8/22/2022 • 2 minutes to read • Edit Online

You can use an Azure container registry as a source of container images for Kubernetes, including clusters you manage, managed clusters hosted in Azure Kubernetes Service (AKS) or other clouds, and "local" Kubernetes configurations such as minikube and kind.

To pull images to your Kuberentes cluster from an Azure container registry, an authentication and authorization mechanism needs to be established. Depending on your cluster environment, choose one of the following methods:

Scenarios

KUBERNETES CLUSTER	AUTHENTICATION METHOD	DESCRIPTION	EXAMPLE
AKS cluster	AKS managed identity	Enable the AKS kubelet managed identity to pull images from an attached Azure container registry. Registry and cluster must be in same Active Directory tenant but can be in the same or a different Azure subscription.	Authenticate with Azure Container Registry from Azure Kubernetes Service
AKS cluster	AKS service principal	Enable the AKS service principal with permissions to a target Azure container registry. Registry and cluster can be in the same or a different Azure subscription or Azure Active Directory tenant.	Pull images from an Azure container registry to an AKS cluster in a different AD tenant
Kubernetes cluster other than AKS	Pod imagePullSecrets	Use general Kubernetes mechanism to manage registry credentials for pod deployments. Configure AD service principal, repository-scoped token, or other supported registry credentials.	Pull images from an Azure container registry to a Kubernetes cluster using a pull secret

Next steps

- Learn more about how to authenticate with an Azure container registry

Authenticate with Azure Container Registry from Azure Kubernetes Service

8/22/2022 • 4 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 operation is implemented as part of the CLI, PowerShell, and Portal experience by granting the required permissions to your ACR. 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 or Azure PowerShell. This integration assigns the AcrPull role to the managed identity associated to the AKS Cluster.

> **NOTE**
> This article covers automatic authentication between AKS and ACR. If you need to pull an image from a private external registry, use an image pull secret.

Before you begin

These examples require:

- Azure CLI
- Azure PowerShell

- **Owner**, **Azure account administrator**, or **Azure co-administrator** role on the **Azure subscription**
- Azure CLI version 2.7.0 or later

To avoid needing an **Owner**, **Azure account administrator**, or **Azure co-administrator** role, you can use an existing managed identity to authenticate ACR from AKS. For more information, see Use an Azure managed identity to authenticate to an Azure container registry.

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 **managed identity** is used. The following command allows you to authorize an existing ACR in your subscription and configures the appropriate **ACRPull** role for the managed identity. Supply valid values for your parameters below.

- Azure CLI
- Azure PowerShell

```
# 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 the following format:

```
/subscriptions/\<subscription-id\>/resourceGroups/\<resource-group-
name\>/providers/Microsoft.ContainerRegistry/registries/\<name\>
```

> **NOTE**
> If you are using an ACR that is located in a different subscription from your AKS cluster, use the ACR resource ID when attaching or detaching from an AKS cluster.

```
az aks create -n myAKSCluster -g myResourceGroup --generate-ssh-keys --attach-acr
/subscriptions/<subscription-
id>/resourceGroups/myContainerRegistryResourceGroup/providers/Microsoft.ContainerRegistry/registries/myConta
inerRegistry
```

This step may take several minutes to complete.

Configure ACR integration for existing AKS clusters

- Azure CLI
- Azure PowerShell

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 <acr-name>
```

or,

```
az aks update -n myAKSCluster -g myResourceGroup --attach-acr <acr-resource-id>
```

> **NOTE**
> Running `az aks update --attach-acr` uses the permissions of the user running the command to create the role ACR assignment. This role is assigned to the kubelet managed identity. For more information on the AKS managed identities, see Summary of managed identities.

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 <acr-name>
```

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:

- Azure CLI

- Azure PowerShell

```
az acr import  -n <acr-name> --source docker.io/library/nginx:latest --image nginx:v1
```

Deploy the sample image from ACR to AKS

Ensure you have the proper AKS credentials

- Azure CLI
- Azure PowerShell

```
az aks get-credentials -g myResourceGroup -n myAKSCluster
```

Create a file called **acr-nginx.yaml** that contains the following. Substitute the resource name of your registry for **acr-name**. Example: *myContainerRegistry*.

```
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: <acr-name>.azurecr.io/nginx:v1
        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
```

Troubleshooting

- Run the az aks check-acr command to validate that the registry is accessible from the AKS cluster.

- Learn more about ACR Monitoring
- Learn more about ACR Health

Pull images from a container registry to an AKS cluster in a different Azure AD tenant

8/22/2022 • 2 minutes to read • Edit Online

In some cases, you might have your Azure AKS cluster in one Azure Active Directory (Azure AD) tenant and your Azure container registry in a different tenant. This article walks through the steps to enable cross-tenant authentication using the AKS service principal credential to pull from the container registry.

> **NOTE**
> You can't attach the registry and authenticate using an AKS managed identity when the cluster and the container registry are in different tenants.

Scenario overview

Assumptions for this example:

- The AKS cluster is in **Tenant A** and the Azure container registry is in **Tenant B**.
- The AKS cluster is configured with service principal authentication in **Tenant A**. Learn more about how to create and use a service principal for your AKS cluster.

You need at least the Contributor role in the AKS cluster's subscription and the Owner role in the container registry's subscription.

You use the following steps to:

- Create a new multitenant app (service principal) in **Tenant A**.
- Provision the app in **Tenant B**.
- Configure the service principal to pull from the registry in **Tenant B**
- Update the AKS cluster in **Tenant A** to authenticate using the new service principal

Step-by-step instructions

Step 1: Create multitenant Azure AD application

1. Sign in to the Azure portal in **Tenant A**.
2. Search for and select **Azure Active Directory**.
3. Under **Manage**, select **App registrations** > **+ New registration**.
4. In **Supported account types**, select **Accounts in any organizational directory**.
5. In **Redirect URI**, enter *https://www.microsoft.com*.
6. Select **Register**.
7. On the **Overview** page, take note of the **Application (client) ID**. It will be used in Step 2 and Step 4.

8. In **Certificates & secrets**, under **Client secrets**, select **+ New client secret**.

9. Enter a **Description** such as *Password* and select **Add**.

10. In **Client secrets**, take note of the value of the client secret. You use it to update the AKS cluster's service principal in Step 4.

Step 2: Provision the service principal in the ACR tenant

1. Open the following link using an admin account in **Tenant B**. Where indicated, insert the **ID of Tenant B** and the **application ID** (client ID) of the multitenant app.

```
https://login.microsoftonline.com/<Tenant B ID>/oauth2/authorize?client_id=<Multitenant application ID>&response_type=code&redirect_uri=<redirect url>
```

2. Select **Consent on behalf of your organization** and then **Accept**.

Step 3: Grant service principal permission to pull from registry

In **Tenant B**, assign the AcrPull role to the service principal, scoped to the target container registry. You can use the Azure portal or other tools to assign the role. For example steps using the Azure CLI, see Azure Container Registry authentication with service principals.

Step 4: Update AKS with the Azure AD application secret

Use the multitenant application (client) ID and client secret collected in Step 1 to update the AKS service principal credential.

Updating the service principal can take several minutes.

Next steps

- Learn more Azure Container Registry authentication with service principals
- Learn more about image pull secrets in the Kubernetes documentation
- Learn about Application and service principal objects in Azure Active Directory

- Learn more about scenarios to authenticate with Azure Container Registry from a Kubernetes cluster

Pull images from an Azure container registry to a Kubernetes cluster using a pull secret

8/22/2022 • 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 using credentials for an Azure container registry. Then, use the secret to pull images from an Azure container registry in a pod deployment.

This example creates a pull secret using Azure Active Directory service principal credentials. You can also configure a pull secret using other Azure container registry credentials, such as a repository-scoped access token.

> **NOTE**
> While pull secrets are commonly used, they bring additional management overhead. If you're using Azure Kubernetes Service, we recommend other options such as using the cluster's managed identity or service principal to securely pull the image without an additional `imagePullSecrets` setting on each pod.

Prerequisites

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
# This script requires Azure CLI version 2.25.0 or later. Check version with `az --version`.

# 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=$containerRegistry
SERVICE_PRINCIPAL_NAME=$servicePrincipal

# Obtain the full registry ID
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query "id" --output tsv)
# echo $registryId

# 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
PASSWORD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role acrpull
--query "password" --output tsv)
USER_NAME=$(az ad sp list --display-name $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: $USER_NAME"
echo "Service principal password: $PASSWORD"
```

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=$containerRegistry
SERVICE_PRINCIPAL_ID=$servicePrincipal

# 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=<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, for example, *myregistry* The `--docker-server` is the fully qualified name of the registry login server
`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: my-awesome-app-pod
  namespace: awesomeapps
spec:
  containers:
    - name: main-app-container
      image: myregistry.azurecr.io/my-awesome-app:v1
      imagePullPolicy: IfNotPresent
  imagePullSecrets:
    - name: acr-secret
```

In the preceding example, `my-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

Make your container registry content publicly available

8/22/2022 • 2 minutes to read • Edit Online

Setting up an Azure container registry for anonymous (unauthenticated) pull access is an optional feature that allows any user with internet access the ability to pull any content from the registry.

Anonymous pull access is a preview feature, available in the Standard and Premium service tiers. To configure anonymous pull access, update a registry using the Azure CLI (version 2.21.0 or later). To install or upgrade, see Install Azure CLI.

About anonymous pull access

By default, access to pull or push content from an Azure container registry is only available to authenticated users. Enabling anonymous (unauthenticated) pull access makes all registry content publicly available for read (pull) actions. Anonymous pull access can be used in scenarios that do not require user authentication such as distributing public container images.

- Enable anonymous pull access by updating the properties of an existing registry.
- After enabling anonymous pull access, you may disable that access at any time.
- Only data-plane operations are available to unauthenticated clients.
- The registry may throttle a high rate of unauthenticated requests.
- If you previously authenticated to the registry, make sure you clear the credentials before attempting an anonymous pull operation.

> **WARNING**
>
> Anonymous pull access currently applies to all repositories in the registry. If you manage repository access using repository-scoped tokens, all users may pull from those repositories in a registry enabled for anonymous pull. We recommend deleting tokens when anonymous pull access is enabled.

Configure anonymous pull access

Enable anonymous pull access

Update a registry using the az acr update command and pass the `--anonymous-pull-enabled` parameter. By default, anonymous pull is disabled in the registry.

```
az acr update --name myregistry --anonymous-pull-enabled
```

> **IMPORTANT**
>
> If you previously authenticated to the registry with Docker credentials, run `docker logout` to ensure that you clear the existing credentials before attempting anonymous pull operations. Otherwise, you might see an error message similar to "pull access denied".

Disable anonymous pull access

Disable anonymous pull access by setting `--anonymous-pull-enabled` to `false`.

```
az acr update --name myregistry --anonymous-pull-enabled false
```

Next steps

- Learn about using repository-scoped tokens.
- Learn about options to authenticate to an Azure container registry.

Azure Container Registry roles and permissions

8/22/2022 • 4 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 (Azure RBAC) to assign specific permissions to users, service principals, or other identities that need to interact with a registry, for example to pull or push container images. You can also define custom roles with fine-grained permissions to a registry for different operations.

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

Assign roles

See Steps to add a role assignment for high-level steps to add a role assignment to an existing user, group, service principal, or managed identity. You can use the Azure portal, Azure CLI, Azure PowerShell, or other Azure tools.

When creating a service principal, you also configure its access and permissions to Azure resources such as a container registry. For an example script using the Azure CLI, see Azure Container Registry authentication with service principals.

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 recommend 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 CLI
- Azure PowerShell

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 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:

- Azure CLI

- Azure PowerShell

```
az provider operation show --namespace Microsoft.ContainerRegistry
```

To define a custom role, see Steps to create a custom role.

> **NOTE**
> In tenants configured with Azure Resource Manager private link, Azure Container Registry supports wildcard actions such as `Microsoft.ContainerRegistry/*/read` or `Microsoft.ContainerRegistry/registries/*/write` in custom roles, granting access to all matching actions. In a tenant without an ARM private link, specify all required registry actions individually in a custom role.

Example: Custom role to import images

For example, the following JSON defines the minimum actions for a custom role that permits importing images to a registry.

```
{
   "assignableScopes": [
     "/subscriptions/<optional, but you can limit the visibility to one or more subscriptions>"
   ],
   "description": "Can import images to registry",
   "Name": "AcrImport",
   "permissions": [
     {
       "actions": [
         "Microsoft.ContainerRegistry/registries/push/write",
         "Microsoft.ContainerRegistry/registries/pull/read",
         "Microsoft.ContainerRegistry/registries/read",
         "Microsoft.ContainerRegistry/registries/importImage/action"
       ],
       "dataActions": [],
       "notActions": [],
       "notDataActions": []
     }
   ],
   "roleType": "CustomRole"
 }
```

To create or update a custom role using the JSON description, use the Azure CLI, Azure Resource Manager template, Azure PowerShell, or other Azure tools. Add or remove role assignments for a custom role in the same way that you manage role assignments for built-in Azure roles.

Next steps

- Learn more about assigning Azure roles to an Azure identity by using the Azure portal, the Azure CLI, Azure PowerShell, or other Azure tools.
- Learn about authentication options for Azure Container Registry.
- Learn about enabling repository-scoped permissions (preview) in a container registry.

Content trust in Azure Container Registry

8/22/2022 • 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 service tier 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.

> **IMPORTANT**
>
> You can't grant trusted image push permission to the following administrative accounts:
>
> - the admin account of an Azure container registry
> - a user account in Azure Active Directory with the classic system administrator role.

> **NOTE**
>
> Starting July 2021, the `AcrImageSigner` role includes both the `Microsoft.ContainerRegistry/registries/sign/write` action and the `Microsoft.ContainerRegistry/registries/trustedCollections/write` data action.

Details for granting the `AcrImageSigner` role in the Azure portal and the Azure CLI follow.

Azure portal

1. Select **Access control (IAM)**.
2. Select **Add** > **Add role assignment** to open the Add role assignment page.
3. Assign the following role. In this example, the role is assigned to an individual user. For detailed steps, see Assign Azure roles using the Azure portal.

SETTING	VALUE
Role	AcrImageSigner
Assign access to	User
Members	Alain

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 a non-administrative user 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
REGISTRY_ID=$(az acr show --name $REGISTRY --query id --output tsv)
```

```
az role assignment create --scope $REGISTRY_ID --role AcrImageSigner --assignee azureuser@contoso.com
```

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 Add or remove Azure role assignments using Azure CLI and Troubleshoot Azure RBAC.

Push a trusted image

To push a trusted image tag to your container registry, enable content trust and push the image with `docker push`. After push with a signed tag completes the first time, 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 with an error similar to the following:

```
$ docker pull myregistry.azurecr.io/myimage:unsigned
Error: remote trust data does not exist
```

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.

> **NOTE**
> Azure Container Registry does not officially support the Notary CLI but is compatible with the Notary Server API, which is included with Docker Desktop. Currently Notary version **0.6.0** is recommended.

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, including docker trust commands and trust delegations. 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.

Tutorial: Create a container image for deployment to Azure Container Instances

8/22/2022 • 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)

8/22/2022 • 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 later 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.

> **NOTE**
> 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 consisting 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. The contents and structure of the directory are as follows:

```
azure-voting-app-redis
│   azure-vote-all-in-one-redis.yaml
│   docker-compose.yaml
│   LICENSE
│   README.md
│
├───azure-vote
│   │   app_init.supervisord.conf
│   │   Dockerfile
│   │   Dockerfile-for-app-service
│   │   sshd_config
│   │
│   └───azure-vote
│       │   config_file.cfg
│       │   main.py
│       │
│       ├───static
│       │       default.css
│       │
│       └───templates
│               index.html
│
└───jenkins-tutorial
        config-jenkins.sh
        deploy-jenkins-vm.sh
```

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
mcr.microsoft.com/azuredocs/azure-vote-front    v1                  84b41c268ad9        9 seconds ago
944MB
mcr.microsoft.com/oss/bitnami/redis             6.0.8               3a54a920bb6c        2 days ago
103MB
tiangolo/uwsgi-nginx-flask                      python3.6           a16ce562e863        6 weeks ago
944MB
```

Run the docker ps command to see the running containers:

```
$ docker ps

CONTAINER ID        IMAGE                                           COMMAND                  CREATED
STATUS              PORTS                           NAMES
d10e5244f237        mcr.microsoft.com/azuredocs/azure-vote-front:v1   "/entrypoint.sh /sta…"   3 minutes ago
Up 3 minutes        443/tcp, 0.0.0.0:8080->80/tcp   azure-vote-front
21574cb38c1f        mcr.microsoft.com/oss/bitnami/redis:6.0.8         "/opt/bitnami/script…"   3 minutes ago
Up 3 minutes        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

8/22/2022 • 4 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 *azure-vote-front* image you just created.

```
docker images
```

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   913MB
```

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      913MB
<acrName>.azurecr.io/azure-vote-front  v1                  052c549a75bf        23 minutes ago      913MB
```

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

Monitor Azure Container Registry

8/22/2022 • 6 minutes to read • Edit Online

When you have critical applications and business processes relying on Azure resources, you want to monitor those resources for their availability, performance, and operation. This article describes the monitoring data generated by Azure Container Registry and how you can use the features of Azure Monitor to analyze and alert on this data.

Monitor overview

The **Overview** page in the Azure portal for each registry includes a brief view of recent resource usage and activity, such as push and pull operations. This high-level information is useful, but only a small amount of data is shown there.

What is Azure Monitor?

Azure Container Registry creates monitoring data using Azure Monitor, which is a full stack monitoring service in Azure that provides a complete set of features to monitor your Azure resources in addition to resources in other clouds and on-premises.

Start with the article Monitoring Azure resources with Azure Monitor, which describes the following concepts:

- What is Azure Monitor?
- Costs associated with monitoring
- Monitoring data collected in Azure

- Configuring data collection
- Standard tools in Azure for analyzing and alerting on monitoring data

The following sections build on this article by describing the specific data gathered for Azure Container Registry and providing examples for configuring data collection and analyzing this data with Azure tools.

Monitoring data

Azure Container Registry collects the same kinds of monitoring data as other Azure resources that are described in Monitoring data from Azure resources.

See Monitoring Azure Container Registry data reference for detailed information on the metrics and logs created by Azure Container Registry.

Collection and routing

Platform metrics and the Activity log are collected and stored automatically, but can be routed to other locations by using a diagnostic setting.

Resource Logs are not collected and stored until you create a diagnostic setting and route them to one or more locations.

See Create diagnostic setting to collect platform logs and metrics in Azure for the detailed process for creating a diagnostic setting using the Azure portal, CLI, or PowerShell. When you create a diagnostic setting, you specify which categories of logs to collect. The categories for Azure Container Registry are listed in Azure Container Registry monitoring data reference.

> **TIP**
> You can also create registry diagnostic settings by navigating to your registry in the portal. In the menu, select **Diagnostic settings** under **Monitoring**.

The following image shows the options when you enable diagnostic settings for a registry.

The metrics and logs you can collect are discussed in the following sections.

Analyzing metrics (preview)

You can analyze metrics for an Azure container registry with metrics from other Azure services using metrics explorer by opening **Metrics** from the **Azure Monitor** menu. See Getting started with Azure Metrics Explorer for details on using this tool.

> **TIP**
> You can also go to the metrics explorer by navigating to your registry in the portal. In the menu, select **Metrics (preview)** under **Monitoring**.

For a list of the platform metrics collected for Azure Container Registry, see Monitoring Azure Container Registry data reference metrics

For reference, you can see a list of all resource metrics supported in Azure Monitor.

Azure CLI

The following Azure CLI commands can be used to get information about the Azure Container Registry metrics.

- az monitor metrics list-definitions - List metric definitions and dimensions
- az monitor metrics list - Retrieve metric values

REST API

You can use the Azure Monitor REST API to get information programmatically about the Azure Container Registry metrics.

- List metric definitions and dimensions
- Retrieve metric values

Analyzing logs

Data in Azure Monitor Logs is stored in tables where each table has its own set of unique properties.

All resource logs in Azure Monitor have the same fields followed by service-specific fields. The common schema is outlined in Azure Monitor resource log schema. The schema for Azure Container Registry resource logs is found in the Azure Container Registry Data Reference.

The Activity log is a platform log in Azure that provides insight into subscription-level events. You can view it independently or route it to Azure Monitor Logs, where you can do much more complex queries using Log Analytics.

For a list of the types of resource logs collected for Azure Container Registry, see Monitoring Azure Container Registry data reference.

For a list of the tables used by Azure Monitor Logs and queryable by Log Analytics, see Monitoring Azure Container Reference data reference.

Sample Kusto queries

> **IMPORTANT**
>
> When you select **Logs** from the **Azure Container Registry** menu, Log Analytics is opened with the query scope set to the current registry. This means that log queries will only include data from that resource. If you want to run a query that includes data from other registries or data from other Azure services, select **Logs** from the **Azure Monitor** menu. See Log query scope and time range in Azure Monitor Log Analytics for details.

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:

Following are queries that you can use to help you monitor your registry resource.

Error events from the last hour

```
union Event, Syslog // Event table stores Windows event records, Syslog stores Linux records
| where TimeGenerated > ago(1h)
| where EventLevelName == "Error" // EventLevelName is used in the Event (Windows) records
 or SeverityLevel== "err" // SeverityLevel is used in Syslog (Linux) records
```

100 most recent registry events

```
ContainerRegistryRepositoryEvents
| union ContainerRegistryLoginEvents
| top 100 by TimeGenerated
| project TimeGenerated, LoginServer, OperationName, Identity, Repository, DurationMs, Region , ResultType
```

Identity of user or object that deleted repository

```
ContainerRegistryRepositoryEvents
| where OperationName contains "Delete"
| project LoginServer, OperationName, Repository, Identity, CallerIpAddress
```

Identity of user or object that deleted tag

```
ContainerRegistryRepositoryEvents
| where OperationName contains "Untag"
| project LoginServer, OperationName, Repository, Tag, Identity, CallerIpAddress
```

Repository-level operation failures

```
ContainerRegistryRepositoryEvents
| where ResultDescription contains "40"
| project TimeGenerated, OperationName, Repository, Tag, ResultDescription
```

Registry authentication failures

```
ContainerRegistryLoginEvents
| where ResultDescription != "200"
| project TimeGenerated, Identity, CallerIpAddress, ResultDescription
```

Alerts

Azure Monitor alerts proactively notify you when important conditions are found in your monitoring data. They allow you to identify and address issues in your system before your customers notice them. You can set alerts on metrics, logs, and the activity log. Different types of alerts have benefits and drawbacks.

The following table lists common and recommended alert rules for Azure Container Registry.

ALERT TYPE	CONDITION	DESCRIPTION
metric	Signal: Storage used Operator: Greater than Aggregation type: Average Threshold value: 5 GB	Alerts if the registry storage used exceeds a specified value.

Example: Send email alert when registry storage used exceeds a value

1. In the Azure portal, navigate to your registry.
2. Select **Metrics (preview)** under **Monitoring**.
3. In the metrics explorer, in **Metric**, select **Storage used**.
4. Select **New alert rule**.
5. In **Scope**, confirm the registry resource for which you want to create an alert rule.
6. In **Condition**, select **Add condition**.
   a. In **Signal name**, select **Storage used**.
   b. In **Chart period**, select **Over the last 24 hours**.
   c. In **Alert logic**, in **Threshold value**, select a value such as *5*. In **Unit**, select a value such as *GB*.
   d. Accept default values for the remaining settings, and select **Done**.
7. In **Actions**, select **Add action groups** > **+ Create action group**.
   a. Enter details of the action group.
   b. On the **Notifications** tab, select **Email/SMS message/Push/Voice** and enter a recipient such as *admin@contoso.com*. Select **Review + create**.
8. Enter a name and description of the alert rule, and select the severity level.
9. Select **Create alert rule**.

Next steps

- See Monitoring Azure Container Registry data reference for a reference of the metrics, logs, and other important values created by Azure Container Registry.
- See Monitoring Azure resources with Azure Monitor for details on monitoring Azure resources.
- See Show registry usage for information about how to get a snapshot of storage usage and other resource consumption in your registry.

Audit compliance of Azure container registries using Azure Policy

8/22/2022 • 7 minutes to read • Edit Online

Azure Policy is a service in Azure that you use to create, assign, and manage policy definitions. These policy definitions 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 policy definitions for Azure Container Registry. Use these definitions to audit new and existing registries for compliance.

There is no charge for using Azure Policy.

Built-in policy definitions

The following built-in policy definitions are specific to Azure Container Registry:

NAME (AZURE PORTAL)	DESCRIPTION	EFFECT(S)	VERSION (GITHUB)
[Preview]: [Preview]: Container Registry should use a virtual network service endpoint	This policy audits any Container Registry not configured to use a virtual network service endpoint.	Audit, Disabled	1.0.0-preview
Configure container registries to disable anonymous authentication.	Disable anonymous pull for your registry so that data not accessible by unauthenticated user. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Configure container registries to disable ARM audience token authentication.	Disable Azure Active Directory ARM audience tokens for authentication to your registry. Only Azure Container Registry (ACR) audience tokens will be used for authentication. This will ensure only tokens meant for usage on the registry can be used for authentication. Disabling ARM audience tokens does not affect admin user's or scoped access tokens' authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.0
Configure container registries to disable local admin account.	Disable admin account for your registry so that it is not accessible by local admin. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.1
Configure Container registries to disable public network access	Disable public network access for your Container Registry resource so that it's not accessible over the public internet. This can reduce data leakage risks. Learn more at https://aka.ms/acr/portal/public-network and https://aka.ms/acr/private-link.	Modify, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Configure container registries to disable repository scoped access token.	Disable repository scoped access tokens for your registry so that repositories are not accessible by tokens. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.0
Configure Container registries with private endpoints	Private endpoints connect your virtual network to Azure services without a public IP address at the source or destination. By mapping private endpoints to your premium container registry resources, you can reduce data leakage risks. Learn more at: https://aka.ms/privateendpoints and https://aka.ms/acr/private-link.	DeployIfNotExists, Disabled	1.0.0
Container registries should be encrypted with a customer-managed key	Use customer-managed keys to manage the encryption at rest of the contents of your registries. By default, the data is encrypted at rest with service-managed keys, but customer-managed keys are commonly required to meet regulatory compliance standards. Customer-managed keys enable the data to be encrypted with an Azure Key Vault key created and owned by you. You have full control and responsibility for the key lifecycle, including rotation and management. Learn more at https://aka.ms/acr/CMK.	Audit, Deny, Disabled	1.1.2

NAME	DESCRIPTION	EFFECT(S)	VERSION
Container registries should have anonymous authentication disabled.	Disable anonymous pull for your registry so that data is not accessible by unauthenticated user. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.0
Container registries should have ARM audience token authentication disabled.	Disable Azure Active Directory ARM audience tokens for authentication to your registry. Only Azure Container Registry (ACR) audience tokens will be used for authentication. This will ensure only tokens meant for usage on the registry can be used for authentication. Disabling ARM audience tokens does not affect admin user's or scoped access tokens' authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.0
Container registries should have exports disabled	Disabling exports improves security by ensuring data in a registry is accessed solely via the dataplane ('docker pull'). Data cannot be moved out of the registry via 'acr import' or via 'acr transfer'. In order to disable exports, public network access must be disabled. Learn more at: https://aka.ms/acr/export-policy.	Audit, Deny, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Container registries should have local admin account disabled.	Disable admin account for your registry so that it is not accessible by local admin. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.1
Container registries should have repository scoped access token disabled.	Disable repository scoped access tokens for your registry so that repositories are not accessible by tokens. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.0
Container registries should have SKUs that support Private Links	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your container registries instead of the entire service, data leakage risks are reduced. Learn more at: https://aka.ms/acr/private-link.	Audit, Deny, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Container registries should not allow unrestricted network access	Azure container registries by default accept connections over the internet from hosts on any network. To protect your registries from potential threats, allow access from only specific private endpoints, public IP addresses or address ranges. If your registry doesn't have network rules configured, it will appear in the unhealthy resources. Learn more about Container Registry network rules here: https://aka.ms/acr/privatelink, https://aka.ms/acr/portal/public-network and https://aka.ms/acr/vnet.	Audit, Deny, Disabled	2.0.0
Container registries should use private link	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network.By mapping private endpoints to your container registries instead of the entire service, you'll also be protected against data leakage risks. Learn more at: https://aka.ms/acr/private-link.	Audit, Disabled	1.0.1
Container registry images should have vulnerability findings resolved	Container image vulnerability assessment scans your registry for security vulnerabilities and exposes detailed findings for each image. Resolving the vulnerabilities can greatly improve your containers' security posture and protect them from attacks.	AuditIfNotExists, Disabled	2.0.1

NAME	DESCRIPTION	EFFECT(S)	VERSION
Public network access should be disabled for Container registries	Disabling public network access improves security by ensuring that container registries are not exposed on the public internet. Creating private endpoints can limit exposure of container registry resources. Learn more at: https://aka.ms/acr/portal/public-network and https://aka.ms/acr/private-link.	Audit, Deny, Disabled	1.0.0

Create policy assignments

- Create policy assignments 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 create or update a policy assignment, it takes some time for the assignment to evaluate 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
--------------------------------------------------------------------------------------  ---------------------
-------------------------------------------------------------------------------------------------------------
Container Registries should not allow unrestricted network access
/subscriptions/<subscriptionID>/providers/Microsoft.Authorization/policyAssignments/b4faf132dc344b84ba68a441
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.

Scan registry images with Microsoft Defender for Cloud

8/22/2022 • 2 minutes to read • Edit Online

To scan images in your Azure container registries for vulnerabilities, you can integrate one of the available Azure Marketplace solutions or, if you want to use Microsoft Defender for Cloud, optionally enable **Microsoft Defender for container registries** at the subscription level.

- Learn more about Microsoft Defender for container registries
- Learn more about container security in Microsoft Defender for Cloud

Registry operations by Microsoft Defender for Cloud

Microsoft Defender for Cloud scans images that are pushed to a registry, imported into a registry, or any images pulled within the last 30 days. If vulnerabilities are detected, recommended remediations appear in Microsoft Defender for Cloud.

After you've taken the recommended steps to remediate the security issue, replace the image in your registry. Microsoft Defender for Cloud rescans the image to confirm that the vulnerabilities are remediated.

For details, see Use Microsoft Defender for container registries.

> **TIP**
> Microsoft Defender for Cloud authenticates with the registry to pull images for vulnerability scanning. If resource logs are collected for your registry, you'll see registry login events and image pull events generated by Microsoft Defender for Cloud. These events are associated with an alphanumeric ID such as `b21cb118-5a59-4628-bab0-3c3f0e434cg6`.

Scanning a network-restricted registry

Microsoft Defender for Cloud can scan images in a publicly accessible container registry or one that's protected with network access rules. If network rules are configured (that is, you disable public registry access, configure IP access rules, or create private endpoints), be sure to enable the network setting to **allow trusted Microsoft services** to access the registry. By default, this setting is enabled in a new container registry.

Next steps

- Learn more about registry access by trusted services.
- To restrict access to a registry using a private endpoint in a virtual network, see Configure Azure Private Link for an Azure container registry.
- To set up registry firewall rules, see Configure public IP network rules.

Scan container images using GitHub Actions

8/22/2022 • 3 minutes to read • Edit Online

Get started with the GitHub Actions by creating a workflow to build and scan a container image.

With GitHub Actions, you can speed up your CI/CD process by building, scanning, and pushing images to a public or private Container Registry from your workflows.

In this article, we'll make use of the Container image scan from the GitHub Marketplace.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- An Azure Container Registry to store all the container images that are built and pushed. You can create an Azure Container Registry from the Azure portal.
- A GitHub account with an active repository. If you don't have one, sign up for free.
  - This example uses the Java Spring PetClinic Sample Application.

Workflow file overview

A workflow is defined by a YAML (.yml) file in the `/.github/workflows/` path in your repository. This definition contains the various steps and parameters that make up the workflow.

The file has three sections:

SECTION	TASKS
Authentication	1. Create a Container Registry on Azure. 2. Create a GitHub secret. 3. Log in to the registry using GH actions.
Build	1. Set up the environment. 2. Build the app.
Build, Scan & Push the Image to the container registry	1. Build the image. 2. Scan the image. 3. Push the image.

Create GitHub secrets

To use Azure Container Registry Login action, you first need to add your Container Registry details as a secret to your GitHub repository.

In this example, you'll create a three secrets that you can use to authenticate with Azure.

1. Open your GitHub repository and go to **Settings**.

2. Select **Secrets** and then **New Secret**.

Name

AZURE_CREDENTIALS

Value

```
{
  "appId": "e6deb690-1674-445c-a9dc-zzzzzzz",
  "displayName": "DISPLAY_NAME",
  "name": "http://DISPLAY_NAME",
  "password": "zzzzzzzzzzzzzzz",
  "tenant": "zzzzzzz-zzz-zzzzzzzzzzz"
}
```

3. Paste the following values for each secret created with the following values from the Azure portal by navigating to the **Access Keys** in the Container Registry.

GITHUB SECRET NAME	AZURE CONTAINER REGISTRY VALUES
`ACR_LOGIN_SERVER`	**Login server**
`REGISTRY_USERNAME`	**Username**
`REGISTRY_PASSWORD`	**password**

4. Save by selecting **Add secret**.

Add a Dockerfile

Use the following snippet to create a `Dockerfile` and commit it to the repository.

```
FROM openjdk:11-jre-stretch
ADD target/spring-petclinic-2.4.2.jar spring-petclinic-2.4.2.jar
EXPOSE 8080
ENTRYPOINT [ "java", "-jar", "spring-petclinic-2.4.2.jar" ]
```

Use the Docker Login action

Use your secret with the Azure Container Registry Login action to authenticate to Azure.

In this workflow, you authenticate using the Azure Container Registry Login with the **login server**, **username, and **password** details of the registry stored in `secrets.ACR_LOGIN_SERVER`, `secrets.REGISTRY_USERNAME` and `secrets.REGISTRY_PASSWORD` respectively in the docker-login action. For more information about referencing GitHub secrets in a workflow file, see Using encrypted secrets in a workflow in GitHub Docs.

```
on: [push]

name: Container Scan

jobs:
  build-image:
    runs-on: ubuntu-latest
    steps:
    - name: Login to the Container Registry
      uses: azure/docker-login@v1
      with:
        login-server: ${{ secrets.ACR_LOGIN_SERVER }}
        username: ${{ secrets.REGISTRY_USERNAME }}
        password: ${{ secrets.REGISTRY_PASSWORD }}
```

Configure Java

Set up the Java environment with the Java Setup SDK action. For this example, you'll set up the environment, build with Maven, and then build, scan and push the image to the container registry.

GitHub artifacts are a way to share files in a workflow between jobs.

```
on: [push]

name: Container Scan

jobs:
  build-image:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - name: Login to the Container Registry
      uses: azure/docker-login@v1
      with:
        login-server: ${{ secrets.ACR_LOGIN_SERVER }}
        username: ${{ secrets.REGISTRY_USERNAME }}
        password: ${{ secrets.REGISTRY_PASSWORD }}

    - name: Setup Java 1.8.x
      uses: actions/setup-java@v1
      with:
        java-version: '1.8.x'

    - name: Build with Maven
      run: mvn package --file pom.xml
```

Build your image

Build and tag the image using the following snippet in the workflow. The following code snippet uses the Docker CLI to build and tag the image within a shell terminal.

```
    - name: Build and Tag image
      run: |
        docker build -f ./Dockerfile -t ${{ secrets.ACR_LOGIN_SERVER }}/spring-petclinic:${{
github.run_number }} .
```

Scan the image

Before pushing the built image into the container registry, make sure you scan and check the image for any vulnerabilities by using the Container image scan action.

Add the following code snippet into the workflow, which will scan the image for any ***critical vulnerabilities*** such that the image that will be pushed is secure and complies with the standards.

You can also add other inputs with this action or add an `allowedlist.yaml` file to your repository to ignore any vulnerabilities and best practice checks.

```
    - name: Scan image
      uses: Azure/container-scan@v0
      with:
        image-name: ${{ secrets.ACR_LOGIN_SERVER }}/spring-petclinic:${{ github.run_number }}
        severity-threshold: CRITICAL
        run-quality-checks: true
```

Example `allowedlist.yaml`.

```
general:
  vulnerabilities:
    - CVE-2003-1307
    - CVE-2011-3374
  bestPracticeViolations:
    - CIS-DI-0005
```

Push the image to a private registry

Once the image is scanned and there are no vulnerabilities found, it's safe to push the built image to private registry. The following code snippet uses the Docker CLI in a shell terminal to push the image.

```
    - name: Push image
      run: |
        docker push ${{ secrets.ACR_LOGIN_SERVER }}/spring-petclinic:${{ github.run_number }}
```

Next steps

- Learn how to Deploy to Azure Container Instances from Azure Container Registry.

Automate container image builds and maintenance with ACR Tasks

8/22/2022 • 7 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 to 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

> **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.

Personal access token

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. Required scopes for the PAT are as follows:

REPO TYPE	GITHUB	DEVOPS
Public repo	repo:status public_repo	Code (Read)
Private repo	repo (Full control)	Code (Read)

To create a PAT, see the GitHub or Azure DevOps documentation.

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 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 main branch	Files within the main (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`

CONTEXT LOCATION	DESCRIPTION	EXAMPLE
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/myap repo#mybranch:myfolder`
Remote tarball	Files in a compressed archive on a remote webserver.	`http://remoteserver/myapp.tar.gz`
Artifact in container registry	OCI artifact files in a container registry repository.	`oci://myregistry.azurecr.io/myartifact:mytag`

> **NOTE**
> When using a Git repo as a context for a task triggered by a source code update, 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 output

Each task run generates log output that you can inspect to determine whether the task steps ran successfully. When you trigger a task manually, 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 run logs in the Azure portal, or use the az acr task logs command.

See more about viewing and managing task logs.

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

8/22/2022 • 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, without a local Docker installation. ACR Tasks is a suite of features within Azure Container Registry to help you manage and modify container images across the container lifecycle. This example shows how to offload your "inner-loop" container image development cycle to the cloud with on-demand builds using a local Dockerfile.

After this quickstart, explore more advanced features of ACR Tasks using the tutorials. 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 an Azure free account before you begin.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- This quickstart requires version 2.0.58 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

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 service tiers.

Build and push image from a Dockerfile

Now use Azure Container Registry to build and push an image. First, create a local working directory and then create a Dockerfile named *Dockerfile* with the single line: `FROM mcr.microsoft.com/hello-world`. This is a simple example to build a Linux container image from the `hello-world` image hosted at Microsoft Container Registry. You can create your own standard Dockerfile and build images for other platforms. If you are working at a bash shell, create the Dockerfile with the following command:

```
echo "FROM mcr.microsoft.com/hello-world" > Dockerfile
```

Run the az acr build command, which builds the image and, after the image is successfully built, pushes it to your registry. The following example builds and 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 mcr.microsoft.com/hello-world
latest: Pulling from 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

8/22/2022 • 9 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.

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.

Prepare your environment for the Azure CLI

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

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*.

> **NOTE**
> The Dockerfile used in the following example depends on a public base container image from Docker Hub. To improve reliability when using public content, import and manage the image in a private Azure container registry, and update your Dockerfile to use your privately managed base image. Learn more about working with public images.

```
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.

```
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...
2020/11/18 18:31:42 Using acb_vol_01185991-be5f-42f0-9403-a36bb997ff35 as the home volume
2020/11/18 18:31:42 Setting up Docker configuration...
2020/11/18 18:31:43 Successfully set up Docker configuration
2020/11/18 18:31:43 Logging in to registry: myregistry.azurecr.io
2020/11/18 18:31:55 Successfully logged in
Sending build context to Docker daemon   21.5kB
Step 1/5 : FROM node:15-alpine
15-alpine: Pulling from library/node
Digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
Status: Image is up to date for node:15-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
2020/11/18 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
2020/11/18 18:32:43 Successfully pushed image: myregistry.azurecr.io/helloacrtasks:v1
2020/11/18 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: 15-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. Use Azure CLI version **2.25.0** or later for these commands:

```
# 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 list --display-name $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:

```
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.

```
Container 'acr-tasks' is in state 'Running'...
(count: 1) (last timestamp: 2020-11-18 18:39:10+00:00) pulling image
"myregistry.azurecr.io/helloacrtasks:v1"
(count: 1) (last timestamp: 2020-11-18 18:39:15+00:00) Successfully pulled image
"myregistry.azurecr.io/helloacrtasks:v1"
(count: 1) (last timestamp: 2020-11-18 18:39:17+00:00) Created container
(count: 1) (last timestamp: 2020-11-18 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

8/22/2022 • 7 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.

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)

Prepare your environment for the Azure CLI

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.

  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.

  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.

  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

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.

> **NOTE**
> The Dockerfile used in the following example depends on a public base container image from Docker Hub. To improve reliability when using public content, import and manage the image in a private Azure container registry, and update your Dockerfile to use your privately managed base image. Learn more about working with public images.

```
az acr task create \
    --registry $ACR_NAME \
    --name taskhelloworld \
    --image helloworld:{{.Run.ID}} \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git#main \
    --file Dockerfile \
    --git-access-token $GIT_PAT
```

This task specifies that any time code is committed to the *main* 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": "2010-11-19T22:42:32.972298+00:00",
  "id": "/subscriptions/<Subscription
ID>/resourceGroups/myregistry/providers/Microsoft.ContainerRegistry/registries/myregistry/tasks/taskhellowor
ld",
  "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#main",
    "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": "main",
          "repositoryUrl": "https://github.com/gituser/acr-build-helloworld-node#main",
          "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. The output is condensed to show key steps.

```
2020/11/19 22:51:00 Using acb_vol_9ee1f28c-4fd4-43c8-a651-f0ed027bbf0e as the home volume
2020/11/19 22:51:00 Setting up Docker configuration...
2020/11/19 22:51:02 Successfully set up Docker configuration
2020/11/19 22:51:02 Logging in to registry: myregistry.azurecr.io
2020/11/19 22:51:03 Successfully logged in
2020/11/19 22:51:03 Executing step: build
2020/11/19 22:51:03 Obtaining source code and scanning for dependencies...
2020/11/19 22:51:05 Successfully obtained source code and scanned for dependencies
Sending build context to Docker daemon  23.04kB
Step 1/5 : FROM node:15-alpine
[...]
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
2020/11/19 22:51:11 Executing step: push
2020/11/19 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
2020/11/19 22:51:21 Successfully pushed image: myregistry.azurecr.io/helloworld:da2
2020/11/19 22:51:21 Step id: build marked as successful (elapsed time in seconds: 7.198937)
2020/11/19 22:51:21 Populating digests for step id: build...
2020/11/19 22:51:22 Successfully populated digests for step id: build
2020/11/19 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: ca6 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 main
```

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.

```
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:

```
Showing logs of the last created run.
Run ID: ca7

[...]

Run ID: ca7 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:

```
RUN ID    TASK            PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  --------------  ----------  ---------  ---------  --------------------  ----------
ca7       taskhelloworld  linux       Succeeded  Commit     2020-11-19T22:54:34Z  00:00:29
ca6       taskhelloworld  linux       Succeeded  Manual     2020-11-19T22:51:47Z  00:00:24
ca5                       linux       Succeeded  Manual     2020-11-19T22:23:42Z  00:00:23
```

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

8/22/2022 • 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.

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)

Prepare your environment for the Azure CLI

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.

  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.

  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.

  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

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: v1.1.0
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#main \
    --file taskmulti.yaml \
    --git-access-token $GIT_PAT
```

This task specifies that any time code is committed to the *main* 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": "2020-11-20T03:14:31.763887+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#main",
    "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": "main",
          "repositoryUrl": "https://github.com/gituser/acr-build-helloworld-node.git#main",
          "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: cab
Waiting for an agent...
2020/11/20 00:03:31 Downloading source code...
2020/11/20 00:03:33 Finished downloading source code
2020/11/20 00:03:33 Using acb_vol_cfe6bd55-3076-4215-8091-6a81aec3d1b1 as the home volume
2020/11/20 00:03:33 Creating Docker network: acb_default_network, driver: 'bridge'
2020/11/20 00:03:34 Successfully set up Docker network: acb_default_network
2020/11/20 00:03:34 Setting up Docker configuration...
2020/11/20 00:03:34 Successfully set up Docker configuration
2020/11/20 00:03:34 Logging in to registry: myregistry.azurecr.io
2020/11/20 00:03:35 Successfully logged into myregistry.azurecr.io
2020/11/20 00:03:35 Executing step ID: acb_step_0. Working directory: '', Network: 'acb_default_network'
2020/11/20 00:03:35 Scanning for dependencies...
2020/11/20 00:03:36 Successfully scanned dependencies
2020/11/20 00:03:36 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
2020/11/20 00:03:43 Successfully executed container: acb_step_0
2020/11/20 00:03:43 Executing step ID: acb_step_1. Working directory: '', Network: 'acb_default_network'
2020/11/20 00:03:43 Launching container with name: acb_step_1
279b1cb6e092b64c8517c5506fcb45494cd5a0bd10a6beca3ba97f25c5d940cd
2020/11/20 00:03:44 Successfully executed container: acb_step_1
2020/11/20 00:03:44 Executing step ID: acb_step_2. Working directory: '', Network: 'acb_default_network'
2020/11/20 00:03:44 Pushing image: myregistry.azurecr.io/hello-world:cf19, attempt 1
[...]
2020/11/20 00:03:46 Successfully pushed image: myregistry.azurecr.io/hello-world:cf19
2020/11/20 00:03:46 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 7.425169)
2020/11/20 00:03:46 Populating digests for step ID: acb_step_0...
2020/11/20 00:03:47 Successfully populated digests for step ID: acb_step_0
2020/11/20 00:03:47 Step ID: acb_step_1 marked as successful (elapsed time in seconds: 0.827129)
2020/11/20 00:03:47 Step ID: acb_step_2 marked as successful (elapsed time in seconds: 2.112113)
2020/11/20 00:03:47 The following dependencies were found:
2020/11/20 00:03:47
- image:
    registry: myregistry.azurecr.io
    repository: hello-world
    tag: cf19
    digest: sha256:6b981a8ca8596e840228c974c929db05c0727d8630465de536be74104693467a
  runtime-dependency:
    registry: registry.hub.docker.com
    repository: library/node
    tag: 15-alpine
    digest: sha256:8dafc0968fb4d62834d9b826d85a8feecc69bd72cd51723c62c7db67c6dec6fa
  git:
    git-head-revision: 1a3065388a0238e52865db1c8f3e97492a43444c

Run ID: cab 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 main
```

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.

```
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:

```
Showing logs of the last created run.
Run ID: cad

[...]

Run ID: cad 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:

```
RUN ID    TASK            PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  --------------  ----------  ---------  ---------  --------------------  ----------
cad       example1        linux       Succeeded  Commit     2020-11-20T00:22:15Z  00:00:35
cac       taskhelloworld  linux       Succeeded  Commit     2020-11-20T00:22:15Z  00:00:22
cab       example1        linux       Succeeded  Manual     2020-11-20T00:18:36Z  00:00:47
```

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: v1.1.0
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#main \
    --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, so that it has permissions to push images. To create the service principal, use the following script:

```
#!/bin/bash
# This script requires Azure CLI version 2.25.0 or later. Check version with `az --version`.

# 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=$containerRegistry
SERVICE_PRINCIPAL_NAME=$servicePrincipal

# Obtain the full registry ID
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query "id" --output tsv)
# echo $registryId

# 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
PASSWORD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME --scopes $ACR_REGISTRY_ID --role acrpull
--query "password" --output tsv)
USER_NAME=$(az ad sp list --display-name $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: $USER_NAME"
echo "Service principal password: $PASSWORD"
```

Pass the service principal application ID and password in the following `az acr task credential add` command. Be sure to update the login server name *mycontainerregistrydate* with the name of your second registry:

```
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...
2020/11/20 04:33:39 Downloading source code...
2020/11/20 04:33:41 Finished downloading source code
2020/11/20 04:33:42 Using acb_vol_4569b017-29fe-42bd-83b2-25c45a8ac807 as the home volume
2020/11/20 04:33:42 Creating Docker network: acb_default_network, driver: 'bridge'
2020/11/20 04:33:43 Successfully set up Docker network: acb_default_network
2020/11/20 04:33:43 Setting up Docker configuration...
2020/11/20 04:33:44 Successfully set up Docker configuration
```

```
2020/11/20 04:33:44 Logging in to registry: mycontainerregistry.azurecr.io
2020/11/20 04:33:45 Successfully logged into mycontainerregistry.azurecr.io
2020/11/20 04:33:45 Logging in to registry: mycontainerregistrydate.azurecr.io
2020/11/20 04:33:47 Successfully logged into mycontainerregistrydate.azurecr.io
2020/11/20 04:33:47 Executing step ID: acb_step_0. Working directory: '', Network: 'acb_default_network'
2020/11/20 04:33:47 Scanning for dependencies...
2020/11/20 04:33:47 Successfully scanned dependencies
2020/11/20 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
2020/11/20 04:33:55 Successfully executed container: acb_step_0
2020/11/20 04:33:55 Executing step ID: acb_step_1. Working directory: '', Network: 'acb_default_network'
2020/11/20 04:33:55 Scanning for dependencies...
2020/11/20 04:33:56 Successfully scanned dependencies
2020/11/20 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
2020/11/20 04:33:57 Successfully executed container: acb_step_1
2020/11/20 04:33:57 Executing step ID: acb_step_2. Working directory: '', Network: 'acb_default_network'
2020/11/20 04:33:57 Launching container with name: acb_step_2
721437ff674051b6be63cbcd2fa8eb085eacbf38d7d632f1a079320133182101
2020/11/20 04:33:58 Successfully executed container: acb_step_2
2020/11/20 04:33:58 Executing step ID: acb_step_3. Working directory: '', Network: 'acb_default_network'
2020/11/20 04:33:58 Launching container with name: acb_step_3
test
2020/11/20 04:34:09 Successfully executed container: acb_step_3
2020/11/20 04:34:09 Executing step ID: acb_step_4. Working directory: '', Network: 'acb_default_network'
2020/11/20 04:34:09 Pushing image: mycontainerregistry.azurecr.io/hello-world:cf1g, attempt 1
The push refers to repository [mycontainerregistry.azurecr.io/hello-world]
[...]
2020/11/20 04:34:12 Successfully pushed image: mycontainerregistry.azurecr.io/hello-world:cf1g
2020/11/20 04:34:12 Pushing image: mycontainerregistrydate.azurecr.io/hello-world:20190503-043342z, attempt
1
The push refers to repository [mycontainerregistrydate.azurecr.io/hello-world]
[...]
2020/11/20 04:34:19 Successfully pushed image: mycontainerregistrydate.azurecr.io/hello-world:20190503-
043342z
2020/11/20 04:34:19 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 8.125744)
2020/11/20 04:34:19 Populating digests for step ID: acb_step_0...
2020/11/20 04:34:21 Successfully populated digests for step ID: acb_step_0
2020/11/20 04:34:21 Step ID: acb_step_1 marked as successful (elapsed time in seconds: 2.009281)
2020/11/20 04:34:21 Populating digests for step ID: acb_step_1...
2020/11/20 04:34:23 Successfully populated digests for step ID: acb_step_1
2020/11/20 04:34:23 Step ID: acb_step_2 marked as successful (elapsed time in seconds: 0.795440)
2020/11/20 04:34:23 Step ID: acb_step_3 marked as successful (elapsed time in seconds: 11.446775)
2020/11/20 04:34:23 Step ID: acb_step_4 marked as successful (elapsed time in seconds: 9.734973)
2020/11/20 04:34:23 The following dependencies were found:
2020/11/20 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: 15-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: 15-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

8/22/2022 • 8 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

Prerequisites

Complete the previous tutorials

This tutorial assumes you've already configured your environment and 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

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
- Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

- This article requires version 2.0.46 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

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:15-alpine --file Dockerfile-base .
```

Create a task

Next, create a task with az acr task create:

```
az acr task create \
    --registry $ACR_NAME \
    --name baseexample1 \
    --image helloworld:{{.Run.ID}} \
    --arg REGISTRY_NAME=$ACR_NAME.azurecr.io \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git#main \
    --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:15-alpine`), the Dockerfile in this task, Dockerfile-app, specifies a base image in the same registry:

```
FROM ${REGISTRY_NAME}/baseimages/node:15-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 baseexample1
```

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.

```
RUN ID    TASK            PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  --------------  ----------  ---------  ---------  --------------------  ----------
cax       baseexample1    linux       Succeeded  Manual     2020-11-20T23:33:12Z  00:00:30
caw       taskhelloworld  linux       Succeeded  Commit     2020-11-20T23:16:07Z  00:00:29
cav       example2        linux       Succeeded  Commit     2020-11-20T23:16:07Z  00:00:55
cau       example1        linux       Succeeded  Commit     2020-11-20T23:16:07Z  00:00:40
cat       taskhelloworld  linux       Succeeded  Manual     2020-11-20T23:07:29Z  00:00:27
```

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 15.2.1a
```

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:15-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.

```
Run ID    TASK            PLATFORM    STATUS     TRIGGER       STARTED               DURATION
--------  --------------  ----------  ---------  ------------  --------------------  ----------
ca11      baseexample1    linux       Succeeded  Image Update  2020-11-20T23:38:24Z  00:00:34
ca10      taskhelloworld  linux       Succeeded  Image Update  2020-11-20T23:38:24Z  00:00:24
cay                       linux       Succeeded  Manual        2020-11-20T23:38:08Z  00:00:22
cax       baseexample1    linux       Succeeded  Manual        2020-11-20T23:33:12Z  00:00:30
caw       taskhelloworld  linux       Succeeded  Commit        2020-11-20T23:16:07Z  00:00:29
cav       example2        linux       Succeeded  Commit        2020-11-20T23:16:07Z  00:00:55
cau       example1        linux       Succeeded  Commit        2020-11-20T23:16:07Z  00:00:40
cat       taskhelloworld  linux       Succeeded  Manual        2020-11-20T23:07:29Z  00:00:27
```

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 "ca11").

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.

For a complete workflow to manage base images originating from a public source, see How to consume and maintain public content with Azure Container Registry Tasks.

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

8/22/2022 • 9 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

Prerequisites

Complete the previous tutorials

This tutorial assumes you've already configured your environment and 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:15-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 $ACR_NAME \
    --name baseexample2 \
    --image helloworld:{{.Run.ID}} \
    --context https://github.com/$GIT_USER/acr-build-helloworld-node.git#main \
    --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:15-alpine`), the Dockerfile in this task, Dockerfile-app, specifies a base image in the base image registry:

```
FROM ${REGISTRY_NAME}/baseimages/node:15-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 baseexample2 --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 baseexample2 \
  --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 baseexample2
```

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
```

```
UN ID    TASK            PLATFORM    STATUS     TRIGGER       STARTED               DURATION
--------  --------------  ----------  ---------  ------------  --------------------  ----------
ca12      baseexample2    linux       Succeeded  Manual        2020-11-21T00:00:56Z  00:00:36
ca11      baseexample1    linux       Succeeded  Image Update  2020-11-20T23:38:24Z  00:00:34
ca10      taskhelloworld  linux       Succeeded  Image Update  2020-11-20T23:38:24Z  00:00:24
cay                       linux       Succeeded  Manual        2020-11-20T23:38:08Z  00:00:22
cax       baseexample1    linux       Succeeded  Manual        2020-11-20T23:33:12Z  00:00:30
caw       taskhelloworld  linux       Succeeded  Commit        2020-11-20T23:16:07Z  00:00:29
```

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 15.2.1a
```

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:15-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
```

```
         PLATFORM    STATUS     TRIGGER       STARTED               DURATION
--------  --------------  ----------  ---------  ------------  --------------------  ----------
ca13      baseexample2    linux       Succeeded  Image Update  2020-11-21T00:06:00Z  00:00:43
ca12      baseexample2    linux       Succeeded  Manual        2020-11-21T00:00:56Z  00:00:36
ca11      baseexample1    linux       Succeeded  Image Update  2020-11-20T23:38:24Z  00:00:34
ca10      taskhelloworld  linux       Succeeded  Image Update  2020-11-20T23:38:24Z  00:00:24
cay                       linux       Succeeded  Manual        2020-11-20T23:38:08Z  00:00:22
cax       baseexample1    linux       Succeeded  Manual        2020-11-20T23:33:12Z  00:00:30
caw       taskhelloworld  linux       Succeeded  Commit        2020-11-20T23:16:07Z  00:00:29
```

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 "ca13").

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: Run an ACR task on a defined schedule

8/22/2022 • 7 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.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

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 run a scheduled task.

Create a task with a timer trigger

Task command

First, populate the following shell environment variable with a value 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 the environment variable, you must manually replace each value wherever it appears in the example commands.

```
ACR_NAME=<registry-name>        # The name of your Azure container registry
```

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 task triggers running the `hello-world` image from Microsoft Container Registry every day at 21:00 UTC. The task runs without a source code context.

```
az acr task create \
  --name timertask \
  --registry $ACR_NAME \
  --cmd mcr.microsoft.com/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 timertask --registry $ACR_NAME --output table
```

```
NAME      PLATFORM    STATUS    SOURCE REPOSITORY       TRIGGERS
--------  ----------  --------  -------------------     -----------------
timertask linux       Enabled                           BASE_IMAGE, TIMER
```

Trigger the task

Trigger the task manually with az acr task run to ensure that it is set up properly:

```
az acr task run --name timertask --registry $ACR_NAME
```

If the container runs successfully, the output is similar to the following. The output is condensed to show key steps

```
Queued a run with ID: cf2a
Waiting for an agent...
2020/11/20 21:03:36 Using acb_vol_2ca23c46-a9ac-4224-b0c6-9fde44eb42d2 as the home volume
2020/11/20 21:03:36 Creating Docker network: acb_default_network, driver: 'bridge'
[...]
2020/11/20 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 timertask --registry $ACR_NAME --output table
```

When the timer is successful, output is similar to the following:

```
RUN ID    TASK       PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  ---------  ----------  ---------  ---------  --------------------  ----------
ca15      timertask  linux       Succeeded  Timer      2020-11-20T21:00:23Z  00:00:06
ca14      timertask  linux       Succeeded  Manual     2020-11-20T20:53:35Z  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 *timertask* created previously. This timer triggers the task every day at 10:30 UTC.

```
az acr task timer add \
  --name timertask \
  --registry $ACR_NAME \
  --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 timertask \
  --registry $ACR_NAME \
  --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 timertask --registry $ACR_NAME
```

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 *timertask*:

```
az acr task timer remove \
  --name timertask \
  --registry $ACR_NAME \
  --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

8/22/2022 • 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

8/22/2022 • 4 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

8/22/2022 • 4 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.

Maintain copies of base images

For any content in your registries that depends on base content maintained in a public registry such as Docker Hub, we recommend that you copy the content to an Azure container registry or another private registry. Then, ensure that you build your application images by referencing the private base images. Azure Container Registry provides an image import capability to easily copy content from public registries or other Azure container registries. The next section describes using ACR Tasks to track base image updates when building application updates. You can track base image updates in your own Azure container registries and optionally in upstream public registries.

Track base image updates

ACR Tasks includes the ability to automatically build images for you when a container's base image is updated. You can use this ability to maintain and update copies of public base images in your Azure container registries, and then to rebuild application images that depend on base images.

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.

Base image notifications

The time between when a base image is updated and when the dependent task is triggered depends on the base image location:

- **Base images from a public repo in Docker Hub or MCR** - For base images in public repositories, an ACR task checks for image updates at a random interval of between 10 and 60 minutes. Dependent tasks are run accordingly.
- **Base images from an Azure container registry** - For base images in Azure container registries, an ACR task immediately triggers a run when its base image is updated. The base image may be in the same ACR where the task runs or in a different ACR in any region.

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 --registry 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

8/22/2022 • 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#main \
    --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#main \
    --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/myUse
rAssignedIdentity"
```

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`.

The `--use-identity` parameter is not optional, if the registry has public network access disabled and relies only on certain trusted services to run ACR tasks. See, example of ACR Tasks as a trusted service.

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

8/22/2022 • 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

For demonstration purposes, as a one-time operation, run [az acr import][az-acr-import] to import a public Node.js image from Docker Hub to your base registry. In practice, another team or process in the organization might maintain images in the base registry.

```
az acr import --name mybaseregistry \
  --source docker.io/library/node:15-alpine \
  --image baseimages/node:15-alpine
```

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#main
-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/myACR
TasksId": {
        "clientId": "xxxxxxxx-f17e-4768-bb4e-xxxxxxxxxxxx",
        "principalId": "xxxxxxxx-1335-433d-bb6c-xxxxxxxxxxxx"
      }
[...]
```

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
```

Proceed to Add target registry credentials to task.

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 <task_name> --registry <registry_name> \
  --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:15-alpine
15-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: 15-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

8/22/2022 • 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/myACR
TasksId": {
        "clientId": "xxxxxxxx-f17e-4768-bb4e-xxxxxxxxxxxx",
        "principalId": "xxxxxxxx-1335-433d-bb6c-xxxxxxxxxxxx"
      }
[...]
```

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
```

Proceed to Manually run the task.

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 <task_name> --registry <registry_name> \
  --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

Run ACR Tasks using Resource Manager templates

8/22/2022 • 6 minutes to read • Edit Online

ACR Tasks is a suite of features within Azure Container Registry to help you manage and modify container images across the container lifecycle.

This article shows Azure Resource Manager template examples to queue a quick task run, similar to one you can create manually using the az acr build command.

A Resource Manager template to queue a task run is useful in automation scenarios and extends the functionality of `az acr build`. For example:

- Use a template to create a container registry and immediately queue a task run to build and push a container image
- Create or enable additional resources you can use in a quick task run, such as a managed identity for Azure resources

Limitations

- You must specify a remote context such as a GitHub repo as the source location for your task run. You can't use a local source context.
- For task runs using a managed identity, only a *user-assigned* managed identity is permitted.

Prerequisites

- **GitHub account** - Create an account on https://github.com if you don't already have one.
- **Fork sample repository** - For the task examples shown here, use the GitHub UI to fork the following sample repository into your GitHub account: https://github.com/Azure-Samples/acr-build-helloworld-node. This repo contains sample Dockerfiles and source code to build small container images.

Example: Create registry and queue task run

This example uses a sample template to create a container registry and queue a task run that builds and pushes an image.

Template parameters

For this example, provide values for the following template parameters:

PARAMETER	VALUE
registryName	Unique name of registry that's created
repository	Target repository for build task
taskRunName	Name of task run, which specifies image tag
sourceLocation	Remote context for the build task, for example, https://github.com/Azure-Samples/acr-build-helloworld-node. The Dockerfile in the repo root builds a container image for a small Node.js web app. If desired, use your fork of the repo as the build context.

Deploy the template

Deploy the template with the az deployment group create command. This example builds and pushes the *helloworld-node:testrun* image to a registry named *mycontainerregistry*.

> **NOTE**
> The Dockerfile used in the following example depends on a public base container image from Docker Hub. To improve reliability when using public content, import and manage the image in a private Azure container registry, and update your Dockerfile to use your privately managed base image. Learn more about working with public images.

```
az deployment group create \
  --resource-group myResourceGroup \
  --template-uri https://raw.githubusercontent.com/Azure/acr/master/docs/tasks/run-as-deployment/quickdockerbuild/azuredeploy.json \
  --parameters \
    registryName=mycontainerregistry \
    repository=helloworld-node \
    taskRunName=testrun \
    sourceLocation=https://github.com/Azure-Samples/acr-build-helloworld-node.git#main
```

The previous command passes the parameters on the command line. If desired, pass them in a parameters file.

Verify deployment

After the deployment completes successfully, verify the image is built by running az acr repository show-tags:

```
az acr repository show-tags \
  --name mycontainerregistry \
  --repository helloworld-node --output table
```

Output:

```
Result
--------
testrun
```

View run log

To view details about the task run, view the run log.

First, get the run ID with az acr task list-runs

```
az acr task list-runs \
  --registry mycontainerregistry --output table
```

Output is similar to:

```
RUN ID    TASK    PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  ------  ----------  ---------  ---------  --------------------  ----------
ca1               linux       Succeeded  Manual     2020-03-23T17:54:28Z  00:00:48
```

Run az acr task logs to view task run logs for the run ID, in this case *ca1*:

```
az acr task logs \
  --registry mycontainerregistry \
  --run-id ca1
```

The output shows the task run log.

You can also view the task run log in the Azure portal.

1. Navigate to your container registry
2. Under **Services**, select **Tasks** > **Runs**.
3. Select the run ID, in this case *ca1*.

The portal shows the task run log.

Example: Task run with managed identity

Use a sample template to queue a task run that enables a user-assigned managed identity. During the task run, the identity authenticates to pull an image from another Azure container registry.

This scenario is similar to Cross-registry authentication in an ACR task using an Azure-managed identity. For example, an organization might maintain a centralized registry with base images accessed by multiple development teams.

Prepare base registry

For demonstration purposes, create a separate container registry as your base registry, and push a Node.js base image pulled from Docker Hub.

1. Create a second container registry, for example *mybaseregistry*, to store base images.
2. Pull the `node:9-alpine` image from Docker Hub, tag it for your base registry, and push it to the base registry:

```
docker pull node:9-alpine
docker tag node:9-alpine mybaseregistry.azurecr.io/baseimages/node:9-alpine
az acr login -n mybaseregistry
docker push mybaseregistry.azurecr.io/baseimages/node:9-alpine
```

Create new Dockerfile

Create a Dockerfile that pulls the base image from your base registry. Perform the following steps in your local fork of the GitHub repo, for example, `https://github.com/myGitHubID/acr-build-helloworld-node.git`.

1. In the GitHub UI, select **Create new file**.
2. Name your file *Dockerfile-test* and paste the following contents. Substitute your registry name for *mybaseregistry*.

   ```
   FROM mybaseregistry.azurecr.io/baseimages/node:9-alpine
   COPY . /src
   RUN cd /src && npm install
   EXPOSE 80
   CMD ["node", "/src/server.js"]
   ```

3. Select **Commit new file**.

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)
```

Give identity pull permissions to the base registry

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
```

Template parameters

For this example, provide values for the following template parameters:

PARAMETER	VALUE
registryName	Name of registry where image is built
repository	Target repository for build task
taskRunName	Name of task run, which specifies image tag
userAssignedIdentity	Resource ID of user-assigned identity enabled in the task

PARAMETER	VALUE
customRegistryIdentity	Client ID of user-assigned identity enabled in the task, used to authenticate with custom registry
customRegistry	Login server name of the custom registry accessed in the task, for example, *mybaseregistry.azurecr.io*
sourceLocation	Remote context for the build task, for example, *https://github.com/\<your-GitHub-ID>/acr-build-helloworld-node*.
dockerFilePath	Path to the Dockerfile at the remote context, used to build the image.

Deploy the template

Deploy the template with the az deployment group create command. This example builds and pushes the *helloworld-node:testrun* image to a registry named *mycontainerregistry*. The base image is pulled from *mybaseregistry.azurecr.io*.

```
az deployment group create \
  --resource-group myResourceGroup \
  --template-uri https://raw.githubusercontent.com/Azure/acr/master/docs/tasks/run-as-deployment/quickdockerbuildwithidentity/azuredeploy.json \
  --parameters \
    registryName=mycontainerregistry \
    repository=helloworld-node \
    taskRunName=basetask \
    userAssignedIdentity=$resourceID \
    customRegistryIdentity=$clientID \
    sourceLocation=https://github.com/<your-GitHub-ID>/acr-build-helloworld-node.git#main \
    dockerFilePath=Dockerfile-test \
    customRegistry=mybaseregistry.azurecr.io
```

The previous command passes the parameters on the command line. If desired, pass them in a parameters file.

Verify deployment

After the deployment completes successfully, verify the image is built by running az acr repository show-tags:

```
az acr repository show-tags \
  --name mycontainerregistry \
  --repository helloworld-node --output table
```

Output:

```
Result
--------
basetask
```

View run log

To view the run log, see steps in the preceding section.

Next steps

- See more template examples in the ACR GitHub repo.

- For details about template properties, see the template reference for Task runs and Tasks.

Run an ACR task on a dedicated agent pool

8/22/2022 • 4 minutes to read • Edit Online

Set up an Azure-managed VM pool (*agent pool*) to enable running your Azure Container Registry tasks in a dedicated compute environment. After you've configured one or more pools in your registry, you can choose a pool to run a task in place of the service's default compute environment.

An agent pool provides:

- **Virtual network support** - Assign an agent pool to an Azure VNet, providing access to resources in the VNet such as a container registry, key vault, or storage.
- **Scale as needed** - Increase the number of instances in an agent pool for compute-intensive tasks, or scale to zero. Billing is based on pool allocation. For details, see Pricing.
- **Flexible options** - Choose from different pool tiers and scale options to meet your task workload needs.
- **Azure management** - Task pools are patched and maintained by Azure, providing reserved allocation without the need to maintain the individual VMs.

This feature is available in the **Premium** container registry service tier. For information about registry service tiers and limits, see Azure Container Registry SKUs.

> **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

- Task agent pools currently support Linux nodes. Windows nodes aren't currently supported.
- Task agent pools are available in preview in the following regions: West US 2, South Central US, East US 2, East US, Central US, West Europe, North Europe, Canada Central, East Asia, USGov Arizona, USGov Texas, and USGov Virginia.
- For each registry, the default total vCPU (core) quota is 16 for all standard agent pools and is 0 for isolated agent pools. Open a support request for additional allocation.
- You can't currently cancel a task run on an agent pool.

Prerequisites

- To use the Azure CLI steps in this article, Azure CLI version 2.3.1 or later is required. If you need to install or upgrade, see Install Azure CLI. Or run in Azure Cloud Shell.
- If you don't already have a container registry, create one (Premium tier required) in a preview region.

Pool tiers

Agent pool tiers provide the following resources per instance in the pool.

TIER	TYPE	CPU	MEMORY (GB)
S1	standard	2	3

TIER	TYPE	CPU	MEMORY (GB)
S2	standard	4	8
S3	standard	8	16
I6	isolated	64	216

Create and manage a task agent pool

Set default registry (optional)

To simplify Azure CLI commands that follow, set the default registry by running the az config command:

```
az config set defaults.acr=<registryName>
```

The following examples assume that you've set the default registry. If not, pass a `--registry <registryName>` parameter in each `az acr` command.

Create agent pool

Create an agent pool by using the az acr agentpool create command. The following example creates a tier S2 pool (4 CPU/instance). By default, the pool contains 1 instance.

```
az acr agentpool create \
    --registry MyRegistry \
    --name myagentpool \
    --tier S2
```

> **NOTE**
> Creating an agent pool and other pool management operations take several minutes to complete.

Scale pool

Scale the pool size up or down with the az acr agentpool update command. The following example scales the pool to 2 instances. You can scale to 0 instances.

```
az acr agentpool update \
    --registry MyRegistry \
    --name myagentpool \
    --count 2
```

Create pool in a virtual network

Add firewall rules

Task agent pools require access to the following Azure services. The following firewall rules must be added to any existing network security groups or user-defined routes.

DIRECTION	PROTOCOL	SOURCE	SOURCE PORT	DESTINATION	DEST PORT	USED

DIRECTION	PROTOCOL	SOURCE	SOURCE PORT	DESTINATION	DEST PORT	USED
Outbound	TCP	VirtualNetwork	Any	AzureKeyVault	443	Default
Outbound	TCP	VirtualNetwork	Any	Storage	443	Default
Outbound	TCP	VirtualNetwork	Any	EventHub	443	Default
Outbound	TCP	VirtualNetwork	Any	AzureActiveDirectory	443	Default
Outbound	TCP	VirtualNetwork	Any	AzureMonitor	443	Default

> **NOTE**
> If your tasks require additional resources from the public internet, add the corresponding rules. For example, additional rules are needed to run a docker build task that pulls the base images from Docker Hub, or restores a NuGet package.

Create pool in VNet

The following example creates an agent pool in the *mysubnet* subnet of network *myvnet*:

```
# Get the subnet ID
subnetId=$(az network vnet subnet show \
        --resource-group myresourcegroup \
        --vnet-name myvnet \
        --name mysubnetname \
        --query id --output tsv)

az acr agentpool create \
    --registry MyRegistry \
    --name myagentpool \
    --tier S2 \
    --subnet-id $subnetId
```

Run task on agent pool

The following examples show how to specify an agent pool when queuing a task.

> **NOTE**
> To use an agent pool in an ACR task, ensure that the pool contains at least 1 instance.

Quick task

Queue a quick task on the agent pool by using the az acr build command and pass the `--agent-pool` parameter:

```
az acr build \
    --registry MyRegistry \
    --agent-pool myagentpool \
    --image myimage:mytag \
    --file Dockerfile \
    https://github.com/Azure-Samples/acr-build-helloworld-node.git#main
```

Automatically triggered task

For example, create a scheduled task on the agent pool with az acr task create, passing the `--agent-pool` parameter.

```
az acr task create \
    --registry MyRegistry \
    --name mytask \
    --agent-pool myagentpool \
    --image myimage:mytag \
    --schedule "0 21 * * *" \
    --file Dockerfile \
    --context https://github.com/Azure-Samples/acr-build-helloworld-node.git#main \
    --commit-trigger-enabled false
```

To verify task setup, run az acr task run:

```
az acr task run \
    --registry MyRegistry \
    --name mytask
```

Query pool status

To find the number of runs currently scheduled on the agent pool, run az acr agentpool show.

```
az acr agentpool show \
    --registry MyRegistry \
    --name myagentpool \
    --queue-count
```

Next steps

For more examples of container image builds and maintenance in the cloud, check out the ACR Tasks tutorial series.

How to consume and maintain public content with Azure Container Registry Tasks

8/22/2022 • 14 minutes to read • Edit Online

This article provides a sample workflow in Azure Container Registry to help you manage consuming and maintaining public content:

1. Import local copies of dependent public images.
2. Validate public images through security scanning and functional testing.
3. Promote the images to private registries for internal usage.
4. Trigger base image updates for applications dependent upon public content.
5. Use Azure Container Registry Tasks to automate this workflow.

The workflow is summarized in the following image:

The gated import workflow helps manage your organization's dependencies on externally managed artifacts - for example, images sourced from public registries including Docker Hub, GCR, Quay, GitHub Container Registry, Microsoft Container Registry, or even other Azure container registries.

For background about the risks introduced by dependencies on public content and how to use Azure Container Registry to mitigate them, see the OCI Consuming Public Content Blog post and Manage public content with Azure Container Registry.

You can use the Azure Cloud Shell or a local installation of the Azure CLI to complete this walkthrough. Azure CLI version 2.10 or later is recommended. If you need to install or upgrade, see Install Azure CLI.

Scenario overview

This walkthrough sets up:

1. Three **container registries**, representing:
   - A simulated Docker Hub (`publicregistry`) to support changing the base image
   - Team registry (`contoso`) to share private images
   - Company/team shared registry (`baseartifacts`) for imported public content
2. An **ACR task** in each registry. The tasks:
   a. Build a simulated public `node` image
   b. Import and validate the `node` image to the company/team shared registry
   c. Build and deploy the `hello-world` image
3. **ACR task definitions**, including configurations for:
4. A collection of **registry credentials**, which are pointers to a key vault
5. A collection of **secrets**, available within an `acr-task.yaml` , which are pointers to a key vault
6. A collection of **configured values** used within an `acr-task.yaml`
7. An **Azure key vault** to secure all secrets
8. An **Azure container instance**, which hosts the `hello-world` build application

Prerequisites

The following steps configure values for resources created and used in the walkthrough.

Set environment variables

Configure variables unique to your environment. We follow best practices for placing resources with durable content in their own resource group to minimize accidental deletion. However, you can place these in a single resource group if desired.

The examples in this article are formatted for the bash shell.

```
# Set the three registry names, must be globally unique:
REGISTRY_PUBLIC=publicregistry
REGISTRY_BASE_ARTIFACTS=contosobaseartifacts
REGISTRY=contoso

# set the location all resources will be created in:
RESOURCE_GROUP_LOCATION=eastus

# default resource groups
REGISTRY_PUBLIC_RG=${REGISTRY_PUBLIC}-rg
REGISTRY_BASE_ARTIFACTS_RG=${REGISTRY_BASE_ARTIFACTS}-rg
REGISTRY_RG=${REGISTRY}-rg

# fully qualified registry urls
REGISTRY_DOCKERHUB_URL=docker.io
REGISTRY_PUBLIC_URL=${REGISTRY_PUBLIC}.azurecr.io
REGISTRY_BASE_ARTIFACTS_URL=${REGISTRY_BASE_ARTIFACTS}.azurecr.io
REGISTRY_URL=${REGISTRY}.azurecr.io

# Azure key vault for storing secrets, name must be globally unique
AKV=acr-task-credentials
AKV_RG=${AKV}-rg

# ACI for hosting the deployed application
ACI=hello-world-aci
ACI_RG=${ACI}-rg
```

Git repositories and tokens

To simulate your environment, fork each of the following Git repos into repositories you can manage.

- https://github.com/importing-public-content/base-image-node.git
- https://github.com/importing-public-content/import-baseimage-node.git
- https://github.com/importing-public-content/hello-world.git

Then, update the following variables for your forked repositories.

The `:main` appended to the end of the git URLs represents the default repository branch.

```
GIT_BASE_IMAGE_NODE=https://github.com/<your-fork>/base-image-node.git#main
GIT_NODE_IMPORT=https://github.com/<your-fork>/import-baseimage-node.git#main
GIT_HELLO_WORLD=https://github.com/<your-fork>/hello-world.git#main
```

You need a GitHub access token (PAT) for ACR Tasks to clone and establish Git webhooks. For steps to create a token with the required permissions to a private repo, see Create a GitHub access token.

```
GIT_TOKEN=<set-git-token-here>
```

Docker Hub credentials

To avoid throttling and identity requests when pulling images from Docker Hub, create a Docker Hub token. Then, set the following environment variables:

```
REGISTRY_DOCKERHUB_USER=<yourusername>
REGISTRY_DOCKERHUB_PASSWORD=<yourtoken>
```

Create registries

Using Azure CLI commands, create three Premium tier container registries, each in its own resource group:

```
az group create --name $REGISTRY_PUBLIC_RG --location $RESOURCE_GROUP_LOCATION
az acr create --resource-group $REGISTRY_PUBLIC_RG --name $REGISTRY_PUBLIC --sku Premium

az group create --name $REGISTRY_BASE_ARTIFACTS_RG --location $RESOURCE_GROUP_LOCATION
az acr create --resource-group $REGISTRY_BASE_ARTIFACTS_RG --name $REGISTRY_BASE_ARTIFACTS --sku Premium

az group create --name $REGISTRY_RG --location $RESOURCE_GROUP_LOCATION
az acr create --resource-group $REGISTRY_RG --name $REGISTRY --sku Premium
```

Create key vault and set secrets

Create a key vault:

```
az group create --name $AKV_RG --location $RESOURCE_GROUP_LOCATION
az keyvault create --resource-group $AKV_RG --name $AKV
```

Set Docker Hub username and token in the key vault:

```
az keyvault secret set \
--vault-name $AKV \
--name registry-dockerhub-user \
--value $REGISTRY_DOCKERHUB_USER

az keyvault secret set \
--vault-name $AKV \
--name registry-dockerhub-password \
--value $REGISTRY_DOCKERHUB_PASSWORD
```

Set and verify a Git PAT in the key vault:

```
az keyvault secret set --vault-name $AKV --name github-token --value $GIT_TOKEN

az keyvault secret show --vault-name $AKV --name github-token --query value -o tsv
```

Create resource group for an Azure container instance

This resource group is used in a later task when deploying the `hello-world` image.

```
az group create --name $ACI_RG --location $RESOURCE_GROUP_LOCATION
```

Create public `node` base image

To simulate the `node` image on Docker Hub, create an ACR task to build and maintain the public image. This setup allows simulating changes by the `node` image maintainers.

```
az acr task create \
  --name node-public \
  -r $REGISTRY_PUBLIC \
  -f acr-task.yaml \
  --context $GIT_BASE_IMAGE_NODE \
  --git-access-token $(az keyvault secret show \
                       --vault-name $AKV \
                       --name github-token \
                       --query value -o tsv) \
  --set REGISTRY_FROM_URL=${REGISTRY_DOCKERHUB_URL}/ \
  --assign-identity
```

To avoid Docker throttling, add Docker Hub credentials to the task. The acr task credentials command may be used to pass Docker credentials to any registry, including Docker Hub.

```
az acr task credential add \
  -n node-public \
  -r $REGISTRY_PUBLIC \
  --login-server $REGISTRY_DOCKERHUB_URL \
  -u https://${AKV}.vault.azure.net/secrets/registry-dockerhub-user \
  -p https://${AKV}.vault.azure.net/secrets/registry-dockerhub-password \
  --use-identity [system]
```

Grant the task access to read values from the key vault:

```
az keyvault set-policy \
  --name $AKV \
  --resource-group $AKV_RG \
  --object-id $(az acr task show \
                 --name node-public \
                 --registry $REGISTRY_PUBLIC \
                 --query identity.principalId --output tsv) \
  --secret-permissions get
```

Tasks can be triggered by Git commits, base image updates, timers, or manual runs.

Run the task manually to generate the `node` image:

```
az acr task run -r $REGISTRY_PUBLIC -n node-public
```

List the image in the simulated public registry:

```
az acr repository show-tags -n $REGISTRY_PUBLIC --repository node
```

Create the `hello-world` image

Based on the simulated public `node` image, build a `hello-world` image.

Create token for pull access to simulated public registry

Create an access token to the simulated public registry, scoped to `pull`. Then, set it in the key vault:

```
az keyvault secret set \
  --vault-name $AKV \
  --name "registry-${REGISTRY_PUBLIC}-user" \
  --value "registry-${REGISTRY_PUBLIC}-user"

az keyvault secret set \
  --vault-name $AKV \
  --name "registry-${REGISTRY_PUBLIC}-password" \
  --value $(az acr token create \
             --name "registry-${REGISTRY_PUBLIC}-user" \
             --registry $REGISTRY_PUBLIC \
             --scope-map _repositories_pull \
             -o tsv \
             --query credentials.passwords[0].value)
```

Create token for pull access by Azure Container Instances

Create an access token to the registry hosting the `hello-world` image, scoped to pull. Then, set it in the key

vault:

```
az keyvault secret set \
  --vault-name $AKV \
  --name "registry-${REGISTRY}-user" \
  --value "registry-${REGISTRY}-user"

az keyvault secret set \
  --vault-name $AKV \
  --name "registry-${REGISTRY}-password" \
  --value $(az acr token create \
              --name "registry-${REGISTRY}-user" \
              --registry $REGISTRY \
              --repository hello-world content/read \
              -o tsv \
              --query credentials.passwords[0].value)
```

Create task to build and maintain `hello-world` image

The following command creates a task from the definition in `acr-tasks.yaml` in the `hello-world` repo. The task steps build the `hello-world` image and then deploy it to Azure Container Instances. The resource group for Azure Container Instances was created in a previous section. By calling `az container create` in the task with only a difference in the `image:tag`, the task deploys to same instance throughout this walkthrough.

```
az acr task create \
  -n hello-world \
  -r $REGISTRY \
  -f acr-task.yaml \
  --context $GIT_HELLO_WORLD \
  --git-access-token $(az keyvault secret show \
                       --vault-name $AKV \
                       --name github-token \
                       --query value -o tsv) \
  --set REGISTRY_FROM_URL=${REGISTRY_PUBLIC_URL}/ \
  --set KEYVAULT=$AKV \
  --set ACI=$ACI \
  --set ACI_RG=$ACI_RG \
  --assign-identity
```

Add credentials to the task for the simulated public registry:

```
az acr task credential add \
  -n hello-world \
  -r $REGISTRY \
  --login-server $REGISTRY_PUBLIC_URL \
  -u https://${AKV}.vault.azure.net/secrets/registry-${REGISTRY_PUBLIC}-user \
  -p https://${AKV}.vault.azure.net/secrets/registry-${REGISTRY_PUBLIC}-password \
  --use-identity [system]
```

Grant the task access to read values from the key vault:

```
az keyvault set-policy \
  --name $AKV \
  --resource-group $AKV_RG \
  --object-id $(az acr task show \
                 --name hello-world \
                 --registry $REGISTRY \
                 --query identity.principalId --output tsv) \
  --secret-permissions get
```

Grant the task access to create and manage Azure Container Instances by granting access to the resource group:

```
az role assignment create \
  --assignee $(az acr task show \
  --name hello-world \
  --registry $REGISTRY \
  --query identity.principalId --output tsv) \
  --scope $(az group show -n $ACI_RG --query id -o tsv) \
  --role owner
```

With the task created and configured, run the task to build and deploy the `hello-world` image:

```
az acr task run -r $REGISTRY -n hello-world
```

Once created, get the IP address of the container hosting the `hello-world` image.

```
az container show \
  --resource-group $ACI_RG \
  --name ${ACI} \
  --query ipAddress.ip \
  --out tsv
```

In your browser, go to the IP address to see the running application.

Update the base image with a "questionable" change

This section simulates a change to the base image that could cause problems in the environment.

1. Open `Dockerfile` in the forked `base-image-node` repo.
2. Change the `BACKGROUND_COLOR` to `Orange` to simulate the change.

```
ARG REGISTRY_NAME=
FROM ${REGISTRY_NAME}node:15-alpine
ENV NODE_VERSION 15-alpine
ENV BACKGROUND_COLOR Orange
```

Commit the change and watch for ACR Tasks to automatically start building.

Watch for the task to start executing:

```
watch -n1 az acr task list-runs -r $REGISTRY_PUBLIC -o table
```

You should eventually see STATUS `Succeeded` based on a TRIGGER of `Commit`:

```
RUN ID    TASK      PLATFORM    STATUS     TRIGGER    STARTED               DURATION
--------  --------  ----------  ---------  ---------  --------------------  ----------
ca4       hub-node  linux       Succeeded  Commit     2020-10-24T05:02:29Z  00:00:22
```

Type **Ctrl+C** to exit the watch command, then view the logs for the most recent run:

```
az acr task logs -r $REGISTRY_PUBLIC
```

Once the `node` image is completed, `watch` for ACR Tasks to automatically start building the `hello-world`

image:

```
watch -n1 az acr task list-runs -r $REGISTRY -o table
```

You should eventually see STATUS `Succeeded` based on a TRIGGER of `Image Update`:

```
RUN ID    TASK         PLATFORM    STATUS     TRIGGER       STARTED               DURATION
--------  -----------  ----------  ---------  ------------  --------------------  ----------
dau       hello-world  linux       Succeeded  Image Update  2020-10-24T05:08:45Z  00:00:31
```

Type **Ctrl+C** to exit the watch command, then view the logs for the most recent run:

```
az acr task logs -r $REGISTRY
```

Once completed, get the IP address of the site hosting the updated `hello-world` image:

```
az container show \
  --resource-group $ACI_RG \
  --name ${ACI} \
  --query ipAddress.ip \
  --out tsv
```

In your browser, go to the site, which should have an orange (questionable) background.

Checking in

At this point, you've created a `hello-world` image that is automatically built on Git commits and changes to the base `node` image. In this example, the task builds against a base image in Azure Container Registry, but any supported registry could be used.

The base image update automatically retriggers the task run when the `node` image is updated. As seen here, not all updates are wanted.

Gated imports of public content

To prevent upstream changes from breaking critical workloads, security scanning and functional tests may be added.

In this section, you create an ACR task to:

- Build a test image
- Run a functional test script `./test.sh` against the test image
- If the image tests successfully, import the public image to the **baseimages** registry

Add automation testing

To gate any upstream content, automated testing is implemented. In this example, a `test.sh` is provided which checks the `$BACKGROUND_COLOR`. If the test fails, an `EXIT_CODE` of `1` is returned which causes the ACR task step to fail, ending the task run. The tests can be expanded in any form of tools, including logging results. The gate is managed by a pass/fail response in the script, reproduced here:

```
if [ ""$(echo $BACKGROUND_COLOR | tr '[:lower:]' '[:upper:]') = 'RED' ]; then
    echo -e "\e[31mERROR: Invalid Color:\e[0m" ${BACKGROUND_COLOR}
    EXIT_CODE=1
else
  echo -e "\e[32mValidation Complete - No Known Errors\e[0m"
fi
exit ${EXIT_CODE}
```

Task YAML

Review the `acr-task.yaml` in the `import-baseimage-node` repo, which performs the following steps:

1. Build the test base image using the following Dockerfile:

```
ARG REGISTRY_FROM_URL=
FROM ${REGISTRY_FROM_URL}node:15-alpine
WORKDIR /test
COPY ./test.sh .
CMD ./test.sh
```

2. When completed, validate the image by running the container, which runs `./test.sh`
3. Only if successfully completed, run the import steps, which are gated with `when: ['validate-base-image']`

```
version: v1.1.0
steps:
  - id: build-test-base-image
    # Build off the base image we'll track
    # Add a test script to do unit test validations
    # Note: the test validation image isn't saved to the registry
    # but the task logs captures log validation results
    build: >
      --build-arg REGISTRY_FROM_URL={{.Values.REGISTRY_FROM_URL}}
      -f ./Dockerfile
      -t {{.Run.Registry}}/node-import:test
      .
  - id: validate-base-image
    # only continues if node-import:test returns a non-zero code
    when: ['build-test-base-image']
    cmd: "{{.Run.Registry}}/node-import:test"
  - id: pull-base-image
    # import the public image to base-artifacts
    # Override the stable tag,
    # and create a unique tag to enable rollback
    # to a previously working image
    when: ['validate-base-image']
    cmd: >
        docker pull {{.Values.REGISTRY_FROM_URL}}node:15-alpine
  - id: retag-base-image
    when: ['pull-base-image']
    cmd: docker tag {{.Values.REGISTRY_FROM_URL}}node:15-alpine {{.Run.Registry}}/node:15-alpine
  - id: retag-base-image-unique-tag
    when: ['pull-base-image']
    cmd: docker tag {{.Values.REGISTRY_FROM_URL}}node:15-alpine {{.Run.Registry}}/node:15-alpine-{{.Run.ID}}
  - id: push-base-image
    when: ['retag-base-image', 'retag-base-image-unique-tag']
    push:
    - "{{.Run.Registry}}/node:15-alpine"
    - "{{.Run.Registry}}/node:15-alpine-{{.Run.ID}}"
```

Create task to import and test base image

```
az acr task create \
--name base-import-node \
-f acr-task.yaml \
-r $REGISTRY_BASE_ARTIFACTS \
--context $GIT_NODE_IMPORT \
--git-access-token $(az keyvault secret show \
                      --vault-name $AKV \
                      --name github-token \
                      --query value -o tsv) \
--set REGISTRY_FROM_URL=${REGISTRY_PUBLIC_URL}/ \
--assign-identity
```

Add credentials to the task for the simulated public registry:

```
az acr task credential add \
  -n base-import-node \
  -r $REGISTRY_BASE_ARTIFACTS \
  --login-server $REGISTRY_PUBLIC_URL \
  -u https://${AKV}.vault.azure.net/secrets/registry-${REGISTRY_PUBLIC}-user \
  -p https://${AKV}.vault.azure.net/secrets/registry-${REGISTRY_PUBLIC}-password \
  --use-identity [system]
```

Grant the task access to read values from the key vault:

```
az keyvault set-policy \
  --name $AKV \
  --resource-group $AKV_RG \
  --object-id $(az acr task show \
                 --name base-import-node \
                 --registry $REGISTRY_BASE_ARTIFACTS \
                 --query identity.principalId --output tsv) \
  --secret-permissions get
```

Run the import task:

```
az acr task run -n base-import-node -r $REGISTRY_BASE_ARTIFACTS
```

NOTE

If the task fails due to `./test.sh: Permission denied`, ensure that the script has execution permissions, and commit back to the Git repo:

```
chmod +x ./test.sh
```

Update `hello-world` image to build from gated `node` image

Create an access token to access the base-artifacts registry, scoped to `read` from the `node` repository. Then, set in the key vault:

```
az keyvault secret set \
  --vault-name $AKV \
  --name "registry-${REGISTRY_BASE_ARTIFACTS}-user" \
  --value "registry-${REGISTRY_BASE_ARTIFACTS}-user"

az keyvault secret set \
  --vault-name $AKV \
  --name "registry-${REGISTRY_BASE_ARTIFACTS}-password" \
  --value $(az acr token create \
              --name "registry-${REGISTRY_BASE_ARTIFACTS}-user" \
              --registry $REGISTRY_BASE_ARTIFACTS \
              --repository node content/read \
              -o tsv \
              --query credentials.passwords[0].value)
```

Add credentials to the **hello-world** task for the base artifacts registry:

```
az acr task credential add \
  -n hello-world \
  -r $REGISTRY \
  --login-server $REGISTRY_BASE_ARTIFACTS_URL \
  -u https://${AKV}.vault.azure.net/secrets/registry-${REGISTRY_BASE_ARTIFACTS}-user \
  -p https://${AKV}.vault.azure.net/secrets/registry-${REGISTRY_BASE_ARTIFACTS}-password \
  --use-identity [system]
```

Update the task to change the `REGISTRY_FROM_URL` to use the `BASE_ARTIFACTS` registry

```
az acr task update \
  -n hello-world \
  -r $REGISTRY \
  --set KEYVAULT=$AKV \
  --set REGISTRY_FROM_URL=${REGISTRY_BASE_ARTIFACTS_URL}/ \
  --set ACI=$ACI \
  --set ACI_RG=$ACI_RG
```

Run the **hello-world** task to change its base image dependency:

```
az acr task run -r $REGISTRY -n hello-world
```

Update the base image with a "valid" change

1. Open the `Dockerfile` in `base-image-node` repo.
2. Change the `BACKGROUND_COLOR` to `Green` to simulate a valid change.

```
ARG REGISTRY_NAME=
FROM ${REGISTRY_NAME}node:15-alpine
ENV NODE_VERSION 15-alpine
ENV BACKGROUND_COLOR Green
```

Commit the change and monitor the sequence of updates:

```
watch -n1 az acr task list-runs -r $REGISTRY_PUBLIC -o table
```

Once running, type **Ctrl+C** and monitor the logs:

```
az acr task logs -r $REGISTRY_PUBLIC
```

Once complete, monitor the **base-image-import** task:

```
watch -n1 az acr task list-runs -r $REGISTRY_BASE_ARTIFACTS -o table
```

Once running, type **Ctrl+C** and monitor the logs:

```
az acr task logs -r $REGISTRY_BASE_ARTIFACTS
```

Once complete, monitor the **hello-world** task:

```
watch -n1 az acr task list-runs -r $REGISTRY -o table
```

Once running, type **Ctrl+C** and monitor the logs:

```
az acr task logs -r $REGISTRY
```

Once completed, get the IP address of the site hosting the updated `hello-world` image:

```
az container show \
  --resource-group $ACI_RG \
  --name ${ACI} \
  --query ipAddress.ip \
  --out tsv
```

In your browser, go to the site, which should have a green (valid) background.

View the gated workflow

Perform the steps in the preceding section again, with a background color of red.

1. Open the `Dockerfile` in the `base-image-node` repo
2. Change the `BACKGROUND_COLOR` to `Red` to simulate an invalid change.

```
ARG REGISTRY_NAME=
FROM ${REGISTRY_NAME}node:15-alpine
ENV NODE_VERSION 15-alpine
ENV BACKGROUND_COLOR Red
```

Commit the change and monitor the sequence of updates:

```
watch -n1 az acr task list-runs -r $REGISTRY_PUBLIC -o table
```

Once running, type **Ctrl+C** and monitor the logs:

```
az acr task logs -r $REGISTRY_PUBLIC
```

Once complete, monitor the **base-image-import** task:

```
watch -n1 az acr task list-runs -r $REGISTRY_BASE_ARTIFACTS -o table
```

Once running, type **Ctrl+C** and monitor the logs:

```
az acr task logs -r $REGISTRY_BASE_ARTIFACTS
```

At this point, you should see the **base-import-node** task fail validation and stop the sequence to publish a `hello-world` update. Output is similar to:

```
[...]
2020/10/30 03:57:39 Launching container with name: validate-base-image
Validating Image
NODE_VERSION: 15-alpine
BACKGROUND_COLOR: Red
ERROR: Invalid Color: Red
2020/10/30 03:57:40 Container failed during run: validate-base-image. No retries remaining.
failed to run step ID: validate-base-image: exit status 1
```

Publish an update to `hello-world`

Changes to the `hello-world` image will continue using the last validated `node` image.

Any additional changes to the base `node` image that pass the gated validations will trigger base image updates to the `hello-world` image.

Cleaning up

When no longer needed, delete the resources used in this article.

```
az group delete -n $REGISTRY_RG --no-wait -y
az group delete -n $REGISTRY_PUBLIC_RG --no-wait -y
az group delete -n $REGISTRY_BASE_ARTIFACTS_RG --no-wait -y
az group delete -n $AKV_RG --no-wait -y
az group delete -n $ACI_RG --no-wait -y
```

Next steps

In this article. you used ACR tasks to create an automated gating workflow to introduce updated base images to your environment. See related information to manage images in Azure Container Registry.

- Recommendations for tagging and versioning container images
- Lock a container image in an Azure container registry

Build and push an image from an app using a Cloud Native Buildpack

8/22/2022 • 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. If not cached by Azure Container Registry, the builder image must be pulled using the `--pull` parameter.

`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:cflinuxfs3` builder.

```
az acr pack build \
    --registry myregistry \
    --image node-app:1.0 \
    --pull --builder cloudfoundry/cnb: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.

```
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.

View and manage task run logs

8/22/2022 • 2 minutes to read • Edit Online

Each task run in Azure Container Registry tasks generates log output that you can inspect to determine whether the task steps ran successfully.

This article explains how to view and manage task run logs.

View streamed logs

When you trigger a task manually, log output is streamed directly to the console. For example, when you trigger a task manually by using the az acr build, az acr run, or az acr task run command, you see log output streamed to the console.

The following sample az acr run command manually triggers a task that runs a container pulled from the same registry:

```
az acr run --registry mycontainerregistry1220 \
  --cmd '$Registry/samples/hello-world:v1' /dev/null
```

Streamed log:

```
Queued a run with ID: cf4
Waiting for an agent...
2020/03/09 20:30:10 Alias support enabled for version >= 1.1.0, please see https://aka.ms/acr/tasks/task-
aliases for more information.
2020/03/09 20:30:10 Creating Docker network: acb_default_network, driver: 'bridge'
2020/03/09 20:30:10 Successfully set up Docker network: acb_default_network
2020/03/09 20:30:10 Setting up Docker configuration...
2020/03/09 20:30:11 Successfully set up Docker configuration
2020/03/09 20:30:11 Logging in to registry: mycontainerregistry1220azurecr.io
2020/03/09 20:30:12 Successfully logged into mycontainerregistry1220azurecr.io
2020/03/09 20:30:12 Executing step ID: acb_step_0. Timeout(sec): 600, Working directory: '', Network:
'acb_default_network'
2020/03/09 20:30:12 Launching container with name: acb_step_0
Unable to find image 'mycontainerregistry1220azurecr.io/samples/hello-world:v1' locally
v1: Pulling from samples/hello-world
Digest: sha256:92c7f9c92844bbbb5d0a101b22f7c2a7949e40f8ea90c8b3bc396879d95e888a
Status: Downloaded newer image for mycontainerregistry1220azurecr.io/samples/hello-world:v1

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

2020/03/09 20:30:13 Successfully executed container: acb_step_0
2020/03/09 20:30:13 Step ID: acb_step_0 marked as successful (elapsed time in seconds: 1.180081)

Run ID: cf4 was successful after 5s
```

View stored logs

Azure Container Registry stores run logs for all tasks. You can view stored run logs in the Azure portal. Or, use the az acr task logs command to view a selected log. By default, logs are retained for 30 days.

If a task is automatically triggered, for example by a source code update, accessing the stored logs is the *only*

way to view the run logs. Automatic task triggers include source code commits or pull requests, base image updates, and timer triggers.

To view run logs in the portal:

1. Navigate to your container registry.
2. In **Services**, select **Tasks** > **Runs**.
3. Select a **Run Id** to view the run status and run logs. The log contains the same information as a streamed log, if one is generated.

To view a log using the Azure CLI, run az acr task logs and specify a run ID, a task name, or a specific image create by a build task. If a task name is specified, the command shows the log for the last created run.

The following example outputs the log for the run with ID *cf4*:

```
az acr task logs --registry mycontainerregistry1220 \
  --run-id cf4
```

Alternative log storage

You might want to store task run logs on a local file system or use an alternative archiving solution such as Azure Storage.

For example, create a local *tasklogs* directory, and redirect the output of az acr task logs to a local file:

```
mkdir ~/tasklogs

az acr task logs --registry mycontainerregistry1220 \
  --run-id cf4 > ~/tasklogs/cf4.log
```

You can also save local log files to Azure Storage. For example, use the Azure CLI, the Azure portal, or other methods to upload files to a storage account.

Next steps

- Learn more about Azure Container Registry Tasks

What is a connected registry?

8/22/2022 • 4 minutes to read • Edit Online

In this article, you learn about the *connected registry* feature of Azure Container Registry. A connected registry is an on-premises or remote replica that synchronizes container images and other OCI artifacts with your cloud-based Azure container registry. Use a connected registry to help speed up access to registry artifacts on-premises and to build advanced scenarios, for example using nested IoT Edge.

> **NOTE**
>
> The connected registry is a preview feature of the **Premium** container registry service tier, and subject to limitations. For information about registry service tiers and limits, see Azure Container Registry service tiers.

Available regions

- Asia East
- EU North
- EU West
- US East

Scenarios

A cloud-based Azure container registry provides features including geo-replication, integrated security, Azure-managed storage, and integration with Azure development and deployment pipelines. At the same time, customers are extending their cloud investments to theiron-premisesand field solutions.

To run with the required performance and reliability in on-premises or remote environments, container workloads need container images and related artifacts to be available nearby. The connected registry provides a performant, on-premises registry solution that regularly synchronizes content with a cloud-based Azure container registry.

Scenarios for a connected registry include:

- Connected factories
- Point-of-sale retail locations
- Shipping, oil-drilling, mining, and other occasionally connected environments

How does the connected registry work?

The following image shows a typical deployment model for the connected registry.

Deployment

Each connected registry is a resource you manage using a cloud-based Azure container registry. The top parent in the connected registry hierarchy is an Azure container registry in an Azure cloud or in a private deployment of Azure Stack Hub.

Use Azure tools to install the connected registry on a server or device on your premises, or an environment that supports container workloads on-premises such as Azure IoT Edge.

The connected registry's *activation status* indicates whether it's deployed on-premises.

- **Active** - The connected registry is currently deployed on-premises. It can't be deployed again until it is deactivated.
- **Inactive** - The connected registry is not deployed on-premises. It can be deployed at this time.

Content synchronization

The connected registry regularly accesses the cloud registry to synchronize container images and OCI artifacts.

It can also be configured to synchronize a subset of the repositories from the cloud registry or to synchronize only during certain intervals to reduce traffic between the cloud and the premises.

Modes

A connected registry can work in one of two modes: *ReadWrite* or *ReadOnly*

- **ReadWrite mode** - The default mode allows clients to pull and push artifacts (read and write) to the connected registry. Artifacts that are pushed to the connected registry will be synchronized with the cloud registry.

  The ReadWrite mode is useful when a local development environment is in place. The images are pushed to the local connected registry and from there synchronized to the cloud.

- **ReadOnly mode** - When the connected registry is in ReadOnly mode, clients may only pull (read) artifacts. This configuration is used for nested IoT Edge scenarios, or other scenarios where clients need to pull a container image to operate.

Registry hierarchy

Each connected registry must be connected to a parent. The top parent is the cloud registry. For hierarchical scenarios such as nested IoT Edge, you can nest connected registries in either mode. The parent connected to the cloud registry can operate in either mode.

Child registries must be compatible with their parent capabilities. Thus, both ReadWrite and ReadOnly mode connected registries can be children of a connected registry operating in ReadWrite mode, but only a ReadOnly mode registry can be a child of a connected registry operating in ReadOnly mode.

Client access

On-premises clients use standard tools such as the Docker CLI to push or pull content from a connected registry. To manage client access, you create Azure container registry tokens for access to each connected registry. You can scope the client tokens for pull or push access to one or more repositories in the registry.

Each connected registry also needs to regularly communicate with its parent registry. For this purpose, the registry is issued a synchronization token (*sync token*) by the cloud registry. This token is used to authenticate with its parent registry for synchronization and management operations.

For more information, see Manage access to a connected registry.

Limitations

- Number of tokens and scope maps is limited to 20,000 each for a single container registry. This indirectly limits the number of connected registries for a cloud registry, because every connected registry needs a sync and client token.
- Number of repository permissions in a scope map is limited to 500.
- Number of clients for the connected registry is currently limited to 20.
- Image locking through repository/manifest/tag metadata is not currently supported for connected registries.
- Repository delete is not supported on the connected registry using ReadOnly mode.
- Resource logs for connected registries are currently not supported.
- Connected registry is coupled with the registry's home region data endpoint. Automatic migration for geo-replication is not supported.
- Deletion of a connected registry needs manual removal of the containers on-premises as well as removal of the respective scope map or tokens in the cloud.
- Connected registry sync limitations are as follows:

- For continuous sync:
  - `minMessageTtl` is 1 day
  - `maxMessageTtl` is 90 days
- For occasionally connected scenarios, where you want to specify sync window:
  - `minSyncWindow` is 1 hr
  - `maxSyncWindow` is 7 days

Next steps

In this overview, you learned about the connected registry and some basic concepts. Continue to the one of the following articles to learn about specific scenarios where connected registry can be utilized.

Overview: Connected registry access

Overview: Connected registry and IoT Edge

Using connected registry with Azure IoT Edge

8/22/2022 • 3 minutes to read • Edit Online

In this article, you learn about using an Azure connected registry in hierarchical IoT Edge scenarios. The connected container registry can be deployed as an IoT Edge module and play an essential role in serving container images required by the devices in the hierarchy.

What is a hierarchical IoT Edge deployment?

Azure IoT Edge allows you to deploy IoT Edge devices across networks organized in hierarchical layers. Each layer in a hierarchy is a gateway device that handles messages and requests from devices in the layer beneath it. You can structure a hierarchy of devices so that only the top layer has connectivity to the cloud, and the lower layers can only communicate with adjacent north and south layers. This network layering is the foundation of most industrial networks, which follow the ISA-95 standard.

To learn how to create a hierarchy of IoT Edge devices, see Tutorial: Create a hierarchy of IoT Edge devices

How do I use connected registry in hierarchical IoT Edge scenarios?

The following image shows how the connected registry can be used to support the hierarchical deployment of IoT Edge. Solid gray lines show the actual network flow, while the dashed lines show the logical communication between components and the connected registries.

Top layer

The top layer of the example architecture, *Layer 5: Enterprise Network*, is managed by IT and can access the container registry for Contoso in the Azure cloud. The connected registry is deployed as an IoT Edge module on the IoT Edge VM and can directly communicate with the cloud registry to pull and push images and artifacts.

The connected registry is shown as working in the default ReadWrite mode. Clients of this connected registry can pull and push images and artifacts to it. Pushed images will be synchronized with the cloud registry. If pushes are not required in that layer, the connected registry can be changed to operate in ReadOnly mode.

For steps to deploy the connected registry as an IoT Edge module at this level, see Quickstart - Deploy a connected registry to an IoT Edge device.

Nested layers

The next lower layer, *Layer 4: Site Business Planning and Logistics*, is configured to communicate only with Layer 5. Thus, when deploying the IoT Edge VM on Layer 4, it needs to pull the module images from the connected registry on Layer 5 instead.

You can also deploy a connected registry working in ReadOnly mode to serve the layers below. This is illustrated with the IoT Edge VM on *Layer 3: Industrial Security Zone*. That VM must pull the module images from the connected registry on *Layer 4*. If clients on lower layers need to be served, a connected registry in ReadOnly mode can be deployed on Layer 3, and so on.

In this architecture, the connected registries deployed on each layer are configured to synchronize the images

with the connected registry on the layer above. The connected registries are deployed as IoT Edge modules and leverage the IoT Edge mechanisms for deployment and network routing.

For steps to deploy the connected registry on nested IoT Edge devices, see Quickstart: Deploy connected registry on nested IoT Edge devices.

Next steps

In this overview, you learned about the use of the connected registry in hierarchical IoT Edge scenarios. Continue to the following articles to learn how to configure and deploy a connected registry to your IoT Edge device.

Quickstart - Create connected registry using the CLI

Quickstart - Create connected registry using the portal

Quickstart - Deploy a connected registry to an IoT Edge device

Tutorial: Deploy connected registry on nested IoT Edge devices

Quickstart: Create a connected registry using the Azure CLI

8/22/2022 • 5 minutes to read • Edit Online

In this quickstart, you use the Azure CLI to create a connected registry resource in Azure. The connected registry feature of Azure Container Registry allows you to deploy a registry remotely or on your premises and synchronize images and other artifacts with the cloud registry.

Here you create two connected registry resources for a cloud registry: one connected registry allows read and write (artifact pull and push) functionality and one allows read-only functionality.

After creating a connected registry, you can follow other guides to deploy and use it on your on-premises or remote infrastructure.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- Azure Container registry - If you don't already have a container registry, create one (Premium tier required) in a region that supports connected registries.

Enable the dedicated data endpoint for the cloud registry

Enable the dedicated data endpoint for the Azure container registry in the cloud by using the az acr update command. This step is needed for a connected registry to communicate with the cloud registry.

```
# Set the REGISTRY_NAME environment variable to identify the existing cloud registry
REGISTRY_NAME=<container-registry-name>

az acr update --name $REGISTRY_NAME \
  --data-endpoint-enabled
```

Import images to your cloud registry

Import the following container images to your cloud registry using the az acr import command. Skip this step if you already imported these images.

Connected registry image

To support nested IoT Edge scenarios, the container image for the connected registry runtime must be available in your private Azure container registry. Use the az acr import command to import the connected registry image into your private registry.

```
# Use the REGISTRY_NAME variable in the following Azure CLI commands to identify the registry
REGISTRY_NAME=<container-registry-name>

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/acr/connected-registry:0.8.0
```

IoT Edge and API proxy images

To support the connected registry on nested IoT Edge, you need to deploy modules for the IoT Edge and API proxy. Import these images into your private registry.

The IoT Edge API proxy module allows an IoT Edge device to expose multiple services using the HTTPS protocol on the same port such as 443.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-agent:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-hub:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-api-proxy:1.1.2

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-diagnostics:1.2.4
```

Hello-world image

For testing the connected registry, import the `hello-world` image. This repository will be synchronized to the connected registry and pulled by the connected registry clients.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/hello-world:1.1.2
```

Create a connected registry resource for read and write functionality

Create a connected registry using the az acr connected-registry create command. The connected registry name must start with a letter and contain only alphanumeric characters. It must be 5 to 40 characters long and unique in the hierarchy for this Azure container registry.

```
# Set the CONNECTED_REGISTRY_RW environment variable to provide a name for the connected registry with
read/write functionality
CONNECTED_REGISTRY_RW=<connnected-registry-name>

az acr connected-registry create --registry $REGISTRY_NAME \
  --name $CONNECTED_REGISTRY_RW \
  --repository "hello-world" "acr/connected-registry" "azureiotedge-agent" "azureiotedge-hub" "azureiotedge-
api-proxy"
```

This command creates a connected registry resource whose name is the value of *$CONNECTED_REGISTRY_RW* and links it to the cloud registry whose name is the value of *$REGISTRY_NAME*. In later quickstart guides, you learn about options to deploy the connected registry.

- The specified repositories will be synchronized between the cloud registry and the connected registry once it is deployed.
- Because no `--mode` option is specified for the connected registry, it is created in the default ReadWrite mode.
- Because there is no synchronization schedule defined for this connected registry, the repositories will be synchronized between the cloud registry and the connected registry without interruptions.

> **IMPORTANT**
>
> To support nested scenarios where lower layers have no Internet access, you must always allow synchronization of the `acr/connected-registry` repository. This repository contains the image for the connected registry runtime.

Create a connected registry resource for read-only functionality

You can also use the az acr connected-registry create command to create a connected registry with read-only functionality.

```
# Set the CONNECTED_REGISTRY_READ environment variable to provide a name for the connected registry with
read-only functionality
CONNECTED_REGISTRY_RO=<connnected-registry-name>
az acr connected-registry create --registry $REGISTRY_NAME \
  --parent $CONNECTED_REGISTRY_RW \
  --name $CONNECTED_REGISTRY_RO \
  --repository "hello-world" "acr/connected-registry" "azureiotedge-agent" "azureiotedge-hub" "azureiotedge-
api-proxy" \
  --mode ReadOnly
```

This command creates a connected registry resource whose name is the value of *$CONNECTED_REGISTRY_RO* and links it to the cloud registry named with the value of *$REGISTRY_NAME*.

- The specified repositories will be synchronized between the parent registry named with the value of *$CONNECTED_REGISTRY_RW* and the connected registry once deployed.
- This resource is created in the ReadOnly mode, which enables read-only (artifact pull) functionality once deployed.
- Because there is no synchronization schedule defined for this connected registry, the repositories will be synchronized between the parent registry and the connected registry without interruptions.

Verify that the resources are created

You can use the connected registry az acr connected-registry list command to verify that the resources are created.

```
az acr connected-registry list \
  --registry $REGISTRY_NAME \
  --output table
```

You should see a response as follows. Because the connected registries are not yet deployed, the connection state of "Offline" indicates that they are currently disconnected from the cloud.

```
NAME                 MODE        CONNECTION STATE    PARENT               LOGIN SERVER    LAST SYNC (UTC)
-------------------  --------    ------------------  -------------------  --------------  -----------------
myconnectedregrw     ReadWrite   Offline
myconnectedregro     ReadOnly    Offline             myconnectedregrw
```

Next steps

In this quickstart, you used the Azure CLI to create two connected registry resources in Azure. Those new connected registry resources are tied to your cloud registry and allow synchronization of artifacts with the cloud registry.

Continue to the connected registry deployment guides to learn how to deploy and use a connected registry on your IoT Edge infrastructure.

Quickstart: Deploy connected registry on IoT Edge

Quickstart: Create a connected registry using the Azure portal

8/22/2022 • 5 minutes to read • Edit Online

In this quickstart, you use the Azure portal to create a connected registry resource in Azure. The connected registry feature of Azure Container Registry allows you to deploy a registry remotely or on your premises and synchronize images and other artifacts with the cloud registry.

Here you create two connected registry resources for a cloud registry: one connected registry allows read and write (artifact pull and push) functionality and one allows read-only functionality.

After creating a connected registry, you can follow other guides to deploy and use it on your on-premises or remote infrastructure.

Prerequisites

- Azure Container registry - If you don't already have a container registry, create one (Premium tier required) in a region that supports connected registries.

To import images to the container registry, use the Azure CLI:

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

Enable the dedicated data endpoint for the cloud registry

Enable the dedicated data endpoint for the Azure container registry in the cloud. This step is needed for a connected registry to communicate with the cloud registry.

1. In the Azure portal, navigate to your container registry.
2. Select **Networking** > **Public access**. Select the **Enable dedicated data endpoint** checkbox.
3. Select **Save**.

Import images to your cloud registry

Import the following container images to your cloud registry using the az acr import command. Skip this step if

you already imported these images.

Connected registry image

To support nested IoT Edge scenarios, the container image for the connected registry runtime must be available in your private Azure container registry. Use the az acr import command to import the connected registry image into your private registry.

```
# Use the REGISTRY_NAME variable in the following Azure CLI commands to identify the registry
REGISTRY_NAME=<container-registry-name>

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/acr/connected-registry:0.8.0
```

IoT Edge and API proxy images

To support the connected registry on nested IoT Edge, you need to deploy modules for the IoT Edge and API proxy. Import these images into your private registry.

The IoT Edge API proxy module allows an IoT Edge device to expose multiple services using the HTTPS protocol on the same port such as 443.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-agent:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-hub:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-api-proxy:1.1.2

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-diagnostics:1.2.4
```

Hello-world image

For testing the connected registry, import the `hello-world` image. This repository will be synchronized to the connected registry and pulled by the connected registry clients.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/hello-world:1.1.2
```

Create a connected registry resource for read and write functionality

The following steps create a connected registry in ReadWrite mode that is linked to the cloud registry.

1. In the Azure portal, navigate to your container registry.
2. Select **Connected registries (Preview) > + Create**.
3. Enter or select the values in the following table, and select **Save**.

ITEM	DESCRIPTION
Parent	Select **No parent** for a connected registry linked to the cloud registry.
Mode	Select **ReadWrite**.
Name	The connected registry name must start with a letter and contain only alphanumeric characters. It must be 5 to 40 characters long and unique in the hierarchy for this Azure container registry.
Logging properties	Accept the default settings.
Sync properties	Accept the default settings. Because there is no synchronization schedule defined by default, the repositories will be synchronized between the cloud registry and the connected registry without interruptions.
Repositories	Select or enter the names of the repositories you imported in the previous step. The specified repositories will be synchronized between the cloud registry and the connected registry once it is deployed.

> **IMPORTANT**
> To support nested scenarios where lower layers have no Internet access, you must always allow synchronization of the `acr/connected-registry` repository. This repository contains the image for the connected registry runtime.

Create a connected registry resource for read-only functionality

The following steps create a connected registry in ReadOnly mode whose parent is the connected registry you created in the previous section. This connected registry enables read-only (artifact pull) functionality once deployed.

1. In the Azure portal, navigate to your container registry.
2. Select **Connected registries (Preview) > + Create**.
3. Enter or select the values in the following table, and select **Save**.

ITEM	DESCRIPTION
Parent	Select the connected registry you created previously.
Mode	Select **ReadOnly**.
Name	The connected registry name must start with a letter and contain only alphanumeric characters. It must be 5 to 40 characters long and unique in the hierarchy for this Azure container registry.
Logging properties	Accept the default settings.
Sync properties	Accept the default settings. Because there is no synchronization schedule defined by default, the repositories will be synchronized between the cloud registry and the connected registry without interruptions.
Repositories	Select or enter the names of the repositories you imported in the previous step. The specified repositories will be synchronized between the parent registry and the connected registry once it is deployed.

View connected registry properties

Select a connected registry in the portal to view its properties, such as its connection status (Offline, Online, or Unhealthy) and whether it has been activated (deployed on-premises). In the following example, the connected registry is not deployed. Its connection state of "Offline" indicates that it is currently disconnected from the cloud.

From this view, you can also generate a connection string and optionally generate passwords for the sync token. A connection string contains configuration settings used for deploying a connected registry and synchronizing content with a parent registry.

Next steps

In this quickstart, you used the Azure portal to create two connected registry resources in Azure. Those new connected registry resources are tied to your cloud registry and allow synchronization of artifacts with the cloud registry.

Continue to the connected registry deployment guides to learn how to deploy and use a connected registry on your IoT Edge infrastructure.

Quickstart: Deploy connected registry on IoT Edge

Quickstart: Deploy a connected registry to an IoT Edge device

8/22/2022 • 7 minutes to read • Edit Online

In this quickstart, you use the Azure CLI to deploy a connected registry as a module on an Azure IoT Edge device. The IoT Edge device can access the parent Azure container registry in the cloud.

For an overview of using a connected registry with IoT Edge, see Using connected registry with Azure IoT Edge. This scenario corresponds to a device at the top layer of an IoT Edge hierarchy.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- Azure IoT Hub and IoT Edge device. For deployment steps, see Quickstart: Deploy your first IoT Edge module to a virtual Linux device.

  > **IMPORTANT**
  > For later access to the modules deployed on the IoT Edge device, make sure that you open the ports 8000, 5671, and 8883 on the device. For configuration steps, see How to open ports to a virtual machine with the Azure portal.

- Connected registry resource in Azure. For deployment steps, see quickstarts using the Azure CLI or Azure portal.
  - A connected registry in either `ReadWrite` or `ReadOnly` mode can be used in this scenario.
  - In the commands in this article, the connected registry name is stored in the environment variable *CONNECTED_REGISTRY_RW*.

Import images to your cloud registry

Import the following container images to your cloud registry using the az acr import command. Skip this step if you already imported these images.

Connected registry image

To support nested IoT Edge scenarios, the container image for the connected registry runtime must be available in your private Azure container registry. Use the az acr import command to import the connected registry image into your private registry.

```
# Use the REGISTRY_NAME variable in the following Azure CLI commands to identify the registry
REGISTRY_NAME=<container-registry-name>

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/acr/connected-registry:0.8.0
```

IoT Edge and API proxy images

To support the connected registry on nested IoT Edge, you need to deploy modules for the IoT Edge and API proxy. Import these images into your private registry.

The IoT Edge API proxy module allows an IoT Edge device to expose multiple services using the HTTPS protocol on the same port such as 443.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-agent:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-hub:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-api-proxy:1.1.2

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-diagnostics:1.2.4
```

Hello-world image

For testing the connected registry, import the `hello-world` image. This repository will be synchronized to the connected registry and pulled by the connected registry clients.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/hello-world:1.1.2
```

Retrieve connected registry configuration

Before deploying the connected registry to the IoT Edge device, you need to retrieve configuration settings from the connected registry resource in Azure.

Use the az acr connected-registry get-settings command to get the settings information required to install a connected registry. The following example specifies HTTPS as the parent protocol. This protocol is required when the parent registry is a cloud registry.

```
az acr connected-registry get-settings \
  --registry $REGISTRY_NAME \
  --name $CONNECTED_REGISTRY_RW \
  --parent-protocol https
```

By default, the settings information doesn't include the sync token password, which is also needed to deploy the connected registry. Optionally, generate one of the passwords by passing the `--generate-password 1` or `generate-password 2` parameter. Save the generated password to a safe location. It cannot be retrieved again.

> **WARNING**
> Regenerating a password rotates the sync token credentials. If you configured a device using the previous password, you need to update the configuration.

Command output includes the registry connection string and related settings. The following example output shows the connection string for the connected registry named *myconnectedregistry* with parent registry *contosoregistry*:

```
{
  "ACR_REGISTRY_CONNECTION_STRING":
"ConnectedRegistryName=myconnectedregistry;SyncTokenName=myconnectedregistry-sync-
token;SyncTokenPassword=xxxxxxxxxxxxxxxx;ParentGatewayEndpoint=contosoregistry.eastus.data.azurecr.io;Parent
EndpointProtocol=https"
}
```

Configure a deployment manifest for IoT Edge

A deployment manifest is a JSON document that describes which modules to deploy to the IoT Edge device. For more information, see Understand how IoT Edge modules can be used, configured, and reused.

To deploy the connected registry and API proxy modules using the Azure CLI, save the following deployment manifest locally as a `manifest.json` file. You will use the file path in the next section when you run the command to apply the configuration to your device.

Connected registry module settings

- Use the token credentials and connection string from the previous sections to update the relevant JSON values in the `env` node.
- The following environment variables are optional in the `env` node:

VARIABLE	DESCRIPTION
`ACR_REGISTRY_LOGIN_SERVER`	Specifies a unique hostname or FQDN. If used, the connected registry only accepts requests made to this login server value. If no value is provided, then the connected registry can be accessed with any login server value.
`ACR_REGISTRY_CERTIFICATE_VOLUME`	If your connected registry will be accessible via HTTPS, points to the volume where the HTTPS certificates are stored. If not set, the default location is `/var/acr/certs`.

VARIABLE	DESCRIPTION
`ACR_REGISTRY_DATA_VOLUME`	Overwrites the default location `/var/acr/data` where the images will be stored by the connected registry. This location must match the volume bind for the container.

> **IMPORTANT**
> If the connected registry listens on a port different from 80 and 443, the `ACR_REGISTRY_LOGIN_SERVER` value (if specified) must include the port. Example: `192.168.0.100:8080`.

- A `HostPort` binding for the connected registry should be set if the API proxy module isn't used. Example:

```
"createOptions": "{\"HostConfig\":{\"Binds\":[\"/home/azureuser/connected-registry:/var/acr/data\"],\"PortBindings\":{\"8080/tcp\":[{\"HostPort\":\"8080\"}]}}}"
```

API proxy module settings

- The API proxy will listen on port 8000 configured as `NGINX_DEFAULT_PORT`. For more information about the API proxy settings, see the IoT Edge GitHub repo.

```
{
    "modulesContent": {
        "$edgeAgent": {
            "properties.desired": {
                "modules": {
                    "connected-registry": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/acr/connected-registry:0.7.0",
                            "createOptions": "{\"HostConfig\":{\"Binds\":[\"/home/azureuser/connected-registry:/var/acr/data\"]}}"
                        },
                        "type": "docker",
                        "env": {
                            "ACR_REGISTRY_CONNECTION_STRING": {
                                "value": "ConnectedRegistryName=<REPLACE_WITH_CONNECTED_REGISTRY_NAME>;SyncTokenName=<REPLACE_WITH_SYNC_TOKEN_NAME>;SyncTokenPassword=REPLACE_WITH_SYNC_TOKEN_PASSWORD;ParentGatewayEndpoint=<REPLACE_WITH_CLOUD_REGISTRY_NAME>.<REPLACE_WITH_CLOUD_REGISTRY_REGION>.data.azurecr.io;ParentEndpointProtocol=https"
                            }
                        },
                        "status": "running",
                        "restartPolicy": "always",
                        "version": "1.0"
                    },
                    "IoTEdgeAPIProxy": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/azureiotedge-api-proxy:1.1.2",
                            "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"8000/tcp\":[{\"HostPort\":\"8000\"}]}}}"
                        },
                        "type": "docker",
                        "env": {
                            "NGINX_DEFAULT_PORT": {
                                "value": "8000"
                            },
                            "CONNECTED_ACR_ROUTE_ADDRESS": {
```

```
                            "CONNECTED_ACR_ROUTE_ADDRESS": {
                                "value": "connected-registry:8080"
                            },
                            "BLOB_UPLOAD_ROUTE_ADDRESS": {
                                "value": "AzureBlobStorageonIoTEdge:11002"
                            }
                        },
                        "status": "running",
                        "restartPolicy": "always",
                        "version": "1.0"
                    }
                },
                "runtime": {
                    "settings": {
                        "minDockerVersion": "v1.25",
                        "registryCredentials": {
                            "cloudregistry": {
                                "address": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io",
                                "password": "<REPLACE_WITH_SYNC_TOKEN_PASSWORD>",
                                "username": "<REPLACE_WITH_SYNC_TOKEN_NAME>"
                            }
                        }
                    },
                    "type": "docker"
                },
                "schemaVersion": "1.1",
                "systemModules": {
                    "edgeAgent": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/azureiotedge-
agent:1.2.4",
                            "createOptions": ""
                        },
                        "type": "docker",
                        "env": {
                            "SendRuntimeQualityTelemetry": {
                                "value": "false"
                            }
                        }
                    },
                    "edgeHub": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/azureiotedge-hub:1.2.4",
                            "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"443/tcp\":
[{\"HostPort\":\"443\"}],\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}]}}}"
                        },
                        "type": "docker",
                        "status": "running",
                        "restartPolicy": "always"
                    }
                }
            }
        },
        "$edgeHub": {
            "properties.desired": {
                "routes": {
                    "route": "FROM /messages/* INTO $upstream"
                },
                "schemaVersion": "1.1",
                "storeAndForwardConfiguration": {
                    "timeToLiveSecs": 7200
                }
            }
        }
    }
}
```

Deploy the connected registry and API proxy modules on IoT Edge

Use the following command to deploy the connected registry and API proxy modules on the IoT Edge device, using the deployment manifest created in the previous section. Provide the ID of the IoT Edge top layer device and the name of the IoT Hub where indicated.

```
# Set the IOT_EDGE_TOP_LAYER_DEVICE_ID and IOT_HUB_NAME environment variables for use in the following Azure
CLI command
IOT_EDGE_TOP_LAYER_DEVICE_ID=<device-id>
IOT_HUB_NAME=<hub-name>

az iot edge set-modules \
  --device-id $IOT_EDGE_TOP_LAYER_DEVICE_ID \
  --hub-name $IOT_HUB_NAME \
  --content manifest.json
```

For details, see Deploy Azure IoT Edge modules with Azure CLI.

To check the status of the connected registry, use the following az acr connected-registry show command. The name of the connected registry is the value of *$CONNECTED_REGISTRY_RW*.

```
az acr connected-registry show \
  --registry $REGISTRY_NAME \
  --name $CONNECTED_REGISTRY_RW \
  --output table
```

After successful deployment, the connected registry shows a status of `Online`.

Next steps

In this quickstart, you learned how to deploy a connected registry to an IoT Edge device. Continue to the next guides to learn how to pull images from the newly deployed connected registry or to deploy the connected registry on nested IoT Edge devices.

Pull images from a connected registry

Tutorial: Deploy connected registry on nested IoT Edge devices

Tutorial: Deploy a connected registry to a nested IoT Edge hierarchy

8/22/2022 • 13 minutes to read • Edit Online

In this tutorial, you use Azure CLI commands to create a two-layer hierarchy of Azure IoT Edge devices and deploy a connected registry as a module at each layer. In this scenario, a device in the top layer communicates with a cloud registry. A device in the lower layer communicates with its connected registry parent in the top layer.

For an overview of using a connected registry with IoT Edge, see Using connected registry with Azure IoT Edge.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- Azure IoT Hub. For deployment steps, see Create an IoT hub using the Azure portal.
- Two connected registry resources in Azure. For deployment steps, see quickstarts using the Azure CLI or Azure portal.
  - For the top layer, the connected registry can be in either ReadWrite or ReadOnly mode. This article assumes ReadWrite mode, and the connected registry name is stored in the environment variable `$CONNECTED_REGISTRY_RW`.
  - For the lower layer, the connected registry must be in ReadOnly mode. This article assumes the connected registry name is stored in the environment variable `$CONNECTED_REGISTRY_RO`.

Import images to your cloud registry

Import the following container images to your cloud registry using the az acr import command. Skip this step if you already imported these images.

Connected registry image

To support nested IoT Edge scenarios, the container image for the connected registry runtime must be available in your private Azure container registry. Use the az acr import command to import the connected registry image into your private registry.

```
# Use the REGISTRY_NAME variable in the following Azure CLI commands to identify the registry
REGISTRY_NAME=<container-registry-name>

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/acr/connected-registry:0.8.0
```

IoT Edge and API proxy images

To support the connected registry on nested IoT Edge, you need to deploy modules for the IoT Edge and API proxy. Import these images into your private registry.

The IoT Edge API proxy module allows an IoT Edge device to expose multiple services using the HTTPS protocol on the same port such as 443.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-agent:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-hub:1.2.4

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-api-proxy:1.1.2

az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/azureiotedge-diagnostics:1.2.4
```

Hello-world image

For testing the connected registry, import the `hello-world` image. This repository will be synchronized to the connected registry and pulled by the connected registry clients.

```
az acr import \
  --name $REGISTRY_NAME \
  --source mcr.microsoft.com/hello-world:1.1.2
```

Retrieve connected registry configuration

To deploy each connected registry to the IoT Edge device in the hierarchy, you need to retrieve configuration settings from the connected registry resource in Azure. If needed, run the az acr connected-registry get-settings command for each connected registry to retrieve the configuration.

By default, the settings information doesn't include the sync token password, which is also needed to deploy the connected registry. Optionally, generate one of the passwords by passing the `--generate-password 1` or `--generate-password 2` parameter. Save the generated password to a safe location. It can't be retrieved again.

> **WARNING**
> Regenerating a password rotates the sync token credentials. If you configured a device using the previous password, you need to update the configuration.

```
# Use the REGISTRY_NAME variable in the following Azure CLI commands to identify the registry
REGISTRY_NAME=<container-registry-name>

# Run the command for each registry resource in the hierarchy

az acr connected-registry get-settings \
  --registry $REGISTRY_NAME \
  --name $CONNECTED_REGISTRY_RW \
  --parent-protocol https

az acr connected-registry get-settings \
  --registry $REGISTRY_NAME \
  --name $CONNECTED_REGISTRY_RO \
  --parent-protocol https
```

Command output includes the registry connection string and related settings. The following example output shows the connection string for the connected registry named *myconnectedregistry* with parent registry *contosoregistry*.

```
{
  "ACR_REGISTRY_CONNECTION_STRING":
"ConnectedRegistryName=myconnectedregistry;SyncTokenName=myconnectedregistry-sync-
token;SyncTokenPassword=xxxxxxxxxxxxxxxx;ParentGatewayEndpoint=contosoregistry.eastus.data.azurecr.io;Parent
EndpointProtocol=https"
}
```

Configure deployment manifests

A deployment manifest is a JSON document that describes which modules to deploy to an IoT Edge device. For more information, see Understand how IoT Edge modules can be used, configured, and reused.

To deploy the connected registry module on each IoT Edge device using the Azure CLI, save the following deployment manifests locally as JSON files. Use the information from the previous sections to update the relevant JSON values in each manifest. Use the file paths in the next section when you run the command to apply the configuration to your device.

Deployment manifest for the top layer

For the device at the top layer, create a deployment manifest file `deploymentTopLayer.json` with the following content. This manifest is similar to the one used in Quickstart: Deploy a connected registry to an IoT Edge device.

> **NOTE**
> If you already deployed a connected registry to a top layer IoT Edge device using the quickstart, you can use it at the top layer of a nested hierarchy. Modify the deployment steps in this tutorial to configure it in the hierarchy (not shown).

Connected registry module settings

- Use the token credentials and connection string from the previous sections to update the relevant JSON values in the `env` node.
- The following environment variables are optional in the `env` node:

VARIABLE	DESCRIPTION

VARIABLE	DESCRIPTION
`ACR_REGISTRY_LOGIN_SERVER`	Specifies a unique hostname or FQDN. If used, the connected registry only accepts requests made to this login server value. If no value is provided, then the connected registry can be accessed with any login server value.
`ACR_REGISTRY_CERTIFICATE_VOLUME`	If your connected registry will be accessible via HTTPS, points to the volume where the HTTPS certificates are stored. If not set, the default location is `/var/acr/certs`.
`ACR_REGISTRY_DATA_VOLUME`	Overwrites the default location `/var/acr/data` where the images will be stored by the connected registry. This location must match the volume bind for the container.

> **IMPORTANT**
> If the connected registry listens on a port different from 80 and 443, the `ACR_REGISTRY_LOGIN_SERVER` value (if specified) must include the port. Example: `192.168.0.100:8080`.

- A `HostPort` binding for the connected registry should be set if the API proxy module isn't used. Example:

```
  "createOptions": "{\"HostConfig\":{\"Binds\":[\"/home/azureuser/connected-
registry:/var/acr/data\"],\"PortBindings\":{\"8080/tcp\":[{\"HostPort\":\"8080\"}]}}}"
```

API proxy module settings

- The API proxy will listen on port 8000 configured as `NGINX_DEFAULT_PORT`. For more information about the API proxy settings, see the IoT Edge GitHub repo.

```
{
    "modulesContent": {
        "$edgeAgent": {
            "properties.desired": {
                "modules": {
                    "connected-registry": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/acr/connected-
registry:0.7.0",
                            "createOptions": "{\"HostConfig\":{\"Binds\":[\"/home/azureuser/connected-
registry:/var/acr/data\"]}}"
                        },
                        "type": "docker",
                        "env": {
                            "ACR_REGISTRY_CONNECTION_STRING": {
                                "value": "ConnectedRegistryName=
<REPLACE_WITH_CONNECTED_REGISTRY_NAME>;SyncTokenName=
<REPLACE_WITH_SYNC_TOKEN_NAME>;SyncTokenPassword=REPLACE_WITH_SYNC_TOKEN_PASSWORD;ParentGatewayEndpoint=
<REPLACE_WITH_CLOUD_REGISTRY_NAME>.
<REPLACE_WITH_CLOUD_REGISTRY_REGION>.data.azurecr.io;ParentEndpointProtocol=https"
                            }
                        },
                        "status": "running",
                        "restartPolicy": "always",
```

```
                        "version": "1.0"
                    },
                    "IoTEdgeAPIProxy": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/azureiotedge-api-
proxy:1.1.2",
                            "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"8000/tcp\":
[{\"HostPort\":\"8000\"}]}}}"
                        },
                        "type": "docker",
                        "env": {
                            "NGINX_DEFAULT_PORT": {
                                "value": "8000"
                            },
                            "CONNECTED_ACR_ROUTE_ADDRESS": {
                                "value": "connected-registry:8080"
                            },
                            "BLOB_UPLOAD_ROUTE_ADDRESS": {
                                "value": "AzureBlobStorageonIoTEdge:11002"
                            }
                        },
                        "status": "running",
                        "restartPolicy": "always",
                        "version": "1.0"
                    }
                },
                "runtime": {
                    "settings": {
                        "minDockerVersion": "v1.25",
                        "registryCredentials": {
                            "cloudregistry": {
                                "address": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io",
                                "password": "<REPLACE_WITH_SYNC_TOKEN_PASSWORD>",
                                "username": "<REPLACE_WITH_SYNC_TOKEN_NAME>"
                            }
                        }
                    },
                    "type": "docker"
                },
                "schemaVersion": "1.1",
                "systemModules": {
                    "edgeAgent": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/azureiotedge-
agent:1.2.4",
                            "createOptions": ""
                        },
                        "type": "docker",
                        "env": {
                            "SendRuntimeQualityTelemetry": {
                                "value": "false"
                            }
                        }
                    },
                    "edgeHub": {
                        "settings": {
                            "image": "<REPLACE_WITH_CLOUD_REGISTRY_NAME>.azurecr.io/azureiotedge-hub:1.2.4",
                            "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"443/tcp\":
[{\"HostPort\":\"443\"}],\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}]}}}"
                        },
                        "type": "docker",
                        "status": "running",
                        "restartPolicy": "always"
                    }
                }
            }
        },
        "$edgeHub": {
            "properties.desired": {
```

```
            "properties.desired": {
                "routes": {
                    "route": "FROM /messages/* INTO $upstream"
                },
                "schemaVersion": "1.1",
                "storeAndForwardConfiguration": {
                    "timeToLiveSecs": 7200
                }
            }
        }
    }
}
```

Deployment manifest for the lower layer

For the device at the lower layer, create a deployment manifest file *deploymentLowerLayer.json* with the following content.

Overall, the lower layer deployment file is similar to the top layer deployment file. The differences are:

- It pulls the required images from the top layer connected registry instead of from the cloud registry.

  When you set up the top layer connected registry, make sure that it syncs all the required images locally, including `azureiotedge-agent`, `azureiotedge-hub`, `azureiotedge-api-proxy`, and `acr/connected-registry`. The lower layer IoT device needs to pull these images from the top layer connected registry.

- It uses the sync token configured at the lower layer to authenticate with the top layer connected registry.

- It configures the parent gateway endpoint with the top layer connected registry's IP address or FQDN instead of with the cloud registry's FQDN.

> **IMPORTANT**
>
> In the following deployment manifest, `$upstream` is used as the IP address or FQDN of the device hosting the parent connected registry. However, `$upstream` is not supported in an environment variable. The connected registry needs to read the environment variable `ACR_PARENT_GATEWAY_ENDPOINT` to get the parent gateway endpoint. Instead of using `$upstream`, the connected registry supports dynamically resolving the IP address or FQDN from another environment variable.
>
> On the nested IoT Edge, there's an environment variable `$IOTEDGE_PARENTHOSTNAME` on the lower layer that is equal to the IP address or FQDN of the parent device. Manually replace the environment variable as the value of `ParentGatewayEndpoint` in the connection string to avoid hard-coding the parent IP address or FQDN. Because the parent device in this example is running `nginx` on port 8000, pass `$IOTEDGE_PARENTHOSTNAME:8000`. You also need to select the proper protocol in `ParentEndpointProtocol`.

```
{
    "modulesContent": {
        "$edgeAgent": {
            "properties.desired": {
                "modules": {
                    "connected-registry": {
                        "settings": {
                            "image": "$upstream:8000/acr/connected-registry:0.7.0",
                            "createOptions": "{\"HostConfig\":{\"Binds\":[\"/home/azureuser/connected-
registry:/var/acr/data\"]}}"
                        },
                        "type": "docker",
                        "env": {
                            "ACR_REGISTRY_CONNECTION_STRING": {
                                "value": "ConnectedRegistryName=
<REPLACE_WITH_CONNECTED_REGISTRY_NAME>;SyncTokenName=<REPLACE_WITH_SYNC_TOKEN_NAME>;SyncTokenPassword=
<REPLACE_WITH_SYNC_TOKEN_PASSWORD>;ParentGatewayEndpoint=$IOTEDGE_PARENTHOSTNAME:8000;ParentEndpointProtocol
```

```
=https"
                            }
                        },
                        "status": "running",
                        "restartPolicy": "always",
                        "version": "1.0"
                    },
                    "IoTEdgeApiProxy": {
                        "settings": {
                            "image": "$upstream:8000/azureiotedge-api-proxy:1.1.2",
                            "createOptions": "{\"HostConfig\": {\"PortBindings\": {\"8000/tcp\":
[{\"HostPort\": \"8000\"}]}}}"
                        },
                        "type": "docker",
                        "version": "1.0",
                        "env": {
                            "NGINX_DEFAULT_PORT": {
                                "value": "8000"
                            },
                            "CONNECTED_ACR_ROUTE_ADDRESS": {
                                "value": "connected-registry:8080"
                            },
                            "NGINX_CONFIG_ENV_VAR_LIST": {
                                    "value":
"NGINX_DEFAULT_PORT,BLOB_UPLOAD_ROUTE_ADDRESS,CONNECTED_ACR_ROUTE_ADDRESS,IOTEDGE_PARENTHOSTNAME,DOCKER_REQU
EST_ROUTE_ADDRESS"
                            },
                            "BLOB_UPLOAD_ROUTE_ADDRESS": {
                                "value": "AzureBlobStorageonIoTEdge:11002"
                            }
                        },
                        "status": "running",
                        "restartPolicy": "always",
                        "startupOrder": 3
                    }
                },
                "runtime": {
                    "settings": {
                        "minDockerVersion": "v1.25",
                        "registryCredentials": {
                            "connectedregistry": {
                                "address": "$upstream:8000",
                                "password": "<REPLACE_WITH_SYNC_TOKEN_PASSWORD>",
                                "username": "<REPLACE_WITH_SYNC_TOKEN_NAME>"
                            }
                        }
                    },
                    "type": "docker"
                },
                "schemaVersion": "1.1",
                "systemModules": {
                    "edgeAgent": {
                        "settings": {
                            "image": "$upstream:8000/azureiotedge-agent:1.2.4",
                            "createOptions": ""
                        },
                        "type": "docker",
                        "env": {
                            "SendRuntimeQualityTelemetry": {
                                "value": "false"
                            }
                        }
                    },
                    "edgeHub": {
                        "settings": {
                            "image": "$upstream:8000/azureiotedge-hub:1.2.4",
                            "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"443/tcp\":
[{\"HostPort\":\"443\"}],\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}]}}}"
                        },
```

```
                        "type": "docker",
                        "status": "running",
                        "restartPolicy": "always"
                    }
                }
            }
        },
        "$edgeHub": {
            "properties.desired": {
                "routes": {
                    "route": "FROM /messages/* INTO $upstream"
                },
                "schemaVersion": "1.1",
                "storeAndForwardConfiguration": {
                    "timeToLiveSecs": 7200
                }
            }
        }
    }
}
```

Set up and deploy connected registry modules

The following steps are adapted from Tutorial: Create a hierarchy of IoT Edge devices and are specific to deploying connected registry modules in the IoT Edge hierarchy. See that tutorial for details about individual steps.

Create top layer and lower layer devices

Create top layer and lower layer virtual machines using an existing ARM template. The template also installs the IoT Edge agent. If you want to deploy from your own devices instead, see Tutorial: Install or uninstall Azure IoT Edge for Linux to learn how to manually set up the device.

> **IMPORTANT**
>
> For later access to the modules deployed on the top layer device, make sure that you open the following ports inbound: 8000, 443, 5671, 8883. For configuration steps, see How to open ports to a virtual machine with the Azure portal.

Create and configure the hierarchy

Use the `iotedge-config` tool to create and configure your hierarchy by following these steps in the Azure CLI or Azure Cloud Shell:

1. Download the configuration tool.

   ```
   mkdir nestedIotEdgeTutorial
   cd ~/nestedIotEdgeTutorial
   wget -O iotedge_config.tar "https://github.com/Azure-
  Samples/iotedge_config_cli/releases/download/latest/iotedge_config_cli.tar.gz"
   tar -xvf iotedge_config.tar
   ```

   This step creates the `iotedge_config_cli_release` folder in your tutorial directory. The template file used to create your device hierarchy is the `iotedge_config.yaml` file found in `~/nestedIotEdgeTutorial/iotedge_config_cli_release/templates/tutorial`. In the same directory, there are two deployment manifests for top and lower layers: `deploymentTopLayer.json` and `deploymentLowerLayer.json` files.

2. Edit `iotedge_config.yaml` with your information. Edit the `iothub_hostname`, `iot_name`, deployment manifest filenames for the top layer and lower layer, and the client token credentials you created to pull

images from upstream from each layer. The following example is a sample configuration file:

```
config_version: "1.0"

iothub:
    iothub_hostname: <REPLACE_WITH_HUB_NAME>.azure-devices.net
    iothub_name: <REPLACE_WITH_HUB_NAME>
    ## Authentication method used by IoT Edge devices: symmetric_key or x509_certificate
    authentication_method: symmetric_key

    ## Root certificate used to generate device CA certificates. Optional. If not provided a self-
signed CA will be generated
    # certificates:
    #   root_ca_cert_path: ""
    #   root_ca_cert_key_path: ""

    ## IoT Edge configuration template to use
configuration:
    template_config_path: "./templates/tutorial/device_config.toml"
    default_edge_agent: "$upstream:8000/azureiotedge-agent:1.2.4"

    ## Hierarchy of IoT Edge devices to create
edgedevices:
    device_id: top-layer
    edge_agent: "<REPLACE_WITH_REGISTRY_NAME>.azurecr.io/azureiotedge-agent:1.2.4" ## Optional. If
not provided, default_edge_agent will be used
    deployment: "./templates/tutorial/deploymentTopLayer.json" ## Optional. If provided, the given
deployment file will be applied to the newly created device
        # hostname: "FQDN or IP" ## Optional. If provided, install.sh will not prompt user for this
value nor the parent_hostname value
    container_auth: ## The token used to pull the image from cloud registry
        serveraddress: "<REPLACE_WITH_REGISTRY_NAME>.azurecr.io"
        username: "<REPLACE_WITH_SYNC_TOKEN_NAME_FOR_TOP_LAYER>"
        password: "<REPLACE_WITH_SYNC_TOKEN_PASSWORD_FOR_TOP_LAYER>"
    child:
        - device_id: lower-layer
          deployment: "./templates/tutorial/deploymentLowerLayer.json" ## Optional. If provided, the
given deployment file will be applied to the newly created device
           # hostname: "FQDN or IP" ## Optional. If provided, install.sh will not prompt user for
this value nor the parent_hostname value
          container_auth: ## The token used to pull the image from parent connected registry
            serveraddress: "$upstream:8000"
            username: "<REPLACE_WITH_SYNC_TOKEN_NAME_FOR_LOWER_LAYER>"
            password: "<REPLACE_WITH_SYNC_TOKEN_PASSWORD_FOR_LOWER_LAYER>"
```

3. Prepare the top layer and lower layer deployment files: *deploymentTopLayer.json* and *deploymentLowerLayer.json*. Copy the deployment manifest files you created earlier in this article to the following folder: `~/nestedIotEdgeTutorial/iotedge_config_cli_release/templates/tutorial`.

4. Navigate to your *iotedge_config_cli_release* directory and run the tool to create your hierarchy of IoT Edge devices.

```
cd ~/nestedIotEdgeTutorial/iotedge_config_cli_release
./iotedge_config --config
~/nestedIotEdgeTutorial/iotedge_config_cli_release/templates/tutorial/iotedge_config.yaml --output
~/nestedIotEdgeTutorial/iotedge_config_cli_release/outputs -f
```

With the `--output` parameter, the tool creates the device certificates, certificate bundles, and a log file in a directory of your choice. With the `-f` parameter, the tool automatically looks for existing IoT Edge devices in your IoT Hub and removes them, to avoid errors and keep your hub clean.

The tool could run for several minutes.

5. Copy the generated *top-layer.zip* and *lower-layer.zip* files generated in the previous step to the corresponding top and lower layer virtual machines using `scp`:

```
scp <PATH_TO_CONFIGURATION_BUNDLE>   <USER>@<VM_IP_OR_FQDN>:~
```

6. Connect to the top layer device to install the configuration bundle.

   a. Unzip the configuration bundle. You'll need to install zip first.

   ```
   sudo apt install zip
   unzip ~/<PATH_TO_CONFIGURATION_BUNDLE>/<CONFIGURATION_BUNDLE>.zip #unzip top-layer.zip
   ```

   b. Run `sudo ./install.sh`. Input the IP address or hostname. We recommend using the IP address.

   c. Run `sudo iotedge list` to confirm that all modules are running.

7. Connect to the lower layer device to install the configuration bundle.

   a. Unzip the configuration bundle. You'll need to install zip first.

   ```
   sudo apt install zip
   unzip ~/<PATH_TO_CONFIGURATION_BUNDLE>/<CONFIGURATION_BUNDLE>.zip #unzip lower-layer.zip
   ```

   b. Run `sudo ./install.sh`. Input the device and parent IP addresses or hostnames. We recommend using the IP addresses.

   c. Run `sudo iotedge list` to confirm that all modules are running.

If you didn't specify a deployment file for device configuration, or if deployment problems occur such as an invalid deployment manifest on the top or lower layer device, manually deploy the modules. See the following section.

Manually deploy the connected registry module

Use the following command to deploy the connected registry module manually on an IoT Edge device:

```
az iot edge set-modules \
  --device-id <device-id> \
  --hub-name <hub-name> \
  --content <deployment-manifest-filename>
```

For details, see Deploy Azure IoT Edge modules with Azure CLI.

After successful deployment, the connected registry shows a status of `Online`.

To check the status of the connected registry, use the following az acr connected-registry show command:

```
az acr connected-registry show \
  --registry $REGISTRY_NAME \
  --name $CONNECTED_REGISTRY_RO \
  --output table
```

You might need to a wait a few minutes until the deployment of the connected registry completes.

After successful deployment, the connected registry shows a status of `Online`.

To troubleshoot a deployment, run `iotedge check` on the affected device. For more information, see Troubleshooting.

Next steps

In this quickstart, you learned how to deploy a connected registry to a nested IoT Edge device. Continue to the next guide to learn how to pull images from the newly deployed connected registry.

Pull images from a connected registry

Understand access to a connected registry

8/22/2022 • 3 minutes to read • Edit Online

To access and manage a connected registry, currently only ACR token-based authentication is supported. As shown in the following image, two different types of tokens are used by each connected registry:

- **Client tokens** - One or more tokens that on-premises clients use to authenticate with a connected registry and push or pull images and artifacts to or from it.
- **Sync token** - A token used by each connected registry to access its parent and synchronize content.

> **IMPORTANT**
>
> Store token passwords for each connected registry in a safe location. After they are created, token passwords can't be retrieved. You can regenerate token passwords at any time.

Client tokens

To manage client access to a connected registry, you create tokens scoped for actions on one or more repositories. After creating a token, configure the connected registry to accept the token by using the az acr connected-registry update command. A client can then use the token credentials to access a connected registry endpoint - for example, to use Docker CLI commands to pull or push images to the connected registry.

Your options for configuring client token actions depend on whether the connected registry allows both push and pull operations or functions as a pull-only mirror.

- A connected registry in the default ReadWrite mode allows both pull and push operations, so you can create a token that allows actions to both *read* and *write* repository content in that registry.

- For a connected registry in ReadOnly mode, client tokens can only allow actions to *read* repository content.

Manage client tokens

Update client tokens, passwords, or scope maps as needed by using az acr token and az acr scope-map commands. Client token updates are propagated automatically to the connected registries that accept the token.

Sync token

Each connected registry uses a sync token to authenticate with its immediate parent - which could be another connected registry or the cloud registry. The connected registry automatically uses this token when synchronizing content with the parent or performing other updates.

- The sync token and passwords are generated automatically when you create the connected registry resource. Run the az acr connected-registry install renew-credentials command to regenerate the passwords.
- Include sync token credentials in the configuration used to deploy the connected registry on-premises.
- By default, the sync token is granted permission to synchronize selected repositories with its parent. You must provide an existing sync token or one or more repositories to sync when you create the connected registry resource.
- It also has permissions to read and write synchronization messages on a gateway used to communicate with the connected registry's parent. These messages control the synchronization schedule and manage other updates between the connected registry and its parent.

Manage sync token

Update sync tokens, passwords, or scope maps as needed by using az acr token and az acr scope-map commands. Sync token updates are propagated automatically to the connected registry. Follow the standard practices of rotating passwords when updating the sync token.

> **NOTE**
>
> The sync token cannot be deleted until the connected registry associated with the token is deleted. You can disable a connected registry by setting the status of the sync token to `disabled`.

Registry endpoints

Token credentials for connected registries are scoped to access specific registry endpoints:

- A client token accesses the connected registry's endpoint. The connected registry endpoint is the login server URI, which is typically the IP address of the server or device that hosts it.
- A sync token accesses the endpoint of the parent registry, which is either another connected registry endpoint or the cloud registry itself. When scoped to access the cloud registry, the sync token needs to reach two registry endpoints:
  - The fully qualified login server name, for example, `contoso.azurecr.io`. This endpoint is used for authentication.
  - A fully qualified regional data endpoint for the cloud registry, for example, `contoso.westus2.data.azurecr.io`. This endpoint is used to exchange messages with the connected registry for synchronization purposes.

Next steps

Continue to the following article to learn about specific scenarios where connected registry can be utilized.

Overview: Connected registry and IoT Edge

Pull images from a connected registry on IoT Edge device

8/22/2022 • 2 minutes to read • Edit Online

To pull images from a connected registry, configure a client token and pass the token credentials to access registry content.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Azure Cloud Shell Quickstart - Bash.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
  - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- Connected registry resource in Azure. For deployment steps, see Quickstart: Create a connected registry using the Azure CLI.
- Connected registry instance deployed on an IoT Edge device. For deployment steps, see Quickstart: Deploy a connected registry to an IoT Edge device or Tutorial: Deploy a connected registry to nested IoT Edge devices. In the commands in this article, the connected registry name is stored in the environment variable *$CONNECTED_REGISTRY_RW*.

Create a scope map

Use the az acr scope-map create command to create a scope map for read access to the `hello-world` repository:

```
# Use the REGISTRY_NAME variable in the following Azure CLI commands to identify the registry
REGISTRY_NAME=<container-registry-name>

az acr scope-map create \
  --name hello-world-scopemap \
  --registry $REGISTRY_NAME \
  --repository hello-world content/read \
  --description "Scope map for the connected registry."
```

Create a client token

Use the az acr token create command to create a client token and associate it with the newly created scope map:

```
az acr token create \
  --name myconnectedregistry-client-token \
  --registry $REGISTRY_NAME \
  --scope-map hello-world-scopemap
```

The command will return details about the newly generated token including passwords.

> **IMPORTANT**
> Make sure that you save the generated passwords. Those are one-time passwords and cannot be retrieved. You can generate new passwords using the az acr token credential generate command.

Update the connected registry with the client token

Use the az acr connected-registry update command to update the connected registry with the newly created client token.

```
az acr connected-registry update \
  --name $CONNECTED_REGISTRY_RW \
  --registry $REGISTRY_NAME \
  --add-client-token myconnectedregistry-client-token
```

Pull an image from the connected registry

From a machine with access to the IoT Edge device, use the following example command to sign into the connected registry, using the client token credentials. For best practices to manage login credentials, see the docker login command reference.

Caution

If you set up your connected registry as an insecure registry, update the insecure registries list in the Docker daemon configuration to include the IP address (or FQDN) and port of your connected registry on the IoT Edge device. This configuration should only be used for testing purposes. For more information, see Test an insecure registry.

```
docker login --username myconnectedregistry-client-token \
  --password <token_password> <IP_address_or_FQDN_of_connected_registry>:<port>
```

For IoT Edge scenarios, be sure to include the port used to reach the connected registry on the device. Example:

```
docker login --username myconnectedregistry-client-token \
  --password xxxxxxxxxxx 192.0.2.13:8000
```

Then, use the following command to pull the `hello-world` image:

```
docker pull <IP_address_or_FQDN_of_connected_registry>:<port>/hello-world
```

Next steps

- Learn more about repository-scoped tokens.
- Learn more about accessing a connected registry.

ACR Tasks reference: YAML

8/22/2022 • 19 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, a base image update, or a schedule. 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 config command with the `set` command, which accepts an `defaults.acr=REGISTRY_NAME` key value pair.

For example, to configure the Azure CLI with a default registry named "myregistry":

```
az config set 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 `stepTimeout` 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 `stepTimeout` property provided by the task. The sum of the step timeout values for a task should equal the value of the task's run `timeout` property (for example, set by passing `--timeout` to the `az acr task create` command). If the tasks's run `timeout` value is smaller, it takes priority.	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	`c:\workspace` in Windows or `/workspace` in Linux

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	OVERRIDE SUPPORTED	DEFAULT VALUE
`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.	Yes	None
`secrets`	[secret, secret, ...]	Yes	Array of secret objects.	No	None
`networks`	[network, network, ...]	Yes	Array of network objects.	No	None
`volumes`	[volume, volume, ...]	Yes	Array of volume objects. Specifies volumes with source content to mount to a step.	No	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`

volume

The volume object has the following properties.

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`name`	string	No	The name of the volume to mount. Can contain only alphanumeric characters, '-', and '_'.	None
`secret`	map[string]string	No	Each key of the map is the name of a file created and populated in the volume. Each value is the string version of the secret. Secret values must be Base64 encoded.	None

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.
`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.

STEP TYPE	DESCRIPTION
`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
`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.

PROPERTIES	TYPE	REQUIRED
`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
`volumeMount`	object	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.

PROPERTY	TYPE	REQUIRED
`env`	[string, string, ...]	Optional
`id`	string	Optional
`ignoreErrors`	bool	Optional
`startDelay`	int (seconds)	Optional
`timeout`	int (seconds)	Optional

PROPERTY	TYPE	REQUIRED
`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:

PROPERTY	TYPE	REQUIRED
`detach`	bool	Optional
`disableWorkingDirectoryOverride`	bool	Optional
`entryPoint`	string	Optional
`env`	[string, string, ...]	Optional
`expose`	[string, string, ...]	Optional

PROPERTY	TYPE	REQUIRED
`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
`volumeMount`	object	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: mcr.microsoft.com/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
  ```

Access secret volumes

The `volumes` property allows volumes and their secret contents to be specified for `build` and `cmd` steps in a task. Inside each step, an optional `volumeMounts` property lists the volumes and corresponding container paths to mount into the container at that step. Secrets are provided as files at each volume's mount path.

Execute a task and mount two secrets to a step: one stored in a key vault and one specified on the command

line:

```
az acr run -f mounts-secrets.yaml --set-secret mysecret=abcdefg123456 https://github.com/Azure-Samples/acr-tasks.git
```

```
# This template demonstrates mounting a custom volume into a container at a CMD step
secrets:
  - id: sampleSecret
    keyvault: https://myacbvault2.vault.azure.net/secrets/SampleSecret # Replace with valid keyvault with access

volumes:
  - name: mysecrets
    secret:
      mysecret1: {{.Secrets.sampleSecret | b64enc}}
      mysecret2: {{.Values.mysecret | b64enc}}

steps:
  - cmd: bash cat /run/test/mysecret1 /run/test/mysecret2
    volumeMounts:
      - name: mysecrets
        mountPath: /run/test
```

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

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`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`
`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`

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`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
`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.	`c:\workspace` in Windows or `/workspace` in Linux

volumeMount

The volumeMount object has the following properties.

PROPERTY	TYPE	OPTIONAL	DESCRIPTION	DEFAULT VALUE
`name`	string	No	The name of the volume to mount. Must exactly match the name from a `volumes` property.	None
`mountPath`	string	no	The absolute path to mount files in the container.	None

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 follow 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.SharedVolume

The unique identifier for a shared volume that is accessible by all task steps. The volume is mounted to `c:\workspace` in Windows or `/workspace` in Linux.

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`

ALIAS	RUN VARIABLE
`Architecture`	`Run.Architecture`
`Commit`	`Run.Commit`
`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:0.5`
`az`	`mcr.microsoft.com/acr/azure-cli:7ee1d7f`
`bash`	`mcr.microsoft.com/acr/bash:7ee1d7f`
`curl`	`mcr.microsoft.com/acr/curl:7ee1d7f`

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

8/22/2022 • 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 Container Registry as an Event Grid source

8/22/2022 • 5 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

- Event Grid event schema
- Cloud event schema

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": "Microsoft.ContainerRegistry.ImagePushed",
  "eventTime": "2018-04-25T21:39:47.6549614Z",
  "data": {
    "id": "31c51664-e5bd-416a-a5df-e5206bc47ed0",
    "timestamp": "2018-04-25T21:39:47.276585742Z",
    "action": "push",
    "location": "westus",
    "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\\\\))"
    },
    "connectedRegistry": {
      "name": "edge1"
    }
  },
  "dataVersion": "2.0",
  "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": "Microsoft.ContainerRegistry.ImageDeleted",
  "eventTime": "2018-04-26T17:56:01.8211268Z",
  "data": {
    "id": "f06e3921-301f-42ec-b368-212f7d5354bd",
    "timestamp": "2018-04-26T17:56:00.996603117Z",
    "action": "delete",
    "location": "westus",
    "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"
    },
    "connectedRegistry": {
      "name": "edge1"
    }
  },
  "dataVersion": "2.0",
  "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",
    "location": "westus",
    "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"
    },
    "connectedRegistry": {
      "name": "edge1"
    }
  },
  "dataVersion": "2.0",
  "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",
    "location": "westus",
    "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"
    },
    "connectedRegistry": {
      "name": "edge1"
    }
  },
  "dataVersion": "2.0",
  "metadataVersion": "1"
}]
```

Event properties

- Event Grid event schema
- Cloud event schema

An event has the following top-level data:

PROPERTY	TYPE	DESCRIPTION
`topic`	string	Full resource path to the event source. This field isn't 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.
`location`	string	The location of the event.

PROPERTY	TYPE	DESCRIPTION
`connectedRegistry`	object	The connected registry information if the event is generated by a connected registry.
`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.
`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.

PROPERTY	TYPE	DESCRIPTION
`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.

The connectedRegistry object has the following properties:

PROPERTY	TYPE	DESCRIPTION
`name`	string	The name of the connected registry that generated this event.

Tutorials and how-tos

TITLE	DESCRIPTION
Quickstart: send container registry events	Shows how to use Azure CLI to send Container Registry events.

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

8/22/2022 • 5 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.

For information about running `az acr check-health`, see Check the health of an Azure container registry.

DOCKER_COMMAND_ERROR

This error means that Docker client for CLI could not be found. As a result, the following additional checks aren't 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 couldn't 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 wasn't 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 wasn't 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 couldn't 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.

CMK_ERROR

This error means that the registry can't access the user-assigned or sysem-assigned managed identity used to

configure registry encryption with a customer-managed key. The managed identity might have been deleted.

Potential solution: To resolve the issue and rotate the key using a different managed identity, see steps to troubleshoot the user-assigned identity.

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 or because access to the registry's public endpoint is not allowed. 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.

NOTARY_VERSION_ERROR

This error means that the CLI is not compatible with the currently installed version of Docker/Notary. Try downgrading your notary.exe version to a version earlier than 0.6.0 by replacing your Docker installation's Notary client manually to resolve this issue. You can also try downloading and installing a pre-compiled binary of Notary earlier than 0.6.0 for 64 bit Linux or macOS X from the Notary repository's releases page on GitHub. For windows download the .exe, place it in the(default path: C:\ProgramFiles\Docker\Docker\resources\bin) and rename it to notary.exe.

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.

Azure Policy built-in definitions for Azure Container Registry

8/22/2022 • 5 minutes to read • Edit Online

This page is an index of Azure Policy built-in policy definitions for Azure Container Registry. For additional Azure Policy built-ins for other services, see Azure Policy built-in definitions.

The name of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Version** column to view the source on the Azure Policy GitHub repo.

Azure Container Registry

NAME (AZURE PORTAL)	DESCRIPTION	EFFECT(S)	VERSION (GITHUB)
[Preview]: [Preview]: Container Registry should use a virtual network service endpoint	This policy audits any Container Registry not configured to use a virtual network service endpoint.	Audit, Disabled	1.0.0-preview
Configure container registries to disable anonymous authentication.	Disable anonymous pull for your registry so that data not accessible by unauthenticated user. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Configure container registries to disable ARM audience token authentication.	Disable Azure Active Directory ARM audience tokens for authentication to your registry. Only Azure Container Registry (ACR) audience tokens will be used for authentication. This will ensure only tokens meant for usage on the registry can be used for authentication. Disabling ARM audience tokens does not affect admin user's or scoped access tokens' authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.0
Configure container registries to disable local admin account.	Disable admin account for your registry so that it is not accessible by local admin. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.1
Configure Container registries to disable public network access	Disable public network access for your Container Registry resource so that it's not accessible over the public internet. This can reduce data leakage risks. Learn more at https://aka.ms/acr/portal/public-network and https://aka.ms/acr/private-link.	Modify, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Configure container registries to disable repository scoped access token.	Disable repository scoped access tokens for your registry so that repositories are not accessible by tokens. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Modify, Disabled	1.0.0
Configure Container registries with private endpoints	Private endpoints connect your virtual network to Azure services without a public IP address at the source or destination. By mapping private endpoints to your premium container registry resources, you can reduce data leakage risks. Learn more at: https://aka.ms/privateendpoints and https://aka.ms/acr/private-link.	DeployIfNotExists, Disabled	1.0.0
Container registries should be encrypted with a customer-managed key	Use customer-managed keys to manage the encryption at rest of the contents of your registries. By default, the data is encrypted at rest with service-managed keys, but customer-managed keys are commonly required to meet regulatory compliance standards. Customer-managed keys enable the data to be encrypted with an Azure Key Vault key created and owned by you. You have full control and responsibility for the key lifecycle, including rotation and management. Learn more at https://aka.ms/acr/CMK.	Audit, Deny, Disabled	1.1.2

NAME	DESCRIPTION	EFFECT(S)	VERSION
Container registries should have anonymous authentication disabled.	Disable anonymous pull for your registry so that data is not accessible by unauthenticated user. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.0
Container registries should have ARM audience token authentication disabled.	Disable Azure Active Directory ARM audience tokens for authentication to your registry. Only Azure Container Registry (ACR) audience tokens will be used for authentication. This will ensure only tokens meant for usage on the registry can be used for authentication. Disabling ARM audience tokens does not affect admin user's or scoped access tokens' authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.0
Container registries should have exports disabled	Disabling exports improves security by ensuring data in a registry is accessed solely via the dataplane ('docker pull'). Data cannot be moved out of the registry via 'acr import' or via 'acr transfer'. In order to disable exports, public network access must be disabled. Learn more at: https://aka.ms/acr/export-policy.	Audit, Deny, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Container registries should have local admin account disabled.	Disable admin account for your registry so that it is not accessible by local admin. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.1
Container registries should have repository scoped access token disabled.	Disable repository scoped access tokens for your registry so that repositories are not accessible by tokens. Disabling local authentication methods like admin user, repository scoped access tokens and anonymous pull improves security by ensuring that container registries exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/acr/authentication.	Audit, Deny, Disabled	1.0.0
Container registries should have SKUs that support Private Links	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your container registries instead of the entire service, data leakage risks are reduced. Learn more at: https://aka.ms/acr/private-link.	Audit, Deny, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Container registries should not allow unrestricted network access	Azure container registries by default accept connections over the internet from hosts on any network. To protect your registries from potential threats, allow access from only specific private endpoints, public IP addresses or address ranges. If your registry doesn't have network rules configured, it will appear in the unhealthy resources. Learn more about Container Registry network rules here: https://aka.ms/acr/privatelink, https://aka.ms/acr/portal/public-network and https://aka.ms/acr/vnet.	Audit, Deny, Disabled	2.0.0
Container registries should use private link	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network.By mapping private endpoints to your container registries instead of the entire service, you'll also be protected against data leakage risks. Learn more at: https://aka.ms/acr/private-link.	Audit, Disabled	1.0.1
Container registry images should have vulnerability findings resolved	Container image vulnerability assessment scans your registry for security vulnerabilities and exposes detailed findings for each image. Resolving the vulnerabilities can greatly improve your containers' security posture and protect them from attacks.	AuditIfNotExists, Disabled	2.0.1

NAME	DESCRIPTION	EFFECT(S)	VERSION
Public network access should be disabled for Container registries	Disabling public network access improves security by ensuring that container registries are not exposed on the public internet. Creating private endpoints can limit exposure of container registry resources. Learn more at: https://aka.ms/acr/portal/public-network and https://aka.ms/acr/private-link.	Audit, Deny, Disabled	1.0.0

Next steps

- See guidance to assign policies and review compliance.
- See the built-ins on the Azure Policy GitHub repo.
- Review the Azure Policy definition structure.
- Review Understanding policy effects.

Monitoring Azure Container Registry data reference

8/22/2022 • 3 minutes to read • Edit Online

See Monitor Azure Container Registry for details on collecting and analyzing monitoring data for Azure Container Registry.

Metrics

Container Registry metrics

Resource Provider and Type: Microsoft.ContainerRegistry/registries

METRIC	EXPORTABLE VIA DIAGNOSTIC SETTINGS?	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
AgentPoolCPUTime	Yes	Seconds	Total	CPU time used by ACR tasks running on dedicated agent pools	None
RunDuration	Yes	Milliseconds	Total	Duration of ACR tasks runs	None
StorageUsed	No	Bytes	Average	Storage used by the container registry Sum of storage for unique and shared layers, manifest files, and replica copies in all repositories[1]	Geolocation
SuccessfulPullCount	Yes	Count	Total	Successful pulls of container images and other artifacts from the registry	None
SuccessfulPushCount	Yes	Count	Total	Successful pushes of container images and other artifacts to the registry	None
TotalPullCount	Yes	Count	Total	Total pulls of container images and other artifacts from the registry	None

METRIC	EXPORTABLE VIA DIAGNOSTIC SETTINGS?	UNIT	AGGREGATION TYPE	DESCRIPTION	DIMENSIONS
TotalPushCount	Yes	Count	Total	Total pushes of container images and other artifacts to the registry	None

[1]Because of layer sharing, registry storage used may be less than the sum of storage for individual repositories. When you delete a repository or tag, you recover only the storage used by manifest files and the unique layers referenced.

For more information, see a list of all platform metrics supported in Azure Monitor.

Metric Dimensions

For more information on what metric dimensions are, see Multi-dimensional metrics.

Azure Container Registry has the following dimensions associated with its metrics.

DIMENSION NAME	DESCRIPTION
Geolocation	The Azure region for a registry or geo-replica.

Resource logs

This section lists the types of resource logs you can collect for Azure Container Registry.

For reference, see a list of all resource logs category types supported in Azure Monitor.

Container Registries

Resource Provider and Type: Microsoft.ContainerRegistry/registries

CATEGORY	DISPLAY NAME	DETAILS
ContainerRegistryLoginEvents	Login Events	Registry authentication events and status, including the incoming identity and IP address
ContainerRegistryRepositoryEvents	Repository Events	Operations on images and other artifacts in registry repositories The following operations are logged: push, pull, untag, delete (including repository delete), purge tag, and purge manifest[1]

[1]Purge events are logged only if a registry retention policy is configured.

Azure Monitor Logs tables

This section refers to all of the Azure Monitor Logs Kusto tables relevant to Azure Container Registry and available for query by Log Analytics.

Container Registry

TABLE	DESCRIPTION
AzureActivity	Entries from the Azure Activity log that provide insight into any subscription-level or management group level events that have occurred in Azure.
AzureMetrics	Metric data emitted by Azure services that measure their health and performance.
ContainerRegistryLoginEvents	Azure Container Registry Login Auditing Logs
ContainerRegistryRepositoryEvents	Azure Container Registry Repository Auditing Logs

For a reference of all Azure Monitor Logs / Log Analytics tables, see the Azure Monitor Log Table Reference.

Activity log

The following table lists operations related to Azure Container Registry that may be created in the Activity log. This list is not exhaustive.

OPERATION	DESCRIPTION
Create or Update Container Registry	Create a container registry or update a registry property
Delete Container Registry	Delete a container registry
List Container Registry Login Credentials	Show credentials for registry's admin account
Import Image	Import an image or other artifact to a registry
Create Role Assignment	Assign an identity an RBAC role to access a resource

Schemas

The following schemas are in use by Azure Container Registry's resource logs.

SCHEMA	DESCRIPTION
ContainerRegistryLoginEvents	Schema for registry authentication events and status, including the incoming identity and IP address
ContainerRegistryRepositoryEvents	Schema for operations on images and other artifacts in registry repositories

Next steps

- See Monitor Azure Container Registry for a description of monitoring an Azure container registry.
- See Monitoring Azure resources with Azure Monitor for details on monitoring Azure resources.

Troubleshoot registry login

8/22/2022 • 5 minutes to read • Edit Online

This article helps you troubleshoot problems you might encounter when logging into an Azure container registry.

Symptoms

May include one or more of the following:

- Unable to login to registry using `docker login`, `az acr login`, or both
- Unable to login to registry and you receive error `unauthorized: authentication required` or `unauthorized: Application not registered with AAD`
- Unable to login to registry and you receive Azure CLI error `Could not connect to the registry login server`
- Unable to push or pull images and you receive Docker error `unauthorized: authentication required`
- Unable to access a registry using `az acr login` and you receive error
  ```
  CONNECTIVITY_REFRESH_TOKEN_ERROR. Access to registry was denied. Response code: 403. Unable to get admin user credentials with message: Admin user is disabled. Unable to authenticate using AAD or admin login credentials.
  ```
- Unable to access registry from Azure Kubernetes Service, Azure DevOps, or another Azure service
- Unable to access registry and you receive error `Error response from daemon: login attempt failed with status: 403 Forbidden` - See Troubleshoot network issues with registry
- Unable to access or view registry settings in Azure portal or manage registry using the Azure CLI

Causes

- Docker isn't configured properly in your environment - solution
- The registry doesn't exist or the name is incorrect - solution
- The registry credentials aren't valid - solution
- The registry public access is disabled. Public network access rules on the registry prevent access - solution
- The credentials aren't authorized for push, pull, or Azure Resource Manager operations - solution
- The credentials are expired - solution

Further diagnosis

Run the az acr check-health command to get more information about the health of the registry environment and optionally access to a target registry. For example, diagnose Docker configuration errors or Azure Active Directory login problems.

See Check the health of an Azure container registry for command examples. If errors are reported, review the error reference and the following sections for recommended solutions.

Follow the instructions from the AKS support doc if you fail to pull images from ACR to the AKS cluster.

> **NOTE**
>
> Some authentication or authorization errors can also occur if there are firewall or network configurations that prevent registry access. See Troubleshoot network issues with registry.

Potential solutions

Check Docker configuration

Most Azure Container Registry authentication flows require a local Docker installation so you can authenticate with your registry for operations such as pushing and pulling images. Confirm that the Docker CLI client and daemon (Docker Engine) are running in your environment. You need Docker client version 18.03 or later.

Related links:

- Authentication overview
- Container registry FAQ

Specify correct registry name

When using `docker login`, provide the full login server name of the registry, such as *myregistry.azurecr.io*. Ensure that you use only lowercase letters. Example:

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

When using az acr login with an Azure Active Directory identity, first sign into the Azure CLI, and then specify the Azure resource name of the registry. The resource name is the name provided when the registry was created, such as *myregistry* (without a domain suffix). Example:

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

Related links:

- az acr login succeeds but docker fails with error: unauthorized: authentication required

Confirm credentials to access registry

Check the validity of the credentials you use for your scenario, or were provided to you by a registry owner. Some possible issues:

- If using an Active Directory service principal, ensure you use the correct credentials in the Active Directory tenant:
  - User name - service principal application ID (also called *client ID*)
  - Password - service principal password (also called *client secret*)
- If using an Azure service such as Azure Kubernetes Service or Azure DevOps to access the registry, confirm the registry configuration for your service.
- If you ran `az acr login` with the `--expose-token` option, which enables registry login without using the Docker daemon, ensure that you authenticate with the username `00000000-0000-0000-0000-000000000000`.
- If your registry is configured for anonymous pull access, existing Docker credentials stored from a previous Docker login can prevent anonymous access. Run `docker logout` before attempting an anonymous pull operation on the registry.

Related links:

- Authentication overview
- Individual login with Azure AD
- Login with service principal
- Login with managed identity
- Login with repository-scoped token
- Login with admin account

- Azure AD authentication and authorization error codes
- az acr login reference

Confirm credentials are authorized to access registry

Confirm the registry permissions that are associated with the credentials, such as the `AcrPull` Azure role to pull images from the registry, or the `AcrPush` role to push images.

Access to a registry in the portal or registry management using the Azure CLI requires at least the `Reader` role or equivalent permissions to perform Azure Resource Manager operations.

If your permissions recently changed to allow registry access though the portal, you might need to try an incognito or private session in your browser to avoid any stale browser cache or cookies.

You or a registry owner must have sufficient privileges in the subscription to add or remove role assignments.

Related links:

- Azure roles and permissions - Azure Container Registry
- Login with repository-scoped token
- Add or remove Azure role assignments using the Azure portal
- Use the portal to create an Azure AD application and service principal that can access resources
- Create a new application secret
- Azure AD authentication and authorization codes

Check that credentials aren't expired

Tokens and Active Directory credentials may expire after defined periods, preventing registry access. To enable access, credentials might need to be reset or regenerated.

- If using an individual AD identity, a managed identity, or service principal for registry login, the AD token expires after 3 hours. Log in again to the registry.
- If using an AD service principal with an expired client secret, a subscription owner or account administrator needs to reset credentials or generate a new service principal.
- If using a repository-scoped token, a registry owner might need to reset a password or generate a new token.

Related links:

- Reset service principal credentials
- Regenerate token passwords
- Individual login with Azure AD

Advanced troubleshooting

If collection of resource logs is enabled in the registry, review the ContainerRegistryLoginEvents log. This log stores authentication events and status, including the incoming identity and IP address. Query the log for registry authentication failures.

Related links:

- Logs for diagnostic evaluation and auditing
- Container registry FAQ
- Best practices for Azure Container Registry

Next steps

If you don't resolve your problem here, see the following options.

- Other registry troubleshooting topics include:
  - Troubleshoot network issues with registry
  - Troubleshoot registry performance
- Community support options
- Microsoft Q&A
- Open a support ticket - based on information you provide, a quick diagnostic might be run for authentication failures in your registry

Troubleshoot network issues with registry

8/22/2022 • 5 minutes to read • Edit Online

This article helps you troubleshoot problems you might encounter when accessing an Azure container registry in a virtual network or behind a firewall or proxy server.

Symptoms

May include one or more of the following:

- Unable to push or pull images and you receive error `dial tcp: lookup myregistry.azurecr.io`
- Unable to push or pull images and you receive error `Client.Timeout exceeded while awaiting headers`
- Unable to push or pull images and you receive Azure CLI error `Could not connect to the registry login server`
- Unable to pull images from registry to Azure Kubernetes Service or another Azure service
- Unable to access a registry behind an HTTPS proxy and you receive error `Error response from daemon: login attempt failed with status: 403 Forbidden` or `Error response from daemon: Get <registry>: proxyconnect tcp: EOF Login failed`
- Unable to configure virtual network settings and you receive error `Failed to save firewall and virtual network settings for container registry`
- Unable to access or view registry settings in Azure portal or manage registry using the Azure CLI
- Unable to add or modify virtual network settings or public access rules
- ACR Tasks is unable to push or pull images
- Microsoft Defender for Cloud can't scan images in registry, or scan results don't appear in Microsoft Defender for Cloud
- You receive error `host is not reachable` when attempting to access a registry configured with a private endpoint.

Causes

- A client firewall or proxy prevents access - solution
- Public network access rules on the registry prevent access - solution
- Virtual network or private endpoint configuration prevents access - solution
- You attempt to integrate Microsoft Defender for Cloud or certain other Azure services with a registry that has a private endpoint, service endpoint, or public IP access rules - solution

Further diagnosis

Run the az acr check-health command to get more information about the health of the registry environment and optionally access to a target registry. For example, diagnose certain network connectivity or configuration problems.

See Check the health of an Azure container registry for command examples. If errors are reported, review the error reference and the following sections for recommended solutions.

If you're experiencing problems using an Azure Kubernetes Service with an integrated registry, run the az aks check-acr command to validate that the AKS cluster can reach the registry.

> **NOTE**
> Some network connectivity symptoms can also occur when there are issues with registry authentication or authorization. See Troubleshoot registry login.

Potential solutions

Configure client firewall access

To access a registry from behind a client firewall or proxy server, configure firewall rules to access the registry's public REST and data endpoints. If dedicated data endpoints are enabled, you need rules to access:

- REST endpoint: `<registryname>.azurecr.io`
- Data endpoint(s): `<registry-name>.<region>.data.azurecr.io`

For a geo-replicated registry, configure access to the data endpoint for each regional replica.

Behind an HTTPS proxy, ensure that both your Docker client and Docker daemon are configured for proxy behavior. If you change your proxy settings for the Docker daemon, be sure to restart the daemon.

Registry resource logs in the ContainerRegistryLoginEvents table may help diagnose an attempted connection that is blocked.

Related links:

- Configure rules to access an Azure container registry behind a firewall
- HTTP/HTTPS proxy configuration
- Geo-replicationin Azure Container Registry
- Monitor Azure Container Registry

Configure public access to registry

If accessing a registry over the internet, confirm the registry allows public network access from your client. By default, an Azure container registry allows access to the public registry endpoints from all networks. A registry can limit access to selected networks, or selected IP addresses.

If the registry is configured for a virtual network with a service endpoint, disabling public network access also disables access over the service endpoint. If your registry is configured for a virtual network with Private Link, IP network rules don't apply to the registry's private endpoints.

Related links:

- Configure public IP network rules
- Connect privately to an Azure container registry using Azure Private Link
- Restrict access to a container registry using a service endpoint in an Azure virtual network

Configure VNet access

Confirm that the virtual network is configured with either a private endpoint for Private Link or a service endpoint (preview). Currently an Azure Bastion endpoint isn't supported.

If a private endpoint is configured, confirm that DNS resolves the registry's public FQDN such as *myregistry.azurecr.io* to the registry's private IP address.

- Run the az acr check-health command with the `--vnet` parameter to confirm the DNS routing to the private endpoint in the virtual network.
- Use a network utility such as `dig` or `nslookup` for DNS lookup.
- Ensure that DNS records are configured for the registry FQDN and for each of the data endpoint FQDNs.

Review NSG rules and service tags used to limit traffic from other resources in the network to the registry.

If a service endpoint to the registry is configured, confirm that a network rule is added to the registry that allows access from that network subnet. The service endpoint only supports access from virtual machines and AKS clusters in the network.

If you want to restrict registry access using a virtual network in a different Azure subscription, ensure that you register the `Microsoft.ContainerRegistry` resource provider in that subscription. Register the resource provider for Azure Container Registry using the Azure portal, Azure CLI, or other Azure tools.

If Azure Firewall or a similar solution is configured in the network, check that egress traffic from other resources such as an AKS cluster is enabled to reach the registry endpoints.

Related links:

- Connect privately to an Azure container registry using Azure Private Link
- Troubleshoot Azure Private Endpoint connectivity problems
- Restrict access to a container registry using a service endpoint in an Azure virtual network
- Required outbound network rules and FQDNs for AKS clusters
- Kubernetes: Debugging DNS resolution
- Virtual network service tags

Configure service access

Currently, access to a container registry with network restrictions isn't allowed from several Azure services:

- Microsoft Defender for Cloud can't perform image vulnerability scanning in a registry that restricts access to private endpoints, selected subnets, or IP addresses.
- Resources of certain Azure services are unable to access a container registry with network restrictions, including Azure App Service and Azure Container Instances.

If access or integration of these Azure services with your container registry is required, remove the network restriction. For example, remove the registry's private endpoints, or remove or modify the registry's public access rules.

Starting January 2021, you can configure a network-restricted registry to allow access from select trusted services.

Related links:

- Azure Container Registry image scanning by Microsoft Defender for container registries
- Provide feedback
- Allow trusted services to securely access a network-restricted container registry

Advanced troubleshooting

If collection of resource logs is enabled in the registry, review the ContainterRegistryLoginEvents log. This log stores authentication events and status, including the incoming identity and IP address. Query the log for registry authentication failures.

Related links:

- Logs for diagnostic evaluation and auditing
- Container registry FAQ
- Azure Security Baseline for Azure Container Registry
- Best practices for Azure Container Registry

Next steps

If you don't resolve your problem here, see the following options.

- Other registry troubleshooting topics include:
  - Troubleshoot registry login
  - Troubleshoot registry performance
- Community support options
- Microsoft Q&A
- Open a support ticket

Troubleshoot registry performance

8/22/2022 • 4 minutes to read • Edit Online

This article helps you troubleshoot problems you might encounter with the performance of an Azure container registry.

Symptoms

May include one or more of the following:

- Pull or push images with the Docker CLI takes longer than expected
- Deployment of images to a service such as Azure Kubernetes Service takes longer than expected
- You're not able to complete a large number of concurrent pull or push operations in the expected time
- You see an HTTP 429 error similar to `Too many requests`
- Pull or push operations in a geo-replicated registry take longer than expected, or push fails with error `Error writing blob` or `Error writing manifest`

Causes

- Your network connection speed may slow registry operations - solution
- Image layer compression or extraction may be slow on the client - solution
- You're reaching a configured limit in your registry service tier or environment - solution
- Your geo-replicated registry has replicas in nearby regions - solution
- You're pulling from a geographically distant registry replica - solution

If you don't resolve your problem here, see Advanced troubleshooting and Next steps for other options.

Potential solutions

Check expected network speed

Check your internet upload and download speed, or use a tool such as AzureSpeed to test upload and download from Azure blob storage, which hosts registry image layers.

Check your image size against the maximum supported size and the supported download or upload bandwidth for your registry service tier. If your registry is in the Basic or Standard tier, consider upgrading to improve performance.

For image deployment to other services, check the regions where the registry and target are located. Consider locating the registry and the deployment target in the same or network-close regions to improve performance.

Related links:

- Azure Container Registry service tiers
- Container registry FAQ
- Performance and scalability targets for Azure Blob Storage

Check client hardware

The disk type and CPU on the docker client can affect the speed of extracting or compressing image layers on the client as part of pull or push operations. For example, layer extraction on a hard disk drive will take longer than on a solid-state disk. Compare pull operations for comparable images from your Azure container registry

and a public registry such as Docker Hub.

Review configured limits

If you're concurrently pushing or pulling multiple or many multi-layered images to your registry, review the supported ReadOps and WriteOps limits for the registry service tier. If your registry is in the Basic or Standard tier, consider upgrading to increase the limits. Check also with your networking provider about network throttling that may occur with many concurrent operations.

Review your Docker daemon configuration for the maximum concurrent uploads or downloads for each push or pull operation on the client. Configure higher limits if needed.

Because each image layer requires a separate registry read or write operation, check the number of layers in your images. Consider strategies to reduce the number of image layers.

Related links:

- Azure Container Registry service tiers
- dockerd

Configure geo-replicated registry

A Docker client that pushes an image to a geo-replicated registry might not push all image layers and its manifest to a single replicated region. This situation 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.

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.

To troubleshoot operations with a geo-replicated registry, you can also temporarily disable Traffic Manager routing to one or more replications.

Related links:

- Geo-replication in Azure Container Registry

Configure DNS for geo-replicated registry

If pull operations from a geo-replicated registry appear slow, the DNS configuration on the client might resolve to a geographically distant DNS server. In this case, Traffic Manager might be routing requests to a replica that is network-close to the DNS server but distant from the client. Run a tool such as `nslookup` or `dig` (on Linux) to determine the replica that Traffic Manager routes registry requests to. For example:

```
nslookup myregistry.azurecr.io
```

A potential solution is to configure a closer DNS server.

Related links:

- Geo-replication in Azure Container Registry
- Troubleshoot push operations with geo-replicated registries
- Temporarily disable routing to replication
- Traffic Manager FAQs

Advanced troubleshooting

If your permissions to registry resources allow, check the health of the registry environment. If errors are

reported, review the error reference for potential solutions.

If collection of resource logs is enabled in the registry, review the ContainterRegistryRepositoryEvents log. This log stores information for operations such as push or pull events. Query the log for repository-level operation failures.

Related links:

- Logs for diagnostic evaluation and auditing
- Container registry FAQ
- Best practices for Azure Container Registry

Next steps

If you don't resolve your problem here, see the following options.

- Other registry troubleshooting topics include:
  - Troubleshoot registry login
  - Troubleshoot network issues with registry
- Community support options
- Microsoft Q&A
- Open a support ticket

Check the health of an Azure container registry

8/22/2022 • 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.

For additional registry troubleshooting guidance, see:

- Troubleshoot registry login
- Troubleshoot network issues with registry
- Troubleshoot registry performance

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
```

Check registry access in a virtual network

To verify DNS settings to route to a private endpoint, pass the virtual network's name or resource ID. The resource ID is required when the virtual network is in a different subscription or resource group than the registry.

```
az acr check-health --name myregistry --vnet myvnet
```

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; skip confirmation to pull image
az acr check-health --name myregistry --ignore-errors --yes
```

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
```

Check if registry is configured with quarantine

Once you enable a container registry to be quarantined, every image you publish to this repository will be quarantined. Any attempts to access or pull quarantined images will fail with an error. For more information, See pull the quarantine image.

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.

Support policies for Azure Container Registry(ACR)

8/22/2022 • 2 minutes to read • Edit Online

This article provides details about Azure Container Registry (ACR) support policies, supported features, and limitations.

Features supported by Azure Container Registry

- Connect to ACR using Azure private link
- Push and pull Helm charts to ACR
- Encrypt using Customer managed keys
- Enable Content trust
- Scan Images using Azure Security Center
- ACR Tasks
- Import container images to ACR
- Image locking in ACR
- Synchronize content with ACR using Connected Registry
- Geo replication in ACR

Microsoft/ACR can't extend support

- Any local network issues that interrupt the connection to ACR service.
- Vulnerabilities or issues caused by running third-party container images using ACR Tasks.
- Vulnerabilities or bugs with images in the ACR customer store.

Microsoft/ACR extends support

- General queries about the supported features of ACR.
- Unable to pull image due to authentication errors, image size, and client-side issues with container runtime.
- Unable to push an image to ACR due to authentication errors, image size, and client-side issues with container runtime.
- Unable to add VNET/Subnet to ACR Firewall across subscription.
- Issues with slow push/pull operations due to client, network, or ACR.
- Issues with integration of ACR with Azure Kubernetes Service (AKS) or with any other Azure service.
- Authentication issues in ACR, authentication errors in integration, repository-based access role(RBAC).

Shared responsibility

- Issues with slow push/pull operations caused by a slow-performing client VM, network, or ACR. Here the customers have to provide the time range, image name, and configuration settings.
- Issues with Integration of ACR with any other Azure service. Here the customers have to provide the details of the client used to build and pull the image and push it to ACR. For example, the customer uses the DevOps pipeline to build the image and push it to ACR.

Customers have to self support

- Microsoft/ACR can't make any changes if there's a detection of base image vulnerability in the security center

(Microsoft Defender for Cloud). Customers can reach out for guidance

- Microsoft/ACR can't make any changes with Dockerfile. Customers have to identify and review it from their end.

ACR SUPPORT	LINK
Create a support ticket	https://aka.ms/acr/support/create-ticket
Service updates and releases	ACR Blog
Roadmap	https://aka.ms/acr/roadmap
FAQ	https://aka.ms/acr/faq
Audit Logs	https://aka.ms/acr/audit-logs
Health-Check-CLI	https://aka.ms/acr/health-check
ACR Links	https://aka.ms/acr/links

API and SDK reference

- SDK for Python
- SDK for .NET
- REST API Reference

Upstream bugs

The ACR support will identify the root cause of every issue raised. The team will report all the identified bugs as an issue in the ACR repository with supporting details. The engineering team will review and provide a workaround solution, bug fix, or upgrade with a new release timeline. All the bug fixes integrate from upstream. Customers can watch the issues, bug fixes, add more details, and follow the new releases.