Introduction to Azure Container Registry

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 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.

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, such as Azure Kubernetes Service (AKS), App Service, Batch, and Service Fabric.

Configure Azure Container Registry 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 container images in parallel in the cloud.

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 tier supports webhook integration, registry authentication with Microsoft Entra ID, 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.

Security and access

You log in to a registry by using the Azure CLI or the standard docker login command. Azure Container Registry transfers container images over HTTPS, and it supports TLS to help secure client connections.

As of January 13, 2020, Azure Container Registry requires 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).

You control access to a container registry by using an Azure identity, a Microsoft Entra service principal, or a provided admin account. Use Azure role-based access control (RBAC) to assign specific registry permissions to users or systems.

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 you push an image to a registry.*

Supported images and artifacts

When images are 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

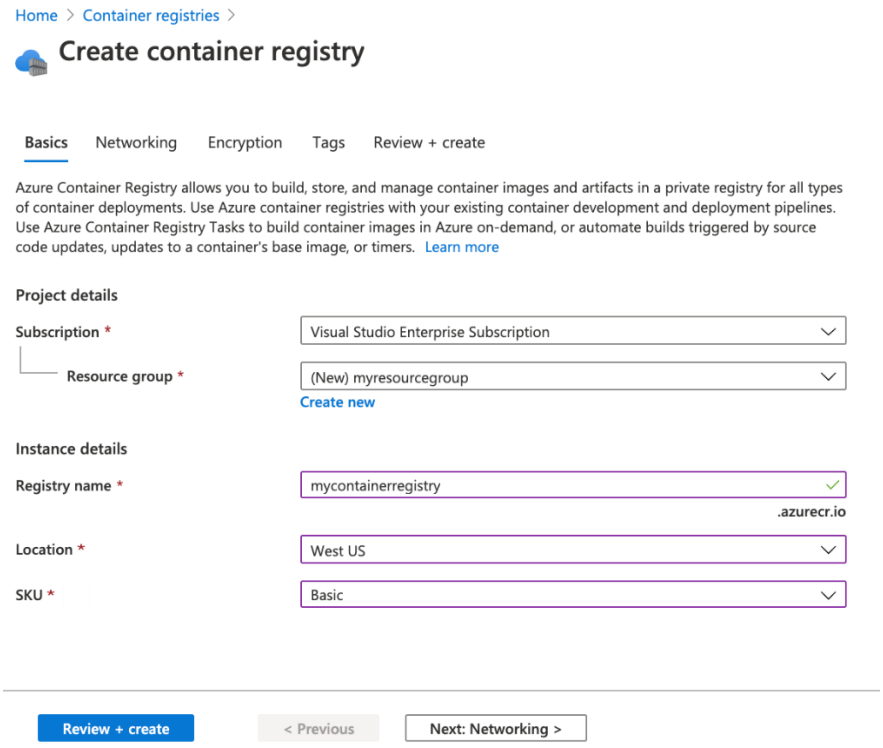
Use Azure Container Registry tasks to streamline building, testing, pushing, and deploying images in Azure. For example, use Azure Container Registry tasks to extend your development inner loop to the cloud by offloading docker build operations to Azure. Configure build tasks to automate your container 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 build and push operations for container images. They can also define the execution of one or more containers, in which each step uses a container as its execution environment.

1.1 Create an Azure container registry using the Azure portal

Azure Container Registry is a private registry service for building, storing, and managing container images and related artifacts.

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



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.

Azure CLI

az acr login --name <registry-name>

Example:
Azure CLI

az acr login --name mycontainerregistry

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
docker push myregistry.azurecr.io/samples/nginx

docker pull myregistry.azurecr.io/samples/nginx

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

1.2 Use the Azure Container Registry client libraries

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*

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*.

Install the package

dotnet add package Azure.Containers.ContainerRegistry --prerelease

Authenticate the client

```
// 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
    });
```

List repositories asynchronously

AsyncPageable - A collection of values that may take multiple service requests to iterate over.

```
// 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

```
// 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();
}}
```

2.1 Azure Container Registry roles and permissions

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.

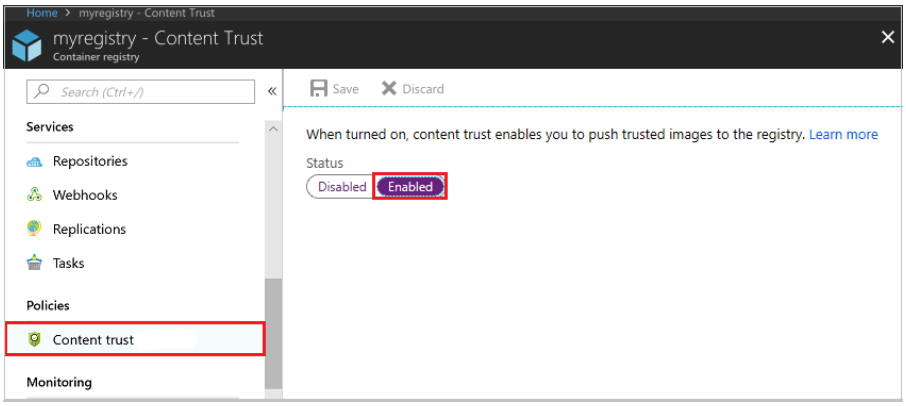
Role/Permission	Access Resource Manager	Create/delete registry	Push image	Pull image	Delete image data	Change policies	Sign images
Owner	X	X	X	X	X	X	
Contributor	X	X	X	X	X	X	
Reader	X			X			
AcrPush			X	X			
AcrPull				X			
AcrDelete					X		
AcrImageSigner							X

2.2 Content trust in Azure Container Registry

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.

Content trust is a feature of the Premium service tier of Azure Container Registry.

Azure Container Registry (ACR) does not support acr import to import images signed with Docker Content Trust (DCT). By design, the signatures are not visible after the import, and the notary v2 stores these signatures as artifacts.



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 Microsoft Entra identities the AcrImageSigner role.

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.

3.1 Run containerized tasks with restart policies

The ease and speed of deploying containers in Azure Container Instances provides a compelling platform for executing run-once tasks like build, test, and image rendering in a container instance.

With a configurable restart policy, you can specify that your containers are stopped when their processes complete. Because Azure bills container instances by the second, you're only charged for the compute resources used while the container executing your task is running.

Container restart policy

When you create a container group in Azure Container Instances, you can specify one of three restart policy settings.

Restart policy	Description
Always	Containers in the container group are always restarted. This policy is the **default** setting applied when no restart policy is specified at container creation.
Never	Containers in the container group are never restarted. The containers run at most once.
OnFailure	Containers in the container group are restarted only when the process executed in the container fails (when it terminates with a nonzero exit code). The containers are run at least once.

Specify a restart policy

How you specify a restart policy depends on how you create your container instances, such as with the Azure CLI, Azure PowerShell cmdlets, or in the Azure portal. In the Azure CLI, specify the --restart-policy parameter when you call az container create.

```
az container create \
   --resource-group myResourceGroup \
   --name mycontainer \
   --image mycontainerimage \
   --restart-policy OnFailure
```

4.1 Quickstart: Deploy a container instance in Azure using the Azure CLI

Use Azure Container Instances to run serverless Docker containers in Azure with simplicity and speed. Deploy an application to a container instance on-demand when you don't need a full container orchestration platform like Azure Kubernetes Service.

Create a resource group

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

Create a container

```
az container create --resource-group myResourceGroup --name mycontainer --image
mcr.microsoft.com/azuredocs/aci-helloworld --dns-name-label aci-demo --ports 80
```

Within a few seconds, you should get a response from the Azure CLI indicating the deployment completed. Check its status with the az container show command:

```
az container show --resource-group myResourceGroup --name mycontainer --query
"{FQDN:ipAddress.fqdn,ProvisioningState:provisioningState}" --out table
```

When you run the command, the container's fully qualified domain name (FQDN) and its provisioning state are displayed.

```
  OutputCopy

  FQDN                          ProvisioningState

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

aci-demo.eastus.azurecontainer.io  Succeeded
```

If the container's ProvisioningState is **Succeeded**, go to its FQDN in your browser. If you see a web page similar to the following, congratulations! You successfully deployed an application running in a Docker container to Azure.

Pull the container logs

When you need to troubleshoot a container or the application it runs (or just see its output), start by viewing the container instance's logs.

az container logs --resource-group myResourceGroup --name mycontainer

Attach output streams

In addition to viewing the logs, you can attach your local standard out and standard error streams to that of the container.

```
az container attach --resource-group myResourceGroup --name mycontainer
```

Clean up resources

When you're done with the container, remove it using the az container delete command:

az container delete --resource-group myResourceGroup --name mycontainer

To verify that the container deleted, execute the az container list command:

az container list --resource-group myResourceGroup --output table

4.1 Authenticate with an Azure container registry

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 a Microsoft Entra service principal

Authentication options

The following table lists available authentication methods and typical scenarios. See linked content for details.

Expand table

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
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

Create an unmanaged ingress controller

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services.

Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. When you use an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

To **create a basic NGINX ingress controller** without customizing the defaults, you'll use **Helm.**

To control image versions, you'll want to import them into your own Azure Container Registry.
The NGINX ingress controller Helm chart relies on three container images. Use az acr import to import those images into your ACR.

Check the load balancer service

Check the load balancer service by using kubectl get services.

```
kubectl get services --namespace ingress-basic -o wide -w ingress-nginx-controller
```

When the Kubernetes load balancer service is created for the NGINX ingress controller, an IP address is assigned under *EXTERNAL-IP*, as shown in the following example output:

```
NAME                       TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)                      AGE   SELECTOR
ingress-nginx-controller   LoadBalancer   10.0.65.205   EXTERNAL-IP   80:30957/TCP,443:32414/TCP   1m    app.kubernetes.io/component=controller,app.kubernetes.io/instance=ingress-nginx,app.kubernetes.io/name=ingress-nginx
```

If you browse to the external IP address at this stage, you see a 404 page displayed. This is because you still need to set up the connection to the external IP, which is done in the next sections.

Create an ingress route

Both applications are now running on your Kubernetes cluster. To route traffic to each application, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

Test the ingress controller

To test the routes for the ingress controller, browse to the two applications. Open a web browser to the IP address of your NGINX ingress controller, such as *EXTERNAL_IP*. The first demo application is displayed in the web browser, as shown in the following example: