**Azure Container Registry**

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

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.

**Use cases**

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

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

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

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

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

**Key features**

* **Registry SKUs** - Create one or more container registries in your Azure subscription. Registries are available in three SKUs: Basic, Standard, and Premium, each of which supports webhook integration, registry authentication with Azure Active Directory, and delete functionality. Take advantage of local, network-close storage of your container images by creating a registry in the same Azure location as your deployments. Use the geo-replication feature of Premium registries for advanced replication and container image distribution scenarios.

You control access to a container registry using an Azure identity, an Azure Active Directory-backed service principal, or a provided admin account. Log in to the registry using the Azure CLI or the standard docker login command.

* **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.
* **Azure Container Registry Tasks** - 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.

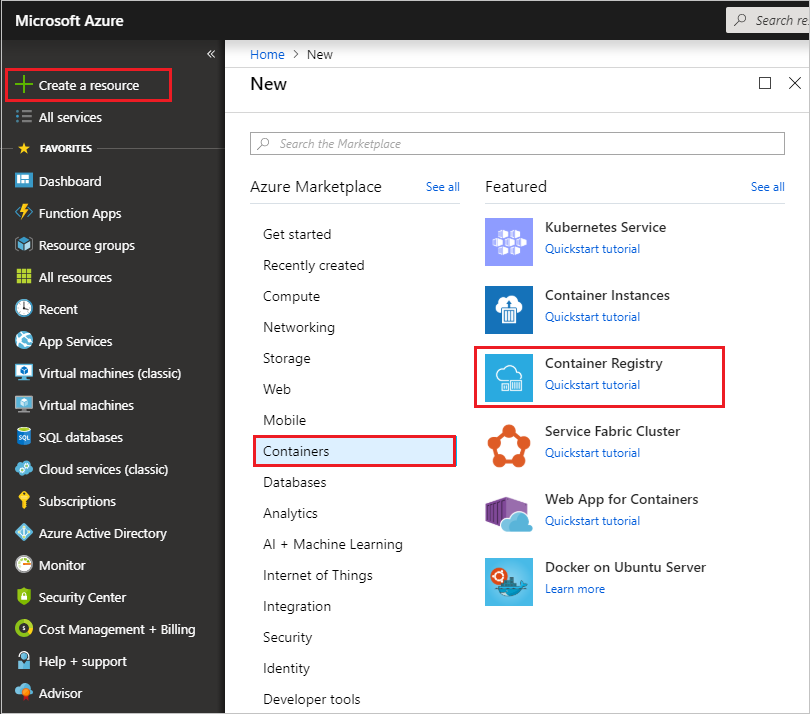
Multi-step tasks provide step-based task definition and execution for building, testing, and patching container images in the cloud. Task steps define individual container image build and push operations. They can also define the execution of one or more containers, with each step using the container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

**For multi-step task you can refer following link:**

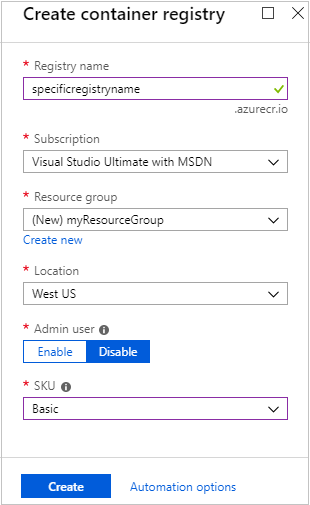
https://docs.microsoft.com/en-us/azure/container-registry/container-registry-tasks-overview#multi-step-tasks

**Create Container Registry using Azure Portal**

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



Enter values for **Registry name** and **Resource group**. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters.

**Azure Container Registry SKUs**

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

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

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

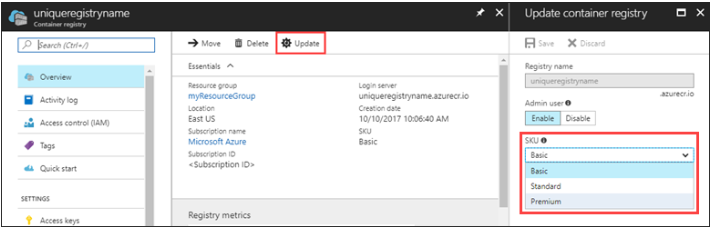
|  |  |
| --- | --- |
| **SKU** | **Description** |
| **Basic** | A cost-optimized entry point for developers learning about Azure Container Registry. Basic registries have the same programmatic capabilities as Standard and Premium (such as Azure Active Directory authentication integration, image deletion,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

**Changing SKUs**

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

**Azure portal**

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



**Azure CLI to update SKU:**

az acr update --name myregistry --sku Premium

**Creating Container Registry using Azure CLI using PowerShell**

**#Login**

az login

**#Setting up variables**

$resourceGroup="myRg2"

$acrName="myacr2905"

$location="eastus"

**#Creating resourceGroup**

az group create --name $resourceGroup --location $location

**#Creating ACR**

**#Setting up admin-enabled true so that at a later point we can extract user id and password**

az acr create --name $acrName --resource-group $resourceGroup --sku Basic --admin-enabled true

**#Login to ACR**

az acr login --name $acrName

**#Query login server and store it to a variable so that it can be use in next steps**

$loginServer=az acr show --name $acrName --query loginServer --output tsv

**#In order to confirm the loginServer you can just print it by following command**

$loginServer

**#Get the credentials and storing in variable, this is why we enabled the admin flag while creating ACR**

$password= (az acr credential show -n $acrName --query "passwords[0].value" -o tsv)

**#Listing repository, currently it is empty as we just created it**

az acr repository list --name $acrName -o table

**Creating a docker image for sample java console hello world application**

1. Create a maven based Java project and create a java class to print hello world (test java project can be found in java samples folder)
2. Now generate a jar file with test.jar name
3. Create a folder call “docker” copy the test.jar in this folder
4. Create a file with name “Dockerfile” remember this file do not have any extension.

**Sample Dockerfile**

|  |  |
| --- | --- |
| FROM java:8 | #Use java 8 as a base image for our image to be created |
| WORKDIR /home | #Specifies the working directory within container |
| ENV FILE\_PATH=message.txt | #Environment variable (this is a sample to just to show) |
| ADD test.jar test.jar | #Add the test.jar file with name test.jar in working directory |
| VOLUME /home | #Mount the volume at /home |
| EXPOSE 8080 | #Port number to export |
| CMD java -jar test.jar | #Command to execute |
|  |  |

**VIMP: Leave a blank line at the end of this file otherwise you may get errors, it is to indicate the EOF.**

**Creating a image using docker (Assumption you already have docker installed and running)**

1. Now open command prompt/terminal and navigate to folder where Dockerfile is created and test.jar file is copied.
2. Now run the below command (notice . at the end of the command)

**docker build .**

this will give you the output like below

Sending build context to Docker daemon 7.168kB

Step 1/7 : FROM java:8

---> d23bdf5b1b1b

Step 2/7 : WORKDIR /home

---> Running in e4858846fe90

Removing intermediate container e4858846fe90

---> 28e0df9c4c26

Step 3/7 : ENV FILE\_PATH=message.txt

---> Running in 4c0f98651839

Removing intermediate container 4c0f98651839

---> c6862dd22783

Step 4/7 : ADD test.jar test.jar

---> 3997c049bf68

Step 5/7 : VOLUME /home

---> Running in 5c2b7a9f8cef

Removing intermediate container 5c2b7a9f8cef

---> 90ecde26261f

Step 6/7 : EXPOSE 8080

---> Running in c4abda3c2e56

Removing intermediate container c4abda3c2e56

---> 7c8f2102f535

Step 7/7 : CMD java -jar test.jar

---> Running in d058f33b6554

Removing intermediate container d058f33b6554

---> 54162a81c4f2

Successfully built **54162a81c4f2**

The bold string in above code output is your newly created image id.

1. Now run the below command

docker image ls

this will give you output like below

REPOSITORY TAG IMAGE ID CREATED SIZE

<none> <none> 54162a81c4f2 About a minute ago 643MB

Note that the image id in step 3 and image id in above output is same. Now we need to tag this image to push to ACR that we created

To tag this run the below command (source image id, repository:tag)

**docker tag 54162a81c4f2 hello-world:v1**

In order to check if the above image is created properly and working fine we will run it locally by using docker run command as given below.

**docker run -it --name mycname hello-world:v1**

it= interactive terminal mode mycname= name of the container to be created and started, then name of image and tag to be used to create the container.

Now It will print the hello world message to the console. It shows that our image is working as expected.

Now in order to push this image to ACR we have to retag it using the below command

**docker tag hello-world:v1 $loginServer/hello-world:v1**

Now to push this image we will use the docker push command as given below:

**docker push $loginServer/hello-world:v1**

This will take some time to push the image to ACR depending upon size of the image and your speed of internet connection.

**Running Containers on Azure Container Instances**

Azure Container Instances offers the fastest and simplest way to run a container in Azure, without having to manage any virtual machines and without having to adopt a higher-level service.

Azure Container Instances is a great solution for any scenario that can operate in isolated containers, including simple applications, task automation, and build jobs.

Containers offer significant startup benefits over virtual machines (VMs). Azure Container Instances can start containers in Azure in seconds, without the need to provision and manage VMs.

**Public IP connectivity and DNS name**

Azure Container Instances enables exposing your containers directly to the internet with an IP address and a fully qualified domain name (FQDN). When you create a container instance, you can specify a custom DNS name label so your application is reachable at *customlabel*.*azureregion*.azurecontainer.io.

**Hypervisor-level security**

Azure Container Instances guarantees your application is as isolated in a container as it would be in a VM.

**Custom sizes**

Containers are typically optimized to run just a single application, but the exact needs of those applications can differ greatly. Azure Container Instances provides optimum utilization by allowing exact specifications of CPU cores and memory. You pay based on what you need and get billed by the second, so you can fine-tune your spending based on actual need.

**Persistent storage**

By default, Azure Container Instances are stateless. If the container crashes or stops, all of its state is lost. To persist state beyond the lifetime of the container, you must mount a volume from an external store.

**Linux and Windows containers**

Azure Container Instances can schedule both Windows and Linux containers with the same API. Simply specify the OS type when you create your container groups.

**Some features are currently restricted to Linux containers:**

* Multiple containers per container group
* Volume mounting (Azure Files, emptyDir, GitRepo, secret)
* Resource usage metrics with Azure Monitor
* Virtual network deployment (preview)
* GPU resources (preview)

**Co-scheduled groups**

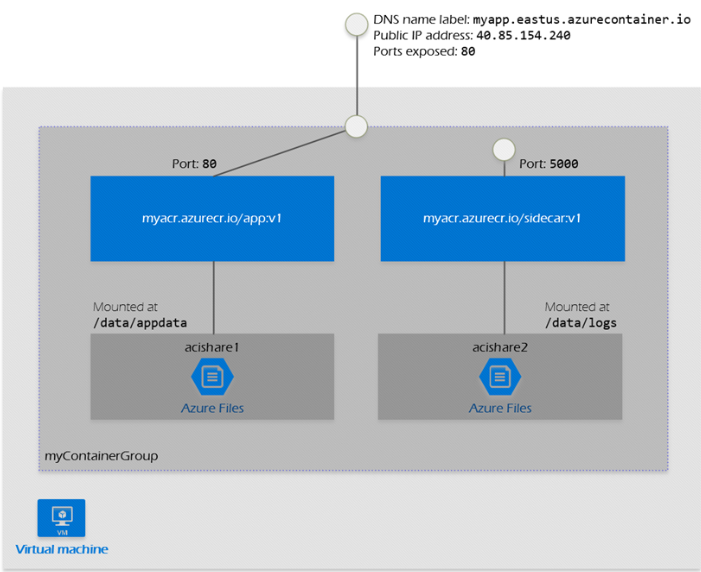
Azure Container Instances supports scheduling of multi-container groups that share a host machine, local network, storage, and lifecycle. This enables you to combine your main application container with other supporting role containers, such as logging sidecars.

**Container groups in Azure Container Instances**

The top-level resource in Azure Container Instances is the container group.

A container group is a collection of containers that get scheduled on the same host machine. The containers in a container group share a lifecycle, resources, local network, and storage volumes. It's similar in concept to a pod in Kubernetes.

The following diagram shows an example of a container group that includes multiple containers:



**This example container group:**

* Is scheduled on a single host machine.
* Is assigned a DNS name label.
* Exposes a single public IP address, with one exposed port.
* Consists of two containers. One container listens on port 80, while the other listens on port 5000.
* Includes two Azure file shares as volume mounts, and each container mounts one of the shares locally.

**Minimum and maximum allocation**

* Allocate a **minimum** of 1 CPU and 1 GB of memory to a container group. Individual container instances within a group can be provisioned with less than 1 CPU and 1 GB of memory.

**Networking**

Container groups share an IP address and a port namespace on that IP address. To enable external clients to reach a container within the group, you must expose the port on the IP address and from the container. Because containers within the group share a port namespace, port mapping isn't supported. Containers within a group can reach each other via localhost on the ports that they have exposed, even if those ports aren't exposed externally on the group's IP address.

**Common scenarios**

Multi-container groups are useful in cases where you want to divide a single functional task into a small number of container images. These images can then be delivered by different teams and have separate resource requirements.

**Example usage could include:**

* A container serving a web application and a container pulling the latest content from source control.
* An application container and a logging container. The logging container collects the logs and metrics output by the main application and writes them to long-term storage.
* An application container and a monitoring container. The monitoring container periodically makes a request to the application to ensure that it's running and responding correctly, and raises an alert if it's not.
* A front-end container and a back-end container. The front end might serve a web application, with the back end running a service to retrieve data.

**Creating Container Groups with the Azure CLI**

azcontainer create

-n mycontainergroup-g myresourcegroup

--image someimage:sometag

--ip-address public

--dns-name-label mysite # mysite.eastus.azurecontainer.io

--ports 80

--os-type Windows # default is Linux

--cpu1 –memory 1.5

-e name=value

--restart-policy never # always, onfailure

--azure-file-volume… # credentials, mount path, share-name

**ACI creation using image from public repository i.e. from docker hub.**

az container create --name mycontainergroup `

--resource-group $resourceGroup `

--image neerajraja2001/sampleapp:v2 `

--ip-address public `

--dns-name-label mydnssitelabel `

--ports 80 `

--cpu 1 --memory 1 `

--restart-policy never

**Getting container logs**

az container logs --resource-group $resourceGroup --name mycontainergroup

**Mounting path on Azure Storage account with file share**

* We have to create an storage account
  1. $storageAccount="mystorageacc2509"
  2. az storage account create -g $resourceGroup -n $storageAccount --sku Standard\_LRS
  3. Getting the connection string and storing it in variable
  4. $connectionString= az storage account show-connection-string -n $storageAccount -g $resourceGroup --query connectionString -o tsv
  5. Setting up and environment variable to mount volume
  6. $env:AZURE\_STORAGE\_CONNECTION\_STRING =$connectionString
  7. create a share that will make use of the environment variable that we just created
  8. $acishare="acishare"
  9. az storage share create -n $acishare
  10. get the storage account key to for authentication
  11. $storageKey = $(az storage account keys list -g $resourceGroup --account-name $storageAccount --query "[0].value" --output tsv)

**Creating container with volume mount (Need to investigate more)**

az container create -g $resourceGroup --name "mycontainergroup" `

--image neerajraja2001/sampleapp:v2 `

--cpu 1 --memory 1 `

--azure-file-volume-account-name $storageAccount `

--azure-file-volume-account-key $storageKey `

--azure-file-volume-share-name $acishare `

--azure-file-volume-mount-path "/home" `

-e FILE\_PATH=/home/message.txt `

--dns-name-label "aciacr1" --ports 80

If you wanted to use ACR then you have to add following two more parameters which we discussed in ACR section

--registry-username myacr2905

--registry-password your password

Executing command in container

az container exec -n mycontainergroup -g $resourceGroup --exec-command sh #opens the shell

echo "hello" /home/message.txt

show files that are created or in our storage share

az storage file list -s acishare o- table