Table of Contents

[Lab Guide: Module 4-1 - Advanced Docker Topics 2](#_Toc23970146)

[Exercise 1: Working with the Private Docker Registry 2](#_Toc23970147)

[Exercise 2: Working with Data Volumes 4](#_Toc23970148)

[Exercise 3: Docker Compose 10](#_Toc23970149)

Lab Guide: Module 4-1 - Advanced Docker Topics

Exercise 1: Working with the Private Docker Registry

In this Exercise, you will show how to perform following two tasks:

* Create a Private Registry using Azure Container Registry
* Push a Custom Image to Private Registry

Tasks

1. Create Private Registry using Azure Container Registry Service

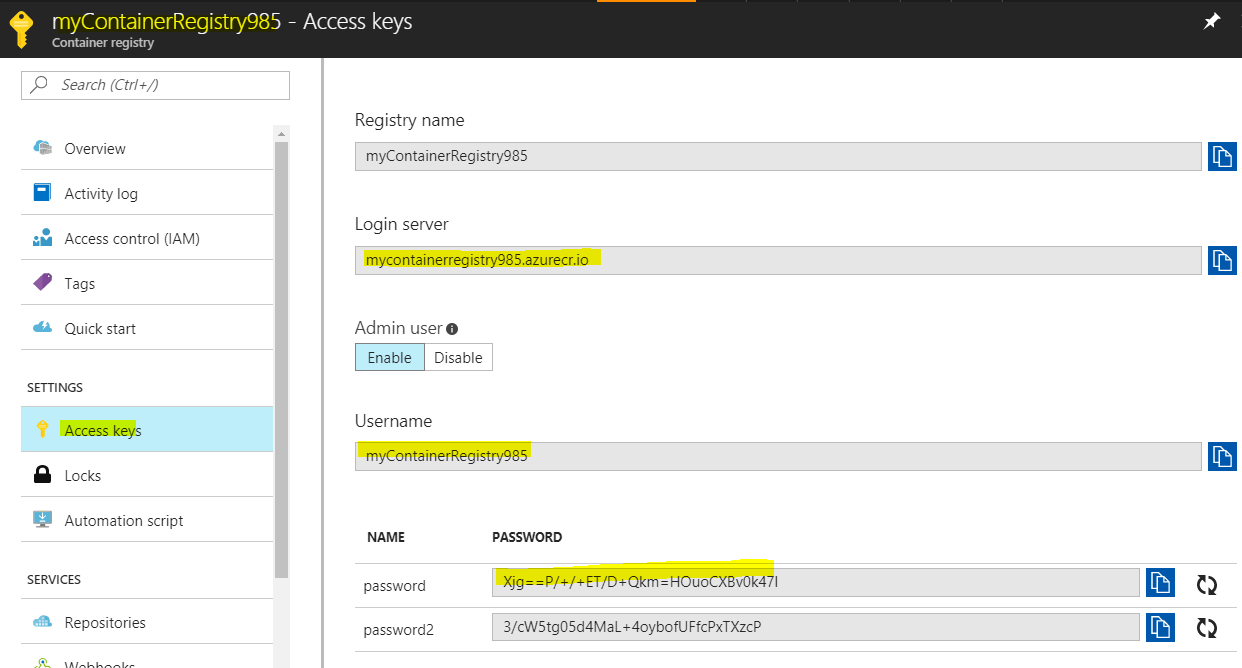
One of the critical components in Container DevOps lifecycle is container registry. Container registry allows you to store and manage your container images. You can use public registries such as docker hub, private registries installed in your on-premise environment or provided by cloud services. In this task, you are going to use Azure Container Registry to manage your Linux or Windows container images.

1. In Azure Portal, click plus button to add a new resource. Type **Azure Container Registry**in the search textbox and click on the result.

1. Click **Create**

1. Select a registry name which should be. You may select the resource group you have used in previous labs to keep resources together. Make sure to enable **Admin user** so that build agents can login to ACR and push container images successfully.

1. On the Azure portal, go to your registry. Hit Access keys. Take note of the following highlighted fields so you can use them to login to your registry in command line.



1. In command line type the following and replace the capitalized variables with your login server, username, and password:

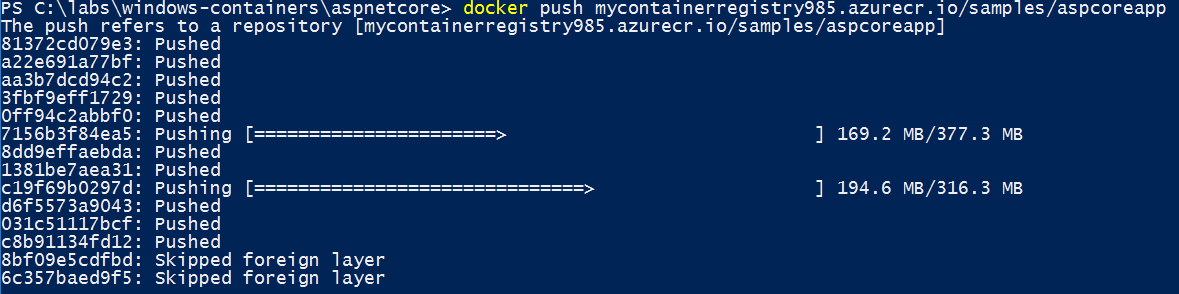
**docker login MYREGISTRY.azurecr.io -u MYUSERNAME -p MYPASSWORD**  


1. You should see “Login succeeded”
2. Tag your aspcoreapp image (you can choose any other custom image for this Exercisenstration that you have build previously) with your registry name. Change MYREGISTRY to your registry name.

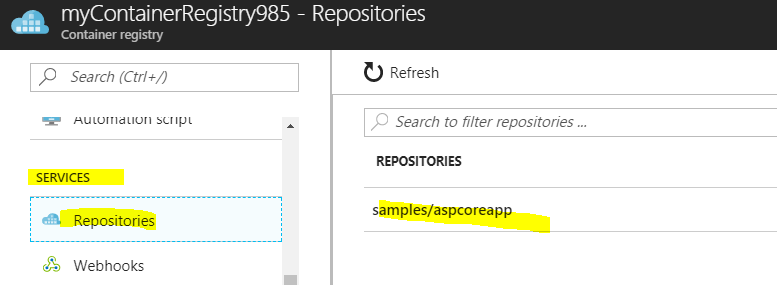
**docker tag bash:latest waelscontainerregistry.azurecr.io/bash**  


1. See your new image with “**docker images**”
2. Use the following command to push to your registry (change MYREGISTRY to the name of your registry):

**docker push waelscontainerregistry.azurecr.io/bash**



1. To view that it has been pushed once your command line shows completed go back to the portal. Under **Services** click on **Repositories**. You should see the following:



1. If you want to pull from your registry you can run this command:

**docker pull waelscontainerregistry.azurecr.io/bash**

Exercise 2: Working with Data Volumes

In this Exercise, you will show how to mount a host directory as a data volume. The host directory will be available inside the container along with all the files (and sub directories). Later you will update a file through a data volume from within the container. Remember that by default, data volumes at the time of mounting are read/write (unless you choose to only enable them for read only access).

Tasks

1. Mount a host directory as a data volume
2. From the host, go to C:\ and create a new directory by running the command “**mkdir c:\MyData**”

A close up of a sign

Description automatically generated

1. Run “ls myData” to show the empty directory
2. Run a container in interactive mode and mount the host directory as a data volume. Run the command. Note that we start a command prompt (not powershell) in a nanoserver container

**docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd**

A close up of a logo

Description automatically generated

*NOTE: Notice the –v switch that is required to mount the host directory C:\MyData inside the container as C:\Data. This will result in container access to contents of C:\MyData on the host inside the container as C:\Data. You can choose same name for the directory inside the container and host but it’s not mandatory as you see in the above command (C:\MyData on the host and C:\Data inside the container)*

1. On the container Console first check the hostname by running the command “hostname”. The actual hostname for your container may be different.

A close up of a sign

Description automatically generated

1. List the directories by running the command “dir”. Notice the *data* directory as part of the listing.

A blue and white sign

Description automatically generated

1. Create a file in the folder and add more text to it. Run the command:

**echo File is updated by container: %COMPUTERNAME% >> c:\data\file.txt**



Note that %COMPUTERNAME% is that same as hostname

1. Look at content inside the file.txt by running the command

**more c:\data\file.txt**

A black sign with white text

Description automatically generated

Note that we need to use more and not gc because we are not in a powershell console.

1. Exit the container session by typing **exit**
2. On the host PowerShell Console run the command “**gc C:\MyData\file.txt**”. Notice that file creation made from the container persist on the host by the file.txt.

A close up of a sign

Description automatically generated

1. To gather more information about container and volumes that has been mounted you can run the command “**docker inspect CONTAINER ID**”. Replace the

CONTAINER ID by the hostname of the container that you have capture in previous step.



1. The Docker Inspect command outputs a rather large JSON file on the display. You may need to scroll down to find the section labeled “Mounts”. Notice that c:\mydata is the source and c:\data is the destination. Also, RW refers to Read/Write.

A screenshot of a cell phone

Description automatically generated

1. Let's run another container in interactive mode and mount the host directory as a data volume. Run the command

**docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd**

A close up of a logo

Description automatically generated

1. Look at content inside the file.txt by running the command

**more c:\data\file.txt**

A black sign with white text

Description automatically generated

1. Add more text to it. Run the command:

**echo File is updated by container: %COMPUTERNAME% >> c:\data\file.txt**

1. On the host machine, go to **C:\MyData** from the file explorer and open **file.txt**

A screenshot of a cell phone

Description automatically generated

1. Update the content of the file with notepad and save it.

A screenshot of text

Description automatically generated

*Note: the two different \*\*hostnames\*\* correspond to the two Ids of the containers that wrote in the file.*

1. Go back to the Powershell windows and check that the container can see the host changes with the command “**more c:\data\file.txt**”

*A black sign with white text

Description automatically generated*

*Note that because of concurrency challenges, you would probably not have multiple containers and hosts writing in the same file. The purpose of this exercise was only to show how we can persistent data across containers beyond their short lifecycle.*

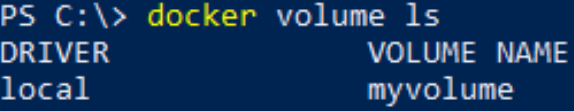
1. Finally, you can run `exit` to stop the running containers
2. Mount a shared-storage volume as a data volume

In this task, you will Exercisenstrate how to create and use a shared-storage volume. To keep the Exercisenstration accessible and easy to follow, you will use the *local* driver which uses local host for the storage. However, the exact same concepts will work against production ready storage drivers like Convoy and others. For more information on the Convoy volume plugin, please visit: <https://github.com/rancher/convoy>

1. First you will create a volume by running the command “**docker volume create -d local myvolume**”



1. You can list all the volumes by running the command “**docker volume ls**”. Notice that myvolume is available as a local driver.



1. You can use docker inspect command with the volumes too. Run the command **docker inspect myvolume**

A screenshot of a cell phone

Description automatically generated

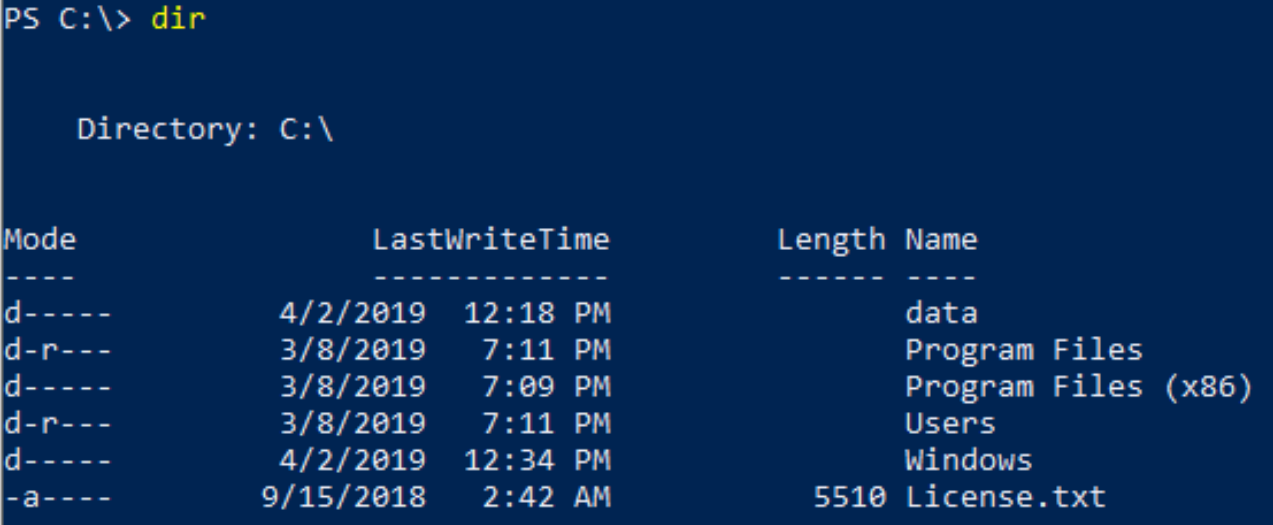
*Notice that Mountpoint is set to location on C drive under ProgramData\docker folder. This is the default location for local storage drivers. If you have used another commercial storage driver, the location may be different.*

1. To launch a container and make that storage volume available inside the container run the command (without quotes)

**docker run -it -v myvolume:C:/Data/ mcr.microsoft.com/windows/servercore:1809 powershell**

This command is like the command from last section where you shared the host directory, except that within the –v switch you are using the name of storage volume rather than path to host directory. Everything else remain the same.

1. On the PowerShell command prompt inside the container run the command “dir” to list the directories available on the container.



1. Notice the data directory. You can now add/remove files to it. Let’s create a new text file and add text content to it. On the command prompt run the command (make sure you have full line below with quotes)

"**File created on the host $(hostname)" >> c:\data\sample.txt**  
  

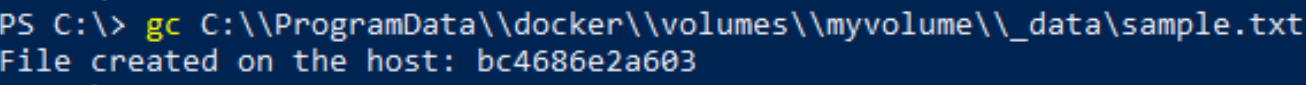

1. Confirm that file sample.txt has been created successfully by running the command **more c:\data\sample.txt**

A close up of a sign

Description automatically generated

1. Now exit the container by running the command “exit”. This will take you back to PowerShell Console on the host.
2. To check the content of sample.txt file from the host run the command

**gc C:\\ProgramData\\docker\\volumes\\myvolume\\\_data\sample.txt**



Exercise 3: Docker Compose

In this Exercise, you will work with a simple “Famous Quotes” micro service that has a Web App with UX that talks to a RESTful API to fetch “Quotes” in a JSON format. Both the Web App and API are developed using ASP.NET Core and each will run in a separate container. As this is a multi- container scenario, you will deal with two challenges which are both addressed using the docker-compose tool:

* + - * How can the Web API be accessed by the Web App without the need to hardcore its FQDN or IP Address? Instead of hardcoding IP Address (or FQDN) you can use docker-compose.yml file to make these services discoverable.
      * Express specific dependencies, such as the Web Api container needs to start before Web App.
      * Bring both applications up and running in separate containers with a single command (i.e., without using individual “docker-run” commands for each container).

Tasks

1. Running Multi-Container Applications using Docker Compose
2. Launch the PowerShell Console (if not already running) and change your current directory to “compose” folder by running the command “cd C:\Module 4-1\compose”

A close up of a sign

Description automatically generated

1. Before proceeding further let’s stop and remove all the running containers from previous task. Run the command “docker rm (docker ps -aq) -f”
2. First, look at directory structure by running the command “dir”.

A blue and white sign

Description automatically generated

1. Notice that you have two folders “mywebapi” and “mywebapp” representing the web API and web application respectively. First, you will inspect the piece of code that is making the RESTful call to mywebapi. To do that run the command: “gc .\mywebapp\Controllers\HomeController.cs”



1. This displays the code within HomeController.cs file. You may need to scroll down to view the code that calls the mywebapi RESTful endpoint. The actual Uri is [http://Exercisewebapi:9000/api/quotes](http://demowebapi:9000/api/quotes). Notice the use of “Exercisewebapi” which is not a FQDN nor IP Address, but rather a service that is defined within the docker-compose.yml file (which we will review next). By using the service name, the web application can simply refer to the Web API app (using that same name) across all environments, including development, test and production etc.



1. Let’s inspect the docker-compose.yml file. Run the command “gc .\docker-compose.yml”



This will emit the content of docker-compose.yml file.

A screenshot of a cell phone

Description automatically generated

First, notice the structure of the file. All .YML files follow the YAML structure (more information about the extension can be found at : <https://www.reviversoft.com/file-extensions/yml)>. For docker compose usage you first define the version number and then specify the structure of your services. In this case, we have two services, namely “*Exercisewebapp*” and “*Exercisewebapi*”. The Exercisewebapp service declaration starts with the build instruction and points to folder “*mywebapp*” that contains the ASP.NET core application and relevant Dockerfile (recall the file entitled, DockerFile, that resides in the root of the application). Note how the compose file contains sections, or “instructions”: Services, networks, etc. The build instruction is equal to the *docker build* command. Then ports are mapped from the host’s port 80 to the container’s port 80. The *depends\_on* directs the docker-compose to launch the *Exercisewebapi* container first since *Exercisewebapp* depends on it. Also, the discoverability is done by using the service names (as mentioned in the paragraph above, *Exercisewebapp* can access *Exercisewebapi* by its service name, rather than FQDN or IP Address).

Next is the *Exercisewebapi* service declaration. It also starts with the build command pointing to the “mywebapi” folder that contains the Dockerfile and relevant ASP.NET Core files. Ports are mapped from host port 9000 to container port 9000.

Finally, networks section keeps the default settings to nat networking. This network declaration is needed for windows containers at this time. Basically, it tells docker compose to use default nat networking.

1. **Docker Compose Up**
2. At this point, you are all set to run the multi-container application with a single command “**docker-compose.exe up -d**”



NOTE: The docker-compose.exe tries to make it simple to start and stop the services (running containers) with commands like up and down. The “-d” switch works the same as when used with the docker build command, which instructs docker to run the container in the background rather than interactively. If you don’t provide any switch parameter, the default is set to interactive.

As the command executes, you will notice that the “mywebapi” container is built first. This is because we mention in the yml file that “mywebapp” depends on it, so it will build first. Also, if the image for “mywebapi” already exists, then it won’t be built again.

A screenshot of a cell phone

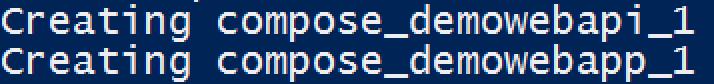
Description automatically generated

Next, Docker will build the container image for “mywebapp.”

A screenshot of a cell phone

Description automatically generated

* *NOTE: You can safely ignore any warnings.*
* Finally, docker-compose will run both containers using the instructions from the docker-compose.yml file.



1. You can check details about running containers by executing the command “**docker ps**”.

A screenshot of a cell phone

Description automatically generated

1. Let's test both the Web Application and Web API. First get the IP address of the web app by doing a docker inspect and putting in the Container ID of the web app:

**docker inspect <container id> | FINDSTR "IPAddress**

A close up of a sign

Description automatically generated

1. Open a browser and go to the IP address found by the previous command.

*NOTE:* You can find the IP address of the virtual machine by going to the VM blade in the Azure Portal. In the upper-right hand corner of the screen, you will see a value for the Public IP address.

A screenshot of a social media post

Description automatically generated

To test that web api you will can do it in two ways. First using the browser (already opened in previous step) select the “Quotes” option from the top menu bar. This will result in a call to web api and results being displayed on the web application.

A screenshot of a cell phone

Description automatically generated

1. Another way to test the web api directly is by using PowerShell to fetch the JSON from the Web API RESTful endpoint. You can run the command

“**curl** [**http://ip\_from\_backend\_container:9000/api/quotes**](http://ip_from_backend_container:9000/api/quotes)”. Notice the response is 200 OK and containing the actual JSON content.

1. **Docker Compose Down**

When you wish to stop and remove the multi-container application that was launched by the docker compose, you will use docker-compose down command. The down command safely stops and remove all the containers that were launched by the up command earlier using the docker-compose.yml file.

***NOTE****: If you only wish to stop the multi-container applications and associated running containers use “docker-compose stop” command instead. However, this command won’t remove the containers.*

1. On the PowerShell Console run the command “**docker-compose.exe down**”. Notice first the containers are stopped and then removed.
2. You have now completed all the tasks in this Exercise.