Table of Contents

Exercise Guide: Module 3 - Advanced Docker Topics 2

Exercise 1: Working with the Private Docker Registry 2

Exercise 2: Working with Data Volumes 4

Exercise 3: Docker Compose 10

Exercise 4: Docker Networking 16

Exercise Guide: Module 3 - 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**

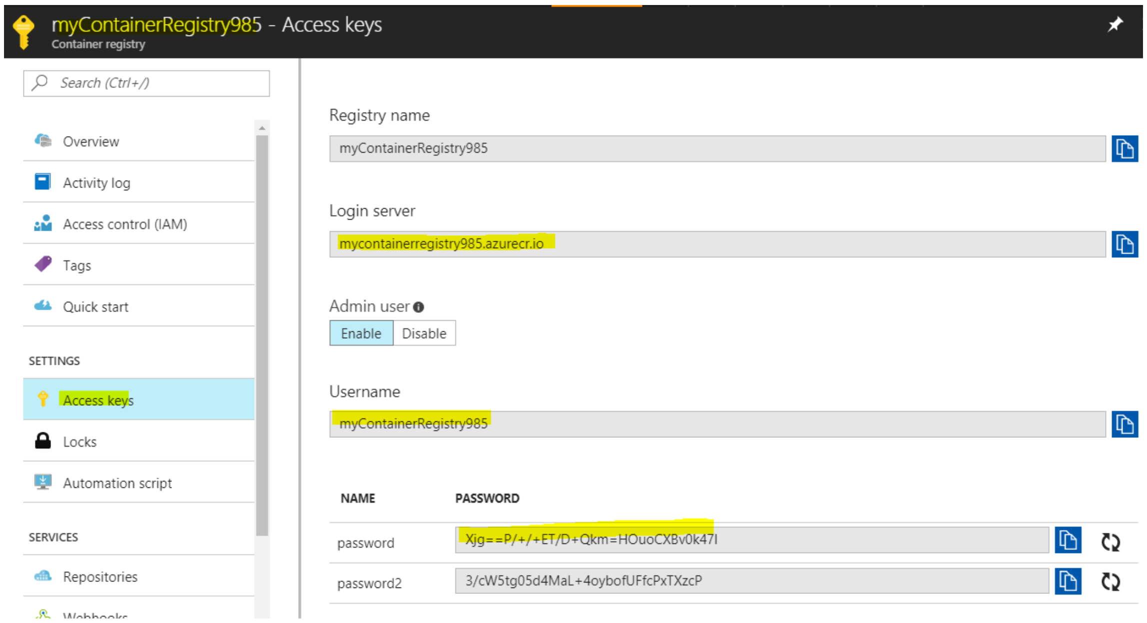
   One of the critical components in Container DevOps lifecycle is container registry. Container registry allows you to store and manage your container images. You can use public registries such as docker hub, private registries installed in your on-premise environment or provided by cloud services. In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

   2. Click **Create**

   3. 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.

   4. 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.



Registry name: myContainerRegistry985

Login server: mycontainerregistry985.azurecr.io

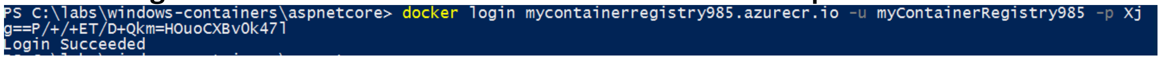
Admin user: [✓ Enable] Disable

Username: myContainerRegistry985

NAME	PASSWORD
password	Xjg==P/+/+ET/D+Qkm=HOuoCXBv0k47l
password2	3/cW5tg05d4MaL+4oybofUFfcPxTXzcP

5. 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`

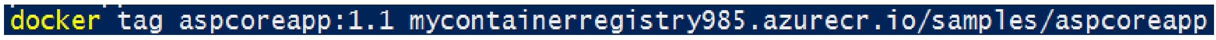


```
PS C:\labs\windows-containers\aspnetcore> docker login mycontainerregistry985.azurecr.io -u myContainerRegistry985 -p Xj
g==P/+/+ET/D+Qkm=HOuoCXBv0k47l
Login Succeeded
```

6. You should see “Login succeeded”

7. Tag your aspcoreapp image (you can choose any other custom image for this demonstration that you have build previously) with your registry name. Change MYREGISTRY to your registry name.

`docker tag bash:latest MYREGISTRY.azurecr.io/bash`

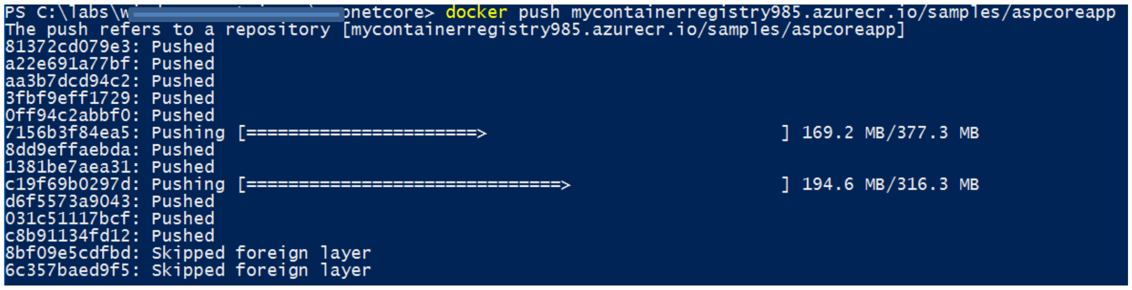


```
docker tag aspcoreapp:1.1 mycontainerregistry985.azurecr.io/samples/aspcoreapp
```

8. See your new image with “**docker images**”

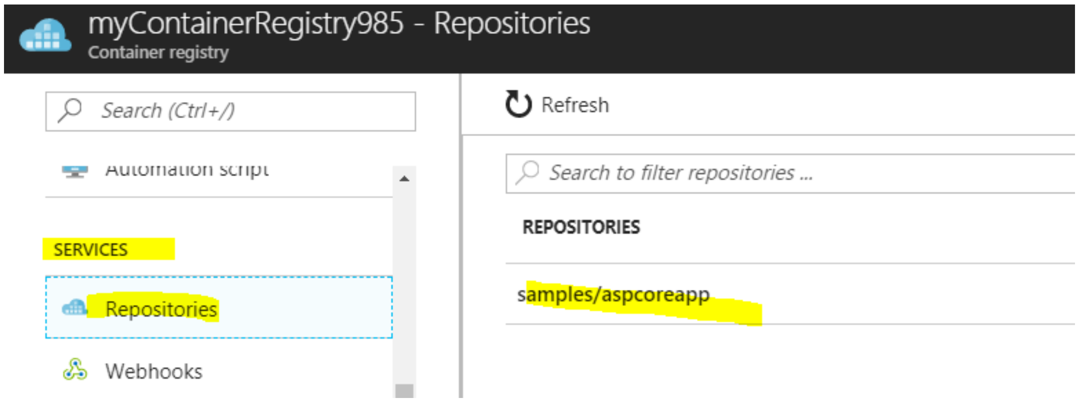
9. Use the following command to push to your registry (change MYREGISTRY to the name of your registry):

docker push MYREGISTRY.azurecr.io/bash



```
PS C:\labs\w[redacted]pnetcore> docker push mycontainerregistry985.azurecr.io/samples/aspcoreapp
The push refers to a repository [mycontainerregistry985.azurecr.io/samples/aspcoreapp]
81372cd079e3: Pushed
a22e691a77bf: Pushed
aa3b7dcd94c2: Pushed
3fbf9eff1729: Pushed
0ff94c2abbf0: Pushed
7156b3f84ea5: Pushing [======================>                            ] 169.2 MB/377.3 MB
8dd9effaebda: Pushed
1381be7aea31: Pushed
c19f69b0297d: Pushing [==============================>                    ] 194.6 MB/316.3 MB
d6f5573a9043: Pushed
031c51117bcf: Pushed
c8b91134fd12: Pushed
8bf09e5cdfbd: Skipped foreign layer
6c357baed9f5: Skipped foreign layer
```

10. 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:



11. If you want to pull from your registry you can run this command:
```
docker pull MYREGISTRY.azurecr.io/bash
```

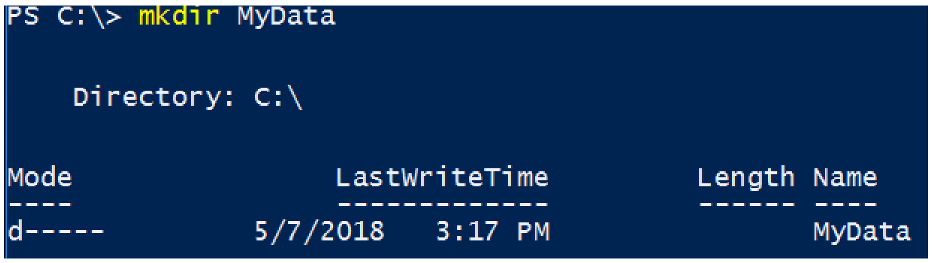
Exercise 2: Working with Data Volumes

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**

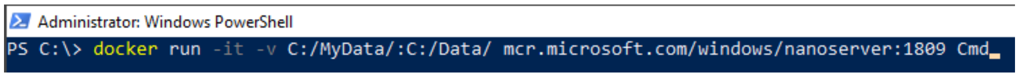
   1. From the host, go to C:\ and create a new directory by running the command "`mkdir c:\MyData`"



```
PS C:\> mkdir MyData

    Directory: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         5/7/2018   3:17 PM                MyData
```

   2. Run "`ls myData`" to show the empty directory

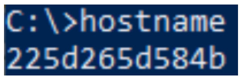
3. Run a container in interactive mode and mount the host directory as a data volume. Run the command. Note that we start a command prompt (not powershell) in a nanoserver container
docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd



```
Administrator: Windows PowerShell
PS C:\> docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd
```

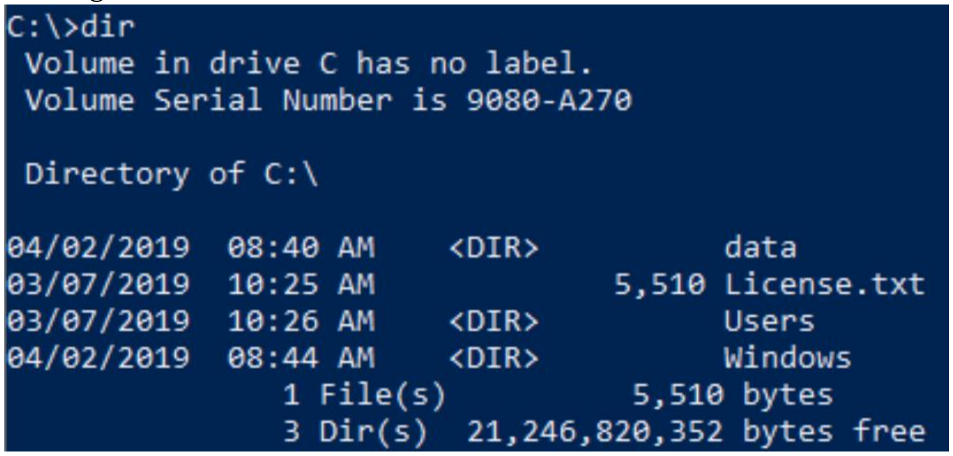
NOTE: Notice the –v switch that is required to mount the host directory C:\MyData inside the container as C:\Data. This will result in container access to contents of C:\MyData on the host inside the container as C:\Data. You can choose same name for the directory inside the container and host but it's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

4. On the container Console first check the hostname by running the command "`hostname`". The actual hostname for your container may be different.



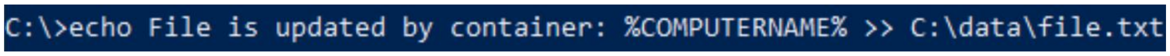
```
C:\>hostname
225d265d584b
```

5. List the directories by running the command "`dir`". Notice the *data* directory as part of the listing.



```
C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 9080-A270

 Directory of C:\

04/02/2019  08:40 AM    <DIR>          data
03/07/2019  10:25 AM             5,510 License.txt
03/07/2019  10:26 AM    <DIR>          Users
04/02/2019  08:44 AM    <DIR>          Windows
               1 File(s)          5,510 bytes
               3 Dir(s)  21,246,820,352 bytes free
```

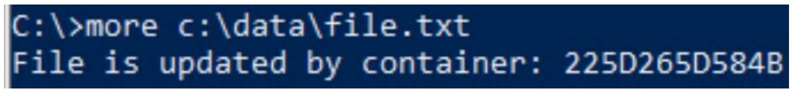
6. Create a file in the folder and add more text to it. Run the command:
echo File is updated by container: %COMPUTERNAME% >> c:\data\file.txt



```
C:\>echo File is updated by container: %COMPUTERNAME% >> C:\data\file.txt
```

Note that %COMPUTERNAME% is that same as hostname

7. Look at content inside the file.txt by running the command
more c:\data\file.txt

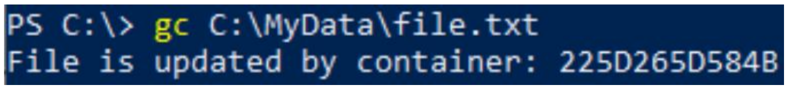


```
C:\>more c:\data\file.txt
File is updated by container: 225D265D584B
```

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

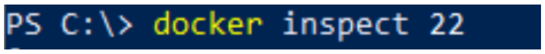
8. Exit the container session by typing **`exit`**

9. 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.



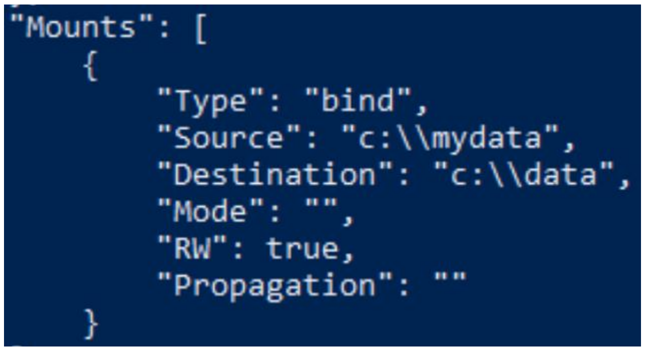
```
PS C:\> gc C:\MyData\file.txt
File is updated by container: 225D265D584B
```

10. 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.



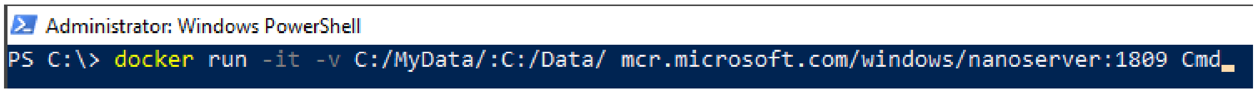
```
PS C:\> docker inspect 22
```

11. 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.



```
"Mounts": [
    {
        "Type": "bind",
        "Source": "c:\\mydata",
        "Destination": "c:\\data",
        "Mode": "",
        "RW": true,
        "Propagation": ""
    }
```

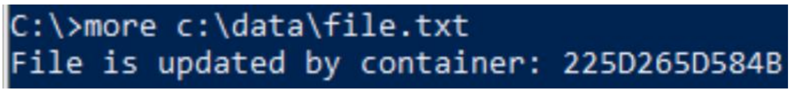
12. Let's run another container in interactive mode and mount the host directory as a data volume. Run the command
`docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd`



Administrator: Windows PowerShell

```
PS C:\> docker run -it -v C:/MyData/:C:/Data/ mcr.microsoft.com/windows/nanoserver:1809 Cmd_
```

13. Look at content inside the file.txt by running the command
`more c:\data\file.txt`

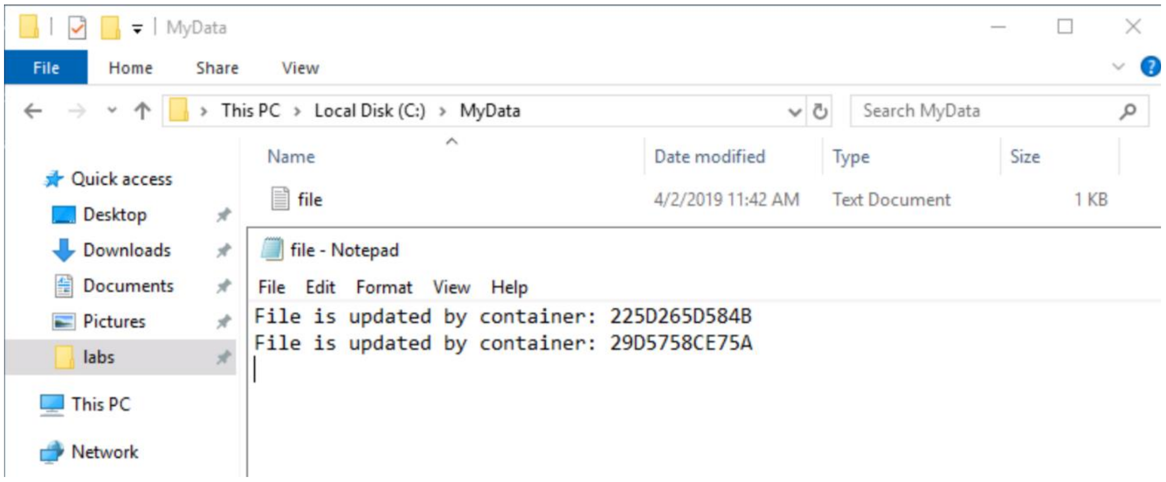


```
C:\>more c:\data\file.txt
File is updated by container: 225D265D584B
```

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

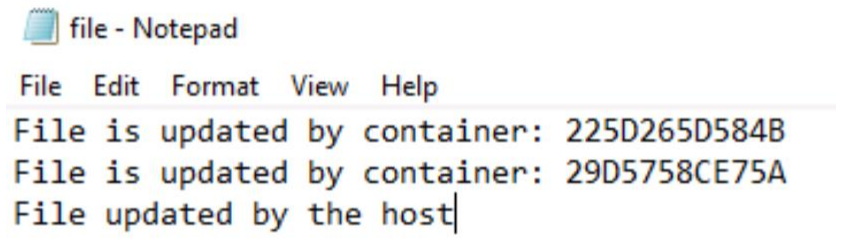
```
echo File is updated by container: %COMPUTERNAME% >>
c:\data\file.txt
```

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



```
File is updated by container: 225D265D584B
File is updated by container: 29D5758CE75A
```

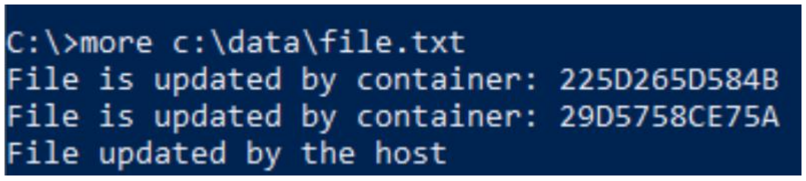
16. Update the content of the file with notepad and save it.



```
File is updated by container: 225D265D584B
File is updated by container: 29D5758CE75A
File updated by the host
```

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

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



```
C:\>more c:\data\file.txt
File is updated by container: 225D265D584B
File is updated by container: 29D5758CE75A
File updated by the host
```

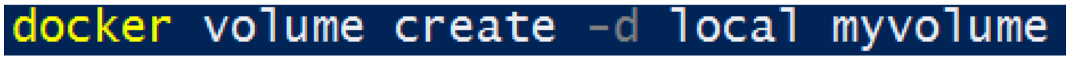
Note that because of concurrency challenges, you would probably not have multiple containers and hosts writing in the same file. The purpose of this exercise was only to show how we can persistent data across containers beyond their short lifecycle.

18. Finally, you can run `exit` to stop the running containers

2. Mount a shared-storage volume as a data volume

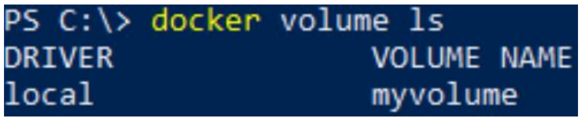
In this task, you will demonstrate how to create and use a shared-storage volume. To keep the demonstration accessible and easy to follow, you will use the *local* driver which uses local host for the storage. However, the exact same concepts will work against production ready storage drivers like Convoy 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`**"



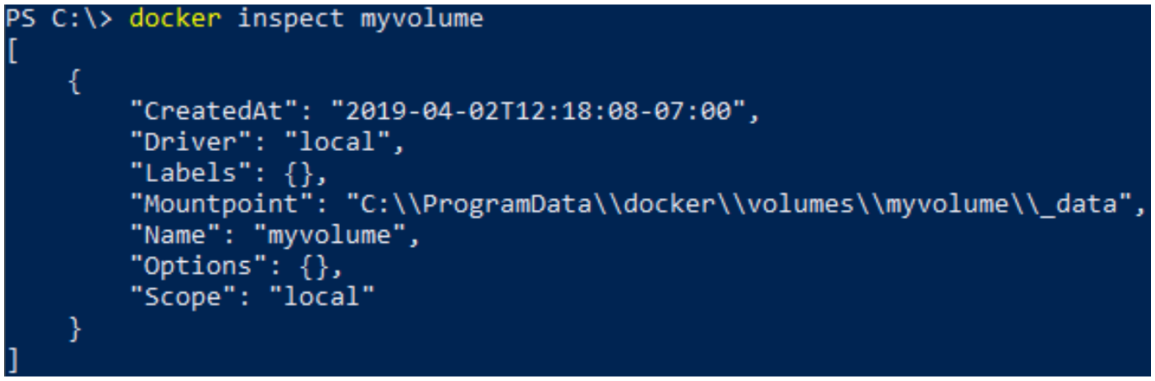
```
docker volume create -d local myvolume
```

2. You can list all the volumes by running the command "**`docker volume ls`**". Notice that myvolume is available as a local driver.



```
PS C:\> docker volume ls
DRIVER              VOLUME NAME
local               myvolume
```

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



```
PS C:\> docker inspect myvolume
[
    {
        "CreatedAt": "2019-04-02T12:18:08-07:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "C:\\ProgramData\\docker\\volumes\\myvolume\\_data",
        "Name": "myvolume",
        "Options": {},
        "Scope": "local"
    }
]
```

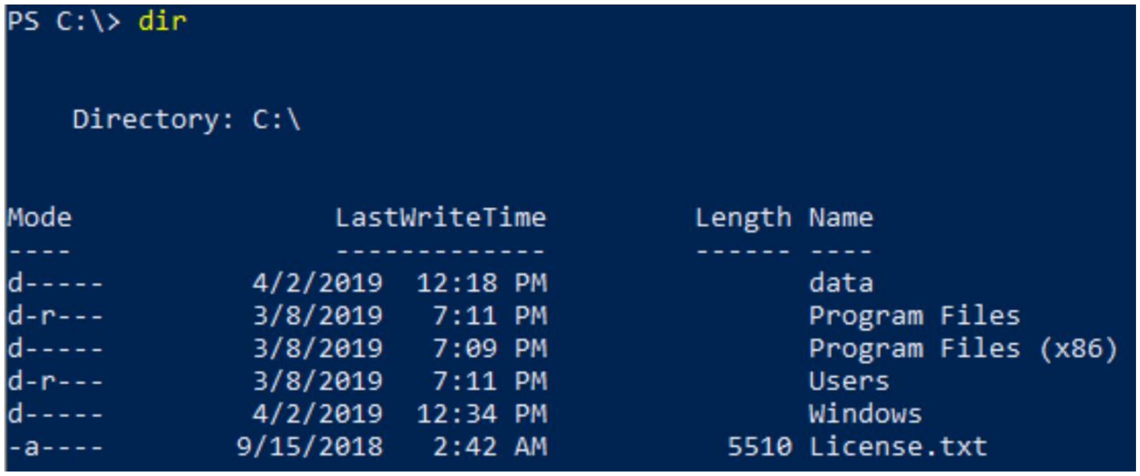
Notice that Mountpoint is set to location on C drive under ProgramData\docker folder. This is the default location for local storage drivers. If you have used another commercial storage driver, the location may be different.

4. 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`

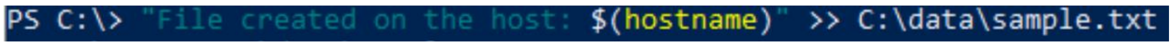
This command is like the command from last section where you shared the host directory, except that within the –v switch you are using the name of storage volume rather than path to host directory. Everything else remain the same.

5. On the PowerShell command prompt inside the container run the command "`dir`" to list the directories available on the container.



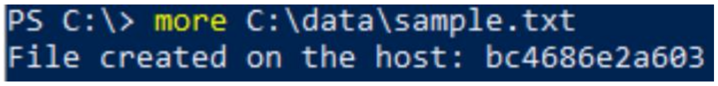
```
PS C:\> dir


    Directory: C:\


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         4/2/2019  12:18 PM                data
d-r---         3/8/2019   7:11 PM                Program Files
d-----         3/8/2019   7:09 PM                Program Files (x86)
d-r---         3/8/2019   7:11 PM                Users
d-----         4/2/2019  12:34 PM                Windows
-a----        9/15/2018   2:42 AM           5510 License.txt
```

6. 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`



```
PS C:\> "File created on the host: $(hostname)" >> C:\data\sample.txt
```

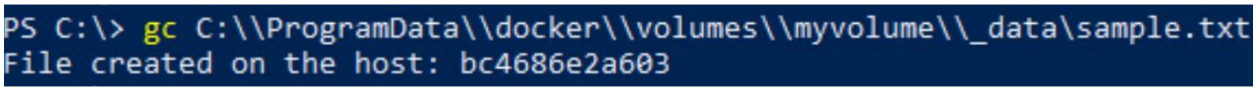
7. Confirm that file sample.txt has been created successfully by running the command
`more c:\data\sample.txt`



```
PS C:\> more C:\data\sample.txt
File created on the host: bc4686e2a603
```

8. Now exit the container by running the command "`exit`". This will take you back to PowerShell Console on the host.

9. To check the content of sample.txt file from the host run the command
`gc C:\\ProgramData\\docker\\volumes\\myvolume\\_data\sample.txt`



```
PS C:\> gc C:\\ProgramData\\docker\\volumes\\myvolume\\_data\sample.txt
File created on the host: bc4686e2a603
```

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.

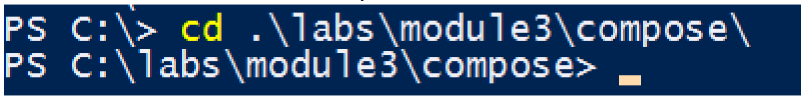
- Express specific dependencies, such as the Web Api container needs to start before Web App.

- 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**

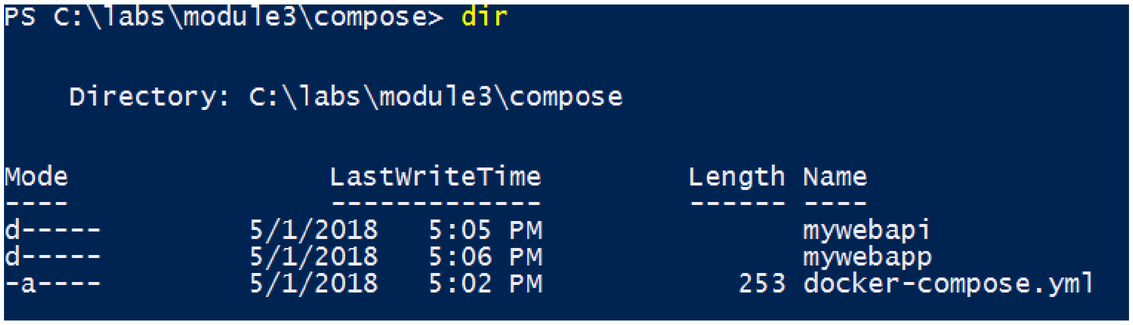
    1. Launch the PowerShell Console (if not already running) and change your current directory to "compose" folder by running the command "`cd C:\labs\module3\compose`"



```
PS C:\> cd .\labs\module3\compose\
PS C:\labs\module3\compose> _
```

    2. Before proceeding further let's stop and remove all the running containers from previous task. Run the command "`docker rm (docker ps -aq) -f`"

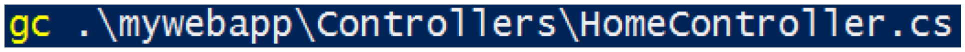
    3. First, look at directory structure by running the command "`dir`".



```
PS C:\labs\module3\compose> dir


    Directory: C:\labs\module3\compose


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         5/1/2018   5:05 PM                mywebapi
d-----         5/1/2018   5:06 PM                mywebapp
-a----         5/1/2018   5:02 PM            253 docker-compose.yml
```

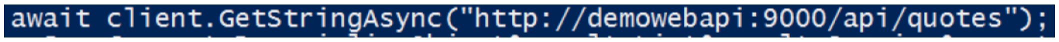
    4. Notice that you have two folders "mywebapi" and "mywebapp" representing the web API and web application respectively. First, you will inspect the piece of code

that is making the RESTful call to mywebapi. To do that run the command: "`gc .\mywebapp\Controllers\HomeController.cs`"



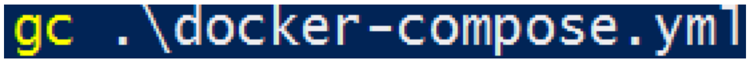
```
gc .\mywebapp\Controllers\HomeController.cs
```

5. 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://demowebapi:9000/api/quotes`. Notice the use of "demowebapi" 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.



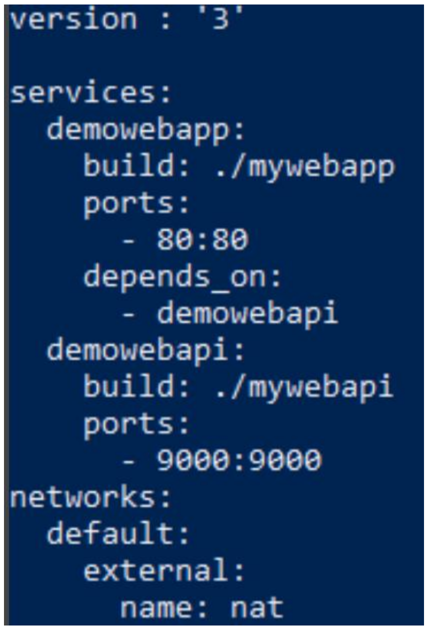
```
await client.GetStringAsync("http://demowebapi:9000/api/quotes");
```

6. Let's inspect the docker-compose.yml file. Run the command "`gc .\docker-compose.yml`"



```
gc .\docker-compose.yml
```

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



```
version : '3'

services:
  demowebapp:
    build: ./mywebapp
    ports:
      - 80:80
    depends_on:
      - demowebapi
  demowebapi:
    build: ./mywebapi
    ports:
      - 9000:9000
networks:
  default:
    external:
      name: nat
```

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 “*demowebapp*” and “*demowebapi*”. The demowebapp 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 *demowebapi* container first since *demowebapp* depends on it. Also, the discoverability is done by using the service names (as mentioned in the paragraph above, *demowebapp* can access *demowebapi* by its service name, rather than FQDN or IP Address).

Next is the *demowebapi* 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.

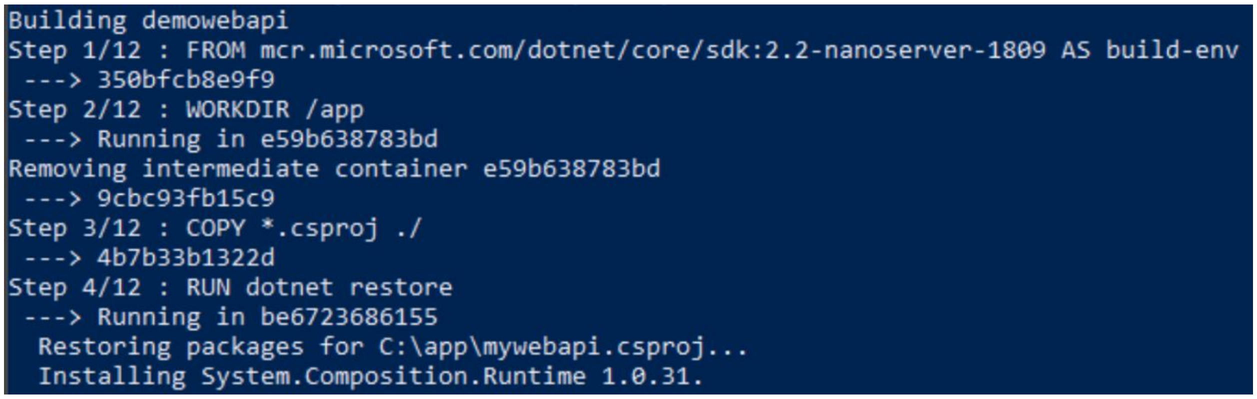
2. **Docker Compose Up**

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

```
docker-compose.exe up
```

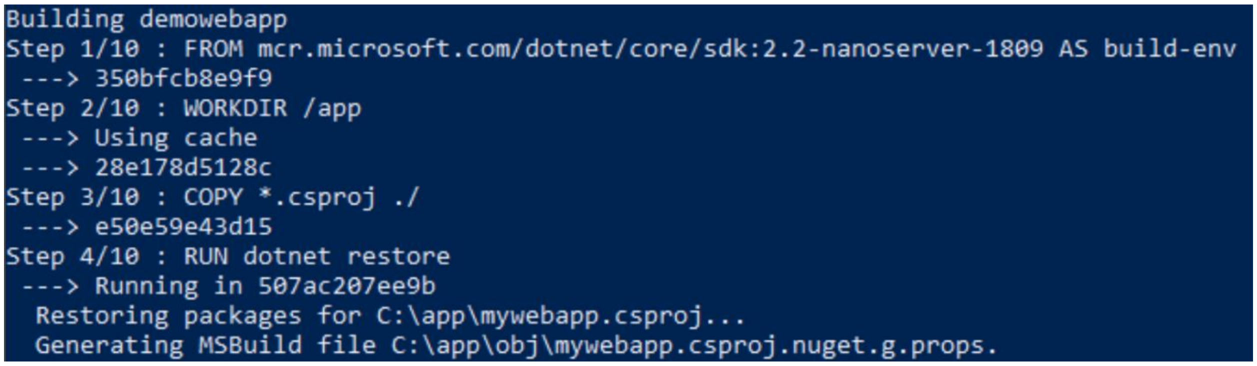
NOTE: The docker-compose.exe tries to make it simple to start and stop the services (running containers) with commands like up and down. The “-d” switch works the same as when used with the docker build command, which instructs docker to run the container in the background rather than interactively. 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.



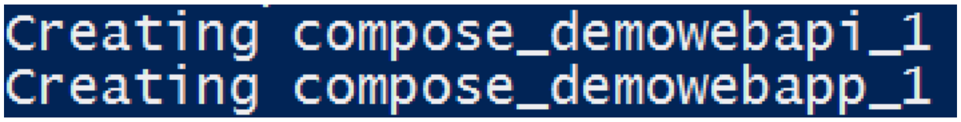
```
Building demowebapi
Step 1/12 : FROM mcr.microsoft.com/dotnet/core/sdk:2.2-nanoserver-1809 AS build-env
 ---> 350bfcb8e9f9
Step 2/12 : WORKDIR /app
 ---> Running in e59b638783bd
Removing intermediate container e59b638783bd
 ---> 9cbc93fb15c9
Step 3/12 : COPY *.csproj ./
 ---> 4b7b33b1322d
Step 4/12 : RUN dotnet restore
 ---> Running in be6723686155
  Restoring packages for C:\app\mywebapi.csproj...
  Installing System.Composition.Runtime 1.0.31.
```

Next, Docker will build the container image for "mywebapp."



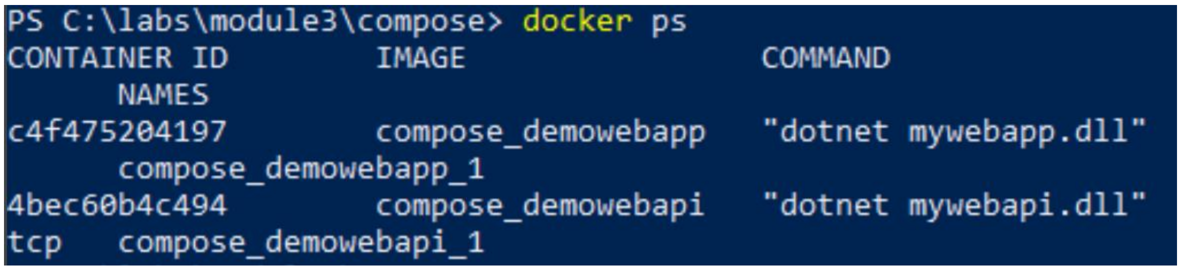
```
Building demowebapp
Step 1/10 : FROM mcr.microsoft.com/dotnet/core/sdk:2.2-nanoserver-1809 AS build-env
 ---> 350bfcb8e9f9
Step 2/10 : WORKDIR /app
 ---> Using cache
 ---> 28e178d5128c
Step 3/10 : COPY *.csproj ./
 ---> e50e59e43d15
Step 4/10 : RUN dotnet restore
 ---> Running in 507ac207ee9b
  Restoring packages for C:\app\mywebapp.csproj...
  Generating MSBuild file C:\app\obj\mywebapp.csproj.nuget.g.props.
```

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



```
Creating compose_demowebapi_1
Creating compose_demowebapp_1
```

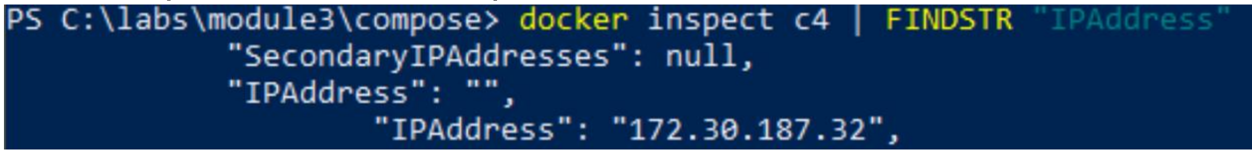
2. You can check details about running containers by executing the command "**`docker ps`**".



```
PS C:\labs\module3\compose> docker ps
CONTAINER ID        IMAGE                 COMMAND
       NAMES
c4f475204197        compose_demowebapp    "dotnet mywebapp.dll"
       compose_demowebapp_1
4bec60b4c494        compose_demowebapi    "dotnet mywebapi.dll"
tcp    compose_demowebapi_1
```

3. 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:

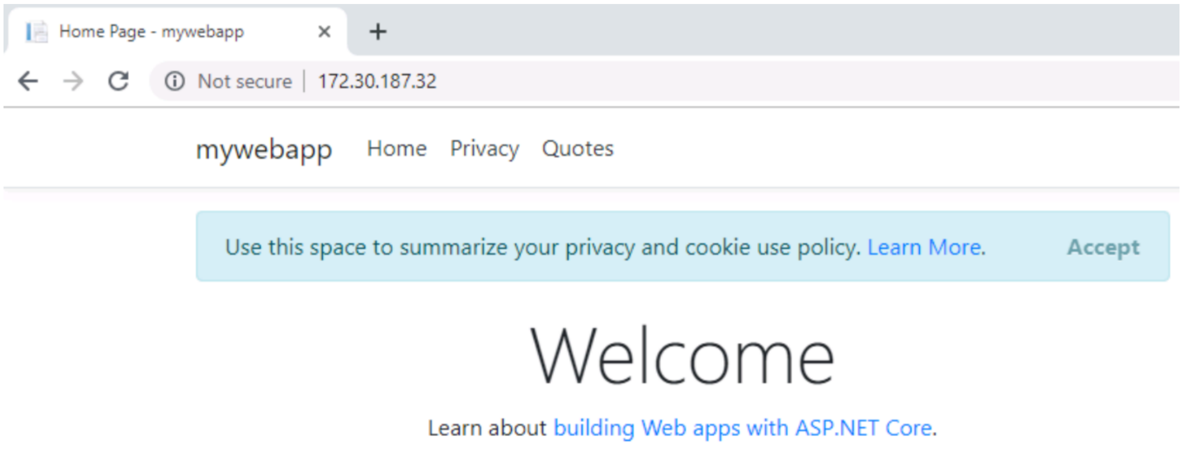
```
docker inspect <container id> | FINDSTR "IPAddress"
```



```
PS C:\labs\module3\compose> docker inspect c4 | FINDSTR "IPAddress"
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.30.187.32",
```

4. 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.

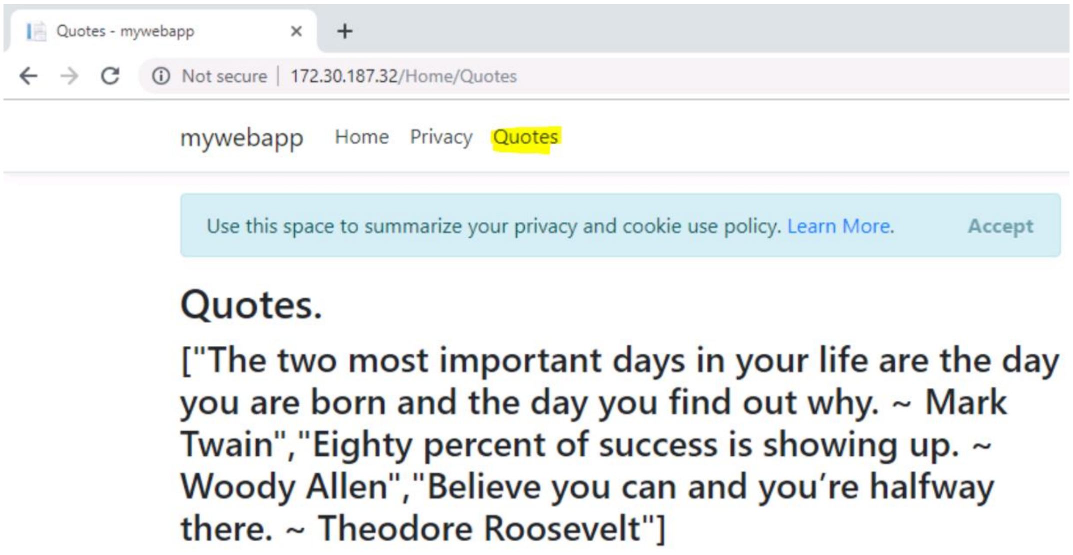


Use this space to summarize your privacy and cookie use policy. Learn More. Accept

Welcome

Learn about building Web apps with ASP.NET Core.

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.



Use this space to summarize your privacy and cookie use policy. Learn More. Accept

Quotes.

["The two most important days in your life are the day you are born and the day you find out why. ~ Mark Twain","Eighty percent of success is showing up. ~ Woody Allen","Believe you can and you're halfway there. ~ Theodore Roosevelt"]

5. 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`”. Notice the response is 200 OK and containing the actual JSON content.

3. **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 down`**”. Notice first the containers are stopped and then removed.

2. You have now completed all the tasks in this exercise.

Exercise 4: Docker Networking

In this exercise you will work with various PowerShell and Docker CLI commands provide walkthrough of following:

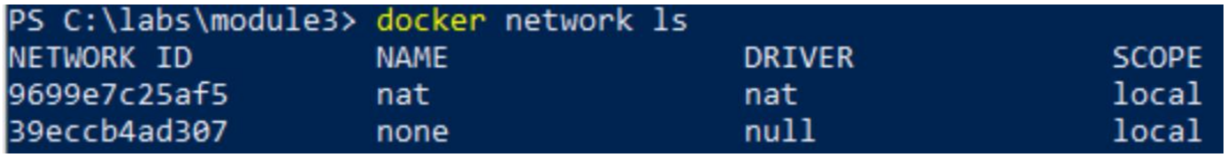
- Docker default networks
- Create custom NAT Network with subnet and gateway and replace default Docker with it
- Remove custom NAT network

Tasks

1. **Display All Docker Networks**

   You can retrieve container networks using the Docker CLI. In Windows server 2016 there was also the PowerShell Get-ContainerNetwork cmdlet but it is not there in Windows Server 2019.

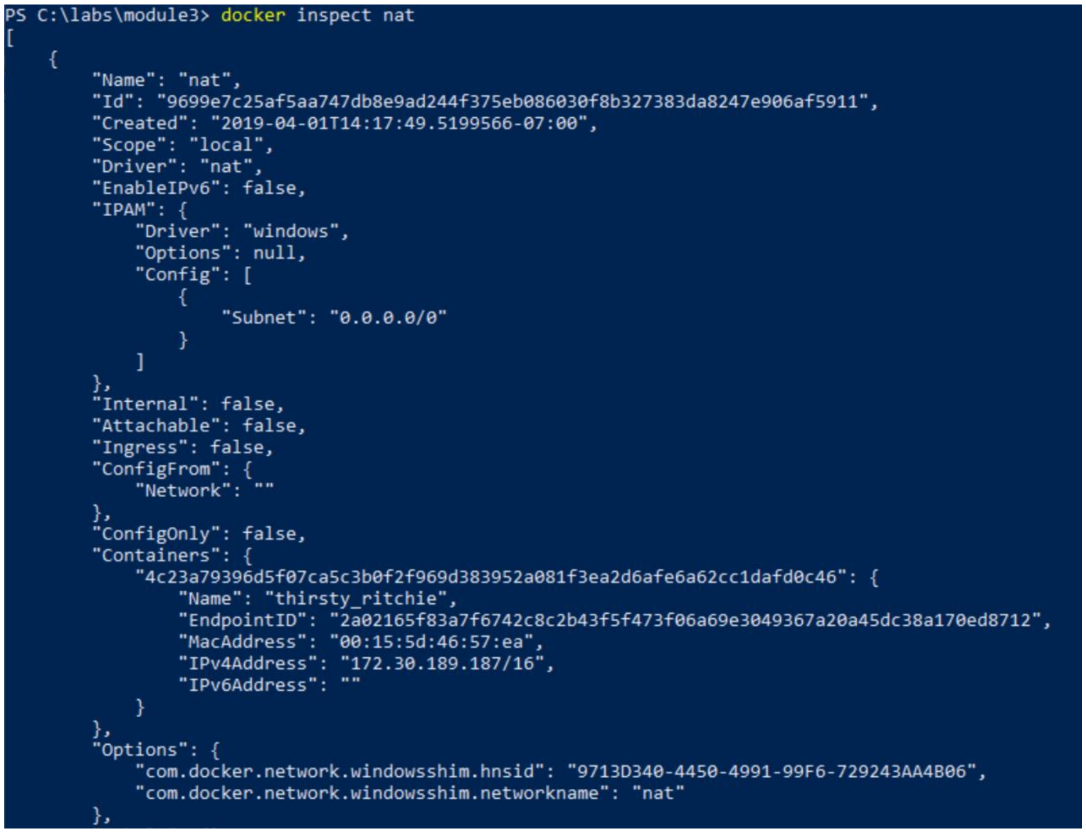
   1. Docker provides native docker command that provides list of networks available to docker. To view the list of networks available to docker run the command **docker network ls**



```
PS C:\labs\module3> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
9699e7c25af5        nat                 nat                 local
39eccb4ad307        none                null                local
```

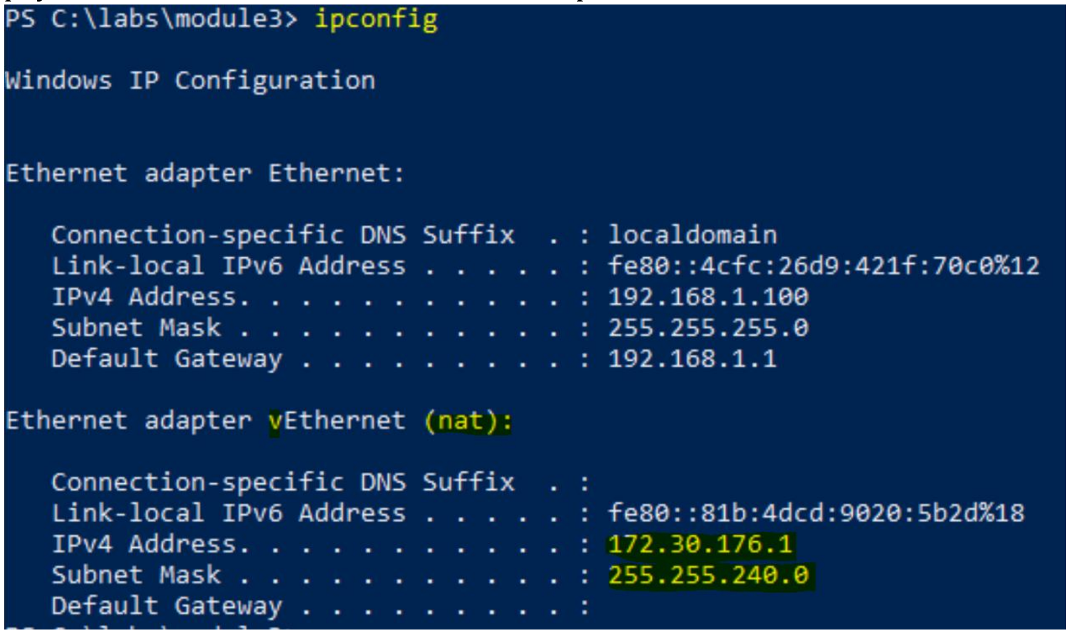
      NOTE: The 'nat' network is the default network for containers running on Windows. Any containers that are run on Windows without any flags or arguments to implement specific network configurations will be attached to the default 'nat' network, and automatically assigned an IP address from the 'nat' network's internal prefix IP range. The default NAT network also supports port forwarding from container host to internal containers. For example, you can simply run SQL Server Express in a container by providing the "p" flag so that specified port numbers will be mapped from host to container

   2. You can view detailed information about any particular Docker network by using docker network inspect command. To get more information about nat network run the command "**docker inspect nat**". Notice the output is in JSON format. The **"Containers" key (which is empty in this case) refers to all containers that are using particular network**. At the moment there is not container running so its empty.



```
PS C:\labs\module3> docker inspect nat
[
    {
        "Name": "nat",
        "Id": "9699e7c25af5aa747db8e9ad244f375eb086030f8b327383da8247e906af5911",
        "Created": "2019-04-01T14:17:49.5199566-07:00",
        "Scope": "local",
        "Driver": "nat",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "windows",
            "Options": null,
            "Config": [
                {
                    "Subnet": "0.0.0.0/0"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "4c23a79396d5f07ca5c3b0f2f969d383952a081f3ea2d6afe6a62cc1dafd0c46": {
                "Name": "thirsty_ritchie",
                "EndpointID": "2a02165f83a7f6742c8c2b43f5f473f06a69e3049367a20a45dc38a170ed8712",
                "MacAddress": "00:15:5d:46:57:ea",
                "IPv4Address": "172.30.189.187/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.windowsshim.hnsid": "9713D340-4450-4991-99F6-729243AA4B06",
            "com.docker.network.windowsshim.networkname": "nat"
        },
```

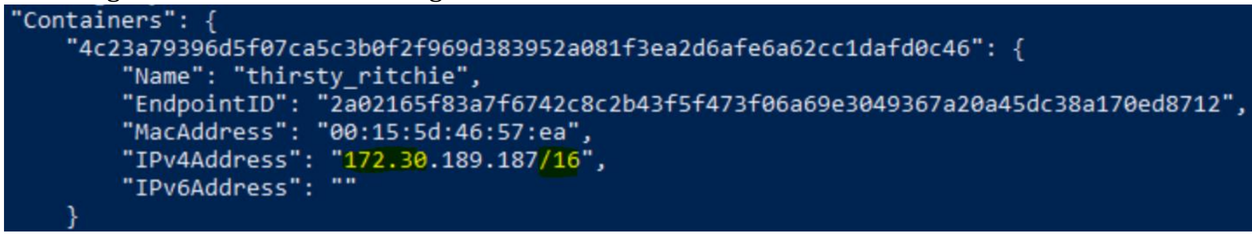
3. Show what it looks like locally on the host by running “**ipconfig**”. Call out the physical network and the virtual ethernet adapter.



```
PS C:\labs\module3> ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::4cfc:26d9:421f:70c0%12
   IPv4 Address. . . . . . . . . . . : 192.168.1.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1

Ethernet adapter vEthernet (nat):

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::81b:4dcd:9020:5b2d%18
   IPv4 Address. . . . . . . . . . . : 172.30.176.1
   Subnet Mask . . . . . . . . . . . : 255.255.240.0
   Default Gateway . . . . . . . . . :
```

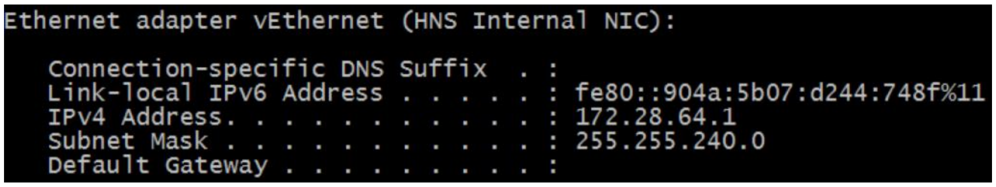
4. Launch a new container by running a command “**docker run -d mcr.microsoft.com/windows/nanoserver:1809 ping -t localhost**”. Once the container is running execute the command “**docker inspect nat**”. Notice that this

time “Containers” section list the container detail that is using the nat network including its ID, IPv4 address along with other details.



```
"Containers": {
    "4c23a79396d5f07ca5c3b0f2f969d383952a081f3ea2d6afe6a62cc1dafd0c46": {
        "Name": "thirsty_ritchie",
        "EndpointID": "2a02165f83a7f6742c8c2b43f5f473f06a69e3049367a20a45dc38a170ed8712",
        "MacAddress": "00:15:5d:46:57:ea",
        "IPv4Address": "172.30.189.187/16",
        "IPv6Address": ""
    }
```

5. You can also verify that the container IPv4 Address (172.28.74.118, as shown above) is coming from subnet range that is part of nat. Now to make sure container host also using the same nat network run the command “ipconfig” and notice the output under section “Ethernet adapter vEthernet (HNS Internal NIC)”. Notice the host IPv4 Address “172.28.64.1”.



```
Ethernet adapter vEthernet (HNS Internal NIC):

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::904a:5b07:d244:748f%11
   IPv4 Address. . . . . . . . . . . : 172.28.64.1
   Subnet Mask . . . . . . . . . . . : 255.255.240.0
   Default Gateway . . . . . . . . . :
```

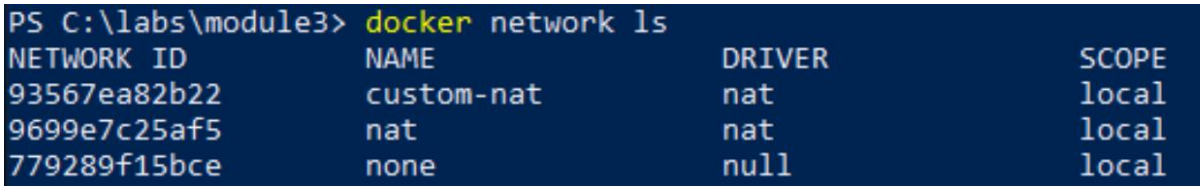
2. Creating Custom Docker NAT Network

Docker allows you to create custom NAT networks. In this task you will create and configure a custom NAT network in addition to the default nat network. The main reason why we keep the default is that it cannot be deleted.

1. Create a new docker nework by running the command:

   docker network create -d nat --subnet=192.168.15.0/24 --gateway=192.168.15.1 custom-nat

   The "d" flag stands for network driver and specifies the network type you want to create. Which in this case is “nat”. You are also providing the IP prefix and gateway address using -subnet and -gateway flags.
2. Use the “**docker network ls**” command and notice that “custom-nat” network is available.



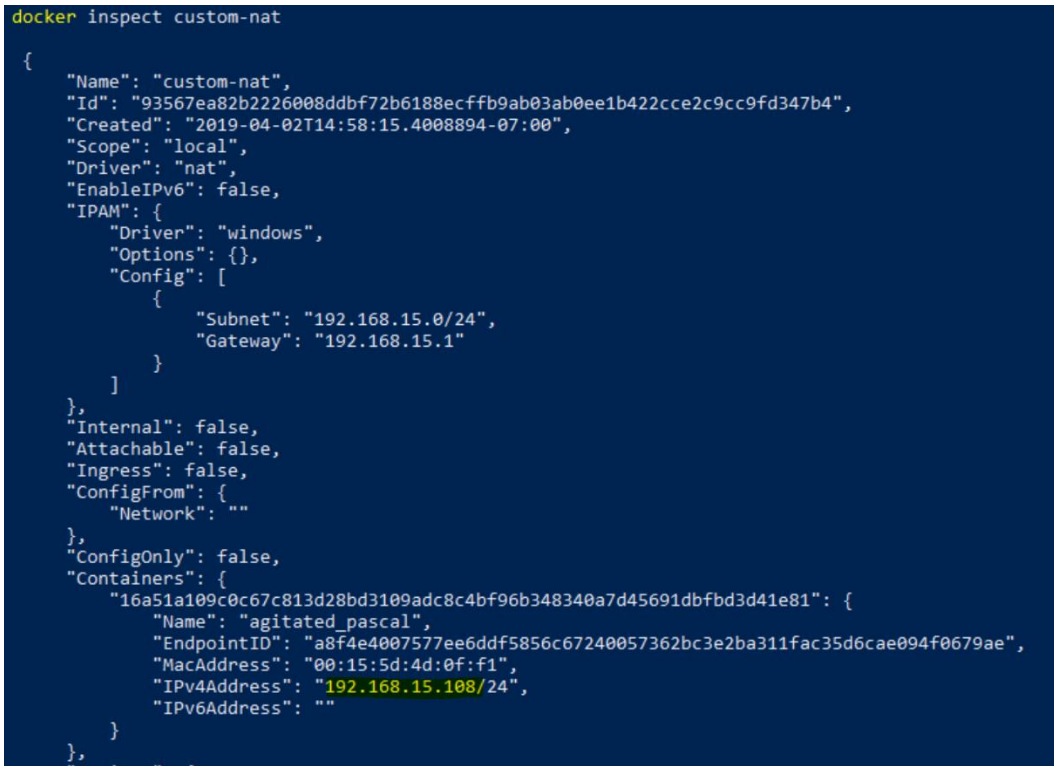
```
PS C:\labs\module3> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
93567ea82b22        custom-nat          nat                 local
9699e7c25af5        nat                 nat                 local
779289f15bce        none                null                local
```

3. To use the new custom nat network for containers launch a new container by using the command:
   docker run -d --network=custom-nat mcr.microsoft.com/windows/nanoserver:1809 ping -t localhost

Notice the use of --network switch which allows you to force docker to use specific network for the container.

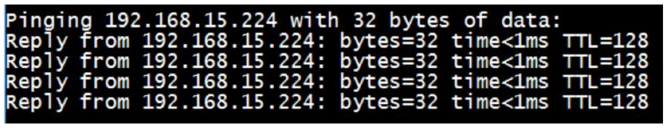
4. Now, use the "**docker inspect custom-nat**" command to get the detailed information about custom-nat network and container(s) that is using it.

   Notice that subnet and gateway values reflect the values you used earlier during the creation of the network. Also, the container IPv4 Address (192.168.15.108, this may be different in your case)is in the custom-nat network.



```
docker inspect custom-nat

 {
     "Name": "custom-nat",
     "Id": "93567ea82b2226008ddbf72b6188ecffb9ab03ab0ee1b422cce2c9cc9fd347b4",
     "Created": "2019-04-02T14:58:15.4008894-07:00",
     "Scope": "local",
     "Driver": "nat",
     "EnableIPv6": false,
     "IPAM": {
         "Driver": "windows",
         "Options": {},
         "Config": [
             {
                 "Subnet": "192.168.15.0/24",
                 "Gateway": "192.168.15.1"
             }
         ]
     },
     "Internal": false,
     "Attachable": false,
     "Ingress": false,
     "ConfigFrom": {
         "Network": ""
     },
     "ConfigOnly": false,
     "Containers": {
         "16a51a109c0c67c813d28bd3109adc8c4bf96b348340a7d45691dbfbd3d41e81": {
             "Name": "agitated_pascal",
             "EndpointID": "a8f4e4007577ee6ddf5856c67240057362bc3e2ba311fac35d6cae094f0679ae",
             "MacAddress": "00:15:5d:4d:0f:f1",
             "IPv4Address": "192.168.15.108/24",
             "IPv6Address": ""
         }
     },
```

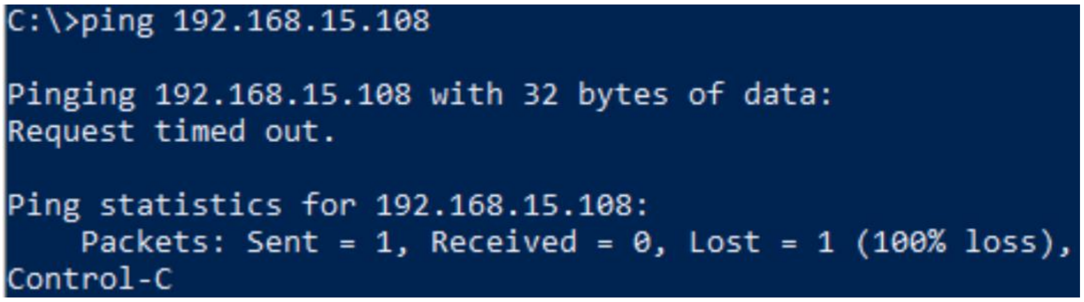
5. To confirm that the container host and access container run the command "**ping <<Container - IPv4 Address >>**". You can look for container IP Address in the output from previous command. Notice that host can successfully access the container using its IP.



```
Pinging 192.168.15.224 with 32 bytes of data:
Reply from 192.168.15.224: bytes=32 time<1ms TTL=128
Reply from 192.168.15.224: bytes=32 time<1ms TTL=128
Reply from 192.168.15.224: bytes=32 time<1ms TTL=128
Reply from 192.168.15.224: bytes=32 time<1ms TTL=128
```

6. Now let's create a new container on the net network and open a command prompt.
   docker run -it --network=nat mcr.microsoft.com/windows/nanoserver:1809 cmd

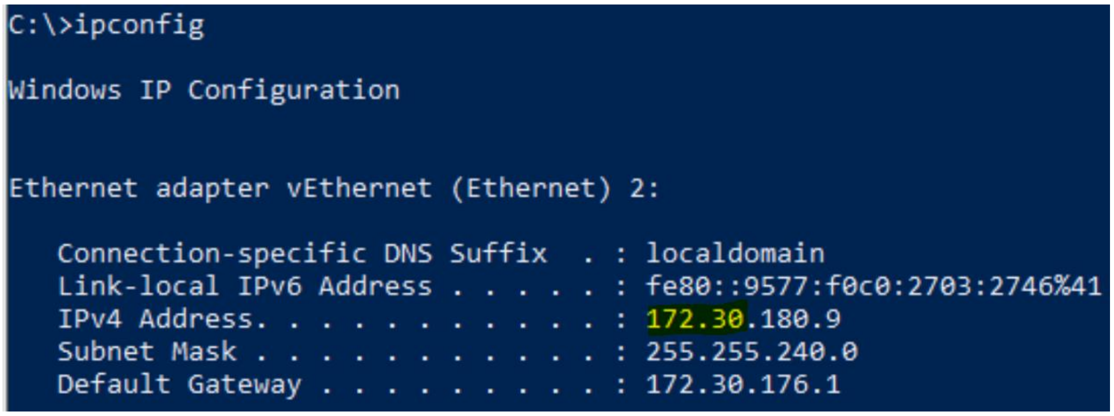
7. We can try to ping the previous container with `**ping <Container - IPv4 Address>**` where the IP Address is the same that we have just pinged from the host. Hit **ctr-C** to stop the ping operation



```
C:\>ping 192.168.15.108

Pinging 192.168.15.108 with 32 bytes of data:
Request timed out.

Ping statistics for 192.168.15.108:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
Control-C
```

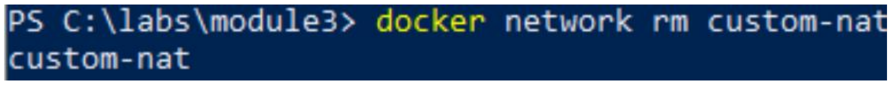
Note: Because the two containers are on separated networks, they cannot ping each other using their IP Address.

8. Run `ipconfig` from the container to check the that IP Address belongs to the nat network. Then run `exit` to go back to the host.



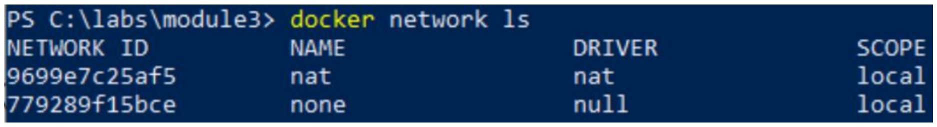
```
C:\>ipconfig

Windows IP Configuration


Ethernet adapter vEthernet (Ethernet) 2:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::9577:f0c0:2703:2746%41
   IPv4 Address. . . . . . . . . . . : 172.30.180.9
   Subnet Mask . . . . . . . . . . . : 255.255.240.0
   Default Gateway . . . . . . . . . : 172.30.176.1
```

9. Remove all containers to be allowed to remove the custom network we have created (if some containers are still attached to the network, the network deletion will fail) `docker rm (docker ps -aq) -f`
10. You may now remove the **custom-nat** network
    docker network rm custom-nat



```
PS C:\labs\module3> docker network rm custom-nat
custom-nat
```

11. Check that only **nat** network remains `docker network ls`



```
PS C:\labs\module3> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
9699e7c25af5        nat                 nat                 local
779289f15bce        none                null                local
```