

Table of Contents

Lab: Module 3 - Docker Advanced Concepts	2
Duration: 75 minutes	2
Exercise 1: Working with Data Volumes	2
Exercise 2: Working with Docker-Compose	9
Exercise 3: Docker Networking	16
Exercise 4: Running Containers with Memory and CPU Constrains	20

Lab: Module 3 - Docker Advanced Concepts

Duration: 75 minutes

Exercise 1: Working with Data Volumes

In this exercise, you will learn 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 on the host shared 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).

Docker containers, by design, are volatile when it comes to data persistence. This means that if you remove a container, for example, using `docker -rm` command, all the data that was in the container (running or stopped) will be lost. This certainly causes a challenge for applications that are running in the container and need to manage state. A good example here would be a SQL Server Database file from a previous lab that is required to be persisted beyond the life of the container running the SQL engine. The solution to this problem is to use data volumes. Data volumes are designed to persist data, independent of the container's lifecycle.

Volumes are initialized when a container is created. Some of the key characteristics of volumes are listed below:

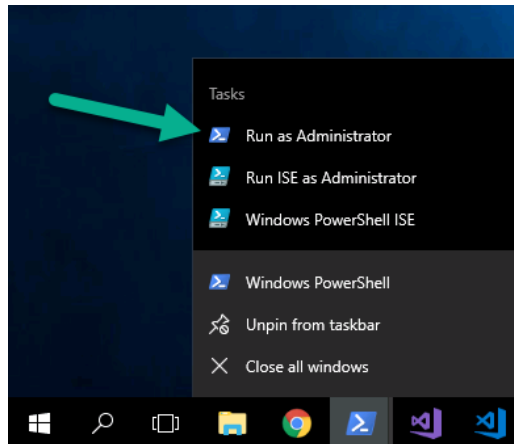
- Data volumes can be shared and reused among containers.
- Changes to a data volume are made directly by the container or the host.
- Data volumes persist even if the container itself is deleted.

NOTE: Docker never automatically deletes volumes when you remove a container nor will it "garbage collect" volumes that are no longer referenced by a container. This means you are responsible for cleaning up volumes yourself.

Tasks

1. Mount a host directory as a data volume

1. Log on to your Windows Server 2016 Lab environment as per previous labs.
2. You will need to run the commands in this section using the PowerShell console as an administrator. Right click the PowerShell icon on the taskbar and select "Run as Administrator".



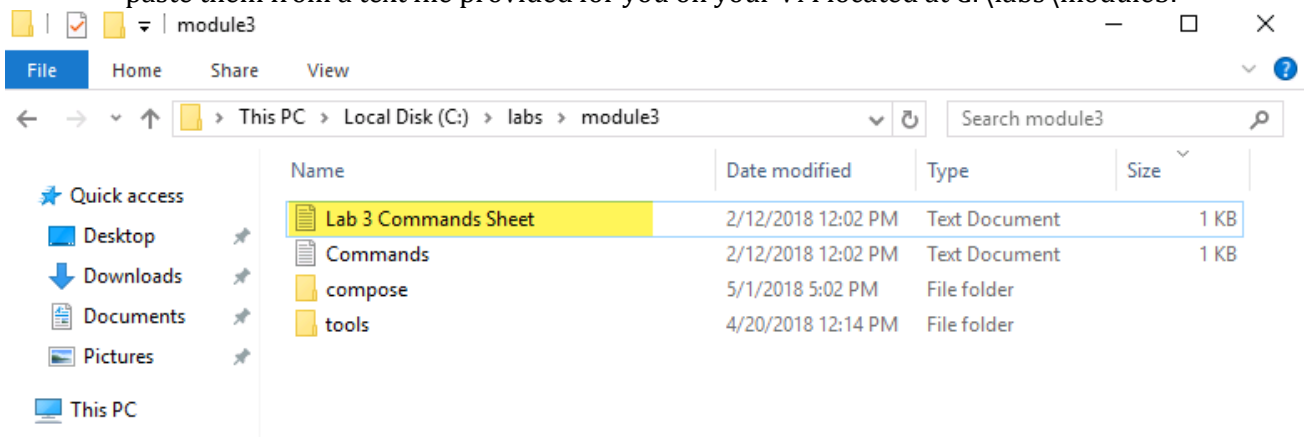
3. Navigate to your C:\ drive.
cd..
4. First you will create a directory on the host operating system and then add a plain text file to it. Create a new directory on the C drive by running the command "mkdir MyData"

```
PS C:\> mkdir MyData

Directory: C:\

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

5. The commands in this lab can be longer than usual, so to help you can copy and paste them from a text file provided for you on your VM located at C:\labs\module3.



6. Inside the MyData directory create a new file "file.txt" and add plain text to it by running the command (run whole line below with quotes)

"File created on the host: \$(hostname)" >> C:\MyData\file.txt

*NOTE: Pay attention to the **quotes that are part of the command** surrounding the "File... hostname)"*

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

7. Display the content of the file you created in previous step by running the command "gc C:\MyData\file.txt". Notice the host name placeholder value reflects the name of your host operating system.

```
PS C:\> gc C:\MyData\file.txt
File created on the host: WIN-EJI3ACSSVHL
```

8. You are now ready to run a container in the interactive mode and mount the host directory as a data volume. Run the command "docker run -it -v C:/MyData:/C:/Data/ microsoft/nanoserver powershell"

```
docker run -it -v C:/MyData:/C:/Data/ microsoft/nanoserver powershell
```

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)

9. On the container PowerShell Console first check the hostname by running the command "hostname". The actual hostname for your container may be different than pictured below. Most importantly though, the container hostname will be different from you VM hostname.

```
PS C:\> hostname
3a45f5770238
```

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

```
PS C:\> dir
```

Directory: C:\				
Mode	LastWriteTime		Length	Name
d----	10/2/2018	11:28 AM		data
d----	10/2/2018	11:29 AM		Program Files
d----	7/16/2016	5:09 AM		Program Files (x86)
d-r--	10/2/2018	11:29 AM		Users
d----	10/2/2018	11:29 AM		Windows
-a----	11/20/2016	3:32 AM	1894	License.txt

11. You can access and update the content of the data directory. First, run the command "dir c:\data" to list the content structure residing inside the data directory. Notice *file.txt* is present inside the data directory. This is the same file you created earlier on the host.

```
PS C:\> dir c:\data
```

Directory: C:\data				
Mode	LastWriteTime		Length	Name
-a----	10/2/2018	11:27 AM	88	file.txt

12. Look at content inside the file.txt by running the command "gc c:\data\file.txt"

```
PS C:\> gc c:\data\file.txt
File created on the host: WIN-EJI3ACSSVHL
```

13. Now update the file by adding more text to it. Run the command (full command below with quotes)

"File is updated by container: \$(hostname)" >> c:\data\file.txt

NOTE: Pay attention to the quotes that are part of the command surrounding the "File... hostname)"

```
PS C:\> "File is updated by container: $(hostname)" >> C:\Data\file.txt
```

14. To check that file.txt has been updated run the command "gc c:\data\file.txt". Notice that a line has been added to the file.txt with the host name of container.

```
PS C:\> gc c:\data\file.txt
File created on the host: WIN-EJI3ACSSVHL
File is updated by container: 3a45f5770238
```

15. You can now exit the container and return to host by running the command “exit”

```
PS C:\> exit
```

16. On the host PowerShell Console run the command “gc C:\MyData\file.txt”. Notice that changes made from the container persist on the host by the file.txt.

```
PS C:\> gc C:\MyData\file.txt
File created on the host: WIN-EJI3ACSSVHL
File is updated by container: 3a45f5770238
```

17. Run “docker ps -a” to get the ID of stopped containers. 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 captured in previous step.

```
PS C:\> docker inspect 3a45f5770238
```

18. 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": ""
  }
]
```

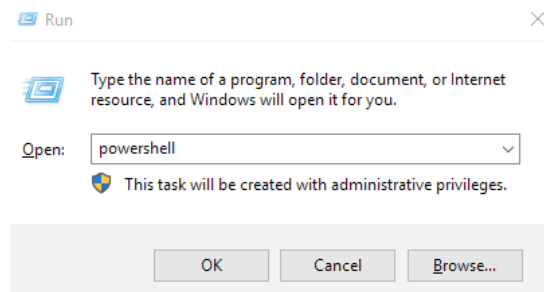
2. Mount a shared-storage volume as a data volume

In the previous section you learn how to mount a directory on the host as a data volume. That's a very handy way to share the content from host to container but it's not ideal in terms of portability. Basically, if you later run the container on a different host there is no guarantee that host will have the same directory. This would cause the app inside the container to break as it depends on a share that is not implemented by the host. In cases like these when a higher level of portability is desired, you can mount a *shared storage volume*. Docker has some volume plugins that allow you to provision and mount shared storage, such as iSCSI, NFS, or FC. A benefit of using shared volumes is that they

are host-independent. This means that a volume can be made available on any host on which a container is started as long as the container has access to the shared storage backend, and has the plugin installed.

In this exercise, you will learn how to create and use a shared-storage volume. To keep the lab 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. In previous task you issued a Docker command to stop a running container. You can also issue command to start the container which was stopped. All you need is a container ID (same container ID you used earlier to stop a container in previous section).
2. To start a container run “`docker start <<CONTAINER-ID>>`”. Replace CONTAINER-ID with container identifier you use in previous section to stop the container.
3. You will need to run the commands in this section using PowerShell console as an administrator. If you don’t have PowerShell console opened, then do so by first pressing “Windows + R” key and then typing “powershell” on the Run window. Finally, press Enter.



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

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

5. 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
```

6. You can use docker inspect command with the volumes too. Run the command
"docker inspect myvolume"

```
PS C:\> docker inspect myvolume
[
  {
    "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.

7. 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/ microsoft/nanoserver powershell"

```
docker run -it -v myvolume:C:/Data/ microsoft/nanoserver 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.

8. 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-----l       4/28/2017 12:44 AM              data
d-----       4/28/2017 12:44 AM      Program Files
d-----       7/16/2016 12:09 PM      Program Files (x86)
d-r---       4/28/2017 12:44 AM          Users
d-----       4/28/2017 12:45 AM        Windows
-a-----      11/20/2016 11:32 AM        1894 License.txt
```

9. 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
```

10. Confirm that file sample.txt has been created successfully by running the command
"gc c:\data\sample.txt"

```
PS C:\> gc c:\data\sample.txt  
File created on the host: 99e6ab615c3d
```

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

12. 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 b1e8ef12d185
```

Exercise 2: Working with Docker-Compose

Challenges with Multi-Container Applications

When working with multi-containers applications, you will need to make sure that applications can discover each other in a seamless fashion. Consider a quite common scenario where a Web Application (that acts as a front-end) calls to a backend RESTful Web API to fetch the content. In this scenario, the Web Application would need to access the Web API in a consistent fashion. Also, as the Web Application has a dependency on Web API, you will need to express that dependency when launching these applications as Containers. Importantly, you will want to have this "discoverability" and "dependency" as a feature for applications running in containers regardless of the environment. Meaning, you should be able to launch and test a multi-container application the same way across development, test and production environments.

Docker has provided a tool entitled, "docker-compose," that lets you describe your applications as services within a YAML file, which, by default has the name, docker-compose.yml. In "Docker speak," a YAML file is a "compose" file that defines services, networks and volumes. A service in this context really means, "a container in production." A service only runs one image, but it codifies the way that image runs – what ports it should use, how many replicas of the container should run (so that the service has the capacity it

needs), and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Working with Docker-Compose

In this task, 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 can be accessed by the Web App without the need to hardcode its FDQN or IP Address? Instead of hardcoding IP Address (or FDQN) you can use docker-compose.yml file to make these services discoverable.
 - *NOTE: Recall from the previous lab where a web application needed to access SQL Server running in a separate container. In that situation, we provided the web application the IP Address of the container running SQL Server in the web.config configuration file.*
- 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 all the running containers from previous task. Run the command:

“docker stop (docker ps -aq)”

```
PS C:\labs\module3\compose> docker stop (docker ps -aq)
b1e8ef12d185
cd0c78cf4daa
11573e615450
aa6f82318f61
89f20498f798
d08522c6d142
```

- 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
```

- 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
```

- 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 FDQN 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");
```

- 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 : '2'

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 whereas, *demowebapp* can access *demowebapi* by its service name, rather than FDQN 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, the networks section keeps the default settings to nat networking. This network declaration is needed for windows containers now. Basically, it tells docker compose to use default nat networking.

2. Docker Compose Up

1. We have pre-downloaded the docker-compose.exe file for you onto the VM, if you would like to see it, you can see the URL here:
https://github.com/docker/compose/releases/download/1.12.0/docker-compose-Windows-x86_64.exe
2. At this point, you are all set to run the multi-container application with a single command “docker-compose.exe up -d”

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

```
Building demowebapi
Step 1/6 : FROM microsoft/dotnet:nanoserver
----> 073e0bd2ca24
Step 2/6 : WORKDIR /app
----> Running in bd55e086532b
----> 52cac0850cee
Removing intermediate container bd55e086532b
Step 3/6 : COPY published ./
----> 22217c03889c
Removing intermediate container 857afbdcdb8b
Step 4/6 : ENV ASPNETCORE_URLS http://+:9000
----> Running in 1df832b62e2f
----> 2aa8878358b7
Removing intermediate container 1df832b62e2f
Step 5/6 : EXPOSE 9000
----> Running in fd5a58468458
----> 0960bd7c8b66
Removing intermediate container fd5a58468458
Step 6/6 : ENTRYPOINT dotnet mywebapi.dll
----> Running in 4d6cb6373d33
----> 88f54e3bb494
Removing intermediate container 4d6cb6373d33
Successfully built 88f54e3bb494
```

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

```

Building demowebapp
Step 1/6 : FROM microsoft/dotnet:nanoserver
----> 073e0bd2ca24
Step 2/6 : WORKDIR /app
----> Using cache
----> 52cac0850cee
Step 3/6 : COPY published ./
----> 82cb00f6c556
Removing intermediate container 2dca0318399b
Step 4/6 : ENV ASPNETCORE_URLS http://+:80
----> Running in e492ebe3fbe2
----> a34282815336
Removing intermediate container e492ebe3fbe2
Step 5/6 : EXPOSE 80
----> Running in 048843d2f4db
----> 85aaceebcafb
Removing intermediate container 048843d2f4db
Step 6/6 : ENTRYPOINT dotnet mywebapp.dll
----> Running in 9681f5369c41
----> 54c17d967ec8
Removing intermediate container 9681f5369c41
Successfully built 54c17d967ec8

```

- *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

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

```

PS C:\labs\module3\compose> docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED
07b7905fb30e	compose_demowebapp	"dotnet mywebapp.dll"	About a minute ago
/tcp	compose_demowebapp_1		
111a2c503732	compose_demowebapi	"dotnet mywebapi.dll"	About a minute ago
9000/tcp	compose_demowebapi_1		

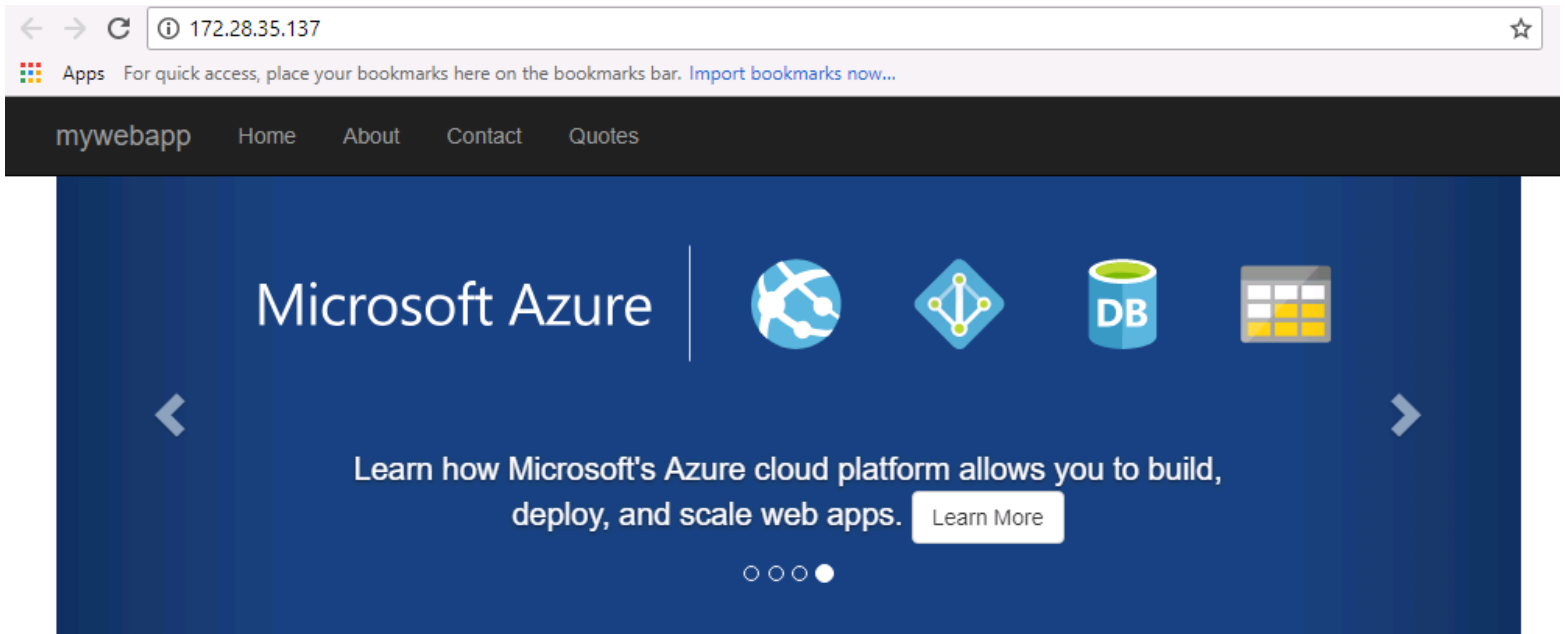
4. Let's test both the Web Application and Web API. First get the IP address of the web app by doing a docker inspect and putting in the Container ID of the web app:
"docker inspect <container id> | FINDSTR "IPAddress"

```

PS C:\labs\module3\compose> docker inspect 07b | FINDSTR "IPAddress"
"SecondaryIPAddresses": null,
"IPAddress": "",
"IPAddress": "172.28.35.137",

```

5. Open web browser of your choice and browse to the IP address from the previous step. You should land on the home page of web application as shown below.



Application uses

- Sample pages using ASP.NET Core MVC
- [Bower](#) for managing client-side libraries
- Theming using [Bootstrap](#)

How to

- [Add a Controller and View](#)
- [Manage User Secrets using Secret Manager.](#)
- [Use logging to log a message.](#)
- [Add packages using NuGet.](#)

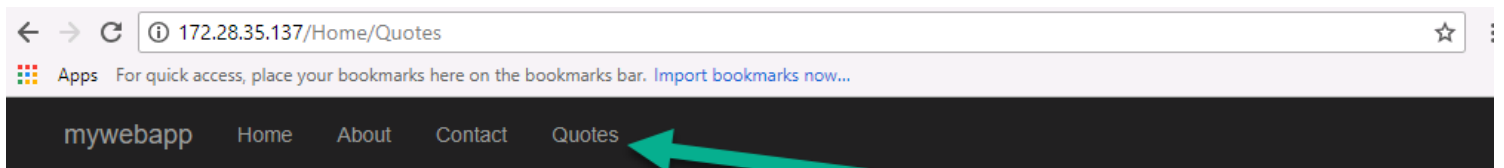
Overview

- [Conceptual overview of what is ASP.NET Core](#)
- [Fundamentals of ASP.NET Core such as Startup and middleware.](#)
- [Working with Data](#)
- [Security](#)

Run & Deploy

- [Run your app](#)
- [Run tools such as EF migrations and more](#)
- [Publish to Microsoft Azure Web Apps](#)

To test the Web API you will can 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.



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"]

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.exe down`”. Notice first the containers are stopped and then removed.
2. You have now completed all the tasks in this exercise.

Exercise 3: Docker Networking

In this exercise you will work with various PowerShell and Docker CLI commands to view Docker default networks, create custom Docker NAT Network and replace default Docker with it. Finally, you will remove the custom NAT network so Docker will create new default NAT network automatically.

Tasks

1. Display All Docker Networks

You can retrieve container networks using the Docker CLI or the PowerShell Get-ContainerNetwork cmdlet.

1. Launch the PowerShell Console (if not already running) and run the command: “`Get-ContainerNetwork`”. Notice the output and name of network NAT which is default network that docker is using. Please note that your subnets value may be different.

Name	Id	Subnets	Mode
----	--	-----	----
nat	4a57d4d8-29ad-40bb-9945-52d3d47894cf	{172.28.64.0/20}	NAT

2. 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`”

NETWORK ID	NAME	DRIVER	SCOPE
743696c2bb9f	nat	nat	local
225d8b0bef1f	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

- To view detail information about any specific Docker network by using docker network inspect command. To get more information about nat network run the command “docker network inspect nat”. Notice the output is in JSON file. The “Containers” key (which is empty in this case) refers to all containers that are using this specific network. Now, there is no container running, so it's empty.

```
{
  "Name": "nat",
  "Id": "743696c2bb9fad927465a8cd2fa10eb4b9366da6bd78415fa8fedc8ab6a2db2b",
  "Created": "2017-08-19T20:33:55.5498926Z",
  "Scope": "local",
  "Driver": "nat",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "windows",
    "Options": null,
    "Config": [
      {
        "Subnet": "0.0.0.0/0",
        "Gateway": "0.0.0.0"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Containers": {},
  "Options": {
    "com.docker.network.windowsshim.hnsid": "4a57d4d8-29ad-40bb-9945-52d3d47894cf",
    "com.docker.network.windowsshim.networkname": "nat"
  },
  "Labels": {}
}
```

- Launch a new container by running a command “docker run -d microsoft/nanoserver ping -t localhost”. Once the container is running execute the command “docker network 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": {
  "2d60e9e3726e0f479c6c7e42af56d630ff182d5ccd35b67c7a14cfb42e2c0e76": {
    "Name": "loving_brown",
    "EndpointID": "7406c94b7b95b288fbb010c4286f6c23402ffb1217f3f379698d73b32b7ae032",
    "MacAddress": "00:15:5d:5d:83:d1",
    "IPv4Address": "172.28.74.118/16",
    "IPv6Address": ""
  }
}
```

- 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 . . . . . : 

```

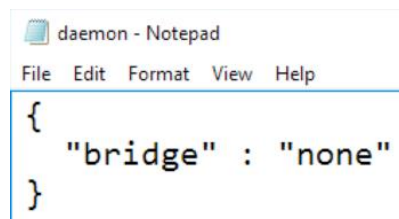
2. Creating Custom Docker NAT Network

Docker allows you to create custom NAT networks. In this task you will create and configure a custom NAT network replacing the default nat network.

1. Launch the PowerShell Console (if not already running) and run the command “net stop docker”. This will stop the docker service on the virtual machine. You must stop the docker service as later configuration won’t be possible otherwise.
2. Run the command “Get-ContainerNetwork | Remove-ContainerNetwork - Force”. This will remove the default NAT network.
3. Docker provides “daemon.json” configuration file which you need to modify. These files may not exist, so create it first by running command
“New-Item -ItemType file -Path C:\ProgramData\docker\config - Name daemon.json”
4. Disabling automatic NAT network creation and using the custom NAT is controlled by the “bridge” option in the daemon.json configuration file. First, open the file in notepad by running the command:

“notepad C:\ProgramData\docker\config\daemon.json”

Add the following JSON text to the file and then save and close it.



```

{
  "bridge" : "none"
}

```

5. Start the Docker service by running the command “net start docker”.
6. Create a new docker network 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.

- Use the “docker network ls” command and notice that “custom-nat” network is available.

NETWORK ID	NAME	DRIVER
ade6d4077f4d	custom-nat	nat
063a0dba81bd	none	null

- To use the new custom nat network for containers launch a new container by using the command:
“docker run -d --network=custom-nat microsoft/nanoserver ping -t localhost”

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

- Now, use the “docker network 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 (192.168.15.224, this may be different in your case) is in the custom-nat network.

```
{
  "Name": "custom-nat",
  "Id": "ade6d4077f4d62a95563ee0c137ef0a8a6aff2fb4fb6934d693930755199282b",
  "Created": "2017-08-19T22:54:02.6548304Z",
  "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,
  "Containers": {
    "525a6550c13e09405e5a942f1779d23c44e06762458b85472dadf06a1e04cb3b": {
      "Name": "wizardly_clarke",
      "EndpointID": "0a6e3647a8d5f387ee50213256ae456433dfc78b590791a7db143dbad2a67b25",
      "MacAddress": "00:15:5d:2d:8d:ea",
      "IPv4Address": "192.168.15.224/24",
      "IPv6Address": ""
    }
  },
  "Options": {
    "com.docker.network.windowsshim.hnsid": "b4b60745-3879-49ff-a899-c4ec74307f96"
  },
  "Labels": {}
}
```

- 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
```

3. Removing Custom Docker NAT Network

In this task you will remove the custom nat network created in previous task. Docker has an inbuilt capability to create new default NAT network if it does not find one. This happens every time Docker daemon/engine starts.

1. First, you need to stop the docker service. This is done by running the command “net stop docker”.
2. Remove the existing container network using the command
“Get-ContainerNetwork | Remove-ContainerNetwork -Force”
3. You now remove the daemon.json file that you have created in the earlier task. To do that run the command
“Remove-Item C:\ProgramData\docker\config\daemon.json”
4. Start the docker service by running the command “net start docker”
5. Once the docker service is started you can now run the “Get-ContainerNetwork” to view the default nat network.

Notice the subnets column and the value {172.28.176.0/20} which is the default subnet that docker uses. This is, of course, different from the subnet that you used in previous task with the custom-nat network.

```

Name Id                               Subnets
---- --
nat  1720f74a-e7b7-4c96-a108-70821a084a8f {172.28.176.0/20}
```

6. You have successfully completed all the tasks in this exercise.

Exercise 4: Running Containers with Memory and CPU Constrains (optional)

By default, container has no resource constraints and can use as much of a given resource as the host's kernel scheduler will allow. Docker provides ways to control how much memory, CPU, or block IO a container can use, setting runtime configuration flags of the docker run command. In this exercise you will launch a container with the constrain on how much host memory and CPU it can use.

Tasks

1. Run container with Memory Limit

1. In this task, you will launch container with pre-defined memory limit, so container won't be able to consume host memory beyond the memory limit. Later, you will test the container memory limit by simulating higher memory consumption inside the container.
2. Before running the container open two new PowerShell Consoles. You will use one of these consoles to run the docker container and interact with it. The other console is used to watch the memory usage of the container.
3. Use one of the PowerShell Console to launch a new container with memory limit of 500 megabytes (MB) by running the command:

```
"docker run -it -m 500M -v C:/labs/module3/tools/:C:/tools/microsoft/windowsservercore powershell"
```

Notice the use of `-m` (or `--memory`) switch within the run command. The switch specifies the maximum amount of memory the container can use. In this case you are setting it to 500 M (where M= Megabytes, other valid options are B = Bytes, K= Kilobytes and G = Gigabytes. These are also not case sensitive).

The use of `-v` switch is not related to memory constrain but rather to share the tools folder on the host to the container. This folder has a Sysinternals tool ["TestLimit64.exe"](#) which you will be using next to test the container memory limit.

4. You should now have access to PowerShell console that is running inside the container. Run the `"hostname"` command and note down the name of the container. You will need it later in this task.

```
PS C:\> hostname
4931d1aa4ff0
```

5. To test the memory limit of container you will use the testlimit tool. Run the following command.

```
"C:\tools\testlimit64.exe -d -c 1024"
```

```

PS C:\> C:\tools\testlimit64.exe -d -c 1024

Testlimit v5.04 - test Windows limits
By Mark Russinovich - www.sysinternals.com

Leaking private bytes with touch (MB)...
Leaked 362 MB of private memory (362 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 6 MB of private memory (368 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 0 MB of private memory (368 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 0 MB of private memory (368 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.

```

Notice that the tool will start putting stress on available memory by trying to push the memory consumption on container to 1024 MB (1 GB). However, since container can't go beyond 500 MB the value of memory consumed will always be under 500 MB (the exact value of how much memory is used will vary but won't go beyond maximum available memory on container which is 500 MB)

6. Go back to the PowerShell Console on the host (this is the second console that you have opened earlier). Run the docker stats command:

```
"docker stats <<Container ID>>"
```

Replace the <<Container ID>> with hostname that you captured earlier in the task. This command gives you a live stream of various vital stats including memory, CPU, etc. directly from the container.

7. Notice the value under the column under "PRIV WORKING SET". This represents the memory usage by the container. This is the value that docker has constrained to the 500 MB, so it will not go higher than that.

CONTAINER	CPU %	PRIV WORKING SET
4931d1aa4ff0	0.00%	473.3 MiB

8. Now, to reclaim the memory occupied by the running tool. Go back to PowerShell Console that was used to run the container and press the key combination "CTRL+C". This will stop the tool and free the memory on the container used by this tool.
9. Go back on the PowerShell console on the host that is displaying the vital stats for the container. Notice that memory usage has dropped significantly.

CONTAINER	CPU %	PRIV WORKING SET
4931d1aa4ff0	0.00%	91.54 MiB

10. Hit "CTRL+C" to stop the docker stats command
11. In the other Powershell window, type "exit" to exit from the running container

2. Run Container with CPU Usage Limit

1. Like memory constrain you can also constrain to CPU usage by the container. By default, Docker does not apply any constrain on container CPU usage which essentially means container is free to utilize host CPU up to 100%. In this task you will put a limit on CPU utilization by the container.
2. Since modern machines have CPUs with multiple cores you will first determine number of cores available to the host virtual machine by running the command.

```
"Get-WmiObject -class Win32_processor | Select -ExpandProperty NumberOfCores"
```

Note down the output which reflect number cores available. In this case its 2 cores but please note that the value you see may be different.

```
PS C:\Users\super> Get-WmiObject -class Win32_processor | Select -ExpandProperty NumberOfCores
2
```

3. You will now launch a new container and limit its host CPU usage to ~50%. Make sure that you set have of the number of CPU that you have available here. If your machine just has 1 core, set the value to 0.5.

```
"docker run -it --cpus 1.0 -v C:/labs/module3/tools/:C:/tools/microsoft/windowsservercore powershell"
```

Notice the use of --cpu switch which will specify how much of the available CPU resources a container can use. For instance, in this instance the host machine has two CPUs and if you set -cpus 1.0, so the container will be guaranteed to be able to access at most one of the CPUs on the host.

4. Go back to the PowerShell Console on the host (this is the second console that you have opened earlier). Run the docker stats command:

```
"docker stats <<Container ID>>"
```

Replace the <<Container ID>> with hostname that you captured earlier in the task.

Notice the CPU utilization for the new container. It should be very low since it's not really doing any active CPU intensive work.

```
Administrator: Windows PowerShell
CONTAINER          CPU %               PRIV WORKING SET
dde60eea0779       0.00%              77.59 MiB
```


5. Go back to the PowerShell Console in the container and make sure that you are authorized to run PS1 scripts by running first the following command.
“Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass”
6. You will now test the CPU constrain of ~50% by stress test the CPU utilization on the container. Switch back to the PowerShell Console that you have used earlier to launch the container with CPU usage limit.

Execute the command:

“C:\tools\cpu-stress.ps1”

```
PS C:\> C:\tools\cpu-stress.ps1
Number of cores to target: 2
```

Id	Name	PSJobTypeName	State
1	Job1	BackgroundJob	Running
3	Job3	BackgroundJob	Running

This command executes the script on the container which stress test CPU targeting all cores available to the container. However, since you have put the constrain on CPU utilization the container will never able to consume more than 1 CPU.

To validate the CPU usage, go back to PowerShell Console displaying docker stats. Notice the container CPU utilization is ~50% and not 100%. The CPU utilization may be slightly lower or higher than 50% so it may not be exact.

```
Administrator: Windows PowerShell
```

CONTAINER	CPU %
35f05a8cb5a3	50.20%

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