

## Table of Contents

<b>Lab: Module 1 - Introduction to Containers .....</b>	<b>2</b>
Duration: 90 minutes .....	2
Prerequisite .....	2
Exercise 1: Running Your First Container .....	2
Exercise 2: Working with Docker Command Line Interface (CLI).....	9
Exercise 3: Building Custom Container Images with Dockerfile: NodeJS, Nginx, and ASP.NET Core 2.x.....	13
Exercise 4: Interaction with a Running Container .....	22
Exercise 5: Tagging .....	27

## Lab: Module 1 - Introduction to Containers

---

**Duration: 90 minutes**

### Prerequisite

You must have labs virtual machines running on the Learn on Demand (LOD) website and register an account on the LOD website.

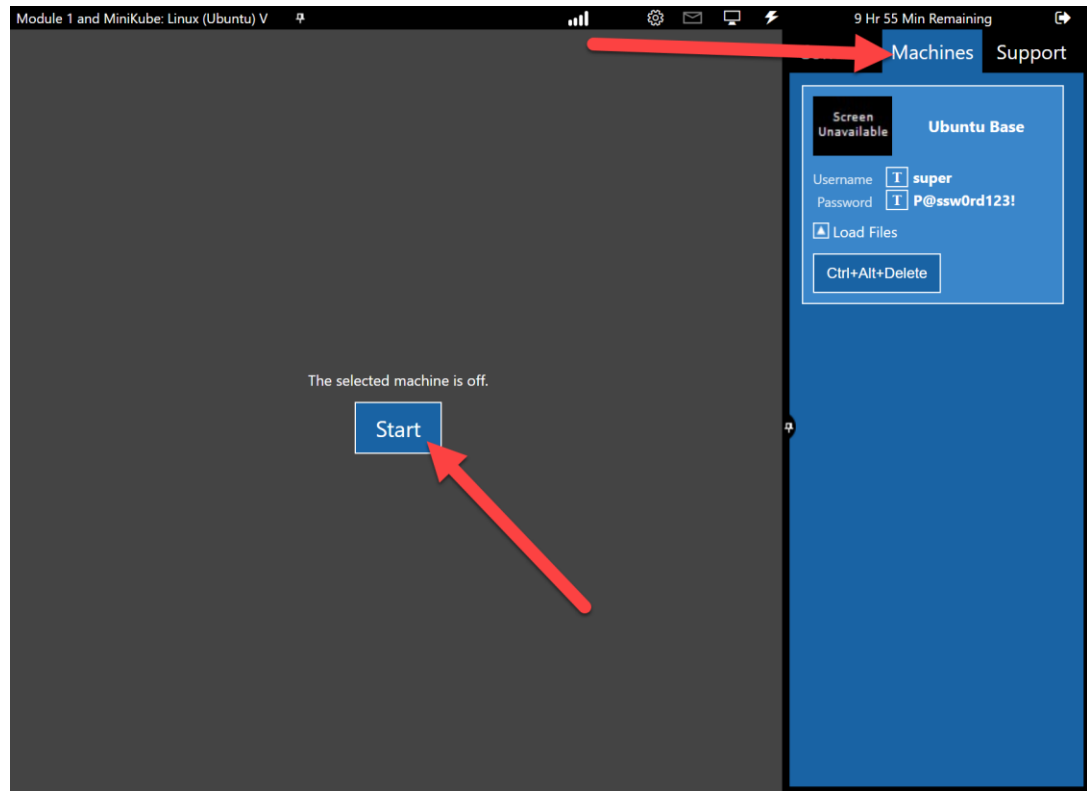
### Exercise 1: Running Your First Container

In this exercise, you will launch a fully functional WordPress blog engine using a Docker Linux container. You will learn the commands needed to pull the container image and then launch the container using Docker CLI. Finally, you will observe that you during the process of running the container you do not need to install any of the WordPress dependencies on the machine since container comes with both the WordPress engine binaries and all its dependencies pre-installed.

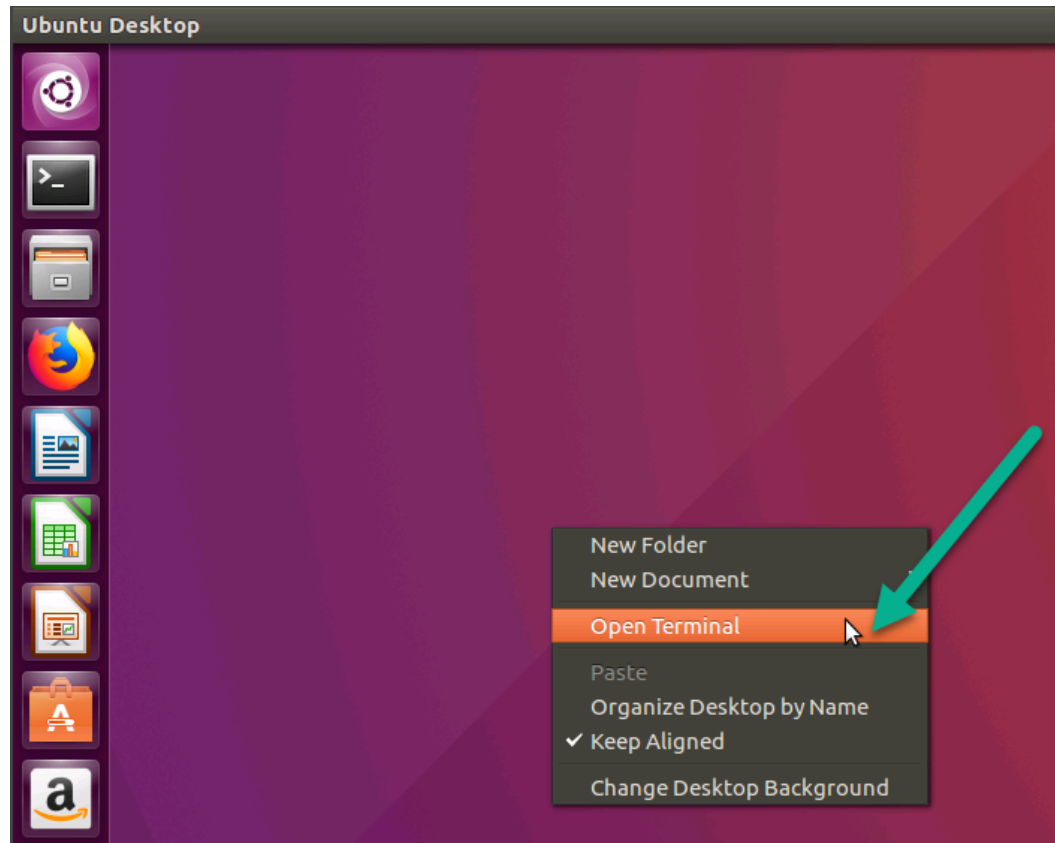
### Tasks

#### 1. Running WordPress Blog Engine Container

1. Go to the Lab on Demand (LOD) Linux Ubuntu Virtual Machine.
2. If the Virtual Machine is not running, Select Start on the VMsude. Click on the Machines tab on the right to get the password for the VM. You can click on the Machines tab at any time if you need to reference the password for the VM again.



3. To open a command line prompt, right click on the desktop and choose **open terminal**.



4. Run “`sudo -i`” to ensure all commands have elevated privileges. You will be prompted for your password, type it in, then press enter. You should see `root@super` in your command line as the user.

```
root@super-Virtual-Machine: ~  
super@super-Virtual-Machine:~$ sudo -i  
[sudo] password for super:  
root@super-Virtual-Machine:~#
```

5. Type “`docker pull tutum/wordpress`”. This will tell Docker client to connect to public Docker Registry and download the latest version of the WordPress container image published by tutum (hence the format `tutum/wordpress`). The container image has been pre-downloaded for you on the VM to save you a few minutes, but you will see each layer that is cached show the text ‘Already exists’.

docker

```
root@super-Virtual-Machine:~# docker pull tutum/wordpress
Using default tag: latest
latest: Pulling from tutum/wordpress
8387d9ff0016: Already exists
3b52deaaf0ed: Already exists
4bd501fad6de: Already exists
a3ed95cae02: Already exists
790f0e8363b9: Already exists
11f87572ad81: Already exists
341e06373981: Already exists
709079cecfb8: Already exists
55bf9bbb788a: Already exists
b41f3cfd3d47: Already exists
70789ae370c5: Already exists
43f2fd9a6779: Already exists
6a0b3a1558bd: Already exists
934438c9af31: Already exists
1cfba20318ab: Already exists
de7f3e54c21c: Already exists
596da16c3b16: Already exists
e94007c4319f: Already exists
3c013e645156: Already exists
6eaab85d8b00: Already exists
0e132cb1ce48: Already exists
1eba2a6e1fe2: Already exists
5a56e7332673: Already exists
d4166ef5fefb: Already exists
752b28beb2cc: Already exists
38aa0ae5e379: Already exists
Digest: sha256:2aa05fd3e8543b615fc07a628da066b48e6bf41ccee8f4b81e189de6eeda77
Status: Image is up to date for tutum/wordpress:latest
```

6. Run the command “docker images” and notice “tutum/wordpress” container image is now available locally for you to use.

```
root@super-Virtual-Machine:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
microsoft/aspnetcore 2.0                c4ca78cf9dca       2 weeks ago
325MB
tutum/wordpress     latest             7e7f97a602ff       2 years ago
477MB
root@super-Virtual-Machine:~#
```

7. That’s it! You can now run the entire WordPress in a container. To do that run the command  
“docker run -d -p 80:80 tutum/wordpress”. Pay close attention to the dash  
“-” symbol in front of “-p” and “-d” in the command.

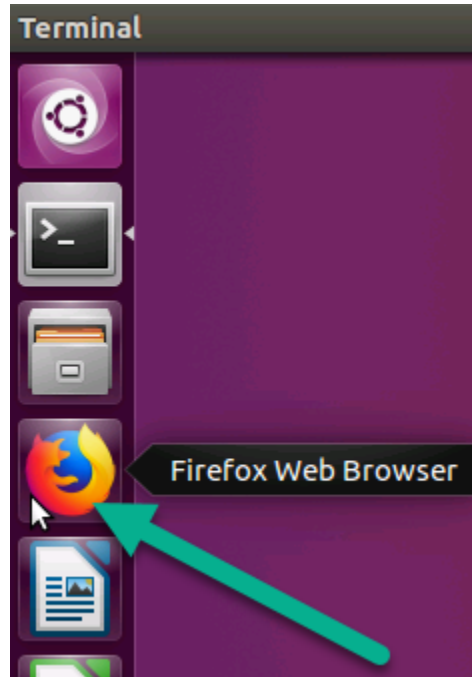
```
root@super-Virtual-Machine:~# docker run -d -p 80:80 tutum/wordpress
84acb95920718929d701f9a9c3ae87e0a154e56fda289a8bf0b90bae4e40f8b9
```

8. Run the following “docker ps” to see the running containers.

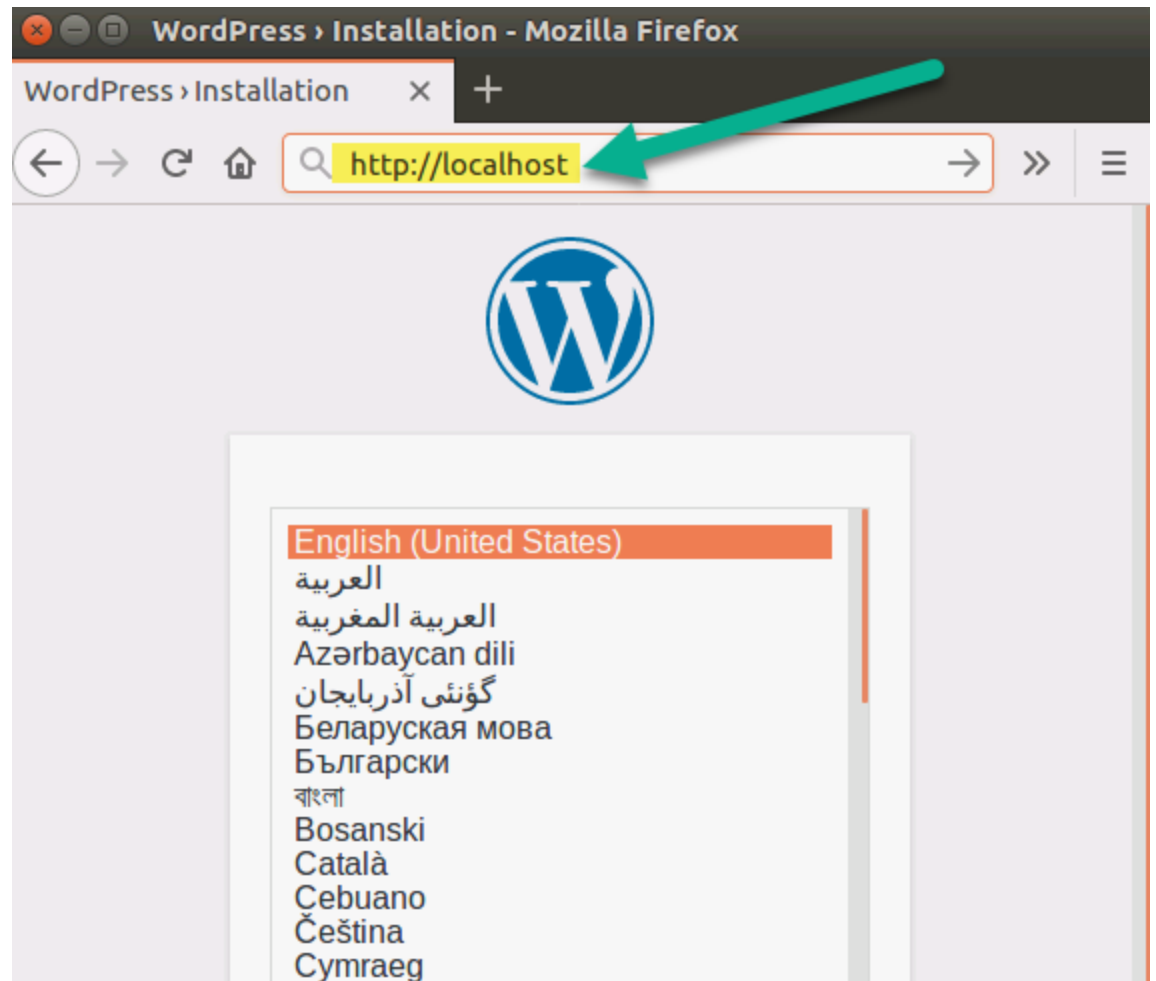
```
root@super-Virtual-Machine:~# docker ps
```

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED
84acb9592071	tutum/wordpress	"/run.sh"		3 minutes ago
Up 3 minutes	0.0.0.0:80->80/tcp,	3306/tcp	vibrant_joliot	

9. Click on the Firefox icon on the left:



10. Navigate to <http://localhost> and you should see WordPress.



11. Let's launch two more containers based on "tutum/wordpress" image. Execute following commands (one line at a time)

```
docker run -d -p 8080:80 tutum/wordpress
docker run -d -p 9090:80 tutum/wordpress
```

```
root@super-Virtual-Machine:~# docker run -d -p 8080:80 tutum/wordpress
b73631e40f498b4902202a2da2536feb0897ac9b5102d6c802b3d69dcaef42a8
root@super-Virtual-Machine:~# docker run -d -p 9090:80 tutum/wordpress
c272f02ac03d38d0f2b8a8935d1b85c892d2ca91fcfe6945aade707e2f5b6165
```

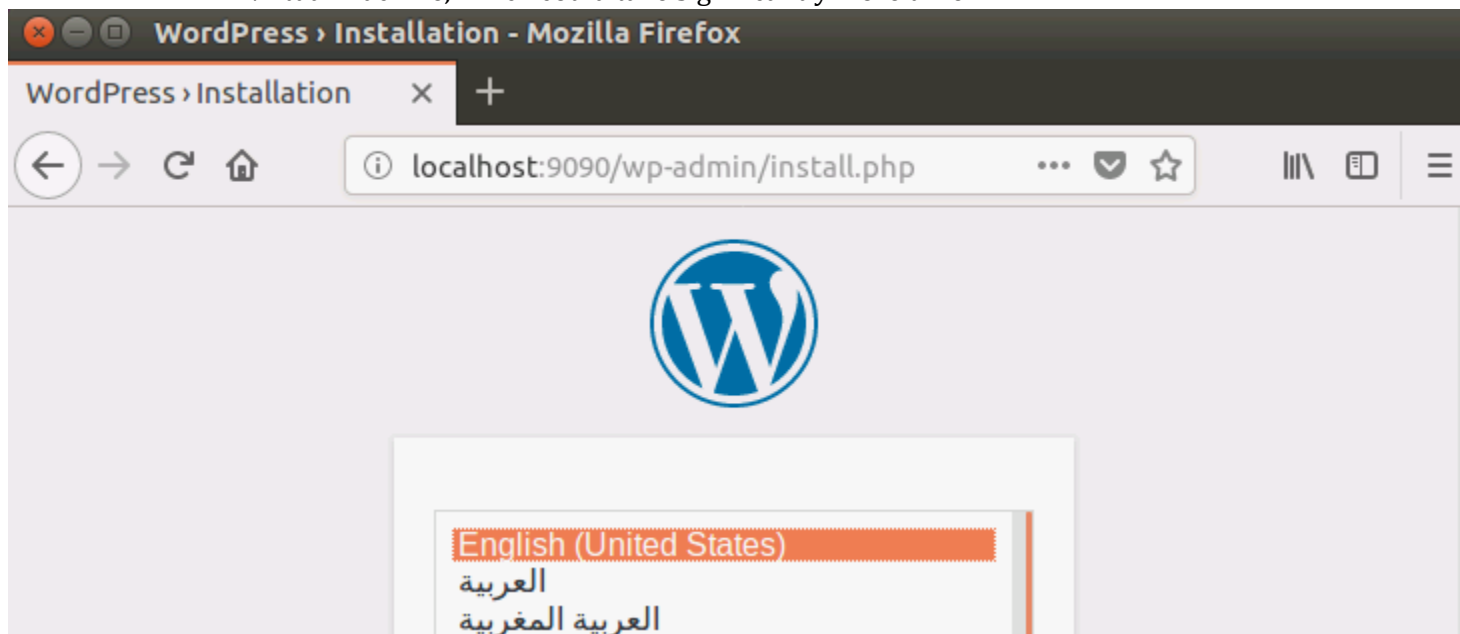
12. Run 'docker ps' to see all 3 running containers and their port numbers:

```

root@super-Virtual-Machine:~# docker ps
CONTAINER ID        IMAGE          COMMAND          CREATED
STATUS            PORTS         NAMES
c272f02ac03d      tutum/wordpress  "/run.sh"       2 minutes ago
Up 2 minutes      3306/tcp, 0.0.0.0:9090->80/tcp  priceless_feynman
b73631e40f49      tutum/wordpress  "/run.sh"       2 minutes ago
Up 2 minutes      3306/tcp, 0.0.0.0:8080->80/tcp  dazzling_engelbart
84acb9592071      tutum/wordpress  "/run.sh"       3 hours ago
Up 3 hours        0.0.0.0:80->80/tcp, 3306/tcp    vibrant_joliot

```

13. Now open a new browser window and navigate to URL (using DNS or IP as before) but with port “8080” append to it. You can also try port “9090”. Notice that you now have three WordPress blog instances running inside separate containers launched within few seconds. Contrast this to instead creating and running WordPress on virtual machine, which could take significantly more time.



14. If you want to run a container with a name, you can specify the parameter like this: ‘docker run --name mycontainer1 -d -p 8081:80 tutum/wordpress’. Run this on port 8081 so that it does not conflict with one of the previously running containers.

```

root@super-Virtual-Machine:~# docker run --name mycontainer1 -d -p 8081:80 tutum/wordpress
b72d657247f0388c311f0b22605e1321b164be367c135f1af1ed0a69b6d309c8

```

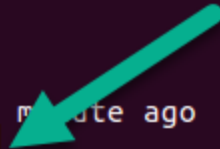
15. And, now if you run ‘docker ps’, you will see that the container has the name you assigned it using the --name parameter.



```

root@super-Virtual-Machine:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS        PORTS          NAMES
b72d657247f0   tutum/wordpress  "/run.sh"              About a minute ago
Up About a minute  3306/tcp, 0.0.0.0:8081->80/tcp  mycontainer1
8adb88bd6e0a   tutum/wordpress  "/run.sh"              2 minutes ago
Up 2 minutes     3306/tcp, 0.0.0.0:9090->80/tcp  mycontainer
b73631e40f49   tutum/wordpress  "/run.sh"              13 minutes ago
Up 13 minutes    3306/tcp, 0.0.0.0:8080->80/tcp  dazzling_engelbart
84acb9592071   tutum/wordpress  "/run.sh"              3 hours ago

```



## Exercise 2: Working with Docker Command Line Interface (CLI)

In this exercise, you will learn about common Docker commands needed to work with containers. A comprehensive list of docker commands are available at:

<https://docs.docker.com/engine/reference/commandline/docker>

### Tasks

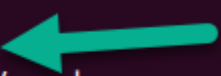
#### 1. Stopping Single Container

1. First list all the containers currently running by executing “docker ps” command. You should see list of all running containers. Notice, the list contains multiple containers based on WordPress image that you run in previous exercise.

```

root@super-Virtual-Machine:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS        PORTS          NAMES
c272f02ac03d   tutum/wordpress  "/run.sh"              2 minutes ago
Up 2 minutes    3306/tcp, 0.0.0.0:9090->80/tcp  priceless_feynman
b73631e40f49   tutum/wordpress  "/run.sh"              2 minutes ago
Up 2 minutes    3306/tcp, 0.0.0.0:8080->80/tcp  dazzling_engelbart
84acb9592071   tutum/wordpress  "/run.sh"              3 hours ago
Up 3 hours      0.0.0.0:80->80/tcp, 3306/tcp    vibrant_joliot

```



2. You can stop a running container by using “docker stop <<CONTAINER-ID>>” command. Where CONTAINER-ID is the identifier of a running container. **\*Note: You can just use the first couple characters to identity the container ID, such as “c27” for the screenshot below.**

```

root@super-Virtual-Machine:~# docker stop c27
c27

```

3. Now run the “docker ps” command and notice the listing show one less container running.

```

root@super-Virtual-Machine:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS            PORTS              NAMES
b73631e40f49       tutum/wordpress    "/run.sh"          5 minutes ago
Up 5 minutes       3306/tcp, 0.0.0.0:8080->80/tcp    dazzling_engelbart
84acb9592071       tutum/wordpress    "/run.sh"          3 hours ago
Up 3 hours         0.0.0.0:80->80/tcp, 3306/tcp      vibrant_joliot

```

4. If you want see the Container ID of the stopped container, and you forgot the Container ID, you can run 'docker ps -a' to see all containers, even ones that are stopped/exited.

```

root@super-Virtual-Machine:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS            PORTS              NAMES
c272f02ac03d       tutum/wordpress    "/run.sh"          7 minutes ago
Exited (137) About a minute ago    priceless_feynman
b73631e40f49       tutum/wordpress    "/run.sh"          7 minutes ago
Up 7 minutes       3306/tcp, 0.0.0.0:8080->80/tcp    dazzling_engelbart
2e9e3bddc8fe       tutum/wordpress    "/run.sh"          7 minutes ago
Created                               mystifying_ardinghelli
84acb9592071       tutum/wordpress    "/run.sh"          3 hours ago
Up 3 hours         0.0.0.0:80->80/tcp, 3306/tcp      vibrant_joliot
root@super-Virtual-Machine:~#

```

## 2. Restart a Container

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), you can also get this using 'docker ps -a'.
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.

```

root@super-Virtual-Machine:~# docker start c27
c27

```

3. To make sure that container has started successfully run "docker ps" command. Notice that WordPress container is now started.

```

root@super-Virtual-Machine:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS             PORTS              NAMES
b72d657247f0       tutum/wordpress    "/run.sh"          6 minutes ago
Up 6 minutes        3306/tcp, 0.0.0.0:8081->80/tcp    mycontainer1
c272f02ac03d       tutum/wordpress    "/run.sh"          18 minutes ago
Up 23 seconds       3306/tcp, 0.0.0.0:9090->80/tcp    priceless_feynman
b73631e40f49       tutum/wordpress    "/run.sh"          19 minutes ago
Up 19 minutes       3306/tcp, 0.0.0.0:8080->80/tcp    dazzling_engelbart
84acb9592071       tutum/wordpress    "/run.sh"          3 hours ago
Up 3 hours          0.0.0.0:80->80/tcp, 3306/tcp      vibrant_joliot

```

### 3. Removing a Container

1. Stopping a container does not remove it and that's the reason why you were able to start it again in the previous task.

To delete/remove a container and free the resources you need to issue a different command. Please note that this command does not remove the underlying image but rather the specific container that was based on the image. To remove the image and reclaim its resources, like disk space, you'll will need to issue a different command which is covered under the later section "Removing Container Image".

2. To remove a container, run "docker rm -f <<CONTAINER-ID>>" command. Replace the CONTAINER-ID with the container identifier you used in previous section. If you don't have it handy, simply run "docker ps" and copy the container ID from the listing.

```

root@super-Virtual-Machine:~# docker rm -f c27
c27

```

The -f switch is used to force the remove operation. It's needed if you are trying to remove a container that is not already stopped.

### 4. Stopping All Containers

1. At times you may want to stop all of the running containers and avoid issuing command to stop one container at a time. Run "docker stop \$(docker ps -aq)" command to stop all running containers. Basically, you are issuing two commands: First the docker ps with relevant switches to capture list of container IDs and then passing list of IDs to docker stop command.

do

```
root@super-Virtual-Machine:~# docker stop $(docker ps -aq)
b72d657247f0
8adb88bd6e0a
d2414edc37a0
3a235465566c
1bb0c6c5839e
5269d62ea056
763e22cd0765
b73631e40f49
2e9e3bddc8fe
84acb9592071
```

## 5. Removing WordPress Container Image

1. Removing a container image from a local system will let you reclaim its disk space. Please note that this operation is irreversible so proceed with caution. In this task you will remove the WordPress container image as you will not be using it any more. You must stop all containers using the image before you can delete the image, unless you use the force parameter.
2. To remove a container image, you'll need its IMAGE ID. Run command "docker images". Note down the IMAGE ID corresponding to the "tutum/wordpress" image that you will remove in next step.

```
root@super-Virtual-Machine:~# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
SIZE			
microsoft/aspnetcore	2.0	c4ca78cf9dca	2 weeks ago
325MB			
tutum/wordpress	latest	7e7f97a602ff	2 years ago
477MB			

3. Run the command "docker rmi <<IMAGE ID>> -f". Replace the IMAGE ID with the image identifier you captured in previous step. Notice the command to remove dockera container is "docker rm" and to remove an image is "docker rmi", with an 'i' for the image. Don't confuse these two commands! The -f is to force the removal, you cannot remove an image associated with a stopped container unless you use the force parameter.

```
root@super-Virtual-Machine:~# docker rmi 7e7f -f
Untagged: tutum/wordpress:latest
Untagged: tutum/wordpress@sha256:2aa05fd3e8543b615fc07a628da066b48e6bf41ccee8b8f4b81e189de6eeda77
Deleted: sha256:7e7f97a602ff0c3a30afaaac1e681c72003b4c8a76f8a90696f03e785bf36b90
```

4. Now, run the command "docker images". Notice that "tutum/wordpress" image is no longer available.

```

root@super-Virtual-Machine:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
microsoft/aspnetcore 2.0                c4ca78cf9dca       2 weeks ago
325MB

```

## Exercise 3: Building Custom Container Images with Dockerfile: NodeJS, Nginx, and ASP.NET Core 2.x

Dockerfile is essentially a plain text file with Docker commands in it to create a new image. You can think of it as a configuration file with set of instructions needed to assemble a new image. In this exercise you will learn common commands that goes into Dockerfile by creating custom images based on common technologies like NGINX and NodeJS.

### Tasks

#### 1. Building and Running Node.JS Application as Container

1. In this task you will create a new image based on the Node.js base image. You will start with a Dockerfile with instructions to copy the files needed to host a custom Node.js application, install necessary software inside the image and expose ports to allow the traffic. Later, you will learn how to build the image using Dockerfile and finally will run and test it out.

The relevant files related to a node.js application along with the Dockerfile are available inside the directory “labs/module1/nodejs”. You can get to that directory by using the “cd” command to navigate.

```

root@super-Virtual-Machine:~# ls
Desktop  labs
root@super-Virtual-Machine:~# cd labs/module1/nodejs/
root@super-Virtual-Machine:~/labs/module1/nodejs#

```

2. On the command prompt type “ls” and press Enter. Notice the available files include “server.js”, “package.json” and “Dockerfile”.

```

root@super-Virtual-Machine:~/labs/module1/nodejs# ls
Dockerfile  package.json  server.js

```

3. Let’s examine the Dockerfile by typing the command “nano Dockerfile” and press Enter (the file is case sensitive, so make sure the D in Dockerfile is capitalized). You can use any other text editor (for example vi etc. but instructions are provided for nano text editor). Notice the structure of Dockerfile.

```

GNU nano 2.5.3                               File: Dockerfile
# Simple Dockerfile for NodeJS

FROM node:boron

MAINTAINER Razi Rais

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 8080

CMD [ "npm", "start" ]

```

4. Move your cursor using the arrow keys to the line starting with “MAINTAINER Razi Rais” and change the text from that to the following: **LABEL maintainer="YourEmail@Email.com"**. Once finish making changes press “CTRL + X” and then press “Y” when asked for confirmation to retain your changes. Finally, you will be asked for file name to write. For that press Enter (without changing the name of the file). This will close the nano text editor.

```

File Name to Write: Dockerfile
^G Get Help          M-D DOS Format
^C Cancel            M-M Mac Format

```

5. You are now ready to build a new image based on the Dockerfile you just modified.
6. Run the command “docker build -t mynodejs .”  
(Pay close attention to the period that is at the end of command.) Notice how the build command is reading instructions from the Dockerfile starting from the top and executing them one at a time. This will take a few minutes to pull the image down to your VM.

```

root@super-Virtual-Machine:~/labs/module1/nodejs# docker build -t mynodejs .
Sending build context to Docker daemon 4.096kB
Step 1/9 : FROM node:boron
boron: Pulling from library/node
3d77ce4481b1: Downloading 1.072MB/54.26MB
534514c83d69: Downloading 6.459MB/17.58MB
d562b1c3ac3f: Downloading 2.639MB/43.25MB
4b85e68dc01d: Waiting
f6a66c5de9db: Waiting
7a4e7d9a081d: Waiting
0ac2388e12a8: Waiting
141249a1c8ee: Waiting

```

7. When it is complete, you will see a couple of npm warnings, these are expected.

```

+-- utils-merge@1.0.1
`-- vary@1.1.2

npm WARN docker_web_app@1.0.0 No repository field.
npm WARN docker_web_app@1.0.0 No license field.
Removing intermediate container dd13acf0e309
---> 1b6c6e3c4478
Step 7/9 : COPY . /usr/src/app
---> 69bfb79d4d02
Step 8/9 : EXPOSE 8080
---> Running in 128725d70616
Removing intermediate container 128725d70616
---> 803a61223f42
Step 9/9 : CMD [ "npm", "start" ]
---> Running in 806f848a8a0b
Removing intermediate container 806f848a8a0b
---> 086d7c3e9a9e
Successfully built 086d7c3e9a9e
Successfully tagged mynodejs:latest

```

8. Run the command “docker images” and notice the new container image appears with the name “mynodejs”. Also notice the presence of parent image “node” that was also pulled from Docker Hub during the build operation.

```

root@super-Virtual-Machine:~/labs/module1/nodejs# docker images

```

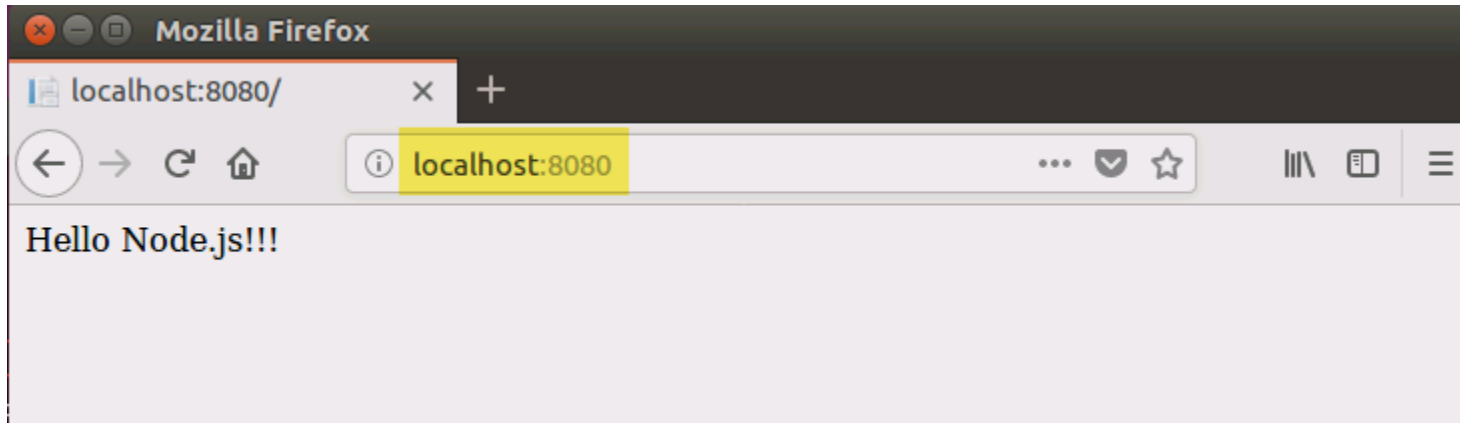
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynodejs	latest	086d7c3e9a9e	About a minute ago	664MB
node	boron	67ed1f028e71	44 hours ago	659MB
microsoft/aspnetcore	2.0	c4ca78cf9dca	2 weeks ago	325MB

9. Finally, lets create and run a new container based on “mynodejs” image. Run command “docker run -d -p 8080:8080 mynodejs”. (The “-d” parameter will run the container in the background, whereas the “-p” parameter publishes the containers ports to the host). Here are binding the port of the container (port number on right-side of colon) to the port of the host machine (port number on the left-side of the colon.)



```
root@super-Virtual-Machine:~/labs/module1/nodejs# docker run -d -p 8080:8080 mynodejs
81c95ab1d59c954a8da7d8aa145782456597e6e3884750519267fb12ab4ff108
```

10. To test the “mynodejs” application, go back to your Firefox browser and go to localhost:8080.



## 2. Building and Running NGINX Container

1. In this task you will create a new image using the NGINX web server base image hosting a simple static html page. You will start with a Dockerfile with instructions to define its base image, then copy the static html file inside the image and then specify the startup command for the image (using CMD instruction). Later, you will learn how to build the image using Dockerfile and finally will run and test it out.

The relevant files including static html file “index.html” along with the Dockerfile are available inside the directory “labs/module1/nginx”.

```
root@super-Virtual-Machine:~/labs/module1/nodejs# cd ..
root@super-Virtual-Machine:~/labs/module1# cd nginx
root@super-Virtual-Machine:~/labs/module1/nginx#
```

2. Type “ls” and press Enter. Notice the available files include “server.js” and “Dockerfile”.

```
root@super-Virtual-Machine:~/labs/module1/nginx# ls
Dockerfile  index.html
```

3. Let’s examine the Dockerfile by typing the command “nano Dockerfile” and press Enter. You can use any other text editor (for example, vi, etc.), but instructions are provided for nano text editor). Notice the structure of Dockerfile.



```
GNU nano 2.5.3 File: Dockerfile
# Simple Dockerfile for NGINX

FROM nginx:stable-alpine

MAINTAINER Razi Rais

COPY index.html /usr/share/nginx/html/index.html

CMD ["nginx", "-g", "daemon off;"]
```

4. Move your cursor by using the arrow keys to the line starting with “MAINTAINER” and change the text from “Razi Rais” to your name (or any other text of your liking). Once finish making changes press “CTRL + X” and then press “Y” when asked for confirmation to retain your changes. Finally, you will be asked for file name to write. For that press Enter (without changing the name of the file). This will close the nano text editor.

```
File Name to Write: Dockerfile
^G Get Help M-D DOS Format
^C Cancel M-M Mac Format
```

5. You are now ready to build a new container image based on the Dockerfile you just modified.  
Run the command “docker build -t mynginx .”

```

root@super-Virtual-Machine:~/labs/module1/nginx# docker build -t mynginx .
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM nginx:stable-alpine
stable-alpine: Pulling from library/nginx
550fe1bea624: Pull complete
af3988949040: Pull complete
d6642feac728: Pull complete
c20f0a205eaa: Pull complete
Digest: sha256:db5acc22920799fe387a903437eb89387607e5b3f63cf0f4472ac182d7bad644
Status: Downloaded newer image for nginx:stable-alpine
--> 24ed1c575f81
Step 2/4 : MAINTAINER Razi Rais
--> Running in da8d083bc585
Removing intermediate container da8d083bc585
--> 1c76d5ee749d
Step 3/4 : COPY index.html /usr/share/nginx/html/index.html
--> 8af1eda2a414
Step 4/4 : CMD ["nginx", "-g", "daemon off;"]
--> Running in 03c881e456d9
Removing intermediate container 03c881e456d9
--> d1a6647bc6d5
Successfully built d1a6647bc6d5
Successfully tagged mynginx:latest

```

*Notice how the build command is reading instructions from the Docker file starting from the top and executing them one at a time. The image will download much faster as this is a very small image.*

6. If you want to see the layers of an image, you can do 'docker history mynginx' and see the one you just built. You can also try running this command on other images you have on your VM too.

```

root@super-Virtual-Machine:~/labs/module1/nginx# docker history mynginx
IMAGE                CREATED              CREATED BY          SIZE
COMMENT
d1a6647bc6d5        About a minute ago  /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon...  0B
8af1eda2a414        About a minute ago  /bin/sh -c #(nop)  COPY file:e4327ecf14628102...  65B
1c76d5ee749d        About a minute ago  /bin/sh -c #(nop)  MAINTAINER Razi Rais        0B
24ed1c575f81        3 months ago       /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon...  0B
<missing>           3 months ago       /bin/sh -c #(nop)  STOPSIGNAL [SIGTERM]        0B
<missing>           3 months ago       /bin/sh -c #(nop)  EXPOSE 80/tcp                0B
<missing>           3 months ago       /bin/sh -c #(nop)  COPY file:1d1ac3b9a14c94a7...  1.09
kB
<missing>           3 months ago       /bin/sh -c #(nop)  COPY file:af94db45bb7e4b8f...  643B
<missing>           3 months ago       /bin/sh -c GPG_KEYS=B0F4253373F8F6F510D42178...  11.5
MB
<missing>           3 months ago       /bin/sh -c #(nop)  ENV NGINX_VERSION=1.12.2    0B
<missing>           3 months ago       /bin/sh -c #(nop)  LABEL maintainer=NGINX Do...  0B
<missing>           3 months ago       /bin/sh -c #(nop)  CMD ["/bin/sh"]             0B
<missing>           3 months ago       /bin/sh -c #(nop)  ADD file:df48d6d6df42a0138...  3.99
MB

```

- Run the command “docker images” and notice the new container image appears with the name “mynginx”. Also notice the presence of parent image “nginx” that was pulled from Docker Hub during the build operation. Take a look at the sizes of different images also—this will become important when you build your own custom images to reduce the size for both security and performance.

```

root@super-Virtual-Machine:~/labs/module1/nginx# docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
mynginx             latest         d1a6647bc6d5   2 minutes ago  15.5MB
mynodejs            latest         086d7c3e9a9e   8 minutes ago  664MB
node                boron          67ed1f028e71   45 hours ago   659MB
microsoft/aspnetcore 2.0            c4ca78cf9dca   2 weeks ago    325MB
nginx               stable-alpine   24ed1c575f81   3 months ago   15.5MB

```

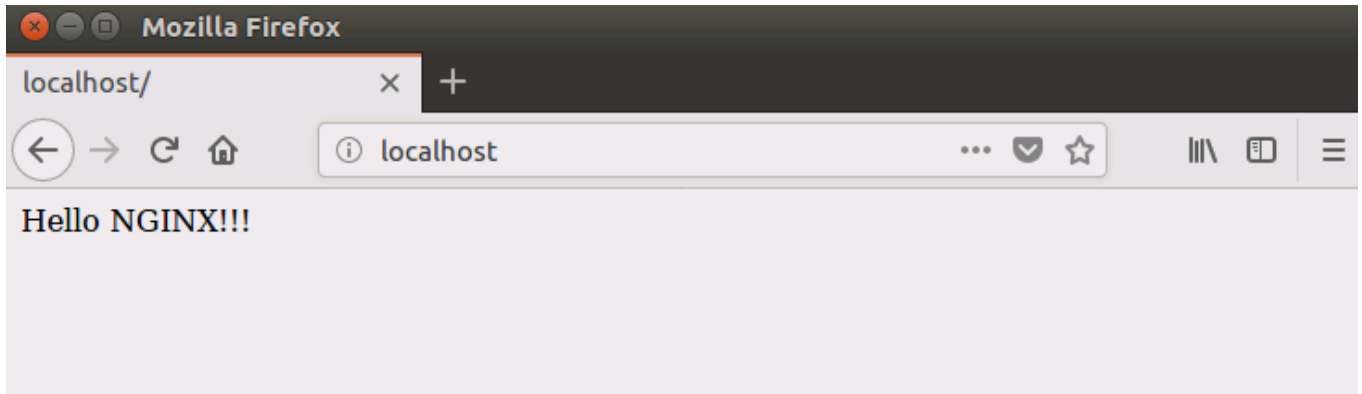
- Finally, create and run a new container based on “mynginx” image. Run command “docker run -d -p 80:80 mynginx”.

```

root@super-Virtual-Machine:~/labs/module1/nginx# docker run -d -p 80:80 mynginx
6a421269b708e8c81197883fb6e7766c8008418cd061d1555d0fc9d0971081a1

```

- To test the node app, go to your Firefox browser and go to localhost.



### 3. Building and Running ASP.NET Core 2.x Application Inside Container

1. In this demo you will build ASP.NET Core 2.x application and then package and run it as a container. Change to the relevant directory

“labs/module1/aspnetcore”:

First, we need to run **dotnet build**, and **publish** to generate the binaries of our application. It can be done either manually or automated as **Dockerfile** instructions. In our case, we will do it manually from the Linux machine to produce the artifacts in a **published** folder. The **Dockerfile** will just contain instructions to copy the files from the **published** folder into the image.

“dotnet build”

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# ls
appsettings.Development.json  bin          Controllers  Models      obj          Properties  Views
appsettings.json             bundleconfig.json  Dockerfile  mywebapp.csproj  Program.cs  Startup.cs  wwwroot
root@super-Virtual-Machine:~/labs/module1/aspnetcore# dotnet build
Microsoft (R) Build Engine version 15.6.82.30579 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 192.24 ms for /root/labs/module1/aspnetcore/mywebapp.csproj.
Restore completed in 63.29 ms for /root/labs/module1/aspnetcore/mywebapp.csproj.
mywebapp -> /root/labs/module1/aspnetcore/bin/Debug/netcoreapp2.0/mywebapp.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:07.45
```

“dotnet publish -o published”

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# dotnet publish -o published
Microsoft (R) Build Engine version 15.6.82.30579 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 109.74 ms for /root/labs/module1/aspnetcore/mywebapp.csproj.
Restore completed in 34.46 ms for /root/labs/module1/aspnetcore/mywebapp.csproj.
mywebapp -> /root/labs/module1/aspnetcore/bin/Debug/netcoreapp2.0/mywebapp.dll
mywebapp -> /root/labs/module1/aspnetcore/published/
root@super-Virtual-Machine:~/labs/module1/aspnetcore# ls
appsettings.Development.json  bundleconfig.json  Models      Program.cs  Startup.cs
appsettings.json             Controllers        mywebapp.csproj  Properties  Views
bin                          Dockerfile        obj          published   wwwroot
```

2. Now that application is ready you will create a container image for it. You are provided with a Dockerfile. View the content of Dockerfile by running a command “nano Dockerfile”. To exit the editor press “CTRL+X”. The Dockerfile contents should match the screenshot below:

```
GNU nano 2.5.3 File:
FROM microsoft/aspnetcore:2.0
WORKDIR /app
COPY published ./
ENTRYPOINT ["dotnet", "mywebapp.dll"]
```

3. To create the container image run the command  
"docker build -t myaspcoreapp:2.0 ."  
Notice the use of tag "2.0" that signifies the use of framework version of dotnet core.

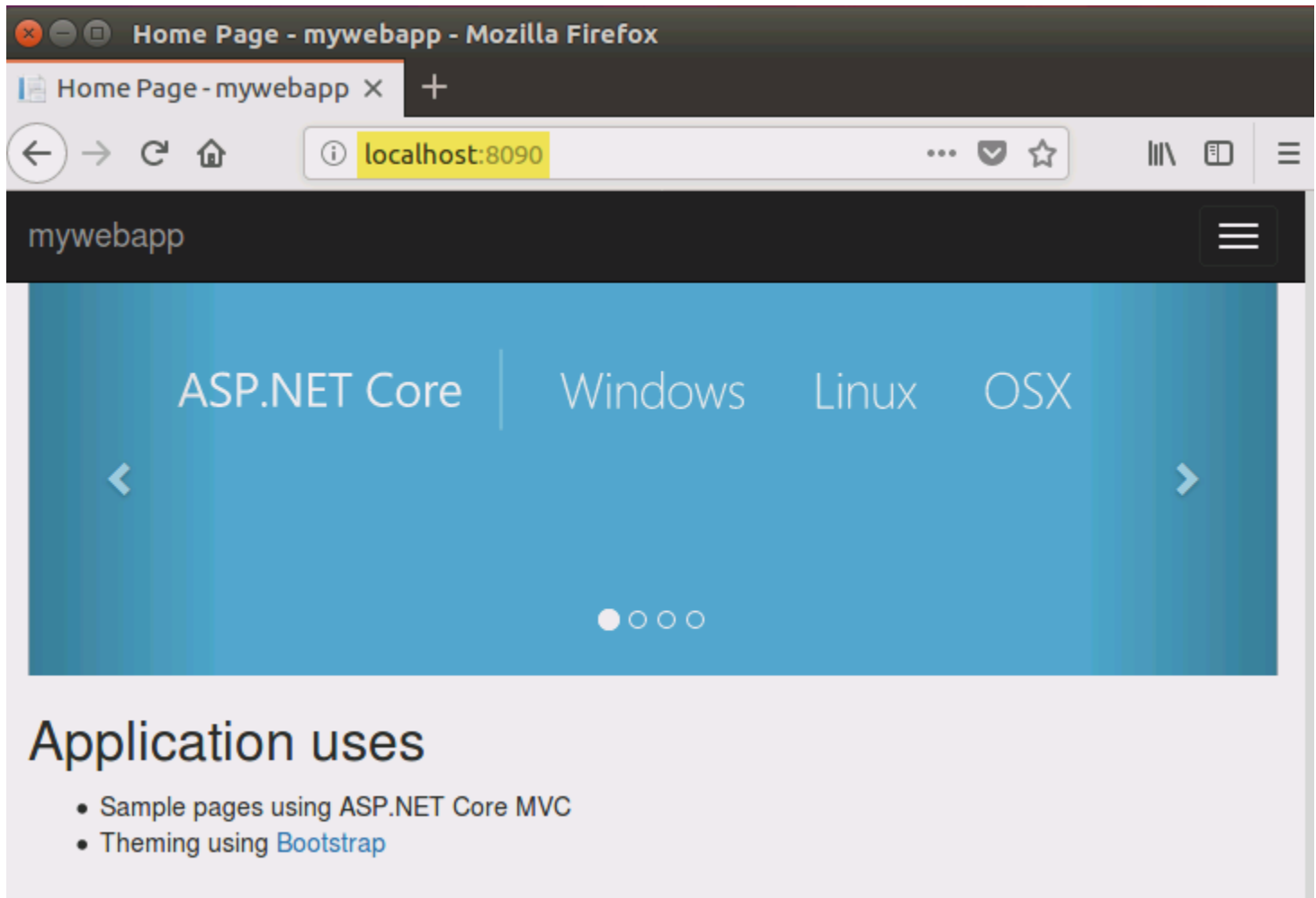
```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker build -t myaspcoreapp:2.0 .
Sending build context to Docker daemon 6.163MB
Step 1/4 : FROM microsoft/aspnetcore:2.0
--> c4ca78cf9dca
Step 2/4 : WORKDIR /app
Removing intermediate container f92a79dcdd93
--> 5304b3e36cbe
Step 3/4 : COPY published ./
--> 05bacf187dd4
Step 4/4 : ENTRYPOINT ["dotnet", "mywebapp.dll"]
--> Running in f4f6910fecdd6
Removing intermediate container f4f6910fecdd6
--> c3732185decc
Successfully built c3732185decc
Successfully tagged myaspcoreapp:2.0
```

4. Launch the container running the app inside it by running the command  
"docker run -d -p 8090:80 myaspcoreapp:2.0"

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker run -d -p 8090:80 myaspcoreapp:2.0
c2d5bc7e58b1eedaabb689a91f688411836a1943a5cc1dd5f1a43f9d8dd7549e5
```

You are now running ASP.NET Core application inside the container listening at port 80 which is mapped to port 8090 on the host.

5. To test the application, go to localhost:8090 in your Firefox browser.



## Exercise 4: Interaction with a Running Container

In the previous exercise you built and ran containers based on Dockerfile. At times you may want to interact with a running container for the purposes of troubleshooting, monitoring etc. You may also want to make changes/updates to a running container and then build a new image from it. Let's see first how to interact with a running container and then make changes to it and then persist it as a new image.

### Tasks

#### 1. Interaction with a Running Container

1. On the command line run "docker ps" to list all the currently running containers on your virtual machine.



```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
c2d5bc7e58b1	myaspcoreapp:2.0	"dotnet mywebapp.dll"	7 minutes ago	Up 7 min
6a421269b708	mynginx	"nginx -g 'daemon of...'"	13 minutes ago	Up 13 mi
81c95ab1d59c	mynodejs	"npm start"	24 minutes ago	Up 2 sec

Notice that multiple containers are running. To establish interactive session with a running container you will need its **CONTAINER ID or NAME**. Let's establish an interactive session to a container based on "**mynodejs**" image. Please note that your **CONTAINER ID or NAME** will probably be different. And, unless you specified a name, Docker came up with a random adjective and noun and smushed them together to come up with its own clever name.

- Run a command "docker exec -it <<CONTAINER ID OR NAME>> bash" on your **mynodejs container**. (docker exec is used to run a command in a running container. The "it" parameter will invoke an interactive bash shell on the container.) Notice that a new interactive session is now established to a running container. Since "bash" is the program that was asked to be executed you now have access to full bash shell inside the container.

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker exec -it 81c bash
root@81c95ab1d59c:/usr/src/app#
```

- You can run a command "ls" to view the listing of files and directories. Notice it has all the files copied by Dockerfile command in previous section.

```
root@81c95ab1d59c:/usr/src/app# ls
Dockerfile node_modules package.json server.js
```

NOTE: For more information regarding running commands inside docker container please visit:

<https://docs.docker.com/engine/reference/commandline/exec>

## 2. Making Changes to a Running Container

While you are interacting and running commands inside a running container, you may also want to make changes/updates to it. Later, you may also create a brand-new image out of these changes. In this task you will make changes to "mynodejs" container image, test them out and finally create a new image (without the need of Dockerfile). Please note that this approach of creating container images is generally used to quickly test various changes, but the recommend best practice way to create container images is to use Dockerfile.

First, you will make updates to "server.js" file. You should have active session already established from previous section (if not then please follow the instructions from previous section to create an active session now).

Before we can edit the “server.js” file we need to install a text editor. Basically, to keep the size of container down to absolute minimum, the “nodejs” container image does not have any extra software installed on the container. This is a common theme when building images and is also recommend practice.

1. Before installing any software run the command “apt-get update” (Note the dash between “apt” and “-get”.)

```
root@81c95ab1d59c:/usr/src/app# apt-get update
Get:1 http://security.debian.org jessie/updates InRelease [94.4 kB]
Ign http://deb.debian.org jessie InRelease
Get:2 http://deb.debian.org jessie-updates InRelease [145 kB]
Get:3 http://deb.debian.org jessie Release.gpg [2434 B]
Get:4 http://deb.debian.org jessie Release [148 kB]
Get:5 http://security.debian.org jessie/updates/main amd64 Packages [650 kB]
38% [2 InRelease gpgv 145 kB] [5 Packages 2440 B/650 kB 0%] 6873 B/s 1min 34s
```

2. To install “nano” run a command “apt-get install nano”

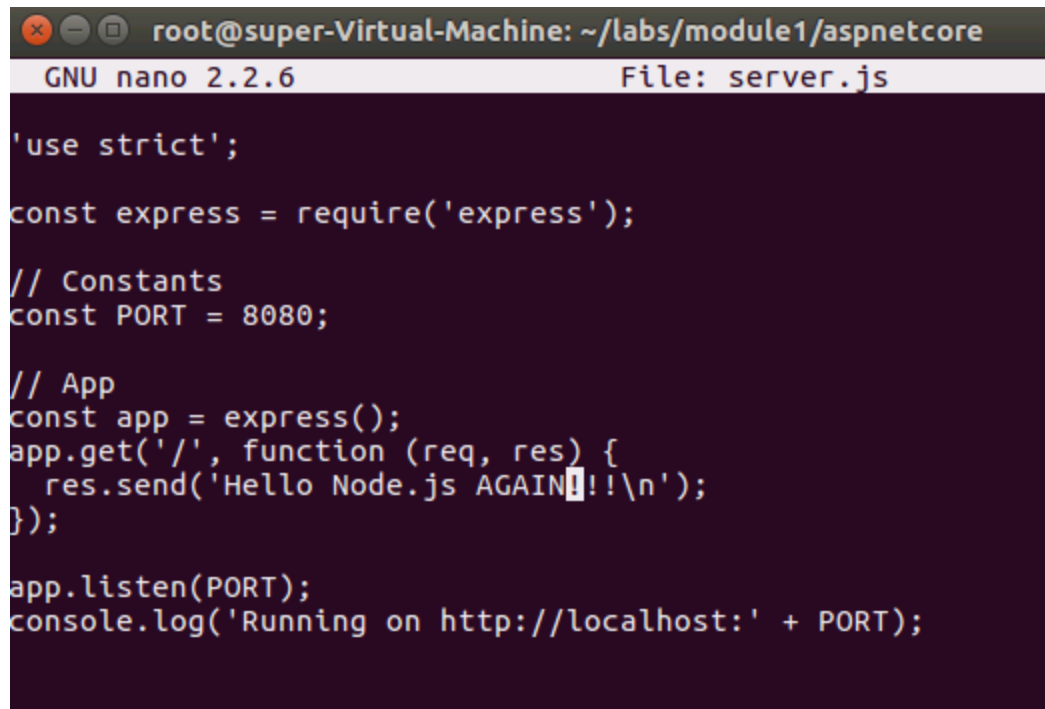
```
root@81c95ab1d59c:/usr/src/app# apt-get install nano
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  spell
The following NEW packages will be installed:
  nano
0 upgraded, 1 newly installed, 0 to remove and 1 not upgraded.
Need to get 369 kB of archives.
After this operation, 1707 kB of additional disk space will be used.
```

3. After “nano” is installed successfully, run the command “nano server.js” to open “server.js” file for editing.

```
root@81c95ab1d59c:/usr/src/app# nano server.js
```

4. Use the arrow keys to go to the line starting with “res.Send(...)” and update the text from “Hello Node.js!!!” to “Hello Node.js AGAIN!!!”. Your final changes should look like following:





```

root@super-Virtual-Machine: ~/labs/module1/aspnetcore
GNU nano 2.2.6 File: server.js

'use strict';

const express = require('express');

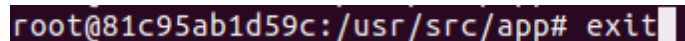
// Constants
const PORT = 8080;

// App
const app = express();
app.get('/', function (req, res) {
  res.send('Hello Node.js AGAIN!!!\n');
});

app.listen(PORT);
console.log('Running on http://localhost:' + PORT);

```

5. Once you finish making changes press “CTRL + X” and then press “Y” when asked for confirmation to retain changes. Finally, you will be asked for file name to write. For that press enter (without changing the name of the file). This will close the Nano text editor.
6. To save the updates and exit the interactive bash session, run the command “exit”

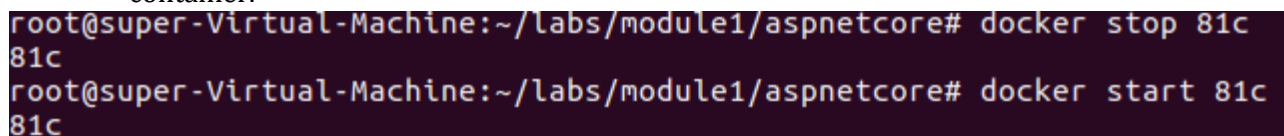


```

root@81c95ab1d59c:/usr/src/app# exit

```

7. The running container needs to be stopped first and then started again to reflect the changes. Run the command “docker stop <<CONTAINER ID>>” to stop the container. Run the command “docker start <<CONTAINER ID>>” to start the container.

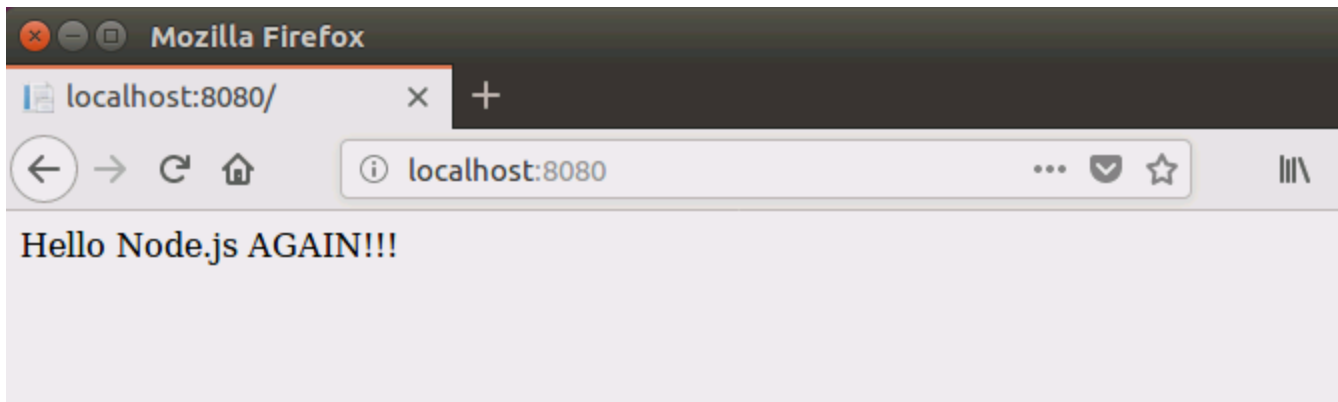


```

root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker stop 81c81c
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker start 81c81c

```

8. Finally, to test the update you have made to the container go to Firefox and localhost:8080. Notice the output “Hello Node.js AGAIN!!!”. This verifies that changes to the container were persisted.



### 3. Interaction with a Running Container

In the task you have made changes to running container. However, these changes are only available to that container and if you were to remove the container, these changes would be lost. One way to address this is by creating a new container image based on running container that has the changes. This way changes will be available as part of a new container image. This is helpful during dev/test phases where rapid development and testing requires a quick turn-around time. However, this approach is generally not recommended as it's hard to manage and scale at production level. Also, if content is the only piece that needs to be changed and shared, then using “volumes” may be another viable option. Volumes are covered in later module.

1. To create new image run the command “`docker commit <<CONTAINER ID>> mynodejsv2`”. (The `docker commit` command is used to create a new image from a container's changes.) If you don't already have it, you can use “`docker ps`” command to get the list of running containers or “`docker ps -a`” to get list of all the containers that are stopped and capture the CONTAINER ID of the container you have updated in previous section.

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker commit 81c mynodejsv2
sha256:f3215f52d6859b51046c9eb6da51699b4a64dcdf0709b860fcfb4d1fbfd38d47
```

2. Now, view the list of all container images by running the command “`docker images`” and notice the availability of new image with name “mynodejsv2”

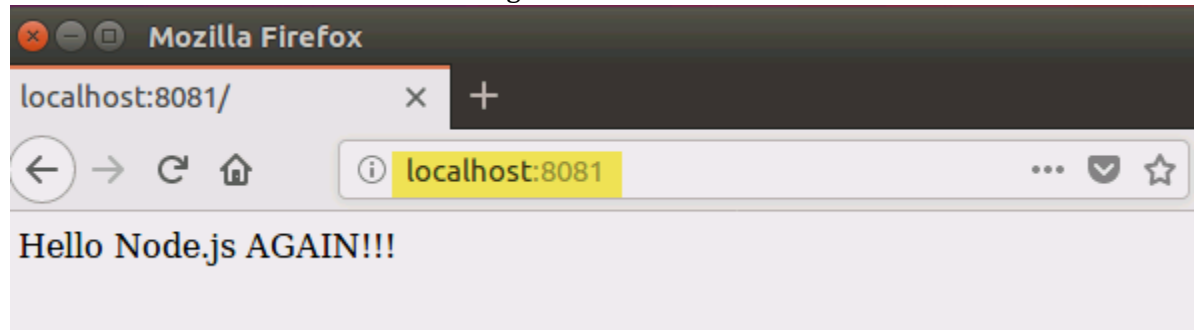
```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
mynodejsv2           latest          f3215f52d685   4 seconds ago   676MB
myaspcoreapp         2.0            c3732185decc   38 minutes ago   328MB
mynginx              latest         d1a6647bc6d5   About an hour ago 15.5MB
mynodejs             latest         086d7c3e9a9e   About an hour ago 664MB
node                 boron          67ed1f028e71   45 hours ago    659MB
microsoft/aspnetcore 2.0            c4ca78cf9dca   2 weeks ago     325MB
nginx                stable-alpine   24ed1c575f81   3 months ago    15.5MB
```

You now have a container image with the changes you made and tested earlier and is ready to be used.

3. To test the new image run a command “`docker run -d -p 8081:8080 mynodejsv2`”. This will create a new container based on the image “mynodejsv2”.

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker run -d -p 8081:8080 mynodejsv2
2e1838a129a7011bfe2b7c85f5aa2620d79796cdff38144da4dfabe79305cf8d
```

4. Finally, to test the container, go to localhost:8081 in Firefox. Notice the text “Hello Node.js AGAIN!!!” is returned from the node.js application. This attest that changes were committed properly to new image and hence available to any container created based on the that image.



## Exercise 5: Tagging

In this exercise you will learn the role of tagging in container and how to tag new and existing container images using Docker commands.

### Tasks

#### 1. Tagging Existing Container Image

In this task you will tag “mynodejs” container image with “v1”. Recall from last task that currently this image has “latest” tag associated with it. You can simply run “`docker images`” to verify that. When working with container images it becomes important to provide consistent versioning information.

Tagging provides you with the ability to tag container images properly at the time of building a new image using the “`docker build -t imagename:tag.`” command and then refer to image (for example inside Dockerfile with FROM statement) using a format “image-name:tag”.

Basically, if you don’t provide a tag, Docker assumes that you meant “latest” and use it as a default tag for the image. It is not good practice to make images without tagging them. **You’d think you could assume latest = most recent image version always? Wrong. That’s not true at all. Latest is just the tag which is applied to an image by default which does not have a tag.** If you push a new image with a tag which is neither empty nor ‘latest’, :latest will not be affected or created. Latest is also easily overwritten by default if you forget to tag something again in the future. Careful!!!

When you run “docker images” notice the “TAG” column and pay attention to the fact that for all the custom images created in the lab so far have tag value of “latest”.

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynodejsv2	latest	f3215f52d685	9 minutes ago	676MB
myaspcoreapp	2.0	c3732185decc	About an hour ago	328MB
mynginx	latest	d1a6647bc6d5	About an hour ago	15.5MB
mynodejs	latest	086d7c3e9a9e	About an hour ago	664MB
node	boron	67ed1f028e71	45 hours ago	659MB
microsoft/aspnetcore	2.0	c4ca78cf9dca	2 weeks ago	325MB
nginx	stable-alpine	24ed1c575f81	3 months ago	15.5MB

To understand importance of tagging take a look at container image created in previous section “mynodejsv2”. The “v2” at the very end was appended to provide an indicator that this is the second version of the image “mynodejs”. The challenge with this scheme is that there is no inherent connection between the “mynodejs” and “mynodejsv2”. With tagging, the same container image will take the format “mynodejs:v2”. This way you are telling everyone that “v2” is different but it does have relation with “mynodejs” container image.

Please note that tags are just strings. So, any string including “v1”, “1.0”, “1.1”, “1.0-beta”, and “banana” all qualify as a valid tag.

However, you should always want to follow consistent nomenclature when using tagging to reflect versioning. This is critical because when you start developing and deploying containers into production you may want to roll back to previous versions in a consistent manner. Not having a well-defined scheme for tagging will make it very difficult particularly when it comes to troubleshooting containers.

NOTE: A good example of various tagging scheme chosen by Microsoft with dotnet core framework is available at:

<https://hub.docker.com/r/microsoft/dotnet/tags>

1. To tag an existing docker image, run the command “docker tag <<IMAGE ID or IMAGE NAME>> mynodejs:v1”. Replace the IMAGE ID with the image id of “mynodejs” container image. To see the updated tag for “mynodejs” image run the command “docker images”.

```

root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker tag mynodejs mynodejs:v1
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynodejsv2	latest	f3215f52d685	11 minutes ago	1676MB
myaspcoreapp	2.0	c3732185decc	About an hour ago	328MB
mynginx	latest	d1a6647bc6d5	About an hour ago	15.5MB
mynodejs	latest	086d7c3e9a9e	About an hour ago	664MB
mynodejs	v1	086d7c3e9a9e	About an hour ago	664MB
node	boron	67ed1f028e71	46 hours ago	659MB
microsoft/aspnetcore	2.0	c4ca78cf9dca	2 weeks ago	325MB
nginx	stable-alpine	24ed1c575f81	3 months ago	15.5MB

Notice how “latest” and “v1” both exist. V1 is technically newer, and latest just signifies the image that did not have a version/tag before and can feel misleading. Also, note the Image ID for both are identical. The image and its content / layers are all cached on your machine. The Image ID is content addressable, so the full content of it is hashed through a hashing algorithm and it spits out an ID. If the content of any two (or more) images are the same, then the Image ID will be the same, and only one copy of the actual layers are on your machine and pointed to by many different image names/tags.

## 2. Tagging New Container Image

1. Tagging a new image is done at the time when you build a container image. it's a straightforward process that requires you to simply add the “:tag” at the end of container image name.
2. Navigate to the directory “labs/module1/nginx” that contains the “nnginx” files along with Dockerfile. You can use the command “cd labs/module1/nginx”
3. Build a new image by running the command “docker build -t nginxsample:v1.” In this case you're creating a new image based on Dockerfile (covered in earlier exercise on NGINX).

```

root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker build -t nginxsample:v1 .
Sending build context to Docker daemon 6.163MB
Step 1/4 : FROM microsoft/aspnetcore:2.0
--> c4ca78cf9dca
Step 2/4 : WORKDIR /app
--> Using cache
--> 5304b3e36cbe
Step 3/4 : COPY published ./
--> Using cache
--> 05bacf187dd4
Step 4/4 : ENTRYPOINT ["dotnet", "mywebapp.dll"]
--> Using cache
--> c3732185decc
Successfully built c3732185decc
Successfully tagged nginxsample:v1

```

4. If you run a “docker images” command, it will list the new container image with tag “v1”

```
root@super-Virtual-Machine:~/labs/module1/aspnetcore# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynodejsv2	latest	f3215f52d685	16 minutes ago	676MB
myaspcoreapp	2.0	c3732185decc	About an hour ago	328MB
nginxsample	v1	c3732185decc	About an hour ago	328MB
mynginx	latest	d1a6647bc6d5	About an hour ago	15.5MB
mynodejs	latest	086d7c3e9a9e	About an hour ago	664MB
mynodejs	v1	086d7c3e9a9e	About an hour ago	664MB
node	boron	67ed1f028e71	46 hours ago	659MB
microsoft/aspnetcore	2.0	c4ca78cf9dca	2 weeks ago	325MB
nginx	stable-alpine	24ed1c575f81	3 months ago	15.5MB