

Table of Contents

Exercise Guide: Module 1 - Introduction to Containers	2
Exercise 1: Running Docker Containers	2
Exercise 2: Searching Container Image on Docker Hub	4
Exercise 3: Inspecting Docker Layers	7
Exercise 4: Building Custom Container Images with Dockerfile	8

Exercise Guide: Module 1 - Introduction to Containers

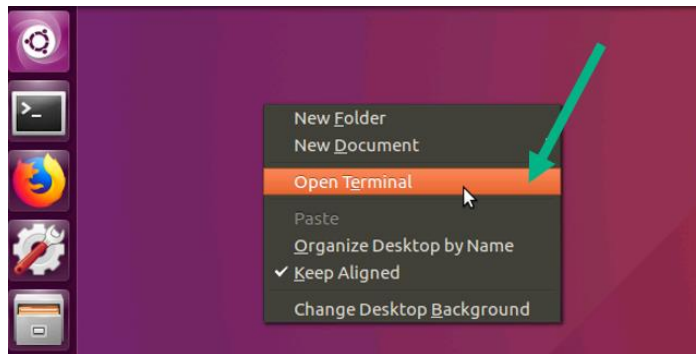
Exercise 1: Running Docker Containers

In this exercise, you will launch a fully functional WordPress blog engine using Docker Linux container. You will show the commands needed to pull the container image and then launch the container using Docker CLI. Finally, you will demonstrate agility of containers by launching multiple containers in a very short time.

Tasks

1. Running Single WordPress Blog Engine Container

1. Go to the Lab on Demand (LOD) Linux Ubuntu Virtual Machine. Start the machine. To open a command line prompt, right click on the desktop and choose **open terminal**.



2. Run “`sudo -i`” to ensure all commands have elevated privileges.
 - a. 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.

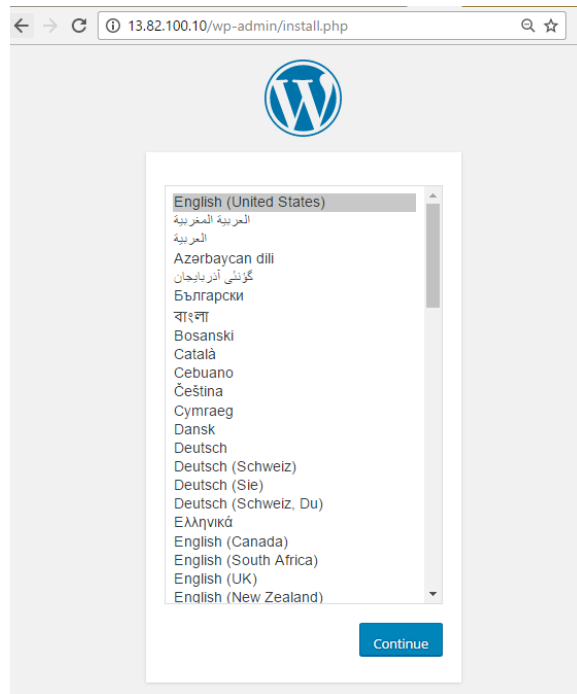
```
root@super-Virtual-Machine: ~  
super@super-Virtual-Machine:~$ sudo -i  
[sudo] password for super:  
root@super-Virtual-Machine:~#
```

3. Type “`docker pull tutum/wordpress`”.
 - a. 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`).
4. Run the command “`docker images`” and notice “`tutum/wordpress`” container image is now available locally for you to use.

5. Run the command “**docker run -d -p 80:80 tutum/wordpress**”.
 - a. 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
```

6. Run the following “**docker ps**” to see the running containers.
7. Click on the Firefox browser. Navigate to <http://localhost> and you should see WordPress.



2. Running Multiple WordPress Blog Engine Containers

1. 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
```

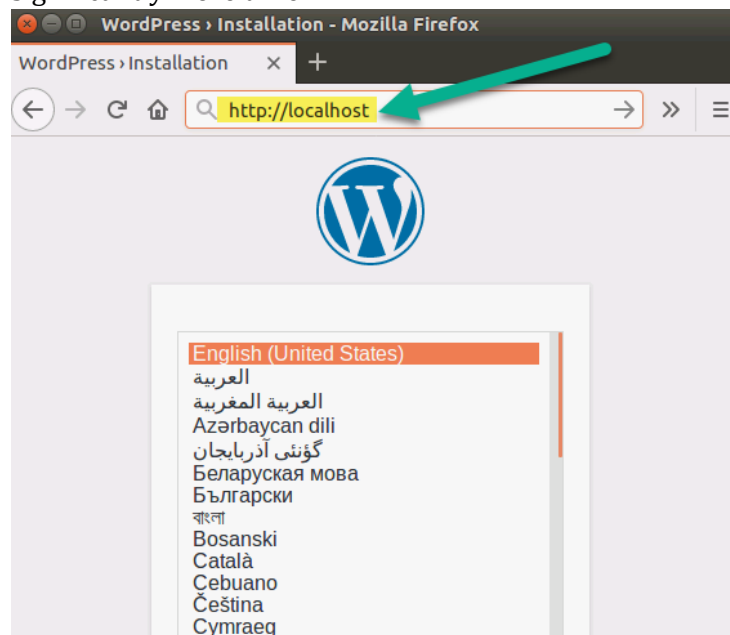
```
docker run -d -p 9090:80 tutum/wordpress
```

```
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
```

2. Now open a new Firefox browser window and Navigate to <http://localhost> but with port “8080” append to it, then “9090”.



- a. 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 may take significantly more time.



Exercise 2: Searching Container Image on Docker Hub

In this exercise you will show how to perform following tasks:

- Search DockerHub for Container Image using Docker CLI
- Search DockerHub website for Container Image
- Explain Docker Images Naming Convention

Tasks

1. Search DockerHub for Container Image using Docker CLI

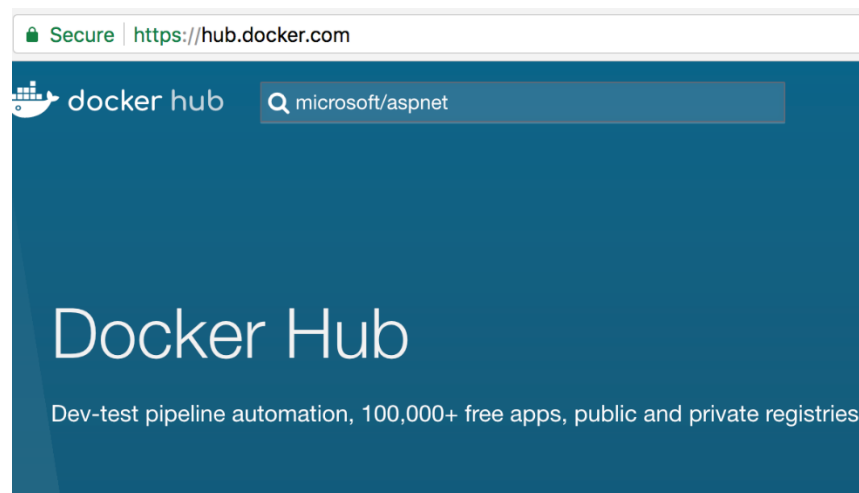
1. On command prompt type “**docker search microsoft**” and press ENTER.
2. Walkthrough the output and explain that search returns all relevant results for the image aspnet image that microsoft published on Docker public registry (Docker Hub).

NAME	DESCRIPTION
microsoft/aspnet	ASP.NET is an open source server-side Web ...
microsoft/dotnet	Official images for .NET Core for Linux an...
microsoft/mssql-server-linux	Official images for Microsoft SQL Server o...
mono	Mono is an open source implementation of M...
microsoft/windowsservercore	Windows Server 2016 Server Core base OS im...
microsoft/aspnetcore	Official images for running compiled ASP.N...
microsoft/nanoserver	Windows Server 2016 Nano Server base OS im...
microsoft/iis	Internet Information Services (IIS) instal...
microsoft/mssql-server-windows-express	Official Microsoft SQL Server Express Edit...
microsoft/azure-cli	Docker image for Microsoft Azure Command L...
microsoft/mssql-server-windows	Official images for Microsoft SQL Server f...
microsoft/aspnetcore-build	Official images for building ASP.NET Core ...
microsoft/dotnet-framework	The official Docker images for .NET Framew...
microsoft/mssql-server-windows-developer	Official Microsoft SQL Server Developer Ed...
microsoft/vsts-agent	Official images for the Visual Studio Team...

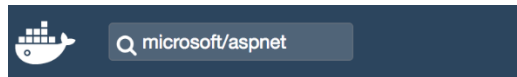
3. Now run the command “**docker search microsoft --limit 5**” which will limit the results to top 5 matches.
4. Finally, browse to <https://docs.docker.com/engine/reference/commandline/search/#examples> and explain to audience various filters available for the search command.

2. Search Docker Hub website for Container Image

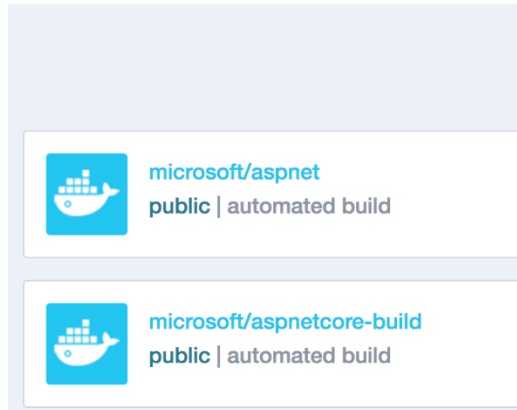
1. Open a browser and navigate to <http://hub.docker.com>
2. In the search box type “microsoft/aspnet” and press ENTER.



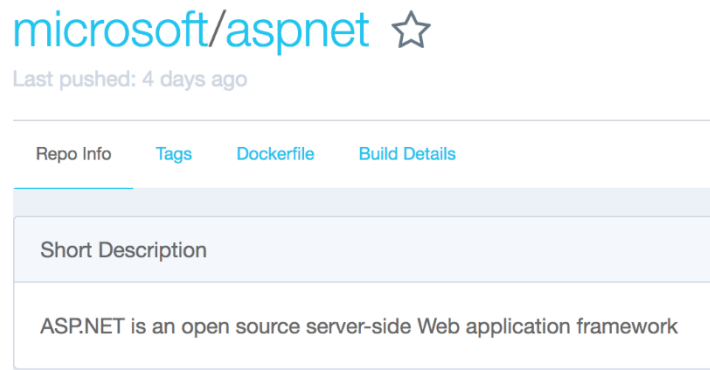
3. Notice the result page displaying all the container images.



Repositories (997)



4. Select the first image titled “microsoft/aspnet” and then walkthrough the details about the specific aspnet image as provided on the page.



5. Keep the browser open.

3. Docker Image Naming Convention

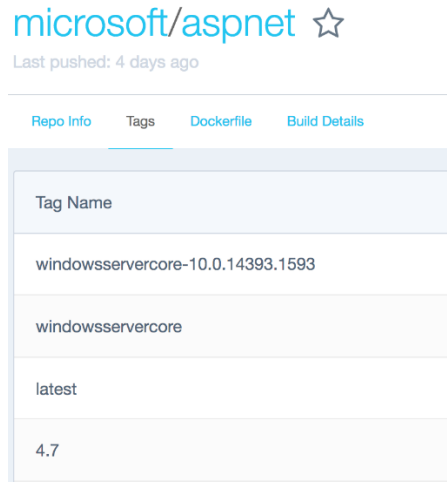
1. Highlight the fact the **docker images names are combination of three pieces of information:**

REPOSITORY-NAME/IMAGE-NAME[:TAG]

Explain that for microsoft/aspnet image these items are as follows

- REPOSITORY-NAME: microsoft
- IMAGE-NAME: aspnet
- TAG: latest (by default)

2. Now select the Tags tab and explain the various tags for microsoft/aspnet image. For example, latest tag is default and represent the aspnet image targeted to Linux platform. However, windowsservercore tag refers to aspnet image that is using Windows Server Core operating system.



Exercise 3: Inspecting Docker Layers

In this exercise you will show:

- How to get information about container image layers using docker inspect command that returns low-level information for Docker objects.

Tasks

1. List All the Layers for a Docker Image

1. On the command prompt run the command **"docker inspect tutum/wordpress"**.
2. Notice the output of inspect command. It's in a JSON file format. The relevant information related to container image layer is at the bottom (you may need to scroll down to get to it)

```
"Layers": [
  "sha256:8698b31c92d5cf4ee37154fc560516040f372bd45707c3161be479443970d8a2",
  "sha256:982549bd6b32c8d59c693d27bf22c3613ac66dc344833881ef4099be656330cb",
  "sha256:0d81735d8272465865b21615e57746b82ffd4180c9ef4a2ed6e7bfe7464cc156",
  "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
  "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
  "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
  "sha256:5b183ee9ec8ef99ab417b94251bbba6de0f0764b16a32c80fc5faa40e0aa6444",
  "sha256:33a3f86fa883bc0fedc99618e46a92fd6e460da35a2167b2e45bf8e1b9f397fe"
```

With SHA256 hash for each layer you can verify that specific layer is changed or not. You can use information on this article

(<https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#data-volumes-and-the-storage-driver>) to further drive the conversation.

2. Look at locally cached images and layers

1. Run “**docker info**” and show the two fields we are interested in: Storage Driver and Docker Root Dir
2. Show where the image database and layer database is managed by Docker
“ls -al /var/lib/docker/image/aufs/”
3. Show the list of images contained in the image database. Call out that it is aligned with the “docker images” result
“ls -al /var/lib/docker/image/aufs/imagedb/content/sha256”
4. Show the list of layers contained in the image database. Call out that it is much bigger than the number of images since images are composed of multiple layers
“ls -al /var/lib/docker/image/aufs/layerdb/sha256”

Exercise 4: Building Custom Container Images with Dockerfile

In this exercise you will show how to perform following tasks:

- Building and running Node.JS Application inside container
- Building and running NGINX container
- Building and running ASP.NET Core inside container
- Tagging an existing container image

Tasks

1. Building and Running Node.JS Application Inside Container

1. The relevant files related to a node.js application along with the Dockerfile are available inside the directory “**labs/ubuntu-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:~# ls
Desktop labs
root@super-Virtual-Machine:~# cd labs/module1/nodejs/
root@super-Virtual-Machine:~/labs/module1/nodejs#
```

3. Let’s examine the Dockerfile by typing the command “nano Dockerfile” and press Enter. Either write out the Dockerfile with the students or explain each line of the Dockerfile and how it works.


```

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. You are now ready to build a new image based on the Dockerfile you just modified. 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.

```

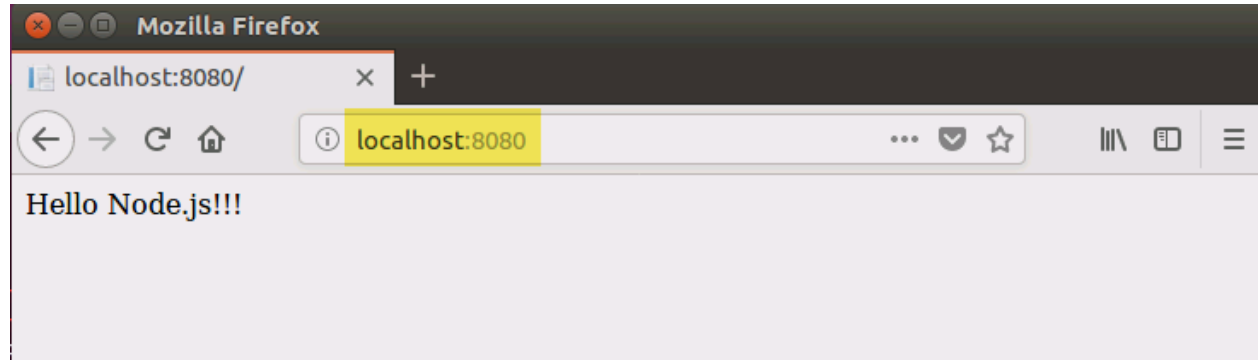
root@super-Virtual-Machine:~/labs/module1/nodejs# docker build -t mynodejs .
Sending build context to Docker daemon  4.096kB
Step 1/9 : FROM node:boron
boron: Pulling from library/node
3d77ce4481b1: Downloading  1.072MB/54.26MB
534514c83d69: Downloading  6.459MB/17.58MB
d562b1c3ac3f: Downloading  2.639MB/43.25MB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

```

5. Run **docker history mynodejs** to see the history of the image.
6. 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.
7. Let's make sure to stop any running containers from previous steps by running **"docker stop \$(docker ps -aq)"**
8. Finally, let's 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 is 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
```

9. 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 exercise you will create a new image using the NGINX web server base image hosting a simple static html page. The relevant files including static html file “index.html” along with the Dockerfile are available inside the directory “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.

6. 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.

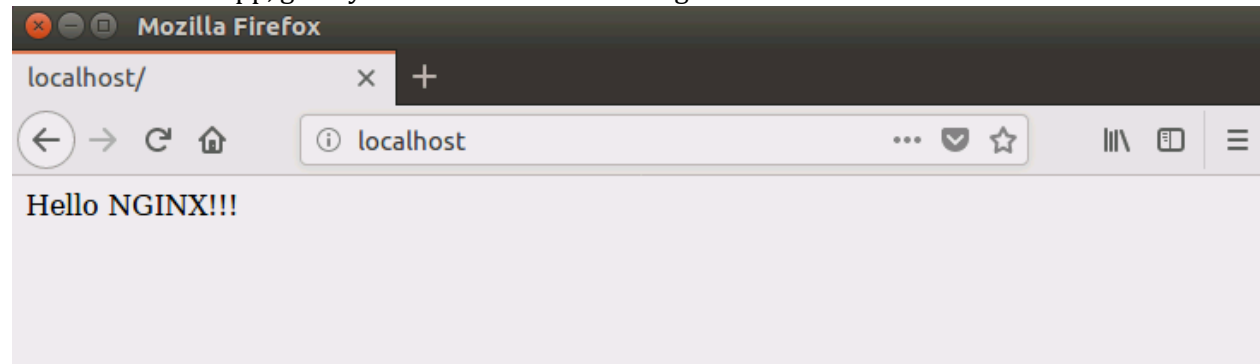
```
root@super-Virtual-Machine:~/labs/module1/nginx# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynginx	latest	dbb68ae46b21	31 seconds ago	16MB
mynodejs	latest	a4a2ecd25ff5	8 minutes ago	889MB
microsoft/dotnet	2.2-aspnetcore-runtime	2f7531819f40	25 hours ago	260MB
nginx	stable-alpine	cafe99fe265b	5 days ago	16MB
node	boron	753cb37bc49c	8 days ago	884MB

7. 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
```

8. 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 exercise 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`”.
2. Even though this could be done in the Dockerfile (like in the next module), we are going to run the commands to build the application. Run the two commands

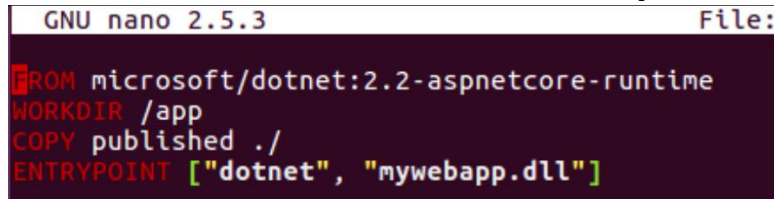
`dotnet build`

`dotnet publish -o published`

Now check that the published folder has been created with the binaries of our application.

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

- 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”.



```

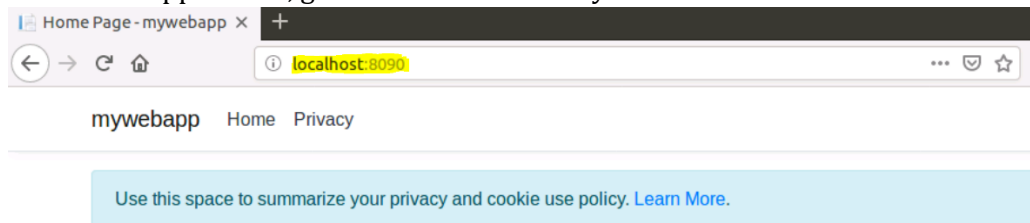
GNU nano 2.5.3 File:
FROM microsoft/dotnet:2.2-aspnetcore-runtime
WORKDIR /app
COPY published ./
ENTRYPOINT ["dotnet", "mywebapp.dll"]

```

- To create the container image run the command
“docker build -t myaspcoreapp:2.2 .”
 Notice the use of tag “2.2” that signifies the use of framework version of dotnet core.
- Launch the container running the app inside it by running the command
“docker run -d -p 8090:80 myaspcoreapp:2.2”

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

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



- You can also run the container in an interactive mode by issuing the following command: **docker run -it myaspcoreapp:2.2 /bin/bash**
Note: Remember to comment out the ENTRYPOINT command

4. 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	15637ed38561	3 minutes ago	909MB
myaspcoreapp	2.2	4812e8e6f51c	43 minutes ago	265MB
mynginx	latest	dbb68ae46b21	3 hours ago	16MB
mynodejs	latest	a4a2ecd25ff5	3 hours ago	889MB
microsoft/dotnet	2.2-aspnetcore-runtime	2f7531819f40	28 hours ago	260MB
nginx	stable-alpine	cafef9fe265b	5 days ago	16MB
node	boron	753cb37bc49c	8 days ago	884MB

Run the command

"docker tag mynodejs mynodejsv2"

To understand importance of tagging take a look at the container image just created, "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	15637ed38561	6 minutes ago	909MB
myaspcoreapp	2.2	4812e8e6f51c	45 minutes ago	265MB
mynginx	latest	dbb68ae46b21	3 hours ago	16MB
mynodejs	latest	a4a2ecd25ff5	3 hours ago	889MB
mynodejs	v1	a4a2ecd25ff5	3 hours ago	889MB
microsoft/dotnet	2.2-aspnetcore-runtime	2f7531819f40	28 hours ago	260MB
nginx	stable-alpine	cafe99fe265b	5 days ago	16MB
node	boron	753cb37bc49c	8 days ago	884MB

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.

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>