

MA PRACTICAL 4

(Installing software packages on Docker, Working with Docker Volumes and Networks)

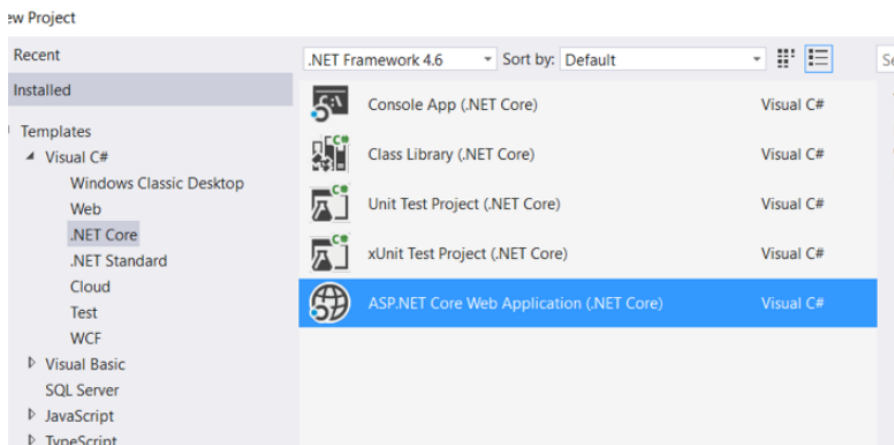
What is the software required for Windows?

- Windows 10 is required for Docker installation.
- Visual Studio 2017 has built-in support for Docker, so this is highly recommended.
- .NET Core SDK
- Docker for Windows
- Docker Tools

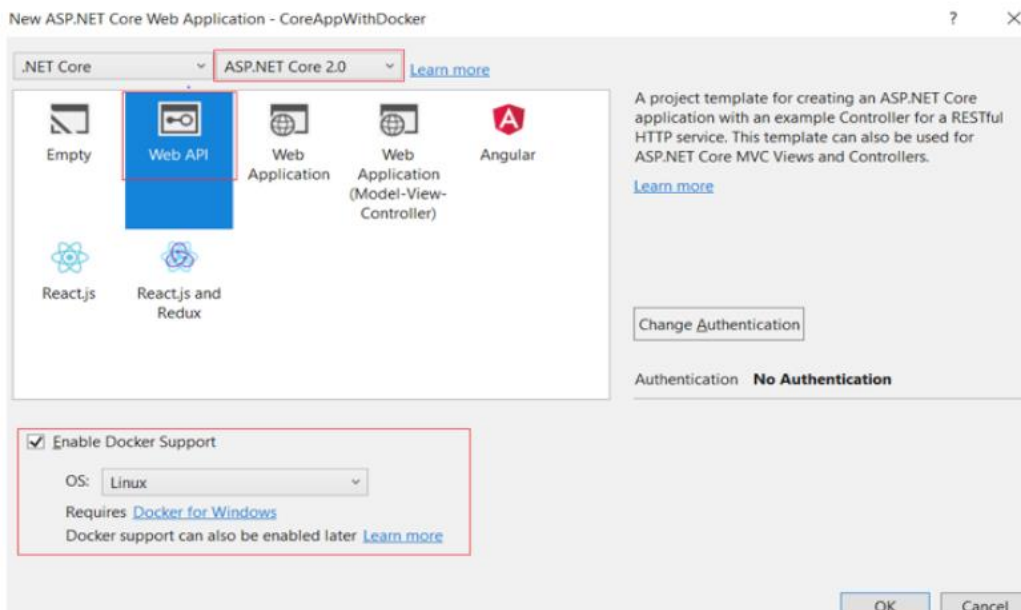
How to create a new microservice using .NET Core and then build and run it using Docker

Step 1: Create a microservice (.NET Core WebAPI) with Docker support as shown below:

Select “ASP.NET Core Web Application (.NET Core)” from the drop-down menu.



Select the “Enable Docker Support” option.



The following Application Structure will be created along with “Docker File.”



Dockerfile

This file is the entry point for running any Docker application.

It is used to build an image of the application's published code in “obj/Docker/publish.”

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "CoreAppWithDocker.dll"]
```

docker-compose.override.yml

This file is used when running an application using Visual Studio.

```
1 version: '3'
2
3 services:
4   coreappwithdocker:
5     environment:
6       - ASPNETCORE_ENVIRONMENT=Development
7     ports:
8       - "80"
```

Step 2: Update Dockerfile and docker-compose.override.yml as shown below and build the application. 80 is the default Docker container port, so you should update it to a different port number, like 83.

Dockerfile

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
ENV ASPNETCORE_URLS http://+:83
EXPOSE 83
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "CoreAppWithDocker.dll"]
```

docker-compose.override.yml

```
1 version: '3'
2
3 services:
4   coreappwithdocker:
5     environment:
6       - ASPNETCORE_ENVIRONMENT=Development
7     ports:
8       - "83"
```

Note: You can run the application using both Visual Studio and the Docker command line.

First Line FROM Microsoft/aspnetcore:2.0 is the base image for this application in order to run the dot net core application.

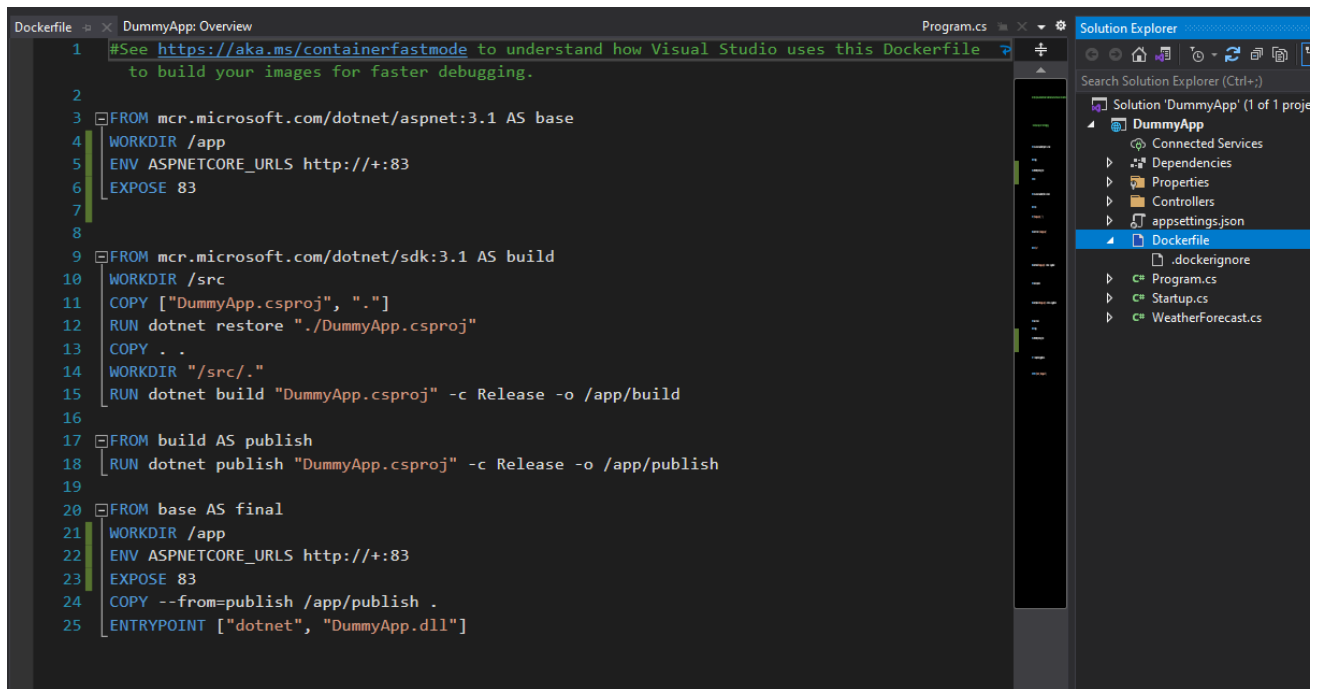
ARG source is the argument which helps to pass data to the image.

WORKDIR /app is the working directory of the image; it will store all DLLs inside the app folder.

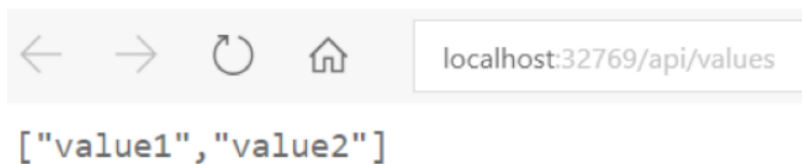
COPY will copy the DLLs of the application to the root directory (here dot represents root) of the image.

ENTRYPOINT is responsible to run the main application with the help of ASPNETCOREAPP.dll.

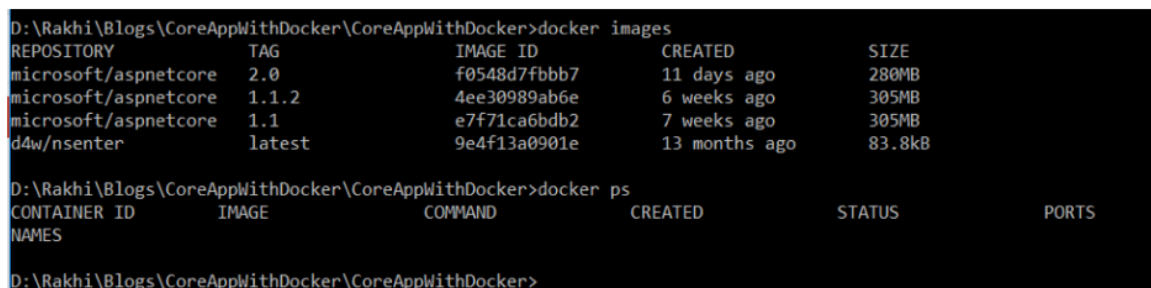
In the case of .NET Core 3.1 the Docker file will look like this



Step 3: Run the application using Visual Studio.



Step 4: Run the application using Docker Command. Open Application folder from command prompt and check the existing images using Docker images and running containers using Docker PS.



As you can see, there is no running container. So, run the following commands to create build:

To restore packages: dotnet restore

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>dotnet restore
Restoring packages for D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker\CoreAppWithDocker.csproj...
Restore completed in 41.91 ms for D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker\CoreAppWithDocker.csproj.
Restore completed in 959.77 ms for D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker\CoreAppWithDocker.csproj.
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

To publish the application code: dotnet publish -o obj/Docker/publish

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>dotnet publish -o obj/Docker/publish
Microsoft (R) Build Engine version 15.4.8.50001 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

CoreAppWithDocker -> D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker\bin\Debug\netcoreapp2.0\CoreAppWithDocker.dll
CoreAppWithDocker -> D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker\obj\Docker\publish\
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

To build the image: docker build -t imagename

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker build -t coreappimage .
Sending build context to Docker daemon 324.6kB
Step 1/6 : FROM microsoft/aspnetcore:2.0
--> f0548d7fbbb7
Step 2/6 : ARG source
--> Running in 592bc21b76ab
--> 89c174bc6537
Removing intermediate container 592bc21b76ab
Step 3/6 : WORKDIR /app
--> 7e8e26f37175
Removing intermediate container 06c848058834
Step 4/6 : EXPOSE 80
--> Running in ed13b8447417
--> a951d3046049
Removing intermediate container ed13b8447417
Step 5/6 : COPY ${source:-obj/Docker/publish} .
--> 615b829e6cbf
Step 6/6 : ENTRYPOINT dotnet CoreAppWithDocker.dll
--> Running in f6540ca62ca4
--> 500f5a37043b
Removing intermediate container f6540ca62ca4
Successfully built 500f5a37043b
Successfully tagged coreappimage:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Now, check the newly created image "coreappimage" in Docker Images.

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
coreappimage         latest              0db1b2442d44       4 seconds ago      280MB
microsoft/aspnetcore 2.0                 f0548d7fbbb7       11 days ago        280MB
microsoft/aspnetcore 1.1.2              4ee30989ab6e       6 weeks ago        305MB
microsoft/aspnetcore 1.1                 e7f71ca6bdb2       7 weeks ago        305MB
d4w/nsenter          latest              9e4f13a0901e       13 months ago      83.8kB
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

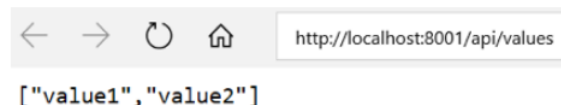
Run the image in a container: `docker run -d -p 8001:83 --name core1 coreappimage`

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker run -d -p 8001:83 --name core1 coreappimage
2c35a96602889c1f93213a4a0598b7903dddb404fde64c06881d8db3bbef663a
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Check the running container: `Docker PS`

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
2c35a9660288  coreappimage "dotnet CoreAppWit..." About a minute ago Up About a minute 0.0.0.0:8001->83/tcp
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Now the application is running in the Core1 container with the URL `http://localhost:8001`.



Case 1: Run the same image in multiple containers

We can run the same image in multiple containers at the same time by using:

`docker run -d -p 8002:83 --name core2 coreappimage`

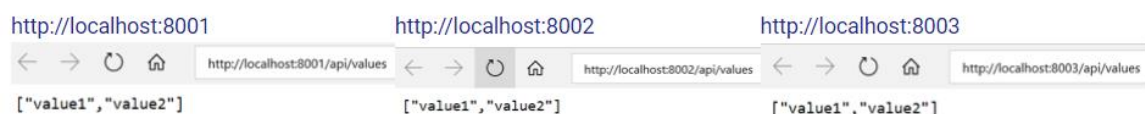
`docker run -d -p 8003:83 --name core3 coreappimage`

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker run -d -p 8002:83 --name core2 coreappimage
3f26f83e36f1bda6aa40d6711a2be71ab66c1b775cc080064062cd9b26b7d40
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker run -d -p 8003:83 --name core3 coreappimage
647d658fa98b989ef58982e7a580ded5278896171dd60bbcbcb8f8df14c486c77b
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Check the running containers by using `Docker PS`.

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
647d658fa98b  coreappimage "dotnet CoreAppWit..." 51 seconds ago Up 51 seconds 0.0.0.0:8003->83/tcp
3f26f83e36f1  coreappimage "dotnet CoreAppWit..." About a minute ago Up About a minute 0.0.0.0:8002->83/tcp
2c35a9660288  coreappimage "dotnet CoreAppWit..." 38 minutes ago Up 38 minutes 0.0.0.0:8001->83/tcp
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

We can see that there are 3 containers running for the same image at 8001, 8002, and 8003.



Case 2: Manage Containers: Stop/Start/Remove Containers

Stop container:

We can stop any running containers using “docker stop containerid/containername”
docker stop core1.

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker stop core1
core1
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Note: Now this container core1 will be listed in the “All Containers” list but not in the running containers list.

Check running containers: docker ps:

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
647d658fa98b   coreappimage  "dotnet CoreAppWit..." 23 minutes ago Up 23 minutes  0.0.0.0:8003->8003
3f26f83e36f1   coreappimage  "dotnet CoreAppWit..." 23 minutes ago Up 23 minutes  0.0.0.0:8002->8002
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Check all containers: docker ps -a

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
647d658fa98b   coreappimage  "dotnet CoreAppWit..." 16 minutes ago Up 16 minutes  0.0.0.0:8003->8003
3f26f83e36f1   coreappimage  "dotnet CoreAppWit..." 16 minutes ago Up 16 minutes  0.0.0.0:8002->8002
2c35a9660288   coreappimage  "dotnet CoreAppWit..." About an hour ago Exited (0) 4 minutes ago
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Note: Now the container running on <http://localhost:8001> will not work.

localhost:8001/api/values



Hmmm...can't reach this page

Try this

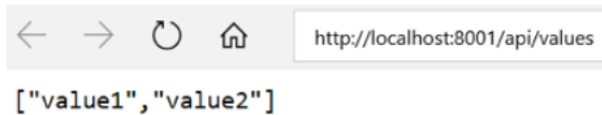
- Make sure you've got the right web address:
<http://localhost:8001>
- Search for "<http://localhost:8001>" on Bing

Start Container:

We can start a stopped container using "docker start containerId/containername"

=> docker start core1

Note: Now it will be listed in the running containers list and <http://localhost:8001> will start working.



Remove Container:

We can remove any stopped container, but then we will not be able to start it again.

So first we should stop the container before removing it:

=> docker stop core1

=> docker rm core1

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker rm core1
core1
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Now this container will not be listed on the containers list:

```
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
647d658fa98b   coreappimage   "dotnet CoreAppWit..." 25 minutes ago Up 25 minutes 0.0.0.0:8003->8
3/tcp   core3
3f26f83e36f1   coreappimage   "dotnet CoreAppWit..." 26 minutes ago Up 26 minutes 0.0.0.0:8002->8
3/tcp   core2
D:\Rakhi\Blogs\CoreAppWithDocker\CoreAppWithDocker>
```

Case 3 – Share the Application on Docker Hub Repository

Share the application

Estimated reading time: 4 minutes

Now that we've built an image, let's share it! To share Docker images, you have to use a Docker registry. The default registry is Docker Hub and is where all of the images we've used have come from.

i Docker ID

A Docker ID allows you to access Docker Hub which is the world's largest library and community for container images. Create a [Docker ID](#) for free if you don't have one.

Create a repo

To push an image, we first need to create a repository on Docker Hub.

1. [Sign up](#) or Sign in to [Docker Hub](#).
2. Click the **Create Repository** button.
3. For the repo name, use `getting-started`. Make sure the Visibility is `Public`.

i Private repositories

Did you know that Docker offers private repositories which allows you to restrict content to specific users or teams? Check out the details on the [Docker pricing](#) page.

4. Click the **Create** button!

If you look at the image below an example **Docker command** can be seen. This command will push to this repo.

Docker commands

[Public View](#)

To push a new tag to this repository,

```
docker push docker/getting-started:tagname
```

In your case use your docker image name instead of getting-started

Push the image

1. In the command line, try running the push command you see on Docker Hub. Note that your command will be using your namespace, not "docker".

```
$ docker push docker/getting-started
The push refers to repository [docker.io/docker/getting-started]
An image does not exist locally with the tag: docker/getting-started
```

Why did it fail? The push command was looking for an image named docker/getting-started, but didn't find one. If you run `docker image ls`, you won't see one either.

To fix this, we need to "tag" our existing image we've built to give it another name.

2. Login to the Docker Hub using the command `docker login -u YOUR-USER-NAME`.
3. Use the `docker tag` command to give the `getting-started` image a new name. Be sure to swap out `YOUR-USER-NAME` with your Docker ID.

```
$ docker tag getting-started YOUR-USER-NAME/getting-started
```

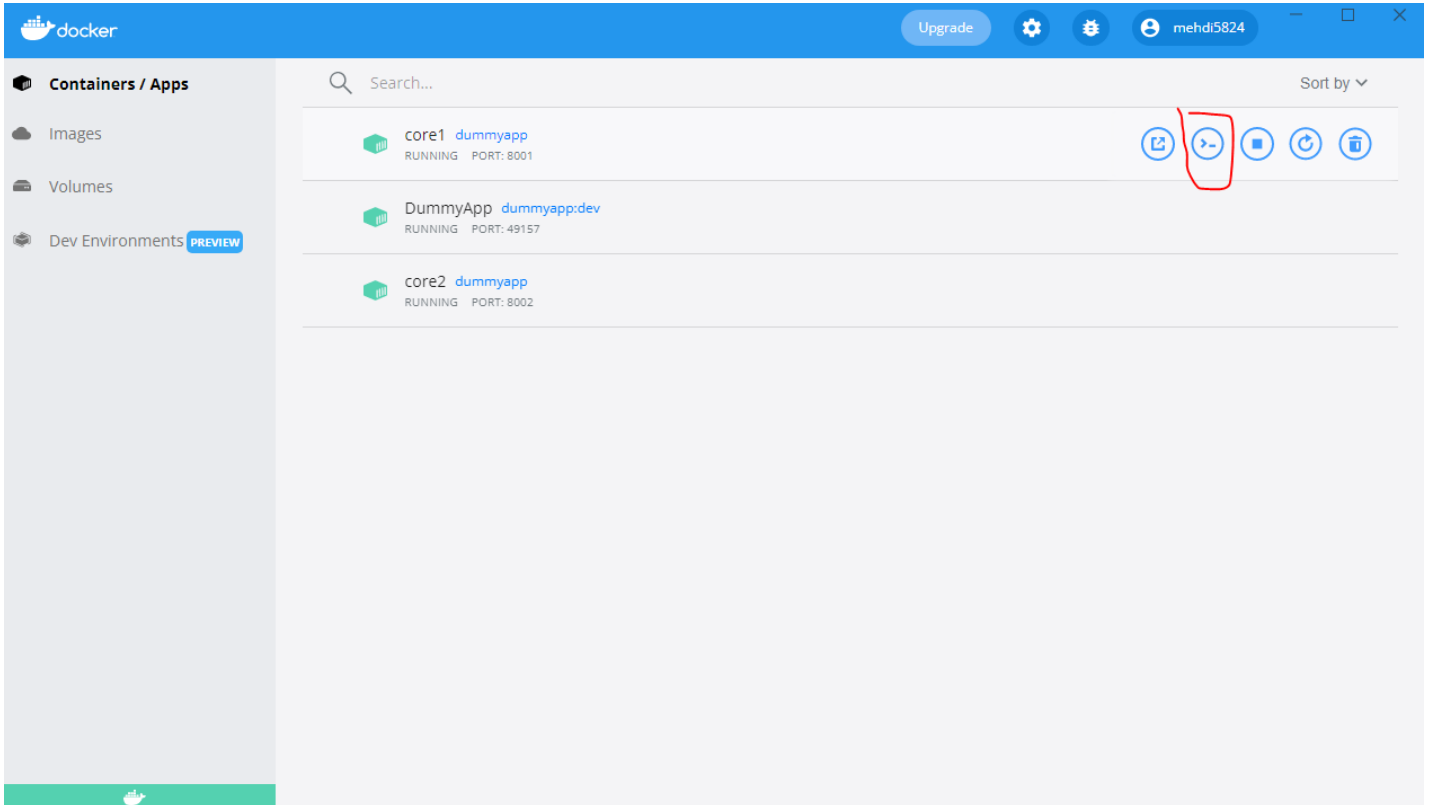
4. Now try your push command again. If you're copying the value from Docker Hub, you can drop the `tagname` portion, as we didn't add a tag to the image name. If you don't specify a tag, Docker will use a tag called `latest`.

```
$ docker push YOUR-USER-NAME/getting-started
```

Part 2

Installing Software Packages in Docker

Step 1 – Go to CLI Option on the container in Docker Desktop



Step 2: Now, you have opened the bash of your Ubuntu Docker Container. To install any packages, you first need to update the OS.

```
apt-get -y update
```

```
root@068f710e29a3:/# apt-get -y update
Get:1 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:2 http://security.ubuntu.com/ubuntu focal-security InRelease [107 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [111 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease [98.3 kB]
Get:5 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [630 kB]
Get:6 http://archive.ubuntu.com/ubuntu focal/multiverse amd64 Packages [177 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [11.3 MB]
Get:8 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [428 kB]
Get:9 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [84.4 kB]
Get:10 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [1170 B]
Get:11 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:13 http://archive.ubuntu.com/ubuntu focal-updates/multiverse amd64 Packages [21.6 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [784 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [840 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [103 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [4277 B]
Fetched 16.3 MB in 21s (779 kB/s)
Reading package lists... Done
root@068f710e29a3:/#
```

Updating the Container

Step 3: After you have updated the Docker Container, you can now install the Firefox and Vim packages inside it.

```
apt-get -y install firefox
apt-get -y install vim
```

```
root@66f710e29a3:/# apt-get -y install Firefox
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  adwaita-icon-theme alsa-topology-conf alsa-ucm-conf at-spi2-core dbus dbus-user-session dconf-gsettings-backend dconf-service
  distro-info-data dmsetup file fontconfig fontconfig-config fonts-dejavu-core glib2.0 glib-networking glib-networking-common
  glib-networking-services gsettings-desktop-schemas gtk-update-icon-cache hicolor-icon-theme humanity-icon-theme krb5-locales
  libapparmor1 libargon2-1 libasound2 libasound2-data libatk-bridge2.0-0 libatk1.0-0 libatk1.0-data libatspi2.0-0 libavahi-client3
  libavahi-common-data libavahi-common3 libbrotli1 libbsd0 libcairo-gobject2 libcairo2 libcanberra0 libcap2 libcolord2 libcryptsetup2
```

Installing Firefox

```
root@66f710e29a3:/# apt-get -y install vim
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  alsa-topology-conf alsa-ucm-conf file libasound2 libasound2-data libcanberra0 libexpat1 libgpm2 libltdl7 libmagic-ngc libmagic1
  libmpdec2 libogg0 libpython3.8 libpython3.8-minimal libpython3.8-stdlib libreadline8 libsqlite3-0 libssl1.1 libtdb1 libvorbis0a
  libvorbisfile3 nmea-support readline-common sound-theme-freedesktop vim-common vim-runtime xxd xz-utils
Suggested packages:
  libasound2-plugins alsa-utils libcanberra-gtk0 libcanberra-pulse gpm readline-doc ctags vim-doc vim-scripts
The following NEW packages will be installed:
```

Installing Vim

You can now easily use these packages through the bash itself.

Step 4: Run vim to verify if the software package has been installed

Container volumes

With the previous experiment, we saw that each container starts from the image definition each time it starts. While containers can create, update, and delete files, those changes are lost when the container is removed and all changes are isolated to that container. With volumes, we can change all of this.

Volumes provide the ability to connect specific filesystem paths of the container back to the host machine. If a directory in the container is mounted, changes in that directory are also seen on the host machine. If we mount that same directory across container restarts, we'd see the same files.

Step 1

Working with Docker Volumes explained below:-

- a) Let us create the volume first. For the reference we will type below command:-
→ `docker volume`

```
F:\Microservices\getting-started-master\app>docker volume

Usage:  docker volume COMMAND

Manage volumes

Commands:
  create      Create a volume
  inspect     Display detailed information on one or more volumes
  ls          List volumes
  prune       Remove all unused local volumes
  rm          Remove one or more volumes

Run 'docker volume COMMAND --help' for more information on a command.

F:\Microservices\getting-started-master\app>
```

- b) Now lets create the actual volume:-

→ `docker volume create myvol1`

```
F:\Microservices\getting-started-master\app>docker volume create myvol1
myvol1

F:\Microservices\getting-started-master\app>
```

As you can see here our volume is created.

c) To list the volume we will write below command:-

```
F:\Microservices\getting-started-master\app>docker volume ls
DRIVER      VOLUME NAME
local       aba83257ee43df3f86bfea2b09c1d1ffe5a59b9ced82c6b7ea5f458e9e298e72
local       d247fdb49990ed914b54fffe365671c1f3b773d5871038817e18d36cf6e288bf
local       myvol1

F:\Microservices\getting-started-master\app>
```

d) To get the details of our volume we have to write below command:-

➔ docker volume inspect myvol1

```
F:\Microservices\getting-started-master\app>docker volume inspect myvol1
[
  {
    "CreatedAt": "2021-05-10T06:36:15Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/myvol1/_data",
    "Name": "myvol1",
    "Options": {},
    "Scope": "local"
  }
]

F:\Microservices\getting-started-master\app>
```

Here you can see all the details of our myvol1 i.e. name, created time, driver, mountpoint.

Our volume is located at the path mentioned in Mountpoint section.

e) To remove your volume you can write below command:-

➔ docker volume rm myvol1

To remove all unused volumes we can write below command

➔ docker volume prune

These are the basic functionalities of docker volume. You can explore more functionalities as well.

Working with docker network explained below:-

To write this command below is the syntax:-

➔ docker network COMMAND

a) To Connect a container to a network

➔ **docker network connect**

- b) To create a network we have to write below command:-
 - ➔ `docker network create`
- c) To disconnect a container from a network
 - ➔ `docker network disconnect`
- d) To display detailed information on one or more networks
 - ➔ `docker network inspect`
- e) To list the network:-
 - ➔ `docker network ls`
- f) To remove all unused networks
 - ➔ `docker network prune`
- g) To remove one or more networks
 - ➔ `docker network rm`