LAB 2 : Packaging .Net Applications with Dockerfile

Author: Gourav Shah

Publisher: School of Devops

Version: v1.0.0

When you start with a container based software delivery, the first step always is to take your application and package it as a container image along with its run time environment. Mastering image build process with all of its aspects is an indispensable skill in the container world. The objective of this module is to help you master that skill. You will learn how to package both, your .NET Core and .NET Framework applications, with docker by building images with windows as base operating system.

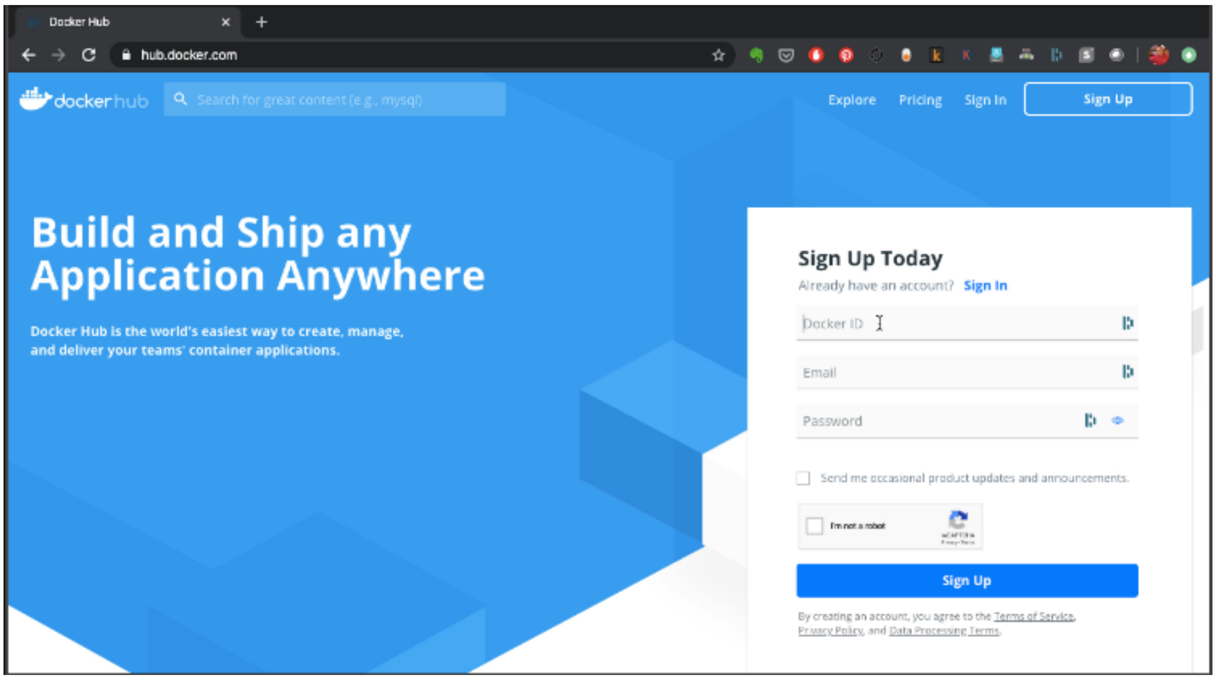
This is what you will learn in this lesson

- How to get sign up to Docker Hub Registry and create your Docker ID
- How to manually test build a Docker Image by launching a container with base image and modifying it
- How to automate the process of building these images by writing a Dockerfile
- How to construct a Dockerfile and what are the best practices while writing one
- Why do you need a Multi Stage Dockerfile and how does it work
- How to package a ASP .NET Core as well as ASP .NET Framework(legacy) application

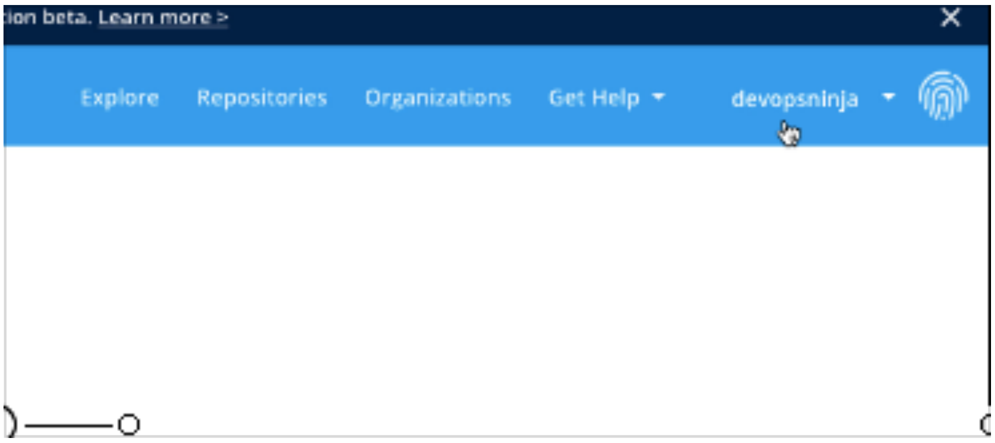
-

Creating a Registry Account (Docker Hub)

By the end of this session, you would have at least a couple of images built, which then you would want to publish to a registry. You could get started with Docker Hub as the default registry and publish your images under your own account. In order to do that, you need to have Docker Hub account. If you have created a Docker ID while setting up Docker Desktop software, its the same account. If not, follow this process to sign up to Docker Hub and validate your account.


Docker Hub
hub.docker.com
docker hub
Search for great content (e.g., mysql)
Explore
Pricing
Sign In
Sign Up
Build and Ship any Application Anywhere
Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications.
Sign Up Today
Already have an account? Sign In
Docker ID
Email
Password
Send me occasional product updates and announcements.
I'm not a robot
Sign Up
By creating an account, you agree to the Terms of Service, Privacy Policy, and Data Processing Terms.

- Visit hub.docker.com
- Sign up by providing username, email and password
- Verify your email address
- Login

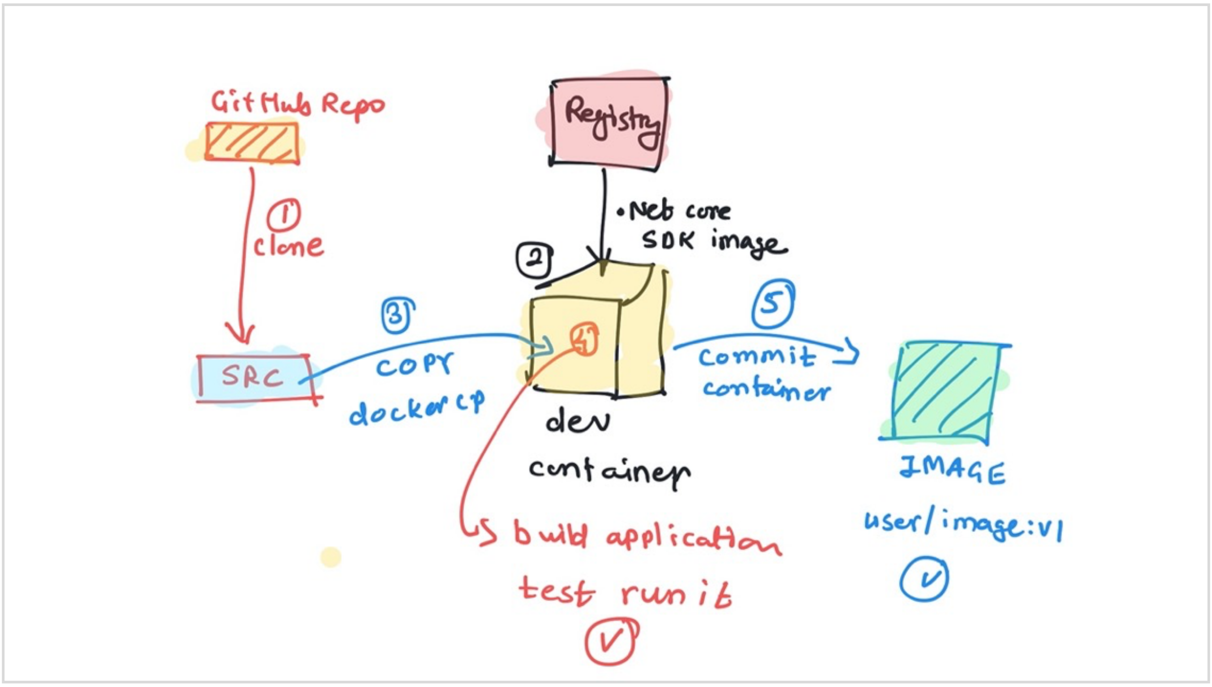


Once logged in, do note down your docker ID from the top right corner. Its your username in all small letters. This is an important step as this is the username / ID you are supposed to use while tagging your images later in this chapter.

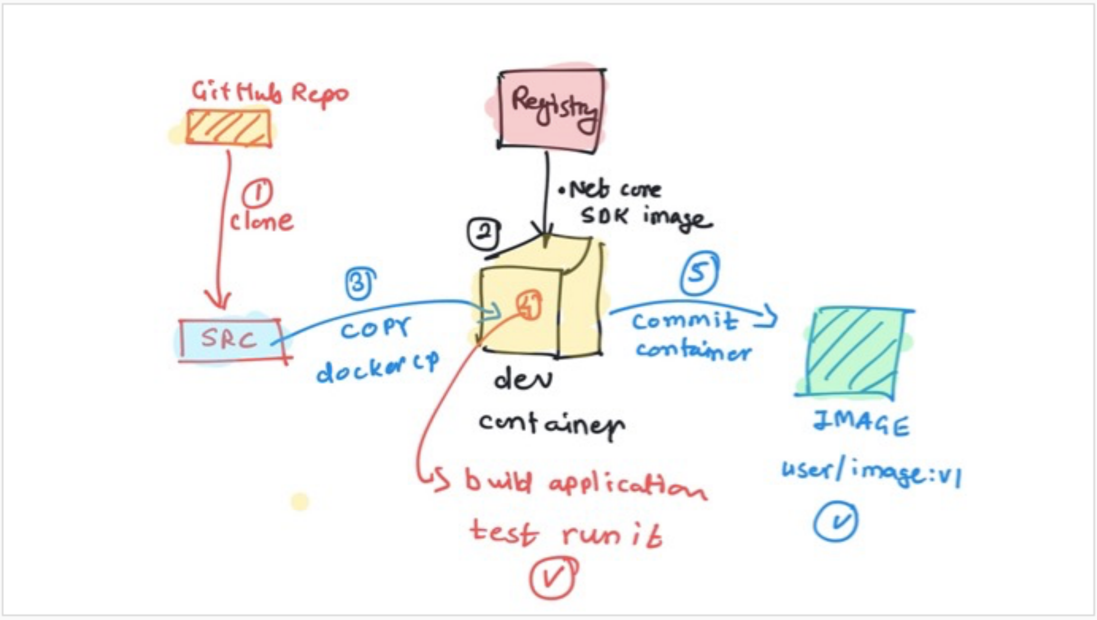
Test build a docker image (manual approach) for a ASP .NET core application

Before you automate the process of building an image, you need to know what you are going to automate. To understand the process doing a manual test build of your application always helps. We take example of this https://github.com/initcron/dotnetcore-samples   ASP .NET core application to try building an image.

You also need a .NET Core SDK environment to build this application, which itself, can be created by using a container image. Following would be the step by step process to create a development environment and to build this sample application.



1. Clone the code from a GitHub repository to local development host.
2. Create a container based development environment build the application. Use a pre built .NET Core SDK image which contains the tooling.
3. Copy over the source code to this dev container.
4. Connect to the container and perform all the tasks necessary to build the application. Also do a test run of the application to ensure its working fine.
5. Once tested, commit the container's changes to an image using `docker



1. Clone the code from a GitHub repository to local development host.
2. Create a container based development environment build the application. Use a pre built .NET Core SDK image which contains the tooling.
3. Copy over the source code to this dev container.
4. Connect to the container and perform all the tasks necessary to build the application. Also do a test run of the application to ensure its working fine.
5. Once tested, commit the container's changes to an image using `` `docker container commit` `` command.

container commit` command.

By the end of step 5, you should get a manually built docker image.   Go ahead and follow along to convert these 5 steps into action.

Step 1: Begin by clinging the source repository and switch to the work directory as,

```
git clone https://github.com/initcron/dotnetcore-samples.git

cd dotnetcore-samples/aspnetapp
```

Step 2 : Now, Create a dev environment with .NET SDK to build this application with. Also login to this container with exec to create a work directory.

```
docker run -idt -p 8000:80 --name dev mcr.microsoft.com/dotnet/core/sdk:3.1 cmd

docker exec -it dev cmd
```

Create a work directory and name it as `source`, which would be used to copy over the code to and run the build from.

```
[inside the container]

mkdir source

exit
```

Step 3: Now, copy over the source to the container . You would have to stop the container if using Windows 10, which uses hyper-v containers and does not support copying files to a running containers.

```
docker stop dev
docker cp . dev:/source
docker start dev
docker exec -it dev cmd
```

Step 4: By the end of the above command sequence, you have started the container and logged into it using exec command. Now, build the application and test run it as follows,

```
cd source
dotnet restore
cd aspnetapp
dotnet publish -c release -o /app --no-restore
```

```
cd /app
SET ASPNETCORE_URLS=http://*:80
dotnet aspnetapp.dll
```

Note : Use `SET ASPNETCORE_URLS=http://*:80` if using **CMD**. Replace it with `$env:ASPNETCORE_URLS="http://*:80"` for **powershell** and with `ASPNETCORE_URLS="http://*:80"` on**linux**.

The last command should have started running the application. You have already mapped port 80 on the dev container to 80 on the docker host. You could now access and verify the application is running by using one of the links below

- http://localhost:8000 if using Docker Desktop
- http://DOCKERHOST_IP:8000 use IP/Hostname of Docker Host if created using a VM, or a cloud Server.

Step 5: So far you started with a base image, added your application and tested it by running it. The dev container that you created contains the application built and ready. Commit the container's changes that you have made so far into an image as,

```
docker container commit dev schoolofdevops/aspnetapp:v1
```

where,

- `dev` is the name of the container you had created to build the app
- `schoolofdevops/aspnetapp:v1` is the image tag. Ensure you replace `schoolofdevops` with your registry (e.g. DockerHub) id .

> Credentials are not validated at the time of committing the image which is local, but when you try to push it to the registry.

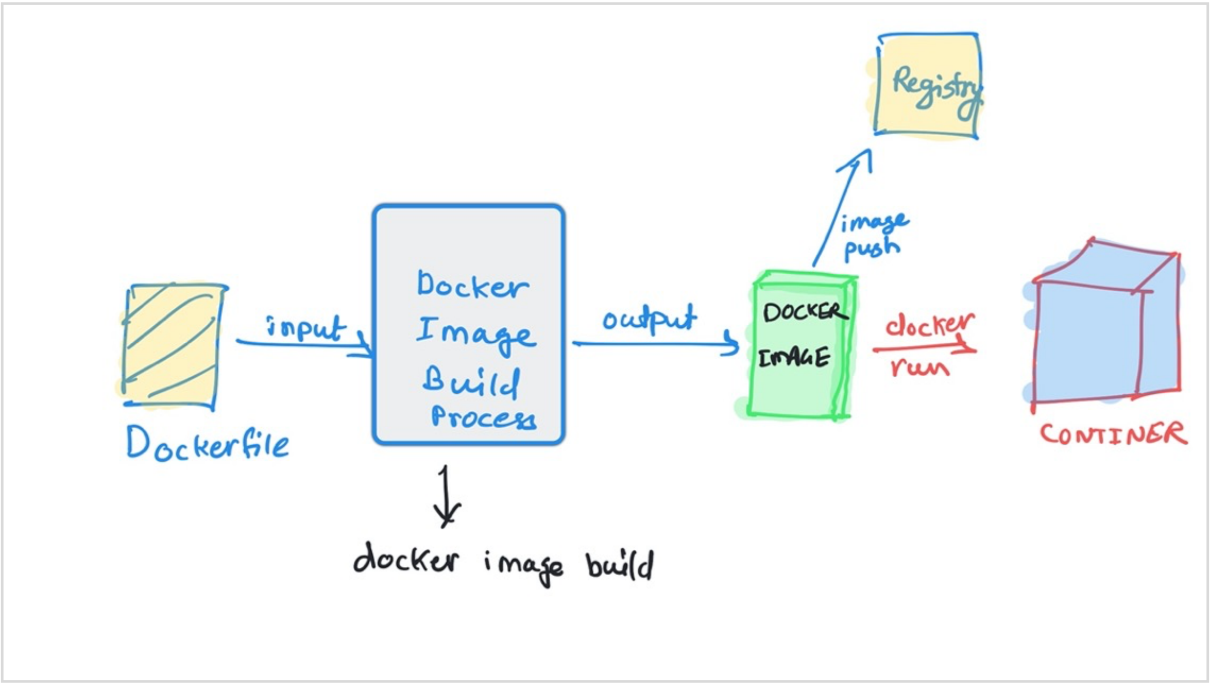
You can list examine the image created with the following commands,

```
docker image ls
docker image history schoolofdevops/aspnetapp:v1
```

Automate image builds with a Dockerfile

Docker provides a way to codify the image build process. Think of it as the above process converted into code using a declarative language.

Following diagram depicts the use of Docker file.



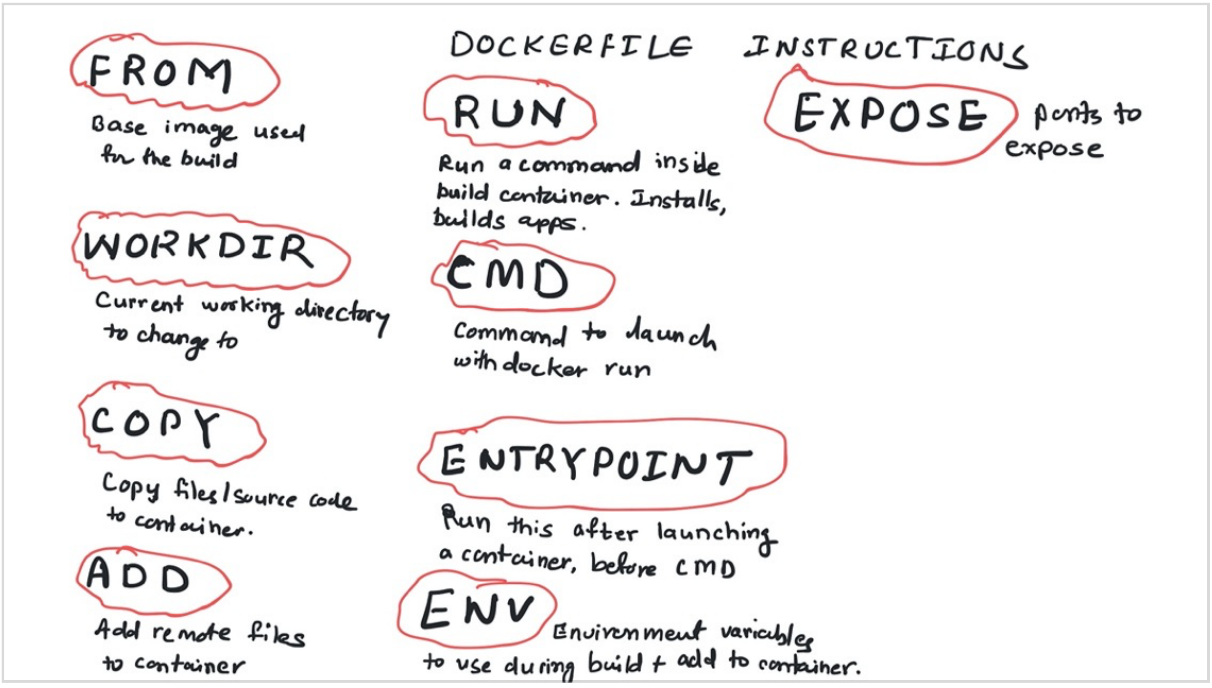
- Dockerfile is fed to the image build process, which is run with `docker image build` command.
- `docker image build` uses Dockerfile as a input, reads each of the instructions, take action based on that and builds the image, one layer at a time.
- The image built is stored locally. This image can then be tagged and pushed to a registry.
- This image then can be used to launch a container.

Dockerfile has its own syntax of the following type

```
INSTRUCTION  arguments
```

Some of the key instructions in Dockerfile are given as follows,

- FROM : defines the base image
- WORKDIR: defines the directory from which subsequent commands are run (e.g. COPY, CMD, ENTRYPOINT etc.)
- COPY : used to copy files to containers
- RUN: run a command, typically installing something inside the container or building an application
- ENV: define environment variables which are available while building as well as while running the container using the image built
- EXPOSE: defines which port your application would listen to
- ENTRYPOINT: defines a command or a script to run before launching the actual command/application
- CMD: defines the command /application/ process to start while launching a container with image built with this Dockerfile


DOCKERFILE INSTRUCTIONS
FROM
Base image used for the build
WORKDIR
Current working directory to change to
COPY
Copy files/source code to container.
ADD
Add remote files to container
RUN
Run a command inside build container. Installs, builds apps.
CMD
Command to launch with docker run
ENTRYPOINT
Run this after launching a container, before CMD
ENV
Environment variables to use during build + add to container.
EXPOSE
ports to expose

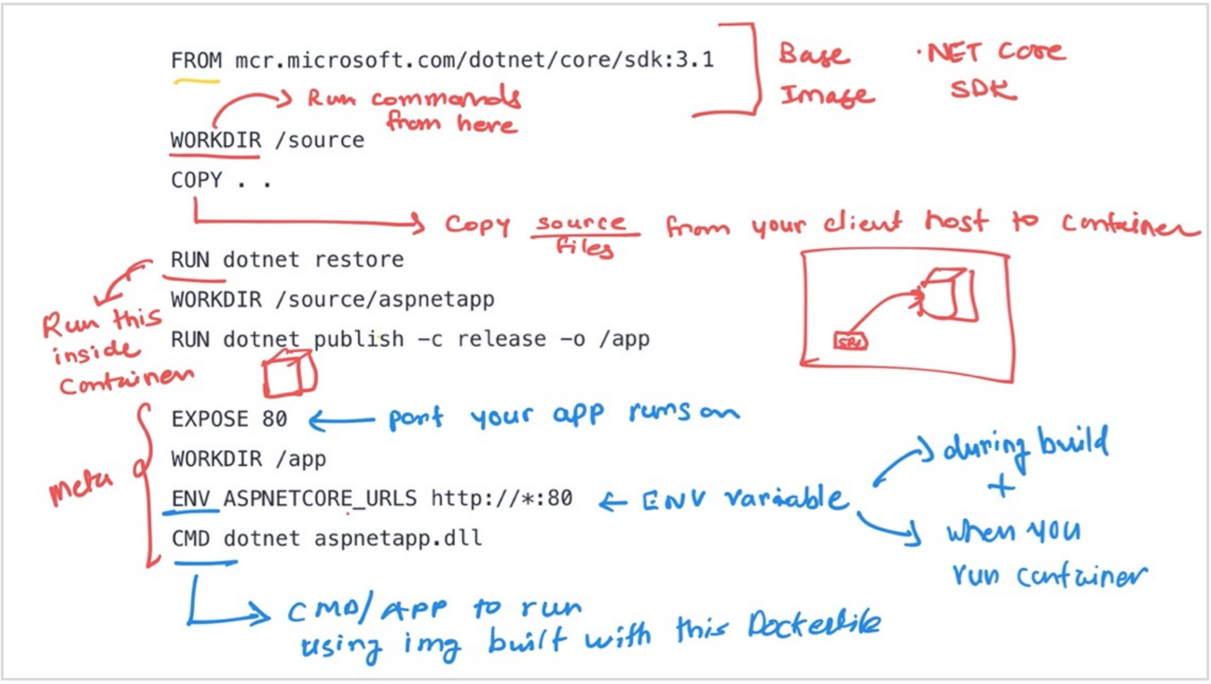
Constructing a Dockerfile

Lets now convert the process above into a Dockerfile using instructions above. This essentially means,

- Converting all of the steps in the manual process, (except for cloning the source from GitHub, as its assumed that its already done, and is out side of the actual application build process.

- Adding metadata that could be used while launching the container. This metadata is of no use during the actual build process.

Following is a annotated version of the Dockrfile



```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1

WORKDIR /source
COPY . .

RUN dotnet restore
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app

EXPOSE 80
WORKDIR /app
ENV ASPNETCORE_URLS http://*:80
CMD dotnet aspnetapp.dll
```

Base Image
.NET Core SDK
Run commands from here
Copy source files from your client host to container
Run this inside container
meta
port your app runs on
ENV variable
during build + when you run container
CMD/APP to run using img built with this Dockerfile

Building image with Dockerfile

Watch my video lessons to understand how to construct a Dockerfile. Try writing it yourself. Only refer to the following code and copy it if you have issues with your's.

```
File: dotnetcore-samples/aspnetapp/Dockerfile
```

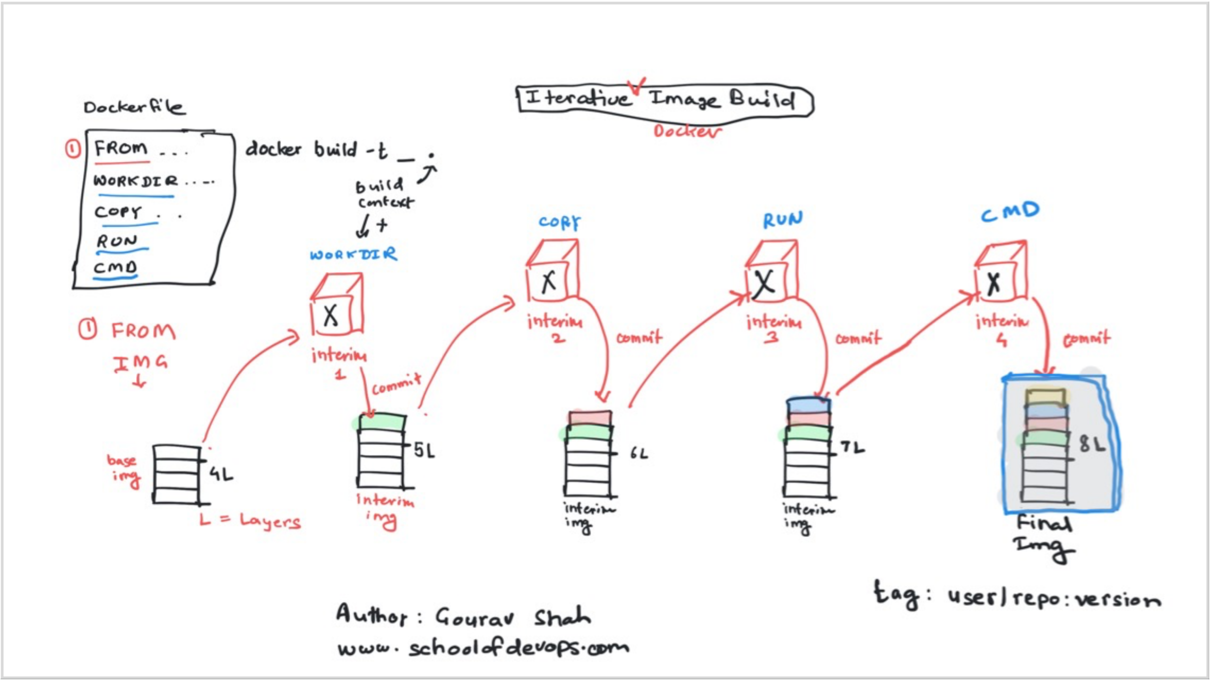
```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1


WORKDIR /source

COPY . .


RUN dotnet restore

WORKDIR /source/aspnetapp

RUN dotnet publish -c release -o /app
```

```
EXPOSE 80
WORKDIR /app
ENV ASPNETCORE_URLS http://*:80
CMD dotnet aspnetapp.dll
```

To build an image with this file use the following command,

```
docker build -t schoolofdevops/aspnetapp:v2 .
```

Even though docker build follows a similar sequence of tasks as the manual test build above, it takes a iterative approach towards building this image as shown in the image below.



When you launch a image build command this is what happens

1. Docker would read the FROM instruction and using the image defined with it as a base, would launch the first intermediate container.
2. It would copy all the files in the build context to the docker daemon and in turn to the intermediate container. These files are then available throughout the build process as this container gets committed into intermediate images for each step.

3. Docker daemon then would read the next instruction (just one) in the Dockerfile and take action based on it e.g. copy file, run commands, define metadata etc.
4. Immediately after step 3 here, it would commit the changes into an image, and delete the intermediate container created earlier. This is what adds a new layer to the image.
5. This process is then repeated until all instruction in the Dockerfile are processed, one at a time, creating and deleting intermediate containers and building intermediate images.
6. At the end of the final instruction, the image is tagged with the option that you provided with `-t` flag to `docker build` command.

   You could observe the difference between the image created by Dockerfile and the one you test built earlier using the following command

```
docker image history schoolofdevops/aspnetapp:v1


docker image history schoolofdevops/aspnetapp:v2
```

Publishing images to the Registry

Now that you have built the images, its time to publish those to registry. In order for you to be able to publish this image, you need to,

- Authenticate with the registry
- Be authorised to publish image to the user/org you have used in the tag e.g. if I am trying to push an image with tag schoolofdevops/aspnetapp:v2 , I should have access to schoolofdevops organisation/user account. This is the reason its important to tag your images properly before you attempt to push.

You could also tag one of your images as `latest` explicitly, as `latest` is not an implicit tag, and is not automatically set/updated. You must set it by yourself by pointing to one of the image versions. Following commands would take you through the process of authenticating, tagging and publishing your images to the registry.

```
[try to push the image]
docker image push schoolofdevops/aspnetapp:v1
```

```
[login to DockerHub with your docker ID]
docker login

[push image with v1 version]
docker image push schoolofdevops/aspnetapp:v1

[tag v2 as latest]
docker image tag schoolofdevops/aspnetapp:v2 schoolofdevops/aspnetapp:latest

[push images with all tags/versions]
docker image push schoolofdevops/aspnetapp
```

By the end of this process, you should see the images appear on the registry. If you are using Docker Hub, you could go to the web console and validate the presence of your images by checking listing your images.

#docker/windows/labs