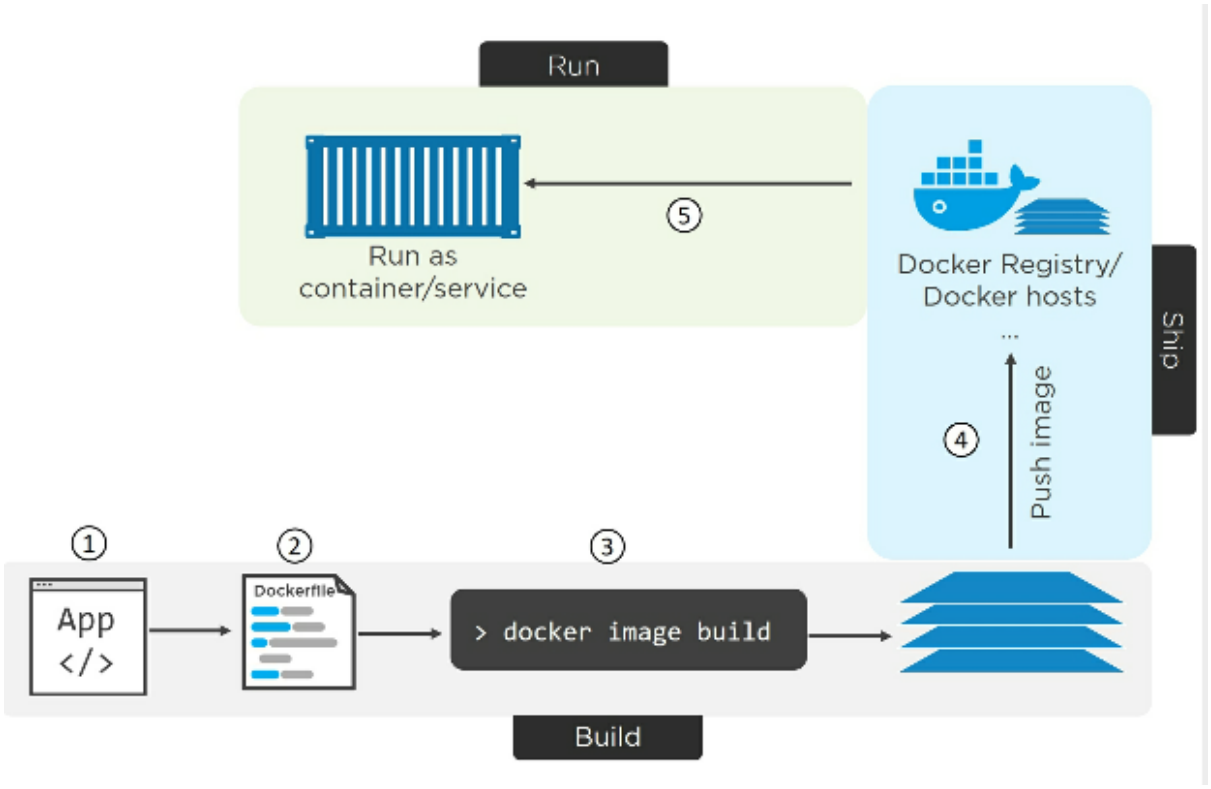
# Containerizing / Dockerizing

The process of taking an application and configuring it to run as a container is called "containerizing". Sometimes we call it "Dockerizing".

Basic flow of Containerizing an App




**High Level Steps to Containerize a Single-Container App:**

1. Get the App Code
2. Containerize the App
3. Run the App
4. Test the App
5. A few best practices

1. **Getting the Application Code :**

   The application used in this example can be cloned form GitHub:
   https://github.com/beginners-sameer/psweb.git
   Clone the sample app from GitHub.

   Command : **git clone** [https://github.com/beginners-sameer/psweb.git](https://github.com/beginners-sameer/psweb.git)

Change the directory to psweb folder

Now if you type “ls –l” (without quotes), it will list all the files

So this directory all the files

```
ubuntu@ip-172-31-31-10:~/psweb$ ls -l
total 28
-rw-rw-r-- 1 ubuntu ubuntu  338 Aug 29 03:59 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu  370 Aug 29 03:59 README.md
-rw-rw-r-- 1 ubuntu ubuntu  341 Aug 29 03:59 app.js
-rw-rw-r-- 1 ubuntu ubuntu  216 Aug 29 03:59 circle.yml
-rw-rw-r-- 1 ubuntu ubuntu  421 Aug 29 03:59 package.json
drwxrwxr-x 2 ubuntu ubuntu 4096 Aug 29 03:59 test
drwxrwxr-x 2 ubuntu ubuntu 4096 Aug 29 03:59 views
ubuntu@ip-172-31-31-10:~/psweb$
```

The above psweb directory contains all the files related to our application source code.

Please Look and have the feel about app.js file

Please type “cat app.js” to see the file content.

```
ubuntu@ip-172-31-31-10:~/psweb$ cat app.js
// Sample node.js web app for Pluralsight Docker CI course
// For demonstration purposes only
'use strict';

var express = require('express'),
    app = express();

app.set('views', 'views');
app.set('view engine', 'pug');

app.get('/', function(req, res) {
    res.render('home', {
  });
});

app.listen(8080);
module.exports.getApp = app;
ubuntu@ip-172-31-31-10:~/psweb$
```

So Our main application code is this file “app.js”

Main aim is to dockerize this app.js file into containers.

**Inspecting the Dockerfile:**

If you see the PSWEB directory, you will see the file called “Dockerfile”

```
total 28
-rw-rw-r-- 1 ubuntu ubuntu  338 Aug 29 03:59 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu  370 Aug 29 03:59 README.md
-rw-rw-r-- 1 ubuntu ubuntu  341 Aug 29 03:59 app.js
-rw-rw-r-- 1 ubuntu ubuntu  216 Aug 29 03:59 circle.yml
-rw-rw-r-- 1 ubuntu ubuntu  421 Aug 29 03:59 package.json
drwxrwxr-x 2 ubuntu ubuntu 4096 Aug 29 03:59 test
drwxrwxr-x 2 ubuntu ubuntu 4096 Aug 29 03:59 views
ubuntu@ip-172-31-31-10:~/psweb$
```

# So What is this Docker File???????

This is the file that describes the application and tells Docker how to build it into an image.

**What is Build Context?**

The Directory containing the Application code is referred to as build context. In our example, our build context directory is below one.

```
ubuntu@ip-172-31-31-10:~/psweb$ ls -l
total 28
-rw-rw-r-- 1 ubuntu ubuntu  338 Aug 29 03:59 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu  370 Aug 29 03:59 README.md
-rw-rw-r-- 1 ubuntu ubuntu  341 Aug 29 03:59 app.js
-rw-rw-r-- 1 ubuntu ubuntu  216 Aug 29 03:59 circle.yml
-rw-rw-r-- 1 ubuntu ubuntu  421 Aug 29 03:59 package.json
drwxrwxr-x 2 ubuntu ubuntu 4096 Aug 29 03:59 test
drwxrwxr-x 2 ubuntu ubuntu 4096 Aug 29 03:59 views
ubuntu@ip-172-31-31-10:~/psweb$
```

Dockerfile starts with Capital “D”

Let’s see what is the content of Dockerfile???? See the below screenshot

```
ubuntu@ip-172-31-31-10:~/psweb$ cat Dockerfile
# Test web-app to use with Pluralsight courses and Docker Deep Dive book
# Linux x64
FROM alpine

LABEL maintainer="nigelpoulton@hotmail.com"

# Install Node and NPM
RUN apk add --update nodejs nodejs-npm

# Copy app to /src
COPY . /src

WORKDIR /src

# Install dependencies
RUN  npm install

EXPOSE 8080

ENTRYPOINT ["node", "./app.js"]
```

Below is the content of Dockerfile:

```
FROM alpine

LABEL maintainer="nigelpoulton@hotmail.com"

RUN apk add --update nodejs nodejs-npm

COPY . /src

WORKDIR /src

RUN npm install

EXPOSE 8080

ENTRYPOINT ["node", "./app.js"]
```

**There are certain keywords which you have to look for.  Marked in Red color Font.**

**Two things are done by Dockerfile. What are they ?**

## Dockerfile Describes the Application

## Dockerfile tells Docker how to containerize the application i.e, creating an image

Highlevel about what is happening with this particular Dockerfile??? :::

1. It starts with the Alpine Image
2. Adds the maintainer. i.e,who is managing this Dockerfile
3. Install Node.js and NPM
4. Copy in the application code to /src
5. Set the Working Directory
6. Install Dependancies
7. Document the App's network port
8. Set App.js as the default application to run

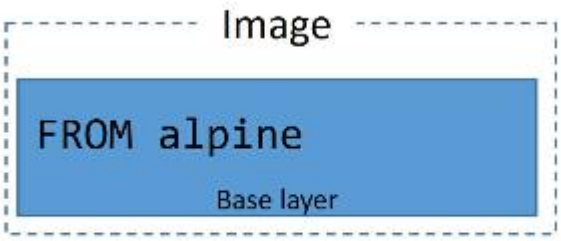
# Let's talk about in Detail about Dockerfile:

All Dockerfile starts with FROM instruction. This will be the base layer of the image

This particular app is Linux App. So FROM instruction should be Linux Based Image.

If you are containerizing a Windows application, you will need to specify the appropriate Windows base image - such as microsoft/aspnetcore-build.

At this point, layer will look like this :



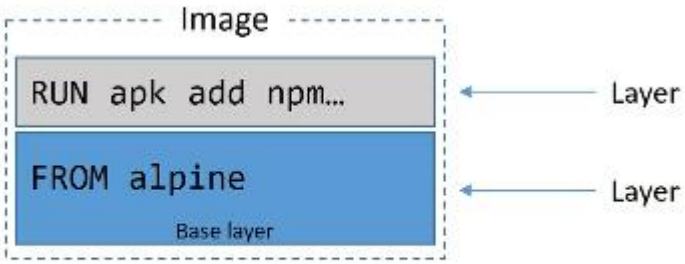

Next, the Dockerfile creates a LABEL that specifies “nigelpoulton@hotmail.com” as the maintainer of the image.

It’s considered a best practice to list a maintainer of an image so that other potential users have a point of contact when working with it.

Note: Maintainer doesn’t do anything. It just adds the metadata. i.e. it just gives you some information. That’s all. Hence layer will not be formed.

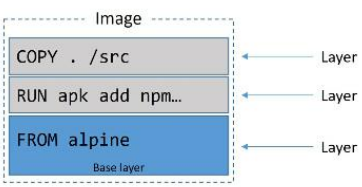
Next –

The RUN apk add --update nodejs nodejs-npm instruction uses the Alpine apk package manager to install nodejs and nodejs-npm into the image. The RUN instruction installs these packages as a new image layer on top of the alpine base image created by the FROM alpine instruction.




Next :

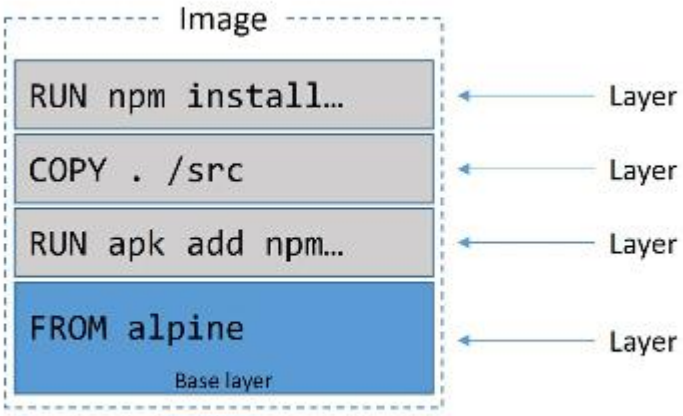
The COPY . /src instruction copies in the app files from the build context. It copies these files into the image as a new layer. The image now has three layers



Next, the Dockerfile uses the WORKDIR instruction to set the working directory for the rest of the instructions in the file.

This also doesn't add any layer. Only the metadata.

Next, Then the RUN npm install instruction uses npm to install application dependencies listed in the package.json file in the build context. It runs within the context of the WORKDIR set in the previous instruction, and installs the dependencies as a new layer in the image. The image now has four layers




Next - The application exposes a web service on TCP port 8080, so the Dockerfile documents this with the EXPOSE 8080 instruction. This is added as image metadata and not an image layer.

Finally, the ENTRYPOINT instruction is used to set the main application that the image (container) should run. This is also added as metadata and not an image layer.

# Now we have done with the Dockefile. What's Next Step >>>>> It is to build the image .

Dockefile -> build the image -> Push the image to the dockerhub -> run the container

Type the below command which will convert your Dockerfile into an image. Make sure you don't forget to type "." at the end of the command.

```
ubuntu@ip-172-31-31-10:~/psweb$ docker image build -t web:latest .
```

Now check the image whether it has been created or not by running the below command

```
ubuntu@ip-172-31-31-10:~/psweb$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
web                 latest              84effb125361        35 seconds ago      71.5MB
alpine              latest              961769676411        8 days ago          5.58MB
```

As you can see it has two images.

Alpine image has been downloaded first from the Dockerfile (FROM instruction) and then the image which we have given the name as web:latest which has been created.

# Congratulations… Your Nodejs APP is containerized.

**Pushing Images:**

Once you've created an image, it's a good idea to store it in an image registry to keep it safe and make it available to others. Docker Hub is the most common public image registry, and it's the default push location for docker image push commands.

In order to push an image to Docker Hub, you need to login with your Docker ID. You also need tag the image appropriately.

Type the below command to enter your dockerhub userid and password

```
ubuntu@ip-172-31-31-10:~/psweb$ docker login
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
ubuntu@ip-172-31-31-10:~/psweb$
```

After this has been done, you need to tag your image in proper format.

Before you type the new tag name, please see your current images

```
ubuntu@ip-172-31-31-10:~/psweb$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
web                 latest              84effb125361        9 minutes ago       71.5MB
alpine              latest              961769676411        8 days ago          5.58MB
ubuntu@ip-172-31-31-10:~/psweb$
```

Now we are going to tag with the new name with the below command

docker image tag web:latest awssameerzulfi/web:latest

the above command, you need to put your dockerhubid inplace of my id.

Now see the below screenshot to understand more.

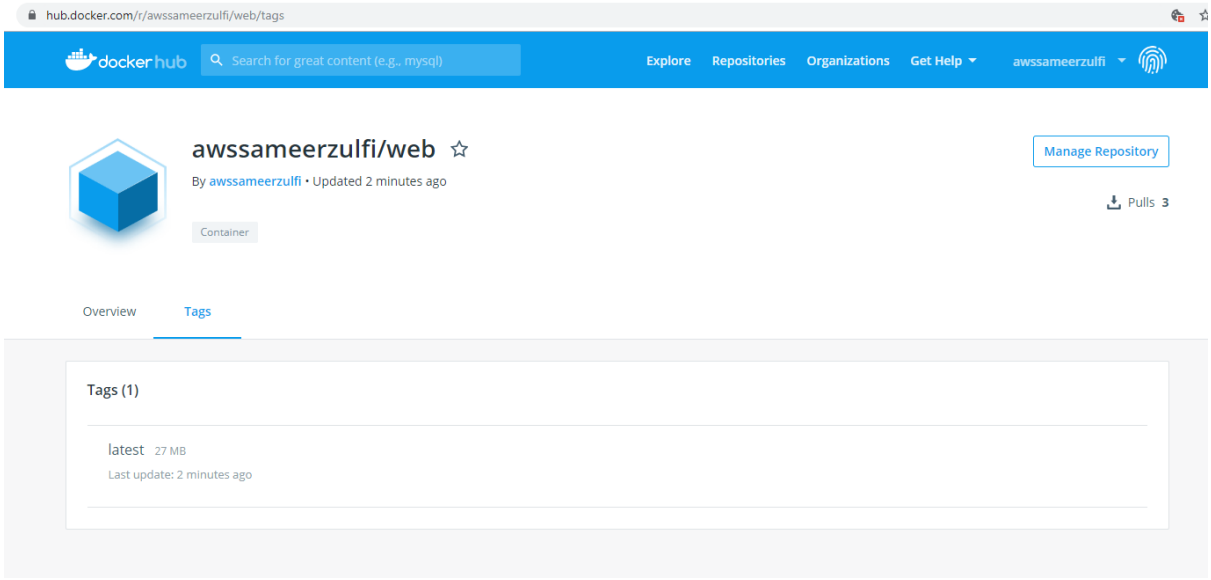
```
ubuntu@ip-172-31-31-10:~/psweb$ docker image tag web:latest awssameerzulfi/web:latest
ubuntu@ip-172-31-31-10:~/psweb$ docker images
REPOSITORY           TAG                 IMAGE ID            CREATED             SIZE
awssameerzulfi/web   latest              84effb125361        10 minutes ago      71.5MB
web                  latest              84effb125361        10 minutes ago      71.5MB
alpine               latest              961769676411        8 days ago          5.58MB
ubuntu@ip-172-31-31-10:~/psweb$
```

Now we can push the image to dockerhub

docker image push awsssameerzulfi/web:latest

```
alpine              latest              961769676411        8 days ago          5.58MB
ubuntu@ip-172-31-31-10:~/psweb$ docker image push awssameerzulfi/web:latest
The push refers to repository [docker.io/awssameerzulfi/web]
aa7d652ab2d9: Pushing [==============>                                    ]  6.515MB/20.57MB
4a59628c456d: Pushing [==================================================>]  81.41kB
ccef222f967c: Pushing [============================>                      ]  25.24MB/45.3MB
03901b4a2ea8: Layer already exists
```

Great Job.. You have succesffully pushed the image to your DockerHub Repository.




Now, Let's run the App : Means.. let's run the container using this particular image

```
ubuntu@ip-172-31-31-10:~/psweb$ docker container run -d --name C1 -p 80:8080 web:latest
```

The Container will start running in the background. That's why we have put –d means.. run in the background.

What is the container name we have given ???? C1 is the name

What is this –p does ??? it publishes the port. i.e, it will do a communication between Host Port : Container Port

80:8080 means Host port 80 : Container Port 8080

If I type 9000:800 means.. Host port 9000 : Container Port 800

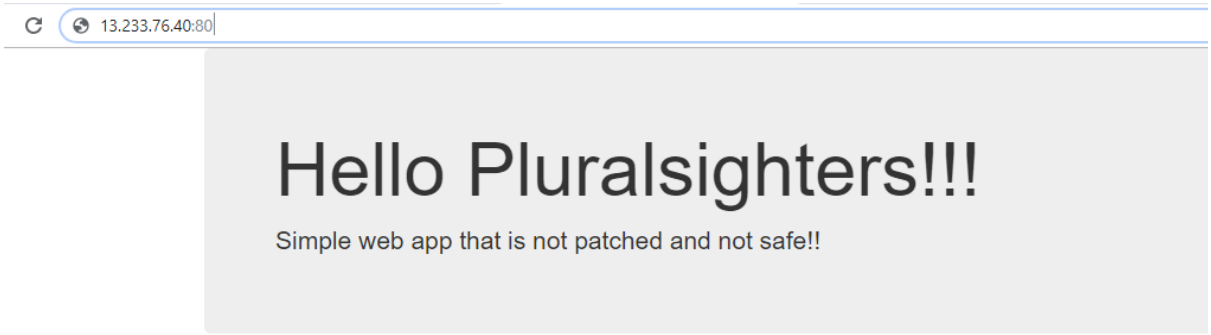
We have to mention this because, When we type PublicIP:9000, the traffic will hit the host port and it will redirect all the traffic to 800 port inside the container.

Let's see this in action.

```
ubuntu@ip-172-31-31-10:~/psweb$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
d1f7b6db6a23        web:latest          "node ./app.js"     3 minutes ago       Up 3 minutes        0.0.0.0:80->8080/tcp    C1
ubuntu@ip-172-31-31-10:~/psweb$
```

Good one container is running.

Let’s open the browser. And see if the traffic hits the host port 80, it automatically redirects to 8080 inside the container port.




Congratulation. Now Application is containerized and running.

Few Commands to run and check :

docker image history web:latest

docker image inspect web:latest