Docker Images

Lab – Dockerfiles I

Docker can build images automatically by reading the instructions in a Dockerfile. A Dockerfile is a text file that
contains all of the commands you would normally execute manually in order to build a Docker image from a container via
`docker container commit`. By calling `docker image build` from your terminal, you can have Docker construct the image
for you, sequentially executing the instructions in your Dockerfile.

In this lab you will create a Dockerfile to specify a simple service image which we will use to host a web server. We
will name the repository “websvr”.

1. Setup a Dockerfile context directory

Dockerfiles are usually placed at the root of a folder hierarchy containing everything you need to build one or more
Docker images. This folder is called the build context. The entire build context will be packaged and sent to the Docker
Engine for building. In simple cases the build context will have nothing in it but a Dockerfile.

To create a build context (directory) on your lab system, simply create a new directory. We will create a directory
called websvr to host our Docker websvr project.

```
user@ubuntu:~$ mkdir websvr

user@ubuntu:~$ cd websvr

user@ubuntu:~/websvr$
```

2. Create a Dockerfile

A Dockerfile contains a list of instructions for the `image build` subcommand to perform when creating the target image.
We will build the following Dockerfile:

```
FROM ubuntu:12.04
LABEL Maintainer docker lab
RUN apt-get update && \
    apt-get install -y apache2 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

Dockerfile statements can be loosely organized into three groups:

- File System Setup – Steps that copy or install files into the image's file system (programs, libraries, etc)
- Image Metadata – Descriptive image data outside of the filesystem (exposed ports, labels, etc.)
- Execution Metadata – Commands and environment settings that tell Docker what to do when users run the image

Our Dockerfile has three statements in the Image Metadata category, FROM, MAINTAINER and EXPOSE. Dockerfiles begin
with a FROM statement which defines the parent image for the new image. We will base our web server on the "Ubuntu
12.04" image. You can create images with no parent by using the reserved name "scratch" as the parent image (e.g. FROM
scratch). Dockerfiles also usually contain a MAINTAINER statement defining the author and/or collaborators (MAINTAINER
was deprecated in v1.13 but is still widely used). The EXPOSE instruction describes ports that the container's
service(s) typically listen on.

The File System Setup section of our Dockerfile will need to run `apt-get` commands to update the system and install
apache2.

The Execution Metadata section of our Dockerfile will need to setup environment variables for the user and group for
the Apache Web Server to run under. This section will also need to define the port to run the Apache web server.

Create a `Dockerfile` for your Apache Web Server in the `websvr` directory as per below:

```
user@ubuntu:~/websvr$ vi Dockerfile

user@ubuntu:~/websvr$ cat Dockerfile

FROM ubuntu:12.04
LABEL Maintainer docker lab

RUN apt-get update \
&& apt-get install -y apache2 \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]

user@ubuntu:~/websvr$
```

The apache2 binary runs the Apache Web Server (in Debian packaging), the *-D* switch passes parameters and the
FOREGROUND parameter tells the web server not to call `setsid()`, keeping it from creating a new shell session
(important if you want the console output to appear in the Docker logs).

Use the Dockerfile reference to look up any commands you are not familiar with from class:
https://docs.docker.com/engine/reference/builder/

3. Build the image via Dockerfile

The `image build` subcommand reads a Dockerfile and creates images from it. The build command provides a *--tag*
switch which you can use to give your new image a repository and tag name.

Build your websvr image as follows:

```
user@ubuntu:~/websvr$ docker image build --tag="lab/websvr:0.2.0" .
...

user@ubuntu:~/websvr$ docker image ls lab/websvr:0.2.0

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZ
lab/websvr          0.2.0               56a4a9dfccbf        41 seconds ago      150

user@ubuntu:~/websvr$
```

Note the size of your new image. Not bad for an Apache web server with a complete Linux distro and configuration. Use
the `docker history` subcommand to display the various image layers in your application's image ancestry.

```
user@ubuntu:~/websvr$ docker image history lab/websvr:0.2.0

IMAGE               CREATED             CREATED BY
56a4a9dfccbf        About a minute ago   /bin/sh -c #(nop)  CMD ["/usr/sbin/apache2
83cd401962c9        About a minute ago   /bin/sh -c #(nop)  EXPOSE 80/tcp
2898db3cc5e8        About a minute ago   /bin/sh -c #(nop)  ENV APACHE_LOG_DIR=/var
9e2f141004a3        About a minute ago   /bin/sh -c #(nop)  ENV APACHE_RUN_GROUP=ww
b591f707adce        About a minute ago   /bin/sh -c #(nop)  ENV APACHE_RUN_USER=www
9e6326bb261f        About a minute ago   /bin/sh -c apt-get update && apt-get insta
b0592e29192c        About a minute ago   /bin/sh -c #(nop)  LABEL Maintainer docker
b384dd9703db        8 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]
<missing>           8 days ago          /bin/sh -c mkdir -p /run/systemd && echo '
<missing>           8 days ago          /bin/sh -c sed -i 's/^#\s*\(deb.*universe\
<missing>           8 days ago          /bin/sh -c rm -rf /var/lib/apt/lists/*
<missing>           8 days ago          /bin/sh -c set -xe   && echo '#!/bin/sh' >
<missing>           8 days ago          /bin/sh -c #(nop) ADD file:494afaaca485a97

user@ubuntu:~/websvr$
```

Older versions of Docker displayed IDs for all images. Content addressable images no longer use parent images to chain
image layers together, rather the image manifest knows all of the metadata and all of the required filesystem layers.
Docker v1.10 and later will show "" for image layers without manifests of their own. This was perhaps a poor
wording choice, the image data is not missing, just the manifest and its ID.

Now run the `docker image ls` subcommand to display the images on the Docker host.

- Which images in your application's ancestry are tagged?
- How can you display the untagged images with the `docker image ls` command?
- What does the (nop) comment in the build history mean?
- Do all of the images in the application's ancestry have filesystem layers?
- How can you tell which images have filesystem layers from the history display?
- Which environment variables are defined in the image ancestry?

Use the `image inspect` subcommand to display the environment variables your image will

configure for new containers:

```
user@ubuntu:~/websvr$ docker image inspect -f '{{json .ContainerConfig.Env}}' lab/w

[
  "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
  "APACHE_RUN_USER=www-data",
  "APACHE_RUN_GROUP=www-data",
  "APACHE_LOG_DIR=/var/log/apache2"
]

user@ubuntu:~/websvr$
```

- Can you find an image in the build history for each of these definitions?
- Who set the PATH?

The PATH is the one environment variable we did not configure. When a new container is created, Docker will set the
following environment variables automatically:

- PATH - Includes popular directories (/usr/local/sbin:/usr/local/bin:…)
- HOME - Set based on the value of USER
- HOSTNAME - The hostname associated with the container
- TERM - xterm if the container is allocated a pseudo-TTY

The PATH variable is placed in the image's metadata, however the HOST, HOSTNAME and TERM variables change with each run
of the image and are configured by Docker dynamically.

4. Run a container using the new image

Like any other image, a Dockerfile generated image can be run with the `container run` subcommand and we can use the
-p switch to map ports from the container to the host interface. Run your image, mapping the container's port 80
(where Apache listens by default) to the host's port 8989:

```
user@ubuntu:~/websvr$ docker container run -d -p 8989:80 lab/websvr:0.2.0

47164ddbb595e0575c8bf6418778a37058e9f69241ed2408b71ce54d0320d103

user@ubuntu:~/websvr$
```

Verify that your web server container is running:

```
user@ubuntu:~/websvr$ docker container ls

CONTAINER ID        IMAGE               COMMAND                  CREATED
47164ddbb595        lab/websvr:0.2.0    "/usr/sbin/apache2..."   23 seconds ago
7b8d0dc8c5f0        registry:2          "/entrypoint.sh /e..."   21 minutes ago

user@ubuntu:~/websvr$
```

Now try to reach the web server on port 80 from the host:

```
user@ubuntu:~/websvr$ wget -O- http://localhost

--2017-03-07 22:24:05--  http://localhost/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:80... failed: Connection refused.
Connecting to localhost (localhost)|127.0.0.1|:80... failed: Connection refused.

user@ubuntu:~/websvr$
```

This fails because Apache is running on port 80 in the container. The container's ports are not directly accessible from
the host. Use the `docker port` subcommand to display the container port mappings for your web server (you will need to
substitute the ID of your own web server container):

```
user@ubuntu:~/websvr$ docker container port 4716

80/tcp -> 0.0.0.0:8989

user@ubuntu:~/websvr$
```

Now try to reach the web server from the host port 8989:

```
user@ubuntu:~/websvr$ wget -qO- http://localhost:8989

<html><body><h1>It works!</h1>
<p>This is the default web page for this server.</p>
<p>The web server software is running but no content has been added, yet.</p>
```

```
</body></html>

user@ubuntu:~/websvr$
```

Success! You now have a running service based on an image you created from a Dockerfile.

5. Rebuild a Dockerfile/Context

The Docker build cache avoids the long waits associated with building the same image successively. If the Docker Engine
sees a Dockerfile instruction with a string and parent image ID that it has already built, it will use the preexisting
image and skip the build step. To demonstrate we will add a LABEL instruction to our existing Dockerfile and rebuild it.

LABELs allow you to add arbitrary metadata to a Docker image. Make the following change to your Dockerfile to add the
"release-notes" label:

```
user@ubuntu:~/websvr$ vi Dockerfile

user@ubuntu:~/websvr$ cat Dockerfile

FROM ubuntu:12.04
LABEL Maintainer docker lab

RUN apt-get update \
&& apt-get install -y apache2 \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
LABEL release_notes="this is a test release"
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]

user@ubuntu:~/websvr$
```

Be sure to add only the release_notes LABEL instruction without tampering with any of the other lines in the file.

Rebuild your image stack with the new 0.3.0 tag:

```
user@ubuntu:~/websvr$ docker image build --tag="lab/websvr:0.3.0" .

Sending build context to Docker daemon 2.048 kB
Step 1/9 : FROM ubuntu:12.04
 ---> b384dd9703db
Step 2/9 : LABEL Maintainer docker lab
 ---> Using cache
 ---> b0592e29192c
Step 3/9 : RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm
 ---> Using cache
 ---> 9e6326bb261f
Step 4/9 : ENV APACHE_RUN_USER www-data
 ---> Using cache
 ---> b591f707adce
Step 5/9 : ENV APACHE_RUN_GROUP www-data
 ---> Using cache
 ---> 9e2f141004a3
Step 6/9 : ENV APACHE_LOG_DIR /var/log/apache2
 ---> Using cache
 ---> 2898db3cc5e8
Step 7/9 : LABEL release_notes "this is a test release"
 ---> Running in 9f6266ca3f7c
 ---> 01630ea9b0e6
Removing intermediate container 9f6266ca3f7c
Step 8/9 : EXPOSE 80
 ---> Running in f4ee699c3a6e
 ---> bc4745f3076a
Removing intermediate container f4ee699c3a6e
Step 9/9 : CMD /usr/sbin/apache2 -D FOREGROUND
 ---> Running in 4f1dab3dd64e
 ---> 6fe00b1b5d6f
Removing intermediate container 4f1dab3dd64e
Successfully built 6fe00b1b5d6f

user@ubuntu:~/websvr$
```

- Why did the first five steps use the cache?
- Why did the unchanged EXPOSE and CMD lines rebuild?

Use the `image inspect` subcommand to view your new LABEL metadata:

```
user@ubuntu:~/websvr$ docker image inspect -f '{{.ContainerConfig.Labels.release_no

this is a test release

user@ubuntu:~/websvr$
```

Examine the new image history:

```
user@ubuntu:~/websvr$ docker image history lab/websvr:0.3.0

IMAGE               CREATED             CREATED BY
6fe00b1b5d6f        49 seconds ago      /bin/sh -c #(nop)  CMD ["/usr/sbin/apache2.
bc4745f3076a        49 seconds ago      /bin/sh -c #(nop)  EXPOSE 80/tcp
01630ea9b0e6        49 seconds ago      /bin/sh -c #(nop)  LABEL release_notes=thi.
2898db3cc5e8        8 minutes ago       /bin/sh -c #(nop)  ENV APACHE_LOG_DIR=/var.
9e2f141004a3        8 minutes ago       /bin/sh -c #(nop)  ENV APACHE_RUN_GROUP=ww.
b591f707adce        8 minutes ago       /bin/sh -c #(nop)  ENV APACHE_RUN_USER=www.
9e6326bb261f        8 minutes ago       /bin/sh -c apt-get update && apt-get insta.
b0592e29192c        9 minutes ago       /bin/sh -c #(nop)  LABEL Maintainer docker.
b384dd9703db        8 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]
<missing>           8 days ago          /bin/sh -c mkdir -p /run/systemd && echo '.
<missing>           8 days ago          /bin/sh -c sed -i 's/^#\s*\(deb.*universe\.
<missing>           8 days ago          /bin/sh -c rm -rf /var/lib/apt/lists/*
<missing>           8 days ago          /bin/sh -c set -xe   && echo '#!/bin/sh' >.
<missing>           8 days ago          /bin/sh -c #(nop) ADD file:494afaaca485a97.

user@ubuntu:~/websvr$
```

Compare the IDs from this build to the prior build.

- Which images are the same between build 0.2.0 and build 0.3.0?
- How can you infer from the build history which images were built together?
- Run the `docker image ls -a` command, what order does it list images in?

Congratulations, you have completed the Dockerfile lab!