Docker Lab 3 -
Building on Ubuntu Images with Dockerfiles

Jun 16th, 2020

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Overview and Objectives

In the previous lab we discussed how to obtain a Ubuntu image, run a container based on the Ubuntu image, stop the container, and remove it. We also discussed how to add packages to a very thin Ubuntu container. We added packages on the command line after starting a Bash session on the running container. In this lab the focus is on how to automate these kinds of steps by augmenting the starting image with additional *layers* representing the steps we would have had to do every time we launched a container from the initial image. These steps are contained in a *Dockerfile*. The initial image, plus layers, will become a new image. And then we launch containers from the new image instead.

- Goal - Introduction to Dockerfiles
- Goal - A simple Dockerfile example with one layer
- Goal - Dockerfiles with entry points and custom entry script
- Goal - Pushing custom images to Docker Hub
- Goal - Using custom images from Docker Hub

2 A Simple Dockerfile Building on a Ubuntu Image

In the previous lab, we obtained a Ubuntu image and proceeded to start a container with a Bash session. We followed this by installing the `lsb-release` package so we could run the `lsb_release` command. This sequence of steps is shown below:

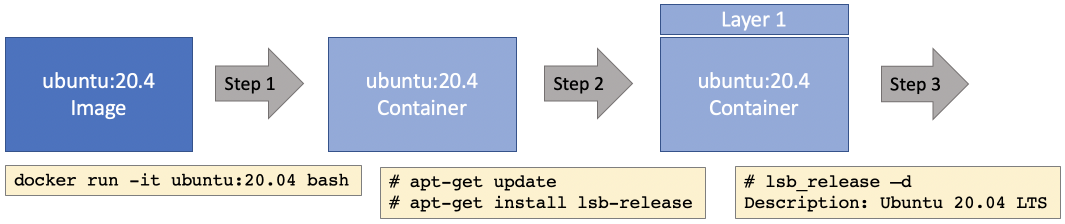

Figure 1: Step 1: A container is started from a basic Ubuntu 20.04 image pulled from DockerHub. Step 2: In a Bash session on the running container, the package `lsb-release` is installed, representing a new layer of the container. Step 3: After the package is installed, the `lsb_release` command can be run.

If we wanted to run the `lsb_release` command every time we start a new container from this image, we would have to install the `lsb-release` package every time the container starts. Instead, in our first `Dockerfile` example, we will create a new (local for now) image which is a combination of (a) the vanilla Ubuntu 20.04 image, plus (b) an extra layer which is related to the `lsb-release` package.

The Dockerfile can be thought of as a kind of script, and usually exists in its own folder. We're going to follow that convention:

```
$ mkdir ubuntu_lsb
$ cd ubuntu_lsb
$ emacs Dockerfile
```

Edit the `Dockerfile` to contain the following content:

```
# Filename: Dockerfile
FROM ubuntu:20.04
RUN apt-get update
RUN apt-get -y install lsb-release
```

The `FROM` instruction specifies the base image to use. All Dockerfiles must have a `FROM` instruction as the first non-comment instruction. The `RUN` instructions specify a shell command to execute inside the image. In this case, we are just installing the `lsb-release` package, to get the `lsb_release` command, [1]. We then build the image with:

```
$ docker build -t my_ubuntu .
$ docker images
REPOSITORY       TAG          IMAGE ID        CREATED          SIZE
my_ubuntu        latest       1391363edaf4    8 seconds ago    134MB
```

Then a container can be launched with the `lsb-release` package already installed:

```
$ docker run -it my_ubuntu bash
# root@def518a593c0:/# lsb_release -a
No LSB modules are available.
Distributor ID:        Ubuntu
Description:        Ubuntu 20.04 LTS
Release:        20.04
Codename:        focal
```

What have we accomplished? This is a first step in preparing an image, based on Ubuntu, with an additional layer containing the needed `lab-release`. We don't want to repeat the installation of this package every time we start a container, so we create this new image that has it baked in. The below figure compares this Dockerfile workflow to that in the previous image.

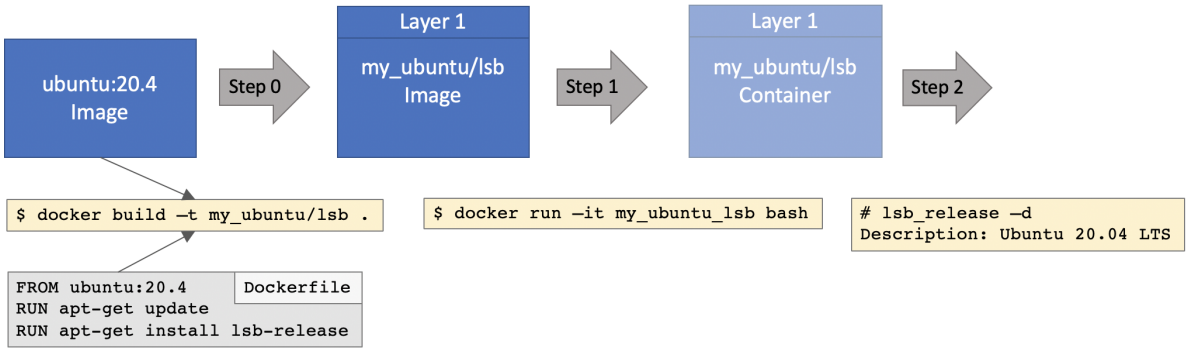

Figure 2: Step 1: A new image is created from a Ubuntu 20.4 image pull from Dockerhub. The Docker `build` command makes a new image adding the layers specified in the `Dockerfile`. Step 2: A container is created from the newly created image. Step 3: The `lsb_release` command can be run immediately since the `lsb-release` package was already installed.

3 Containers vs. Virtual Machines

It seems that most discussions about the difference between containers and virtual machines revolve around *implementation*, e.g., virtualized hardware vs. virtualized operating system. It's also worth discussing the difference between how they are *used*. Both support many use cases, so none of this discussion should be taken in the absolute sense. It's more of a personal opinion and reflection of the goal of these labs for building up a capability for working with containers, with a particular set of use cases in mind.

The most common experience I've encountered with virtual machines is the need to replicate a particular OS on one machine within another OS. Most typically it is a GNU/Linux machine running on either a Mac or Windows machine. The VM supports project work that lasts perhaps months or longer and the local file system of the VM is like any other work computer. A container however, is relatively short lived. Even though the container may be stopped, paused and restarted or unpaused while retaining changes to the local file system, the container is more likely viewed as a throw-away. The key reason for existence for a container is to provide services to a user or other piece of software. The internal state of the container is not important once the services are no longer needed, thus regarded as expendible. The idea is shown in Figure 3.

Figure 3: A *virtual machine* often may persist for months with users periodically starting a work session, saving files locally, and treating it just like any other work computer before logging out or stopping the VM. A *container* is more ephemeral, supporting a series of queries or a service for a short period of time, after which it may be stopped with no local information saved.

In the first trivial Dockerfile example above, the `lsb_release` command is run within a Bash session on a Ubuntu container. Since the interaction is initiated within a Bash session within the container, it feels more like the example on the left in Figure 3. In the next example, the container will be designed to interact from a Bash shell session in the host container, and the container will feel more like a service. This is a first step in our general progression of setting up containers as rather ephemeral entities that provide a service.

4 A Second Dockerfile Example - Cowsay

The second Dockerfile example is, the "cowsay" example borrowed from [1]. It uses the two packages, `cowsay` and `fortune`, which provide the `cowsay` and `fortune` commands respectively. The `fortune` command generates a short fortune or joke:

```
$ fortune
I used to think that the brain was the most wonderful organ in my body.
Then I realized who was telling me this.
                -- Emo Phillips
```

The `cowsay` command will take given text input and wrap it in a cute picture:

```
$ cowsay "We don't know who discovered water, but we're certain it wasn't a fish."
 ---------------------------------------
/  We don't know who discovered water, \
\ but we're certain it wasn't a fish.  /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

And the `cowsay` command can be fed the output of the `fortune` command using the command line pipe:

```
$ fortune | cowsay
 ____________________________________
/ Rule #1:                           \
|                                    |
| The Boss is always right.          |
|                                    |
| Rule #2:                           |
|                                    |
\ If the Boss is wrong, see Rule #1. /
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

In this section we will build an image with these two packages installed, and generate output like the above examples using a container built from this image.

We will begin by creating a new folder to contain a new Dockerfile:

```
$ mkdir ubuntu_cowsay
$ cd ubuntu_cowsay
$ emacs Dockerfile
```

Edit the `Dockerfile` to contain the following content:

```
# Filename: Dockerfile
FROM ubuntu:20.04
RUN apt-get update
RUN apt-get -y install cowsay fortune
ENV PATH "$PATH:/usr/games"
```

Note the last line of the Dockerfile: `ENV PATH "$PATH:/usr/games"`. The Ubuntu package manager regards both of these commands as "games", and therefore installs them in the `/usr/games/` folder. So in order to invoke these commands from Bash within containers made from this image, this folder needs to be added to the Bash path. We then build the image with:

```
$ docker build -t ubuntu_cowsay .
$ docker images
REPOSITORY       TAG          IMAGE ID        CREATED          SIZE
ubuntu_cowsay    latest       a49848e50195    3 seconds ago    143MB
```

Then a container can be launched with the `fortune` and `cowsay` packages already installed:

```
$ docker run -it --rm ubuntu_cowsay bash
# root@def518a593c0:/# fortune | cowsay
 ____________________________________
/ Do not sleep in a eucalyptus tree \
\ tonight.                           /
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note that the container is run with the `--rm` argument which will remove the container from our local system when this container is exited - a step toward using containers as ephemeral entities.

4.1 Creating an Entry Point

So far our example is not so different than the `lsb_release` example in that our work is similar to the left side of Figure 3. The work is carried out by (a) starting a container (b) interacting with a

Bash session in the container, and (c) exiting the container. We would like to eliminate the Bash session and migrate toward the scenario on the right in Figure 3.

To do this we will continue the "cowsay" example from [1], and create an *entry point* to our image, by adding another line to the Dockerfile:

```
# Filename: Dockerfile
FROM ubuntu:20.04
RUN apt-get update
RUN apt-get -y install cowsay fortune
ENV PATH "$PATH:/usr/games"
ENTRYPOINT ["/usr/games/cowsay"]                    <-- new line
```

The last `ENTRYPOINT` line of this file is new and it specifies the command to be run when this container is launched. As before, we build the image, overwriting the previous version:

```
$ docker build -t ubuntu_cowsay .
$ docker images
REPOSITORY       TAG      IMAGE ID        CREATED          SIZE
ubuntu_cowsay    latest   2525fd4e70be    39 seconds ago   143MB
```

Then a container can be launched with the cowsay quote on the docker run command line instead of Bash:

```
$ docker run --rm ubuntu_cowsay "Hello World"
 -------------
< Hello World >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note that we have not yet tied the `fortune` command to `cowsay`. We would like to pipe (re-direct) the output of `fortune` to `cowsay` as before, but the following modification to the entry point is not supported:

```
ENTRYPOINT ["/usr/games/fortune | /usr/games/cowsay"]
```

However, we can accomplish our goal by providing our own script, `entrypoint.sh`, and this is discussed next.

4.2 Creating a Custom Entry Point Script

A custom entry point script can be created to launch an arbitrarily complex sequence of actions when a container is run. And of course this script, like a Bash script, can launch other scripts or programs and so on. The basic steps are as follows:

- Create script, typically in the same folder as the Dockerfile
- Add a `COPY` command to the Dockerfile to copy the script from the local file system to the image file system
- Reference the script from the `ENTRYPOINT` command in the Dockerfile.

To get started, we will create and edit our script in our `ubuntu_cowsay` folder:

```
$ cd ubuntu_cowsay
$ emacs entrypoint.sh
```

Edit the `entrypoint.sh` script to contain the following content:

```
01  #!/bin/bash
02  if [ $# = 0 ]; then
03      fortune | cowsay
04  else
05      /usr/games/cowsay " $@ "
06  fi
07
08  # From Mouat, Adrian. Using Docker: Developing and Deploying Software with
09  # Containers (Kindle Locations 527-529).  O'Reilly Media. Kindle Edition.
```

The conditional on line 2 checks to see if the number of command line arguments is zero. If so, a fortune will be generated and sent to `cowsay`. Otherwise, if there are indeed command line arguments, they will be sent directly to `cowsay` on line 5.

As with any script, it needs to have its executable permissions set. These permissions will be carried over when the script is copied to the image:

```
$ chmod +x entrypoint.sh
```

Now the Dockerfile needs to be edited to include the last two lines as shown:

```
# Filename: Dockerfile
FROM ubuntu:20.04
RUN apt-get update
RUN apt-get -y install cowsay fortune
ENV PATH "$PATH:/usr/games"
COPY entrypoint.sh /                              <-- new line
ENTRYPOINT ["/entrypoint.sh"]                     <-- new line
```

Once again, we will build the image using the new Dockerfile:

```
$ docker build -t ubuntu_cowsay .
$ docker images
REPOSITORY       TAG      IMAGE ID        CREATED              SIZE
ubuntu_cowsay    latest   c96d44f493e6    About a minute ago   143MB
```

Now we can run the cowsay container and either pass it our own quote as an argument (first example below), or pass no arguments at all and let the `fortune` command generate a quote automatically and pass it to `cowsay`.

```
$ docker run --rm ubuntu_cowsay "Hello World"
 -------------
< Hello World >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

$ docker run --rm ubuntu_cowsay
 ---------------------------------------
/ You'll wish that you had done some of \
| the hard things when they were easier |
\ to do.                                /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

With this simple example, we have shown that we can use a container, constructed with just the packages and script we need, to act as service that can be invoked essentially instantly, and removed just as quickly. In our case, the `fortune` and `cowsay` commands could have just as easily been installed on the local file system. But that misses at least three points:

- In certain situations we would like to have the functionality of a container like cowsay, but may not have permission to install new software on the local computer.
- Some functionality may require packages or software that is incompatible with the local file system.
- As improvements or updates to the functionality provided by the container are made available, only an updated image is required. No requirements are needed on the local computer, which sometimes triggers a domino effect of upgrades to package.

This is an important milestone. The last part of this this lab will discuss how to make images we create from our Dockerfiles available across multiple machines.

5 Posting Images on DockerHub

In our examples so far, we have built our images with Dockerfiles, and each image started with a basic Ubuntu 20.04 image. This image was obtained from the Docker Hub (https://hub.docker.com). Although Docker Hub contains curated images like Ubuntu, downloaded by thousands of users, it is

also serves as a place to store images create by a user after creating a free account. In this section, we will take our `ubuntu_cowsay` image and make it available on the Docker Hub, after creating a user account.

5.1 Create a Docker Hub Account

Creating an account on Docker Hub is quick and free. Go to https://hub.docker.com and follow the instructions for signing up.

5.2 A Closer Look at Image Names

Thus far, when building our cowsay image, a simple image name was provided: `"ubuntu_cowsay"`:

```
$ docker build -t ubuntu_cowsay .
$ docker images
REPOSITORY       TAG      IMAGE ID        CREATED          SIZE
ubuntu_cowsay    latest   2525fd4e70be    39 seconds ago   143MB
```

Since we didn't specify a tag, the default `"latest"` was used. In general, an image name has three components: The *registry*, the *repository*, and the *tag*. The format is:

`registry/repository:tag`

When working locally it is fine to leave the registry empty and work with an image name `ubuntu_cowsay:latest`. However, when we want to push an image to the Docker Hub, we need to give our image a registry name that is consistent with our account name on Docker Hub. At this point we can either delete and recreate our cowsay image, or create an alias for it with the `docker tag` command:

```
$ docker images
REPOSITORY               TAG      IMAGE ID        CREATED          SIZE
ubuntu_cowsay            latest   2525fd4e70be    39 seconds ago   143MB
```

```
$ docker tag ubuntu_cowsay mikerbenj/ubuntu_cowsay
$ docker images
REPOSITORY               TAG      IMAGE ID        CREATED          SIZE
ubuntu_cowsay            latest   2525fd4e70be    39 seconds ago   143MB
mikerbenj/ubuntu_cowsay  latest   2525fd4e70be    39 seconds ago   143MB
```

Notice the new `"mikerbenj/ubuntu_cowsay"` image is just an *alias* of the first image. They have the same `IMAGE ID`, and have identical create time and size information. We will use the second image name when uploading to Docker Hub since it has the registry name component that matches the username of the account.

5.3 Push Your Own Image to Docker Hub

Assuming we have created an account with Docker Hub, and logged in, the new image can be pushed:

```
$ docker push mikerbenj/ubuntu_cowsay
The push refers to repository [docker.io/mikerbenj/ubuntu_cowsay]
1cc234b344af: Pushed
7eb048c28e96: Pushed
6ea7dc072711: Pushed
8891751e0a17: Pushed
2a19bd70fcd4: Pushed
9e53fd489559: Pushed
7789f1a3d4e9: Pushed
latest: digest: sha256:0d03ce885f63278f29052454a72d32e993fb5fb727309df758da98f23c1d717a size: 1783
```

5.4 Getting and Using Our Image from Docker Hub

Now that our image is on Docker Hub, the fortune/cowsay script can be run on any machine in the world with Docker installed. The following command does not require the `mikerbenj/ubuntu_cowsay` image to be installed on your local computer. The `docker run` command, if it doesn't find the image locally, will automatically retrieve if it from Docker Hub, and run the container:

```
$ docker run --rm mikerbenj/ubuntu_cowsay
Unable to find image 'mikerbenj/ubuntu_cowsay:latest' locally
latest: Pulling from mikerbenj/ubuntu_cowsay
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
d08511a5de10: Pull complete
941896404914: Pull complete
6b47ed4caf50: Pull complete
Digest: sha256:f7b8589cddfb3eb6946fa8c3ea91aefbf571e816f7e1764ec9a4b3a327aa78ed
Status: Downloaded newer image for mikerbenj/ubuntu_cowsay:latest
 ____________________________________
< You may be recognized soon. Hide. >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Granted, to run this program the first time, without the image already local on your machine, there is a 150Mb download, but this is a pretty impressive demonstration on portability.

URL References

[1] Mouat, Adrian. Using Docker: Developing and Deploying Software with Containers (Kindle Locations 488-490). O'Reilly Media. Kindle Edition.