Constructing a docker image

Running containers others made is useful, but if you want to use docker for production, chances are you want to construct a container on your own.

Dockerfile commands summary

Here's a quick summary of some basic commands we will use in our Dockerfile.

> As a rule of thumb, all commands in CAPITAL LETTERS are intended for the docker engine.

- FROM
- RUN
- ADD and COPY
- CMD
- EXPOSE
- ENTRYPOINT
- `FROM` is always the first item in the Dockerfile. It is a requirement that the Dockerfile starts with the `FROM` command. Images are created in layers, which means you can use another image as the base image for your own. The `FROM` command defines your base layer. As argument, it takes the name of the image. Optionally, you can add the Docker Hub username of the maintainer and image version, in the format `username/imagename:version`.
- `RUN` is used to build up the image you're creating. For each `RUN` command, Docker will run the command then create a new layer of the image. This way you can roll back your image to previous states easily. The syntax for a `RUN` instruction is to place the full text of the shell command after the `RUN` (e.g., `RUN mkdir /user/local/foo`). This will automatically run in a `/bin/sh` shell. You can define a different shell like this: `RUN /bin/bash -c 'mkdir /user/local/foo'`
- `COPY` copies local files into the container. The files need to be in the same folder (or a sub folder) as the Dockerfile itself. An example is copying the requirements for a python app into the container: `COPY requirements.txt /usr/src/app/`.
- `CMD` defines the commands that will run on the image at start-up. Unlike a `RUN`, this does not create a new layer for the image, but simply runs the command. There can only be one `CMD` in a Dockerfile. If you need to run multiple commands, the best way to do that is to have the `CMD` run a script. `CMD` requires that you tell it where to run the command, unlike `RUN`. So example `CMD` commands would be:

```
CMD ["python", "./app.py"]

CMD ["/bin/bash", "echo", "Hello World"]
```

- `EXPOSE` creates a hint for users of an image that provides services on ports. It is included in the information which can be retrieved via `$ docker container inspect <container-id>`.

> **Note:** The `EXPOSE` command does not actually make any ports accessible to the host! Instead, this requires publishing ports by means of the `-p` or `-P` flag when using `$ docker container run`.

- `ENTRYPOINT` configures a command that will run no matter what the user specifies at runtime.

Write a Dockerfile

We want to create a Docker image with a Python web app.

As mentioned above, all user images are based on a *base image*. Since our application is written in Python, we will build our own Python image based on Ubuntu. We'll do that using a **Dockerfile**.

> **Note:** If you want to learn more about Dockerfiles, check out Best practices for writing Dockerfiles.

A Dockerfile is a text file containing a list of commands that the Docker daemon calls while creating an image. The Dockerfile contains all the information that Docker needs to know to run the app; a base Docker image to run from, location of your project code, any dependencies it has, and what commands to run at start-up.

It is a simple way to automate the image creation process. The best part is that the commands you write in a Dockerfile are *almost* identical to their equivalent Linux commands. This means you don't really have to learn new syntax to create your own Dockerfiles.

1. Create a file called **Dockerfile**, and add content to it as described below. We have made a small boilerplate file and app for you in the /building-an-image folder, so head over there.

> If you want to make a file from scratch, you can use the linux command `touch <filename>` to create an empty file, and the text editor `nano <filename>` to manipulate the file.

We'll start by specifying our base image, using the `FROM` keyword:

```
docker FROM ubuntu:latest
```

1. The next step is usually to write the commands of copying the files and installing the dependencies. But first we will install the Python pip package to the ubuntu linux distribution. This will not just install the pip package but any other dependencies too, which includes the python interpreter. Add the following RUN command next:`RUN apt-get update -y`
   `RUN apt-get install -y python-pip python-dev build-essential`
2. Let's add the files that make up the Flask Application. Install all Python requirements for our app to run. This will be accomplished by adding the lines: `COPY requirements.txt /usr/src/app/`
   `RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt`

Copy the application app.py into our image by using the COPY command. `COPY app.py /usr/src/app/`

3. Specify the port number which needs to be exposed. Since our flask app is running on `5000` that's what we'll expose. `EXPOSE 5000`
   The `EXPOSE` instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. You need the `-p`/`-P` command to actually open the host ports.
4. The last step is the command for running the application which is simply - `python ./app.py`. Use the CMD command to do that: `CMD ["python", "/usr/src/app/app.py"]`
   The primary purpose of `CMD` is to tell the container which command it should run by default when it is started.
5. Verify your Dockerfile. Our `Dockerfile` is now ready. This is how it looks:

```
  # The base image
  FROM ubuntu:latest

  # Install python and pip
  RUN apt-get update -y
  RUN apt-get install -y python-pip python-dev build-essential

  # Install Python modules needed by the Python app
  COPY requirements.txt /usr/src/app/
  RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

  # Copy files required for the app to run
  COPY app.py /usr/src/app/

  # Declare the port number the container should expose
  EXPOSE 5000

  # Run the application
  CMD ["python", "/usr/src/app/app.py"]
  ```

Build the image

Now that you have your `Dockerfile`, you can build your image. The `docker build` command does the heavy-lifting of creating a docker image from a `Dockerfile`.

The `docker build` command is quite simple - it takes an optional tag name with the `-t` flag, and the location of the directory containing the `Dockerfile` - the `.` indicates the current directory:

```bash
$ docker build -t myfirstapp .
Sending build context to Docker daemon   5.12kB
Step 1/8 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
b6f892c0043b: Pull complete
55010f332b04: Pull complete
2955fb827c94: Pull complete
```

```
3deef3fcbd30: Pull complete
cf9722e506aa: Pull complete
Digest: sha256:382452f82a8bbd34443b2c727650af46aced0f94a44463c62a9848133ecb1aa8
Status: Downloaded newer image for ubuntu:latest
 ---> ebcd9d4fca80
Step 2/8 : RUN apt-get update -y
 ---> Running in 42d5752a0faf
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]

...

Get:20 http://archive.ubuntu.com/ubuntu xenial-backports/main amd64 Packages [4927
B]
Get:21 http://archive.ubuntu.com/ubuntu xenial-backports/universe amd64 Packages
[6022 B]
Fetched 24.0 MB in 6s (3911 kB/s)
Reading package lists...
 ---> 07205bd484c9
Removing intermediate container 42d5752a0faf
Step 3/8 : RUN apt-get install -y python-pip python-dev build-essential
 ---> Running in 43e6e25b8c6b
Reading package lists...
Building dependency tree...
Reading state information...
The following additional packages will be installed:
  binutils bzip2 ca-certificates cpp cpp-5 dpkg-dev fakeroot file g++ g++-5
...
  python2.7-minimal rename xz-utils
0 upgraded, 87 newly installed, 0 to remove and 3 not upgraded.
Need to get 85.4 MB of archives.
After this operation, 268 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu xenial/main amd64 libatm1 amd64 1:2.5.1-1.5
[24.2 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial/main amd64 libmnl0 amd64 1.0.3-5
[12.0 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial/main amd64 libgdbm3 amd64 1.8.3-13.1
[16.9 kB]
...
Get:86 http://archive.ubuntu.com/ubuntu xenial/universe amd64 python-wheel all
0.29.0-1 [48.0 kB]
Get:87 http://archive.ubuntu.com/ubuntu xenial/main amd64 rename all 0.20-4 [12.0
kB]
debconf: delaying package configuration, since apt-utils is not installed
Fetched 85.4 MB in 24s (3514 kB/s)
Selecting previously unselected package libatm1:amd64.
(Reading database ... 4764 files and directories currently installed.)
Preparing to unpack .../libatm1_1%3a2.5.1-1.5_amd64.deb ...
Unpacking libatm1:amd64 (1:2.5.1-1.5) ...
...
Selecting previously unselected package python-wheel.
Preparing to unpack .../python-wheel_0.29.0-1_all.deb ...
Unpacking python-wheel (0.29.0-1) ...
Selecting previously unselected package rename.
Preparing to unpack .../archives/rename_0.20-4_all.deb ...
Unpacking rename (0.20-4) ...
...
```

```
Setting up python-setuptools (20.7.0-1) ...
Setting up python-wheel (0.29.0-1) ...
Setting up rename (0.20-4) ...
update-alternatives: using /usr/bin/file-rename to provide /usr/bin/rename (rename)
in auto mode
Processing triggers for libc-bin (2.23-0ubuntu7) ...
Processing triggers for systemd (229-4ubuntu17) ...
Processing triggers for ca-certificates (20160104ubuntu1) ...
Updating certificates in /etc/ssl/certs...
173 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
 ---> 61881e70baa5
Removing intermediate container 43e6e25b8c6b
Step 4/8 : COPY requirements.txt /usr/src/app/
 ---> b323c089d44a
Removing intermediate container 96aca854f3c4
Step 5/8 : RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
 ---> Running in f92f9c719287
Collecting Flask==0.10.1 (from -r /usr/src/app/requirements.txt (line 1))
  Downloading Flask-0.10.1.tar.gz (544kB)
Collecting Werkzeug>=0.7 (from Flask==0.10.1->-r /usr/src/app/requirements.txt
(line 1))
  Downloading Werkzeug-0.12.2-py2.py3-none-any.whl (312kB)
Collecting Jinja2>=2.4 (from Flask==0.10.1->-r /usr/src/app/requirements.txt (line
1))
  Downloading Jinja2-2.9.6-py2.py3-none-any.whl (340kB)
Collecting itsdangerous>=0.21 (from Flask==0.10.1->-r /usr/src/app/requirements.txt
(line 1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->Flask==0.10.1->-r
/usr/src/app/requirements.txt (line 1))
  Downloading MarkupSafe-1.0.tar.gz
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, itsdangerous, Flask
  Running setup.py install for MarkupSafe: started
    Running setup.py install for MarkupSafe: finished with status 'done'
  Running setup.py install for itsdangerous: started
    Running setup.py install for itsdangerous: finished with status 'done'
  Running setup.py install for Flask: started
    Running setup.py install for Flask: finished with status 'done'
Successfully installed Flask-0.10.1 Jinja2-2.9.6 MarkupSafe-1.0 Werkzeug-0.12.2
itsdangerous-0.24
You are using pip version 8.1.1, however version 9.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
 ---> c1f2dc732c7c
Removing intermediate container f92f9c719287
Step 6/8 : COPY app.py /usr/src/app/
 ---> 6ed47d3c544a
Removing intermediate container 61a68a949d68
Step 7/8 : EXPOSE 5000
 ---> Running in 1f939928b7d5
 ---> 6c14a93b72f2
Removing intermediate container 1f939928b7d5
Step 8/8 : CMD python /usr/src/app/app.py
 ---> Running in 8e5d5619c75e
 ---> 61fedfc3fcad
Removing intermediate container 8e5d5619c75e
```

```
Successfully built 61fedfc3fcad
Successfully tagged myfirstapp:latest
```

If you don't have the `ubuntu:latest` image, the client will first pull the image and then create your image. If you do have it, your output on running the command will look different from mine.

If everything went well, your image should be ready! Run `docker image ls` and see if your image (`myfirstapp`) shows.

Run your image

The next step in this section is to run the image and see if it actually works.

```
$ docker container run -p 8888:5000 --name myfirstapp myfirstapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Head over to `http://localhost:8888` or your server's URL and your app should be live.

and layers

When dealing with docker images, a layer, or image layer is a change on an image, or an intermediate image. Every command you specify (FROM, RUN, COPY, etc.) in your Dockerfile causes the previous image to change, thus creating a new layer. You can think of it as staging changes when you're using Git: You add a file's change, then another one, then another one...

Consider the following Dockerfile:

```
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev build-essential
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
COPY app.py /usr/src/app/
EXPOSE 5000
CMD ["python", "/usr/src/app/app.py"]
```

First, we choose a starting image: `ubuntu:latest`, which in turn has many layers. We add another layer on top of our starting image, running an update on the system. After that yet another for installing the python ecosystem. Then, we tell docker to copy the requirements to the container. That's another layer.

The concept of layers comes in handy at the time of building images. Because layers are intermediate images, if you make a change to your Dockerfile, docker will build only the layer that was changed and the ones after that. This is called layer caching.

Each layer is build on top of it's parent layer, meaning if the parent layer changes, the next layer does as well.

If you want to concatenate two layers (e.g. the update and install which is a good idea), then do them in the same RUN command:

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y \
 python-pip \
 python-dev \
 build-essential
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
COPY app.py /usr/src/app/
EXPOSE 5000
CMD ["python", "/usr/src/app/app.py"]
```

If you want to be able to use any cached layers from last time, they need to be run *before the update command*.

> NOTE: Once we build the layers, Docker will reuse them for new builds. This makes the builds much faster. This is great for continuous integration, where we want to build an image at the end of each successful build (e.g. in Jenkins). But the build is not only faster, the new image layers are also smaller, since intermediate images are shared between images.

Try to move the two `COPY` commands before for the `RUN` and build again to see it taking the cached layers instead of making new ones.

Every layer can be a container

As stated above, all FROM, RUN, ADD, COPY, CMD and EXPOSE will create a new layer in your image, and therefore also be an image of their own.

Take a look again at some of the output from building the image above:

```
 ---> c1f2dc732c7c
Removing intermediate container f92f9c719287
Step 6/8 : COPY app.py /usr/src/app/
 ---> 6ed47d3c544a
Removing intermediate container 61a68a949d68
Step 7/8 : EXPOSE 5000
 ---> Running in 1f939928b7d5
 ---> 6c14a93b72f2
 ```

So what docker actually does is

```
- Taking the layer created just before
- make a container based of it
- run the command given
- save the layer.

in a loop untill all the commands have been made.
Try to create a container from your `COPY app.py /usr/src/app/` command.
The id of the layer will likely be different than the example above.

`docker container run -ti -p 5000:5000 6ed47d3c544a bash`.

You are now in a container run from _that_ layer in the build script. You can't
make the `EXPOSE` command, but you can look around, and run the last python app:

```bash

root@cc5490748b2a:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv
sys  tmp  usr  var

root@cc5490748b2a:/# ls /usr/src/app/
app.py  requirements.txt

root@cc5490748b2a:/# python /usr/src/app/app.py
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

And just like the image you builded above, you can browse the website now.

Delete your image

If you make a `docker container ls -a` command, you can now see a container with the name *myfirstapp* from the image named *myfirstapp*.

```
sofus@Praq-Sof:/4$ docker container ls -a
CONTAINER ID        IMAGE                    COMMAND                  CREATED
STATUS                      PORTS
NAMES
fcfba2dfb8ee        myfirstapp               "python /usr/src/a..."   About a
minute ago   Exited (0) 28 seconds ago
myfirstapp
```

Make a `docker image ls` command to see that you have a docker image with the name `myfirstapp`

Try now to first:

- remove the container
- remove the image file as well with the `image rm` command.
- make `docker image ls` again to see that it's gone.

Instructions
```

Here is the list of all the instructions that can be used in a Dockerfile:

- .dockerignore
- FROM Sets the Base Image for subsequent instructions.
- RUN execute any commands in a new layer on top of the current image and commit the results.
- CMD provide defaults for an executing container.
- EXPOSE informs Docker that the container listens on the specified network ports at runtime. NOTE: does not actually make ports accessible.
- ENV sets environment variable.
- ADD copies new files, directories or remote file to container. Invalidates caches. Avoid `ADD` and use `COPY` instead.
- COPY copies new files or directories to container. Note that this only copies as root, so you have to chown manually regardless of your USER / WORKDIR setting. See https://github.com/moby/moby/issues/30110
- ENTRYPOINT configures a container that will run as an executable.
- VOLUME creates a mount point for externally mounted volumes or other containers.
- USER sets the user name for following RUN / CMD / ENTRYPOINT commands.
- WORKDIR sets the working directory.
- ARG defines a build-time variable.
- ONBUILD adds a trigger instruction when the image is used as the base for another build.
- STOPSIGNAL sets the system call signal that will be sent to the container to exit.
- LABEL apply key/value metadata to your images, containers, or daemons.

Summary

You learned how to write your own docker images in a `Dockerfile` with the use of the `FROM` command to choose base-images like Alpine or Ubuntu and keywords like `RUN` for executing commands,`COPY` to add resources to the container, and `CMD` to indicate what to run when starting the container. You also learned that each of the keywords generates an image layer on top of the previous, and that everyone of the layers can be converted to a running container.