docker-workshop

This is the documentation for an **Introduction to Docker** for Redapt's "Docker Workshop".

by Christoph Champ for Redapt, Inc. (March 2018)

Docker directives

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build`, users can create an automated build that executes several command-line instructions in succession.

This section will describe some of the common Dockerfile commands (aka "instructions" / "directives").

FROM

The `FROM` instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a `FROM` instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories. All of the examples shown in this workshop will pull base images from the public repositories (aka Docker Hub).

Every example of a Dockerfile shown in this workshop will begin with the `FROM` directive.

RUN

The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

Notes on the order of execution:

```
FROM centos:latest
LABEL maintainer="bob@example.com"

RUN useradd -ms /bin/bash bob
USER bob

RUN echo "export PATH=/path/to/my/app:$PATH" >> /etc/bashrc


$ docker build -t centos7/config:v1 .
 ...
 /bin/sh: /etc/bashrc: Permission denied
```

The order of execution matters! Prior to the directive `USER bob`, the user was root. After that directive, the user is now bob, who does not have super-user privileges. Move the `RUN echo ...` directive to before the `USER bob` directive for a successful build.

USER

The `USER` instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any `RUN`, `CMD`, and `ENTRYPOINT` instructions that follow it in the Dockerfile.

```
$ cat << EOF > Dockerfile
# Non-privileged user entry
FROM centos:latest
MAINTAINER bob@example.com

RUN useradd -ms /bin/bash bob
USER bob
EOF
```

Note: The use of `MAINTAINER` has been deprecated in newer versions of Docker. You should use `LABEL` instead, as it is much more flexible and its key/values show up in `docker inspect`. From here forward, I will only use `LABEL`.

```
$ docker build -t centos7/nonroot:v1 .
$ docker exec -it <container_name> /bin/bash
```

We are user "bob" and are unable to become root. The workaround (i.e., how to become root) is like so:

```
$ docker exec -u 0 -it <container_name> /bin/bash
```

NOTE: For the remainder of this section, I will omit the `$ cat << EOF > Dockerfile` part in the examples for brevity.

ENV

The `ENV` instruction sets the environment variable `key` to the value `value`. This value will be in the environment of all "descendant" Dockerfile commands and can be replaced inline in many as well.

*Note: The following is a **terrible** way of building a container. I am purposely doing it this way so I can show you a much better way later (see below).*

- Build a CentOS 7 Docker image with Oracle Java 8 installed:

```
# SEE: https://gist.github.com/P7h/9741922 for various Java versions
FROM centos:latest
LABEL maintainer="bob@example.com"

RUN yum update -y
RUN yum install -y net-tools wget

RUN echo "SETTING UP JAVA"
# The tarball method:
# RUN cd ~ && wget --no-cookies --no-check-certificate \
#     --header "Cookie: gpw_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-
cookie" \
#     "http://download.oracle.com/otn-pub/java/jdk/8u91-b14/jdk-8u91-linux-x64.tar.gz"
# RUN tar xzvf jdk-8u91-linux-x64.tar.gz
# RUN mv jdk1.8.0_91 /opt
# ENV JAVA_HOME /opt/jdk1.8.0_91/

# The rpm method:
RUN cd ~ && wget --no-cookies --no-check-certificate \
    --header "Cookie: gpw_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie"
\
    "http://download.oracle.com/otn-pub/java/jdk/8u161-b12/2f38c3b165be4555a1fa6e98c45e0808/jdk-8u161-
linux-x64.rpm"
RUN yum localinstall -y /root/jdk-8u161-linux-x64.rpm

RUN useradd -ms /bin/bash bob
USER bob
```

```
# User specific environment variable
RUN cd ~ && echo "export JAVA_HOME=/usr/java/jdk1.8.0_161/jre" >> ~/.bashrc
# Global (system-wide) environment variable
ENV JAVA_BIN /usr/java/jdk1.8.0_161/jre/bin



$ docker build -t centos7/java8:v1 .
```

CMD vs. RUN

The main purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

There can only be one `CMD` instruction in a Dockerfile. If you list more than one `CMD` then only the last `CMD` will take effect.

```
FROM centos:latest
LABEL maintainer="bob@example.com"

RUN useradd -ms /bin/bash bob
CMD ["echo", "Hello from within my container"]
```

The `CMD` directive *only* executes when the container is started, whereas the `RUN` directive is executed during the build of the image.

```
$ docker build -t centos7/echo:v1 .
$ docker run centos7/echo:v1
Hello from within my container
```

The container starts, echos out that message, then exits.

ENTRYPOINT

An `ENTRYPOINT` allows you to configure a container that will run as an executable. Examples include an Apache or Nginx webserver, a Python Jupyter notebook, an API service, etc.

```
FROM centos:latest
LABEL maintainer="bob@example.com"

RUN useradd -ms /bin/bash bob
ENTRYPOINT "This command will display this message on EVERY container that is run from it"



$ docker build -t centos7/entry:v1 .

$ docker run centos7/entry:v1
This command will display this message on EVERY container that is run from it

$ docker run centos7/entry:v1 /bin/echo "Can you see me?"
This command will display this message on EVERY container that is run from it

$ docker run centos7/echo:v1 /bin/echo "Can you see me?"
Can you see me?
```

Note the difference between the output of the "echo" and the "entry" containers.

EXPOSE

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

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. To actually publish the port when running the container, use the `-p` flag on docker run to publish and map one or more ports, or the `-P` flag to publish all exposed ports and map them to high-order ports (i.e., from 32769 - 65535).

```
FROM centos:latest
LABEL maintainer="bob@example.com"

RUN yum update -y
RUN yum install -y httpd net-tools

RUN echo "This is a custom index file built during the image creation" > /var/www/html/index.html

ENTRYPOINT apachectl -DFOREGROUND  # BAD WAY TO DO THIS!



$ docker build -t centos7/apache:v1 .

# Run the above built image as a container in dettached mode:
$ docker run -d --name webserver centos7/apache:v1

$ docker exec webserver /bin/cat /var/www/html/index.html
This is a custom index file built during the image creation

# Get the internal container IP address:
$ docker inspect webserver -f '{{.NetworkSettings.IPAddress}}'  # => 172.17.0.6
#~OR~
$ docker inspect webserver | jq -crM '.[] | .NetworkSettings.IPAddress'  # => 172.17.0.6

$ curl 172.17.0.6
This is a custom index file built during the image creation
$ curl -sI 172.17.0.6 | awk '/^HTTP|^Server/{print}'
HTTP/1.1 200 OK
Server: Apache/2.4.6 (CentOS)

# Stop the container:
$ time docker stop webserver
real   0m10.275s  # <- notice how long it took to stop the container
user   0m0.008s
sys    0m0.000s

# Remove the container
$ docker rm webserver
```

It took ~10 seconds to stop the above container. This is because of the way we are (incorrectly) using `ENTRYPOINT`. The `SIGTERM` signal when running `docker stop webserver` actually timed out instead of exiting gracefully. A much better method is shown below, which *will* exit gracefully and in less than 300 ms.

- Expose ports from the CLI

```
$ docker run -d --name webserver -p 8080:80 centos7/apache:v1
$ curl localhost:8080
This is a custom index file built during the image creation
$ docker stop webserver && docker rm webserver
```

- Explicitly expose a port in the Docker image:

```
FROM centos:latest
LABEL maintainer="bob@example.com"
```

```
RUN yum update -y && \
    yum install -y httpd net-tools && \
    yum autoremove -y && \
    echo "This is a custom index file built during the image creation" > /var/www/html/index.html

EXPOSE 80

ENTRYPOINT ["/usr/sbin/httpd", "-D", "FOREGROUND"]



$ docker build -t centos7/apache:v2 .
$ docker run -d --rm --name webserver -P centos7/apache:v1

# Get the higher-order external port automatically assigned to the container:
$ docker container ls --format '{{.Names}} {{.Ports}}'
webserver 0.0.0.0:32769->80/tcp
#~OR~
$ docker port webserver | cut -d: -f2
32769
#~OR~
$ docker inspect webserver | jq -crM '[.[] | .NetworkSettings.Ports."80/tcp"[] | .HostPort] | .[]'
32769

$ curl localhost:32769
This is a custom index file built during the image creation

# Stop the container
$ time docker stop webserver
real   0m0.283s  # <- Note how much faster the container stopped
user   0m0.004s
sys    0m0.008s
```

Note that we passed `--rm` to the `docker run` command so that the container will be removed when we stop the container. Also note how much faster the container stopped (~300ms vs. 10 seconds above).

Container volume managment

```
$ docker run -it --name voltest -v /mydata centos:latest /bin/bash

[root@bffdcb88c485 /]# df -h
Filesystem                  Size  Used Avail Use% Mounted on
none                        213G  173G   30G  86% /
tmpfs                       7.8G     0  7.8G   0% /dev
tmpfs                       7.8G     0  7.8G   0% /sys/fs/cgroup
/dev/mapper/ubuntu--vg-root  213G  173G   30G  86% /mydata
shm                          64M     0   64M   0% /dev/shm
tmpfs                       7.8G     0  7.8G   0% /sys/firmware

[root@bffdcb88c485 /]# echo "testing" >/mydata/mytext.txt

$ docker inspect voltest | jq -crM '.[] | .Mounts[].Source'
/var/lib/docker/volumes/2a53fd295595690200a63def8a333b54682174923339130d560fb77ecbe41a3b/_data

$ sudo cat
/var/lib/docker/volumes/2a53fd295595690200a63def8a333b54682174923339130d560fb77ecbe41a3b/_data/mytext.txt

testing

$ sudo /bin/bash -c \
  "echo 'this is from the host OS'
>/var/lib/docker/volumes/2a53fd295595690200a63def8a333b54682174923339130d560fb77ecbe41a3b/_data/host.txt"
```

```
[root@bffdcb88c485 /]# cat /mydata/host.txt
this is from the host OS
```

- Cleanup

```
$ docker rm voltest
$ docker volume ls
$ docker volume rm 2a53fd295595690200a63def8a333b54682174923339130d560fb77ecbe41a3b
```

- Mount host's current working directory inside container:

```
$ echo "my config" >my.conf
$ echo "my message" >message.txt
$ echo "aerwr3adf" >app.bin
$ chmod +x app.bin
$ docker run -it --name voltest -v ${PWD}:/mydata centos:latest /bin/bash

[root@f5f34ccb54fb /]# ls -l /mydata/
total 24
-rwxrwxr-x 1 1000 1000 10 Mar  8 19:29 app.bin
-rw-rw-r-- 1 1000 1000 11 Mar  8 19:29 message.txt
-rw-rw-r-- 1 1000 1000 10 Mar  8 19:29 my.conf
[root@f5f34ccb54fb /]# touch /mydata/foobar

$ ls -l ${PWD}
total 24
-rwxrwxr-x 1 bob   bob   10 Mar  8 11:29 app.bin
-rw-r--r-- 1 root  root   0 Mar  8 11:36 foobar
-rw-rw-r-- 1 bob   bob   11 Mar  8 11:29 message.txt
-rw-rw-r-- 1 bob   bob   10 Mar  8 11:29 my.conf

$ docker rm voltest
```

Images

Saving and loading images

```
$ docker pull centos:latest
$ docker run -it centos:latest /bin/bash

[root@29fad368048c /]# yum update -y
[root@29fad368048c /]# echo xtof >/root/built_by.txt
[root@29fad368048c /]# exit

$ docker commit reverent_elion centos:xtof
$ docker rm reverent_elion
$ docker images
REPOSITORY   TAG      IMAGE ID       CREATED         SIZE
centos       xtof     e0c8bd35ba50   3 seconds ago   463MB
centos       latest   980e0e4c79ec   1 minute ago    197MB

$ docker history centos:xtof
IMAGE          CREATED             CREATED BY                                      SIZE
e0c8bd35ba50   27 seconds ago      /bin/bash                                       266MB
980e0e4c79ec   18 months ago       /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      18 months ago       /bin/sh -c #(nop)  LABEL name=CentOS Base ...   0B
<missing>      18 months ago       /bin/sh -c #(nop) ADD file:e336b45186086f7...   197MB
<missing>      18 months ago       /bin/sh -c #(nop)  MAINTAINER <nowiki>https://gith...</nowiki>   0B
```

- Save the original `centos:latest` image we pulled from Docker Hub:

```
$ docker save --output centos-latest.tar centos:latest
```

Note that the above command essentially tars up the contents of the image found in `/var/lib/docker/image` directory.

```
$ tar tvf centos-latest.tar
-rw-r--r-- 0/0        2309 2016-09-06 14:10
980e0e4c79ec933406e467a296ce3b86685e6b42eed2f873745e6a91d718e37a.json
drwxr-xr-x 0/0           0 2016-09-06 14:10
ad96ed303040e4a7d1ee0596bb83db3175388259097dee50ac4aaae34e90c253/
-rw-r--r-- 0/0           3 2016-09-06 14:10
ad96ed303040e4a7d1ee0596bb83db3175388259097dee50ac4aaae34e90c253/VERSION
-rw-r--r-- 0/0        1391 2016-09-06 14:10
ad96ed303040e4a7d1ee0596bb83db3175388259097dee50ac4aaae34e90c253/json
-rw-r--r-- 0/0   204305920 2016-09-06 14:10
ad96ed303040e4a7d1ee0596bb83db3175388259097dee50ac4aaae34e90c253/layer.tar
-rw-r--r-- 0/0         202 1969-12-31 16:00 manifest.json
-rw-r--r-- 0/0          89 1969-12-31 16:00 repositories
```

- Save space by compressing the tar file:

```
$ gzip centos-latest.tar  # .tar -> 195M; .tar.gz -> 68M
```

- Delete the original `centos:latest` image:

```
$ docker rmi centos:latest
```

- Restore (or load) the image back to our local repository:

```
$ docker load --input centos-latest.tar.gz
```

Tagging images

- List our current images:

```
$ docker images
REPOSITORY    TAG    IMAGE ID            CREATED             SIZE
centos        xtof   e0c8bd35ba50        About an hour ago   463MB
```

- Tag the above image:

```
$ docker tag e0c8bd35ba50 xtof/centos:v1
$ docker images
REPOSITORY    TAG    IMAGE ID            CREATED             SIZE
centos        xtof   e0c8bd35ba50        About an hour ago   463MB
xtof/centos   v1     e0c8bd35ba50        About an hour ago   463MB
```

Note that we did not create a new image, we just created a new tag of the same/original `centos:xtof` image.

Note: The maximum number of characters in a tag is 128.

Docker networking

Default networks

```
$ ip addr show docker0
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:75:70:13 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:c0ff:fe75:7013/64 scope link
       valid_lft forever preferred_lft forever
#~OR~
$ ifconfig docker0
docker0   Link encap:Ethernet  HWaddr 02:42:c0:75:70:13
          inet addr:172.17.0.1 Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:c0ff:fe75:7013/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:420654 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1162975 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:85851647 (85.8 MB)  TX bytes:1196235716 (1.1 GB)

$ docker network inspect bridge | jq '.[] | .IPAM.Config[].Subnet'
"172.17.0.0/16"
```

So, the usable range of IP addresses in our 172.17.0.0/16 subnet is: 172.17.0.1 - 172.17.255.254

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
bf831059febc        bridge              bridge              local
266f6df5c44e        host                host                local
ce79e4043a20        none                null                local
$ docker ps -q | wc -l
#~OR~
$ docker container ls --format '{{.Names}}' | wc -l
4  # => 4 running containers
$ docker network inspect bridge | jq '.[] | .Containers[].IPv4Address'
"172.17.0.2/16"
"172.17.0.5/16"
"172.17.0.4/16"
"172.17.0.3/16"
```

The output from the last command are the IP addresses of the 4 containers currently running on my host.

Custom networks

- Create a Docker network

```
$ man docker-network-create  # for details
$ docker network create --subnet 10.1.0.0/16 --gateway 10.1.0.1 --ip-range=10.1.4.0/24 \
    --driver=bridge --label=host4network br04
```

- Use the above network with a given container:

```
$ docker run -it --name net-test --net br04 centos:latest /bin/bash
```

- Assign a static IP to a given container in the above (user created) network:

```
$ docker run -it --name net-test --net br04 --ip 10.1.4.100 centos:latest /bin/bash
```

Note: You can "only" assign static IPs to user created networks (i.e., you "cannot" assign them to the default "bridge" network).

Monitoring and logging

```
$ docker top <container_name>
$ docker stats <container_name>
$ docker logs <container_name>
```

Events

```
$ docker events
$ docker events --since '1h'
$ docker events --since '2018-03-08T16:00'
$ docker events --filter event=attach
$ docker events --filter event=destroy
$ docker events --filter event=attach --filter event=die --filter event=stop
```