Docker Hands-on Guide : Writing a Dockerfile

In this part, we shall take a look at how to build our own Docker images via a Dockerfile. We saw building our own image earlier via running a Container, installing our software and doing a commit to create the image. However, writing a Dockerfile is a more consistent and repeatable way to build your own images.

A Dockerfile is a text file that has a series of instructions on how to build your image. It supports a simple set of commands that you need to use in your Dockerfile. There are several commands supported like FROM, CMD, ENTRYPOINT, VOLUME, ENV and more. We shall look at some of them.

Let us first start with the the overall flow, which goes something like this:

1. You create a Dockerfile with the required instructions.
2. Then you will use the docker build command to create a Docker image based on the Dockerfile that you created in step 1.

With this information, let us get going.

First - launch a command window and create a folder named images as shown below. This will be our root folder where we will create our Dockerfile.

```
$ mkdir images
```

Then we navigate into that directory via cd images.

Now, open up the vi editor and create our first Dockerfile as shown below:

```
FROM busybox:latest
MAINTAINER Romin Irani (email@domain.com)
```

Since, a Docker image is nothing but a series of layers built on top of each other, we start with a base image. The FROM command sets the base image for the rest of the instructions. The MAINTAINER command tells who is the author of the generated images. This is a good practice. You could have taken any other base image in the FROM instruction too, for e.g. ubuntu:latest or ubuntu:14.04, etc.

Now, save the file and come back to the prompt.

Go to the **/images** folder and execute the following in the **/images** folder as shown below:

```
$ docker build -t myimage:latest .
 Sending build context to Docker daemon 2.048 kB
 Sending build context to Docker daemon
 Step 0 : FROM busybox:latest
 ---> 8c2e06607696
 Step 1 : MAINTAINER Romin Irani (email@domain.com)
 ---> Running in 5d70f02a83e1
 ---> 3bc3545a1f64
 Removing intermediate container 5d70f02a83e1
 Successfully built 3bc3545a1f64
 docker@boot2docker:~/images$
```

What we have done here is the step 2 that we highlighted in the overall process above i.e. we have executed the docker build command. This command is used to build a Docker image. The parameters that we have passed are:

- -t is the Docker image tag. You can give a name to your image and a tag.
- The second parameter (a '.') specifies the location of the Dockerfile that we created. Since we created the Dockerfile in the same folder in which we are running the docker build, we specified the current directory.

Notice the various steps that the build process goes through to build out your image.

If you run a docker images command now, you will see the myimage image listed in the output as shown below:

```
$ docker images
 REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
 myimage latest 3bc3545a1f64 3 minutes ago 2.433 MB
```

You can now launch a container, any time via the standard docker run command:

```
$ docker run -it myimage
 / #
```

And we are back into the **myimage** shell.

Now, let us modify the Dockerfile with a new instruction CMD as shown below:

```
FROM busybox:latest
MAINTAINER Romin Irani (email@domain.com)
CMD ["date"]
```

Now, build the image and run a container based on it again. You will find that it printed out the date for you as shown below:

```
$ docker run -it myimage
 Sun Aug 2 00:38:08 UTC 2015
```

The CMD instruction takes various forms and when it is used individually in the file without the ENTRYPOINT command (which we will see in a while), it takes the following format:

```
CMD ["executable","param1","param2"]
```

So in our case, we provided the date command as the executable and when we ran a container based on the myimage now, it printed out the data.

In fact, while launching the container, you can override the default CMD by providing it at the command line as shown below. In this example, we are saying to launch the shell , thereby overriding the default CMD instruction for the Docker Image. Notice that it will lead us into the shell.

```
$ docker run -it myimage /bin/sh
 / #
```

Exercise: Try out modifying the CMD instruction. Give some other commands like CMD [“ls”,”-al”] , build the image and run a container based on that.

The best practice is to use another command ENTRYPOINT together with CMD. The ENTRYPOINT is used to specify the default app that you want to run (This is the way to configure a container that will run as an executable.) The CMD will then provide only the list of parameters to that ENTRYPOINT application. You could still override the CMD parameters at the command line while launching the container.

Let us understand that with the following example. Change your Dockerfile to the following:

```
FROM busybox
MAINTAINER Romin Irani (email@domain.com)
ENTRYPOINT ["/bin/cat"]
CMD ["/etc/passwd"]
```

Now when you build the image and run a container as follows:

```
$ docker run -it myimage
 root:x:0:0:root:/root:/bin/sh
 daemon:x:1:1:daemon:/usr/sbin:/bin/sh
 bin:x:2:2:bin:/bin:/bin/sh
 sys:x:3:3:sys:/dev:/bin/sh
 sync:x:4:100:sync:/bin:/bin/sync
 mail:x:8:8:mail:/var/spool/mail:/bin/sh
 proxy:x:13:13:proxy:/bin:/bin/sh
 www-data:x:33:33:www-data:/var/www:/bin/sh
 backup:x:34:34:backup:/var/backups:/bin/sh
 operator:x:37:37:Operator:/var:/bin/sh
 haldaemon:x:68:68:hald:/:/bin/sh
 dbus:x:81:81:dbus:/var/run/dbus:/bin/sh
 ftp:x:83:83:ftp:/home/ftp:/bin/sh
 nobody:x:99:99:nobody:/home:/bin/sh
 sshd:x:103:99:Operator:/var:/bin/sh
 default:x:1000:1000:Default non-root user:/home/default:/bin/sh
```

So, what happened what it took the default ENTRYPOINT i.e. cat command and used the parameters that the CMD provided to run the command.

Try to override the CMD by running the container with a non-existent file name:

```
$ docker run -it myimage somefile.txt
 cat: can't open 'somefile.txt': No such file or directory
```

You get the point?

Now, let us look at another Dockerfile shown below:

```
FROM ubuntu
MAINTAINER Romin Irani (email@domain.com)
RUN apt-get update
```

```
RUN apt-get install -y nginx
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

Here, what we are building is an image that will run the nginx proxy server for us. Look at the set of instructions and it should be pretty clear. After the standard FROM and MAINTAINER instructions, we are executing a couple of RUN instructions. A RUN instruction is used to execute any commands. In this case we are running a package update and then installing nginx. The ENTRYPOINT is then running the nginx executable and we are using the EXPOSE command here to inform what port the container will be listening on. Remember in our earlier chapters, we saw that if we use the -P command, then the EXPOSE port will be used by default. However, you can always change the host port via the -p parameter as needed.

If you build the image and run the container as follows:

```
docker run -d -p 80:80 --name webserver myimage
```

, you will find that it will have nginx started on port 80. And if you visit the page via the host IP, you will see the following point:

Now, how about adding your own pages into the nginx server instead of the default page.

So, let's say that you were in the images folder still and have the Dockerfile present over there. Create an index.html file over there with just some simple text like <h1>Hello Docker</h1>

The updated Dockerfile is shown below:

```
FROM ubuntu
MAINTAINER Romin Irani
RUN apt-get update
RUN apt-get install -y nginx
COPY index.html /usr/share/nginx/html/
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

Notice a new command COPY. This will copy the files/folders from the source to the destination in the container. On building and running this container, you will see your default page. Learn about the ADD instruction too.

Using these commands now, it should be clear to you how you can now begin to package Docker images with your software. Just think in terms of all the steps that you would normally do to install and run your software.

There is a lot more to writing Dockerfiles and I suggest that via this part, you have understood the basics of writing them. The best way is to jump into writing a file, playing with the commands and using some of the best practices (an article which I have referenced in the Additional Reading section).

Additional Reading

There are plenty of resources on writing Dockerfiles. Here are 3 that you should go through:

- Official documentation as a reference to understand any command. You should go through this.
- Best Practices article on writing Docker files that I recommend.
- Test your knowledge on Dockerfile.