Dockerfile

Objective: Participants will learn to Dockerize applications, create Docker images by using a Dockerfile, and get a more complete look at individual commands we can utilize.<br>

Preparation: Open two Terminal windows, and `cd` into the appropriate Lab folder.<br>

Outcome: Don't wanna ruin the surprise, but once you're done you'll have dockerized a C application and learn about Dockerfiles useing Figlet, which is a cool, custom made image for learning.<br>

Data Files: `Dockerfile, helloworld.c` (Both are created during the lab.)

Step 1. Create and Execute a Dockerfile

Creating a new Dockerfile is easy. Follow alone to make a new directory and initialize a new, empty Dockerfile.

1. Create a directory to hold our Dockerfile:

```
$ mkdir myimage
```

2. Create a Dockerfile inside this directory:

```
$ cd myimage
$ vim Dockerfile
```

> *NOTE: Vim is used by default, feel free to use another available text editor. Ask the instructor if this is confusing.*

3. Copy the following text inside the new Dockerfile:

```
FROM ubuntu
RUN apt-get -y update
RUN apt-get install -y figlet
```

4. After saving the file, execute as follows:

```
$ docker build -t myfiglet .
```

Output:

```
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
---> xxxxxxxxxxxx
Step 1 : RUN apt-get update
---> Running in xxxxxxxxxxxx
---> xxxxxxxxxxxx
Removing intermediate container xxxxxxxxxxxx
Step 2 : RUN apt-get install figlet
---> Running in xxxxxxxxxxxx
```

```
---> xxxxxxxxxxxx
Removing intermediate container xxxxxxxxxxxx
Successfully built xxxxxxxxxxxx
```

> *NOTE: Number's will defer depending on the build number, per host.*

5. Now, the fun part! To confimr success, let's spin up a new container and give it a command, unique to Figlet:

```
$ docker run -it myfiglet
```

```
# figlet hello
# exit
```

> *NOTE: Again, please understand that we haved used (`#`) here, because after we issused the run command and included the (`-it`) option, we are operating from inside the container itself.*

Step 2. History of a Dockerfile

This may not seem like a big deal now, but image when your Dockerfile consist of many more layers.

1. Issue a command to see the history of our newest Dockerfile:

```
docker history myfiglet
```

2. Let’s change our Dockerfile to include JSON syntax, as follows:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
```

3. Then, build the new Dockerfile again, just as we did before:

```
$ docker build -t myfiglet .
```

Output:

```
...
Successfully built xxxxxxxxxxxx
```

4. Now, compare the new history along side the other:

```
docker history myfiglet
```

Step 3. Adding CMD and ENTRYPOINT

This step will show us how to add two important Dockerfile commands: `CMD` and `ENTRYPOINT`. These two commands are often used and can be of great benefit.
These commands allow us to set a default command to run in our container.

1. We want to greet people who run a container from our image with a

nice hello, and

we want a custom font to be used. For that, we will use the `CMD` to bake `figlet -f script hello` inside our Dockerfile Image.

So, the Dockerfile we are working on should look like:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
CMD figlet -f script hello
```

2. Now that we've made our additions, let's build it:

```
$ docker build -t myfiglet .
```

Output:

```
...
Successfully built xxxxxxxxxxxx
```

3. Let's run it, like always:

```
$ docker run myfiglet
```

Output:

```

 _          _   _
```

```
| |        | | | |
| |     _  | | | |  __
|/ \   |/  |/  |/  /  \_
|   |_/|__/|__/|__/\__/
```

4. Now, let’s try to interject our own message upon running our image, like this:

```
$ docker run myfiglet howdy!
```

> *NOTE: Our attempt to interject our own message fails, as it should.*

5. Let’s replace `CMD` now with an `ENTRYPOINT` to see if that makes a difference. Open up the Dockerfile once again, and copy the following code in place of the existing, like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
```

5. Okay, build it:

```
$ docker build -t myfiglet .
```

Output:

```
...
Successfully built xxxxxxxxxxxx
```

6. And run it:

```
$ docker run myfiglet howdy!
```

Output:

```
 _
| |                    |        |
| |     __           __|        |
|/ \   /  \_|  |  |_/  |  |   | |
|   |_/\__/  \/ \/  \_/|_/ \_/|/o
                             /|
                             \|
```

Great job!

7. Almost done! Let’s add both `CMD` and `ENTRYPOINT` to our Dockerfile. This will give us the option to insert a custom message, or to print the default message. Confirm that your Dockerfile looks like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
```

```
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

8. Let’s build it:

```
$ docker build -t myfiglet .
```

Output:

```
...
Successfully built xxxxxxxxxxxx
```

9. And now run it and include a custom message:

```
$ docker run myfiglet Ahoy
```

Output:

```

  ___,   _
 /   |  | |
|    |  | |     __
|    |  |/ \   /  \_|   |
 \__/\_/|   |_/\__/  \_/|/
                       /|
                       \|
```

10. Alright, if we were successful, we should be able to print our

default message, like this:

```
$ docker run myfiglet
```

Output:

```
 _          _   _                             _
| |        | | | |                           | |    |
| |     _  | | | |  __             __   ,_   | |  __|
|/ \   |/  |/  |/  /  \_  |  |  |_/  \_/  |  |/  /  |
|   |_/|__/|__/|__/\__/    \/ \/  \__/    |_/|__/\_/|_/
```

> *NOTE: What if we want to run a shell in our container? We cannot just do `docker run myfiglet bash` because that would just tell figlet to display the word "bash." We use the `--entrypoint` parameter, like this: `$ docker run -it --entrypoint bash figlet`*

Step 4. Dockerizing an App

When we are baking up an image from a dockerfile, we need to consider which files we might we preinstalled onto our image. For that we will copy files from the "build context," which is the directory containing the Dockerfile. We can use this to "dockerize" a application. Check this out!

1. Let's now create a new directory and name it `helloworld`.

2. Inside that directory we will create a new file, we will name that file

`helloworld.c`, and we copy the following content into it:

```
int main () {
puts("Hello, world!");
return 0;
}
```

3. Now, from inside our `helloworld` directory, create a new Dockerfile and copy the following code into it:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y build-essential
COPY helloworld.c /
RUN make helloworld
CMD /helloworld
```

> *NOTE: See how we are using the COPY command in our Dockerfile to bring our application into our container. This is called “dockerizing” and it’s very useful.*

4. Okay, now for the fun part! We can test our project as we have done before. First, let’s build out our Dockerfile to create the image we will use, like this:

```
$ docker build -t worldhello .
```

5. Run it!

```
docker run worldhello
```

Output:

```
Hello, world!
```

Conclusion

We’ve covered a lot, and we have opened several cans of worms in the process. In a nutshell this has been only a small introduction into the use cases and benefits involved in using Dockerfiles. At least now we know how to create a Dockerfile, build it, explore the history, add commands, entry points, and how to copy entire apps in. This is not increditible easy stuff. Now, do a reset and let’s move on. Congrats!