Creating images - Understanding the Dockerfile

For a traditionally deployed application, building and packaging an application was often quite tedious. With the aim to automate the building and packaging of the application, people turned to different utilities, such as GNU Make, maven, gradle, etc., to build the application package. Similarly, in the Docker world, a Dockerfile is an automated way to build your Docker images. The Dockerfile contains special instructions, which tell the Docker Engine about the steps required to build an image. To invoke a build using Docker, you issue the Docker build command.

Here, an example of a typical Dockerfile example :

```
#Download base image ubuntu 16.04
FROM ubuntu:16.04

# Update Software repository
RUN apt-get update

# Install nginx, php-fpm and supervisord from ubuntu repository
RUN apt-get install -y nginx php7.0-fpm supervisor && \
rm -rf /var/lib/apt/lists/*

#Define the ENV variable
ENV nginx_vhost /etc/nginx/sites-available/default
ENV php_conf /etc/php/7.0/fpm/php.ini
ENV nginx_conf /etc/nginx/nginx.conf
ENV supervisor_conf /etc/supervisor/supervisord.conf

# Enable php-fpm on nginx virtualhost configuration
COPY default ${nginx_vhost}
RUN sed -i -e 's/;cgi.fix_pathinfo=1/cgi.fix_pathinfo=0/g' ${php_conf} && \
echo "\ndaemon off;" >> ${nginx_conf}

#Copy supervisor configuration
COPY supervisord.conf ${supervisor_conf}
RUN mkdir -p /run/php && \
chown -R www-data:www-data /var/www/html && \
chown -R www-data:www-data /run/php

# Volume configuration
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d",
"/var/log/nginx", "/var/www/html"]

# Configure Services and PortCOPY start.sh /start.sh
CMD ["./start.sh"]
EXPOSE 80 443
```

This Dockerfile describe how to deploy php7 on a nginx server.

Building Using Docker Build

We’ll return to the sample Dockerfile a bit later. Let’s try a simple Dockerfile first. Copy the following contents to a file and save it as a Dockerfile:

```
FROM ubuntu:latest
CMD echo Hello World!
```

Now build this image:

```
docker build .
```

We can see that the Docker build works in steps, each step corresponding to one instruction of the Dockerfile.

Try the build process again. In this case, the build process is much faster since Docker has already cached the layers and doesn’t have to pull them again. To run this image, use the docker run command followed by the image ID.
For example :

```
docker run 7ae54947f6a4
Hello World!
```

The Docker runtime was able to start a container and run the command defined by the CMD instruction. Hence, we get the output. Now, starting a container from an image by typing the image ID gets tedious fast. You can make this easier by tagging the image with an easy-to-remember name. You can do this by using the docker tag command, as follows:

```
docker tag image_id tag_name
docker tag 7ae54947f6a4 my_hello_world_image
```

You can also do this as part of the build process itself:

```
docker build -t my_hello_world_image .
```

You can get some informations on your image :

```
docker images  my_hello_world_image
```

Dockerfile Instructions

Currently there are about a dozen different set of commands which Dockerfiles can contain to have Docker build an image. In this section, we will go over all of them, individually, before working on a Dockerfile example.

Note: All these commands are to be listed (i.e. written) successively, inside a single plain text file (i.e. Dockerfile), in the order you would like them performed (i.e. executed) by the docker daemon to build an image. However, some of these commands (e.g. MAINTAINER) can be placed anywhere you seem fit (but always after FROM command), as they do not constitute of any execution but rather *value of a definition* (i.e. just some additional information).

ADD

The ADD command gets two arguments: a source and a destination. It basically copies the files from the source on the host into the container's own filesystem at the set destination. If, however, the source is a URL (e.g. http://github.com/user/file/), then the contents of the URL are downloaded and placed at the destination.

Example:

```
# Usage: ADD [source directory or URL] [destination directory]
ADD /my_app_folder /my_app_folder
```

CMD

The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during build, but when a container is instantiated using the image being built. Therefore, it should be considered as an initial, default command that gets executed (i.e. run) with the creation of containers based on the image.

To clarify: an example for CMD would be running an application upon creation of a container which is already installed using RUN (e.g. RUN apt-get install …) inside the image. This default application execution command that is set with CMD becomes the default and replaces any command which is passed during the creation.

Example:

```
# Usage 1: CMD application "argument", "argument", ..
CMD "echo" "Hello docker!"
```

ENTRYPOINT

ENTRYPOINT argument sets the concrete default application that is used every time a container is created using the image. For example, if you have installed a specific application inside an image and you will use this image to only run that application, you can state it with ENTRYPOINT and whenever a container is created from that image, your application will be the target.

If you couple ENTRYPOINT with CMD, you can remove "application" from CMD and just leave "arguments" which will be passed to the ENTRYPOINT.

Example:

```
# Usage: ENTRYPOINT application "argument", "argument", ..
# Remember: arguments are optional. They can be provided by CMD
#           or during the creation of a container.
ENTRYPOINT echo

# Usage example with CMD:
# Arguments set with CMD can be overridden during *run*
CMD "Hello docker!"
ENTRYPOINT echo
```

ENV

The ENV command is used to set the environment variables (one or more). These variables consist of "key value" pairs which can be accessed within the container by scripts and applications alike. This functionality of Docker offers an enormous amount of flexibility for running programs.

Example:

```
# Usage: ENV key value
ENV SERVER_WORKS 4
```

EXPOSE

The EXPOSE command is used to associate a specified port to enable networking between the running process inside the container and the outside world (i.e. the host).

Example:

```
# Usage: EXPOSE [port]
EXPOSE 8080
```

> To learn about Docker networking, check out the Docker container networking documentation.

FROM

FROM directive is probably the most crucial amongst all others for Dockerfiles. It defines the base image to use to start the build process. It can be any image, including the ones you have created previously. If a FROM image is not found on the host, Docker will try to find it (and download) from the **Docker Hub** or other container repository. It needs to be the first command declared inside a Dockerfile.

Example:

```
# Usage: FROM [image name]
FROM ubuntu
```

MAINTAINER

One of the commands that can be set anywhere in the file - although it would be better if it was declared on top - is MAINTAINER. This non-executing command declares the author, hence setting the author field of the images. It should come nonetheless after FROM.

Example:

```
# Usage: MAINTAINER [name]
MAINTAINER authors_name
```

RUN

The RUN command is the central executing directive for Dockerfiles. It takes a command as its argument and runs it to form the image. Unlike CMD, it actually **is** used to build the image (forming another layer on top of the previous one which is committed).

Example:

```
# Usage: RUN [command]
RUN aptitude install -y riak
```

USER

The USER directive is used to set the UID (or username) which is to run the container based on the image being built.

Example:

```
# Usage: USER [UID]
USER 751
```

VOLUME

The VOLUME command is used to enable access from your container to a directory on the host machine (i.e. mounting it).

Example:

```
# Usage: VOLUME ["/dir_1", "/dir_2" ..]
VOLUME ["/my_files"]
```

WORKDIR

The WORKDIR directive is used to set where the command defined with CMD is to be executed.

Example:

```
# Usage: WORKDIR /path
WORKDIR ~/
```