Section 8 - Build Images - The Dockerfile Basics

2 Build Docker Images

docker build

- Use the `docker build` command to build an image from a Dockerfile and a context.
- Usage: docker build [OPTIONS] PATH
- The most common option is `-t` to specify a Name and optionally a tag in the *name:tag* format.
- The `PATH` argument defines the context of the build, it is usually set to "." (current working directory) and by default will search for a file named Dockerfile.

docker build - example (1a)

- In the following example we are going to use the official nginx Dockerfile, available in the `resources/dockerfile-sample-2` directory, to build a custom image:

```
# cd resources/dockerfile-sample-2/
# ls
Dockerfile

# docker build -t custom_nginx .
Sending build context to Docker daemon  6.144kB
Step 1/9 : FROM debian:stretch-slim
 ---> c08899734c03
Step 2/9 : LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.
 ---> Using cache
 ---> c0199fd74028
Step 3/9 : ENV NGINX_VERSION 1.15.12-1~stretch
 ---> Using cache
 ---> e15a5d67a4e9
...
```


docker build - example (1b)

- Each Step corresponds to a line in the Dockerfile.
- Each Step will create an image layer that we can later refer to it by the hash number e.g. `---> c08899734c03`.
- The image with all the related layers are stored to the local cache.

docker build - example (1c) - local cache

- The next time that the build process takes place, before actually executing every single step, it will search in the local cache if any related image layer already exists.
- During the build process the "Docker engine" will understand for which layers of the image is possible to use the build cache and when the build cache cannot be used because:
  1. there are changes in the Dockerfile or
  2. there are changes in the files that are included in the image.
- During the build process we can see from the output `---> Using cache` when the cached is used.

docker build - example (2a)

- In the following example we will edit the Dockerfile to expose an additional TCP port `8080` and then we will perform the build process again to see which built layers are taken from the cache and which ones are created again.
- We expect the build process to be much faster that the first time since most of the layer already exists in the cache from the build process of the first example.

```
# vim Dockerfile
...
EXPOSE 80 8080
...
```

docker build - example (2b)

- Notes:
  - The fact the we expose port `8080` does not actually mean that any service will be listening to that port.
  - By exposing port 8080 we just allow the container to receive request on this port but, since there is no application listening on this port, nothing will happen.

docker build - example (2c)

```
# docker build -t custom_nginx .
...
Step 7/9 : EXPOSE 80 8080
 ---> Running in 705cb071e800
Removing intermediate container 705cb071e800
 ---> c44320477981
...
```

- The cache memory was used for all steps until `Step 7`.
- At this point cache is invalidated and all remaining layers are re-builded.

Dockerfile - order of the commands

- The order of the instructions specified in the Dockefile is important.
- Instructions that usually will cause a layer to change should be placed at the end of the Dockefile.
- For example, a command that adds our application code should be placed at the end of the Dockefile file, since it is the one that changes more often.
- Instructions that usually build the same layer should be placed on the top.

docker image default tag - *latest*

- By default the created image will be tagged as latest

```
# docker image ls
REPOSITORY                  TAG                 IMAGE ID            CREA
custom_nginx                latest              916effbcb643        5 mi
nginx                       1.15                27a188018e18        12 c
gerassimos/nginx            latest              27a188018e18        12 c
gerassimos/nginx            test1               27a188018e18        12 c
...
```