# Docker Basics

Published: 2019-04-03
Updated: 2019-04-03
Web: https://fritzthecat-blog.blogspot.com/2019/04/docker-basics.html

Docker is for deployment. It is a *virtualization at operating-system level*. When you pack your application into a docker image and run that image, the application will believe that it has the whole computer on its own. All dependencies will be in that image, as configured in a Dockerfile of your application together with build- and runtime-environment variables. You don't depend on any software being installed on your deployment target.

*Virtualization* is what virtual machines did for us until now, but docker can do that more lightweight and much faster. Originally it was available on LINUX only, because the LINUX kernel can somehow "duplicate" itself (cgroups, namespaces), which feels like more than one operating-system "instances" running on the same hardware. For WINDOWS 10 this is available as "Hyper V" feature, you must activate virtualization in the BIOS of your computer; alternatively run a LINUX VM that hosts the docker installation for WINDOWS. Such said you will understand that, instead of a VM, docker has to be installed on the machine where you want to run your docker image.

You can find all what's in this Blog on the docker homepage. Lots of docker images are available freely on the internet.

## Mind that ....

- *docker aims at the cloud, the packed app may be expected to be a 12-factor app*
- *docker doesn't provide operating-system-independence, the target operating-system must support the packed app*
- *all data that a dockerized app writes to the file system it runs upon will be lost when its container gets removed*
- *two Java apps packed into two docker images will carry two JREs and not share them*
- *all dockerized Java apps will see the same number of CPU cores and the same memory amount, so if they scale themselves (like a JVM does), they may overuse system resources (switch to Java 9 or newer and use JVM options `-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitFo` ).*

## Build Image, Run Containers

The **image** is what you **build**. The **container** is what you **run**.

The image won't change, it's the template for the container, while the container is a process that can take on states. You can instantiate many containers from one image. Containers are still present after they terminated, unless they were created using the `--rm` flag, like in `docker run --rm imagename`.

## Useful Commands

You can manage your docker platform by lots of commands that all start with "docker".

| Purpose | Command |
|---|---|
| List all images installed on computer | `docker images`<br>`docker image ls` |
| List all running containers on computer | `docker ps`<br>`docker container ls` |
| List all running and stopped containers on computer | `docker ps -a`<br>`docker container ls -a` |
| Instantiate a container from image "hello-world", with download if the image is not present | `docker run hello-world` |
| Look at the logs of container "mysql" | `docker logs --follow mysql` |
| Terminate the container with identity "mysql" | `docker stop mysql` |
| Start again the stopped container with identity "mysql", using the same file system | `docker start mysql` |
| Stop and start again the running container with identity "mysql" | `docker restart mysql` |
| Remove a container "hello-world" | `docker rm hello-world` |
| Remove an image "hello-world" | `docker rmi hello-world` |
| Start an interactive shell (bash) inside a running container "mysql" | `docker exec -it mysql bash` |
| Copy a file `/opt/mysql/my.ini` from a container "mysql" into current directory (".") | `sudo docker cp mysql:/opt/mysql/my.ini .` |
| Display the layers inside the image "hello-world" | `docker history hello-world` |
| Create an image out of the app in current directory where a *Dockerfile* exists | `docker build .` |
| Remove all stopped containers, unused images and other resources | `docker system prune -a` |
| Remove all images not connected to a container | `docker image prune -a` |

## Dockerfile Keywords

A *Dockerfile* normally is placed in the root directory of the project that needs to be packed into a docker image. It's like a C *makefile*, or a Java/Maven *pom.xml*, but it has its own syntax. Following is a short reference of some important keywords used in such a file.

| Keyword | Meaning |
|---|---|
| # | Exclusively at line start, opens a comment or a parser-directive like "`# directive=...`", "`# syntax=...`", "`# escape=...`" |
|  | The docker image this app builds on. A Java app would put a JRE here. Dependencies of |

| | |
|---|---|
| FROM | dependencies need not to be listed. A valid *Dockerfile* always starts with FROM. |
| ARG | Buildtime variable, optionally with default-value. Can be set or overwritten by an option in the `docker build` command.<br>An ENV- or ARG-variable `XXX` can be used inside a *Dockerfile* via `$XXX` or `${XXX}`. |
| ENV | Runtime variable for OS-environment of the contained app. Can be overwritten by an option in the `docker run` command. |
| LABEL | Metadata, can be recalled by `docker inspect`. |
| EXPOSE | The port where the contained app listens, defaults to a TCP type port. |
| VOLUME | Creates a directory (or mount point) shared with the outside world that will not be removed when the container gets removed. |
| WORKDIR | Sets the working directory for all `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, `ADD` statements following in the *Dockerfile*. In case `WORKDIR` doesn't exist, it will be created. |
| USER | Sets the system-user for all `RUN`, `CMD`, `ENTRYPOINT` statements following in the Dockerfile. |
| COPY | Buildtime, copies files from file system into the image to build. |
| ADD | Buildtime, copies files from file system or URLs from the network into the image to build. |
| RUN | Buildtime, executes the command to the right of RUN. |
| CMD | Runtime, starts the contained app on `docker run`. Only one CMD statement is possible in a *Dockerfile*. |

Mind that setting the image's name inside a *Dockerfile* is not supported!

## Layers and Gotchas

A *Dockerfile* like this one

```
FROM debian:jessie
ADD large_file /var/www/large_file
RUN chown www-data /var/www/large_file
RUN chmod 756 /var/www/large_file
```

that packs a `large_file` of 1 GB size and then changes its access rights will lead to a 3 GB image with following layers:

```
IMAGE          CREATED          CREATED BY                                      SIZE

49b4a4ea228a   36 seconds ago   /bin/sh -c chmod 756 /var/www/large_file        1.074

09d77316932b   2 minutes ago    /bin/sh -c chown www-data /var/www/large_file   1.074

7adb7c72c3ef   2 minutes ago    /bin/sh -c #(nop) ADD file:a86f6dedfb4ba54972   1.074

f50f9524513f   8 weeks ago      /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>      8 weeks ago      /bin/sh -c #(nop) ADD file:b5391cb13172fb513d   125.1
```

That means, every build step like `FROM, RUN, COPY, ADD` creates a new layer, and files referenced in the commands executed there will be copied always newly into the image.

This also affects final cleanups of left-over files, they would be deleted in the topmost layer only.
Docker observes the file system and notices any change after a build step (resulting in a new layer), but it doesn't associate steps and draw conclusions.

**Workaround:**

Commands that change the file system should be linked together by the '&&' operator to just one command:

```
RUN apt-get update \
    && apt-get install -y vim \
    && rm -rf /var/lib/apt
```

A `chown` could be done by a preparing "`RUN usermod -u 1000 www-data`".
Alternatively the container can be started by a script that provides all necessary circumstances:

```
docker run hello-world --entrypoint=/bin/myscript.sh
```

In the latter case the `/bin/myscript.sh` should be inside the docker image.

---