*This document was originally written approx. on 01/09/2019*

Table of Contents

[Introduction 2](#_Toc17390332)

[Tips 3](#_Toc17390333)

[Get last container id 3](#_Toc17390334)

[Cleanup 4](#_Toc17390335)

[Cleanup before docker image rebuild 5](#_Toc17390336)

[View source of the Docker image 6](#_Toc17390337)

# Introduction

* This documents is collection of commands that I found useful for *development* docker files.
* Almost all command can be applied as is in Linux, but I tested them from Git Bash on Windows.
* Here <https://github.com/alex-ber/AlexBerDocs/blob/master/Docker/Windows/Docker%20on%20Windows.docx> you can found how to install Docker on Windows.
* Commands that are provide here was tested on Docker on Windows version 19.03.1.

# Tips

## Get last container id

docker ps -q -n=1

docker ps command shows running containers.

-q means print only id  
-n=1 - show only 1 result

See [Cleanup before docker image rebuild](#_Cleanup_before_docker) section for usage example.

## Print Working Directory

On Linux the command is pwd.

But if you run in Git Bash on Windows you should use pwd –W

You can use it inside another command. For example,  
  
docker run -v $(pwd –W):/work -d --name *my\_container\_name* *image\_name* tail -f /dev/null  
  
Alternatively, you can write

docker run -v `pwd –W`:/work -d --name *my\_container\_name* *image\_name* tail -f /dev/null

## Cleanup

See <https://stackoverflow.com/a/17237701>

<https://stackoverflow.com/a/39860665>

docker container prune -f

docker system prune -f

1. First command will remove all stopped containers.
2. Second command will clean up all unused containers, networks, images (both dangling and unreferenced), and optionally, volumes, in one command.

## Cleanup before docker image rebuild

docker container stop $(docker ps -q -n=1); docker rm $(docker ps -q -n=1); docker rmi image\_for\_ container

1. First command stops last running container (you can provide container name or container id).
2. Second command removes last running container (you can provide container name or container id).
3. Third command removes image form which container was built.

## View source of the Docker image

docker history --no-trunc image\_name

--no-truck means don’t truncate output – print the whole command for each layer.

## Building Docker Image

### Typical case

Go to the directory that contains your Dockerfile and type:

docker build . -t -*imagename*

Note:

* The dot between build and -t is required, it represent a “context”. The docker build command builds Docker images from a Dockerfile and a “context”. A build’s context is the set of files located in the specified PATH or URL.
* -t is used to tag an image. For example, docker build -t vieux/apache:2.0 the repository name will be vieux/apache and the tag will be 2.0

### Specify a Dockerfile (-f)

$ docker build -f dockerfiles/Dockerfile.debug -t myapp\_debug .

$ docker build -f dockerfiles/Dockerfile.prod -t myapp\_prod .

The above commands will build the current build context (as specified by the .) twice, once using a debug version of a Dockerfile and once using a production version.

$ curl example.com/remote/Dockerfile | docker build -f - .

The above command will use the current directory as the build context and read a Dockerfile from stdin.

### Squashing

See <https://vsupalov.com/build-docker-image-clone-private-repo-ssh-key/>

In Docker 1.13, a new --squash parameter was added. It can be used to reduce the size of an image by removing files which are not present anymore, and reduce multiple layers to a single one between the origin and the latest stage. You’ll need to run the daemon with experimental features enabled to use it.

This has also the convenient side effect, of removing files which were created and then deleted.

You tell docker to squash away layers when executing docker build:

$ docker build --squash [...]

Note:

* You’re not making use of Docker layer caching as much as you could.
* **Docker history command will still show the history**, that is, if you run command

$ docker history --no-trunc *imagename*

you will see all commands that creates the layer (with the name missing attached to it; the layers itself is missing, but you do see the commands itself).

### How to hide build commands

See <https://blog.developer.atlassian.com/minimal-java-docker-containers/>   
  
If you want to strip an image of all its layers and flatten it, that is to reduce the whole image to one layer, you can use this technique. For example, it can be helpful for your base images.

docker run **-**d *image\_name*  
docker export $(docker ps **-**q **-**n=1) | docker import **–** *imagename-stripped*

1. You should create container for the given image (perhaps, you should built it from source first).
2. This export container’s filesystem as a tar archive and immediately imports tar file as image.

End result you see only one layer (the import) and you don’t see any details.

Note:

* If your image file is big you will want to split the command to the following:  
    
  docker export $(docker ps -q -n=1) > latest.tgz; cat latest.tgz | docker import - *imagename-stripped;* rm latest.tgz
* If you run command

docker history --no-trunc *imagename-stripped*

You will see only one layer (import). That is this enable to hide “the source code” (commands that creates each layer) of the Dockerfile.

### Multi-stage Builds

See <https://vsupalov.com/build-docker-image-clone-private-repo-ssh-key/>

When working with multi-stage builds, you are building multiple Docker images in a single Dockerfile, but only the last one is the real result. The other ones are there to support it. Anything but the final image don’t leave any traces.

Here is an example of a multi-stage Dockerfile:

# this is our first build stage, it will not persist in the final image

FROM ubuntu as intermediate

# install git

RUN apt-get update

RUN apt-get install -y git

# add credentials on build

ARG SSH\_PRIVATE\_KEY

RUN mkdir /root/.ssh/

RUN echo "${SSH\_PRIVATE\_KEY}" > /root/.ssh/id\_rsa

# make sure your domain is accepted

RUN touch /root/.ssh/known\_hosts

RUN ssh-keyscan bitbucket.org >> /root/.ssh/known\_hosts

RUN git clone git@bitbucket.org:your-user/your-repo.git

FROM ubuntu

# copy the repository form the previous image

COPY --from=intermediate /your-repo /srv/your-repo

# ... actually use the repo :)

There are two images defined here. One of them is named “intermediate”, the final one doesn’t have a name. The “intermediate” image is referenced, and we’re copying the repository data over from it into the final image.

The SSH\_PRIVATE\_KEY is passed when issuing the build command with --build-arg or in the build block of your docker-compose.yml file. That ARG variable is not used in the final image, the value will not be available using the history command.

Using multi-stage builds also has the great side effect of **significantly** reducing the size of your final Docker images, as they don’t need to contain traces of Git and other build tools if used correctly.

## 