*This document was originally written approx. on 03/12/2019*

Table of Contents

[Introduction 2](#_Toc17952757)

[Tips 3](#_Toc17952758)

[Get last container id 3](#_Toc17952759)

[Print Working Directory 4](#_Toc17952760)

[Cleanup 5](#_Toc17952761)

[Cleanup before docker image rebuild 6](#_Toc17952762)

[View source of the Docker image 7](#_Toc17952763)

[Building Docker Image 8](#_Toc17952764)

[Typical case 8](#_Toc17952765)

[Specify a Dockerfile (-f) 9](#_Toc17952766)

[Squashing 10](#_Toc17952767)

[How to strip an image and hide build commands 11](#_Toc17952768)

[Multi-stage Builds 12](#_Toc17952769)

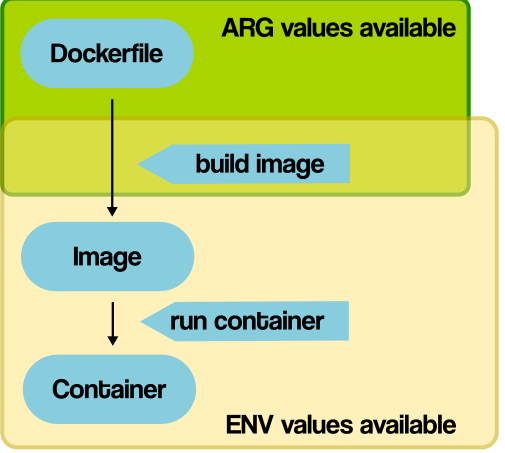
[Starting Docker Container 13](#_Toc17952770)

[Typical case 13](#_Toc17952771)

# 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.
* The structure of this document follows following picture:  
  - [General tips](#_Tips) (cleanup, etc)  
  - [Building docker Image](#_Building_Docker_Image_1)  
  - [Starting docker container](#_Starting_Docker_Container)  
  - [Attaching to running docker container](#_Attaching_to_running_1)  
  - [Create new docker image from a docker container’s changes](#_Create_new_docker) (docker commit)  
  - [Pushing docker image](#_Publishing_docker_image) (docker push)  
  - [Logs of docker container](#_Logs_of_docker)

* Some advanced techniques are put in appendix. Appendix mimics the structure above.
* In particular, see [Mutli-stage Builds](#_Multi-stage_Builds), [network](#_Network), [busy wait](#_Busy_wait).



# Tips

## Cleanup before docker image rebuild

docker rmi -f *image\_name*  
docker rm –f *container\_name*

Or

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

1. docker rmi –f if there is running container from this image, it will be stop it and remove it. Anyway, it will remove docker image.
2. docker rm –f if container is running it will stop it first and then it will remove it.
3. First command stops [last running container](#_Get_last_container) (you can provide container name or container id).
4. Second command removes [last running container](#_Get_last_container) (you can provide container name or container id).
5. Third command removes image form which container was built.

## 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.

# Building Docker Image

See <https://docs.docker.com/engine/reference/commandline/build/>   
  
See also [multi-stage builds](#_Multi-stage_Builds).

## Typical case

Go to the directory that contains your Dockerfile and type:

docker build . -t –*image\_name*

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.
* If you use dot as described above all files in the current directory and **all subdirectories** will be send to docker daemon. In particular, if you have .git folder with some big file (even in unreachable commit history) it will be send to docker daemon. This can cause to the build to take some extra time.
* -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 *image\_name*

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).

## No-cache

docker build --no-cache. -t -*imagename*

Do not use cache when building the image. This can be useful, if you use git clone to build your application. You don’t want to use cache, you want to execute each step.

## Build-args

#docker build --no-cache --squash . -t my-web-service --build-arg TOMCAT\_SERVER\_USERNAME --build-arg TOMCAT\_SERVER\_PASSWORD

For explanation about --no-cache see [this](#_No-cache) section.

For explanation about --squash see [this](#_Squashing) section.

If your dockerfile has following lines:  
  
ARG GIT\_USERNAME

ARG GIT\_PASSWORD  
  
You can supply these argument by specifying --build-arg.

You can specify arguments explicitly such as -build-arg TOMCAT\_SERVER\_USERNAME=admin or implicitly as in example above. In implicit way, the value of the variable will be taken from the ENV (environment variables in Windows, exported variable in Linux) of the Host OS.  
  
See also [Docker ARG vs ENV](#_Docker_ARG_vs).

# Starting Docker Container

See <https://docs.docker.com/engine/reference/commandline/run/>

## Typical case

docker run -p 8000-8100:8000-8100 -p 10000:10000 -d --name *new\_container\_name* *image\_name*

The docker run command first creates a writeable container layer over the specified image, and then starts it using the specified command.

Note:

* -p publish a container’s port(s) to the host. For example, Port 10000 on the host OS will mapped to port 10000 on the docker container. You can also maps ranges of ports.
* -d (detach mode) - Run container in background and print container ID. Note: it is better that container will not finish to run immediately (see more advanced example below how you can overcome this), it should wait, otherwise, container will immediately stops.
* --name Assign a name to the container. Note, that you can’t start new container with the same name. So, if you give the name to container and you want to rerun it, you should [remove the previous container first](#_Cleanup_before_docker).

## One-off shout

docker run **-**it *some\_image\_name* mvn **-**version

We want to execute some command inside container and we want to see the result. In this example, we will see that maven was successfully installed (smoke test).  
  
See also [attaching to running Docker container](#_Attaching_to_running). The difference between docker exec and docker run, that in the former case we’re running command inside running docker container (for example, it has running java service and we want to validate that he is up), in the latter case, the is no running docker container, we’re spinning new one and we use our command as CMD. This command will be executed and docker container will be stopped. For more details see [Busy wait](#_Busy_wait).  
  
Note:

* The -it instructs Docker to allocate a pseudo-TTY connected to the container’s stdin.
* “mvn -version" will be run in attached pseudo-TTY. Actually, this command will be interpreted by *entrypoint* (the default one is “/bin/sh”). **Note:** This is not bash, **this is sh** shell.

## Overriding entrypoint

docker run **-**it *--entrypoint="bash"*

*some\_image\_name*

We want to enter bash shell inside docker container.

Note:

* The -it instructs Docker to allocate a pseudo-TTY connected to the container’s stdin.
* *bash* will be interpreted by *entrypoint* (the default one is “/bin/sh”). So, when docker container will be initialized, bash will be started.

**Note:** If your dockerfile has **CMD** [**"/bin/sh"**] command and you want to overwrite it with bash and you want that container will stay active you can use this command.  
  
**Note:** See also [busy wait](#_Busy_wait).

# Attaching to running Docker container

See <https://docs.docker.com/engine/reference/commandline/exec/>

docker exec -it $(docker ps -q -n=1) bash

docker exec -it $(docker ps -q -n=1) ps aux | grep java

The commands above use [last running container](#_Get_last_container) (you can provide container name or container id).

The first command above will start bash session inside the running container.

The second command above will print java process that runs inside docker container.  
  
See also [One-off shout](#_One-off_shout). The difference between docker exec and docker run, that in the former case we’re running command inside running docker container (for example, it has running java service and we want to validate that he is up), in the latter case, the is no running docker container, we’re spinning new one and we use our command as CMD. This command will be executed and docker container will be stopped. For more details see [Busy wait](#_Busy_wait).

\*\*\*

The docker exec command runs a new command in a running container.

The command started using docker exec only runs while the container’s primary process (PID 1) is running.

COMMAND should be an executable.

A chained or a quoted command will not work. Example: docker exec -it my\_container "echo a && echo b" will not work, but docker exec -ti my\_container sh -c "echo a && echo b" will.

# Create new docker image from a docker container’s changes

See <https://docs.docker.com/engine/reference/commandline/commit/>

docker commit --change "ENV DEBUG true" $(docker ps -q -n=1) new\_image\_name  
  
f5283438590d

docker inspect -f "{{ .Config.Env }}" f5283438590d  
  
[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin DEBUG=true]

docker commit --change='CMD ["apachectl", "-DFOREGROUND"]' --change "EXPOSE 80" $(docker ps -q -n=1) new\_image\_name

f5283438590d

The commands above use [last running container](#_Get_last_container) (you can provide container name or container id).  
  
First command changes ENV, it adds DEBUG=true (you can see the change in docker inspect in the next line). You can alter any variable in such a way.

Second command changes CMD to ‘apachectl -DFOREGROUND' and change EXPOSE to add port 80.

\*\*\*  
  
  
It can be useful to commit a container’s file changes or settings into a new image.

The commit operation will not include any data contained in volumes mounted inside the container.

By default, the container being committed and its processes will be paused while the image is committed. This reduces the likelihood of encountering data corruption during the process of creating the commit. If this behavior is undesired, set the --pause option to false.

The --change option will apply Dockerfile instructions to the image that is created. Supported Dockerfile instructions: CMD|ENTRYPOINT|ENV|EXPOSE|LABEL|ONBUILD|USER|VOLUME|WORKDIR

# Publishing docker image

See <https://docs.docker.com/engine/reference/commandline/tag/>

<https://docs.docker.com/engine/reference/commandline/push/>

*docker tag alex-docker-hive alexberkovich/docker-hive:0.1.0  
docker push alexberkovich/docker-hive:0.1.0  
docker tag alex-docker-hive alexberkovich/docker-hive  
docker push alexberkovich/docker-hive*

First command creates tag *docker-hive* with version 0.1.0 in repository *alexberkovich* in Dockerhub from local image alex-docker-hive.

Second command pushes newely created tag to Dockerhub.

Third command creates tag *docker-hive* with latest version in repository *alexberkovich* in Dockerhub from local image alex-docker-hive.

Forth command pushes newely created tag to Dockerhub.  
  
**Note**: If you run command one after another (it is sufficient that you don’t change alex-docker-hive image) than Dockerhub will detect that all layers that are needed to create *docker-hive:latert* are present in Dockherhub and he will reuse them (the layers of docker container will not be sent twice, only their metada).

# Logs of docker container

See also <https://docs.docker.com/config/containers/logging/>

<https://docs.docker.com/engine/reference/commandline/logs/>

*docker logs $(docker ps –q –n=1)*

This command will fetch the logs of the [last running container](#_Get_last_container) (you can provide container name or container id).  
  
\*\*\*

The information that is logged and the format of the log depends almost entirely on the container’s endpoint command.  
  
By default, docker logs or docker service logs shows the command’s output just as it would appear if you ran the command interactively in a terminal. UNIX and Linux commands typically open three I/O streams when they run, called STDIN, STDOUT, and STDERR. STDIN is the command’s input stream, which may include input from the keyboard or input from another command. STDOUT is usually a command’s normal output, and STDERR is typically used to output error messages. By default, docker logs shows the command’s STDOUT and STDERR.

If you use a [logging driver](https://docs.docker.com/config/containers/logging/configure/) which sends logs to a file, an external host, a database, or another logging back-end, docker logs may not show useful information. In this case, your logs are processed in other ways and you may choose not to use docker logs.

If your image runs a non-interactive process such as a web server or a database, that application may send its output to log files instead of STDOUT and STDERR.  
  
There are 2 workarounds in this case. One is used in [official nginx image](https://github.com/nginxinc/docker-nginx/blob/a973c221f6cedede4dab3ab36d18240c4d3e3d74/mainline/alpine/Dockerfile) and another is used in [official httpd image](https://github.com/docker-library/httpd/blob/3616f04657a48b960d842956c4a6b1581e7cecdb/2.4/alpine/Dockerfile).  
  
1. The official nginx image creates a symbolic link from /var/log/nginx/access.log to /dev/stdout, and creates another symbolic link from /var/log/nginx/error.log to /dev/stderr, overwriting the log files and causing logs to be sent to the relevant special device instead.  
  
**RUN** \  
*# forward request and error logs to docker log collector***&&** ln **-**sf **/**dev**/**stdout **/**var**/**log**/**nginx**/**access.log \  
**&&** ln **-**sf **/**dev**/**stderr **/**var**/**log**/**nginx**/**error.log

2. The official httpd driver changes the httpd application’s configuration to write its normal output directly to /proc/self/fd/1 (which is STDOUT) and its errors to /proc/self/fd/2 (which is STDERR)

**RUN** \  
sed **-**ri \  
 **-**e **'s!^(\s\*CustomLog)\s+\S+!\1 /proc/self/fd/1!g'** \  
 **-**e **'s!^(\s\*ErrorLog)\s+\S+!\1 /proc/self/fd/2!g'** \  
 **-**e **'s!^(\s\*TransferLog)\s+\S+!\1 /proc/self/fd/1!g'** \  
 **"**$***HTTPD\_PREFIX*/conf/httpd.conf"** \  
 **"**$***HTTPD\_PREFIX*/conf/extra/httpd-ssl.conf"** \  
;

# Appendix

## Tips Adanced

### 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.

### 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.

### 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

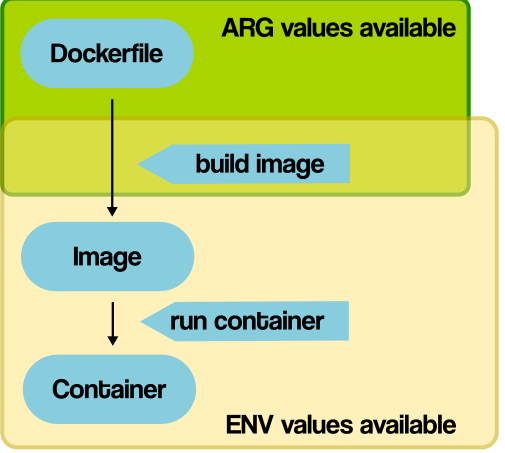
docker run -v `pwd –W`:/work -d --name *my\_container\_name* *image\_name* tail -f /dev/null

## Buiding Docker Image Advanced

### Docker ARG vs ENV

See <https://vsupalov.com/docker-arg-vs-env/>

ENV is mainly meant to provide default values for your future environment variables. Running dockerized applications can access environment variables.

ARG values are not available after the image is built. A running container won’t have access to an ARG variable value. You can imagine the ARG and ENV as two overlapping rectangles:  
  


Notice how both ARG and ENV overlap **during the image build**? This causes confusion sometimes: from within your Dockerfile, both ARG and ENV seem very similar. Both can be accessed from within your Dockerfile commands in the same manner.

ARG VAR\_A 5

ENV VAR\_B 6

RUN echo $VAR\_A

RUN echo $VAR\_B

Just looking at the RUN commands, you couldn’t tell which one is an ARG and which one is an ENV variable.

Build arguments can be set to a default value inside of a Dockerfile:

ARG VAR\_NAME 5

but also changed by providing a --build-arg VAR\_NAME=6 argument when you build your image. In a similar way, you can specify default values for ENV variables:

ENV VAR\_NAME\_2 6

But unlike ARG, you can’t override ENV values directly from the commandline when building your image. However, *you can use ARG values to dynamically set default values of ENV* variables during the build like this:

# You can set VAR\_A while building the image

# or leave it at the default

ARG VAR\_A 5

# VAR\_B gets the (overridden) value of VAR\_A

ENV VAR\_B $VAR\_A

### How to strip an image and 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.

## Starting Docker Container Advanced

### Mount

docker run **-**v C:**/**dev**/**work:**/**work –d --name *container\_name* **-**it *some\_image\_name*

This command will run named container from the image in detached mode, using pseudo-TTY connected to the container’s stdin. Also host directory C:**/**dev**/**work (My host OS is Windows) will be mapped to the docker directory. In this example, /work will be available in docker.

Note:

* It has read/write access:
* All changes made in the Host OS are visible to docker.
* All changes made in the docker are visible to Host OS.
* When docker container stop all changes in the mapped volume persists (in the Host OS).
* If you want to specify current directory in the mapping [you can use pwd command. On Windows, you should use `pwd –W`.](#_Building_Docker_Image)

## Attaching to running Docker container Advanced

### Busy wait

**CMD** tail **-**f **/**dev**/**null

When you are developing new dockerfile, especially when you build base image, sometimes you want to stop regular image building and run the command in the shell (bash) environment by yourself.  
  
For example, you have some problem with command executing, but what you see in the console is only code 1 and you want to see more details. You can run command docker logs $(docker ps -q -n=1); because there is no running container (you don’t have even docker image).

So, what you can do is to change CMD command to the one written above. You can comment out the command that fails (and, optionally, all subsequent command), build the docker image, [start it](#_Building_Docker_Image) (with docker run), then [attach to the running container](#_Exec) (*docker exec -it $(docker ps -q -n=1); bash*). Now, you inside docker image in the bash. You can run the problematic command and investigate the failure.

Let’s walk through the syntax. **CMD** means that this command will be run as initialize command of the docker running, that is it will be run when you will start your container. By default, it is /bin/sh (shell). In such case docker container will start and immediately ends. It can be overwritten in your parent docker file.

If want to docker container to stay alive, we should run some command that will take time for accomplish. It can be, for example, pause 500000.

tail - prints content, usually of the file (tail **–**f). but it can be any file descriptor. **/**dev**/**null is the **null device,** it is a convenient empty file for input streams. Formally, it's described as being an infinite data sink. Because it is infinite (but empty) this command will never ends.

**Note:** Theoretically, you can overwrite the **CMD** command when you [start](#_Building_Docker_Image) the container with docker run (this is similar to what you’re doing in docker exec command above; *bash* will be executing inplace of **CMD** command). Practically, however, at least Git bash on Windows fails to do it.

**Note:** If your dockerfile has **CMD** [**"/bin/sh"**] command and you want to overwrite it with bash and you want that container will stay active you can use [*docker run -it --entrypoint="bash" image\_name* command](#_Overriding_entrypoint).

## Misc

### Network

--network="host" (--link)

By default, all containers have networking enabled and they can make any outgoing connections. Publishing ports and linking to other containers only works with the default (bridge).

Supported networks :

| **Network** | **Description** |
| --- | --- |
| **none** | No networking in the container. |
| **bridge** (default) | Connect the container to the bridge via veth interfaces. |
| **host** | Use the host's network stack inside the container. |
| **container**:<name|id> | Use the network stack of another container, specified via its *name* or *id*. |
| **NETWORK** | Connects the container to a user created network (using docker network create command) |

#### Network: bridge

With the network set to bridge a container will use docker’s default networking setup. A bridge is setup on the host, commonly named docker0, and a pair of veth interfaces will be created for the container. One side of the veth pair will remain on the host attached to the bridge while the other side of the pair will be placed inside the container’s namespaces in addition to the loopback interface. An IP address will be allocated for containers on the bridge’s network and traffic will be routed though this bridge to the container.

Containers can communicate via their IP addresses by default. To communicate by name, they must be linked.

#### Network: host

With the network set to host a container will share the host’s network stack and all interfaces from the host will be available to the container. The container’s hostname will match the hostname on the host system.

Compared to the default bridge mode, the host mode gives significantly better networking performance since it uses the host’s native networking stack whereas the bridge has to go through one level of virtualization through the docker daemon. It is recommended to run containers in this mode when their networking performance is critical, for example, a production Load Balancer or a High Performance Web Server.

**Note**: --network="host" gives the container full access to local system services such as D-bus and is therefore considered insecure.

#### --link

-link="" : Add link to another container (<name or id>:alias or <name or id>)

If the operator uses --link when starting a new client container in the default bridge network, then the client container can access the exposed port via a private networking interface. If --link is used when starting a container in a user-defined network, it will provide a named alias for the container being linked to.