Step-by-step guide to Docker

Basic commands

1. Run the hello-world Docker container to verify basic functionality

```
$ docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest:
sha256:49a1c8800c94df04e9658809b006fd8a686cab8028d33cfba2cc049724254202
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

2. Pull an image from Docker Hub

```
docker pull <image name>
```

EXAMPLE:

```
$ docker pull debian

Using default tag: latest
latest: Pulling from library/debian
4c25b3090c26: Pull complete
Digest:
sha256:38988bd08d1a5534ae90bea146e199e2b7a8fca334e9a7afe5297a7c919e96ea
```

```
Status: Downloaded newer image for debian:latest
docker.io/library/debian:latest
```

3. Run a container and print OS information

```
docker run [options] <image name> <command>
```

We can display information about the OS by printing the `/etc/os-release` file:

```
$ docker run debian cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

Compare this information with those from your native OS.

4. Run an interactive shell inside a container

```
docker run -it <image name> bash
```

`-i` stands for interactive

`-t` allocates a pseudo-TTY

EXAMPLE:

```
$ docker run -it debian bash

root@9eed5b3d3044:/# whoami
root
root@9eed5b3d3044:/# ls
bin   boot   dev   etc   home   lib   lib64   media   mnt   opt   proc   root   run   sbin
srv   sys   tmp   usr   var
root@9eed5b3d3044:/# exit
```

5. List Docker images in the system

```
$ docker images

REPOSITORY     TAG       IMAGE ID       CREATED         SIZE
debian         latest    fe3c5de03486   7 days ago      124MB
hello-world    latest    bf756fb1ae65   19 months ago   13.3kB
```

6. Run a container from an image with a tag different from `latest`

   The general identifier of Docker images is in the form `<registry>/<user name>/<image name>:<image tag>`; in case of officially hosted images on Docker Hub, the form is simply `<image name>:<tag>`. If the tag is not specified, Docker will default it to `latest`.

   EXAMPLE:

```
$ docker pull debian:stretch

stretch: Pulling from library/debian
08224db8ce18: Pull complete
Digest:
sha256:06f9296409de8cfecaff43aaee6d608ed48a95c0cac0da2a418ff526acedf67b
Status: Downloaded newer image for debian:stretch
docker.io/library/debian:stretch


$ docker run debian:stretch cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
NAME="Debian GNU/Linux"
VERSION_ID="9"
VERSION="9 (stretch)"
VERSION_CODENAME=stretch
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

   Compare this example with those from points 2 and 3.

7. Write a simple Dockerfile

   Dockerfiles are made of a sequence of commands to incrementally build the environment that will constitute a Docker image. They are similar to Bash scripts, with the addition of some specific keywords, called instructions. Every statement in a Dockerfile must start with an instruction.

   The simplest and most useful instructions are:

   - **FROM**: identify an already existing image as a base image; the subsequent instructions in the Dockerfile will add stuff on top of what's already defined in that image.
   - **COPY**: copy files and directories from a source path into the image. The destination path will be automatically created if it does not exist.

- **RUN**: execute any command as if you were into a shell. RUN instructions usually make up the most of a Dockerfile, either installing software from the package manager or downloading and compiling resources.
- **ENV**: create a new environment variable in the image

EXAMPLE:

```
FROM debian:latest

COPY script.sh /app/

RUN chmod 755 /app/script.sh
RUN apt-get update && apt-get install -y wget

ENV NAME JohnDoe
```

8. Build an image from a Dockerfile

```
docker build -t <name:tag> -f <Dockerfile path> <build context>
```

`-t` associates a user-supplied identifier to the new image. It is useful to already choose an identifier suitable for Docker Hub.

`-f` indicates the location of the Dockerfile to use. When the option is not provided, Docker defaults to a file called `Dockerfile` in the current directory. `-f` is thus useful when Dockerfiles have more elaborate names or reside in a different directory.

The build context is the set of files which will be available to the image builder. It usually corresponds to the current directory (i.e. `.`). The build context is used, for example, to find the files used by a `COPY` instruction.

EXAMPLE (with default Dockerfile and build context in the current directory):

```
$ docker build -t my_user/my_image:latest .
```

NOTE: In the example above and throughout the rest of the document, `my_user` and `my_image` are used as placeholders in image identifiers. You are encouraged to replace them with your Docker Hub ID and chosen image name respectively!

9. Login and push an image to Docker Hub

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID,     head over to https://hub.docker.com to create one.
Username (<last logged user>):
Password:
Login Succeeded

$ docker push my_user/my_image:latest
```

Additional commands

1. List all Docker containers in the system (even stopped ones)

```
$ docker ps -a

CONTAINER ID    IMAGE              COMMAND                 CREATED
STATUS                       PORTS    NAMES
b8de0659bae5    debian:stretch     "cat /etc/os-release"   12 minutes ago
Exited (0) 12 minutes ago             hopeful_payne
9eed5b3d3044    debian             "bash"                  25 minutes ago
Exited (0) 25 minutes ago             zen_poitras
dc0faa01b3a4    debian             "cat /etc/os-release"   26 minutes ago
Exited (0) 26 minutes ago             wizardly_herschel
526cc2a156f6    hello-world        "/hello"                35 minutes ago
Exited (0) 35 minutes ago             admiring_mendeleev
```

2. Run a container with automatic removal upon exit

   By default, Docker does not delete containers after they complete the tasks assigned to them and return control to the shell. Instead, those containers remain in a stopped state, ready to be resumed if the user wishes so. To run a container that will be automatically removed when it exits, use the `--rm` option of `docker run`.

   EXAMPLE:

```
$ docker ps -a

CONTAINER ID    IMAGE              COMMAND                 CREATED
STATUS                       PORTS    NAMES
b8de0659bae5    debian:stretch     "cat /etc/os-release"   12 minutes ago
Exited (0) 12 minutes ago             hopeful_payne
9eed5b3d3044    debian             "bash"                  25 minutes ago
Exited (0) 25 minutes ago             zen_poitras
dc0faa01b3a4    debian             "cat /etc/os-release"   26 minutes ago
Exited (0) 26 minutes ago             wizardly_herschel
526cc2a156f6    hello-world        "/hello"                35 minutes ago
Exited (0) 35 minutes ago             admiring_mendeleev

$ docker run --rm debian:latest cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

$ docker ps -a

CONTAINER ID    IMAGE              COMMAND                 CREATED
```

```
STATUS                        PORTS    NAMES
b8de0659bae5    debian:stretch    "cat /etc/os-release"   15 minutes ago
Exited (0) 15 minutes ago            hopeful_payne
9eed5b3d3044    debian            "bash"                  28 minutes ago
Exited (0) 27 minutes ago            zen_poitras
dc0faa01b3a4    debian            "cat /etc/os-release"   29 minutes ago
Exited (0) 29 minutes ago            wizardly_herschel
526cc2a156f6    hello-world       "/hello"                38 minutes ago
Exited (0) 38 minutes ago            admiring_mendeleev
```

3. Remove containers

```
docker rm <container ID or name> [<container ID or name>...]
```

EXAMPLE:

```
$ docker ps -a

CONTAINER ID    IMAGE             COMMAND                 CREATED
STATUS                        PORTS    NAMES
b8de0659bae5    debian:stretch    "cat /etc/os-release"   15 minutes ago
Exited (0) 15 minutes ago            hopeful_payne
9eed5b3d3044    debian            "bash"                  28 minutes ago
Exited (0) 27 minutes ago            zen_poitras
dc0faa01b3a4    debian            "cat /etc/os-release"   29 minutes ago
Exited (0) 29 minutes ago            wizardly_herschel
526cc2a156f6    hello-world       "/hello"                38 minutes ago
Exited (0) 38 minutes ago            admiring_mendeleev

$ docker rm b8de0659bae5

b8de0659bae5

$ docker rm 9eed5b3d3044 dc0faa01b3a4 admiring_mendeleev

9eed5b3d3044
dc0faa01b3a4
admiring_mendeleev

$ docker ps -a
CONTAINER ID    IMAGE             COMMAND                 CREATED
STATUS                        PORTS    NAMES
```

A useful command to remove all *stopped* containers on the system at once is `docker container prune`:

EXAMPLE:

```
$ docker container prune

WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
```

```
7062ceb8a5d5cd1374538d400c0e048a97b39d9a3744dc4b03c4d06851132f2a
0b5de7d8a0f50845ef0a7357bb3da384dc3abdbf03ac2323026bdf634f88b0b7
fc34d157aa1cdf37cc6d03f4c0a61b90453ae30968d1f2309256941aa3c5a51c
a8c90e2678e72c3db402b6df4669e909bdd39aec49fc9ef8664158feace68db2

Total reclaimed space: 15B
```

4. Remove Docker images

```
docker rmi <image ID or name> [<image ID or name>...]
```

EXAMPLE:

```
$ docker images

REPOSITORY     TAG        IMAGE ID        CREATED           SIZE
debian         stretch    2c3ad12c6ecf    2 weeks ago       101MB
debian         latest     0980b84bde89    2 weeks ago       114MB
hello-world    latest     bf756fb1ae65    19 months ago     13.3kB

$ docker rmi 2c3ad12c6ecf 0980b84bde89 hello-world

Untagged: debian:stretch
Untagged:
debian@sha256:06f9296409de8cfecaff43aaee6d608ed48a95c0cac0da2a418ff526acedf67b
Deleted:
sha256:2c3ad12c6ecf3c5b9dd9893c516374a5ec060e5b7881616d936b5c82208e9b65
Deleted:
sha256:40093787e10fa68c547b983c1adbf4dfb0fdef10110306ad3cf59dbaec7fc688
Untagged: debian:latest
Untagged:
debian@sha256:cc58a29c333ee594f7624d968123429b26916face46169304f07580644dde6b2
Deleted:
sha256:0980b84bde890bbdd5db43522a34b4f7c3c96f4d026527f4a7266f7ee408780d
Deleted:
sha256:afa3e488a0ee76983343f8aa759e4b7b898db65b715eb90abc81c181388374e3
Untagged: hello-world:latest
Untagged: hello-
world@sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c3db9a9
Deleted:
sha256:bf756fb1ae65adf866bd8c456593cd24beb6a0a061dedf42b26a993176745f6b
Deleted:
sha256:9c27e219663c25e0f28493790cc0b88bc973ba3b1686355f221c38a36978ac63

$ docker images
REPOSITORY     TAG        IMAGE ID        CREATED         SIZE
```

When failing to complete a build from a Dockerfile, we may end up with *dangling* images, i.e. images which are not tagged and are not referenced by any container. An untagged image, and thus possibly dangling, is indicated by the value `<none>` in the `REPOSITORY` and `TAG` columns of the list printed by `docker images`. When developing complex Dockerfiles, it is not unusual to accumulate a number of dangling images, which take up disk space. To cleanup dangling images, we can use the `docker image prune` command.

EXAMPLE:

```
$ docker images

REPOSITORY    TAG       IMAGE ID       CREATED              SIZE
<none>        <none>    d7203f4672a2   About a minute ago   114MB
debian        buster    0980b84bde89   2 weeks ago          114MB

$ docker image prune

WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Deleted Images:
deleted:
sha256:d7203f4672a2b74f1a07fe125602658a11ce3059ea51ea1e7fc2209fbed3afa3
deleted:
sha256:158425ddb467311f175f87c3e812de6adad1c3bdf34e31fa7412dcb20fe76c76
deleted:
sha256:e68814ceff7a2ef05f3b1c708d5a6cdb71f62f586b195f7b6b64e5d2d1e6ed6e
deleted:
sha256:93205f715db86e455980490f3ea71db8d6247abf524ea62c34769516c019a65c

Total reclaimed space: 732B
```

NOTE: `docker image prune` also accepts a `-a` option, which will remove **all** images not in use by existing containers. This may remove useful images you wish to keep on the system, so use the command with care.

5. Assign a different identifier to an existing image

```
docker tag <source image> <target image>
```

EXAMPLE:

```
$ docker tag dummy_image my_user/awesome_image:latest
```

Additional Dockerfile instructions

- **ADD**: copy files, directories and remote file URLs from a source into the image. It also automatically extracts tar archives into the image, which constitutes its best use case. Since the additional features of `ADD` are not immediately obvious, the official Docker documentation indicates `COPY` as the preferred instruction if files have to simply be transferred into an image.
- **WORKDIR**: set the working directory for subsequent instructions in the Dockerfile. If the `WORKDIR` doesn't exist, it will be created. Without a `WORKDIR` instruction, all actions in a Docker file happen at the filesystem root.
- **CMD**: provide default arguments for a container. These are the arguments used when nothing is entered after the image name in a `docker run` command. Providing arguments as part of `docker run` overrides the defaults set by `CMD`.
- **LABEL**: add metadata to an image in a key-value pair. An image can have multiple labels. Labels are additive, including labels in the base image indicated with `FROM`. Labels are useful for improved image classification and are sometimes used by third-party software.

Basic Dockerfile good practices

- **Do not use too many image layers:** Docker images are built from a series of layers, stacked on top of each other. Each layer represents an instruction in the image's Dockerfile and is simply a set of differences from the layer before it.

  Try to achieve a balance between readability of the Dockerfile and reducing the number of image layers. Minimizing the number of layers also benefits total image size and performance of build and pull processes.

- **Cleanup after installations:** Because of the layered structure of Docker images, if files are downloaded and removed with different instructions, a copy of those files will still exist in the layer associated with the first instruction that retrieved them.

  You can reduce the total image size by using a single `RUN` instruction that also cleans the package manager cache, or performs a complete installation from source and removes the original code.

  Examples:

```
# Install from package manager and clean its cache
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
           build-essential \
           wget \
    && rm -rf /var/lib/apt/lists/*

# Install from source and remove the code
RUN wget -q http://www.something.org/source-package.tar.gz \
    && tar xf source-package.tar.gz \
    && cd source-package \
    && ./configure \
    && make \
    && make install \
    && cd .. \
    && rm -rf source-package \
    && rm source-package.tar.gz
```

- **Avoid invalidating the build cache:** Each time `docker build` executes a Dockerfile instruction successfully, it caches the resulting image (even if it is an intermediate layer). When carrying out future builds of the Dockerfile, Docker will look for a match of a given instruction in its cache and, if found, it will reuse the cached layer instead of re-building it. Thus, proper use of the build cache can greatly speed up the creation of images.

  Generally, if an instruction changes in a previously built Dockerfile, the lookup will fail and the build cache will be invalidated. When this happens, all subsequent Dockerfile commands will re-build new layers and the cache will not be used.

  Special care should be used with `ADD` and `COPY` instructions: the contents of the files in the images are checksummed and, during cache lookup, the new checksum is compared against the checksum in the cached images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated. This means that even if the Dockerfile is identical, but you changed the files copied by an `ADD` or `COPY` instruction, a full rebuild will happen from that instruction onwards.

Building an "MPI Hello World" Docker image

In this extended example, we will package an "Hello World" MPI program in a Docker image. The program consists of a single C source file, called `hello_mpi.c`:

```
/* hello_mpi.c */

#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Print off a hello world message
    printf("Hello world from rank %d of %d\n",
           rank, size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

It can be compiled with the accompanying Makefile, which uses the MPI compiler wrapper `mpicc`:

```
# Makefile

BIN=hello_mpi
MPICC?=mpicc

all: ${BIN}

hello_mpi: hello_mpi.c
	${MPICC} -o hello_mpi hello_mpi.c

clean:
	rm ${BIN}
```

To create the container image, we need to write a Dockerfile covering the following points:

1. Provide a Linux distribution of choice
2. Install the necessary compilation tools
3. Install an MPI implementation
4. Assuming the "Hello MPI" program sources are available locally, copy them into the container
5. Call `make` to compile the "Hello MPI" program

A possible Dockerfile performing these tasks is provided below. The chosen Linux distribution is Debian 10 and the compilation toolchain is provided by the `build-essential` package. The MPI implementation of choice is MPICH 3.1.4, which is built from source after retrieving the corresponding archive with the `wget` utility.

```
# Dockerfile

FROM debian:buster

RUN apt-get update && apt-get install -y \
        build-essential             \
        wget                        \
        ca-certificates             \
        --no-install-recommends     \
    && rm -rf /var/lib/apt/lists/*


RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz \
    && tar xf mpich-3.1.4.tar.gz \
    && cd mpich-3.1.4 \
    && ./configure --disable-fortran --enable-fast=all,O3 --prefix=/usr \
    && make -j$(nproc) \
    && make install \
    && ldconfig \
    && cd .. \
    && rm -rf mpich-3.1.4 \
    && rm mpich-3.1.4.tar.gz

COPY . /hello_mpi

RUN cd /hello_mpi && make
```

The image is then created with `docker build`:

```
docker build -t my_user/hello_mpi .
```

We can verify the correct functionality of the image by running multiple "Hello MPI" ranks with `mpirun` inside a container:

```
$ docker run --rm my_user/hello_mpi mpirun -n 4 /hello_mpi/hello_mpi

Hello world from rank 0 of 4
Hello world from rank 3 of 4
Hello world from rank 2 of 4
Hello world from rank 1 of 4
```