My Docker Journey

Container

Company / Organization: becke.ch

Scope: 0.0 {language={en};technology--document={ott};organization={becke.ch};interest={business};platform={docker}}

Version: 1.1.0

File-name: becke-ch--docker--s0-v1.odt

Author: docker--s0-v1@becke.ch

Document Version History

Version	Date	Author	Description
1.0.0	20.05.2018	Raoul Becke	Initial documentation of the requirement with version 1.0.0.
1.1.0	13.09.2018	Raoul Becke	Documentation fixed according to requirement version 1.1.0

Module / Artifact / Component / Work-Product Version History

Version	Date	Author	Requirements	Components Changed
1.0.0	20.05.2018	Raoul Becke	Create a docker documentation called: “My Docker Journey” containing: Docker CE versus Docker EE 6, Installation (Ubuntu, Permission), Overview (Daemon, Client, Registry, Objects, Technology, Storage & Union File System, Network), Get Started (Container: Dockerfile, Application, Naming Convention, docker run, docker login, docker push, docker commit)	This document
1.1.0	13.09.2018	Raoul Becke	Reworked chapter "Network" - detailed information on network creation, how to assigne fix IP address to a container and how to remove a network	This document

Table of Contents

1. Introduction....6
1.1. Editions: Docker CE & Docker EE....6
2. Installation....7
2.1. Ubuntu....7
2.1.1. Docker Community Edition (Docker CE)....7
2.2. Permission....9
3. Overview....10
3.1. The Docker daemon....10
3.2. The Docker client....10
3.3. Docker registries....10
3.4. Docker objects....10
3.4.1. Images....11
3.4.2. Containers....11
3.5. The underlying technology....11
3.6. Storage & Union File System....11
3.7. Network (docker network ls|inspect|create|connect|rm)....12
4. Get Started....16
4.1. Container....17
4.1.1. Define a container with Dockerfile....17
4.1.1.1. Dockerfile....17
4.1.2. The app itself....18
4.1.3. Naming convention, building (docker build) & tagging (docker tag) the app....18
4.1.3.1. Remove an image (docker rmi)....20
4.1.4. Run the app (docker run)....20
4.1.5. Create docker account....22
4.1.6. Share the image (docker login & docker push)....22
4.1.7. Dockerfile versus docker commit....22
4.2. Warnings, Errors & Solutions....24
5. Optimizations....24
5.1. Tuning the docker image....24
6. Landscape....27
7. References and glossary....28
7.1. References....28
7.2. Glossary (terms, abbreviations, acronyms)....28
A. Appendix....29
A.1. Appendix A1....29
A.1.1. Appendix A2....29

Illustration Index

Illustration 1: Operating System Layers: Container (Docker) versus VM (Virtual Machine)....6
Illustration 2: Docker Editions: Docker CE versus Docker EE: Feature Comparison....7
Illustration 3: Client-Server Architecture: Client, Docker Host & Registry....10
Illustration 4: Storage & Union File System: Image Layers (R/O) & Container Layer (R/W)....12
Illustration 5: Network diagram....13
Illustration 6: Access the python test application in browser....21

Index of Tables

Table 1: References....28
Table 2: Glossary....28

1. Introduction

https://docs.docker.com/get-started/#docker-concepts

A **container** is launched by running an **image**. An image is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a **runtime instance of an image**--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in Linux.

A **container** runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

By contrast, a **virtual machine (VM)** runs a full-blown “guest” operating system with virtual access to host resources through a **hypervisor**. In general, VMs provide an environment with more resources than most applications need.

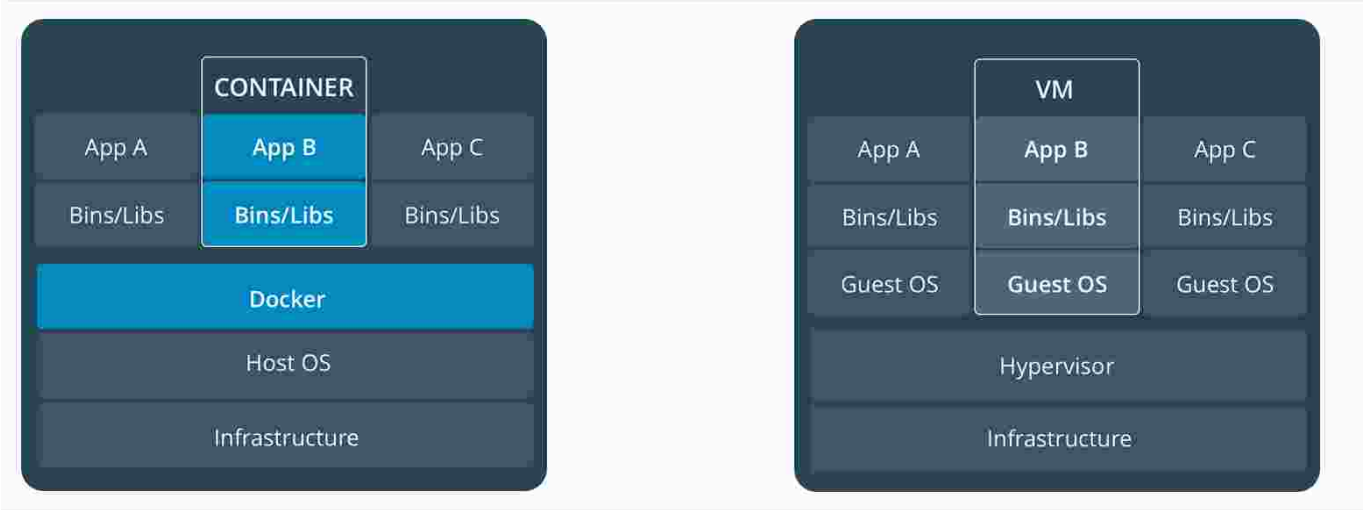

Illustration 1: Operating System Layers: Container (Docker) versus VM (Virtual Machine)

1.1. Editions: Docker CE & Docker EE

Docker CE or Docker EE: Docker is available in two editions: Community Edition (CE) and Enterprise Edition (EE).

Docker Community Edition (CE) is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker CE has two update channels, stable and edge:

- Stable gives you reliable updates every quarter
- Edge gives you new features every month

Capabilities	Community Edition	Enterprise Edition Basic	Enterprise Edition Standard	Enterprise Edition Advanced
Container engine and built in orchestration, networking, security	✓	✓	✓	✓
Certified infrastructure, plugins and ISV containers		✓	✓	✓
Image management			✓	✓
Container app management			✓	✓
Image security scanning				✓

Illustration 2: Docker Editions: Docker CE versus Docker EE: Feature Comparison

2. Installation

2.1. Ubuntu

https://docs.docker.com/install/linux/ubuntu/

The instructions for installing Docker on Ubuntu depend on whether you are using **Docker Enterprise Edition** (**Docker EE**) or **Docker Community Edition** (**Docker CE**).

2.1.1. Docker Community Edition (Docker CE)

https://docs.docker.com/install/linux/docker-ce/ubuntu/

1. **Uninstall old versions** (if you get “not installed” warnings then you can just ignore them):

```
$ sudo apt-get remove docker docker-engine docker.io
root@hp-elitebook-850-g5--s0-v1:~# sudo apt-get remove docker docker-engine docker.io
...
Package 'docker-engine' is not installed, so not removed
Package 'docker' is not installed, so not removed
Package 'docker.io' is not installed, so not removed

...
```

Supported storage drivers: For new installations on version 4 and higher of the Linux kernel (Ubuntu Xenial 16.04 and newer), overlay2 is supported and preferred over aufs.

2. **Install using the repository** (this is the preferred installation method)
   a. Update the apt package index:

```
$ sudo apt-get update
```

   b. Install packages to allow apt to use a repository over HTTPS:

```
$ sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
root@hp-elitebook-850-g5--s0-v1:~# sudo apt-get install apt-transport-https ca-certificates curl
software-properties-common
Reading package lists... Done
Building dependency tree
Reading state information... Done
ca-certificates is already the newest version (20180409).
curl is already the newest version (7.58.0-2ubuntu3).
curl set to manually installed.
software-properties-common is already the newest version (0.96.24.32.1).
…
The following NEW packages will be installed:
  apt-transport-https
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 1'692 B of archives.
After this operation, 152 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

…

c. Add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

d. Verify that you now have the key with the **fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88**, by searching for the last 8 characters of the fingerprint.

```
$ sudo apt-key fingerprint 0EBFCD88
root@hp-elitebook-850-g5--s0-v1:~# sudo apt-key fingerprint 0EBFCD88
pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

e. Use the following command to set up the stable repository

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
root@hp-elitebook-850-g5--s0-v1:~# sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
Hit:1 http://ch.archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:3 http://ch.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 http://ch.archive.ubuntu.com/ubuntu bionic-backports InRelease
Get:5 https://download.docker.com/linux/ubuntu bionic InRelease [64.4 kB]
Fetched 64.4 kB in 1s (83.8 kB/s)
Reading package lists... Done
```

3. Install Docker CE

a. Update the apt package index.

```
$ sudo apt-get update
```

b. Install the latest version of Docker CE

```
$ sudo apt-get install docker-ce
root@hp-elitebook-850-g5--s0-v1:~# sudo apt-get install docker-ce
Reading package lists... Done
Building dependency tree
Reading state information... Done
Package docker-ce is not available, but is referred to by another package.
This may mean that the package is missing, has been obsoleted, or
is only available from another source

E: Package 'docker-ce' has no installation candidate
```

ERROR: Ubuntu 18.04 bionic: `Package docker-ce is not available, but is referred to by another package.`

Solution: For Ubuntu 18.04 bionic: https://unix.stackexchange.com/questions/363048/unable-to-locate-package-docker-ce-on-a-64bit-ubuntu

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic test"
$ sudo apt-get update
$ sudo apt-get install docker-ce
root@hp-elitebook-850-g5--s0-v1:~# sudo apt-get install docker-ce
...
The following additional packages will be installed:
  aufs-tools cgroupfs-mount pigz
The following NEW packages will be installed:
  aufs-tools cgroupfs-mount docker-ce pigz
...
Need to get 34.1 MB of archives.
After this operation, 182 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
…
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /lib/systemd/system/docker.service.
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket → /lib/systemd/system/docker.socket.
Processing triggers for ureadahead (0.100.0-20) ...
ureadahead will be reprofiled on next reboot
Setting up cgroupfs-mount (1.4) ...
Processing triggers for libc-bin (2.27-3ubuntu1) ...
Processing triggers for systemd (237-3ubuntu10) ...
Processing triggers for man-db (2.8.3-2) ...
Setting up pigz (2.4-1) ...
Processing triggers for ureadahead (0.100.0-20) ...
```


2.2. Permission

The user needs to be in the docker group to access the docker commands!

Add user to docker group: **`sudo usermod –a –G docker $USER`**

```
root@hp-elitebook-850-g5--s0-v1:~# sudo usermod -a -G docker raoul-becke--s0-v1
```

AND log-out and log-in again from Ubuntu!

3. Overview

https://docs.docker.com/engine/docker-overview/

Docker uses a **client-server architecture**. The **Docker client** talks to the **Docker daemon**, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a **REST API, over UNIX sockets or a network interface**.

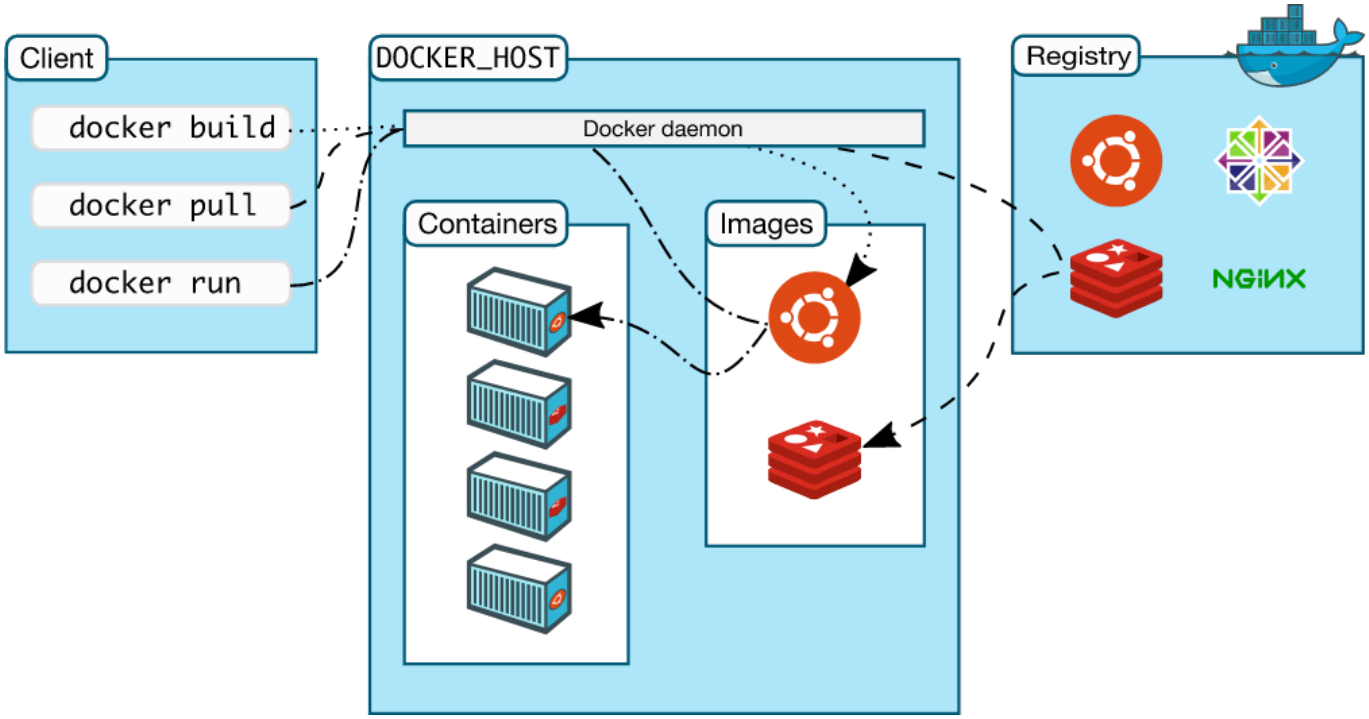

Illustration 3: Client-Server Architecture: Client, Docker Host & Registry

3.1. The Docker daemon

The Docker daemon (**`dockerd`**) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

3.2. The Docker client

The Docker client (**`docker`**) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to `dockerd`, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

3.3. Docker registries

A Docker registry stores Docker images. **Docker Hub** and **Docker Cloud** are **public registries** that anyone can use, and Docker is configured to look for images on **Docker Hub by default**. You can even run your own **private registry**. If you use **Docker Datacenter (DDC)**, it includes **Docker Trusted Registry (DTR)**.

When you use the **`docker pull`** or **`docker run`** commands, the required images are pulled from your configured registry. When you use the **`docker push`** command, your image is pushed to your configured registry.

Docker store allows you to buy and sell Docker images or distribute them for free.

3.4. Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

3.4.1. Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.

To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt.

3.4.2. Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

```
$ docker run -i -t ubuntu /bin/bash
```

1. If you do not have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run `docker pull ubuntu` manually.
2. Docker creates a new container, as though you had run a `docker container create` command manually.
3. Docker **allocates a read-write filesystem to the container**, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
4. Docker creates a **network interface** to connect the container to the **default network**, since you did not specify any networking options. This includes **assigning an IP address to the container**. By default, containers can connect to external networks using the host machine's network connection.
5. Docker starts the container and executes `/bin/bash`. Because the container is run **interactively** and attached to your terminal (due to the `-i` **and** `-t` **flags**), you can provide input using your keyboard and output is logged to your terminal.
6. When you **type** `exit` to terminate the /bin/bash command, the **container stops** but is not removed. You can start it again or remove it.

3.5. The underlying technology

https://docs.docker.com/engine/docker-overview/#the-underlying-technology

- Namespaces
- Control groups
- Union file systems
- Container format

3.6. Storage & Union File System

https://docs.docker.com/storage/storagedriver/

https://washraf.gitbooks.io/the-docker-ecosystem/content/Chapter%201/Section%203/union_file_system.html

Union file system (AUFS or overlayfs) represents file system by grouping directories and files in branches. A **Docker image** is made up of filesystems layered over each other and grouped together. At the base is a **boot filesystem, bootfs**, which resembles the typical Linux/Unix boot filesystem. Each **layer** represents an **instruction** in the image's **Dockerfile**. Each **layer** except the very last one is **read-only**. Consider the following Dockerfile:

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

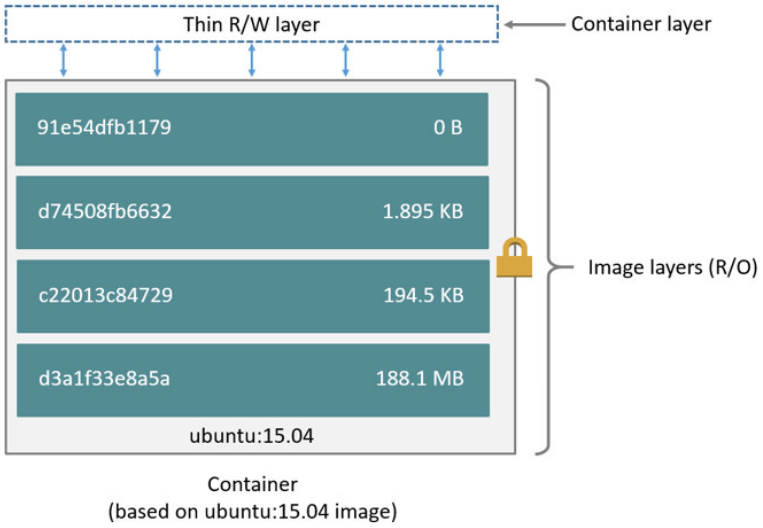

Illustration 4: Storage & Union File System: Image Layers (R/O) & Container Layer (R/W)

This Dockerfile contains four commands, each of which creates a layer. The FROM statement starts out by creating a layer from the ubuntu:15.04 image. The COPY command adds some files from your Docker client's current directory. The RUN command builds your application using the make command. Finally, the last layer specifies what command to run within the container.

Each **layer** is only a set of **differences** from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "**container layer**". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

3.7. Network (`docker network ls|inspect|create|connect|rm`)

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- **bridge**: The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
- **host**: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. host is only available for swarm services on Docker 17.06 and higher.
- **overlay**: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers.
- **macvlan**: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.
- **none**: For this container, disable all networking.

After the installation the **default bridge** called "**docker0**" is installed:

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ ip addr show
...
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:bf:e3:15:db brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:bfff:fee3:15db/64 scope link
       valid_lft forever preferred_lft forever
...
```

`docker network ls`: List networks. The network named bridge is a special network. Unless you tell it otherwise, Docker always launches your containers in this network.


```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d3674ca2832a        bridge              bridge              local
0c1e90550cb3        host                host                local
062c048272a7        none                null                local
```

`docker network inspect bridge`: Inspecting the network is an easy way to find out the container's IP address.

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker run -d -p 4000:80 becke-ch–python-test--s0-v1
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ ip addr show
...
6: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:bf:e3:15:db brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:bfff:fee3:15db/64 scope link
       valid_lft forever preferred_lft forever
20: veth07fa6a1@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
group default
    link/ether 22:d2:00:2b:34:e8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::20d2:ff:fe2b:34e8/64 scope link
       valid_lft forever preferred_lft forever
...
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker network inspect bridge
[
...
        "ConfigOnly": false,
        "Containers": {
            "b5d495840c254647f303f89e0fea2b0362fff5afc7119b93aa0402b9a8fcf19c": {
                "Name": "nostalgic_lewin",
                "EndpointID": "5cf0d0e1ca354f7ad476da036d3fd2128073a4859438043a51b32f91fbda5804",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
...
]
```

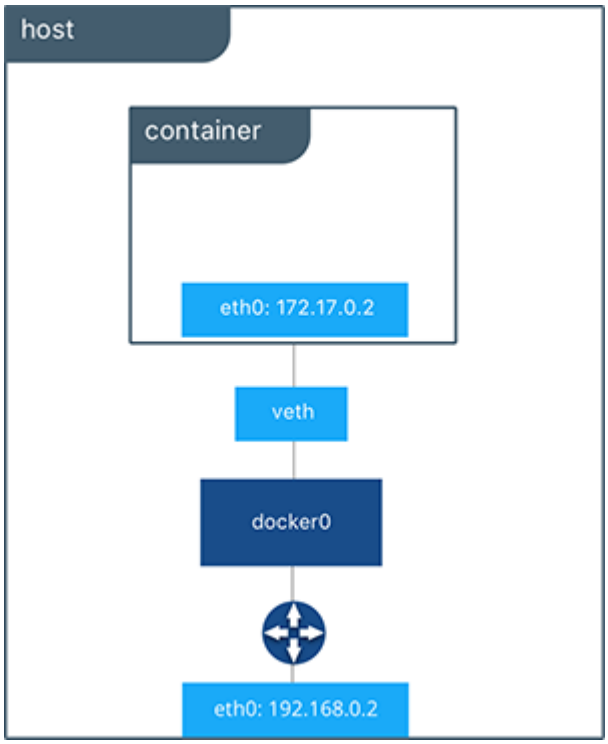

Illustration 5: Network diagram

`docker network create -d bridge --subnet=10.0.0.0/16 --ip-range=10.0.0.0/24 -o com.docker.network.bridge.name=docker--s0-v1 docker--s0-0-v1-0` : Creates a new network. The `DRIVER` accepts `bridge` or `overlay` which are the built-in network drivers. If you have installed a third party or

your own custom network driver you can specify that `DRIVER` here also. If you don't specify the `--driver` option, the command automatically creates a bridge network for you. When you install Docker Engine it creates a bridge network automatically. This network corresponds to the `docker0` bridge that Engine has traditionally relied on. When you launch a new container with docker run it automatically connects to this bridge network. You cannot remove this default bridge network, but you can create new ones using the `network create` command. Besides creating a docker network, in the background the command "`brctl addbr docker--s0-v1;brctl setfd br0 0;ifconfig docker--s0-v1 10.0.0.0 netmask 255.255.0.0`" is executed and a new bridge is created or an existing bridge is attached and configured!

- `-d bridge`: Tells Docker to use the bridge driver for the new network. You could have left this flag off as bridge is the default value for this flag.
- `--subnet=10.0.0.0/16`: Subnet in CIDR format that represents a network segment. Maps to the `address` (`10.0.0.0`) and `netmask` (`255.255.0.0`) parameter in the `ifconfig` command.
- `--ip-range=10.0.0.0/24`: Allocate container ip from a sub-range, starting from `10.0.0.0` up to `10.0.0.255` (in this example) giving docker an ip range of 255 addresses. This does not map to a parameter in the `ifconfig` command but is only used docker internally and the mechanism is comparable to a DHCP server leasing IP addresses to its containers.
- `-o com.docker.network.bridge.name=docker--s0-v1`: Bridge name to be used when creating the Linux bridge. If a bridge interface with the same name already exists then this existing bridge is used instead of creating a new bridge interface.

Normally I use the private IP address range `192.168.0.0` up to `192.168.255.255` but due to, my own defined range `/16` and `/24`, behavior of "`--subnet=.../16`" and within the sub-net the "`--ip-range=.../24`" the lower 2 bytes are already used and therefore I use the private IP address range "`10.0.0.0/8`" which leaves me `10.0` up to `10.255` a total of 255 sub-nets I can use for bridge interfaces. Within "`--subnet=10.0.0.0/16`" respective the corresponding bridge interface "`docker-s0-v1`" I can create 255 docker bridges "`--ip-range=10.0.0.0/24`" up to "`--ip-range=10.0.255.0/24`" with the name "`docker-s0-0-v1-0`" up to "`docker-s0-255-v1-0`".

Important: Precondition: To avoid that docker creates a bridge device with a random name the bridge device should be created and persisted upfront as follows (the following applies to Ubuntu) (**Attention** the max length of an interface name is 15 characters!). The name and configuration here needs to match the name and configuration in docker otherwise it will get overwritten:

```
sudo vi /etc/network/interfaces
…
#Comment the following line(s) if you don't want to bring the bridge interfaces up automatically
auto docker--s0-v1
…
iface docker--s0-v1 inet static
 address 10.0.0.1
 netmask 255.255.0.0
 bridge_ports none
 bridge_stp off
 bridge_fd 0
 bridge_maxwait 0
…
```

And started as follows (unless the bridge is started automatically using `auto`):

```
sudo ifup docker--s0-v1
```

Which give us the following result: `ip addr show`:

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ ip addr show
...
4: docker--s0-v1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
qlen 1000
    link/ether de:ca:8c:7a:7d:ed brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/16 brd 10.0.255.255 scope global docker--s0-v1
       valid_lft forever preferred_lft forever
    inet6 fe80::dcca:8cff:fe7a:7ded/64 scope link
       valid_lft forever preferred_lft forever
...
```

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker network create -d bridge --subnet=10.0.0.0/16 --
ip-range=10.0.0.0/24 -o com.docker.network.bridge.name=docker--s0-v1 docker--s0-0-v1-0
5ff4ef6ea4c4aeb1be0b374328c7ed46606cb5c65493fda6f96906df199c8ea4
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
7e0f1baa9f15        bridge              bridge              local
5ff4ef6ea4c4        docker--s0-0-v1-0   bridge              local
0c1e90550cb3        host                host                local
062c048272a7        none                null                local
```

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker network inspect docker--s0-0-v1-0
[
    {
        "Name": "docker--s0-0-v1-0",
        "Id": "5ff4ef6ea4c4aeb1be0b374328c7ed46606cb5c65493fda6f96906df199c8ea4",
        "Created": "2018-09-13T07:10:36.522190304+02:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "10.0.0.0/16",
                    "IPRange": "10.0.0.0/24"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.name": "docker--s0-v1"
        },
        "Labels": {}
    }
]
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ ip addr show
...
4: docker--s0-v1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/16 brd 10.0.255.255 scope global docker--s0-v1
       valid_lft forever preferred_lft forever
    inet6 fe80::dcca:8cff:fe7a:7ded/64 scope link
       valid_lft forever preferred_lft forever
...
```

`docker run -d --net bridge docker--s0-0-v1-0 --name db training/postgres`: You can add containers to a network when you first run a container.

`docker run -d --net docker--s0-0-v1-0 --ip 10.0.0.10 --name db training/postgres`: Runs the container on a fix IP address: `10.0.0.10`.

`docker network connect docker--s0-0-v1-0 <solution-name>:s0[-0[-yyyymmdd]]-v1[-0[-yyyymmdd]]`: Docker networking allows you to attach a container to as many networks as you like. You can also attach an already running container.

`docker network rm docker--s0-0-v1-0`: Removes one or more networks by name or identifier. To remove a network, you must first disconnect any containers connected to it.

4. Get Started

https://docs.docker.com/get-started/

1. Test Docker version

```
$ docker --version
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker --version
Docker version 18.05.0-ce-rc1, build 33f00ce

$ docker info
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 18.05.0-ce-rc1
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 773c489c9c1b21a6d78b5c538cd395416ec50f88
runc version: 4fc53a81fb7c994640722ac585fa9ca548971871
init version: 949e6fa
Security Options:
 apparmor
 seccomp
  Profile: default
Kernel Version: 4.15.0-20-generic
Operating System: Ubuntu 18.04 LTS
OSType: linux
Architecture: x86_64
CPUs: 8
Total Memory: 15.48GiB
Name: hp-elitebook-850-g5--s0-v1.becke.ch
ID: 4XSK:F4XI:X7RN:JJAO:DL6I:XTCV:FYLW:5DJW:EB3L:IZAH:KHVP:CMBP
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false

WARNING: No swap limit support
```

2. Test Docker installation

```
docker run hello-world
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9bb5a5d4561a: Pull complete
Digest: sha256:f5233545e43561214ca4891fd1157e1c3c563316ed8e237750d59bde73361e77
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/engine/userguide/
```

3. List the hello-world image that was downloaded to your machine:

docker image ls

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              e38bc07ac18e        2 weeks ago         1.85kB
```

4. List the hello-world container (spawned by the image) which exits after displaying its message. If it were still running, you would not need the **--all option**:

docker container ls --all

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker container ls --all
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                     PORTS               NAMES
506e24fa2e00        hello-world         "/hello"            5 minutes ago       Exited (0) 5 minutes ago                       
quizzical_keldysh
```

4.1. Container

https://docs.docker.com/get-started/part2/

We start at the bottom of the hierarchy of such an app, which is a **container**. Above this level is a **service**, which defines how containers behave in production. Finally, at the top level is the **stack**, defining the interactions of all the services.

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These **portable images** are defined by something called a **Dockerfile**.

4.1.1. Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like **networking interfaces** and **disk drives** is **virtualized** inside this environment, which is isolated from the rest of your system, so you need to **map ports to the outside world**, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

4.1.1.1. Dockerfile

Create an **empty directory**: `/ws/app/becke-ch--docker--s0-v1/dockerfile/`**`becke-ch--python-test--s0-v1`**. Change directories (cd) into the new directory, create a file (Dockerfile) called: **`Dockerfile`**, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80
```

```
# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

This Dockerfile refers to a couple of files we haven't created yet, namely **`app.py`** and **`requirements.txt`**.

4.1.2. The app itself

Create two more files, **`requirements.txt`** and **`app.py`**, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, `app.py` and `requirements.txt` is present because of that **Dockerfile's ADD command**, and the output from app.py is accessible over **HTTP** thanks to the **EXPOSE command**.

`/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s0-v1/`**`requirements.txt`**

```
Flask
Redis
```

`/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s0-v1/`**`app.py`**

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Now we see that **`pip install -r requirements.txt`** installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to socket.gethostname(). Finally, because Redis isn't running (as we've only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

4.1.3. Naming convention, building (`docker build`) & tagging (`docker tag`) the app

https://medium.com/@mccode/the-misunderstood-docker-tag-latest-af3babfd6375

https://stackoverflow.com/questions/41520614/docker-tag-vs-name-clarification

Run the build command. This creates a Docker image, which we're going to **tag using -t** so it has a **friendly and unique name according to naming convention: <hub-user>/<repo-name>[:<tag>]**

FQN: <hub-user>/<repo-name>[:<tag>]: The fully qualified name respective its single parts: "hub-user", "repo-name" and "tag" are nowhere exposed and visible outside the docker ecosystem. Outside the docker ecosystem on the file-system or on the hub only a SHA256 ID is shown and the mapping of the SHA256 to the FQN is only maintained within a docker specific file in "`/var/lib/docker/image/overlay2/repositories.json`" (the subdirectory "overlay2" varies depending on the Union File System technology being used). And therefore there is no need to include the name "docker" in any parts of the fully qualified name! And this results in the following naming convention:

- **`becke-ch--s`**`0`**`-v`**`1`**`/<solution-name>:s`**`0[-0[-yyyymmdd]]`**`-v`**`1[-0[-yyyymmdd]]`
  The major, minor and patch numbers of the solution's scope "sX-Y-Z" respective version "vX-Y-Z" should be

adapted according to the major, minor and patch changes made to the solution! Minor and patch numbers are optional and therefore put into brackets "[]".
Unfortunately even this FQN is allowed according to the docker syntax (including single "-" and double "--" hyphens) (see as well discussion in https://github.com/docker/distribution/issues/1056) there exist docker repositories like for example https://hub.docker.com/ that do not support this and therefore had to "downgrade" the naming convention (docker.com only supports for "hub-user" lowercase letters and number and for "repo-name" only lowercase letters, numbers and single hyphen see discussion on https://github.com/docker/hub-feedback/issues/373):
beckechs0**v**1**/<solution-name>:s**0[-0[-yyyymmdd]]**-v**1[-0[-yyyymmdd]]
For more information on how to create a "hub-user" aka docker-id on https://hub.docker.com see chapter 4.1.5.

For more information on scope- & version- respective naming-convention see documents in [1] respective [2].

Incremental Naming: Building & Tagging: Often we deploy the same docker image into different environments respective hubs. Therefore the "tag" assigned during the build process is performed without the <hub-user>!

- **docker build -t <solution-name>:s**0[-0[-yyyymmdd]]**-v**1[-0[-yyyymmdd]] .
  Respective **docker build -t <solution-name>:s**0[-0[-yyyymmdd]]**-v**1[-0[-yyyymmdd]] .

```
docker build -t python-test:s0-0-v1-0 .
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker build -t python-test:s0-0-v1-0 .
Sending build context to Docker daemon  4.608kB
Step 1/7 : FROM python:2.7-slim
 ---> 829e955d463b
Step 2/7 : WORKDIR /app
Removing intermediate container 02e3dd41d691
 ---> bb566876c11a
Step 3/7 : ADD . /app
 ---> 38cc67a54701
Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt
 ---> Running in e3379032d33e
Collecting Flask (from -r requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dacb4696386634607af824ea997202cd0edb4
b/Flask-1.0.2-py2.py3-none-any.whl (91kB)
Collecting Redis (from -r requirements.txt (line 2))
  Downloading
https://files.pythonhosted.org/packages/3b/f6/7a76333cf0b9251ecf49efff635015171843d9b977e4ffcf59f9c442805
2/redis-2.10.6-py2.py3-none-any.whl (64kB)
Collecting Werkzeug>=0.14 (from Flask->-r requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/20/c4/12e3e56473e52375aa29c4764e70d1b8f3efa6682bef8d0aae04fe33524
3/Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
Collecting click>=5.1 (from Flask->-r requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/34/c1/8806f99713ddb993c5366c362b2f908f18269f8d792aff1abfd700775a7
7/click-6.7-py2.py3-none-any.whl (71kB)
Collecting Jinja2>=2.10 (from Flask->-r requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763ba4d277a78ca76f32659220a73
1/Jinja2-2.10-py2.py3-none-any.whl (126kB)
Collecting itsdangerous>=0.24 (from Flask->-r requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/dc/b4/a60bcdba945c00f6d608d8975131ab3f25b22f2bcfe1dab221165194b2d
4/itsdangerous-0.24.tar.gz (46kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10->Flask->-r requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/4d/de/32d741db316d8fdb7680822dd37001ef7a448255de9699ab4bfcbdf4172
b/MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory:
/root/.cache/pip/wheels/2c/4a/61/5599631c1554768c6290b08c02c72d7317910374ca602ff1e5
  Running setup.py bdist_wheel for MarkupSafe: started
  Running setup.py bdist_wheel for MarkupSafe: finished with status 'done'
  Stored in directory:
/root/.cache/pip/wheels/33/56/20/ebe49a5c612fffe1c5a632146b16596f9e64676768661e4e46
Successfully built itsdangerous MarkupSafe
Installing collected packages: Werkzeug, click, MarkupSafe, Jinja2, itsdangerous, Flask, Redis
Successfully installed Flask-1.0.2 Jinja2-2.10 MarkupSafe-1.0 Redis-2.10.6 Werkzeug-0.14.1 click-6.7
itsdangerous-0.24
Removing intermediate container e3379032d33e
 ---> 361cd2e308d2
Step 5/7 : EXPOSE 80
 ---> Running in d7b5978e6ae0
Removing intermediate container d7b5978e6ae0
```



```
 ---> 47c18f14fefb
Step 6/7 : ENV NAME World
 ---> Running in 77c6a4aabe65
Removing intermediate container 77c6a4aabe65
 ---> 2ca594589014
Step 7/7 : CMD ["python", "app.py"]
 ---> Running in 124133631a3c
Removing intermediate container 124133631a3c
 ---> c68c656d9e24
Successfully built c68c656d9e24
Successfully tagged python-test:s0-0-v1-0
```

Tagging: Once build successfully finished, before the image is published respective shared, the image needs to be tagged:

- **`docker tag <solution-name>:s`**`0[-0[-yyyymmdd]]`**`-v`**`1[-0[-yyyymmdd]]` **`becke-ch--s`**`0`**`-v`**`1`**`/<solution-name>:s`**`0[-0[-yyyymmdd]]`**`-v`**`1[-0[-yyyymmdd]]`
  Respective **`docker tag <solution-name>:s`**`0[-0[-yyyymmdd]]`**`-v`**`1[-0[-yyyymmdd]]` **`beckechs`**`0`**`v`**`1`**`/<solution-name>:s`**`0[-0[-yyyymmdd]]`**`-v`**`1[-0[-yyyymmdd]]`

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker tag python-test:s0-0-v1-0 beckechs0v1/python-test:s0-0-v1-0
```

Once the image has been tagged it can be shared as described in chapter 4.1.6.

Where is your built image? It's in your machine's local Docker image registry:

```
$ docker image ls
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker image ls
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
beckechs0v1/python-test   s0-0-v1-0           c68c656d9e24        About an hour ago   156MB
python-test               s0-0-v1-0           c68c656d9e24        About an hour ago   156MB
...
```

4.1.3.1. Remove an image (`docker rmi`)

First run `docker image ls` to retrieve the image-id that should get deleted:

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker image ls
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
becke-ch--python-test--s0-v1    latest              806ad867df6e        4 minutes ago       156MB
python                          2.7-slim            829e955d463b        3 days ago          144MB
hello-world                     latest              e38bc07ac18e        2 weeks ago         1.85kB
```

And then run **`docker rmi image-id`**:

```
docker rmi 806ad867df6e
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker rmi 806ad867df6e
Untagged: becke-ch--python-test--s0-v1:latest
Deleted: sha256:806ad867df6ec561443954ebab6b1c0b65a3e371a44c0a02ed5d8be1baf39f49
Deleted: sha256:89de5beaa32d89e6570f0fbe33abb7ae79b5e298f4759ab85dd2e23bce7771f4
Deleted: sha256:75067f37e1c3cf23209f67a56990a45c203c680bb0d2806710d47c8889af6621
Deleted: sha256:95a3c18473aef91430803bee893a4793eb0bef850eae78c20913cbb220c10bc8
Deleted: sha256:13e46d4b7e2eb1007ff7f1f109d44ffed0dd0c3e2cc047449679e6e191b4cc09
Deleted: sha256:6aba8cee545c48b085a2c89531db593af6dbb7dc3a0c8ee6356739443a97311a
Deleted: sha256:f7030d9279d5f45c47eae70c39431dd999c55bd7dca4db376c538370dadcfa2f
Deleted: sha256:52add4b9cdf5280441f31fd84daf6ab0a963f83a540483eed77aaad163fabfd4
Deleted: sha256:7cb6d18c3b0151410cfb8961925a3f5c12e36b76de5108425ced518f7a5dd538
```

And last but not least run again `docker image ls` to make sure the image has been removed!

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python              2.7-slim            829e955d463b        3 days ago          144MB
hello-world         latest              e38bc07ac18e        2 weeks ago         1.85kB
```

4.1.4. Run the app (`docker run`)

Run the app, **mapping** your **machine's port 4000** to the **container's published port 80** using -p:

```
docker run -p 4000:80 python-test:s0-0-v1-0
(Alternately run the container and give it a name: docker run -p 4000:80 --name python-test:s0-0-0-v1-0-0
python-test:s0-0-v1-0)
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker run -p 4000:80 becke-ch--python-test--s0-v1
 * Serving Flask app "app" (lazy loading)
 * Environment: production
```

```
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
172.17.0.1 - - [30/Apr/2018 19:36:01] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [30/Apr/2018 19:36:02] "GET /favicon.ico HTTP/1.1" 404 -
```

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn't know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page.

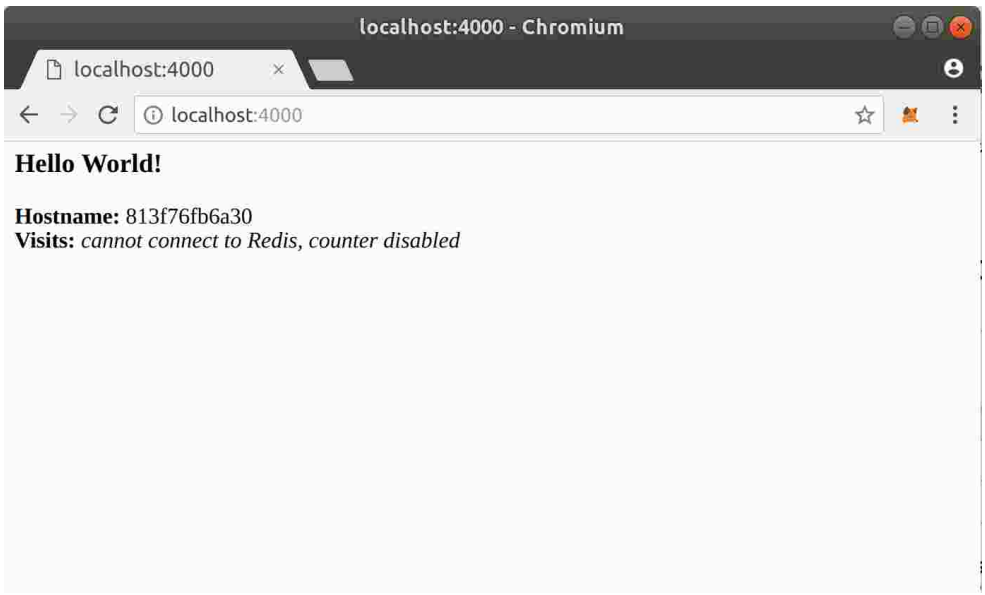

Illustration 6: Access the python test application in browser

This port remapping of 4000:80 is to demonstrate the difference between what you EXPOSE within the Dockerfile, and what you publish using `docker run -p`. In later steps, we just map port 80 on the host to port 80 in the container and use http://localhost .

Hit `CTRL+C` in your terminal to quit.

Now let's run the app in the background, in detached mode:

`docker run -d -p 4000:80 python-test:s0-0-v1-0`

(Alternately run the container and give it a name: `docker run -d -p 4000:80 --name python-test:s0-0-0-v1-0-0 python-test:s0-0-v1-0`)

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker run -d -p 4000:80 python-test:s0-0-v1-0
13fac6c041cef82678bc30ee84ba709ea739682e0945e822def0cab02aa16b3f
```

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls (and both work interchangeably when running commands):

`docker container ls`

raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s0-v1$ docker container ls

```
CONTAINER ID        IMAGE                       COMMAND             CREATED             STATUS              PORTS                   NAMES
13fac6c041ce        python-test:s0-0-v1-0       "python app.py"     4 minutes ago       Up 4 minutes        0.0.0.0:4000->80/tcp
cocky_albattani
```

Notice that CONTAINER ID matches what's on http://localhost:4000.

Connecting to a running container can be achieved with the command "**`docker exec -i -t COTAINER-ID /bin/bash`**" which executes a command in a running container but with the options "`-i`" for interactive and "`-t`" for pseudo-TTY console opens directly an interactive console running the "`/bin/bash`" command respective shell:

`docker exec -i -t 13fac6c041ce /bin/bash`

Now use docker container stop to end the process, using the CONTAINER ID, like so:

`docker container stop 13fac6c041ce`
`raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--`


```
python-test--s0-v1$ docker container stop 13fac6c041ce
13fac6c041ce
```

4.1.5. Create docker account

https://docs.docker.com/docker-id/#/register-for-a-docker-id

Creating a docker account is optional but required when uploading and publishing an image. Regarding naming convention first consult chapter 4.1.3!

1. Go to: https://cloud.docker.com/ (attention consider restrictions in https://success.docker.com/article/how-do-you-register-for-a-docker-id)
2. Click on “Signup” and enter your data
   a. **Docker ID**: **dockers0v1**
      ATTENTION: Your Docker ID must be between 4 and 30 characters long, and can only contain numbers and lowercase letters e.g. the following is not possible “docker--s0-v1” !
   b. **Email**: **docker--s0-v1@becke.ch**
   c. **Password**: eo…

4.1.6. Share the image (`docker login` & `docker push`)

Precondition is that you’ve built and tagged the image according to chapter 4.1.3 and that you’ve created a docker account as described in previous chapter:

1. **Login** to the docker repository: **`docker login`**
   Alternately you can login as follows: `docker login --username=dockers0v1`

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ 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: dockers0v1
Password:
WARNING! Your password will be stored unencrypted in /home/raoul-becke--s0-v1/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

2. **Publish the image**: Upload your tagged image to the repository: **`docker push beckechs0v1/`**`python-test`**`:s`**`0-0`**`-v`**`1-0`

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker push beckechs0v1/python-test:s0-0-v1-0
The push refers to repository [docker.io/beckechs0v1/python-test]
3defcd5015c9: Pushed
acaeb4462dc8: Pushed
ee863d037687: Pushed
c940e8897d87: Pushed
d18e1264e20f: Pushed
89d43612fdd9: Pushed
43efe85a991c: Pushed

s0-0-v1-0: digest: sha256:5716b7231fa1afc77323dc875fed0ea6f0da838d401cb376a571ef881e4e3560 size: 1788
```

4.1.7. Dockerfile versus docker commit

https://jaxenter.de/10-wege-docker-images-zu-bauen-1-61421

https://hackernoon.com/to-commit-or-not-to-commit-5ab72f9a466e

Instead of using a dockerfile and building an image; one can alternatively start the container, run the installation- & setup-steps and finally commit the container to get a docker image as follows (applied to the example above):

1. Create an empty directory: `/ws/app/becke-ch--docker--s0-v1/dockerfile/`**`becke-ch--python-test--s1-v1`** and change (cd) into the new directory:

```
mkdir -p /ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s1-v1
cd /ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s1-v1
```

2. Create the python file "`/ws/app/becke-ch--docker--s0-v1/dockerfile/`**`becke-ch--python-test--s1-v1/app.py`**" - content of the file see previous chapter 4.1.2.
3. Run the python image version "2.7-slim" in interactive mode, mapping the port 4000 to the port 80. (running an image triggers the following commands in the background: If you do not have the python image locally "`docker run -i -t python:2.7-slim`", creates a new container "`docker container create`", Docker allocates a read-write filesystem to the container, Docker creates a network interface to connect the container to the default network bridge "`docker0`"):

```
docker run -w /app -v /ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s1-v1:/app -p 4000:80 -i -t python:2.7-slim bash
```

   a. "`-w /working-directory`": Working directory inside the container
   b. "`-v /host-directory-absolute-path:/docker-directory`": Bind mount a volume. IMPORTANT the path in the host directory needs to be absolute and not relative!
   c. "`-i`": Keep STDIN open even if not attached
   d. "`-t`": Allocate a pseudo-TTY

4. Install the packages "`Flask`" and "`Redis`" running the pip command:

```
root@c51d7f604259:/# pip install --trusted-host pypi.python.org Flask
Collecting Flask
  Downloading
https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dacb4696386634607af824ea997202cd0edb4b/Flask-1.0.2-py2.py3-none-any.whl (91kB)
    100% |████████████████████████████████| 92kB 873kB/s
Collecting Werkzeug>=0.14 (from Flask)
  Downloading
https://files.pythonhosted.org/packages/20/c4/12e3e56473e52375aa29c4764e70d1b8f3efa6682bef8d0aae04fe335243/Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
    100% |████████████████████████████████| 327kB 1.6MB/s
Collecting click>=5.1 (from Flask)
  Downloading
https://files.pythonhosted.org/packages/34/c1/8806f99713ddb993c5366c362b2f908f18269f8d792aff1abfd700775a77/click-6.7-py2.py3-none-any.whl (71kB)
    100% |████████████████████████████████| 71kB 5.5MB/s
Collecting Jinja2>=2.10 (from Flask)
  Downloading
https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763ba4d277a78ca76f32659220a731/Jinja2-2.10-py2.py3-none-any.whl (126kB)
    100% |████████████████████████████████| 133kB 8.9MB/s
Collecting itsdangerous>=0.24 (from Flask)
  Downloading
https://files.pythonhosted.org/packages/dc/b4/a60bcdba945c00f6d608d8975131ab3f25b22f2bcfe1dab221165194b2d4/itsdangerous-0.24.tar.gz (46kB)
    100% |████████████████████████████████| 51kB 2.9MB/s
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10->Flask)
  Downloading
https://files.pythonhosted.org/packages/4d/de/32d741db316d8fdb7680822dd37001ef7a448255de9699ab4bfcbdf4172b/MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous ... done
  Stored in directory:
/root/.cache/pip/wheels/2c/4a/61/5599631c1554768c6290b08c02c72d7317910374ca602ff1e5
  Running setup.py bdist_wheel for MarkupSafe ... done
  Stored in directory:
/root/.cache/pip/wheels/33/56/20/ebe49a5c612fffe1c5a632146b16596f9e64676768661e4e46
Successfully built itsdangerous MarkupSafe
Installing collected packages: Werkzeug, click, MarkupSafe, Jinja2, itsdangerous, Flask
Successfully installed Flask-1.0.2 Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1 click-6.7 itsdangerous-0.24

root@c51d7f604259:/# pip install --trusted-host pypi.python.org Redis
Collecting Redis
  Downloading
https://files.pythonhosted.org/packages/3b/f6/7a76333cf0b9251ecf49efff635015171843d9b977e4ffcf59f9c4428052/redis-2.10.6-py2.py3-none-any.whl (64kB)
    100% |████████████████████████████████| 71kB 668kB/s
Installing collected packages: Redis
Successfully installed Redis-2.10.6
```

5. You can now exit the container and commit the container to an image. To get the ID of the container you want to commit you need to run "**`docker ps -a`**", the "`-a`" option is necessary because "`docker ps`" only returns the currently actively running containers! In case you forgot something you can restart again the inactive container running "**`docker start putHereTheContainerID`**" & "**`docker attach putHereTheContainerID`**". It is not necessary but suggested that you already have a docker account registered see chapter 4.1.5.

```
root@70d1104421db:/app# exit
$ docker ps -a
CONTAINER ID        IMAGE                     COMMAND                  CREATED             STATUS                      PORTS
```

```
NAMES
70d1104421db        python:2.7-slim             "bash"                  10 hours ago        Exited (0) 4 seconds ago
vibrant_ptolemy
…
$ docker commit -a "Raoul Becke <docker--s0-v1@becke.ch>" -m "finalized python installation" 70d1104421db
dockers0v1/python-test:s1-0-v1-0
```

6. And last but not least you can start the new image/container again running:

```
$ docker run -i -t dockers0v1/python-test:s1-0-v1-0
```

BUT The general recommendation is not to use docker commit because:

- **Transparency & Reproduce-ability**: The installation & setup steps that are executed within the container are therefore not transparent and not reproduce-able.
- **Layering: Increment and rollback:** The docker file-system is layered i.e. every instruction in the dockerfile creates a new read-only file-system layer containing only the file-differences compared to the layer below. Therefore using a dockerfile; the installation- & setup-commands are incremental and can be rolled back respective based on a lower layer image and accordingly can be replaced easily with up-to-date installation- & setup-commands.
- **Documentation**: The dockerfile can be seen as kind of installation and setup documentation describing the single steps to get a running application.

But there exist some exceptions

- Experiment: You are not sure whether the installation commands are correct and therefore you need to experiment. But as well and especially in this case do not commit this experimental and sub-optimal image!

4.2. Warnings, Errors & Solutions

ERROR: Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.37/info: dial unix /var/run/docker.sock: connect: permission denied

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:~$ docker info
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock:
Get http://%2Fvar%2Frun%2Fdocker.sock/v1.37/info: dial unix /var/run/docker.sock: connect: permission
denied
```

SOLUTION A: https://askubuntu.com/questions/941816/permission-denied-when-running-docker-after-installing-it-as-a-snap

Add user to docker group: **`sudo usermod -a -G docker $USER`**

```
root@hp-elitebook-850-g5--s0-v1:~# sudo usermod -a -G docker raoul-becke--s0-v1
```

AND log-out and log-in again from Ubuntu!

SOLUTION B: Running as root: **`sudo docker info`**

```
root@hp-elitebook-850-g5--s0-v1:~# sudo docker info
```

ERROR: unable to prepare context: unable to evaluate symlinks in Dockerfile path: lstat /ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--python-test--s0-v1/Dockerfile: no such file or directory

```
raoul-becke--s0-v1@hp-elitebook-850-g5--s0-v1:/ws/app/becke-ch--docker--s0-v1/dockerfile/becke-ch--
python-test--s0-v1$ docker build -t becke-ch--python-test--s0-v1 .
unable to prepare context: unable to evaluate symlinks in Dockerfile path: lstat /ws/app/becke-ch--
docker--s0-v1/dockerfile/becke-ch--python-test--s0-v1/Dockerfile: no such file or directory
```

SOLUTION: **`docker build -t becke-ch--python-test--s0-v1 . --file becke-ch--python-test--s0-v1--dockerfile`**

5. Optimizations

5.1. Tuning the docker image

According to chapter 3.6 “*Each layer represents an instruction in the image’s Dockerfile*” and therefore the aim is to have as little as possible instructions in the dockerfile to save space but as many as necessary to still maintain a clear layering and modularisation to build upon.

https://jaxenter.de/10-wege-docker-images-zu-bauen-1-61421

```
FROM centos:centos7
RUN yum install epel-release -y && \
    yum update -y && \
    yum install redis -y && \
    yum clean all
EXPOSE 6379
ENTRYPOINT ["/usr/bin/redis-server"]
```

Hier wird von einem Centos-7-Basis-Image gestartet. Im Anschluss wird ein Redis-Server über yum installiert und der Port 6379 als exportierbar deklariert. Schließlich wird als Entrypoint der entsprechende Befehl eingerichtet. In diesem Beispiel werden on-top zu dem Basis-Image drei weitere Layer erzeugt. Man beachte, dass in der RUN-Zeile (die aus drei Textzeilen besteht, die mit \ zusammengefügt sind) mehrere Kommandos mit & & verknüpft werden. Dieser altbekannte Trick wird benutzt, um die Anzahl der erzeugten Schichten und damit den Platzverbrauch niedrig zu halten. Leider gibt es keine direkte Möglichkeit, Einfluss auf die Gestaltung der Imageschichten zu nehmen. Schön wäre es hier, eine Möglichkeit der Klammerung zu haben, um z.B. alle Kommandos innerhalb eines BEGIN .... COMMIT Blockes zu einer einzigen Schicht zusammenführen zu können.

Eine weitere Neuerung sind Multi-Stage Builds, die es seit Docker 17.06 gibt. Dabei können mehrere FROM-Direktiven in einem Dockerfile verwendet werden. Interessant sind Multi-Stage Builds vor allem, wenn man das Bauen der Applikation selbst mit der Image-Erstellung kombinieren möchte, sodass ein docker build auch gleich die Applikation selbst kompiliert.

Das Ganze lässt sich am besten in einem Beispiel verdeutlichen. In dem folgenden Dockerfile wird eine Go-Anwendung kompiliert und daraus ein Image gebaut:

```
1
2
3
4
5
6
7
8
```

```
FROM golang AS builder
WORKDIR /tmp
COPY app.go .
RUN go build app.go

FROM scratch
COPY --from=builder /tmp/app .
CMD ["./app"]
```

In der ersten FROM-Anweisung wird ein Image ausgewählt, dass einen Go Compiler enthält. Der Zusatz AS builder gibt diesem Image einen Namen, den wir weiter unten referenzieren. Nun wird die Anwendung kompiliert (RUN go build app.go), was in ein statisch gelinktes Programm /tmp/app mündet.

Das eigentliche Image wird nach dem letztem FROM erzeugt. Hier ist es das FROM scratch, was ein „nacktes“ Docker Image auswählt. Da unsere Anwendung komplett statisch kompiliert ist, benötigt es auch kein Betriebsystem als Basis-Image. Der Clou hier ist nun die COPY-Anweisung. Dabei wird über das --from=builder-

Argument auf das vorherige Image referenziert und auf dessen Dateisystem zugegriffen. Somit enthält das finale Image nur das fertig kompilierte Programm, nicht jedoch die gesamte Go Toolchain aus dem golang-Image.

Diese Technik erlaubt es, sehr schlanke Images zu erzeugen und bei kompilierenden Programmiersprachen (Go, Java, C, C++, …) wird nicht die Toolchain inklusive Compiler und Buildsystem in das endgültige Image gepackt.

6. Landscape

7. References and glossary

7.1. References

Reference	Location	Remarks
[1]	http://becke.ch/data/becke-ch--scope-and-version-convention--s0-v1/document/	**Scope & Version**: Describes the scope and version convention which is basis for the naming convention.
[2]	http://becke.ch/data/becke-ch--naming-convention--s0-v1/document/	**Naming Convention**: Describes the naming convention that should be used.

Table 1: References

7.2. Glossary (terms, abbreviations, acronyms)

Terms / Abbreviations / Acronyms	Description

Table 2: Glossary

A. Appendix

A.1. Appendix A1

A.1.1. Appendix A2