



formatting_adjusted

Bolzern Iwan authored just now

ae176f5d

 **hands-on.md** 25.2 KB

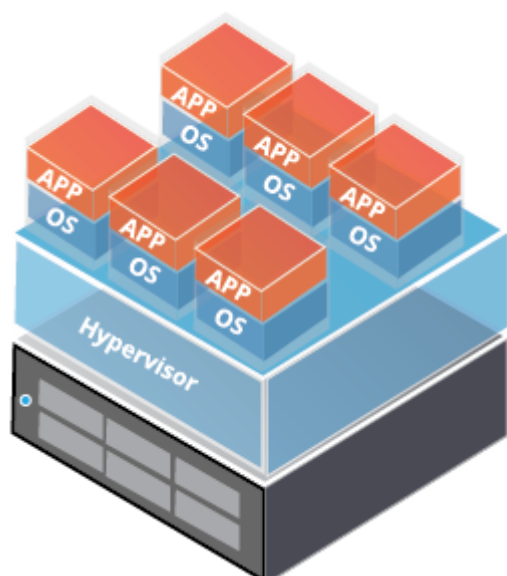
Docker hands on workshop

Table of Contents

1. [Definition virtualization](#)
2. [Types of Virtualizations](#)
3. [Hypervisor vs. OS Virtualization](#)
4. [Why do we need docke?](#)
5. [Docker Toolbox vs. Docker Desktop](#)
6. [Image vs. Container](#)
7. [Introduction](#)
8. [Our first Container](#)
 1. [Cleanup containers](#)
9. [Our first own image \(Dockerfile\)](#)
 1. [The Docker file](#)
 2. [The .dockerignore file](#)
 3. [Best practices for writing a Dockerfile](#)
 1. [Understanding build caches](#)
 2. [Understanding multi-stage builds](#)
 3. [Difference between COPY and ADD](#)
10. [Docker Networks](#)
 1. [Network driver summary.](#)
11. [Environment variables and secrets](#)
 1. [Docker secrets](#)
12. [Docker Bind Mounts](#)
 1. [Bind Mounts vs. Docker Volumes](#)
13. [Sharing your Docker image](#)
14. [The docker-compose.yml](#)
15. [It's your turn!](#)
16. [Where to go next](#)
17. [References](#)

Definition virtualization

Virtualisierung bezeichnet in der Informatik die Nachbildung eines Hard- oder Software-Objekts durch ein ähnliches Objekt vom selben Typ mit Hilfe eines Abstraktions-Layers.



Types of Virtualizations

Virtualisierung

Hardware
Partitionierung



Hardware
Virtualisierung
(Hypervisor)

1. Binary Translation
2. Para-Virtualisierung
3. Hardware-Assisted

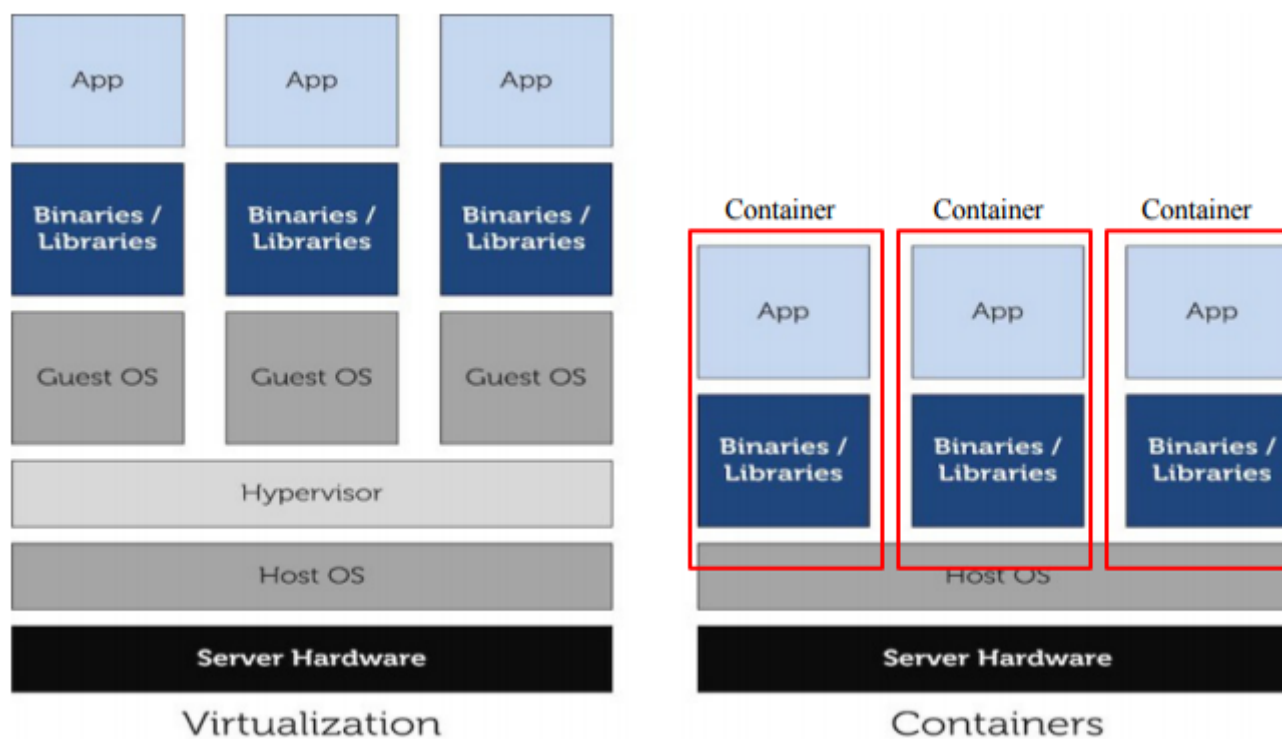


OS
Virtualisierung



Hypervisor vs. OS Virtualization

„Hypervisors are the living proof of operating system's incompetence“



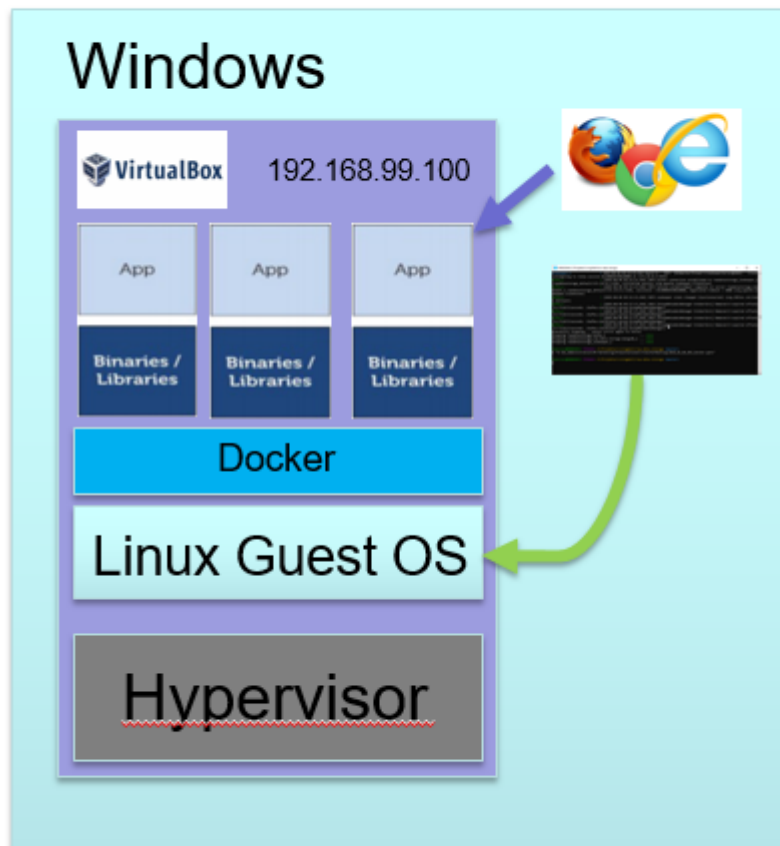
Why do we need docke?

- Leichtgewichtig
- Isolation
- Klares Dependency Management
- Portabel (gleiches Verhalten überall)
- Skalierbar innert Sekunden
- Schnellere CI/CD Zyklen
- Fundament für Microservices

«develop, deploy, and run»

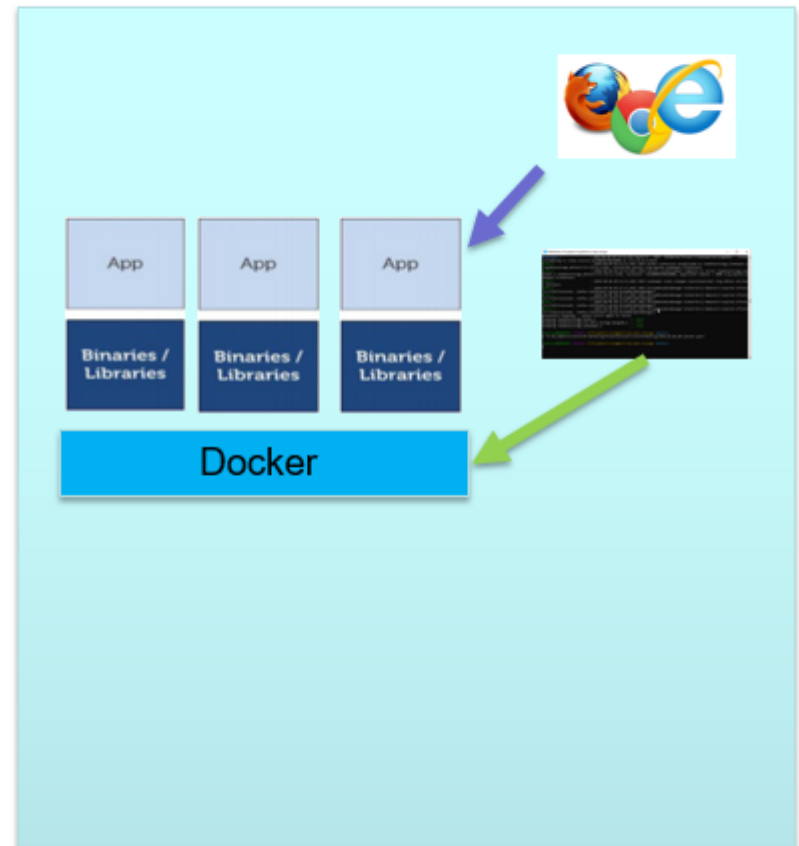
Docker Toolbox vs. Docker Desktop

Docker Toolbox



Access by IP (192.168.99.100)

Docker Desktop



Access by localhost

Image vs. Container

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)

Introduction

First we check if all of us have the appropriate docker version. Therefore, run:

```
docker version
```

To get some information about docker, run:

```
docker info
```

Our first Container

To pull the our first image from docker-hub onto our local laptop, run:

```
docker pull hello-world
```

We can now see our first image by typing:

```
docker image ls
```

 or the short form

```
docker images
```

Every image has a REPOSITORY and a TAG. The repository is basically the name of the image and the TAG the version of the image.

To start our just pulled image, use the run command:

```
docker run hello-world
```

With run an image is turned into a container and executed. The hello-world docker image explains in a few sentence what we just did:

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.

If we want to see which containers are currently running, we can use:

```
docker container ls
```

 or the short form

```
docker ps
```

Surprisingly, we can't see the hello-world container. This is because it only outputs the given msg and shut down afterwards. To also see the stopped containers, we can use:

```
docker container ls --all
```

 or the short form

```
docker ps --all
```

Cleanup containers

We can delete a container by using:

```
docker rm [NAME or CONTAINER ID]
```

It is convenient to use the following command to remove all stopped containers:

```
docker container prune
```

 but be careful ;-)

As we can see the hello-world image still persists on our laptop. To remove also the unused image we can type: `docker image rm [IMAGE ID oder REPOSITORY]` or the short form `docker rmi [IMAGE ID oder REPOSITORY]`

The same as for deleting containers we can also type:

```
docker image prune
```

 clean up dangling images

```
docker image prune -a
```

 clean up unused images

dangling=all images where no name is given - when you run `docker build .` without a name (is part of the quantity of unused images)

unused=all images that are not associated with any container

Our first own image (Dockerfile)

We have created an application that publishes a web page and counts visits. The source code for this web page is written in python and looks as follows:

app.py:

```
from os import path

from flask import Flask
import os
import socket

FILE_PATH = 'data/count.txt'
app = Flask(__name__)

@app.route("/")
def hello():
    visits = 0
    if path.exists(FILE_PATH):
        with open(FILE_PATH, 'r') as count_file:
            visits = int(count_file.read())

    visits += 1
    with open(FILE_PATH, 'w') as count_file:
        count_file.write(str(visits))

    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)
```

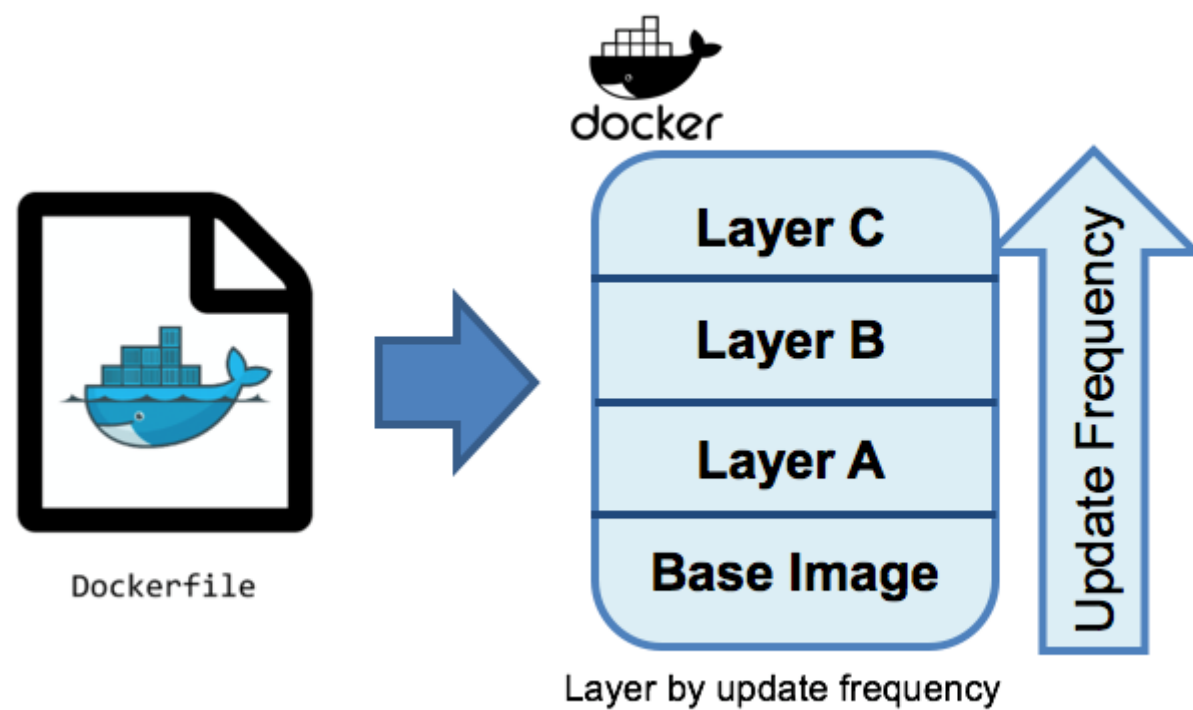
And the corresponding requirements.txt.

requirements.txt:

```
Flask
```

The Docker file

The question now is. How does docker know, how to create a docker image. To answer for this question is written in the **Dockerfile**.



The Dockerfile for our project is listed bellow:

```
# Use an official Python runtime as a parent image
FROM python:3.7-alpine

# Set the working directory to /app
WORKDIR /app

# Copy and install any needed packages specified in requirements.txt
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy all source code
COPY . .

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

# Define default environment variable
ENV NAME World

# Command which is executed when docker run is executed
CMD ["python", "app.py"]
```

We are ready to build the app. Now run the build command. This creates a Docker image, which we're going to name using the --tag option. Use -t if you want to use the shorter option.

```
docker build --tag=accesscounter .
```

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

As you see, the image was tagged with "latest". To give a specific version to the docker image, use the command: `docker tag`

```
accesscounter:latest accesscounter:0.0.1
```

This can also be done directly in the build command like: `docker build --tag=accesscounter:0.0.1 .`

How to list the content of my image:

It is not possible to list the content of an image without starting it. But didn't we said that whenever the image is started the command defined in CMD is executed? This is true, but the start command can be overwritten and because basically all images inherit from a linux distribution, `sh` is installed in all of them. With the command bellow, we can run an interactive docker session: `docker run -it accesscounter:0.0.1 sh` To navigate inside the container we can use the normal linux commands like `cd` or `ls`. A listing of our app directory looks like:

```
-rwxr-xr-x  1 root    root      530 Jun  4 14:03 Dockerfile
drwxr-xr-x  2 root    root     4096 Jun  4 14:08 _not_for_production
-rwxr-xr-x  1 root    root      714 Jun  4 13:57 app.py
drwxr-xr-x  2 root    root     4096 Jun  4 13:51 data
-rwxr-xr-x  1 root    root        5 Jun  4 13:49 requirements.txt
```

In the listing is a folder called `_not_for_production`. I'm not sure, if it should be in our production ready docker image. ;-)

The .dockerignore file

The **.dockerignore** acts like a `.gitignore`. Everything defined in it, won't be copied when `COPY . .` is run. To exclude the unwanted folder, add the following line into the `.dockerignore`:

_not_for_production

When you run the build command again. You should see, that the folder does not appear any longer.

Best practices for writing a Dockerfile

- always set the timezone of your container. Otherwise the timezone from the host is taken.
- include the `.git` and `.idea` folder in the `.dockerignore` file
- first copy dependencies afterwards source code.
- check your docker image with `docker history [IMAGE ID]` to get an impression what is increasing the size of your image
- remove unused packages after build or use multi-stage builds

Taken from experience and here [Best practices for writing a Dockerfile](#)

Understanding build caches

When building an image, Docker steps through the instructions in your `Dockerfile`, executing each in the order specified. As each instruction is examined, Docker looks for an existing image in its cache that it can reuse, rather than creating a new (duplicate) image.

If you do not want to use the cache at all, you can use the `--no-cache=true` option on the `docker build` command. However, if you do let Docker use its cache, it is important to understand when it can, and cannot, find a matching image. The basic rules that Docker follows are outlined below:

- Starting with a parent image that is already in the cache, the next instruction is compared against all child images derived from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated.
- In most cases, simply comparing the instruction in the `Dockerfile` with one of the child images is sufficient. However, certain instructions require more examination and explanation.
- For the `ADD` and `COPY` instructions, the contents of the file(s) in the image are examined and a checksum is calculated for each file. The last-modified and last-accessed times of the file(s) are not considered in these checksums. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated.
- Aside from the `ADD` and `COPY` commands, cache checking does not look at the files in the container to determine a cache match. For example, when processing a `RUN apt-get -y` update command the files updated in the container are not examined to determine if a cache hit exists. In that case just the command string itself is used to find a match.

Once the cache is invalidated, all subsequent `Dockerfile` commands generate new images and the cache is not used.

Understanding multi-stage builds

Multi stage builds are used to decrease image size drastically. With having more than one stage, we can create one stage for building our application, where we need to install all build packages we need. This build packages aren't needed to run the application afterwards. This is the place where the second stage comes in. We can create a minimal installation with only the dependencies we need to run the application.

An example of a multi-stage build for building an Angular Application and serving it with Python (flask) can look as follows:

```
# get latest LTS version of node from https://hub.docker.com/_/node/
FROM node:11-alpine

# Create app directory
WORKDIR /tmp

# Clone terminal1-app and build it
RUN apk update && apk add --no-cache bash git openssh
RUN git clone https://gitlab+deploy-token-1:XXXX@git.ihomelab.ch/we-know-you/terminal-app.git .
RUN npm install
RUN $(npm bin)/ng build --prod

# create flask container
FROM python:3.7

# setup timezone
ENV TZ=Europe/Zurich
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

WORKDIR /app

# install flask application
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
```



```
# copy previously build terminal-app
COPY --from=0 /tmp/dist/terminal-app/ src/terminal_app/public/

# run application
WORKDIR /app/src
EXPOSE 5000
CMD ["gunicorn", "--worker-class", "gevent", "-w", "1", "-b", "0.0.0.0:5000", "__init__:app"]
```

Difference between COPY and ADD

COPY and **ADD** are both **Dockerfile** instructions that serve similar purposes. They let you copy files from a specific location into a Docker image.

COPY takes in a src and destination. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.

ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

Docker Networks

As we did with the hello-world image, we can start our image by running:

```
docker run accesscounter:0.0.1
```

The log messages on the screen indicate that the webserver is running and we can access it on <http://localhost:80>. But we only get an error: Not able to connect to website....

Docker is able to create five different types of (isolated) networks. If nothing is specified during the run command, docker creates a so called **bridge** network. Which is basically a standalone network where we can forward ports from our host network to the docker network. Bellow is a short description and a summary about each network type:

- **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. See bridge networks.
- **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. See use the host network.
- **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. See overlay networks.
- **macvlan:** Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks.
- **none:** For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See disable container networking.
- **Network plugins:** You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

Network driver summary

User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.

Host networks are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.

Overlay networks are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.

Macvlan networks are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.

Third-party network plugins allow you to integrate Docker with specialized network stacks.

With this theory in mind we somehow have to forward port 80 from the host to port 80 from the docker container. It is achieved by the **-p** parameter for **docker run**:

```
docker run -p 80:80 accesscounter:0.0.1
```

When you want to run the container in a different process than the terminal is running you can pass **-d** for "detached" mode.

Environment variables and secrets

We can now run our container and access port 80 from outside the docker container. But how is it possible to configure a docker container. There are two options: First, we can pass a configuration file from the host into the docker container. For this check the next chapter. Second, we can set environment variable, which the application inside the container can read and act accordingly. In our source code, this is done on line 26. To pass such an environment variable, we have to use the `--env` or `-e` of the `docker run` command: `docker run -p 80:80 -e NAME=Iwan accesscounter:0.0.1`

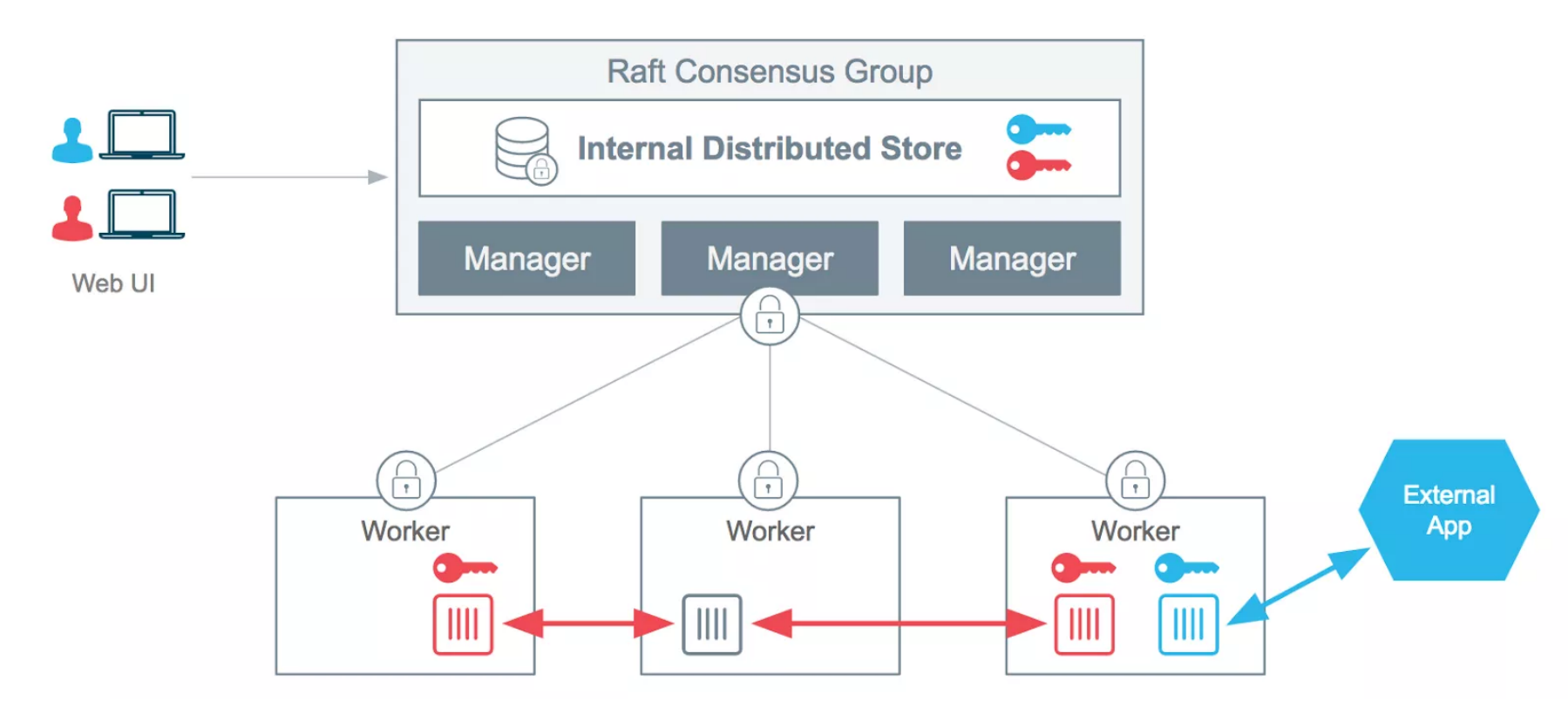
If you have a huge list of env variables to set, you can also use `--env-file` to pass the content of the file as env variable. The file need to be formatted this way:

```
VAR1=value1
VAR2=value2
USER=denis
```

Docker secrets

Docker secrets are a way to pass sensitive information such as password, certificates, keys,... into a docker container. The sad thing is that docker secrets only works with docker swarm. (when you're running a cluster) Anyway, the concept behind secrets is great. When you have for example a password you do: `docker secret create my_secret -` afterwards type `"this is my password"` and press `ctrl+c`. This way your password is not stored in the bash history.

By default, the container can access the secret at `/run/secrets/<secret_name>`, but you can customize the file name on the container using the target option. This way you can pass a secret into a swarm service: `docker service create --name redis --secret my_secret redis:alpine`



Docker Bind Mounts

One of the greatest feature of docker is that the container itself is fully stateless. This means we can simply make a change in our source code and deploy it, without doing any migrations. Please replace the following line in your app: Replace line 23:

```
html = "<h3>Hello {name}</h3>" \
      "<b>Hostname:</b> {hostname}<br/>" \
      "<b>Visits:</b> {visits}"
```

with (change [Your Name] with your name)

```
html = "<h3>Hello to the great new version from [Your Name]</h3>" \
      "<b>Hostname:</b> {hostname}<br/>" \
      "<b>Visits:</b> {visits}"
```

and now build and start the new version (don't forget to increase your version number `accesscounter:0.0.2`). Mhhh, it seems that our visitor count is gone. :(This is because the persistent data file was inside our container and we replaced this container. The container wasn't as stateless as we thought before. To overcome this, we need to mount local file paths into our docker container. The same as we do with network drives we need to define where the files are stored on the server (in our case this is the host system) and where this path is mounted in the client (in our case this is the container). For this docker uses the `-v` parameter in the `run` command.

```
docker run -v [HOST PATH]:[CONTAINER PATH]
```

And because we know that our persistent directory in the container is `/app/data` we can now mount this directory to any location on the host e.g.:

```
docker run -v /c/tmp/accesscounter/data:/app/data -p 80:80 accesscounter:0.0.2
```

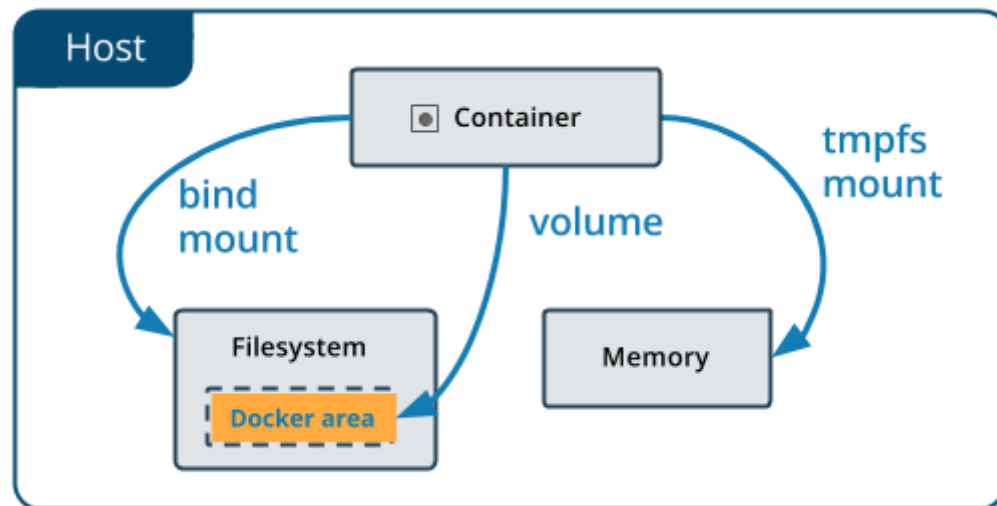
When you now switch between versions, access count should remain.

Bind Mounts vs. Docker Volumes

There is a difference between bind mounts and docker volumes. While bind mounts are dependent on the directory structure of the host machine, volumes are completely managed by Docker. According to Docker Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.

This picture explains the difference between bind mount and volumes:



To do the same as we did with bind mount we first have to create our own volume: `docker volume create accesscounter-data` With `docker volume ls` we can see the created volume. If we want to access the physical directory of the volume we can get the mount point with `docker inspect accesscounter-data`. To start the container with the volume instead of the mounted path, it works the same as with bind mounts: `docker run -v accesscounter-data:/app/data -p 80:80 accesscounter:0.0.2`

Sharing your Docker image

To demonstrate the portability of what we just created, let's upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker's public registry (hub.docker.com) by default. But we at iHomeLab have a private registry accessible with <https://nexus.ihomelab.ch>. With nexus we host not only a docker registry but also maven repos, npm repos, pip repos, ... To differentiate between this repositories, we need to assign ports. The port for the iHL docker registry is 5000. First we need to connect the local docker installation with this remote repository.

```
docker login nexus.ihomelab.ch:5000
```

When we want to push images to a private remote docker registry, the repository must be formatted this way: `[Remote repository server]/[repository name]:[TAG]`. If we wouldn't do so, docker CLI would try to push it to hub.docker.com For iHomeLab the repository name has again two parts. The first part is the project name and the second part the name of the component/service. In our case this means we have to retag the created image with the following TAG. Please replace [Name] with your name.

```
docker tag accesscounter:0.0.2 nexus.ihomelab.ch:5000/docker-workshop/accesscounter-[NAME]:0.0.2
```

Now we are ready to share our images with others :-)

```
docker push nexus.ihomelab.ch:5000/docker-workshop/accesscounter-[NAME]:0.0.2
```

Now go to <https://nexus.ihomelab.ch> and check if you can find your image.

The docker-compose.yml

Docker-compose is a way to start multiple images at once in a comfortable way. We are now going to start the accesscounter of three of your colleagues. The `docker-compose.yml` has the following structure:

```
version: '3'
services:
  accesscounter-iwan:
    image: nexus.ihomelab.ch:5000/docker-workshop/accesscounter-iwan:0.0.2
    ports:
      - 80:80/tcp
    volumes:
      - accesscounter-v1:/app/data
  accesscounter-martin:
    image: nexus.ihomelab.ch:5000/docker-workshop/accesscounter-martin:0.0.2
    ports:
      - 81:80/tcp
    volumes:
      - accesscounter-v2:/app/data
```

```
accesscounter-rolf:
  image: nexus.ihomelab.ch:5000/docker-workshop/accesscounter-rolf:0.0.2
  ports:
    - 82:80/tcp
  volumes:
    - accesscounter-v3:/app/data

volumes:
  accesscounter-v1:
    external: false
  accesscounter-v2:
    external: false
  accesscounter-v3:
    external: false
```

When you now type `docker-compose up` it checks for a file named `docker-compose.yml` and starts all services. It have never been easier to start three highly complex applications. Just to remember, you haven't installed any version of python or bother with dependencies. :-)

It's your turn!

Take an application of your choice and try to dockerize it!

Where to go next

You could try to scale your service with swarm and loadbalancing requests. An example docker-compose for a load balanced App can be found here: <https://docs.docker.com/get-started/part3/>. You could also try to run your container on one of the big cloud providers or use a scaling service like <https://kubernetes.io/>.

References

- <https://docs.docker.com/get-started/>
- <https://docs.docker.com/compose/compose-file>
- <https://docs.docker.com/samples/>