Tools SS 2022

Introduction to Docker

Florian Beenen

Schaltungstechnik und Simulation

22.06.2022

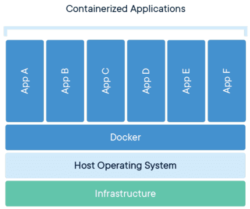
Contents I

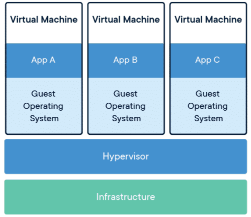
Introduction

What are Containers?

To understand what Docker can do, we must first understand what Containers are...

- Containers provide an isolated environment to execute programs.
- Containers isolate the application (and its runtime environment) from the infrastructure.
- Very similar to Virtual Machines (VMs): You can run multiple virtual operating systems on a host that are separated from each other.
- Yet different that VMs: A container shares the "operating system" with the host but may have its own separated network, compute resources, storage and execution environment.





Source: `https://www.docker.com/resources/what-container/`

Why use Containers?

- Containers can be set up and torn down *much* faster than VMs.
  - One application = one container!
  - Example: Compiler (gcc) in Container. For each compilation job, launch a Container, let it do its thing, tear down Container.
- Containers should (ideally) be stateless: A "clean" execution environment is created upon start and the Container should not have any side effects.
- Containers are portable: They bundle everything they need so they can be moved/executed on another host without issues (like VMs – with the same restrictions).
- Dependencies of programs can be managed in a fine grained way:
  - App A needs a library with version 1.2
  - App B requires version 1.3 of this library.
  - Libraries are not spread across system folders but are inside the Container!

Applications of Containers

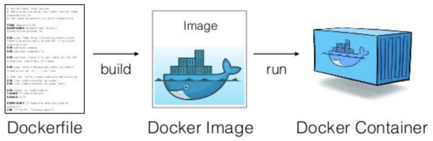
- As "VM Lite"
  - Long-living Containers that contain independent services.
  - Examples: Database-Servers, Webservers, Loadbalancers.
  - Same approach as deploying VMs but with less overhead.
- Microservice-based applications made out of multiple Containers
  - Applications are developed in a distributed fashion.
  - Applications use numerous micro-services that can be scaled out individually.
- As "job environment" for scheduled tasks
  - Used by build tools like Jenkins, GitLab, ...
  - Task: Create Container, execute a script in a defined environment, deliver an artifact, destroy Container.

What is Docker

- Docker is a platform/software to run Containers!
- Docker is used to manage Containers:
  - Start/Stop Container
  - Access Container (e. g. run a shell)
  - Attach networking to Containers
  - Manage storage shares between host and Containers
- Docker provides means to define/create the "working environment" of a Container (➥ this is called an Image).

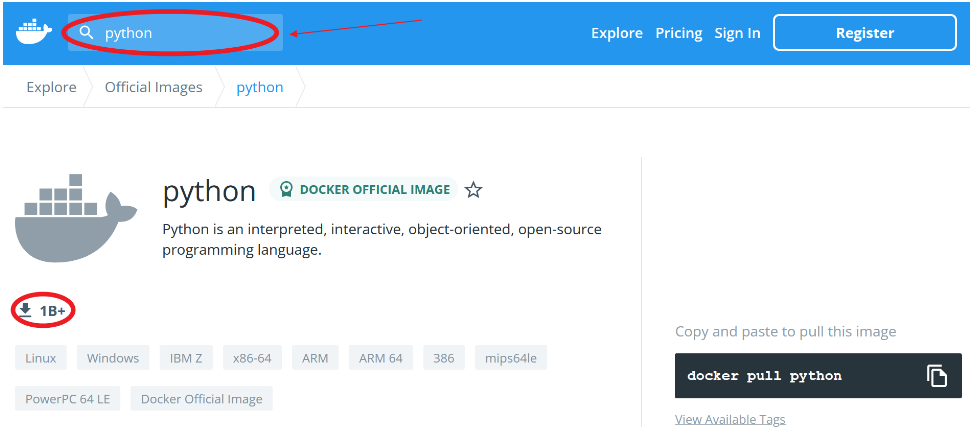
What are Images

- An Image is the bundle of software and configurations that are used as base for a Container.
- Container vs. Image?
  - A Container is the instance of an Image!
  - Multiple Containers may be started from the same Image and are independent.
- What makes up an Image is defined in a **Dockerfile**
  - The Dockerfile can be regarded as a script to compile the Image.
  - If the Dockerfile is built, the Image can be used to start Containers.
  - You can share the Image with others via a Registry (e. g. `https://hub.docker.com/`).



Source: `https://medium.com/platformer-blog/practical-guide-on-writing-a-dockerfile-for-your-application-89376f88b3b5`

Find Images on the Hub



python DOCKER OFFICIAL IMAGE

Python is an interpreted, interactive, object-oriented, open-source programming language.

Copy and paste to pull this image

```
docker pull python
```

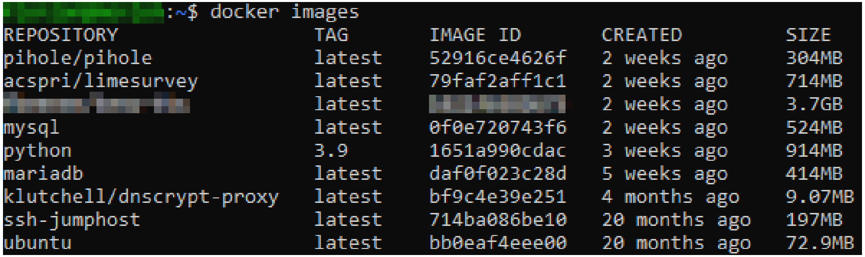
View Available Tags

Source: `https://hub.docker.com/_/python`

Using Docker

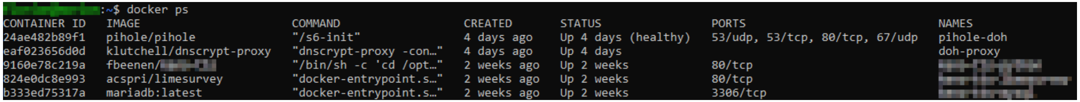
The Docker Daemon

- Docker is managed by the Docker daemon (➥ a system service).
- You can interact with the daemon by using the `docker` command.
- To do this, you need very high privileges (i. e. you need to be `root` effectively...).
- Basic commands
  - `docker images` : Shows Images currently available on your machine
  - `docker ps` : Shows running Containers



```
[redacted]:~$ docker images
REPOSITORY                 TAG       IMAGE ID       CREATED          SIZE
pihole/pihole              latest    52916ce4626f   2 weeks ago      304MB
acspri/limesurvey          latest    79faf2aff1c1   2 weeks ago      714MB
[redacted]                 latest    [redacted]     2 weeks ago      3.7GB
mysql                      latest    0f0e720743f6   2 weeks ago      524MB
python                     3.9       1651a990cdac   3 weeks ago      914MB
mariadb                    latest    daf0f023c28d   5 weeks ago      414MB
klutchell/dnscrypt-proxy   latest    bf9c4e39e251   4 months ago     9.07MB
ssh-jumphost               latest    714ba086be10   20 months ago    197MB
ubuntu                     latest    bb0eaf4eee00   20 months ago    72.9MB
```

Showing Running Containers



```
[redacted]:~$ docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED       STATUS                  PORTS                            NAMES
24ae482b89f1   pihole/pihole              "/s6-init"               4 days ago    Up 4 days (healthy)     53/udp, 53/tcp, 80/tcp, 67/udp   pihole-doh
eaf023656d0d   klutchell/dnscrypt-proxy   "dnscrypt-proxy -con…"   4 days ago    Up 4 days                                                doh-proxy
9160e78c219a   fbeenen/[redacted]         "/bin/sh -c 'cd /opt…"   2 weeks ago   Up 2 weeks              80/tcp                           [redacted]
824e0dc8e993   acspri/limesurvey          "docker-entrypoint.s…"   2 weeks ago   Up 2 weeks              80/tcp                           [redacted]
b333ed75317a   mariadb:latest             "docker-entrypoint.s…"   2 weeks ago   Up 2 weeks              3306/tcp                         [redacted]
```

Starting our first Container

- Remember: Docker Containers are meant to be shipped with everything they need to execute...
- Our first Container launch will be a "hello world" example.
- For this purpose, the `hello-world` Image exists on the Hub.
- We run the Container by typing `docker run hello-world`

Starting our first Container (2)

```
user@server:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:80f31da1ac7b312ba29d65080fddf797dd76acfb870e677f390d5acba9741b17
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
```

Starting Ubuntu

- We want to experiment a little bit in a clean environment and therefore want to have a Ubuntu Container.
- We launch the "default" Ubuntu Image which will be fetched from the Hub
- `docker run -it ubuntu`
  - Why special options?
  - `-i` gives us an interactive session (➥ attaches `stdin`).
  - `-t` allocates a terminal for us, so we can type in commands.
- Result: `root@cf62f1786c1a:/#` (... and now...?)
  - We operate as the `root` user inside the Container.
  - The Container has a strange name (... so what?)
- We can now do whatever we want!

Using Ubuntu

- Let's go and install some software!
- Run `apt update` to load newest package lists
- Run `apt install cowsay` to install something funny...
- After installation run `/usr/games/cowsay "Docker is the best"`

```
root@965e4292f2be:/# /usr/games/cowsay "Docker is the best"
 ____________________
< Docker is the best >
 --------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Using Ubuntu (2)

- This Ubuntu Image is *extremely* minimalistic!
- This is good to keep its size down (only 78 MB).
- You need to install nearly everything manually.
  - This is also good because you are forced to think about your dependencies!
- Remember: The Container is a throw-away instance!
- If you run `exit`, the session will terminate and the Container is stopped.
- Once the Container is stopped, you cannot access its data anymore.
- You cannot restart a stopped Container!

Working with Containers

- Every Container needs a primary process (➥ the `bash` process in our Ubuntu Container).
- This process needs to be blocking!
- If this process terminates or returns control, the Container terminates.
- In our example, we had a direct keyboard input to the primary process.
- If we want to "detach" the input and return to our host, we can use Ctrl + P Q .
- We can "re-attach" again by running `docker attach <Container Name>`.
- Note: This detach/re-attach thing is rarely the way to go. Only use this for the purpose shown here...
- If you want to get a shell in a *running* Container, do this: `docker exec -it <Container-Name> /bin/bash`.
- Hint: As `<Container-Name>` you may use either the name reported under `docker ps` or a non-ambiguous prefix of the ID!

Working with Containers (2)

- Demonstration what happens when the primary process terminates:
- Just run `docker run ubuntu`.
- What happens? ➥ Nothing?
- Wrong: Ubuntu Container is started, main process (implicit `/bin/bash`) is launched, no commands there, process terminates, Container terminates.

Stopping Containers

- When you detach from a running Container or start Containers in the background, `docker ps` still shows you that the Container is up.
- You can stop the Container via the Docker Daemon by running `docker stop <Container-Name>` (graceful shutdown).
- Most of the time I rather use `docker kill <Container-Name>` ☺.

Source: https://www.redbubble.com/de/i/sticker/Docker-t%C3%B6ten-von-Caldofran/38462159.EJUG5

Cleaning up the Mess

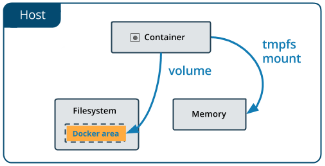
- By default, Docker leaves the "junk" of stopped Containers on your machine.
- You may want to clean up the files of stopped Containers on a regular basis.
- I have a script that does this – I use this on every machine I have...

```bash
#!/bin/bash
docker rm -v $(docker ps -a -q -f status=exited)
docker rm -v $(docker ps -a -q -f status=created)
docker rmi $(docker images -f "dangling=true" -q)
```

- `docker rm` removes the stopped Containers.
- `docker rmi` removes (unreferenced) Images.
- You can find out about your stopped Containers with `docker ps -a`.

Volumes

- Once you stop a Container, all its data is lost ☹.
- There are *countless* use-cases where you want to preserve some data.
  - You run a database server in Docker: ➥ You need to preserve the database.
  - You run a webserver in Docker: ➥ You need to have access to files from the host.
- **Volumes** lets you access directories/files from the host inside the Container in a transparent way!
- You tell the Daemon which host directory should be "mounted" where inside the Container.



Source: `https://docs.docker.com/storage/volumes/`

Example: Volumes

- We want to write a program in C on the host, compile it in a Container and launch the program on the host again.
- Use case: We may not want to install a compiler on our host.
- Specify Volume at Container start:
  - `-v /path/on/host:/path/in/container`
  - Example: `-v /tmp/tools/:/opt`
- We can also specify the working directory at Container launch:
  - `-w /path/in/container`

```
docker run \
  -v /tmp/tools/:/opt \    # Mount the Volume to /opt in the Container
  -w /opt \                # Specify working directory
  gcc \                    # This is the Image Name!
  gcc prog.c -o myapp      # What command to run inside the Container
```

Example: Databases

- Another example: We want to run a database (➥ MariaDB) inside Docker.
- Check out the Image `mariadb` (https://hub.docker.com/_/mariadb)
- We need to preserve the directory, where MariaDB stores the data! Otherwise it will be lost on Container termination.
- The Containers also recognizes some environment variables to configure the Container (➥ `-e` parameter).

```
docker run \
    --rm \                                  # Delete Container when stopped
    --name "my-mysql-container" \           # Custom name for Container
    -v /opt/docker-mysql/:/var/lib/mysql \  # Mount the data directory
    -e MARIADB_ROOT_PASSWORD="adminadmin123" \ # Set mysql root password
    -e MARIADB_USER="myUser" \              # Create a user account
    -e MARIADB_PASSWORD="userpass123" \     # Password for user account
    mariadb                                 # Image Name
```

Dockerfile

- The Dockerfile is a description/script how an Image is created.
- A Dockerfile can be built/compiled into an Image.
- Every operation in the Dockerfile creates an Image layer, which can be shared among other Images (for reuseability).
- Usual structure and directives:
  - `FROM <OtherImage>` : Specifies a base Image that is used as a foundation. Common Images are `debian`, `ubuntu` or `alpine`.
  - `RUN apt update && apt install -y <package-list>` : Install packages in the new Image (example for Debian/Ubuntu based Images).
  - `COPY /hostpath /dockerpath` : Copy a file from `/hostpath` on the host into `/dockerpath` inside the new Image.
  - `CMD <path-to-program>` : Specify which program should be executed when starting the Container (➥ e. g. `/bin/bash`).

Full Example of Dockerfile

Simplified version of a TexLive Image:

```
FROM ubuntu
WORKDIR /root
ENV TZ=Europe/Berlin
RUN apt-get update && apt-get install -y git unzip curl openssh-client texlive-
    full
CMD /bin/bash
```

- Name this file `Dockerfile` (no file extension) and place it in an empty directory.
- We can now build the Image with `docker build -t myTexLiveImage /home/user/DirWithDockerfile`.
- This is available on the Docker Hub via the Image name `fbeenen/texlive`.
- Writing Dockerfiles can be very complicated – these examples are just a simple yet very useful demonstration.

Networking in Containers

- Docker handles the network inside the Containers.
- By default (usual case), the Containers are attached to a special network (e. g. via the `docker0` interface on the host).
- IP addresses of Containers are managed by Docker.
- Communication to the outside world is done via NAT on the host. Docker can create the required IPTables rules for this. If NFTables are used, you need to do everything manually.
- You can create new networks managed by docker with the `docker network` command (➥ advanced topic).
- Even more advanced: You can connect a Container to the host network interface, so it shares the IP with the host (do not use this, unless required).

List of Networks

```
user@host:~$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
2c416e43aa4e   bridge    bridge    local
77229220d687   host      host      local
8625ccc786b4   none      null      local
599f15a99615   servers   bridge    local
```

- `bridge` : The default network
- `host` : The network of the Docker host
- `none` : For Containers that need to network
- `servers` : Manually created network for (my) VM-like "servers".

Communication between Containers

- Setting: You have 2 Containers that need to communicate (e. g. a webserver Container and a database Container).
- You can "link" the Containers with the `docker run` command and reach a Container with a custom name!

```
user@host:~$ docker run -d \                          # Run detached
    -v "/opt/uhd-tools-apache/:/var/www/html" \ # Mount webserver directory
    -v "/etc/localtime:/etc/localtime:ro" \   # Sync time of Container and Host
    -e TZ="Europe/Berlin"  \                    # Set timezone
    --hostname="myserver.example.org" \         # Set Hostname
    --restart=always \                    # Restart Container on Docker restart
    --name=tools-vl-web \                  # Container Name
    --link mysqlcontainername:mysql       # Link MySQL Container to name 'mysql'
    php:7.2-apache
```

➥ We can now connect to the MySQL-Container using the name `mysql`. Docker handles the name resolution.

Connection to the Outside World

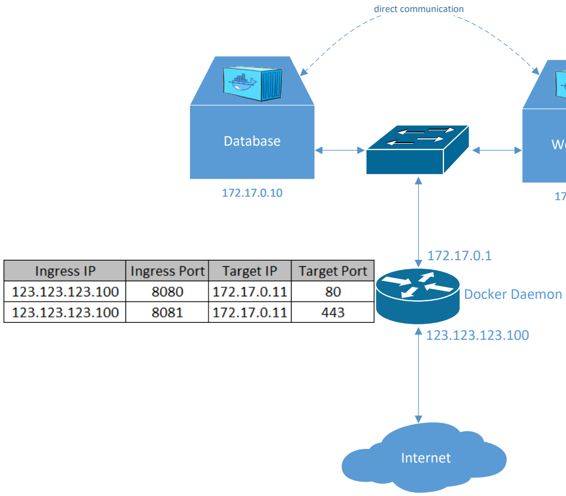
- If you want to access a service (that runs on a port) from outside the Docker environment, you need to **publish** the port!
- Publishing a port means that Docker forwards traffic towards the host on a specific port to the Container (i. e. Docker does either NAT or acts as a proxy, depending on configuration).

Example:

```
user@host:~$ docker run -d \                               # Run detached
    -v "/opt/uhd-tools-apache/:/var/www/html" \ # Mount webserver directory
    -v "/etc/localtime:/etc/localtime:ro" \    # Sync time of Container and Host
    -e TZ="Europe/Berlin"  \                   # Set timezone
    --hostname="myserver.example.org" \        # Set Hostname
    --restart=always \                   # Restart Container on Docker restart
    --name=tools-vl-web \                      # Container Name
    -p 8080:80 \                       # Map host port 8080 to Container port 80
    php:7.2-apache
```

➥ If host has IP `123.123.123.120`, you can now access the Container via `http://123.123.123.120:8080`.

Connection to the Outside World (2)



Ingress IP	Ingress Port	Target IP	Target Port
123.123.123.100	8080	172.17.0.11	80
123.123.123.100	8081	172.17.0.11	443

Some Thoughts about Security

- Docker is cool because you package everything you need in your image.
- Docker is really dangerous because you package everything you need...
- Example: You build your Image off of `ubuntu:latest`. Your app runs for 2 years. You will **never** receive security updates for your runtime libraries because they are bundled in your Image!
- Solution: Re-build **all** your Images on a regular basis (e. g. by a GitLab CI pipeline)!
- Trust issues: When using Images from the Hub, you trust that this Image does not contain malware/bad things. It is also **very** difficult to guarantee that **every** sub-Image that you use is clean and built on a regular basis (see above).
- Solution: Build **everything** yourself. You can use the Dockerfiles from the Hub, but do not rely on their (old) build!
- Please watch this talk for your dose of WTF moments and Docker in action: `https://www.youtube.com/watch?v=RqhxphyVL88`.

Exercises

Test System

- We have a testing system that you can use to try the following examples:
- Connect to it via SSH!
- Hostname: `sustools22.ziti.uni-heidelberg.de`
- Port: 22
- Username: Your Uni-ID
- Password: The password for your Uni-ID
- You can only log in from the university network **and** as a participant of the SS22 tools lecture.

Hello World

Try and re-create the "Hello World" launch from Slide 12.

Hello World

- Connect to a Docker-ready host via SSH (or else).
- Issue `docker ps` to check, if Docker runs.
- Now run `docker run hello-world` and see, if everything works.
- Afterwards, ensure that the downloaded Image is listed under `docker images`.
- Now, delete the Image again with `docker rmi`. Why is this not possible? Try to resolve this (Hint: `docker ps -a`).

Running Interactive Commands

Running Interactive Commands

- We want to run a few commands in an interactive Ubuntu Container, like on Slide 14!
- Start a clean Ubuntu Container (Remember `docker -it ubuntu`).
- Look around – ensure that you are indeed executing commands in the Container.
- Close the Container with `exit`. Ensure that you are now in the shell of the host again.
- Re-start the Container. Install the package `sl` (run `apt update && apt install sl`).
- Do you like trains? Run `/usr/games/sl`.
- Detach the session like shown on Slide 17.
- Check that the Container is still running with `docker ps`.
- Kill the Container with `docker kill`.

Writing a Dockerfile

Writing a Dockerfile

- We want to build an Image that prints the current time and then exists.
- The current time should be obtained by the WorldTimeAPI. We can get the current time with `curl "http://worldtimeapi.org/api/timezone/Europe/Berlin.txt"`
- We can isolate the output properly with `curl -s "https://worldtimeapi.org/api/timezone/Europe/Berlin.txt" | grep -Po '(?<=^datetime:  ).*'` (extra task: try to understand what is going on here).
- Now, write a Dockerfile:
  - Use `ubuntu:latest` as the base Image
  - Install `curl` into the Image
  - Set the above command as `CMD` entrypoint.
- Build the Image!
- Run the Image as a Container!
- Output should be identical to running `docker run fbeenen/time`.