Professorship for Computer Science
Communication Services, Telecommunication Systems and Computer Networks

Container Virtualization

Docker by Example

September 11, 2018

Marcel Großmann, Stefan Kolb, Dr. Andreas Schönberger, Gabriel Nikol

Content

What you can expect (and what not)

What you learn

- you will learn the concept of docker
- you will learn the most used commands (not representative)
- you get a toolset to dockerize your applications
- you get some practive with the tools (use the tools in exercises)

But: Everything you will learn is in the docker documentation, too.

- you won't be a pro with docker (this needs practice)
- if you only dockerize your application from time to time this is mostly enough
- sometime you need to look into the documentation (or stackoverflow) for special cases/challenges
- this is not a guide to configure proxies, webserver etc. we focus on docker

Docker Concepts

What does Docker do:

- "Docker allows you to **package an application with all of its dependencies into a standardized** unit for software development."
- "Docker **containers** wrap up a piece of software in a complete filesystem that **contains everything it needs to run:** code, runtime, system tools, system libraries - anything you can install on a server. This guarantess that it will always run the same, regardless of the environment it is running in."

These artifacts are packed and can be exchanged.

- **Build, ship and run everywhere!**

Background: Evolution of Virtualization

- Lightweight containers enable easy horizontal scalability.

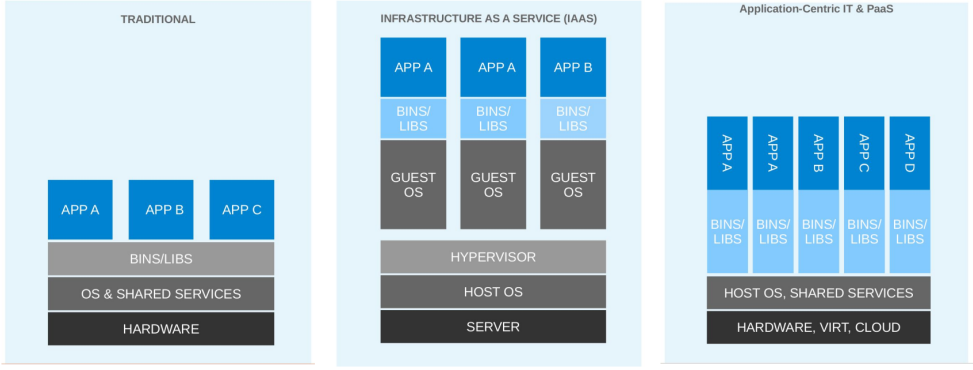

Figure 1: Evolution of Virtualization

Virtual Machines vs. Containers

Hypervisor

- Runs Operating System
- Heavyweight isolated virtual machines
- Can theoretically emulate any architecture
- VMs start via a full boot-up process
- Platform-oriented solution
- Optimized for generality

Container-enabled Kernel

- Runs processes
- Lightweight kernel namespaces
- Is less flexible in architecture emulation
- Very fast namespace + process creation
- Service-oriented solution
- Optimized for minimalism and speed

Figure 2: Virtual Machines vs. Container

Docker's Execution Drivers

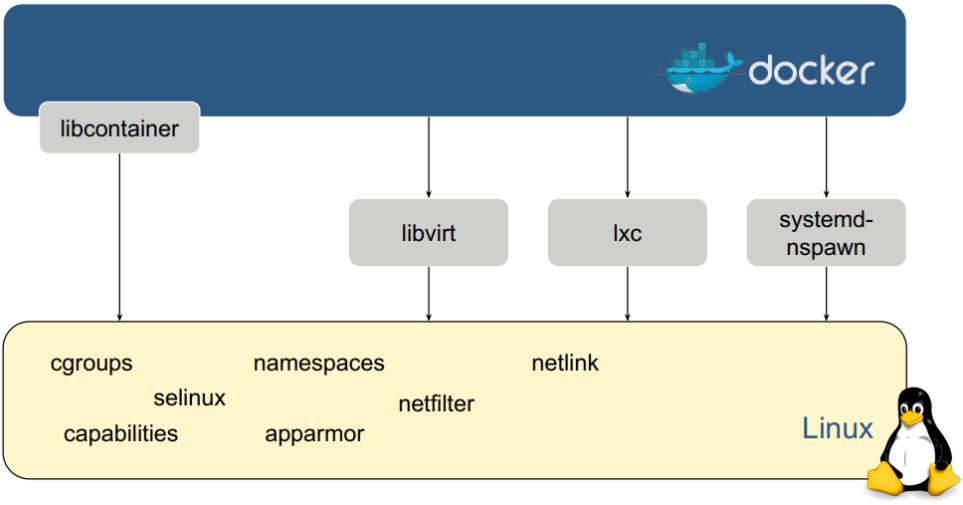

Figure 3: Docker's Execution Drivers

Container vs. Images

- An image is blueprint for a container
- An image contains of a read-only set of filesystem layers
- An instance of an image is called container
- You can have many running containers of the same image
- Major difference: container adds a writeable filesystem layer on top of the image

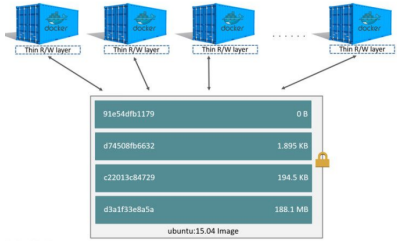

Figure 4: Containers vs Images

Changes and Updates

Copying makes containers eddicient

- only filesystem deltas need to be saved
- multiple containers can safely share a single underlying image

Changes made to a running container can always be commited to a new image

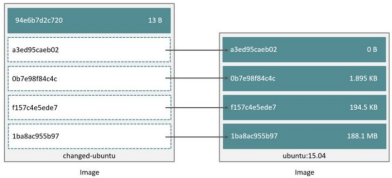

Figure 5: Changes And Updates

Data Volumes

- When a container is deleted, any data written to the contaienr that is not stored in a *data volume* is deleted along with the container.
- A data volume is a directory or file in the host's filesystem that is mounted directly into a container
- Multiple volumes can be mounted by a container and volumes can also be shared between containers

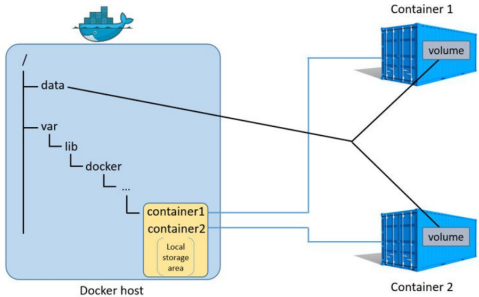

Figure 6: Data Volumes

Overall Architecture

- Docker CLI (Command Line Interface) is the docker client

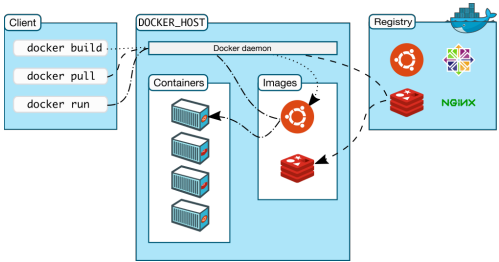

Figure 7: Docker overview

- **Daemon**: daemon of the server process to manage containers
- **Client**: user client to (remotely) controll the daemon
- **Registry**: platform for sharing and managing images

Docker List Commands

Shows **state** of containers, volumes and images.
Show **running** container only:

- *docker container ls*

Show **all** containers (both commands do the same):

- *docker container ls -a*
- *docker container ls –all*

The same for volumes, networks and images:

- docker volume ls –all
- docker image ls –all
- docker network ls

Docker Pull Command

The **docker pull IMAGE** command downloads an image from docker registry.
In our case (default) from the **public** registry - docker hub
Pull the **latest** image of busybox (default):

- docker pull busybox

Pull busybox with tag **1.29.2-glibc**:

- docker pull busybox:1.29.2-glibc

Docker Run Command

The **docker run [OPTIONS] IMAGE[:TAG]** command starts container from images.

- docker run busybox

Start container for interactive processes (*-i*) and allocate a tty (*-t*). Together written as *-it*.

Open busybox with shell:

- docker run -it busybox

Docker Run Command contd.

Other very important docker run **[OPTIONS]**: Give container names (unique):

- docker run -it –name busy busybox

delete container foobar after running:

- docker run –rm busybox

Docker Inspect Command

You can also inspect container, volumes, networks and images giving you detailed information:

- docker container inspect ID|NAME
- docker volume inspect VOLUMENAME
- docker image inspect IMAGE
- docker network inspect ID|NAME

Exercise 1

1. Open a first (power)shell.
2. download the image: busybox and ubuntu.
   What do you see? What is different between the two images?
3. start a **ubuntu** container with the name **foo**.
   List all directories (*ls*), create a new directory (*mkdir DIRNAME*), List all directories again.
4. Open a second (power)shell.
5. List all containers. What is the Id and name of your container?
6. Go back to the first (power)shell and exit the container.
   Grateful with *exit*, abort with *STRG+C*.
7. List **all** containers again. What changes?
8. inspect the ubuntu container and image
9. delete the container by using: *docker container rm ID|NAME* and list all containers again

Docker handling detached containers

Normally container run in the background without interaction.
Run container without active tty with *–detach* or *-d* option:

- docker run –detach –name influx –rm influxdb

Inspect stdout with *logs*

- docker logs influx

Stop running container with

- docker stop influx

Stop running container with

- docker stop influx

environment variables

You can define environment variables passed to a container.
Syntax: *-e=VARIABLE* or *–env=VARIABLE*
Example: print out value of environment variable foo on console

- docker run -it -e foo=bar ubuntu
- / # echo $foo

publish ports

By default no ports are accessible outside of container networks. You need to make the explicitly available to the host.

Syntax: **-p EXTERN:INTERN** or - **-publish EXTERN:INTERN**

Example: start a http server on port 80

- docker run -p 80:80 httpd
- docker run -publish 80:80 httpd

Docker volume option

Mapping of local file system into container file system.
Syntax:

- **-v LOCALFS:CONTAINERFS**
- - **-volume LOCALFS:CONTAINERFS**

Local file system path: C:/User/Name/html Example: start php container with C:/User/Name/html mapped into /var/www/html

- docker run -v C:/User/Name/html:/var/www/html php:apache

Exercise 2

1. GoTo: `https://hub.docker.com/_/mysql/` and find the setable environment variables.
2. Run all container in detached mode
3. Run a mysql container named database with root password **foobar**.
4. Check the logs of the mysql
5. Stop the mysql container
6. Run a php:7.2-apache container named webserver, publish port 80 and map directory Exercise2/html into /var/www/html
7. Ensure that the webservice's html page is available
8. list all running container and compare the output with Figure 8. It should be equal.

Figure 8: *Exercise2: docker container ls*

Docker volumes

Volumes without mounted host directory, so no absolute path needed. Increases portability of applications.
Create a volume named database.

- docker volume create database

Remember: list volumes

- docker volume ls

Map a volume like a file system volume by its name.

- docker run -v database:/var/lib/influxdb –name influx –rm influxdb

Exercise 3

1. open a first (power)shell
2. create a volume named shared_volume
3. run a ubuntu image and map shared_volume to /shared_volume
4. open a second (power)shell
5. run a influxdb image and map shared_volume to /var/lib/influxdb
6. change back to the first (power)shell
7. list (ls) the content of /shared_volume

Docker network

Until now, container were not interconnection (except over publish ports). We can link container via their names, this is provided by an internal docker DNS server. For this they must be in the same docker network.
Create a network named mynet.

- docker network create mynet

Remember: list networks

- docker networks ls

Join a container to a network:

- docker run –name somecontainer –net mynet busybox

Exercise 4

1. open a first (power)shell
2. create a network named foobar
3. run a busybox image named foo and join the foobar
4. open a second (power)shell
5. run a busybox image named bar and join the foobar
6. ping foo from the second and bar from the first (power)shell
7. stop container foo, what happens in the bar container?
8. inspect the foobar network
9. stop all containers, delete the network and clean up the environment

Docker CLI - cleanup

You can prune (delete unused) containers, volumes, networks, images.

- docker container prune
- docker image prune
- docker volume prune
- docker network prune

You can do all at once with:

- docker system prune

Afterwards check with docker [container | volume | network | image] ls

Outlook

Other usefull commands:
Execute something inside a running container - in most cases a shell: Example: detached httpd webserver and exec into bash

- docker run -d -p 80:80 –name webserver httpd
- sudo docker exec -i -t webserver /bin/bash

Override default entrypoint with docker run: Example: instead of executing apache go to bash

- docker run -it –entrypoint "/bin/bash" php

Dockerfile - Motivation

Motivation

- until now we used existing images
- you can commit changes to created conainers, this creates new images (not addressed in this tutorial)
- but cumbersome to share local created images, hard to reproduce
- solution: declarative way to create images => Dockerfiles

Dockerfile - Concept

Concept

- blueprints for images
- a list of commands to create a state of an image
- makes image creation reproducible
- each RUN instruction adds a delta-layer

Instructions

- single line statements and contain a keyword
- not case sensitive

Example:

Figure 9: *Minimalistic Dockerfile*

Dockerfile Instructions 1/6

Every Dockerfile starts with an image it is based on. For this the FROM Keyword is used.

- FROM <image>[:<tag>] [AS <name>]

Example: Ubuntu is a good start for every new docker application

- FROM ubuntu:latest

You can add comments to you Dockerfile with the # (hash) Example: A comment

- # this is comment in a Dockerfile

Dockerfile Instructions 2/6

A Dockerfile is a list of instructions to create a image. To execute commands on the shell use the RUN keyword. There exist two forms (shell and exec form): shell form:

- RUN <command>

exec form (this form is required for paths containing whitespace):

- RUN ["executable", "param1", "param2"]

In the shell form you can use a \(backslash) to continue a single RUN instruction onto the next line. This increases readability!
Example: install git inside an ubuntu image and clean up apt list for thin layers

- RUN apt-get update && apt-get install -y \
  git \
  && rm -rf /var/lib/apt/lists/*

Dockerfile Instructions 3/6

Until now you never explicitly said what application to run when you started a container. When writing dockerfile you need to define this by using the CMD or ENTRYPOINT command. This sets the image's main command.

CMD

- CMD [“executable”, “param1”, “param2”...]
- CMD executable param1

ENTRYPOINT (exec form preferred)

- ENTRYPOINT [“executable”, “param1”, “param2”...]
- ENTRYPOINT executable param1

Docker CLI - Building Images

You can build you images from a Dockerfile by using the Docker CLI.

1. path to Dockerfile (Default is 'PATH/Dockerfile') Syntax:
   - -f path/to/Dockerfile
   - –file path/to/Dockerfile
2. name and tag of image Syntax:
   - -t name:tag
   - –tag name:tag
3. last option is the path to the build context (mostly .)

Example: Build a Dockerfile with buildcontext in current (power)shell directory

- docker build -t foo:bar -f path/to/Dockerfile .

Dockerfile - Exercise 1

1. go to directory dockerfile/Exercise1
2. create a new Dockerfile and use ubuntu as base image
3. install cowsay and lolcat in one layer (optional: clean up)
4. create an ENTRYPOINT in exec format executing */usr/games/cowsay* and *"Hello"* as first parameter
5. create a CMD in exec format with *"World"* (will be second parameter to cowsay)
6. build your image named cowsay
7. run the image cowsay without parameter and with parameter (do not change the Dockerfile)
8. comment out the ENTRYPOINT and CMD
9. append a new CMD with shell format executing the following: */usr/games/cowsay "What does the cow say?"* | */usr/games/lolcat*
10. build your image named lolsay
11. run the image lolsay

Dockerfile Instructions 4/6

Until now you never added local files like prebuild executables or html/php sides to the images. Each of these commands adds a new delta-layer to the images. You can add files by using ADD and COPY.

Syntax (chown only works on linux containers):

- ADD [–chown=<user>:<group>] <src>... <dest>
- ADD [–chown=<user>:<group>] ["<src>",... "<dest>"]

Example: adds all files starting with "hom"

- ADD files* /somedir/

Syntax (chown only works on linux containers):

- COPY [–chown=<user>:<group>] <src>... <dest>
- COPY [–chown=<user>:<group>] ["<src>",... "<dest>"]

Example: ? is replaced with any single character, e.g., "home.txt"

- COPY hom?.txt /mydir/

Dockerfile Instructions 5/6

You need to explicitly tell docker in your dockerfiles which port are used. Only these ports can be published during docker run -p EXTERNALPORT:INTERNALPORT
Syntax:

- EXPOSE INTERNALPORT

Example: expose port 8081 and publish it to 8080

1. in dockerfile:
   - EXPOSE 8081
2. docker run command:
   - docker run -p 8080:8081 imagename

Dockerfile - Exercise 2

- go to directory dockerfile/Exercise2
- create a new Dockerfile
- use ubuntu as base image
- install apache2 (and cleanup)
- add html to /var/www/html
- copy etc/apache2/ports.conf to /etc/apache2/ports.conf
- copy etc/apache2/site-available/000-default.conf to etc/apache2/site-available/000-default.conf
- expose port 8080
- set entrypoint to start apache2 with *apachectl -D FOREGROUND*
- create image named ubuntu with the tag apache2
- run your created image, publish port to 80 (optional run it in detached mode)
- check if your webside is locally reachable

Dockerfile Outlook

There are many more Dockerfile Instruction e.g.

- WORKDIR PATH
- ENV KEY VALUE
- ARGS
- VOLUME creates a volume at runtime
- many more. e.g.

Docker Compose - Motivation

- Tpyically, an application is not one service, but orchestration of multiple smaller isolated service units

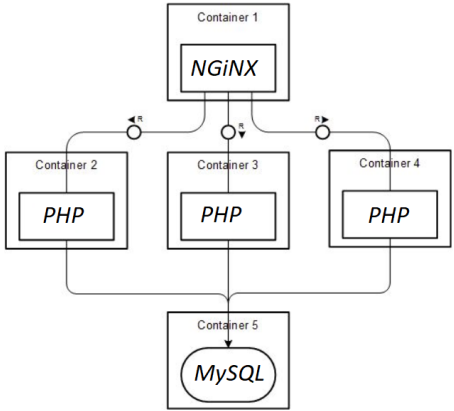

Figure 10: *NGinx, PHP, MySQL Service-Stack*

Docker Compose - Motivation

Multi-container apps are a hassle

- Until now:
- pull images, build dockerfiles
- create networks
- create container with long configuration lines (volumes, publish, networks)
- naming is global
- container logs for each container by name or id
- for a lot of containers!!

Multi-container apps are a hassle

- docker-compose up does everything automatically (after configuration)

Docker Compose File - Overview

- must be named docker-compose.yml or docker-compose.yaml
- containers become services
- yaml file (use no tabulator, space only)

```yaml
version: '3'

services:
  httpserver:
    image: httpd
    ports:
      - 80:80
    volumes:
      - .:/code
      - logvolume01:/var/log

  customservice:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
```

Docker Compose CLI

There exists a wide range of docker-compose cli commands. See docker-compose cli overview. Most important commands:
start services in foreground:

- docker-compose up

start services detached:

- docker-compose up -d

show service(s) log (console output):

- docker-compose logs [servicename]

stop services (force deletes running container)

- docker-compose rm [–force] | [–stop]

Docker Compose File - Ports

publish becomes ports with a list of port mappings.

```
3  services:
4    httpserver:
5      image: httpd
6      ports:
7        - 80:80
```

Figure 12: *Example: publish port 80 to 80*

Docker Compose File - Volumes

Local volume mapping stays volumes with a list of volume mappings. Volumes are listed at the end of the yml-file
Example: local mapping and with volume logvolume01

```
8   volumes:
9       - .:/code
10      - logvolume01:/var/log
```

Figure 13: *Example: Mapping of . to /db, logvolume01 to /var/log*

Example: volume definition of logvolume01

```
23  volumes:
24      logvolume01:
```

Figure 14: *Example: volumes defintions*

Docker Compose File - Build Images with Docker Compose

Add a build command inside service. Image's name and tag is generated automatically by the service's name. The context and path to the dockerfile can be set by key-value

```
12  customservice:
13    build:
14        context: .
15        dockerfile: Dockerfile
```

Figure 15: *Example: build with context and dockerfile*

Create the images then with:

- docker-compose build

Docker Compose File - Environment Variables

Add a environment command inside service with a list of environment key-values
Example: docker run -e Foo=Bar ...

```
16    environment:
17      - Foo=Bar
```

Figure 16: *Example: environment varible key=foo and value=bar*

Exercise Docker Compose - PHP Application 1/2

First Service: PHP with Apache2

- Dockerfile
  - expose port 8080
  - create dockerfile from php:apache2
  - copy scripts to /var/www/html
- publish port 8080
- create a new service named phpapache2

Second Service: Database with MySQL

- create a new service named db_mysql
- map volume
- define environment variables user passwd

Docker Compose File - Command Translation

Add the networks: keyword at the end of the yml-file. Add a list of networks.
Example: Usage of networks in service

```
4   httpserver:
5     image: httpd
6     ports:
7       - 80:80
8     volumes:
9       - .:/code
10      - logvolume01:/var/log
11    networks:
12      - example_network
13
```

Figure 17: *Example: usage of network example_network in service httpserver*

Example: network defintion of example_network

```
26  networks:
27    example_network:
```

Figure 18: *Example: networks definition of example_network*

Exercise Docker Compose - PHP Application with NGINX 2/2

User your artifact from the last exercise First Service: NGINX

- create a new service named nginx_reverse_proxy
- expose port 80
- volume conf.d/default.conf into
- depend on php_apache2
- join an appropriate network

Edit Service: PHP with Apache2

- remove the publish

Second Service: Database with MySQL

- create a new service named db_mysql
- map volume etc

Networks

- the Database service shall only be reachable by the php_apache2 service, so put them in the same network
- the php_apache2 shall be rechable from the nginx reverse proxy but not from the outside
- nginx shall be accesible over port 80 from the outside

Questions ?

made by Gabriel Nikol
https://github.com/S3ler/DockerByExample