

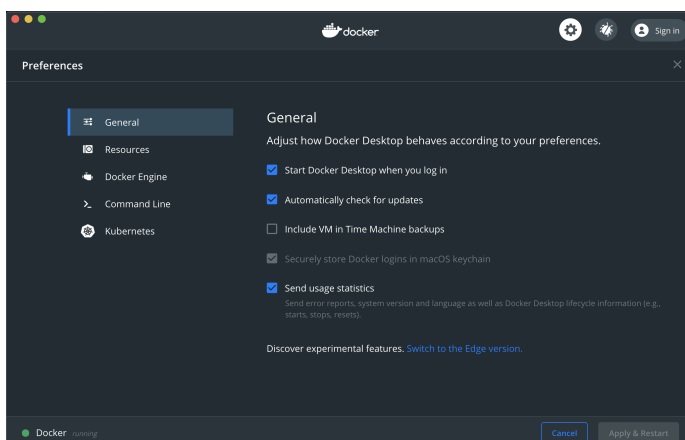
Docker einrichten

Die Software Docker hat das Unternehmen Docker Inc. ursprünglich in zwei Varianten angeboten: *Docker Enterprise* und *Docker Community*. Nachdem die Enterprise-Sparte an Mirantis verkauft wurde, ist nur noch die Open-Source-Software *Docker Community* interessant.

Für Windows und macOS

Es gibt die Software als Desktop-Version mit grafischer Oberfläche zum Download. Wählen Sie am besten die Stable-Variante, Edge enthält Beta-Funktionen:

- [Download bei Docker Inc.](#)



Für Linux und Linux-Server

Linux-Desktop-Nutzer müssen ohne grafische Oberfläche auskommen (was nicht schwerfällt). Auf dem Server gibt es ohnehin keinen Grund für grafische Oberflächen.

Die Paketquellen der gängigen Linux-Distributionen sind keine geeignete Anlaufstelle. Die Container-Welt ist zu schnelllebig.

Es gibt Paketquellen (.deb und .rpm) [direkt von Docker Inc.](#) Der schnellste Weg zur Docker-Engine auf dem Server ist das Installations-Skript:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Das Skript lädt Docker herunter und installiert es. Anschließend sollten Sie den aktuellen Benutzer zur Gruppe `docker` hinzufügen, damit Sie auch als normaler Benutzer auf den Docker-Daemon und die laufenden Container einwirken können:

```
sudo usermod -aG docker $USER
```

Docker-Compose

Unter Windows und macOS müssen Sie Docker-Compose nicht separat installieren, es ist Teil der Desktop-Umgebung.

Unter Linux gibt es ebenfalls einen Zweizeiler:

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.26.0/docker-  
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose  
  
sudo chmod +x /usr/local/bin/docker-compose
```

Lab 1: Erste Schritte mit Docker

- Docker bedient man auf der Kommandozeile. Der erste Befehl:

```
docker version
```

- Alle laufenden Container zeigt

```
docker ps
```

1. Der erste Container

Jeder Container besteht aus einem Image. Es enthält das Dateisystem mit allen Dateien, die für den Betrieb gebraucht werden.

Einen Container starten Sie mit `docker run <Name des Images>`. Ein Abbild für Linux-Experimente heißt `busybox`.

- Starten Sie Ihren ersten Container mit

```
docker run -it busybox
```

Der Parameter `-it` weist Docker an, dass der Container für eine interaktive Sitzung gestartet wird.

Wenn Sie genau hinsehen, bemerken Sie, dass Sie sich nicht mehr in der Konsole des Gastgebers befinden:

```
root@test1:~# docker run busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
76df9210b28c: Pull complete
Digest: sha256:95cf004f559831017cdf4628aaf1bb30133677be8702a8c5f2994629f637a209
Status: Downloaded newer image for busybox:latest
/ #
```

- Sehen Sie sich im Dateisystem um.

- Schauen Sie sich die Netzwerkkonfiguration mit `ifconfig` an
- Versuchen Sie, eine Verbindung ins Internet herzustellen (z.B. `ping heise.de`)
- Verlassen Sie den Container mit `exit`

2. Der erste Server

Kommandozeilenwerkzeuge wie `busybox` sind nicht das natürliche Habitat für Docker-Container. Es ist Zeit für den ersten Serverdienst, einen kleinen Nginx-Webserver. Ein Image für Nginx heißt schlicht: `nginx`.

- Fahren Sie Ihren Server hoch:

```
docker run -p 80:80 nginx
```

Der neue Parameter `-p 80:80` richtet eine Portweiterleitung ein. Anfragen an Port 80 des Gastgebers landen am Port 80 des Containers.

Das Ergebnis sieht in etwa so aus:

```
root@test1:~# docker run -p 80:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
8559a31e96f4: Pull complete
8d69e59170f7: Pull complete
3f9f1ec1d262: Pull complete
d1f5ff4f210d: Pull complete
1e22bfa8652e: Pull complete
Digest: sha256:21f32f6c08406306d822a0e6e8b7dc81f53f336570e852e25fbe1e3e3d0d0133
Status: Downloaded newer image for nginx:latest
```

- Öffnen Sie die Adresse Ihres Test-Servers im Browser. Das Ergebnis sollte so aussehen:

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

- Beobachten Sie währenddessen die Kommandozeile. Sie zeigt in Echtzeit die Log-Ausgaben.
- Beenden Sie den Server mit Strg+C.

2.2 Server im Hintergrund-Modus

Ein richtiger Server soll laufen, ohne dass eine Kommandozeile offen ist. Dafür gibt es den detached-Modus:

```
docker run -d -p 80:80 nginx
```

Ausgabe ist nur eine Zeile, die ausgewürfelte ID. Prüfen Sie im Browser, dass der Server läuft.

Dass der Container existiert, sehen Sie mit

```
docker ps
```

In der ersten Spalte sehen Sie die ID, in der letzten den Namen (Terminal auf die Breite aufziehen).

Wenn Sie im detached-Modus sehen wollen, was im Container passiert ist, öffnen Sie die Logs mit:

```
docker logs <id>
```

- Stoppen Sie Ihren Webserver:

```
docker stop <ID>
```

Der Browser findet die Seite nicht mehr, `docker ps` bleibt auch leer.

Praxistipp: Angehalten ist nicht weg

Ein gestoppter Container ist nicht weg. Sie finden ihn mit `docker ps -a`. Erst mit einem `docker rm` ist er endgültig weg. Alle gestoppten Container entsorgen Sie mit:

```
docker rm $(docker ps -aq)
```

Ein Befehl, den man leider garantiert vergisst und öfter braucht.

2.3 Arbeiten im Container

Die Willkommen-Seite soll einer eigenen Seite weichen. Starten Sie eine neue Instanz des Nginx-Servers:

```
docker run -d -p 80:80 --name test nginx
```

Durch den Parameter `--name test` hat er direkt den Namen `test` bekommen. Um im Container etwas zu ändern, können Sie in diesen hineinspringen:

```
docker exec -it test bash
```

Der Befehl `docker exec` führt ein Programm im Container aus. Das Programm in diesem Fall heißt `bash`. Nicht jedes Container-Abbild hat `bash` installiert. Nutzen Sie dann `sh`.

Die Beispiel-Datei liegt im Pfad `"/usr/share/nginx/html"`. Navigieren Sie dorthin:

```
cd /usr/share/nginx/html
```

Zum Bearbeiten soll `nano` zum Einsatz kommen. Das Programm ist aber noch nicht installiert. Betrachten Sie den Container als weitgehend leeren Linux-Server. Der Nginx-Container basiert auf Debian Buster – das hat nichts mit dem System zu tun, auf dem Docker läuft! Die Demo-Server laufen mit Ubuntu.

In diesem Container ist `apt` der Paketmanager des Vertrauens. Installieren Sie `nano` und öffnen Sie die Webseite mit:

```
apt update
apt install nano
nano index.html
```

Nehmen Sie eine Änderung an der Datei vor und speichern Sie mit "Strg+O" und "Enter", verlassen Sie `nano` mit "Strg+X". Prüfen Sie im Browser den Erfolg Ihrer Änderung.

Mit `exit` verlassen Sie den Container wieder.

Manipulationen im Container

Manipulationen dieser Art im Container sind nur während der Einrichtung und bei der Fehlersuche sinnvoll. An einem Container in Produktion nimmt man keine Änderungen vor – was Sie dort ändern, ist flüchtig: Wird der Container ausgetauscht, verschwindet die Änderung. In einem der nächsten Kapitel erfahren Sie, wie man für dauerhafte Änderungen vorgehen muss.

3. Bleibende Daten (Volumes)

Was im Container geschrieben wird, verschwindet nach dem Löschen. Das ist für Nutzdaten und Konfigurationen hinderlich. Damit etwas das Entsorgen eines Containers übersteht, müssen Sie ein sogenanntes Volume in den Container hineinreichen. Das ist im einfachsten Fall eine Datei oder ein Ordner auf dem Gastgeber.

Entsorgen Sie zunächst den Nginx-Container. Legen Sie dann auf dem Linux-Server im aktuellen Verzeichnis (z.B. dem Home-Verzeichnis) die Datei `index.html` mit Inhalt an, zum Beispiel mit:

```
echo "Container in der Praxis" > index.html
```

Diese Datei soll jetzt direkt in den Container weitergereicht werden:

```
docker run -d -p 80:80 -v ${PWD}/index.html:/usr/share/nginx/html/index.html --  
name testvolumes nginx
```

Für die Weiterleitung ist der Parameter `-v` verantwortlich. Vor dem Doppelpunkt steht der Pfad zur Datei oder zum Ordner auf der lokalen Maschine, nach dem Doppelpunkt das Ziel im Container.

Hinweis: Die Volumes müssen als absoluter Pfad angegeben werden. Daher kommt `${PWD}` zum Einsatz.

4. Zusammenfassung

Sie haben gesehen, wie Sie Container starten, stoppen, löschen, Ports und Volumes in den Container reichen.

Die wichtigsten Befehle in der Übersicht:

Befehl	Funktion	Beispiel
<code>docker ps</code>	zeigt alle Container an	<code>docker ps -a</code>
<code>docker run</code>	führt einen Container aus	<code>docker run -it busybox sh</code>
<code>docker stop <id></code>	beendet einen Container	<code>docker rm testserver</code>
<code>docker rm <id></code>	entfernt einen beendeten Container	<code>docker rm testserver</code>
<code>docker rm \$(docker ps -aq)</code>	alle gestoppten Container entsorgen	

Lab 2: Arbeiten mit Images

Im ersten Teil haben Sie bereits zwei Images benutzt: `busybox` und `nginx`. Aber wo kommen diese überhaupt her?

Der Reihe nach. Ursprung eines Images ist ein Rezept, das *Dockerfile*. Ein kleines Beispiel:

```
FROM debian:buster-slim
RUN apt-get update && apt-get install -y iputils-ping
CMD ping -c 4 heise.de
```

In der ersten Zeile wird das Basis-Image definiert. Damit bekommt der Container das Dateisystem eines leeren Debian-Systems. Anschließend können beliebige Linux-Operationen ausgeführt werden, zum Beispiel Installationen per Paketmanager.

Am Ende des Dockerfile folgt das Schlüsselwort `CMD`. Es definiert den Prozess, der im Container läuft. Erinnern Sie sich an den Grundsatz:

"Ein Container, ein Prozess"

Solange dieser Prozess läuft, existiert der Container.

Schlüsselwort	Funktion	Beispiel
FROM	definiert das Basis-Image	FROM <code>debian:buster-slim</code>
RUN	führt einen Befehl während des Bauprozesses aus	RUN <code>apt update && apt install ping</code>
COPY	kopiert Dateien und Ordner in den Container	COPY <code>settings.conf /etc/test/settings.conf</code>
CMD	definiert den Prozess, der im Container laufen soll	CMD <code>ping heise.de</code>

1. Das erste eigene Image

Legen Sie eine Datei `Dockerfile` an und kopieren Sie den Inhalt des obenstehenden Beispiels in die Datei. Dann kann der Bau beginnen.

- Starten Sie den Bau mit:

```
docker build .
```

Der Punkt am Ende sagt: "Suche ein Dockerfile im aktuellen Verzeichnis". Sie können den Bauprozess live verfolgen. Am Ende erfahren Sie die ID, die sich der Docker-Daemon ausgedacht hat:

```
Successfully built 72a2c8964f90
```

Kopieren Sie Ihre ID heraus und starten Sie den Container:

```
docker run <id>
```

Der Container hat eine kurze Lebenszeit. Nach vier Pings stoppt der Prozess und Docker stoppt den Container.

1.2 Images verwalten

Alle Images auf dem System sehen Sie mit

```
docker image ls
```

Dort finden Sie auch Ihr frisch gebautes Image. Sie können ihm einen sprechenden Namen geben:

```
docker tag <id> simpleping:latest
```

Danach können Sie das Image unter seinem Namen ansprechen:

```
docker run simpleping
```

Das Taggen können Sie mit dem Parameter `-t` auch direkt mit dem Bauprozess verbinden:

```
docker build -t simpleping .
```

Zum Löschen eines Images nutzen Sie:

```
docker image rm <id oder name>
```

Der Docker-Daemon wird sich beschweren, solange noch ein Container auf Basis des Images existiert (auch gestoppte Container zählen dazu). Nutzen Sie [Ihr Wissen aus Lab 1](#), um die gestoppten Container zu identifizieren und alle Images zu entsorgen.

Images aufräumen

Ungenutzte Images kosten Platz auf der Platte. Sie müssen sie nicht per Hand einzeln löschen. Für diesen Zweck gibt es:

```
docker image prune
```

Damit ein gestoppter Container nicht herumliegt und sofort abgeräumt wird, gibt es den Parameter `--rm`:

```
docker run --rm simpleping
```

2. Gute Images

Gute Images erkennen

1. Überprüfen Sie, ob es ein offizielles Image für die Software gibt (ohne / im Namen).
2. Wählen Sie einen geeigneten Tag. Abgespeckte Images sind kleiner und bieten weniger Angriffsfläche. Alpine ist oft eine gute Alternative.
3. Gibt es kein offizielles Image, schauen Sie sich direkt beim Entwickler der Software um. Einige bieten eigene Docker-Images an.
4. Gibt es keine Images der Entwickler, suchen Sie im Hub oder in der Suchmaschine.
5. Abrufzahlen, Dokumentation und Sterne im Docker Hub geben einen Hinweis, ob das Projekt aktiv benutzt wird.
6. Kennen Sie den Ersteller nicht, bevorzugen Sie Images mit automatischen Builds und schauen Sie ins Dockerfile.

Lab 3: Komplexe Zusammenstellungen mit Docker-Compose

Bisher haben Sie Docker-Container mit `docker run` gestartet. Schon in den ersten Beispielen, in denen Volumes und Ports zum Einsatz kamen, wurden die Befehle lang und unübersichtlich.

Ein Befehl wie

```
docker run \  
-p 80:80 \  
-p 443:443 \  
-v /path/to/file:/etc/file \  
-v /config.conf:/etc/config.conf \  
-e DATABASE='foo' \  
-e DATABASE_HOST='bar' \  
-e DATABASE_PASSWORD='secret' \  
--name testserver nginx:latest
```

ist auf Dauer keine Lösung. Das Grundprinzip "Ein Prozess, ein Container" hat zur Folge, dass man für die meisten Anwendungen mehr als einen Container braucht. Zum Glück gibt es Docker-Compose – das ist der Weg, wie Sie zukünftig mit Docker arbeiten möchten.

Prüfen Sie, ob Docker-Compose installiert ist:

```
docker-compose version
```

Grundlagen: YAML

Docker-Compose arbeitet mit YAML-Dateien. YAML ist eine Auszeichnungssprache für Objekte und in der Cloud-Native-Welt. YAML arbeitet mit Einrückungen zur Auszeichnung von Verschachtelungen:

```
auto:  
  name: Golf  
  hersteller:  
    name: Volkswagen  
    ort:
```

```
    name: Wolfsburg
    plz: 38440
tags:
  - auto
  - kfz
```

2. WordPress in Containern

Ein gutes Beispiel ist ein WordPress-Blog. WordPress braucht:

- einen Container mit dem Webserver (Apache)
- einen Datenbankserver (MariaDB oder MySQL)
- der Webserver muss mit seiner Datenbank sprechen können.
- Port 80 des Containers muss auf der externen Netzwerkkarte lauschen

Legen Sie eine Datei mit dem Namen "docker-compose.yml" an. Das ist der Standardname, den Docker-Compose erwartet. Die folgende Zusammenstellung beschreibt eine simple WordPress-Zusammenstellung:

```
version: '3.8'

services:
  db:
    image: mysql:5.7
    volumes:
      - ./data/db:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: secret123root
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: 12345wordpress54321

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "80:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
```

```
WORDPRESS_DB_USER: wordpress
WORDPRESS_DB_PASSWORD: 12345wordpress54321
working_dir: /var/www/html
volumes:
- ./data/wordpress:/var/www/html/wp-content
```

Wenn die Datei auf dem Server liegt, können Sie die Zusammenstellung starten:

```
docker-compose up
```

Jetzt brauchen Sie etwas Geduld: Docker lädt die unhandlichen Images aus dem Hub herunter und fährt die Datenbank hoch. Das dauert, weil die Datenbank noch leer ist – MySQL richtet sich erstmal ein.

Wie bei `docker run` gibt es auch bei Docker-Compose einen detached-Modus. Ein fertiges Setup würden Sie mit starten mit:

```
docker-compose up -d
```

Das Gegenteil von `up` lautet (logisch):

```
docker-compose down
```

Und auch andere bekannte Befehle finden Sie wieder:

```
docker-compose ps
docker-compose exec -it <service> sh
```

2.2 Umgebungsvariablen

Die Konfiguration gehört nicht in die Compose-Datei! Sie sollten die Compose-Datei so flexibel wie möglich halten. Dann ist es leicht, eine Dev-, Staging- und Prod-Umgebung aus demselben File zu erzeugen.

Docker-Compose arbeitet dafür mit Ersetzungen:

```
wordpress:
  depends_on:
  - db
```

```
image: "wordpress:${TAG}"
ports:
  - "80:80"
restart: always
environment:
  WORDPRESS_DB_HOST: db:3306
  WORDPRESS_DB_USER: ${DB_USER}
  WORDPRESS_DB_PASSWORD: ${DB_PASS}
working_dir: /var/www/html
volumes:
  - ./data/wordpress:/var/www/html/wp-content
```

Diese Ersetzungen kommen aus einer Datei mit dem Namen `.env`, die sich neben dem Compose-File befinden muss:

```
TAG=latest
DB_USER=wordpress
DB_PASS=secret
```

3: Docker-Compose macht vieles einfacher:

- Die Struktur wird schnell sichtbar
- Mit `depends_on` bestimmen Sie (grob), welcher Container zuerst startet
- Pfadangaben funktionieren relativ vom Pfad des Docker-Compose-Files

Infrastruktur ist Code

Um dem Ziel einen wesentlichen Schritt näher zu kommen, eine reproduzierbare Umgebung zu bauen, sollten Sie Ihre Docker-Compose-Files von Anfang an in eine Versionsverwaltung legen (Git). Und auch Dockerfiles von eigenen Images gehören in die Versionsverwaltung. Im nächsten Schritt wird dann automatisch gebaut (Continuous Integration).

Guter Stil ist es, eine vorbereitete `.env-example` bereitzulegen. Die kann ein Nutzer des Repos umbenennen (`mv .env-example .env`) und direkt starten.

Lab 4: Ein gutes Image, automatisch gebaut

Zur Erinnerung: Bisher haben wir Images wie folgt gebaut:

Irgendwo auf der Festplatte des Servers haben wir ein `Dockerfile` angelegt und dieses per `docker build .` zu einem Image zusammengebaut. Ein solches Image könnte man mit `docker push <id>` in den Docker-Hub kopieren. Doch so flüchtig soll unser Produktivsystem nicht aufgebaut werden. In diesem Lab soll einmal systematisch eine Automation eingerichtet werden.

Voraussetzung um die Anleitung nachzuvollziehen, ist ein Account bei Docker und einer bei GitHub (letzteren dürften die meisten Admins und Entwickler haben). Hier geht es zum Anlegen der Accounts:

- [Account bei GitHub anlegen](#)
- [Account bei Docker anlegen](#)

1. Das Repo für den Code

Als simples Beispiel soll ein Container mit der Open-Source-Dokumentationssoftware "MkDocs" gebaut werden. In diesen Container sollen die Inhalte der Dokumentation (die ebenfalls im Repository liegen) eingebackten werden. Der fertige Container soll also ohne Volumes einfach laufen und die komplette Doku ausliefern. Mit jedem Push ins Repo soll ein aktuelles Image erzeugt werden.

Die Reise beginnt bei einem GitHub-Repository für den Code. Legen Sie ein solches an: <https://github.com/new> Das Repo darf grundsätzlich `public` oder `private` sein. Für das Beispiel nehmen Sie zunächst ein öffentliches Repo.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)


Repository template

Start your repository with a template repository's contents.

No template ▾

Owner

Repository name *

 jamct ▾ / demo-docs ✓

Great repository names are short and memorable. Need inspiration? How about [laughing-meme?](#)

Description (optional)

An mkdocs demo

☒  **Public**

Anyone on the the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾

Add a license: **None** ▾



Create repository

Das Repo für dieses Beispiel heißt `demo-docs`. Wenn Sie es anders nennen, müssen Sie später ggf. den Namen ändern.

1.2 Das Dockerfile

Dieses Lab soll auch die Denkweise veranschaulichen, wie man an das Bauen eines Images herangeht:

- Gibt es ein MkDocs-Image (am besten der Entwickler) -> nein
- Gibt es Drittanbieter-Images -> eher nein

Also muss ein eigenes Image her. Dazu lohnt ein Blick in die [Doku von MkDocs](#). Was dort steht, muss als Rezept in ein Dockerfile gelangen:

```
pip install --upgrade pip
pip install mkdocs
mkdocs serve
```

MkDocs läuft also mit Python und kommt mit dem Python-Paketmanager Pip auf eine Maschine. Als Basis-Image sollten wir uns nach Python umsehen. Die gute Nachricht: Es gibt ein offizielles Image und auch eins auf Alpine-Basis. Damit können wir ein `Dockerfile` im Repo anlegen:

```
FROM python:3-alpine
RUN pip install --upgrade pip && pip install mkdocs
EXPOSE 8080
CMD ["mkdocs", "serve"]
```

Was jetzt folgt, ist Versuch und Irrtum. Versuchen wir, dieses Dockerfile zu bauen:

```
docker build .
```

Der Bauvorgang beginnt vielversprechend, scheitert aber nach 30 Sekunden mit viel rotem Output:

```
[...]
building 'regex._regex' extension
creating build/temp.linux-x86_64-3.8
creating build/temp.linux-x86_64-3.8/regex_3
gcc -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O3 -Wall -
DTHREAD_STACK_SIZE=0x1000000 -fPIC -I/usr/local/include/python3.8 -c regex_3/
_regex.c -o build/temp.linux-x86_64-3.8/regex_3/_regex.o
unable to execute 'gcc': No such file or directory
error: command 'gcc' failed with exit status 1
-----
ERROR: Command errored out with exit status 1: /usr/local/bin/python -u -c
'import sys, setuptools, tokenize; sys.argv[0] = ''''/tmp/pip-install-iygh9d3r/
regex/setup.py''''; __file__ = ''''/tmp/pip-install-iygh9d3r/regex/
setup.py'''';f=getattr(tokenize, ''''open'''', open)
(__file__);code=f.read().replace(''''\n''',
'''\n''');f.close();exec(compile(code, __file__, ''''exec'''))' install --
record /tmp/pip-record-rzdt9kz/install-record.txt --single-version-externally-
managed --compile --install-headers /usr/local/include/python3.8/regex Check the
logs for full command output.
The command '/bin/sh -c pip install --upgrade pip && pip install mkdocs'
returned a non-zero code: 1
```

Jetzt gilt es, den Fehler zu finden und Google zu befragen. Der entscheidende Hinweis: `'gcc': No such file or directory`. Der Python-Paketmanager braucht also den C-Compiler `gcc` für irgendein Paket. Ein Detail, das uns die Doku von MkDocs verschwieg. Auf einem klassischen Server würden wir das nachinstallieren und dann vergessen. Beim Dockern wird man gezwungen, das anständig reproduzierbar zu bauen. Gcc gibt es per Alpine-Paketquellen im Paket `build-base` (sagte die Suchmaschine). Nächster Versuch, neu ist Zeile 2:

```
FROM python:3-alpine
RUN apk add build-base
RUN pip install --upgrade pip && pip install mkdocs
EXPOSE 8080
CMD ["mkdocs", "serve"]
```

Kopieren Sie diesen Schnipsel und lassen Sie Docker bauen.



Schichten sparen

Während das läuft, eine Anmerkung zum `&&`. Damit kann ich unter Linux zwei Befehle aneinander kleben. Ich könnte auch für jeden Befehl eine Zeile mit `RUN` ins Dockerfile schreiben. Docker erzeugt aber für jede Zeile eine Schicht. Das sollte ich vermeiden, wo ich nur kann. Man könnte auch überlegen, das `apk add` in die Zeile zu setzen. Das spart beim Bauen später Zeit und beim Herunterladen ebenso.

Docker hat unterdessen gebaut und ein Image erzeugt, das man mal starten könnte.

```
docker run -p 80:8080 <id>
```

Der Container fährt hoch, stürzt aber sofort ab. Ab jetzt ist MkDocs aber lauffähig, es fehlt: der Inhalt.

Die Doku soll ja direkt im Repo liegen – dafür schaffen wir im Repo den Ordner `mkdocs`. Zunächst hätte MkDocs gern eine YAML-Datei mit Metadaten. Legen Sie die `mkdocs/mkdocs.yml` an:

```
site_name: Test-Doku
dev_addr: "0.0.0.0:8080"
site_author: "Ihr Name"

markdown_extensions:
  - attr_list
  - admonition
nav:
  - Willkommen: index.md
```

Außerdem brauchen Sie einen weiteren Unterordner `docs` (eine Vorgabe von MkDocs) und darin eine `index.md` mit etwas Markdown-Text:

```
# Test

Das ist Markdown

* Test
* Test
```

Die Struktur des Repos sieht jetzt so aus:


```
Dockerfile
mkdocs
  mkdocs.yml
  docs
    index.md
```

Der ganze Ordner `mkdocs` soll beim Bau in den Container kopiert werden. Achtung: Das ist kein Volume! Der Ordner wird einmalig beim Bau kopiert, nicht zur Laufzeit.

```
FROM python:3-alpine
RUN apk add build-base
COPY ./mkdocs/ /mkdocs/
WORKDIR /mkdocs/
RUN pip install --upgrade pip && pip install mkdocs
EXPOSE 8080
CMD ["mkdocs", "serve"]
```

3. Die Automation

Zwei Registries bieten sich an: Die öffentliche Registry im Docker-Hub und die Paket-Registry von GitHub. Erstere ist schnell in Betrieb genommen. Klicken Sie im Repo auf "Packages" und wählen Sie dort "Docker":



Get started with GitHub Packages

Package type: **Docker** ▾ [Learn more](#)

```
# Step 1: Authenticate
$ cat ~/GH_TOKEN.txt | docker login docker.pkg.github.com -u jamct --password-stdin

# Step 2: Tag
$ docker tag IMAGE_ID docker.pkg.github.com/jamct/demo-docs/IMAGE_NAME:VERSION

# Step 3: Publish
$ docker push docker.pkg.github.com/jamct/demo-docs/IMAGE_NAME:VERSION
```

GitHub bringt seit fast einem Jahr eine mächtige CI/DCD-Lösung mit. Sie reagiert z.B. auf Pushes ins Repo. Um Sie einzurichten, erstellen Sie die Datei `.github/workflows/docker.yml`. Passen Sie den Namen Ihres Repos an:

```
name: Build docs

on:
  push:
    branches: [master]

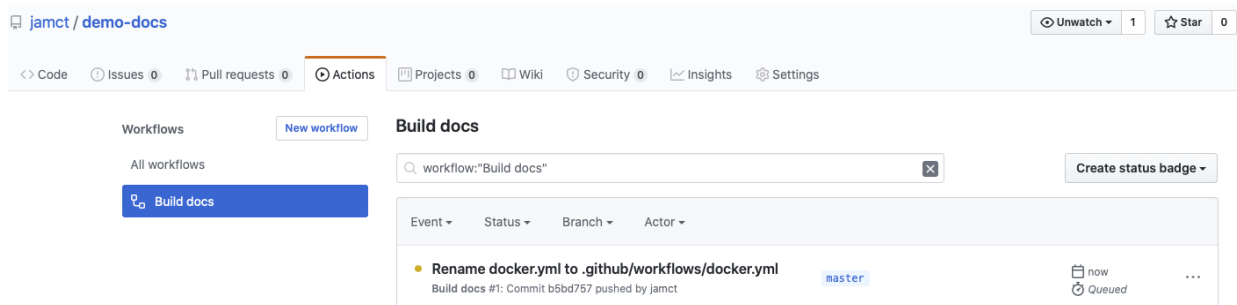
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: docker/build-push-action@v1
        with:
          dockerfile: Dockerfile
          path: .
          username: jamct
          password: ${ secrets.GITHUB_TOKEN }
          repository: jamct/demo-docs/mkdocs
```

```
registry: docker.pkg.github.com
tags: latest
```

Ein paar Anmerkungen zum (groben) Verständnis: Die GitHub-Registry orientiert sich an den Repos. Innerhalb eines Repos kann es mehrere Images geben. Daher ist der Name des Containers auch länger: `jamct/demo-docs/mkdocs:latest`.

Innerhalb des Repos gibt es ein Token (`{{ secrets.GITHUB_TOKEN }}`), mit dem sich der Actions-Worker an der Registry anmeldet.

Wenn Sie die Datei speichern, beginnt sofort die Action. Das dürfen Sie live verfolgen über den Reiter "Actions".



Danach sehen Sie das fertige Paket unter "1 package" im Reiter "Code".

Kopieren Sie den ganzen Pull-Befehl mit dem Namen heraus und fügen ihn im Server ein, zum Beispiel:

```
docker pull docker.pkg.github.com/jamct/demo-docs/mkdocs:latest
```

Docker beklagt sich, dass eine Anmeldung nötig ist (aktuell auch bei öffentlichen Images, GitHub ist da dran). Zum Anmelden sollten Sie nicht Ihr Kennwort, sondern ein Token nehmen. Das generieren Sie unter der Adresse

<https://github.com/settings/tokens>

Angehakt sein müssen die Punkte `repo` und `read:packages`:

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

demo

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> write:packages	Upload packages to github package registry
<input checked="" type="checkbox"/> read:packages	Download packages from github package registry
<input type="checkbox"/> delete:packages	Delete packages from github package registry

Auf dem Server können Sie sich anmelden mit:

```
docker login docker.pkg.github.com
Username: <Username>
Password: <das Token>
```

Danach klappt auch

```
docker pull docker.pkg.github.com/jamct/demo-docs/mkdocs:latest
```

Und anschließend

```
docker run -p 80:8080 docker.pkg.github.com/jamct/demo-docs/mkdocs:latest
```

3.2 Image im Docker-Hub

(diesen Abschnitt empfehlen wir für das Selbststudium, das Prinzip ist sehr ähnlich):

- Melden Sie sich im Docker-Hub an
- Erstellen Sie mit "Create Repository" ein Repo und geben ihm einen Namen
- Erzeugen Sie ein Token unter <https://hub.docker.com/settings/security>
- Wechseln Sie zu GitHub und im Repo auf "Settings", dort auf "Secrets". Legen Sie das Secret `DOCKER_TOKEN` an und kopieren Sie das Token aus dem Docker-Hub hier hinein.
- Öffnen Sie die YAML-Datei `.github/workflows/docker.yml` und ergänzen Sie einen Block für den Docker-Hub:

```
name: Build docs

on:
  push:
    branches: [master]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: docker/build-push-action@v1
        with:
          dockerfile: Dockerfile
          path: .
          username: jamct
          password: ${ secrets.GITHUB_TOKEN }
          repository: jamct/demo-docs/mkdocs
          registry: docker.pkg.github.com
          tags: latest
      # neue Action für den Hub:
      - uses: docker/build-push-action@v1
```



```
with:
  dockerfile: Dockerfile
  path: .
  username: jamct
  password: ${ secrets.DOCKER_TOKEN }
  repository: jamct/mkdocs
  tags: latest
```

Die Action läuft sofort an und pusht in den Hub.

4. Entwickeln mit Automationen im Alltag

Auf der eigenen Maschine zu entwickeln, macht Spaß. Einer der Vorteile von Docker: Ich kann mir eine Entwicklungsumgebung zum Mitnehmen bauen. Dafür legt man sich eine `docker-compose-dev.yml` ins Repo:

```
version: "3.8"
services:
  docs:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8080:8080
    volumes:
      - ./mkdocs/:/mkdocs
```

Diese Datei baut das Image bei der Ausführung und hängt den "Code", also die Doku als Volume ein (ein Volume überschreibt den Inhalt, der dort schon liegt). Als Entwickler kann man dieses Setup lokal hochfahren, die Doku schreiben und das Ergebnis direkt betrachten. Ist man zufrieden, pusht man den Code zu GitHub und der Container wird mit Änderungen gebaut (auch diese Doku zum Workshop wurde so entwickelt).

Zum Hochfahren:

```
docker-compose -f docker-compose-dev.yml
```

Sicherheitstipp für Entwicklermaschinen

Wer im Unternehmensnetz Docker auf der eigenen Maschine laufen lässt und solche Entwicklungsumgebungen hochfährt, veröffentlicht sie auch auf der Netzwerkkarte des PCs. Sie ist also ggf. im Unternehmensnetz sichtbar (wenn keine Firewall auf der Maschine dazwischengeht). Das kann nützlich sein, um Kollegen in einem Meeting die Ergebnisse live vorzuführen.

Wenn man zum Experimentieren mit produktiven Daten arbeitet, sollte man den Zugriff aber auf das Loopback-Interface einschränken. Das geht mit:

```
- 127.0.0.1:80:8080
```

Lab 5: Nützliche Helfer

Container lösen alte Problem und schaffen einige neue. Gegen diese gibt es ein paar nützliche Helfer.

1. Container updaten

Eine häufige Frage: Wie schaffe ich es, dass meine Images immer aktuell sind? Wie reagiere ich automatisch auf Updates durch die Maintainer? Ich will mich ja nicht aktiv mit meinen Containern beschäftigen!

Für diesen Zweck haben Entwickler des Unternehmens V2Tec das Image "Watchtower" erfunden. Diesen Compose-Schnipsel kann man auf einer Docker-Maschine einbauen:

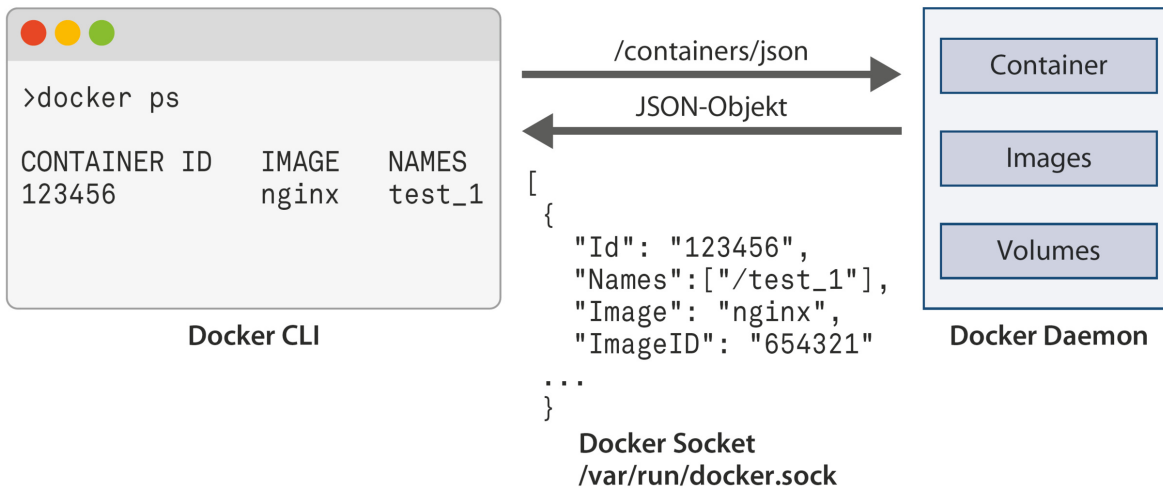
```
watchtower:
  image: containrrr/watchtower
  command: --cleanup --label-enable
  restart: always
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - ~/.docker/config.json:/config.json
```

Hier werden zwei erklärungsbedürftige Volumes verwendet:

- Die `config.json`. Sie liegt im Home-Verzeichnis und wird von Docker angelegt, wenn man sich an einer Registry anmeldet. Sie enthält die Base64-kodierten Token für die Registries. Damit kann Watchtower auch private Images updaten
- `/var/run/docker.sock` ist ein Unix-Socket und kann als Volume in den Container übergeben werden. Um diesen Socket zu verstehen, ist ein kleiner Exkurs nötig:

Docker Daemon und Docker-Kommandozeile

Das Kommandozeilentool fragt die Information beim Docker Daemon über ein REST-API ab, bekommt ein JSON-Objekt und formatiert es.



Die Docker-CLI und auch Docker-Compose sprechen mit diesem Socket. Wenn man den Socket in den Container übergibt, kann man Docker aus einem Container steuern (in diesem Fall: Images pullen und Container ersetzen). Das Volume

```
/var/run/docker.sock:/var/run/docker.sock
```

werden Sie immer dann finden, wenn Sie ein Image nutzen, das irgendwas mit Docker-Containern anstellt. Soll der Container nur auf dem Socket lesen, können Sie das einschränken:

```
/var/run/docker.sock:/var/run/docker.sock:ro
```

1.2. Container updaten lassen

In der Einstellung oben (`--label-enable`) schaut Watchtower minütlich für alle Container mit einem speziellen Label, ob es ein neues Image gibt. Das Label heißt

`com.centurylinklabs.watchtower.enable=true` (bei der Firma centurylinklabs haben die Entwickler früher mal gearbeitet und das nie geändert):

```
docs:
  image: nginx:alpine
  restart: always
  labels:
    - "com.centurylinklabs.watchtower.enable=true"
```

Aktualisieren die Nginx-Macher ihr Image, haben Sie es maximal eine Minute später im Produktivsystem. Gleiches gilt für eigene Container – vom Push bis zum Produktivsystem können weniger als 5 Minuten vergehen. So langsam fügen sich die Teile zusammen...

2. Grafische Oberflächen

Docker bedient man meist auf der Kommandozeile, oder – noch besser – gar nicht (weil Automationen den Job erledigen). Gerade auf Servern, die von vielen Nutzern zum Dockern benutzt werden, will man vielleicht aber mal überprüfen, was auf dem Host passiert und das ggf. auch Nicht-Entwicklern zeigen. Eine Kommandozeile ist da nicht der richtige Weg. Für diese Zielgruppe gibt es grafische Oberflächen. Eine gute Wahl ist die Open-Source-Oberfläche *Portainer*.

Zum Betrieb reicht der Compose-Schnipsel:

```
portainer:
  image: portainer/portainer
  ports:
    - 80:9000
  restart: always
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - ./portainer:/data
```

Container list

Containers

Portainer support

admin

my account

log out

Containers

Columns Settings

Start Stop Kill Restart Pause Resume Remove Add container

Search...

<input type="checkbox"/>	Name	State Filter	Quick actions	Stack	Image	Created	Published Ports	Ownership
<input type="checkbox"/>	vibrant_clarke	created		-	7a7a34ab7a3d	2020-06-16 17:43:27	-	administrators
<input type="checkbox"/>	root_portainer_1	running		root	portainer/portainer	2020-06-16 20:07:48	80:9000	administrators
<input type="checkbox"/>	root_db_1	running		root	mysql:5.7	2020-06-16 12:24:49	-	administrators
<input type="checkbox"/>	unruffled_sammet	stopped		-	docker.pkg.github.com/jamct/demo-docs/mkdocs:latest	2020-06-16 18:45:22	-	administrators
<input type="checkbox"/>	eager_hawking	stopped		-	docker.pkg.github.com/jamct/demo-docs/mkdocs:latest	2020-06-16 18:45:11	-	administrators
<input type="checkbox"/>	recursing_newton	stopped		-	mk	2020-06-16 18:19:10	-	administrators
<input type="checkbox"/>	infallible_bassi	stopped		-	835a93f9bd7b	2020-06-16 18:12:14	-	administrators
<input type="checkbox"/>	fervent_murdock	stopped		-	7a7a34ab7a3d	2020-06-16 18:07:38	-	administrators
<input type="checkbox"/>	relaxed_mcclintock	stopped		-	7a7a34ab7a3d	2020-06-16 17:43:47	-	administrators
<input type="checkbox"/>	keen_moore	stopped		-	8ecf5a48c789	2020-06-16 17:30:09	-	administrators

Items per page

10

<

1

2

>

Lab 5: HTTP-Routing mit Traefik

Bisher haben wir die Server-Anwendungen alle nacheinander gestartet. Es darf ja immer nur ein Prozess Port 80 belegen. So kann man eine Maschine natürlich nicht effizient nutzen. Man braucht einen Router, der eingehenden (HTTP-)Verkehr entgegennimmt und an den richtigen Container weitergibt. Nur der Router lauscht an Port 80 – und an Port 443, weil er ganz nebenbei noch für TLS und Zertifikatsbeschaffung zuständig ist.

Ein solcher Router, der wahrhaft "Cloud-Native" ist und sich gut auf einem Docker-Host macht, ist Træfik (oder Traefik). Wie alle verwendeten Tools ist auch der Open-Source, es gibt auch eine Enterprise-Edition mit Supportvertrag.

Traefik einrichten

Für die Traefik-Einrichtung sollten Sie am besten ein eigenes Repository mit dem Compose-File und einigen Config-Files anlegen. In die Compose-Datei stecken Sie ggf. noch Watchtower und Portainer. Ins Repo kann man dann noch ein Bash-Script legen, das Docker installiert. Dann hat man einen Ordner mit allen Rezepten für einen neuen Docker-Server.

Die Arbeit beginnt mit dem Compose-File `docker-compose.yml`:

```
version: "3.7"
services:
  traefik:
    image: traefik:v2.2
    command: --providers.docker
    restart: always
    ports:
      - 80:80
      - 443:443
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:rw
      - ./traefik/static.yml:/etc/traefik/traefik.yml
      - ./traefik/dynamic.yml:/etc/traefik/dynamic/dynamic.yml
      - ./traefik/acme.json:/etc/traefik/acme/acme.json
    networks:
      - router

# [...] Ergänzen Sie ggf. Watchtower, Portainer und andere Tools nach Belieben...
```

```
networks:
  router:
    external:
      name: router-network
```

Hier kommt erstmals ein neues Konzept zum Einsatz: Der Router wird an ein Netzwerk mit dem Namen `router` gebunden. Am Ende der Datei wird dieses Netzwerk bekannt gemacht: Docker-Compose soll ein extern erzeugtes Netzwerk, das draußen `router-network` heißt, innerhalb des Files als `router` bekannt machen.

Bevor Sie das Netzwerk nutzen können, muss es einmalig auf dem Docker-Host eingerichtet werden (auch das gehört in ein Installations-Skript):

```
docker network create --driver=bridge --attachable --internal=false router-
network
```

Jetzt fehlen noch drei Dateien, mit denen Traefik gesteuert wird. Legen Sie innerhalb des Repos (oder für die Experimentierphase zunächst auf dem Server) den Unterordner "traefik" an und dort die Datei "static.yml":

```
entryPoints:
  web:
    address: ":80"
  web-secure:
    address: ":443"

providers:
  docker:
    watch: true
  file:
    directory: /etc/traefik/dynamic
    watch: true
    filename: dynamic.yml

certificatesResolvers:
  default:
    acme:
      email: jam@ct.de
      storage: /etc/traefik/acme/acme.json
      httpChallenge:
        entryPoint: web
```


Traefik arbeitet wie folgt: Beim Start liest es die Static-Yaml-Datei ein. Darin stehen Dinge für den Grundbetrieb. Ändert man hier etwas, muss man Traefik neu starten (und den HTTP-Router im Produktivsystem will man nicht ständig neu starten).

Traefik soll Zertifikate bei Let's Encrypt bestellen und sie automatisch austauschen (nie wieder Arbeit mit Zertifikaten). In der Static-Datei ist nur eine Änderung nötig: Ersetzen Sie die Mailadresse durch Ihre. Die Mailadresse wird von Let's Encrypt benutzt, Sie per Mail zu warnen, wenn ein Zertifikat ausläuft und nicht automatisch erneuert wurde (nach meiner Erfahrung nie). Die Adresse wird nicht im Zertifikat veröffentlicht!

In der statischen Datei wird auf eine dynamische referenziert. Legen Sie `dynamic.yml` an:

```
http:
  routers:
    https-redirect:
      rule: "HostRegexp(`{any:.*}`)"
      middlewares:
        - https-redirect
      service: redirect-all
  middlewares:
    https-redirect:
      redirectScheme:
        scheme: https
  services:
    redirect-all:
      loadBalancer:
        servers:
          - url: ""
```

Diesen Block müssen Sie jetzt nicht Zeile für Zeile verstehen. Er kümmert sich darum, dass Anfragen an `http://` automatisch auf `https://` umgeleitet werden.

Jetzt fehlt noch eine letzte Datei, die am Anfang leer ist. Legen Sie die `./traefik/acme.json` einfach an und setzen Sie dann die Rechte:

```
chmod 600 acme.json
```

Dieser Schritt ist dringend notwendig. In dieser Datei speichert Traefik die Zertifikate.

Das Repository sieht jetzt so aus:

```
docker-compose.yml
traefik/
  acme.json
  static.yml
  dynamic.yml
```

Wenn Sie das Netzwerk angelegt haben, ist es an der Zeit, Traefik zu starten. Traefik steht auf `restart: always`, wird nach einem Neustart also zurückkommen.

Fahren Sie die Zusammenstellung hoch:

```
docker-compose up -d
```

Bisher passiert noch nichts. Wenn Sie eine beliebige Subdomain Ihres Servers im Browser öffnen, sehen Sie eine Fehlermeldung. Es gibt ja noch keinen Dienst, den Traefik veröffentlichen könnte.

Es ist an der Zeit, Ihr selbstgebautes Image zu veröffentlichen (mit TLS und unter einem schönen Hostnamen). Wechseln Sie dazu das Verzeichnis (Traefik arbeitet einfach still im Hintergrund und braucht unsere Aufmerksamkeit heute nicht mehr) und legen Sie eine neue Datei `docker-compose.yml` an:

```
version: "3.8"
services:
  docs:
    image: docker.pkg.github.com/jamct/demo-docs/docs:latest
    restart: always
    labels:
      - "traefik.http.routers.docs.rule=Host(`docs.00.liefer.it`)"
      - "traefik.http.routers.docs.tls.certResolver=default"
      - "traefik.http.routers.docs.tls=true"
      - "com.centurylinklabs.watchtower.enable=true"
    networks:
      - router
networks:
  router:
    external:
      name: router-network
```

Diesen Schnipsel können Sie als Referenz für weitere Projekte nutzen. Wichtig ist folgendes:

- Traefik wird über die Labels konfiguriert. Jeder Container bekommt einen `router`, danach folgt ein Name, der hier `docs` lautet.

- Für weitere Container müssen Sie einen anderen Router-Namen vergeben
- Der Router bekommt eine Regel, nach der Traefik den eingehenden Verkehr filtern soll. Hier ist die Regel auf die Subdomain gesetzt
- Andere Filter-Regeln erklärt die (recht gute) [Doku von Traefik](#)

Jetzt sollte auch klar werden, warum wir das Kunststück mit dem Netzwerk eingebaut haben. Wenn Sie ein Compose-Projekt mit der gesamten Infrastruktur und Traefik haben, kann das dauerhaft laufen. Davon unabhängig können Sie andere Projekte in eigenen Compose-Files hoch- und runterfahren.

Fahren Sie Ihre Doku mit Docker-Compose hoch. Am Anfang wird sich der Browser beklagen – Traefik fängt sofort an, ein Zertifikat passend zur Host-Regel zu bestellen. Das dauert maximal fünf Minuten.

Windows-Container unter Windows



Als Gastgeber für Windows Container brauchen Sie einen Windows Server (2016 oder 2019).

Installiert wird Docker per PowerShell:

```
Install-Module -Name DockerMsftProvider -Repository PSGallery -Force  
Install-Package -Name docker -ProviderName DockerMsftProvider
```

Anschließend neustarten:

```
Restart-Computer -Force
```

Die Befehle aus der Linux-Welt funktionieren ebenfalls. Prüfen Sie die Funktion des Docker Daemons:

```
docker ps
```

Ein Image finden

Unter Windows ist die Trennung zwischen Kernel und Userland nicht so sauber wie unter Linux. Deshalb muss man beim Wählen des Basis-Images genau aufpassen.

Grundsätzlich gibt es zwei (relevante) Basis-Images:

- mcr.microsoft.com/windows/nanoserver (~100 MByte)
- mcr.microsoft.com/windows/servercore (~2 GByte)

Entscheidend ist die Wahl des Tags. Der Tag muss zum Kernel des Gastgebers passen. Für den Server Core gibt es Tags, die an die 3-jährigen Zyklen des klassischen Windows Server angepasst sind:

- mcr.microsoft.com/windows/servercore:ltsc2019
- mcr.microsoft.com/windows/servercore:ltsc2016

Für den Nanoserver gibt es die nicht. Der Nanoserver wird im halbjährlichen Zyklus ausgeliefert:

- mcr.microsoft.com/windows/nanoserver:2004
- mcr.microsoft.com/windows/nanoserver:1903

Damit das läuft, brauchen Sie den halbjährlich aktualisierten Server. Die schlechte Nachricht: Dafür müssen Sie [Software Assurance bei Microsoft buchen](#). Es gibt aber einen Ausweg aus der misslichen Lage: Docker unter Windows kann Hyper-V zur Isolierung einsetzen:

```
docker run --isolation=hyperv mcr.microsoft.com/windows/nanoserver:1903
```

Die komplette Auflistung, bis wann welche Basis-Images unterstützt werden, [finden Sie bei Microsoft](#).

Nützliche Images aus der Windows-Welt

Bei Windows-Containern geht es meist um Projekte aus den Themenbereichen IIS und ASP.NET. Für solche Projekte müssen Sie nicht beim nackten Nanoserver beginnen. Stattdessen nutzen Sie Images, die Microsoft für diese Zwecke bereits vorbereitet hat:

- [IIS](#)

- [ASP.NET \(mit IIS\)](#)

Microsofts Sicht auf die Dinge

Eine Anekdote am Rande: In einem Gespräch mit Corey Sanders, Corporate Vice President für Azure bei Microsoft, ging es auch um die Bedeutung von Windows-Containern. Microsofts Meinung zum Thema:

"Windows-Container sind für Legacy-Anwendungen! Wenn Sie eine neue Anwendung planen, nehmen Sie Linux!"

Auch klassische Windows-Server-Software wie den *SQL Server* gibt es als Linux-Abbild, direkt von Microsoft:

- mcr.microsoft.com/mssql/server (basiert auf Ubuntu)

Ausblick und Abschluss

Zunächst ein Tipp zum Schluss: Um wirklich ins Thema einzusteigen, müssen Sie mit Docker arbeiten – viele Grundlagen waren Inhalt des Workshops, eigene Erfahrungen kann das nicht ersetzen. Am besten steigen Sie mit einem nicht-kritischen Teil ein. Ideen für Beispiel-Projekte:

- Eine Doku (Anregungen in diesem Repo)
- Eine Status-Seite für firmeninterne Dienste

Offene Fragen

Hier noch ein Nachtrag zu offenen Fragen nach der Veranstaltung:

Frage:

Wie kann ich das Dateisystem für den Container begrenzen, damit ein "verrückt gewordener" Container nicht den Host und damit alle anderen Container beeinflusst?

Antwort:

Viele Images arbeiten mit dem Nutzer "root". Die Kapselung von Docker ist aber relativ robust (nicht zu 100%, aber zu 99%). Der Prozess im Container sieht das Dateisystem, das der Docker Daemon ihm vorsetzt. Also die Summe aller Schichten und die Volumes, die man ihm reinreicht. Aus diesen Volumes kommt ein Prozess nicht raus.

Wenn Sie eigene Images bauen, können Sie im Dockerfile einen Benutzer anlegen und dann Docker dazu bringen, diesen zu nutzen. Beispiel:

```
FROM ubuntu:latest
RUN groupadd -r postgres && useradd -u 1005 --no-log-init -r -g postgres postgres
USER postgres
```

Der Nutzer hat dann die UID 1005. Wenn Sie denselben Nutzer mit derselben UID auf dem Gastgeber anlegen, erleichtern Sie sich ggf. das Leben, wenn Sie zum Beispiel einen Job erzeugen, der die Daten sichern soll.

Frage:

Wie bekomme ich extern gebaute Artefakte (z.B. Maven) in einen automatischen Bau-Prozess?

Antwort:

Wenn Sie GitHub Actions benutzen möchten, müssen Sie Ihre Artefakte so bereitstellen, dass man sie über das Internet beziehen kann. Im Dockerfile kann man das dann zum Beispiel von externer Stelle laden:

```
RUN wget https://internal-artifacts.example.org/app.tar.gz \
&& tar zxvpf app.tar.gz && rm app.tar.gz
```

Frage:

Am ersten Tag wurde erwähnt, dass ein Image auch gar kein Base-Image haben kann. Wie mache ich das konkret, wenn ich wirklich nur eine ausführbare Datei habe?

Antwort:

Als Beispiel nehmen wir mal Go und bauen ein Multistage-Image (schematische Darstellung, ungetestet!):

```
FROM golang
COPY ./code .
RUN GOOS=linux go build -o app .

FROM scratch
COPY --from=0 app .
CMD [ "./app" ]
```

In der ersten Stufe wird der Code kopiert, der Go-Compiler baut das Executable `app`. In der zweiten Stufe wird ein Image `FROM scratch` erzeugt. Das enthält absolut nichts. Darin kopiert Docker dann `app` aus Stufe 1 und führt es aus. Das Image ist so groß wie das Executable. Klappt nur mit Anwendungen, die komplett bedürfnislos sind.

Frage:

Kann Traefik auch HTTP/2 ausliefern? Wie aktiviere ich das?

Antwort:

Ja, da gibt es nichts zu aktivieren. Wenn die Seite per TLS ausgeliefert wird, wird auch HTTP/2 unterstützt!

Weitere Software

Im Workshop wurde an einigen auf Themen und Konzepte verwiesen. Hier noch einmal eine kleine Übersicht und Links:

1. Kubernetes

Kubernetes (K8S) ist ein Container-Orchestrator für große Umgebungen. Aber auch in mittleren Umgebungen kann es uns das Leben erleichtern. Kubernetes löst folgende Probleme:

- Cluster aus mehreren Servern
- API mit ausgefeiltem Berechtigungsmodell
- ein riesiges Umfeld mit Software für viele spezielle Probleme
- Managed-Kubernetes als Produkt von Cloud-Anbietern (keine Verwaltung des Gastgebers).
- Rolling Updates (Austausch von Containern ohne Ausfall)
- definierte Reihenfolgen von Starts und Neustarts

Wichtig vorm Einstieg in K8S: Solides Docker-Wissen und – noch wichtiger – Erfahrung. Zu Erfahrungen gehört auch das Scheitern.

2. Eigene Registry

Zwei Registries haben wir behandelt: Den Docker-Hub und die GitHub-Registry. Wenn Sie eine eigene Container-Registry brauchen, können folgende Produkte interessant sein:

- [Harbor](#)

- [GitLab](#)

Über diese Anleitung

Diese Dokumente sind Teil des Online-Workshops "Docker und Container in der Praxis" von c't-Redakteur Jan Mahn für Heise Events.