Docker Grundlagen

Kursheft

Kapitel: Grundlagen

DATAMICS
machine intelligence consulting services

Build Dockerfile

Im Folgenden findest du einige **dockerfile**-Befehle, die du kennen solltest:

FROM Das Basis-Image zum Erstellen eines neuen Images. Dieser Befehl muss sich am Anfang der dockerfile befinden.

RUN Wird verwendet, um einen Befehl während des Buildvorgangs des Docker-Images auszuführen.

COPY/ADD Kopiere eine Datei vom Hostcomputer in das neue Docker-Image. Bei **ADD** gibt es eine Option, um eine URL für die Datei zu verwenden, Docker lädt diese Datei dann in das Zielverzeichnis herunter.

CMD
Wird zum Ausführen von Befehlen verwendet, wenn wir einen neuen Container aus dem Docker-Image erstellen.

ENTRYPOINT
Definiert den Standardbefehl, der ausgeführt wird, wenn der Container ausgeführt wird.

WORKDIR
Dies ist das Verzeichnis für die Ausführung des CMD-Befehls.

ENV
Definiert eine Umgebungsvariable.

VOLUME
Aktiviert das Link-Verzeichnis zwischen dem Container und dem Hostcomputer.

Erstellen eines Images mit einem einfachen Dockerfile (FROM/RUN)

Erstelle eine leere Datei:

```
touch Dockerfile
```

und füge (mit einem Text Editor deiner Wahl) den folgenden Inhalt ein:

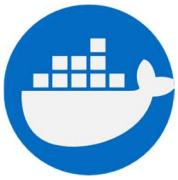
```
FROM ubuntu

RUN echo "hello world"
```

Erstelle das Image:

```
docker build .
docker images
```

Übergib mit dem **-t** Parameter den **REPOSITORY** Namen und einen TAG:

```
docker build -t mein_docker:latest .
docker images
```

Praxistipps:

- Erleichtere dir nach Möglichkeit spätere Änderungen, indem du mehrzeilige Argumente alphanumerisch sortierst. Dies hilft dir, Doppelungen von Paketen zu vermeiden und die Liste einfacher zu bearbeiten. Es lässt sich auch viel einfacher lesen und überprüfen.
- Das Hinzufügen eines Leerzeichens vor einem Backslash hilft ebenfalls.
- Kombiniere immer **RUN** apt-get **update** mit apt-get **install** in der gleichen RUN-Anweisung.

Hier ist ein Beispiel:

```
FROM ubuntu

RUN apt-get update && apt-get install -y \
  cvs \
  git \
  subversion
```

```
docker build -t mein_docker:latest .
```


Die LABEL-Anweisung fügt einem Image Metadaten hinzu.

```
FROM ubuntu

LABEL maintainer="Rene@email.de"

LABEL version="1.0"

RUN echo "hello world"
```

```
docker build -t mein_docker:latest .
```

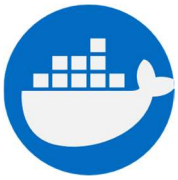
Übertrage Dateien in dein Docker Image (COPY/ADD)

Erstelle eine Datei mit einem Inhalt

```
echo "hallo meine Datei…" > eine_datei.txt
```

```
FROM ubuntu

COPY eine_datei.txt .

RUN cat /eine_datei.txt
```

```
docker build -t mein_docker:latest .
```



Praxistipp: COPY oder ADD

Obwohl **ADD** und **COPY** funktional ähnlich sind, wird COPY im Allgemeinen bevorzugt. Das liegt daran, dass es transparenter ist als ADD. COPY unterstützt nur das grundlegende Kopieren lokaler Dateien in den Container, wohingegeb ADD über einige Funktionen verfügt (z.B. lokale tar-Extraktion und Remote-URL-Unterstützung), die nicht sofort offensichtlich sind. Folglich ist die beste Verwendung für ADD die automatische Extraktion der lokalen tar-Datei in das Image, wie in ADD rootfs.tar.xz /.

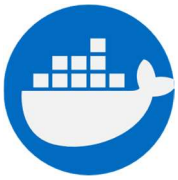
Verwenden von CMD

RUN - Befehl wird ausgeführt, während wir das Docker-Image erstellen:
Mit der RUN-Anweisung kannst du zum Beispiel deine Anwendung und die dafür erforderlichen Pakete installieren. Sie führt alle Befehle über dem aktuellen Bild aus und erstellt eine neue Ebene (Layer), indem die Ergebnisse übertragen werden. Oft findest du mehrere RUN-Anweisungen in einer Dockerfile.

CMD – Befehl wird ausgeführt, während wir das erstellte Docker-Image starten:
Mit der CMD-Anweisung kannst du einen Standardbefehl festlegen, der nur ausgeführt wird, wenn du den Container ausführst, ohne einen Befehl anzugeben. Wenn der Docker-Container mit einem Befehl ausgeführt wird, wird der Standardbefehl ignoriert. Wenn das Dockerfile über mehr als eine CMD-Anweisung verfügt, wird nur die letzte CMD-Anweisung ausgeführt.

```
FROM ubuntu

CMD ["/bin/bash"]
```

```
docker build -t mein_docker:latest .
docker run -it mein_docker
```



Praxistipp:

Verwende immer die Arraysyntax, wenn du CMD (und ENTRYPOINT) verwendest

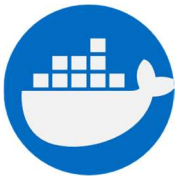
```
CMD /bin/echo
```

oder

```
CMD ["/bin/echo"]
```

Dies sieht vielleicht nicht so aus, als wäre es ein Problem, aber der Teufel steckt im Detail. Wenn du die zweite Syntax verwendest, bei der es sich bei **CMD** (oder **ENTRYPOINT**) um ein Array handelt, funktioniert es genauso, wie du es erwarten würdest. Wenn du die erste Syntax ohne Array verwendest, wird mit Docker `/bin/sh -c` bei deinem Befehl davorgesetzt.

Das Voranstehende `/bin/sh -c` kann zu unerwarteten Problemen und Funktionen führen, die nicht leicht verständlich sind, wenn man nicht weiß, dass Docker die **CMD** geändert hat. Daher solltest du **immer** die Arraysyntax für beide Anweisungen verwenden. Denn beide werden genauso ausgeführt, wie du es eingibst.



ENTRYPOINT (und CMD)

Der **ENTRYPOINT** gibt einen Befehl an, der immer ausgeführt wird, wenn der Container gestartet wird. Die **CMD** gibt Argumente an, die dem **ENTRYPOINT** zugeführt werden.

Beispiel: **ENTRYPOINT** oder **CMD**?

```
ping localhost
```

Fall 1: Wenn dein Dockerfile den **ENTRYPOINT** und die **CMD** verwendet:

```
FROM debian:wheezy
ENTRYPOINT ["/bin/ping"]
CMD ["localhost"]
```

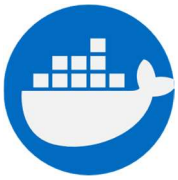
```
docker build -t mein_docker:latest .
```

Wenn du das Image ohne Argument ausführst, dann wird der lokale Host angepingt:

```
docker run -it mein_docker
```

Wenn du nun das Image mit einem Argument ausführst, dann wird das Argument angepingt (hier: google.com):

```
docker run -it mein_docker google.com
```

Fall 2: Wenn dein Dockerfile nur die CMD verwendet:

```
FROM debian:wheezy
CMD ["/bin/ping", "localhost"]
```

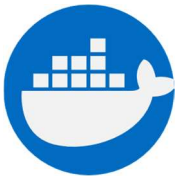
```
docker build -t mein_docker:latest .
```

Wenn du das Image ohne Argument ausführst, dann wird der lokale Host angepingt:

```
docker run -it mein_docker
```

Durch Ausführen des Images mit einem Argument wird jedoch das Argument ausgeführt. Das ist hier im Beispiel der Befehl *bash*:

```
docker run -it mein_docker bash
```



WORKDIR

```
FROM debian:wheezy

RUN mkdir -p /app

COPY eine_datei.txt /app

WORKDIR /app

RUN cat eine_datei.txt
```

Umgebungsvariablen

```
FROM debian:wheezy

ENV name Rene

RUN echo "Hello, $name"
```

```
docker build -t mein_docker:latest .
```

Praxisbeispiele für Umgebungsvariablen:

```
ENV php_conf /etc/php/7.0/fpm/php.ini
ENV nginx_conf /etc/nginx/nginx.conf
```

Praxisprojekt Flask Webserver

Flask ist ein leichtes WSGI-Webanwendungsframework. Es wurde entwickelt, um den Einstieg schnell und einfach zu machen, mit der Möglichkeit, auf komplexe Anwendungen zu skalieren. Es begann als einfacher Wrapper um Werkzeug und Jinja und hat sich zu einem der beliebtesten Python-Webanwendungsframeworks entwickelt. https://de.wikipedia.org/wiki/Flask

Erstelle als erstes einen Ordner namens *app*. Die Dateien für die Flask App werden dann hier gespeichert.

```
mkdir app
cd app
```

Erstelle anschließend eine HTML Seite mit dem Namen flyboard_hello.html, in der du den folgenden HTML Code hinzufügst:

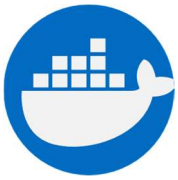
```
mkdir templates
touch templates/flyboard_hallo.html
```

```
<h1 style="color:#eb4034">Ich fliege!!</h1>


<img src="static/images/output.gif">
```

Kopiere dann das output.gif, welches wir im vorherigen Praxisprojekt erstellt haben, in das Verzeichnis *app/static/images/*

Hinweis: Das Bild sollte genau in diesem Pfad liegen.

```
mkdir -p static/images
cp ../output.gif static/images
```



Erstelle anschließend eine Datei mit dem Namen app.py, in der du den folgenden Code hinzufügst:

```
touch app.py
```

```
from flask import Flask, render_template


app = Flask(__name__)


@app.route('/')
def fly():
    return render_template("flyboard_hallo.html")


if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Hinweis: Um eine einfachere Version auszuführen, kannst du auch die folgende Flask App ohne eigene HTML verwenden:

```
from flask import Flask


app = Flask(__name__)
```

```
@app.route('/')
def hello_whale():
    return 'Ich fliege!'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Da wir Flask verwenden, müssen wir das noch als Paket in Python installieren. Erstelle dazu die folgende Datei **requirements.txt** innerhalb des app-Ordners und füge die folgende Zeile hinzu:

```
touch requirements.txt
```

```
Flask==1.1.1
```

Erstelle jetzt das Dockerfile, welches unser Image erstellt und dann bereitstellt. Benenne dieses Dockerfile

```
touch Dockerfile
```

und füge den folgenden Code ein:

```
FROM python:latest
RUN apt-get update -y
COPY . /app
WORKDIR /app
```

```
RUN pip install -r requirements.txt

ENTRYPOINT ["python"]

CMD ["app.py"]
```

Um dieses Image zu erstellen, führe den folgenden Befehl aus:

```
docker build -t flask-beispiel:latest .
```

Und um den Container auszuführen, führe aus:

```
docker run -d -p 5000:5000 flask-beispiel
```

Überprüfe ob der Container läuft:

```
docker ps
```

Super! Dein Container läuft mit Port 5000!

Gehe jetzt in einem Browser zu localhost:5000

Stoppe nun den Container:

```
docker stop [DEINE CONTAINER NUMMER HIER]
```