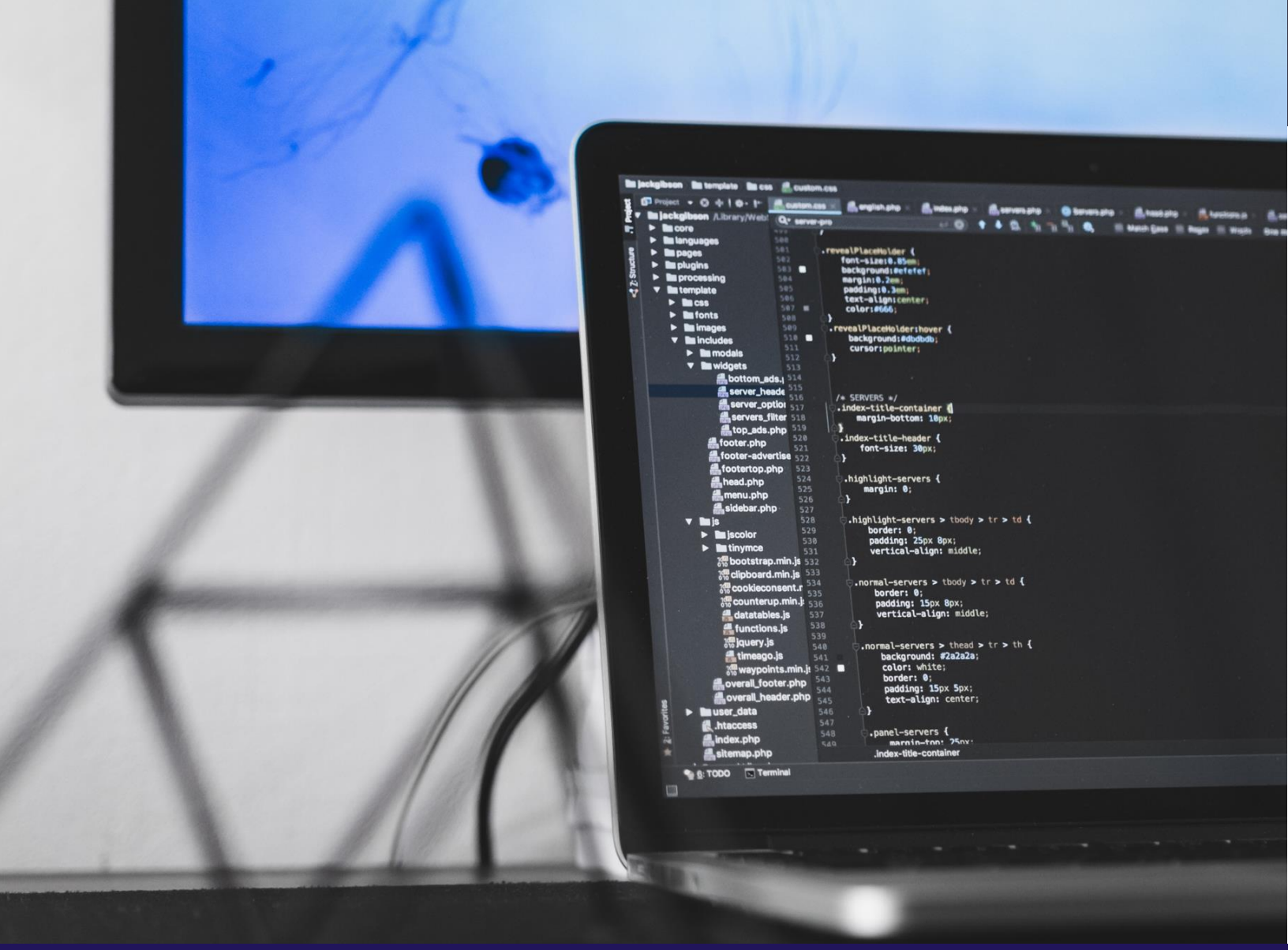


Docker Grundlagen

Kursheft

Kapitel: Docker Compose

DATAMICS
machine intelligence consulting services

Inhaltsverzeichnis

Inhaltsverzeichnis 1

Docker Compose 2

- **Grundvoraussetzungen** 2
- **Schritt 1: Setup** 2
- **Schritt 2: Erstelle ein Dockerfile** 4
- **Schritt 3: Definiere Services in einem Composefile** 6
- **Schritt 4: Erstelle und betreibe deine App mit Compose** 7
- **Schritt 5: Hinzufügen von„Bind Mounts“ zum Compose File** 9
- **Schritt 6: Aktualisieren und Testen der App mit Compose** 10
- **Schritt 7: Die Anwendung Updaten** 11
- **Schritt 8: Experimentiere mit anderen Befehlen** 12

Docker Compose

Mit dieser Lerneinheit erstellst du eine einfache Python-Webanwendung, die mit Docker Compose ausgeführt wird. Die Anwendung verwendet das Flask-Framework und verwaltet einen Hit Counter (Trefferabfrage) in Redis. Obwohl das Beispiel Python verwendet, sollte es auch verständlich sein, wenn man kein Pythonprofi ist.

Grundvoraussetzungen

Stelle sicher, dass du Docker Engine und Docker Compose installiert hast. Python und Redis musst du nicht zusätzlich installieren, da beide mit Docker Images sowieso bereits vorhanden sind. Teste Docker Compose:

```
docker-compose --version
```

Schritt 1: Setup

Definiere die anwendungsspezifischen Abhängigkeiten.

1. Lege ein Verzeichnis für dein Projekt an:

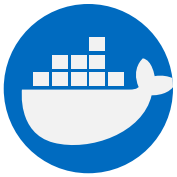
```
$ mkdir composetest
$ cd composetest
```

2. Erstelle ein File mit dem Namen app.py in deinem Projektverzeichnis und kopiere es in:

```
import time
```

```
import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)



def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)



@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

In diesem Beispiel ist `redis` der Hostname des Redis Containers im Netzwerk. Wir verwenden den Defaultport für Redis, `6379`.



Behandlung vorübergehender Fehler (transient errors)

Beachte, wie die Funktion `get_hit_count` geschrieben wird. Mit dieser grundlegenden Wiederholungsschleife können wir unsere Anfrage mehrmals wiederholen, wenn der Redis-Dienst nicht verfügbar ist. Dies ist beim Start hilfreich, wenn die Anwendung online geschaltet wird, macht sie jedoch auch widerstandsfähiger, wenn der Redis-Dienst während der Lebensdauer der Anwendung neu gestartet werden muss. In einem Cluster hilft dies auch bei der Reaktion auf vorübergehender Verbindungsabbrüche zwischen Knoten.

Erstelle eine weitere Datei namens `requirements.txt` in deinem Projektverzeichnis und füge diese hier ein:

```
flask
redis
```

Schritt 2: Erstelle ein Dockerfile

In diesem Schritt, schreibst du ein Dockerfile das ein Docker Image erstellt. Das Image enthält alle nötigen Abhängigkeiten, die die Python App benötigt, inklusive Python selbst.

Erstelle in deinem Projektverzeichnis eine Datei mit dem Namen `Dockerfile` und füge Folgendes ein:

```
FROM python:3.7-alpine

WORKDIR /code

ENV FLASK_APP app.py

ENV FLASK_RUN_HOST 0.0.0.0
```

```
RUN apk add --no-cache gcc musl-dev linux-headers

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["flask", "run"]
```

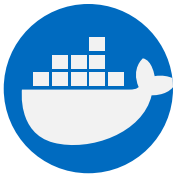
Sollte hier ein Fehler auftreten, dann versuche es damit, die Zeile **RUN apk add --no-cache gcc musl-dev linux-headers** auszukommentieren. Das machst du wie folgt (# …):

```
# RUN apk add --no-cache gcc musl-dev linux-headers
```

Dies weist Docker an:

- Erstelle ein Image, das mit dem Python 3.7-Image beginnt.
- Setze das Arbeitsverzeichnis auf `/code`
- Lege die Umgebungsvariablen fest, die vom Befehl `flask` verwendet werden.
- Installiere gcc, damit Python-Pakete wie MarkupSafe und SQLAlchemy Speedups kompilieren können.
- Kopiere die Datei `requirements.txt` und installiere die Python-Abhängigkeiten.
- Kopiere das aktuelle Verzeichnis in das Projekt zum workdir. im Image.
- Stelle den Standardbefehl für den Container auf `flask run`

Für weiter Infos, wie man Dockerfiles schreibt, kannst du natürlich gerne auch auf Docker selbst nachschlagen (Docker user guide und Dockerfile reference) oder gehe in unserem Kurs nochmals ein paar Schritte zurück, um dir alles genau anzuschauen.



Schritt 3: Definiere Services in einem Composefile

Erstelle eine Datei mit dem Namen docker-compose.yml in deinem Projektverzeichnis und füge Folgendes ein:

```
version: '3'

services:

  web:

    build: .

    ports:

      - "5000:5000"

  redis:

    image: "redis:alpine"
```

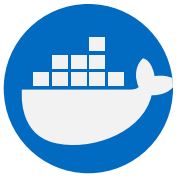
Dieses Compose File definiert zwei Services: web und redis.

Web service

Der web -Service verwendet ein Image, das aus dem Dockerfile im aktuellen Verzeichnis erstellt wurde. Anschließend werden der Container und der Hostcomputer an den freien Port 5000 gebunden. Dieser Beispiel-Service verwendet den Standardport für den Flask-Webserver 5000.

Redis service

Der redis Service verwendet das öffentliche Redis Image, das auf der Docker Hub-Registry abgerufen werden kann.



Schritt 4: Erstelle und betreibe deine App mit Compose

1. Starte deine Anwendung jetzt aus deinem Projektverzeichnis, indem `docker-compose up -d` ausführst.

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

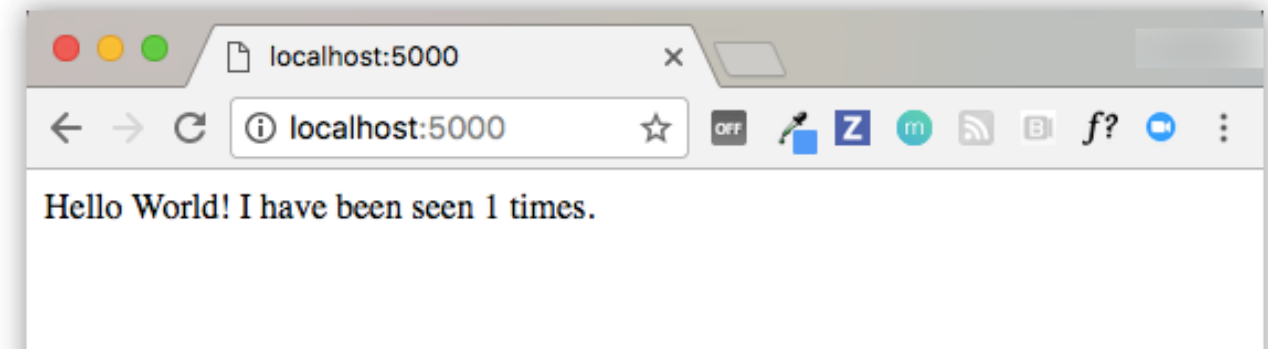
Compose ruft ein Redis-Image ab, erstellt ein Image für deinen Code und startet die von dir definierten Services. In diesem Fall wird der Code beim Erstellen statisch in das Image kopiert.

2. Gib nun im Browser http://localhost:5000/ ein, um die Anwendung laufen zu sehen. Wenn du Docker standardmäßig unter Linux, Docker Desktop für Mac oder Docker Desktop für Windows verwendest, sollte die Web-App jetzt auf Port 5000 auf Ihrem Docker-Daemon-Host abrufbar sein. Gehe in deinem Browser auf http://localhost:5000, um die `Hello World`-Nachricht anzusehen. Wenn diese nicht erscheint, kannst du auch noch http://127.0.0.1:5000 ausprobieren.

   Wenn du Docker Machine auf einem Mac oder Windows verwendest, rufe die IP-Adresse deines Docker-Hosts mit der `docker-machine ip MACHINE_VM` ab. Öffnen dann `http://MACHINE_VM_IP:5000` in einem Browser.

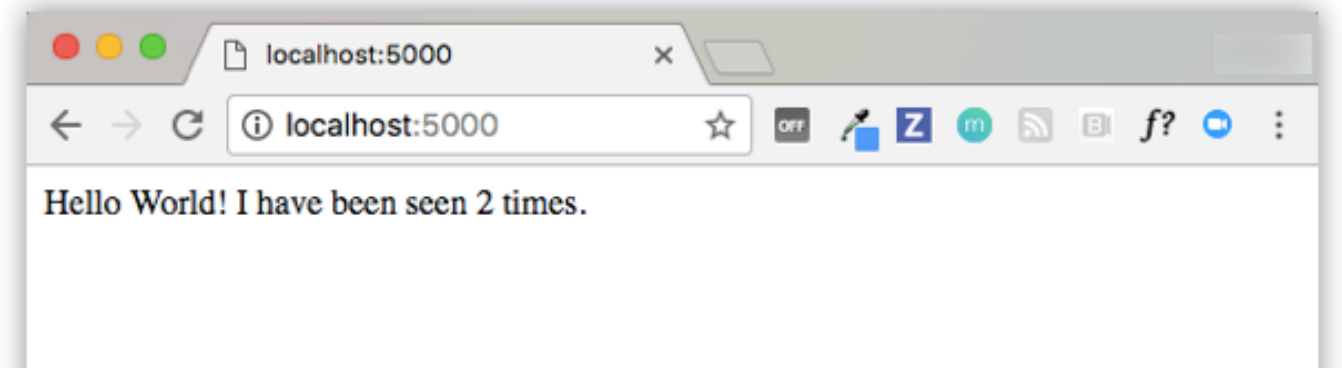
   Du solltest in deinem Browser folgende Nachricht finden:

```
Hello World! I have been seen 1 times.
```



3. Lade die Seite erneut. Die Zahl sollte sich nun um eins erhöhen.

```
Hello World! I have been seen 2 times.
```



4. Wechsle zu einem anderen Terminalfenster und gib `docker image ls` ein, um die lokalen Images abzurufen. Die abgerufenen Images sollten `redis` und `web` zurückgeben.

```
$ docker image ls

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
composetest_web     latest              e2c21aa48cc1        4 minutes ago       93.8MB
python              3.4-alpine          84e6077c7ab6        7 days ago          82.5MB
```

```
redis          alpine          9d8fa9aa0e5b        3 weeks ago         27.5MB
```

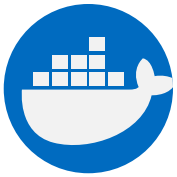
Du kannst die Images mit `docker inspect <tag or id>` untersuchen.

5. Beende die Anwendung, indem du `docker-compose down` in deinem Projektverzeichnis im zweiten Terminal ausführst oder STRG + C im ursprünglichen Terminal, in dem du die App gestartet hast.

Schritt 5: Hinzufügen von „Bind Mounts“ zum Compose File

Bearbeite das YAML `docker-compose.yml` in deinem Projektverzeichnis, um ein bind mount für den `web` Service hinzuzufügen:

```
version: '3'

services:

  web:

    build: .

    ports:

      - "5000:5000"

    volumes:

      - .:/code

    environment:

      FLASK_ENV: development

  redis:
```



```
image: "redis:alpine"
```

Die neue volumes Datei referenziert das Projektverzeichnis (aktuelles Verzeichnis) direkt auf den Host und wird in /code im Container eingebunden, sodass der Code im Handumdrehen geändert werden kann, ohne das Image neu erstellen zu müssen. Der environment Key legt die Umgebungsvariable FLASK_ENV fest, mit der flask run angewiesen wird, im Entwicklungsmodus ausgeführt zu werden und den Code bei Änderungen neu zu laden. Dieser Modus sollte nur in der Entwicklung verwendet werden.

Schritt 6: Aktualisieren und Testen der App mit Compose

Aus deinem Projektverzeichnis, gib nun docker-compose up ein, um deine Anwendung mit dem neuen Docker Compose-File zu erstellen und zu testen:

```
$ docker-compose up
```

Prüfe nun nochmals die Hello World Nachricht in einem Web Browser und drücke refresh (Aktualisieren), um die Zählung der Aufrufe zu überprüfen.

Shared folders, volumes, and bind mounts

Wenn sich dein Projekt außerhalb des User Verzeichnisses (cd ~) befindet, musst du das Laufwerk oder den Speicherort der Docker-Datei und des verwendeten Volumes freigeben. Wenn du Laufzeitfehler erhältst, die darauf hinweisen, dass eine Anwendungsdatei nicht gefunden wurde, ein Volume-Mount abgelehnt wurde oder ein Service nicht gestartet werden kann, aktiviere die Datei- oder Laufwerksfreigabe. Für die Bereitstellung von Volumes („Volume Mounting“) sind freigegebene Laufwerke für Projekte erforderlich, die sich außerhalb von C:\Users (Windows) oder /Users (Mac) befinden. Sie sind für alle Projekte auf Docker Desktop für Windows erforderlich, die Linux Containers verwenden. Weitere Informationen findest du unter

Shared Drives bei Docker Desktop für Windows, File sharing bei Docker für Mac und in den allgemeinen Beispielen zum Verwalten von Daten in Containern hier: Manage data in containers.

Wenn du Oracle VirtualBox au einem älteren Windows OS verwendest, begegnest du evtl. Problemen mit freigegebenen Ordnern, wie in diesem Ticket beschrieben: VB trouble ticket. Neuere Windows Versionen erfüllen die Anforderungen von Docker Desktop for Windows und benötigen kein VirtualBox.

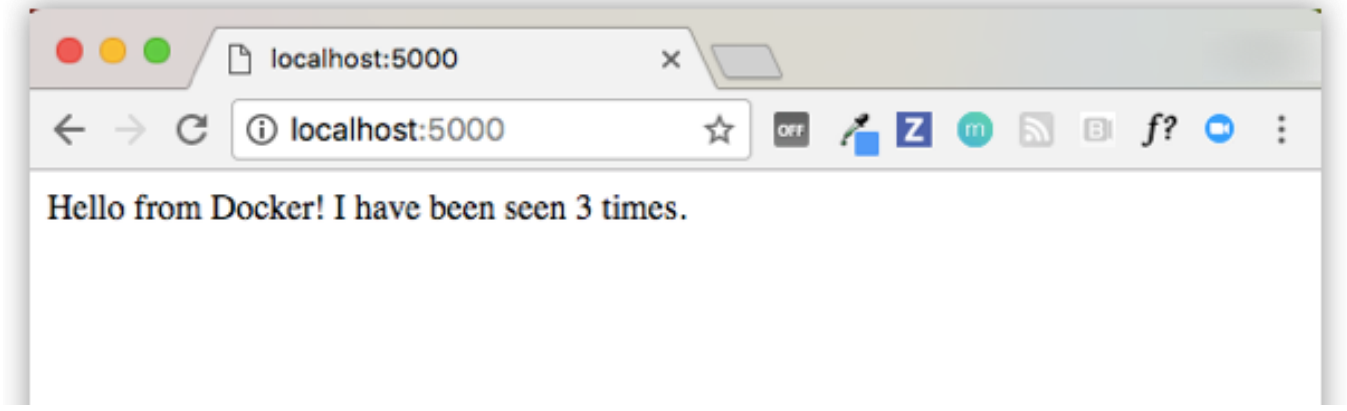
Schritt 7: Die Anwendung Updaten

Da der Anwendungscode jetzt mithilfe eines Volumes in den Container eingebunden wird, können Änderungen am Code vorgenommen werden und sofort angezeigt werden, ohne das Image neu erstellen zu müssen.

1. Ändere die Begrüßung in app.py und speichere sie ab. Zum Beispiel statt Hello World! in Hello from Docker!:

```
return 'Hello from Docker! I have been seen {} times.\n'.format(count)
```

2. Aktualisiere die Anwendung in deinem Browser. Die Begrüßung sollte sich nun updaten, die Zählung sollte jedoch fortlaufend bleiben und sich um eins erhöhen:



Schritt 8: Experimentiere mit anderen Befehlen

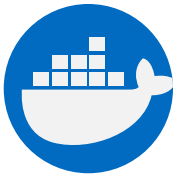
Wenn du die Dienste im Hintergrund ausführen möchtest, kannst du das Flag -d (für „detached“ Modus) an `docker-compose up` übergeben und mit `docker-compose ps` überprüfen, was gerade ausgeführt wird:

```
$ docker-compose up -d

Starting composetest_redis_1...

Starting composetest_web_1...


$ docker-compose ps

Name                Command           State       Ports

------------------------------------------------------------------

composetest_redis_1   /usr/local/bin/run         Up

composetest_web_1     /bin/sh -c python app.py   Up      5000->5000/tcp
```

Der Befehl `docker-compose run` erlaubt es dir einmalige Befehle für deine Dienste laufen zu lassen, beispielsweise um zu sehen, welche Umgebungsvariablen für den Webdienst verfügbar sind:

```
$ docker-compose run web env
```

In `docker-compose --help` findest du andere verfügbare Befehle. Du kannst auch command completion installieren (für bash + zsh shell), um weitere Befehle angezeigt zu bekommen.



Wenn du Compose mit `docker-compose up -d` startest, beende den Dienst, wenn du mit ihm fertig bist:

```
$ docker-compose stop
```

Du kannst auch alles damit beenden, dass du deinen Container komplett löschst, mit `down` oder mit `–volumes,` um ebenfalls Dateninhalte zu entfernen, die von den Redis Containern genutzt wurden.

```
$ docker-compose down --volumes
```