5 Docker in der Entwicklung einsetzen

In Teil II dieses Buches werden wir eine einfache Webanwendung entwickeln, die für einen gegebenen String ein eindeutiges Bild zurückgibt – ähnlich den Identicons von GitHub und StackOverflow für Benutzer ohne eigenes Bild. Wir werden die Anwendung in Python mit dem Flask-Webframework schreiben. Python wurde für dieses Beispiel gewählt, weil es eine weite Verbreitung gefunden hat, kurz und prägnant ist, dabei aber gut lesbar bleibt. Machen Sie sich keine Sorgen, wenn Sie nicht in Python programmieren. Wir werden uns auf die Interaktion mit Docker konzentrieren, nicht auf die Details des Python-Codes.[1] Aus ähnlichen Gründen wurde Flask gewählt – es ist schlank und leicht zu verstehen. Wir werden Docker verwenden, um all unsere Abhängigkeiten zu verwalten, so dass es nicht notwendig ist, Python oder Flask auf Ihrem Host-Computer zu installieren.

Dieses Kapitel wird sich darauf konzentrieren, einen auf Containern basierenden Workflow und die notwendigen Tools einzurichten, bevor wir im nächsten Kapitel mit der Entwicklung beginnen.

5.1 Sag »Hallo Welt!«

Beginnen wir damit, einen Webserver zu erstellen, der einfach nur »Hallo Welt!« zurückgibt. Legen Sie zuerst ein neues Verzeichnis mit dem Namen *identidock* an, in dem unser Projekt liegen wird. Innerhalb dieses Verzeichnisses erstellen Sie ein Unterverzeichnis *app* für unseren Python-Code. In diesem Verzeichnis erstellen Sie eine Datei namens *identidock.py*:

```
$ tree identidock/
identidock/
└── app
    └── identidock.py

1 directory, 1 file
```

1. Wollen Sie mehr über Python und Flask erfahren, schauen Sie sich *Flask Web Development* von Miguel Grinberg an (O'Reilly, *http://shop.oreilly.com/product/0636920031116.do*), insbesondere, wenn Sie Webanwendungen schreiben wollen.

Fügen Sie nun in *identidock.py* folgenden Code ein:

```
from flask import Flask
app = Flask(__name__)  ❶

@app.route('/')  ❷
def hello_world():
    return 'Hallo Welt!\n'

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

Erklären wir kurz den Code:

❶ Initialisiert Flask und richtet das Anwendungsobjekt ein.

❷ Erstellt eine Route, die mit der URL verbunden ist. Wann immer diese URL angefordert wird, führt das zu einem Aufruf der Funktion `hello_world`.

❸ Initialisiert den Python-Webserver. Durch den Einsatz von `0.0.0.0` als Host-Argument (statt `localhost` oder `127.0.0.1`) wird an alle Netzwerkschnittstellen gebunden, was notwendig ist, damit der Container auch vom Host oder von anderen Containern aus angesprochen werden kann. Die `if`-Anweisung in der Zeile darüber stellt sicher, dass der Code nur ausgeführt wird, wenn die Datei als Standalone-Programm läuft und nicht als Teil einer größeren Anwendung.

Quellcode

Sie finden den Quellcode für dieses Kapitel auf GitHub unter *https://github.com/using-docker/using_docker_in_dev*. Es gibt Git-Tags für die verschiedenen Zustände des Codes im Kapitel.

Mir wurde gesagt, dass sich der Code eher schlecht aus dem E-Book herauskopieren lässt, daher nutzen Sie am besten das GitHub-Repository, wenn Sie Probleme haben.

Jetzt benötigen wir einen Container, in dem wir diesen Code unterbringen und ihn ausführen können. Erstellen Sie im Verzeichnis *identidock* eine Datei namens *Dockerfile* mit folgendem Inhalt:

```
FROM python:3.4

RUN pip install Flask==0.10.1
WORKDIR /app
COPY app /app

CMD ["python", "identidock.py"]
```

Dieses Dockerfile nutzt als Basis ein offizielles Python-Image, welches eine Installation von Python 3 enthält. Darauf installiert es Flask und kopiert unseren Code hinein. Die `CMD`-Anweisung führt dann einfach unseren Identidock-Code aus.

Offizielle Imagevarianten

Viele der offiziellen Repositories für beliebte Programmiersprachen wie Python, Go und Ruby enthalten mehrere Images für unterschiedliche Zwecke. Neben den Images für verschiedene Versionsnummern werden Sie sehr wahrscheinlich mindestens eines der folgenden vorfinden:

slim
: Diese Images sind ausgedünnte Versionen der Standardimages. Viele gebräuchliche Pakete und Bibliotheken werden fehlen. Diese Images sind notwendig, wenn Sie die Imagegröße für die Bereitstellung reduzieren müssen, aber sie erfordern zusätzlichen Aufwand beim Installieren und Warten der Pakete, die im Standardimage schon vorhanden sind.

onbuild
: Diese Images nutzen die Dockerfile-Anweisung `ONBUILD`, um mit dem Ausführen bestimmter Befehle zu warten, bis aus dem Image ein neues »Kind«-Image erstellt wird. Diese Befehle werden als Teil der `FROM`-Anweisung für das Kind-Image verarbeitet und sind im Allgemeinen dazu gedacht, Code zu kopieren oder einen Kompilierungsschritt auszuführen. Solche Images können den Einstieg in eine Sprache vereinfachen, aber langfristig sind sie meist zu eingeschränkt und verwirrend. Ich empfehle, Onbuild-Images nur zu verwenden, wenn Sie ein Repository erstmalig durchstöbern.

Für unsere Beispielanwendung werden wir ein Standardbasisimage für Python 3 verwenden und nicht auf eine dieser Varianten zurückgreifen.

Jetzt können wir unsere einfache Anwendung bauen und ausführen:

```
$ cd identidock
$ docker build -t identidock .
...
$ docker run -d -p 5000:5000 identidock
0c75444e8f5f16dfe5aceb0aae074cc33dfc06f2d2fb6adb773ac51f20605aa4
```

Hier habe ich das Flag `-d` an `docker run` übergeben, um den Container im Hintergrund zu starten. Sie können es aber auch weglassen, um die Ausgabe des Webservers zu sehen. Das Argument `-p 5000:5000` teilt Docker mit, dass wir Port 5000 im Container an Port 5000 auf dem Host weiterleiten wollen.

Testen wir das Ganze:

```
$ curl localhost:5000
Hallo Welt!
```

Docker Machine-IPs

Lassen Sie Docker mithilfe von Docker Machine laufen (was der Fall ist, wenn Sie Docker mit der Docker Toolbox auf dem Mac oder unter Windows installiert haben), werden Sie für die URL nicht `localhost` nutzen können.[2] Stattdessen müssen Sie die IP-Adresse der VM verwenden, in der Docker läuft. Mit der `ip`-Anweisung von Docker Machine können Sie das automatisieren. Zum Beispiel:

```
$ curl $(docker-machine ip default):5000
Hallo Welt!
```

Dieses Buch geht davon aus, dass Docker lokal auf ihrer Linux Maschine läuft. Achten Sie ansonsten darauf, `localhost` durch die passende IP-Adresse zu ersetzen.

Ausgezeichnet! Es gibt allerdings ein ziemlich großes Problem mit dem Workflow: Für jede kleine Änderung am Code müssen wir das Image neu bauen und den Container wieder starten. Zum Glück gibt es eine einfache Lösung. Wir können den Quellcode-Ordner auf dem Host über das entsprechende Verzeichnis im Container *bind-mounten*. Der folgende Code stoppt und entfernt den letzten laufenden Container (wenn das vorige Beispiel nicht der letzte laufende Container war, müssen Sie dessen ID mit `docker ps` herausfinden), bevor ein neuer gestartet wird, bei dem das Codeverzeichnis nach */app* gemountet ist:

```
$ docker stop $(docker ps -lq)
0c75444e8f5f
$ docker rm $(docker ps -lq)
$ docker run -d -p 5000:5000 -v "$PWD"/app:/app identidock
```

Das Argument `-v $PWD/app:/app` mountet das App-Verzeichnis im Container als */app*. Der Inhalt von */app* innerhalb des Containers wird überschrieben, und das Verzeichnis ist auch beschreibbar (wenn Sie das nicht wollen, können Sie ein Volume auch als schreibgeschützt mounten). Argumente von `-v` müssen absolute Pfade sein, daher haben wir `$PWD` genutzt, um auf das aktuelle Verzeichnis zuzugreifen, was uns einige Schreibarbeit erspart und alles portabel hält.

Bind Mount

Wird ein Host-Verzeichnis für ein Volume bei `docker run` mit dem Argument `-v HOST_DIR:CONTAINER_DIR` angegeben, bezeichnet man es im Allgemeinen als »Bind Mount«, da es einen Ordner (oder eine Datei) auf dem Host an einen Ordner (oder eine Datei) im Container bindet. Das ist ein bisschen verwirrend, weil alle Volumes technisch gesehen bind-mounted sind, aber

2. Mit der Bereitstellung von Docker for Mac/Windows gibt es nun eine direkte Anbindung via xhyve und hyper-v, die eine vollständigere Integration ermöglicht.

wir müssen ein wenig mehr Aufwand treiben, um das Verzeichnis auf dem Host zu finden, wenn es nicht explizit angegeben wurde.

Beachten Sie, dass sich `HOST_DIR` immer auf die Maschine bezieht, auf der die Docker Engine läuft. Sind Sie mit einem entfernten Docker Daemon verbunden, muss der Pfad auf dem entfernten Rechner existieren. Nutzen Sie eine lokale VM, die von Docker Machine bereitgestellt wird (was Sie tun, wenn Sie Docker über Toolbox installiert haben), wird Ihr Home-Verzeichnis cross-mounted, um während der Entwicklung die Sache nicht noch komplizierter zu gestalten.

Prüfen Sie, ob immer noch alles funktioniert:

```
$ curl localhost:5000
Hallo Welt!
```

Auch wenn wir das gleiche Verzeichnis gemountet haben, das mit der `COPY`-Anweisung zuvor hinzugefügt wurde, wird nun exakt das gleiche Verzeichnis auf dem Host und im Container genutzt und nicht eine Kopie aus dem Image. Daher können wir jetzt *identidock.py* bearbeiten und unsere Änderungen direkt sehen:

```
$ sed -i '' s/Welt/Docker/ app/identidock.py
$ curl localhost:5000
Hallo Docker!
```

Hier habe ich das sed-Tool genutzt, um schnell etwas in der Datei *identidock.py* zu ändern. Steht sed nicht zur Verfügung oder sind Sie nicht vertraut damit, können Sie auch einfach einen normalen Texteditor verwenden, um das Wort »Welt« durch »Docker« auszutauschen.

Bitte? Keine virtualenv?

Sind Sie erfahrener Python-Entwickler, werden Sie vielleicht überrascht sein, dass wir nicht virtualenv verwenden *(https://virtualenv.pypa.io/en/latest)*, um unsere Anwendung zu entwickeln. virtualenv ist ein ausgesprochen nützliches Tool, um Python-Umgebungen zu isolieren. Es ermöglicht es Entwicklern, für jede Anwendung eigene Python-Versionen und zugehörige Bibliotheken vorzuhalten. Normalerweise ist das in der Python-Entwicklung ein Standardvorgehen.

Wenn Sie mit Containern arbeiten, ist das allerdings weniger nützlich, da wir schon eine isolierte Umgebung zur Hand haben. Sind Sie an virtualenv gewöhnt, können Sie es natürlich immer noch im Container verwenden, aber Sie werden vermutlich keinen Vorteil darin sehen, sofern Sie nicht Konflikte mit anderen Anwendungen oder Bibliotheken haben, die im Container installiert sind.

Jetzt haben wir eine ziemlich normale Entwicklungsumgebung, nur dass all unsere Abhängigkeiten – der Python-Compiler und die Bibliotheken – innerhalb eines Docker-Containers gekapselt sind. Es gibt aber immer noch ein entschei-

dendes Problem. Wir haben keine Möglichkeit, diesen Container produktiv zu verwenden, vor allem weil er den Standard-Flask-Webserver laufen lässt. Der ist nur für die Entwicklung gedacht, weil er für ein Produktivumfeld zu ineffizient und zu unsicher ist. Ein kritischer Punkt beim Einbinden von Docker ist das Verringern der Unterschiede zwischen Entwicklung und Produktion, also wollen wir uns mal anschauen, wie wir das erreichen können.

uWSGI *(https://uwsgi-docs.readthedocs.org/en/latest)* ist ein produktiv einsetzbarer Anwendungsserver, der auch hinter einem Webserver wie nginx sitzen kann. Durch den Einsatz von uWSGI statt des Standard-Flask-Webservers erhalten wir einen flexiblen Container, den wir in vielen verschiedenen Situationen nutzen können. Der Container lässt sich mit nur zwei geänderten Zeilen im Dockerfile auf uWSGI umstellen:

```
FROM python:3.4

RUN pip install Flask==0.10.1 uWSGI=2.0.8  ❶
WORKDIR /app
COPY app /app

CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", \
     "/app/identidock.py", "--callable", "app", "--stats", \
     "0.0.0.0:9191"]  ❷
```

❶ uWSGI zur Liste der zu installierenden Python-Pakete hinzufügen.

❷ Einen neuen Befehl zum Ausführen von uWSGI erzeugen. Hier erklären wir uWSGI, einen HTTP-Server zu starten, der auf Port 9090 lauscht, und die Anwendung `app` aus */app/identidock.py* zu starten. Zudem wird ein Stats-Server auf Port 9191 gestartet. Wir könnten alternativ auch das `CMD` über den Befehl `docker run` überschreiben.

Bauen Sie das Image und starten Sie es, damit wir den Unterschied sehen:

```
$ docker build -t identidock .
...
Successfully built 3133f91af597
$ docker run -d -p 9090:9090 -p 9191:9191 identidock
00d6fa65092cbd91a97b512334d8d4be624bf730fcb482d6e8aecc83b272f130
$ curl localhost:9090
Hallo Docker!
```

Wenn Sie jetzt `docker logs` mit der Container-ID ausführen, werden Sie die Logging-Informationen für uWSGI erhalten und damit die Bestätigung haben, dass wir tatsächlich den uWSGI-Server verwenden. Zudem haben wir uWSGI gebeten, ein paar Statistiken bereitzustellen, die Sie über *http://localhost:9191* erhalten. Der Python-Code, der normalerweise den Standardwebserver startet, wurde nicht ausgeführt, da er nicht direkt von der Befehlszeile aus aufgerufen wurde.

Der Server arbeitet nun korrekt, aber ein paar administrative Dinge müssen wir noch erledigen. Wenn Sie sich die Logs des uWSGI anschauen, werden Sie feststellen, dass sich der Server darüber beschwert, dass er als root läuft. Das ist ein unnötiges Sicherheitsloch, welches wir im Dockerfile leicht stopfen können, indem wir einen Benutzer festlegen, unter dem der Server laufen soll. Zudem deklarieren wir explizit die Ports, an denen der Server lauscht:

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi  ❶
RUN pip install Flask==0.10.1 uWSGI==2.0.8
WORKDIR /app
COPY app /app

EXPOSE 9090 9191  ❷
USER uwsgi  ❸

CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", \
     "/app/identidock.py", \ "--callable", "app", "--stats", \
     "0.0.0.0:9191"]
```

Das bedeuten die neuen Zeilen:

❶ Erstellt Benutzer und Gruppe `uwsgi` auf dem üblichen Unix-Weg.

❷ Nutzt `EXPOSE`, um die Ports zu deklarieren, die für den Host und andere Container ansprechbar sind.

❸ Setzt den Benutzer für die folgenden Zeilen (einschließlich `CMD` und `ENTRYPOINT`) auf uwsgi.

Benutzer und Gruppen in Containern

Der Linux-Kernel nutzt *UIDs* und *GIDs*, um Benutzer zu identifizieren und deren Zugriffsrechte zu ermitteln. Das Abbilden von UIDs und GIDs auf Kennungen geschieht im Userspace durch das Betriebssystem. Darum sind UIDs im Container die gleichen wie auf dem Host, nur dass innerhalb von Containern erzeugte Benutzer und Gruppen nicht an den Host kommuniziert werden. Ein Nebeneffekt davon ist, dass Zugriffsberechtigungen verwirrend sein können – Dateien scheinen eventuell innerhalb und außerhalb des Containers unterschiedlichen Benutzer zu gehören. Achten Sie zum Beispiel auf die verschiedenen Besitzer der folgenden Datei:

```
$ ls -l test-file
-rw-r--r--    1 docker   staff 0 Dec 28 18:26 test-file
$ docker run -it -v $(pwd)/test-file:/test-file \
    debian bash
root@e877f924ea27:/# ls -l test-file
-rw-r--r-- 1 1000 staff 0 Dec 28 18:26 test-file
root@e877f924ea27:/# useradd -r test-user
root@e877f924ea27:/# chown test-user test-file
root@e877f924ea27:/# ls -l /test-file
```

```
-rw-r--r-- 1 test-user staff 0 Dec 28 18:26 /test-file
root@e877f924ea27:/# exit
exit
docker@boot2docker:~$ ls -l test-file
-rw-r--r-- 1 999         staff 0 Dec 28 18:26 test-file
```

Bauen Sie dieses Image wie üblich und testen Sie die neuen Benutzervorgaben:

```
$ docker build -t identidock .
...
$ docker run identidock whoami
uwsgi
```

Beachten Sie, dass wir die Standard-`CMD`-Anweisung überschrieben haben, die den Webserver startet, und stattdessen den Befehl `whoami` aufrufen. Dieser Befehl gibt den Namen des Benutzers zurück, der im Container zum Einsatz kommt.

Setzen Sie immer einen USER

Es ist wichtig, die `USER`-Anweisung in all Ihren Dockerfiles zu setzen (oder ihn in einem `ENTRYPOINT`- oder `CMD`-Skript zu ändern). Wenn Sie das nicht tun, laufen Ihre Prozesse im Container als root. Da UIDs innerhalb eines Containers und auf dem Host identisch sind, hat ein Angreifer, der sich Zugang zum Container verschafft, auch root-Zugriff auf dem Host-Rechner.

Bei Redaktionsschluss des englische Originals dieses Buches wurde noch daran gearbeitet, den root-Benutzer in einem Container auf einen Host-Benutzer mit großer UID abzubilden. Mit der Version 1.10 wurde dieses Feature mit der Daemon-Option `--userns-remap` verfügbar.

Super, jetzt laufen die Befehle innerhalb des Containers nicht mehr als root. Starten wir den Container erneut, jetzt aber mit etwas anderen Argumenten:

```
$ docker run -d -P --name port-test identidock
```

Dieses Mal haben wir die Ports auf dem Host nicht spezifiziert, an die gebunden werden soll. Stattdessen haben wir das Argument `-P` genutzt, welches dafür sorgt, dass Docker auf dem Host automatisch einen zufälligen Port im oberen Bereich auf jeden »bereitgestellten« Port des Containers abbildet. Wir müssen Docker dann fragen, um welche Ports es sich dabei handelt, bevor wir den Service ansprechen können:

```
$ docker port port-test
9090/tcp -> 0.0.0.0:32769
9191/tcp -> 0.0.0.0:32768
```

Hier sehen wir, dass 9090 an 32769 und 9191 an 32768 gebunden wurden, so dass wir den Service erreichen können (beachten Sie, dass die Ports bei Ihnen sehr wahrscheinlich andere sind):

```
$ curl localhost:32769
Hallo Docker!
```

Auf den ersten Blick scheint das einen unnötigen Zusatzschritt zu erfordern – was in diesem Fall durchaus zutrifft –, aber wenn Sie mehrere Container haben, die auf einem einzelnen Host laufen, ist es viel einfacher, Docker darum zu bitten, freie Ports automatisch zu mappen, statt sich selbst darum kümmern zu müssen.

Wir haben jetzt also einen laufenden Webservice, der schon sehr nahe an dem dran ist, wie er im Produktivumfeld laufen würde. Es gibt immer noch eine ganze Menge zu tun – zum Beispiel das Anpassen der uWSGI-Optionen für Prozesse und Threads –, aber im Vergleich zum Standard-Python-Debug-Webserver ist dieser viel ähnlicher.

Nun stellt sich uns aber ein neues Problem: Wir haben den Zugriff auf die Entwicklungstools verloren, wie zum Beispiel die Debugging-Ausgabe und das Live-Code-Reloading, das der Standard-Python-Webserver bietet. Wir konnten zwar die Unterschiede zwischen Entwicklungs- und Produktivumgebung massiv verringern, aber es gibt trotzdem grundsätzlich unterschiedliche Anforderungen, die immer ein paar Änderungen erfordern. Idealerweise wollen wir das gleiche Image für Entwicklung und Produktion verwenden und abhängig von der Umgebung ein leicht unterschiedliches Feature-Set einsetzen. Das können wir erreichen, indem wir eine Umgebungsvariable und ein einfaches Skript nutzen, das die Features abhängig von der Umgebung umschaltet.

Erstellen Sie eine Datei namens *cmd.sh* im gleichen Verzeichnis wie das Dockerfile und fügen Sie folgenden Inhalt ein:

```
#!/bin/bash
set -e

if [ "$ENV" = 'DEV' ]; then
  echo "Running Development Server"
  exec python "identidock.py"
else
  echo "Running Production Server"
  exec uwsgi --http 0.0.0.0:9090 --wsgi-file /app/identidock.py \
             --callable app --stats 0.0.0.0:9191
fi
```

Es sollte ziemlich klar sein, was dieses Skript bewirkt. Ist die Variable `ENV` auf `DEV` gesetzt, wird der Debug-Webserver gestartet, ansonsten der Produktivserver.[3] Der Befehl `exec` wird verwendet, um zu verhindern, dass ein neuer Prozess erzeugt wird. Damit ist sichergestellt, dass Signale (wie zum Beispiel `SIGTERM`) auch von unserem uwsgi-Prozess erhalten werden, statt dass der Eltern-Prozess sie verschluckt.

3. Wir haben jetzt Variablen wie zum Beispiel Port-Nummern, die in mehreren Dateien vorkommen. Das lässt sich durch Argumente oder Umgebungsvariablen lösen.

Konfigurationsdateien und Helper-Skripten verwenden

Um es nicht zu kompliziert zu machen, habe ich alles in das Dockerfile gesteckt. Wenn die Anwendung aber wächst, ist es sinnvoll, vieles wann immer möglich in zusätzlichen Dateien und Skripten unterzubringen. Insbesondere die pip-Abhängigkeiten sollten in eine Datei *requirements.txt* umziehen, und die uWSGI-Konfiguration kann in eine *.ini*-Datei wandern.

Als Nächstes müssen wir das Dockerfile so anpassen, dass auch das Skript berücksichtigt wird:

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8
WORKDIR /app
COPY app /app
COPY cmd.sh /  ❶

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"]  ❷
```

❶ Fügt das Skript zum Container hinzu.

❷ Ruft es per `CMD`-Anweisung auf.

Bevor wir die neue Version ausprobieren, müssen wir noch alte, laufende Container anhalten. Mit den folgenden Zeilen werden alle Container auf dem Host gestoppt und entfernt. Nutzen Sie sie *nicht*, wenn Sie Container weiterlaufen lassen wollen:

```
$ docker stop $(docker ps -q)
c4b3d240f187
9be42abaf902
78af7d12d3bb
$ docker rm $(docker ps -aq)
1198f8486390
c4b3d240f187
9be42abaf902
78af7d12d3bb
```

Jetzt können wir das Image mit dem Skript neu bauen und es ausprobieren:

```
$ chmod +x cmd.sh
$ docker build -t identidock .
...
$ docker run -e "ENV=DEV" -p 5000:5000 identidock
Running Development Server
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
```

Gut. Wenn wir es jetzt mit `-e "ENV=DEV"` laufen lassen, erhalten wir einen Entwicklungsserver, andernfalls den Produktionsserver.

Entwicklungsserver

Vielleicht reicht Ihnen der Standard-Python-Server bei der Entwicklung nicht aus, insbesondere wenn Sie mehrere Container miteinander verbinden. In diesem Fall können Sie uWSGI auch in der Entwicklung nutzen. Sie werden aber trotzdem die Möglichkeit anwenden wollen, die Umgebungen zu wechseln, um uWSGI-Features wie Live-Code-Reloading zu aktivieren – was im Produktivumfeld nicht genutzt werden sollte.

5.2 Mit Compose automatisieren

Um alles noch etwas einfacher zu gestalten, können wir mit der Automation noch weiter gehen. Docker Compose *(http://docs.docker.com/compose)* ist dazu gedacht, Docker-Entwicklungsumgebungen schneller erstellen zu können. Dabei werden YAML-Dateien genutzt, um die Konfiguration für eine Gruppe von Containern abzulegen, womit Entwickler sich das wiederholte und fehlerbehaftete Eingeben und Bereitstellen ihrer eigenen Lösung ersparen. Unsere Anwendung ist so einfach, dass wir davon nicht so viel haben, aber das kann sich schnell ändern, wenn alles etwas komplexer wird. Compose befreit uns davon, unsere eigenen Skripten für die Orchestrierung warten zu müssen – einschließlich des Startens, Verlinkens, Aktualisierens und Stoppens unserer Container.

Haben Sie Docker über die Docker Toolbox installiert, sollte Compose schon verfügbar sein. Wenn das nicht der Fall ist, folgen Sie den Anweisungen auf der Docker-Website unter *https://docs.docker.com/compose/install*. Ich habe in diesem Kapitel Version 1.4.0 genutzt, aber wir verwenden nur die grundlegende Funktionalität, daher sollte alles nach 1.2 in Ordnung sein.

Erstellen Sie eine Datei namens *docker-compose.yml* im Verzeichnis *identidock* mit folgendem Inhalt:

```
identidock:  ❶
  build: .   ❷
  ports:     ❸
    - "5000:5000"
  environment:  ❹
    ENV: DEV
  volumes:   ❺
    - ./app:/app
```

❶ Die erste Zeile deklariert den Namen des zu bauenden Containers. Es können in einer YAML-Datei mehrere Container (in der Compose-Sprache häufig als Services bezeichnet) definiert werden.

❷ Der Schlüssel `build` weist Compose an, das Image für diesen Container aus einem Dockerfile zu bauen, das im aktuellen Verzeichnis (`.`) liegt. Jede Con-

tainerdefinition benötigt entweder einen `build`- oder einen `image`-Schlüssel. `image`-Schlüssel erwarten das Tag oder die ID eines Image, das für den Container genutzt werden soll – genau wie das Imageargument von `docker run`.

❸ Der Schlüssel `ports` ist das direkte Äquivalent zu `-p` bei `docker run`, um Ports zu veröffentlichen. Hier bilden wir Port 5000 im Container auf Port 5000 auf dem Host ab. Ports können ohne Anführungszeichen spezifiziert werden, aber es ist besser, sie zu nutzen, da YAML ansonsten Anweisungen wie 56:56 als eine Zahl zur Basis 60 parst.

❹ Der Schlüssel `environment` entspricht dem Argument `-e` von `docker run`, mit dem Umgebungsvariablen im Container gesetzt werden. Hier setzen wir `ENV` auf `DEV`, um den Flask-Entwicklungswebserver zu starten.

❺ Der Schlüssel `volumes` entspricht dem Argument `-v` von `docker run`, um Volumes zu definieren. Hier bind-mounten wir das Verzeichnis *app* wie zuvor in den Container, um vom Host aus Änderungen am Code vornehmen zu können.

In den YAML-Dateien von Compose können noch viele weitere Schlüssel zum Einsatz kommen, die meist direkt zu Argumenten von `docker run` passen.

Führen Sie nun `docker-compose` aus, werden Sie nahezu das gleiche Ergebnis erhalten wie beim vorigen Ausführen von `docker run`:

```
$ docker-compose up
Creating identidock_identidock_1...
Attaching to identidock_identidock_1
identidock_1 | Running Development Server
identidock_1 | * Running on http://0.0.0.0:5000/
identidock_1 | * Restarting with reloader
```

In einem anderen Terminal können Sie das Ganze wieder testen:

```
$ curl localhost:5000
Hallo Docker!
```

Sind Sie mit dem Ausführen der Anwendung fertig, drücken Sie einfach Strg-C, um den Container zu stoppen.

Um zum uWSGI-Server zu wechseln, müssten wir in der YAML-Datei die Schlüssel `environment` und `ports` anpassen. Das erreichen Sie durch ein Ändern von *docker-compose.yml* oder durch das Erstellen einer neuen YAML-Datei für die Produktivumgebung, auf die Sie `docker-compose` mit dem Flag `-f` oder der Umgebungsvariable `COMPOSE_FILE` zeigen lassen.

5.2.1 Der Compose-Workflow

Die folgenden Befehle werden bei der Arbeit mit Compose häufig zum Einsatz kommen. Die meisten sind selbsterklärend und besitzen direkte Docker-Äquivalente, aber es lohnt sich, sie trotzdem einmal kennenzulernen:

`up`
: Startet alle Container, die in der Compose-Datei definiert sind, und sammelt die Ausgabe. Meist werden Sie das Argument `-d` verwenden, um Compose im Hintergrund laufen zu lassen.

`build`
: Baut alle Images aus Dockerfiles neu. Der Befehl `up` baut nur fehlende Images, daher verwenden Sie diesen Befehl, wenn Sie ein Image aktualisieren müssen.

`ps`
: Stellt Informationen zum Status der durch Compose verwalteten Container bereit.

`run`
: Ruft einen Container für das einmalige Ausführen eines Befehls auf. Damit werden auch alle verlinkten Container gestartet, sofern nicht das Argument `--no-deps` mitgegeben wird.

`logs`
: Gibt aggregierte und farbig hervorgehobene Logs für die durch Compose verwalteten Container aus.

`stop`
: Stoppt Container, ohne sie zu entfernen.

`rm`
: Entfernt gestoppte Container. Mit dem Argument `-v` werden auch alle durch Docker verwalteten Volumes entfernt.

Ein normaler Workflow beginnt mit dem Aufruf von `docker-compose up -d`, um die Anwendung zu starten. Die Befehle `docker-compose logs` und `ps` können genutzt werden, um den Status der Anwendung zu prüfen und um Hilfe beim Debuggen zu erhalten.

Nach Änderungen am Code rufen Sie `docker-compose build` auf, gefolgt von `docker-compose up -d`. Damit wird das neue Image gebaut und der laufende Container ausgetauscht. Beachten Sie, dass Compose alte Volumes aus dem Originalcontainer aufhebt, was heißt, dass Datenbanken und Caches über Container-Lebenszyklen hinweg bestehen bleiben (das kann verwirren, seien Sie also aufmerksam). Wenn Sie kein neues Image benötigen, aber die Compose-YAML-Datei angepasst haben, sorgt ein Aufruf mit `-d` dafür, dass Compose den Container durch einen mit den neuen Parametern ersetzt. Wollen Sie Compose zwingen, alle Container zu stoppen und neu zu erstellen, nutzen Sie das Flag `--force-recreate`.

Sind Sie mit der Anwendung fertig, sorgt ein Aufruf von `docker-compose stop` für ein Anhalten der Anwendung. Die gleichen Container werden mit `docker-compose start` oder `up` neu gestartet, wenn sich kein Code geändert hat. Mit `docker-compose rm` werden sie hingegen komplett gelöscht.

Einen vollständigen Überblick über alle Befehle finden Sie auf der Referenzseite von Docker unter *https://docs.docker.com/compose/reference*.

5.3 Zusammenfassung

Wir haben nun eine funktionierende Umgebung und können damit beginnen, unsere Anwendung zu entwickeln. In diesem Kapitel haben wir einiges kennengelernt:

- wie man die offiziellen Images einsetzt, um schnell eine portierbare und reproduzierbare Entwicklungsumgebung zu erhalten, ohne irgendwelche Tools auf dem Host zu installieren
- wie man Volumes nutzt, um dynamische Änderungen am Code vornehmen zu können, der in Containern läuft
- wie man mit einem einzelnen Container sowohl eine Produktiv- als auch eine Entwicklungsumgebung wartet
- wie man mit Compose den Entwickler-Workflow automatisiert

Docker hat uns eine vertraute Entwicklungsumgebung zur Verfügung gestellt, die alle notwendigen Tools enthält – und gleichzeitig können wir unseren Code schnell in einer Umgebung testen, die die Produktivumgebung widerspiegelt.

Es gibt trotzdem noch viel zu tun – insbesondere in Bezug auf Testen und Continuous Integration/Delivery, aber wir werden uns damit in den nächsten Kapiteln befassen, wenn wir mit der Entwicklung fortfahren.