Der goldene Käfig

Micro-Services in Java realisieren – Teil 2: Web-Apps in Docker-Umgebungen

Phillip Ghadir

In dem ersten Teil dieser zweiteiligen Reihe haben wir uns mit der Realisierung von Web-Apps mit dem leichtgewichtigen Framework DropWizard beschäftigt. In diesem Teil beschäftigen wir uns mit der offenen Plattform Docker. Damit lassen sich Umgebungen definieren und voneinander isolieren. Wie sich Docker von virtuellen Maschinen unterscheidet und was das für unsere Anwendungen bringt, stellt dieser Artikel vor.

Was ist Docker?

Docker ist eine offene Plattform für verteilte Systeme und richtet sich sowohl an Entwickler als auch an Systemadministratoren. Der Slogan „Build, Ship, and Run Any App, Anywhere" erinnert an das Motto von Java, „Write once, Run anywhere", geht aber darüber hinaus. Denn Docker erlaubt das Definieren von Laufzeit-Umgebungen für unsere Systeme – unabhängig davon, ob sie in der JVM laufen oder andere Laufzeitsysteme erfordern. So ist es möglich, unsere selbst gebauten Applikationen mit denselben Mechanismen wie andere Services – wie zum Beispiel E-Mail-Server, Datenbank-Server oder ähnliches – aufzusetzen.

Images und Container

Eine Umgebung wird in Docker durch ein Image definiert, das wiederum typischerweise durch ein sogenanntes Dockerfile definiert wird, das beschreibt, was in der Umgebung alles vorhanden sein muss und welcher Service beim Start der Umgebung automatisch gestartet werden muss. Man beachte: Docker erlaubt nur ein Skript zum Start auszuführen. Wer mehrere will, kapselt deren Aufrufe in einem eigenen Startskript.

Ein Docker-Image liegt in einem Repository oder auf der Platte. Wenn es gestartet wird, initialisiert Docker dafür einen separaten Container.

Image ist also ein statisches und Container ein dynamisches Element unserer Architektur. Man kann mehrere parallele Container für ein Image starten. Ob die Container dann auf einer oder auf mehreren (virtuellen) Maschinen laufen, spielt keine Rolle.

Docker in der Konsole

Bedient wird Docker über die Kommandozeile. Sie erlaubt das Erzeugen, Suchen, Herunterladen, Manipulieren und Einchecken von Images sowie auch das Verwalten von Containern.

Docker arbeitet mit einem lokalen Repository von Images, hat Zugriff auf Remote-Repositories, über die Abhängigkeiten aufgelöst werden können, und verhält sich bei der Manipulation von Images ein wenig wie ein modernes Versionsverwaltungssystem.

Leichtgewicht unter Linux

Das Schöne an Docker ist, dass es im Gegensatz zu virtuellen Maschinen (VMs) nur sehr wenig Overhead erfordert. Dazu nutzt Docker ein Linux-Containern (LXC) ähnliches Konzept, das auf Ebene des Linux-Kernels jeden Container isoliert. Das gilt für CPU-Kapazität, Threads, I/O, Dateisysteme, Netzwerk und sogar Benutzerberechtigungen. Jeder Anwendung eine definierte Umgebung bereitzustellen, die unabhängig von allen anderen Anwendungen ist, erfordert eigentlich nur deren Lauffähigkeit in einer Linux-Umgebung. Abbildung 1 verdeutlicht den Unterschied zwischen Docker-Containern und VMs.

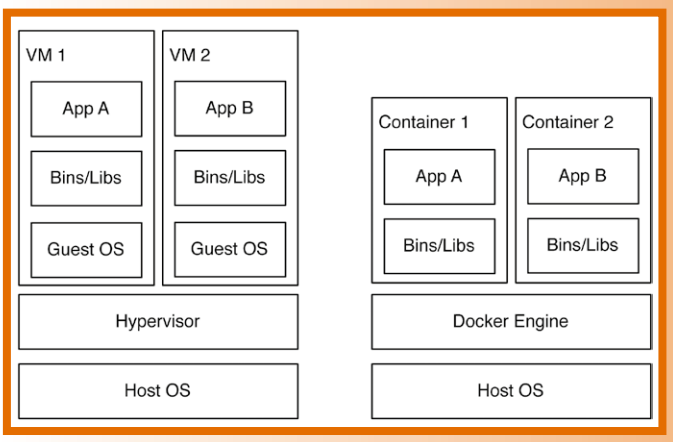

Abb. 1: Gegenüberstellung der Ansätze von virtuellen Maschinen und Docker

Virtualisiert auf Windows und OSX

Wer als Host-System statt auf Linux auf Microsoft Windows oder Apple OSX setzt, verwendet Stand Juli 2014 am Besten eine kleine Linux-Umgebung in einer virtuellen Maschine, in der dann Docker für die Provisionierung von leichtgewichtigen Umgebungen verwendet werden kann. Mit boot2docker [bo2d] gibt es eine vorgefertigte Lösung für beide Betriebssysteme, die ich seit einiger Zeit in beiden Umgebungen verwende und die gut funktioniert.

Micro-Services verteilen

Im ersten Teil dieses Beitrags [Gha14] haben wir uns mit dem Bau eines Single-Purpose Monolithen – auch bekannt als Micro-Service – beschäftigt. Mit Hilfe des DropWizard-Frameworks sind wir in der Lage, leichtgewichtige eigenständige Applikationen zu bauen, die über Web-Schnittstellen für Anwender und für Admins verfügen sowie einfach über die Kommandozeile gestartet, gestoppt und konfiguriert werden können.

Das Zusammenfügen einer komplexen Software aus einzelnen Micro-Services stellt uns nun vor zwei Herausforderungen, die wir im Anschluss einzelnen angehen:

- Definition der statischen Umgebung für den Micro-Service,
- Bereitstellung der zuliefernden Services.

Dockerfiles und Images

Beginnen wir als erstes mit dem Definieren der statischen Umgebung für unseren Micro-Service.

Im ersten Teil [Gha14] haben wir den Micro-Service SimpleDocumentStore zu einem ausführbaren JAR zusammengebaut, das alle benötigten Bibliotheken und Frameworks enthält und das auf der Kommandozeile per

```
java -jar SimpleDocumentStore-1.0-Snapshot.jar
  server some_config_file.yml
```

aufgerufen wird. Wir brauchen also neben unserem JAR mindestens eine Konfigurationsdatei im YAML-Format sowie eine Java-Runtime. Das sind also drei Dinge, die wir in einem Docker-Image benötigen.

Beginnen wir mit dem Dockerfile, das festlegt, wie das entsprechende Image gebaut werden soll.

Dockerfile für den SimpleDocumentStore

In einem Dockerfile kann man mit dem Hash-Zeichen „#“ eine Kommentarzeile einleiten. Sonst kann man per „<ANWEISUNG> <Parameter-Liste>“ verschiedene Aktionen ausführen. Dockerfiles beschreiben mit Hilfe weniger primitiver Anweisungen die einzelnen Schritte, die für den Bau eines Docker-Images nötig sind. Der Kasten „Primitive im Dockerfile“ erläutert die einzelnen Anweisungen. Im Folgenden begnügen wir uns mit den Anweisungen, die wir für unseren `SimpleDocumentStore` benötigen.

Ein Dockerfile startet stets mit der `FROM`-Anweisung, die die Basis für alle folgenden Anpassungen der zu definierenden Umgebung bildet. Mit der `MAINTAINER`-Anweisung lässt sich der Autor des Dockerfiles benennen und dokumentieren.

Mit der `RUN`-Anweisung können Unix-Kommandos in der Umgebung aufgerufen werden, die aus dem aktuellen Zustand des zu bauenden Images einen neuen Zustand erzeugen. `RUN` gibt es in zwei Varianten: In der Shell-Form folgt nach der Anweisung das Shell-Kommando. In der Exec-Form folgt ein Array, in dem an erster Stelle das Kommando und in den folgenden die Parameter folgen. Hier ein Beispiel, das zwar nicht das Image ändert, aber die Syntax veranschaulicht

```
# entweder in der Shell-Form
RUN echo "Hallo Welt"
# oder in der Exec-Form
RUN ["echo", "Hallo Welt"]
```

Die `CMD`-Anweisung deklariert, welches Kommando beim Starten eines Docker-Containers ausgeführt werden soll. Es wird stets nur die letzte `CMD`-Anweisung berücksichtigt, selbst wenn in einem Dockerfile mehrere davon existieren. Für `CMD` wird üblicherweise die Exec-Form bevorzugt.

Mit der `EXPOSE`-Anweisung zeigen wir Docker, dass in dem Container auf einem Netzwerk-Port gehorcht wird. Das werden wir noch benötigen, wenn wir mehrere Docker-Container miteinander verbinden wollen.

Per `ADD`-Anweisung können wir in unser Image Dateien aus dem Dateisystem des Hosts in das Docker-Image kopieren.

Mit den Anweisungen können wir nun das Dockerfile für die Laufzeitumgebung unseres Micro-Services – wie in Listing 1 dargestellt – schreiben. Zu beachten ist hier, dass in der ersten Zeile das `XXXXXX` angemessen ersetzt werden muss. Mehr dazu weiter unten.

```
FROM XXXXXX/Java8
MAINTAINER phillip.ghadir@innoq.com

RUN mkdir /usr/local/lib/sds/

ADD target/SimpleDocumentStore-1.0-Snapshot.jar/usr/
  local/lib/sds/
ADD configs/sds_qa_env_simple_auth.Yml/usr/local/lib/sds/

EXPOSE 8080
EXPOSE 8081
WORKDIR /usr/local/lib/sds/
CMD ["/usr/bin/java","SimpleDocumentStore-1.0-Snapshot.
  jar","sds_qa_env_simple_auth.yml"
```

Listing 1: Dockerfile für den SimpleDocumentStore Micro-Service

Das Dockerfile in der Entwicklungsumgebung

Mit Hilfe des Dockerfiles kann Docker nun ein Image bauen. Dazu legen wir einen Verzeichnisbaum in der Build-Umgebung an, der das Dockerfile an der Wurzel und alle Abhängigkeiten zur lokalen Umgebung enthält.

Das Dockerfile aus Listing 1 definiert zwei Abhängigkeiten zur lokalen Umgebung, erkennbar an den `ADD`-Anweisungen. Parallel zum Dockerfile

Primitive im Dockerfile

Die folgende Tabelle liefert einen kurzen Überblick über die Anweisungen, die in einem Dockerfile zur Verfügung stehen. Erläutert werden sie in der Docker-Referenz-Dokumentation (siehe [docker-ref]).

FROM	Gibt das Basis-Image an
MAINTAINER	Gibt den Autoren des Dockerfiles an
RUN	Führt im aktuellen Image eine Anweisung aus und erzeugt eine neue Image-Version
CMD	Deklariert, welches Skript beim Container-Start aufgerufen werden soll. Es wird stets nur die letzte CMD-Deklaration berücksichtigt
EXPOSE	Deklariert einen Port, der nach Außen exponiert werden können soll
ENV	Dient dem Setzen von Umgebungsvariablen
ADD	Kopiert zur Build-Zeit aus der lokalen Umgebung Dateien an die gewünschte Stelle im Image. Unter Umständen müssen hinterher mit RUN … Owner, Gruppen und Berechtigungen gesetzt werden
COPY	Analog zu ADD
ENTRYPOINT	Analog zu CMD, ermöglicht aber die Übergabe von Kommandozeilenparametern beim Aufruf des Docker-Containers
VOLUME	Damit lassen sich Mount Points definieren, um vom Host aus auf Bereiche im Dateisystem des Containers (wie z. B. Log-Dateien) zugreifen zu können.
USER	Definiert mit welchem User alle folgenden RUN-Aufrufe innerhalb des Images aufgerufen werden.
WORKDIR	Definiert das Verzeichnis innerhalb des Images, in dem alle folgenden Aufrufe ausgeführt werden
ONBUILD	Ermöglicht das Definieren von nicht geschachtelten Trigger-Aktionen, die ausgeführt werden, wenn ein Image gebaut wird, das als FROM-Anweisung dieses Image referenziert. So lassen sich zum Beispiel Zeitstempel aktualisieren oder Artefakte kompilieren

muss das Verzeichnis target/ mit der darin enthaltenen JAR-Datei unseres Micro-Services liegen. Ebenfalls parallel zum Dockerfile muss im Verzeichnis configs/ die Konfigurationsdatei mit dem unaussprechlichen Namen sds_qa_env_simple_auth.yml liegen.

Wir wählen also die Verzeichnisse am besten so, dass wir das Dockerfile in die Projektwurzel (oder bei entsprechendem Setup: ins target/-Verzeichnis) legen. Dann können wir das Docker-Image in dem Verzeichnis mit dem Dockerfile mit dem folgenden Kommando aus der Konsole bauen

```
docker build .
```

Beim ersten Aufruf muss Docker erst einmal alle Abhängigkeiten auflösen und die Images der Zwischenstufen bauen, die nötig sind, um darauf die Anweisungen unseres eigenen Dockerfiles auszuführen. Die Abhängigkeiten kommen in unser Dockerfile über die `FROM`-Anweisung.

Nachdem alle Abhängigkeiten heruntergeladen und unser Image gebaut wurde, endet `docker build .` mit der Ausgabe unserer Image-ID, die wir benötigen, wenn wir das Image ausführen wollen. Weil wir ja eigentlich wissen, wie wir unser Image nennen wollen, benennen wir es am Besten gleich richtig. Hier kürze ich SimpleDocumentStore einfach mit „sds“ ab:

```
docker build -t sds .
```

Sobald alle Abhängigkeiten lokal verfügbar sind, beansprucht der Build praktisch kaum noch Zeit.

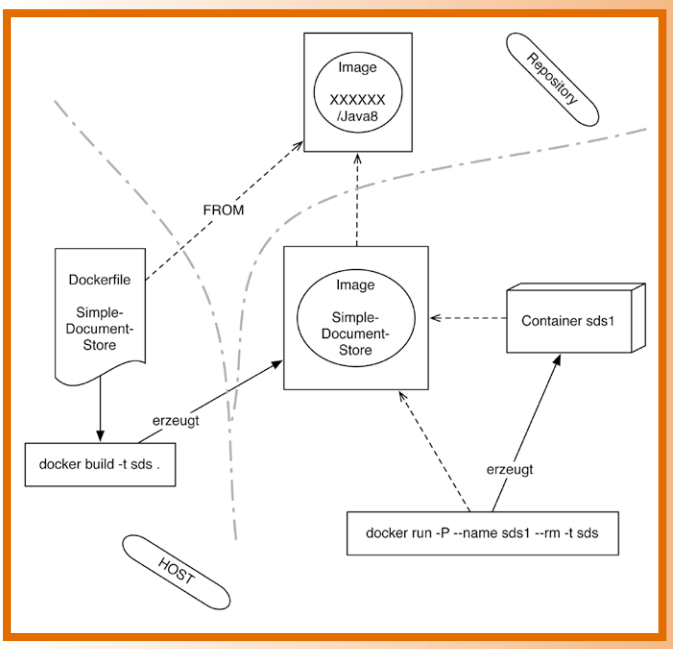

Abb. 2: Zusammenhang zwischen Host-Umgebung, Docker-Umgebung und Docker-Container

Den Micro-Service starten

Unseren Service können wir nun in der gleichen Konsole mit dem folgenden Kommando ausführen

```
docker run –d –p 8080:8080 –t sds
```

Wer will, kann anstatt das „-t sds“ auch die Image-ID verwenden. Aber die ändert sich ständig, wenn wir etwas an der Umgebung ändern, daher ist das Verwenden des Tags sinnvoller.

Bei dem Aufruf haben wir Docker mit „-p 8080:8080“ explizit angewiesen, dass der Port 8080 innerhalb des Containers von außen über Port 8080 zugreifbar ist. Wir haben zwar im Image bereits konfiguriert, dass der Port exponiert wird. Dennoch erfordert Docker, die Freigabe von Ports eines Containers beim Start explizit anzugeben. (Anstatt jeden einzelnen freizugeben, könnte man mit dem Parameter „-P“ auch ohne weitere Angaben alle im Image exponierten Ports freigeben.)

Unser SimpleDocumentStore läuft nun also isoliert im Docker-Container und ist von außen über Port 8080 erreichbar.

Den Micro-Service benutzen

Wer Docker direkt unter Linux einsetzt, kann jetzt im Browser auf `http://localhost:8080/dir` den SimpleDocumentStore öffnen. Wer allerdings nicht unter Linux entwickelt, muss noch etwas tun.

Sowohl in meiner Windows-Umgebung als auch in meiner OSX-Umgebung läuft Docker in einer Virtual Box [bo2d]. Auf dem Windows-Rechner habe ich meine Virtual Box [ovb] mit einem zweiten Host-Only-Netzwerk samt Port-Forwarding von Port 8080 der virtuellen Maschine auf meinem Windows-Rechner auf den gleichen Port konfiguriert.

Wenn ich nun mit meinem Browser auf dem Windows-Rechner `http://localhost:8080/dir` `http://localhost:8080/` abrufe, sehe ich nun die Antwort vom SimpleDocumentStore. Jippi!

In meiner OSX-Umgebung habe ich ebenfalls einen zusätzlichen Host-Only-Adapter aber ohne Port-Forwarding konfiguriert. Um in dieser Konstellation an den Micro-Service zu gelangen, muss ich also erst einmal die IP-Adresse der Virtual Box ermitteln. In der Shell der Virtuellen Maschine – der Docker-Umgebung – kann man mit dem Aufruf von „ifconfig“ die IP der zweiten Ethernet-Karte (dem Host-Only-Adapter) ermitteln. Im Browser sehe ich die Antwort des Micro-Service beim Abrufen der entsprechenden URL – bei mir: `http://192.168.58.101:8080/dir`.

Auf den Schultern von Entwicklern

Wie man an Listing 1 sehen kann, braucht es für unseren Micro-Service mit Docker gar nicht viel. Anstatt ausgehend von einer minimalen Linux-Umgebung schrittweise alle Abhängigkeiten für die Installation von Java selbst zu installieren, um dann die zwei spezifischen Dateien unseres SimpleDocumentStores hineinzukopieren, setzt unser Dockerfile auf einem Docker-Image namens `XXXXXX/Java8` auf, das bereits alle vorausgesetzten Abhängigkeiten beinhaltet, sodass wir uns auf die für uns wesentlichen Elemente konzentrieren können.

Der Haken: Es gibt zum Zeitpunkt des Schreibens kein Image mit dem referenzierten Namen. Ich wollte nicht durch das Wählen eines Basis-Images eines besonders hervorheben. Jeder Interessierte muss also selbst ein entsprechendes Basis-Image finden und auswählen.

Trusted Builds und Docker-Hub

Das Erstellen eines Docker-Images sollte stets mit dem Schreiben eines Dockerfiles beginnen. Die erste Anweisung im Dockerfile – `FROM` – referenziert bekanntlich das Basis-Image, auf dem nun eine neue Umgebung definiert wird.

Docker verfügt über ein offizielles Repository namens Docker-Hub [docker-hub], in dem sowohl offizielle Images (von Docker selbst) als auch vertrauenswürdige Images gelistet sind, zu denen die Dockerfiles bekannt und öffentlich einsehbar sind.

Wer sich gerade in der Konsole befindet und ein passendes Image für Java 8 benötigt, kann mittels

```
docker search Java8
```

passende Images auflisten lassen. Ich bevorzuge vertrauenswürdige Builds, die man in der Konsolenausgabe an dem `[Ok]` erkennen kann, die angemessen beschrieben sind und deren Dockerfile nur tut, was ich gern hätte.

Man kann natürlich auch über den Web-Browser suchen und durch die Alternativen stöbern. Nach dem Aussuchen eines geeigneten Docker-Images müssen wir nur in unserem Dockerfile die `FROM`-Anweisung aktualisieren. Damit ist unser Docker-Image wohl definiert. Jetzt können wir mit `docker build …` bauen, wie gehabt.

Was Docker nützt

Selbst in einer Umgebung mit nur zwei Prozessorkernen und Hyper-Threading bleiben sowohl der Host als auch die Docker-Container reaktionsfreudig. Auf derselben Maschine macht sich der Overhead mehrerer virtueller Maschinen sofort negativ bemerkbar. Docker ist daher ein nützlicher Begleiter für die Entwicklung. Selbst die Verwendung von Docker in einer VM [bo2d] ist eine vielfach agilere Lösung als das parallele Aufsetzen von VMs.

Zudem verfolgt Docker das Konzept, dass eine Umgebung beziehungsweise ein Image unveränderlich ist. Wenn ich also in einem Image mittels der `RUN`-Anweisung eine Änderung an der Umgebung vornehme, verfährt Docker dabei wie eine Versionsverwaltung und legt eine neue Version der Umgebung an. Das Beschreiben der erforderlichen Umgebung vereinfacht das Konfigurationsmanagement ungemein. Änderungen am Dockerfile – und damit an der Umgebung – lassen sich so effizient versionieren, ohne den Überblick zu verlieren.

System aus mehreren Systemen

Wenn wir nun ein System bauen, das verschiedene andere Micro-Services benötigt, könnten wir sie alle in einem Docker-Image zusammenfassen. Das widerspräche aber ein wenig dem Grundgedanken, Micro-Services als eigenständige Applikationen einfach starten und stoppen zu können.

Zur Erinnerung: In [Gha14] haben wir die Merkmale sogenannter 12-Factor-Apps aufgeführt. Eines lautet: „Skaliert wird über das Starten zusätzlicher paralleler Prozesse (Strategie: horizontal scale out).“

Wir wollen bei Bedarf also eine zweite Instanz eines überlasteten Service hochfahren und die Last auf beide verteilen, ohne alle davon benötigten Services erneut zu instanziieren. Um eine Instanz mit bestehenden zu liefernden Services zu verbinden, muss man dem Docker-Container mitteilen, mit welchen Containern er verbunden werden soll.

Container-Links

Wir können einen Container mit einem Namen versehen. Wir stoppen unseren Container mit dem SimpleDocumentStore und starten ihn mit zwei zusätzlichen Argumenten neu

```
docker run –d -p 8080:8080 --name docStore --rm –t sds
```

Mit diesem Aufruf vergeben wir nun den eindeutigen Namen „docStore“ für den Container. Dieser Name muss eindeutig für alle Container in der Docker-Umgebung sein. Wir erinnern uns: Ein Docker-Container ist ein dynamisches Element. Ein Docker-Container ist eine Laufzeit-Instanz, die vielleicht gerade ausgeführt wird, oder auch nicht.

Der Parameter „--rm“ sorgt dafür, dass der Name beim Beenden des Containers gelöscht wird, sodass wir ihn beim erneuten Aufruf wieder vergeben können. Ohne den Parameter bliebe der Name auch nach Beenden des Containers vergeben und wir könnten keinen zweiten Container mit diesem Namen instanziieren.

Haben wir das „--rm“ einmal vergessen, geht auch

```
docker rm <name_des_containers>
```

Wenn wir jetzt bereits den Client unseres SimpleDocumentStores in einem separaten Docker-Image namens RatingService gekapselt hätten [GhaSch14], könnten wir diesen mit

```
docker run –d --name rating1 --link docStore:docArchive
```

starten. Der Parameter „--link docStore:docArchive“ sorgt dafür, dass der Container `rating1` abhängig von dem Container `docStore` ist, den es innerhalb von `rating1` als `docArchive` kennt.

Container, die nur von sie verlinkenden Containern aus benutzt werden sollen, brauchen nicht einmal die Ports beim Aufruf von „docker run …“ – mit „-p“ oder „-P“ – durchgeschleift werden. Docker verbindet verlinkte Container automatisch direkt und erlaubt abhängigen verlinkten Containern den Zugriff auf alle vom Image exponierten Ports.

Zusammenfassung

Wir haben in [Gha14] mit DropWizard das fachliche Beispiel des Rating-Service aus [GhaSch14] aufgegriffen und einen Baustein des Gesamtsystems – das Archiv-System – herausgegriffen. Das haben wir zu einem eigenständigen Micro-Service realisiert, den wir SimpleDocumentStore genannt haben und der den Regeln der 12-Factor-Apps gehorcht. Insbesondere haben wir den SimpleDocumentStore zu einem ausführbaren, alles beinhaltenden JAR gepackt, das wir auf der Kommandozeile einfach starten können.

Mit Docker provisionieren wir die Laufzeitumgebungen. Das ist insbesondere im Vergleich zu normalen virtuellen Maschinen sehr leichtgewichtig, benötigt aber ein Linux als Host. Es ermöglicht das Bilden von Systemverbünden mit einfachen Mitteln und lässt sich komplett über die Kommandozeile steuern. Mit Hilfe eines zentralen Repository namens Docker-Hub lassen sich Images für viele Umgebungen bereits finden und darauf aufbauend eigene Umgebungen definieren. Das kann man auch interaktiv festlegen, aber der bessere und nachvollziehbare Weg geht über ein Dockerfile, mit dem die Inhalte eines Images beschrieben werden.

Docker-Container sind Laufzeitinstanzen von Docker-Images. Sie können mit Namen versehen und miteinander über Namen verlinkt werden. Dadurch bildet Docker geschützte Verbindungen zwischen den Containern über alle in den Image-Definitionen exponierten Ports.

Docker verwendet eine sehr intuitive Standard-Netzwerk-Konfiguration, die sowohl das parallele Instanziieren mehrerer gleichartiger Container ohne zusätzliche Konfiguration zulässt als auch Schutz vor ungewünschtem Zugriff bietet.

Ausblick

Wenn man ein komplexes System bestehend aus mehreren Teilsystemen baut, kann man um Docker herum Funktionalität bauen, mit der das Benennen und das Verbinden von Containern sowie das automatische Anmelden bei Lastverteilern usw. leichter fällt. Es gibt bereits verschiedene Ansätze dazu, so etwas allgemeingültig bereitzustellen. Ein Beispiel dafür ist [Decking]. Das gehört aber nun nicht mehr in diesen Artikel.

Links

[bo2d] boot2docker, `https://github.com/boot2docker/boot2docker`
[Decking] `http://decking.io`
[Docker] `http://docker.io`
[docker-hub] `https://registry.hub.docker.com/`
[docker-ref] `https://docs.docker.com/reference/builder/`
[Gha14] Ph. Ghadir, Micro-Services in Java realisieren – Teil 1: Leichtgewichtige Apps mit DropWizard, in: JavaSPEKTRUM, 4/2014, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2014/04/ghadir_JS_04_14_uVvU.pdf`
[GhaSch14] Ph. Ghadir, Ph. Schirmacher, Domain-Driven Design in Clojure, in:JavaSPEKTRUM, 2/2014, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2014/02/ghadir_schirmacher_JS_02_14_eveR.pdf`
[ovb] Oracle Virtual Box, Benutzer-Handbuch, `http://download.virtualbox.org/virtualbox/UserManual.pdf`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com