01-2016 | Frühjahr | www.ijug.eu

# Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

# Java ist auf dem richtigen Kurs

## Funktionale Programmierung
Java 8 und Haskell

## Docker
Überblick und Infrastruktur-Deployment

## Aktuell
Software-Lizenzrecht für Entwickler

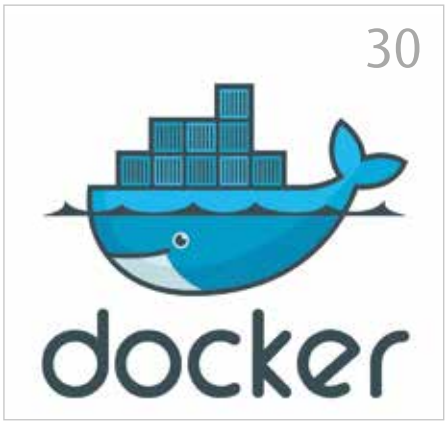
## RESTful-Microservices
Dropwizard und JAX-RS


D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 01

iJUG Verbund


Die Brownies Collections Library stellt Alternativen bereit




Einsatzmöglichkeiten für Docker


3 Editorial

5 Das Java-Tagebuch
*Andreas Badelt*

8 Software-Lizenzrecht für Entwickler
*Antje Kilián*

11 Funktionale Programmierung mit Java 8 und Haskell
***Nicole Rauch***

15 Java als Integrationslösung in einer gewachsenen Anwendungslandschaft
*Claus Straube*

20 WildFly-Installationen parametrisieren und mit Git verwalten
*Martin Welß*

23 Generics, Type Erasure und Fallstricke in der Praxis
*Michael Müller*

26 High-Performance Lists in Java
*Thomas Mauch*

30 Demystifying Docker
*Bernd Fischer*

36 Infrastruktur-Deployment mit Ansible und Docker
*Robert Reiz*

40 Good bye Swing – hello JavaFX
*Christoph Rein und Stefan Kühnlein*

46 2000 Zeilen Java oder 50 Zeilen SQL?
*Lukas Eder*

51 Puppet für Entwickler
*Sebastian Hempel*

55 RESTful-Microservices mit Dropwizard
*Felix Braun*

60 RESTliche Featuritis – modulare Anwendungen mit JAX-RS
*Markus Karg*

63 DukeCon ist mehr als eine Javaland-App!
*Gerd Aschemann*

64 „Vor diesem Hintergrund ist die starke Community das Rückgrat von Java …"
*Interview mit Björn Martin*

65 Entwurfsmuster – Das umfassende Handbuch
*gelesen vom Daniel Grycman*

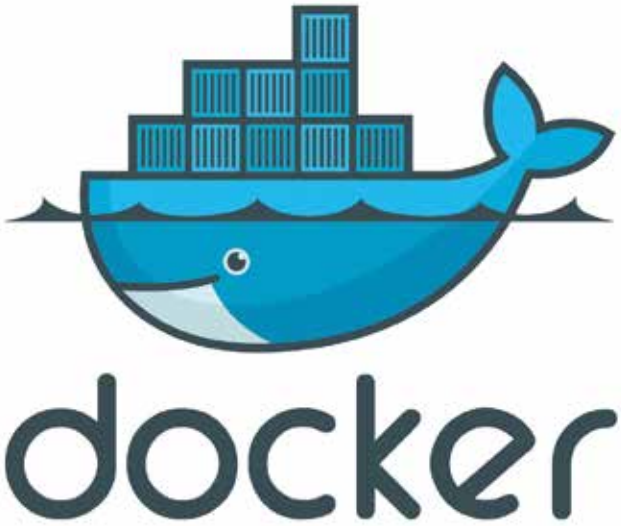
66 Inserentenverzeichnis

66 Impressum



Infrastruktur-Deployment mit Docker


RESTful-Microservices mit Dropwizard




# Demystifying Docker

***Bernd Fischer, MindApproach GmbH***

*Die Themen „Docker" und „Container" sind zurzeit auf jeder größeren Konferenz und in (fast) jeder Zeitschrift omnipräsent vertreten. Handelt es sich dabei tatsächlich um eine neue zukunftsträchtige Technologie oder ist es nur alter Wein in neuen Schläuchen? Vor allem: Was bedeutet es für den (Java-)Entwickler? Was kommt auf ihn zu und was verändert sich für ihn?*

Dieser Artikel zeigt ausgehend von einer (sehr) einfachen Java-Web-Anwendung einige Einsatzmöglichkeiten für Docker auf (Strategien) und beschreibt Konsequenzen für den Aufbau der Kern-Entwicklungsumgebung, mit anderen Worten: für den Entwickler-Arbeitsplatz. Nach einer kurzen Einführung in das Thema wird eine einfache Java-Web-Anwendung in mehreren Schritten „dockerisiert" und anhand dieses Beispiels einige Einsatzmöglichkeiten für Docker in der Java-Entwicklung diskutiert.

Die Antwort auf die Frage, warum sich ein Java-Entwickler mit dem Thema „Docker" beschäftigen sollte, lässt sich im Rahmen dieses Artikels nicht annähernd beantworten. Um sich jedoch nicht gänzlich um die Antwort zu drücken, zählt der Autor einige grundsätzliche Aspekte auf, die ihn auf diesen Weg führten. Eine ganz wesentliche Einflussgröße war und ist die DevOps-Bewegung, die für ihn als Software-Entwickler durch den Ausspruch von Werner Vogels „You build it you run it" *[1]* auf den Punkt gebracht wird. Der Bogen lässt sich dann weiter über „Continuous Delivery" *[2]* hin zu „Automate Almost Everything" und „Infrastructure as Code" beziehungsweise „Immutable Infrastructure" spannen. Konkret beschäftigte er sich vor allem damit, die Aufgabenstellung, die verschiedenen Arbeits-Umgebungen innerhalb des Entwicklungsprozesses, also die Entwickler-Workspaces und die Test- beziehungsweise QS-Landschaften, so identisch wie sinnvoll möglich mit der Produktivumgebung zu gestalten.

## Was ist Docker?

Docker realisiert eine spezielle Form der Virtualisierung, die sich vor allem von anderen bekannten Vertretern wie VirtualBox, VMware, Parallels oder KVM dadurch unterscheidet, dass keine mehr oder weniger vollständige Nachbildung eines Rechners inklusive Hardware, BIOS etc. erfolgt, sondern im Grunde lediglich Prozesse voneinander isoliert sind *(siehe*

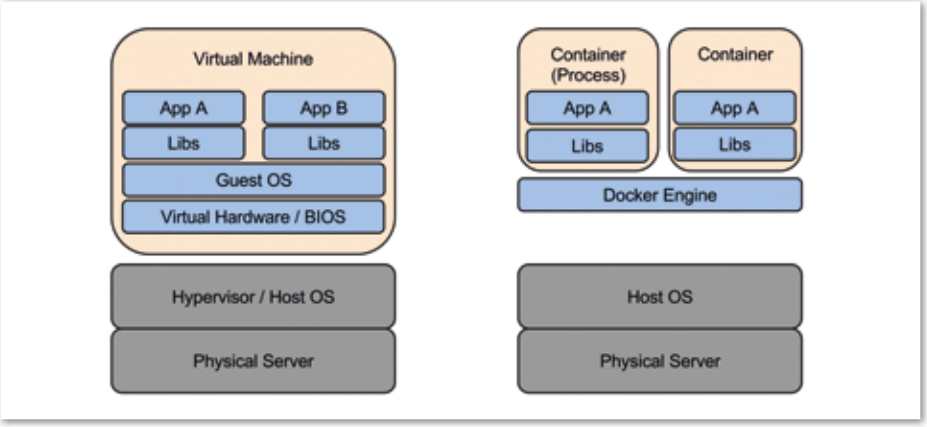



*Abbildung 1: Container vs. Virtual Machine (VM)*

*Abbildung 1)*. Aus diesem Grund wird diese Form häufig auch „Prozess-Virtualisierung" genannt (siehe auch *[4]*, Kapitel 1.3, Virtualisierungstechniken).

Ein Prozess, auch wenn er in einem Container, wie diese virtuelle Einheit oft auch genannt wird, beheimatet ist, wird im Grunde ganz normal durch das Betriebssystem des Host-Rechners gestartet und ausgeführt. Jedoch kann und wird die Sichtbarkeit von Betriebssystem-Ressourcen – dazu zählen die Prozess-Namen beziehungsweise -IDs, das Filesystem, die Netzwerk-Interfaces, die User etc. – zwischen den auf einem Rechner laufenden Containern untereinander und gegenüber dem Host eingeschränkt beziehungsweise so gesteuert, dass für den einzelnen Container der Eindruck entsteht, er wäre allein in seinem eigenem Universum. Auch wenn sich in diesem Aspekt Container und virtuelle Maschinen im Grundsatz stark ähneln, sind die durch (Voll- oder Para-) Virtualisierungstechnologien errichteten Mauern zwischen VM und Host sowie zwischen den VMs um einiges stärker als bei den Containern.

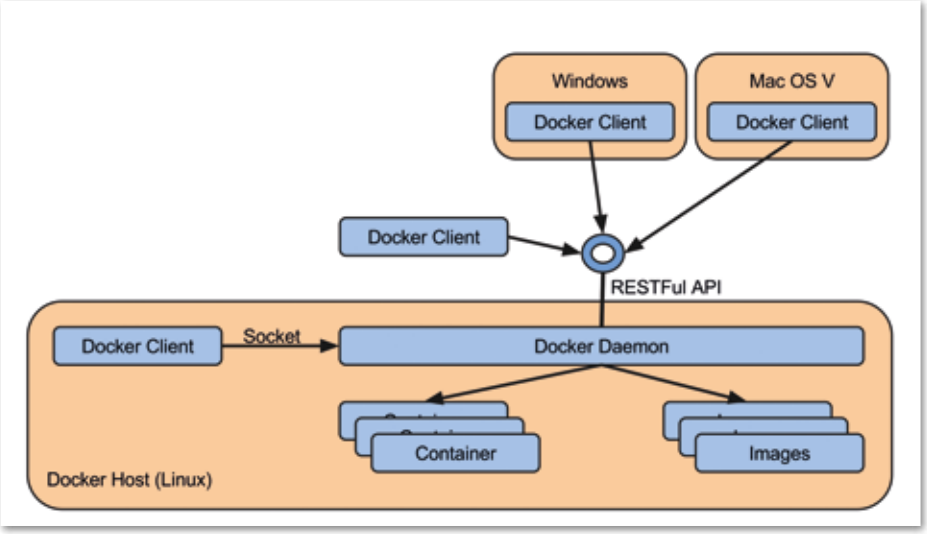



*Abbildung 4: Architektur der Docker Engine*

Auf der anderen Seite punkten diese durch ihre Leichtgewichtigkeit, die sich vor allem in einem praktisch kaum vorhandenen Overhead hinsichtlich CPU-Leistung und Speicherverbrauch im Vergleich mit außerhalb von Containern gestarteten Prozessen und erst recht im Vergleich zu virtuellen Maschinen zeigt. So lag Mitte September 2015 der Rekord der „Raspberry Pi DockerCon Challenge", bei der es um

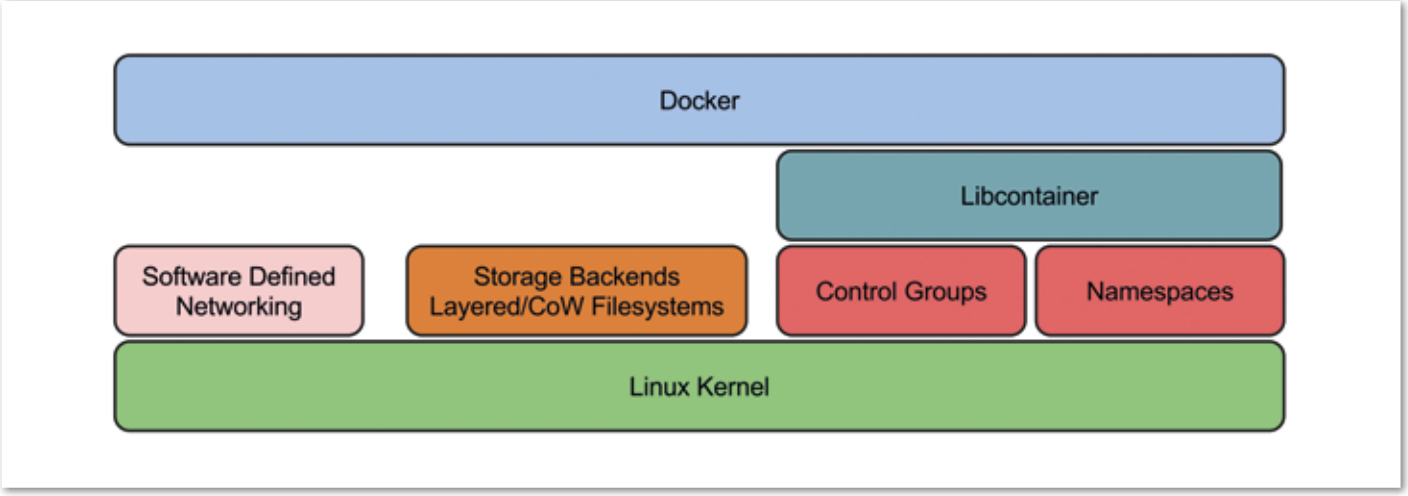



*Abbildung 2: Technologien des Linux-Kernels zur Prozess-Isolation*

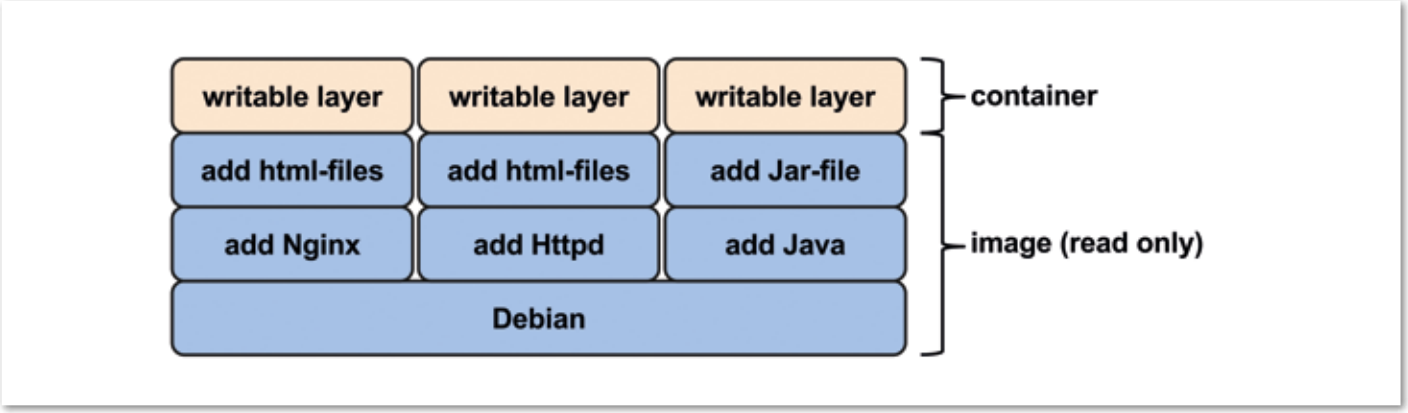



*Abbildung 3: Images und Container*

die größtmögliche Anzahl gleichzeitig laufender Web-Server auf einem einzelnen Pi 2 geht, bei 2.334.

Docker erweitert die für die eben beschriebene Funktionalität notwendige „Magic", die Bestandteil der Host-Betriebssysteme (im Moment vor allem Linux ab Kernel-Version 3.8, siehe *Abbildung 2*) ist, um eine intelligente Art und Weise der Speicherung und der Verteilung von Anwendungen, die später in einem Container ausgeführt werden sollen.

Statt aus einer oder mehreren monolithischen Dateien bestehen „Images", wie diese im Docker-Jargon genannt werden, aus einzelnen schreibgeschützten („read-only") Schichten („Layern"), die einzeln verteilt, referenziert und auf diese Art und Weise wiederverwendet werden können. Beim Start eines Containers werden die Schichten eines Image im Sinne eines Overlay-Filesystems zu einem Ganzen überlagert und durch eine zusätzliche beschreibbare Schicht, in der alle Änderungen, die durch den oder die im Container laufenden Prozesse hervorgerufen werden, gespeichert sind *(siehe Abbildung 3)*. Dieses Prinzip ist vergleichbar mit dem der Life-CDs, bei der die „Read-only"-CD durch ein beschreibbares Dateisystem ergänzt und dem Anwender, der praktisch von oben auf diesen Stapel von Dateisystemen schaut, als Einheit präsentiert wird.

## Docker verwenden

Das inzwischen „Docker Engine" genannte Docker-Kernprojekt verwendet die im vorigen Abschnitt beschriebenen Funktionalitäten und stellt darauf aufbauend ein (mehr oder weniger) RESTful-API über den „Docker-Daemon" für folgende Funktionen bereit:

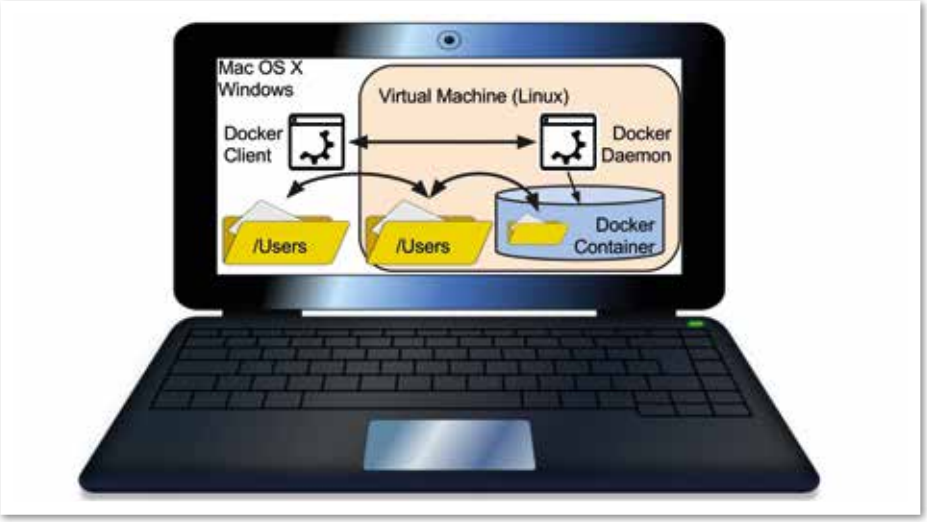



*Abbildung 5: Aufbau einer Entwicklungsumgebung mit Docker*

- Starten von Prozessen
- Verwalten der Images und der Container
- Organisation der (Netzwerk-)Verbindungen der Container untereinander und zur Außenwelt

Der ebenfalls in diesem Projekt enthaltene Kommandozeilen-Client (CLI) „docker" verbindet sich über einen Socket (lokal unter Linux) oder per Http(s) mit diesem Daemon *(siehe Abbildung 4)*.

Leider gibt es für die beiden am häufigsten auf Entwicklerarbeitsplätzen anzutreffenden Betriebssysteme Windows und Mac OS X (noch) keine native Implementierung des Docker-Daemon, sodass entweder auf eine Hilfskonstruktion unter Einsatz einer auf geringen Ressourcenverbrauch zugeschnittenen Linux-VM wie dem relativ bekannten Projekt „Boot2Docker" und einem trickreichen Mapping von Verzeichnissen *(Abbildung 5* verdeutlicht das Prinzip) oder auf einen Linux-Rechner zurückgegriffen werden muss.

```
HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/
plain;charset=UTF-8
Date: Tue, 29 Sep 2015 09:09:22
GMT
Server: Apache-Coyote/1.1
X-Application-Context: application

Hello World!
```

*Listing 1*

Das Projekt „Docker Machine" mit dem gleichnamigen Kommandozeilen-Client erzeugt und verwaltet im Zusammenwirken mit einer Vielzahl von Virtualisierungs- und Cloud-Anbietern, darunter Oracle VirtualBox für den lokalen Arbeitsplatz oder Amazon EC2, Microsoft Azure, Digital Ocean oder Google, in relativ kurzer Zeit fertig konfigurierte Docker-Hosts. Natürlich können sowohl die VMs als auch der Docker-Daemon manuell erstellt und konfiguriert werden, die

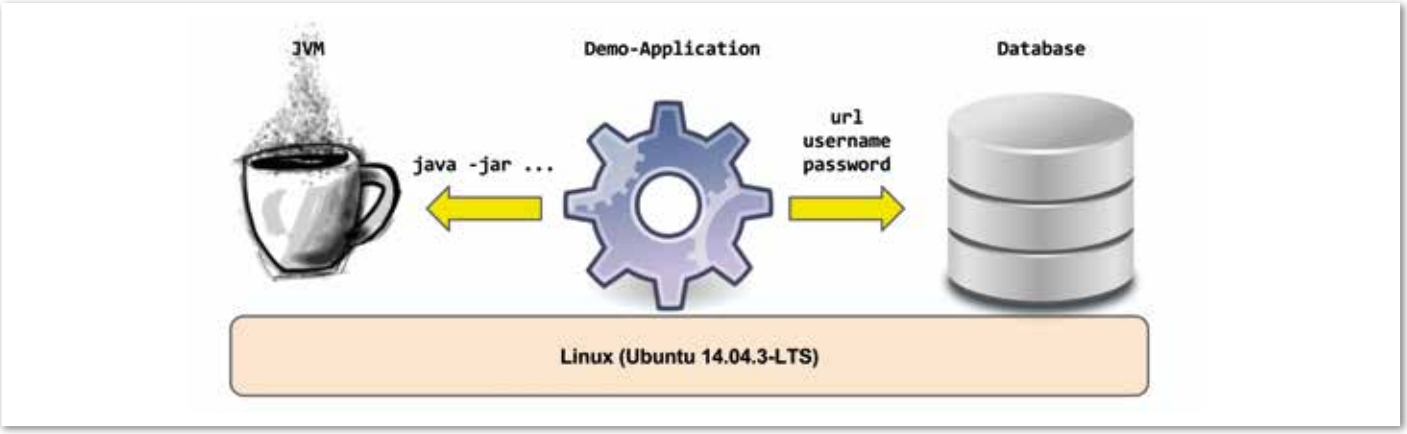



*Abbildung 6: Architektur der Beispielanwendung*

```
docker ps -a
CONTAINER ID  IMAGE  COMMAND  CRE-
ATED  STATUS  PORTS  NAMES
```

Listing 2

```
# s01-docker-compose.yaml
mysql:
  image: mysql:5.6.26
  expose:
    - "3306"
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=9876
    - MYSQL_USER=test
    - MYSQL_PASSWORD=1234
    - MYSQL_DATABASE=test
```

Listing 3

Anleitung hierfür würde allerdings den Umfang dieses Artikels sprengen.

## Ausgangspunkt: Die Spring-Boot-Anwendung

Das Beispiel ist im Grunde eine relativ einfache Java-Web-Anwendung, die auf dem Spring-Boot-Framework basiert, eine MySQL-Datenbank zur Datenspeicherung verwendet und zwei Http-Services (die Bezeichnung „REST" wird in diesem Zusammenhang nicht verwendet) anbietet: Bei Aufruf der URL „/" werden der statische Text „Hello World" ausgegeben sowie unter „/todos" die in der Datenbank gespeicherten To-dos gelesen und zurückgeliefert *(siehe Abbildung 6)*.

Für die weiteren Betrachtungen sind zwei Eigenschaften des Projekts wesentlich: Es besteht aus mehreren Komponenten (der JVM, der MySQL-Datenbank und der Anwendung selbst) und per Default werden Spring-Boot-Projekte als startbares „Fat-Jar-Archive" verpackt, also als eine Datei, die die Anwendung selbst einschließlich aller Abhängigkeiten (inklusive Embedded Tomcat-Server, aber ohne JVM) enthält. Nachdem mithilfe von Maven *[12]* per „mvn clean package" das erwähnte Projekt-Artefakt erstellt wurde, kann es im Anschluss mit dem relativ einfachen Kommandozeilenaufruf „java -jar target/demo-helloworld-web-1.0-Local.jar" gestartet werden. Die Funktionalität der Anwendung testet man am einfachsten unter Verwendung von Kommandozeilen-orientierten Tools wie „curl" oder – noch besser – „httpie" *[13]* per *„http http://local-host:8080"*. *Listing 1* zeigt die entsprechende Ausgabe.

Der vollständige Sourcecode der Beispielanwendung steht auf Bitbucket *[14]* zur Verfügung. Da die Entwicklung einer Spring-Boot-Anwendung nicht im Fokus dieses Artikels ist, wird für weitere Details auf die sehr gute Dokumentation des Projekts verwiesen *[18]*.

## Stufe 1: MySQL im Container

In den Betrachtungen zur Beispiel-Anwendung wurde bislang völlig außer Acht gelassen, wie und woher die notwendigen externen Bausteine wie die MySQL-Datenbank auf beziehungsweise in die Entwicklungsumgebung gelangen. Die klassische Variante der nativen Installation auf dem Entwicklungsrechner hat einen wesentlichen Nachteil: Mit der Zeit werden es immer mehr Tools und manchmal schließen sich verschiedene Versionen der gleichen Software auch gegenseitig aus.

Besser wäre es natürlich, wenn diese Tools separat und isoliert voneinander installiert und einfach rückstandslos entfernt werden könnten – genau das lässt sich mit Docker und Containern erreichen. Dazu erstellen wir als Erstes mittels „Docker Machine" einen lokalen Docker-Host und richten unsere Kommandozeile entsprechend ein, sodass der Docker-Client weiß, mit welchem Docker-Daemon er sprechen soll: „docker-machine create --driver=virtualbox demo" und „eval $(docker-machine env demo)".

Im Anschluss prüfen wir, ob wir uns mit dem Docker-Daemon der VM „demo" verbinden können, indem wir uns die vom Docker-Daemon gestarteten und verwalteten Prozesse (Container) anzeigen lassen. Diese Liste sollte unmittelbar nach der Installation leer sein *(siehe Listing 2)*.

Auch wenn wir im Moment nur einen Container erstellen, verwenden wir gleich „Docker Compose" *[09]*, das aufbauend auf der „Docker Engine" die Verknüpfung mehrerer Container auf einem Host erlaubt. Die notwendigen Informationen erhält „Docker Compose" aus einer speziellen Konfigurationsdatei im YAML-Format *[6, 19]*, die für unseren Anwendungsfall wie in *Listing 3* aussieht.

Mit „mysql" wird auf der obersten Hierarchieebene ein gleichnamiger Service definiert, der von einem oder mehreren Containern realisiert wird und auf dem Image „mysql:5.6.26" basiert, das automatisch aus der zentralen öffentlichen Docker-Image-Registry, dem „Docker Hub" *[20]* (vergleichbar mit Maven-Central), geladen wird. Mithilfe des Schlüsselworts „ports" wird der interne Port 3306 auf dem Host unter der gleichen Portnummer verfügbar gemacht und am Ende werden noch einige Umgebungsvariablen gesetzt, die die Autoren des Image für die Konfiguration vorgesehen haben *[11]*.

```
docker-compose \
  -f ./src/main/docker/s01-docker-
compose.yaml -p dmo stop
docker-compose \
  -f ./src/main/docker/s01-docker-
compose.yaml -p dmo rm
```

Listing 4

```
# Auszug aus s02-docker-compose.
yaml
app:
  image: java:8
  ports:
    - "8080:8080"
  environment:
    - SPRING_PROFILES_
ACTIVE=initdb
  working_dir: /opt/demo-hellow-
orld-web
  links:
    - mysql:mysql
  entrypoint: [
      "java",
      "-Djava.security.egd=file:/
dev/./urandom",
      "-jar",
"/opt/demo-helloworld-web/demo-
helloworld-web-1.0-Local.jar"
    ]
  volumes:
    - ../../../target:/opt/demo-
helloworld-web:ro
```

Listing 5

```
# Auszug aus application.yml
spring:
  datasource:
    # muss auf einer Zeile ste-
hen
    url: jdbc:mysql://
${MYSQL_PORT_3306_TCP_
ADDR:jsd2015.local}:
${MYSQL_PORT_3306_TCP_PORT:3306}/
${MYSQL_ENV_MYSQL_DATABASE:test}
    username: ${MYSQL_ENV_MYSQL_
USER:test}
    password: ${MYSQL_ENV_MYSQL_
PASSWORD:1234}
```

Listing 6

Ein einfacher Aufruf auf der Kommandozeile aus dem Startverzeichnis des Projekts erzeugt und startet („up") den Container als Daemon („-d"): „docker-compose \ -f ./src/main/docker/s01-docker-compose.yaml -p dmo up –d". Nach kurzer Wartezeit können wir mit den üblichen Tools wie der MySQL-Workbench und natürlich aus unserer Anwendung heraus auf die Datenbank zugreifen. Die dazu notwendige IP der „demo"-VM erfahren wir durch einen Aufruf von „Docker Machine": „docker-machine ip demo", 192.168.99.103. Wenn wir die MySQL-Datenbank nicht mehr benötigen, können wir sie sehr schnell und rückstandslos entfernen *(siehe Listing 4)*.

Das wir aber die MySQL-Datenbank im weiteren Verlauf noch verwenden wollen, müssen wir das nicht unbedingt ausprobieren. Falls doch: Auch nicht so schlimm. Einfach „Docker Compose" wieder mit „up -d" aufrufen und eine neue (leere) Datenbank wird erstellt.

## Stufe 2: Java im Container

Nachdem wir die MySQL-Datenbank in einen Container verpackt haben, wollen wir das natürlich auch mit dem Baustein „JVM" vollziehen. Im Gegensatz zur Vorgehensweise aus Stufe 1, bei der wir den durch den Container angebotenen „Service" über einen Netzwerkzugriff benutzen, muss der Java-Prozess direkten Zugriff auf das „.jar"-File besitzen.

Da dieses Artefakt während der Entwicklung relativ häufig neu erstellt wird, sparen wir uns für den Moment das direkte Einpacken und stellen stattdessen das Ausgabeverzeichnis des Projektes virtuell im Container zur Verfügung. Dazu ergänzen wir die Konfigurationsdatei um einen weiteren Service namens „app" *(siehe Listing 5)*.

Ausgangspunkt ist das Docker-Image „java:8" *[21]*, dass die JVM enthält. Danach stellen wir den Port 8080 auch auf dem Host-Rechner zur Verfügung und verbinden den Service „mysql" unter dem gleichnamigen Alias. Dadurch werden zusätzliche Informationen als Umgebungsvariable bereitgestellt, die wir in der Konfiguration unseres Projektes verwenden können *(siehe Listing 6)*.

Das Schlüsselwort „entrypoint" definiert das beim Start des Containers ausgeführte Kommando, während „volumes" die bereits angesprochene Bereitstellung des „target"-Verzeichnisses vom Host in den Container beschreibt. Im Shell-Skript „s02.docker-build-and-run-dev.sh" im Startverzeichnis des Projekts *[14]* ist ein typischer Ablauf vom Bauen des Projekts durch Maven über das Erzeugen und Starten der eben definierten „Docker Composition", die Kontrolle der Log-Datei (Hinweis: Beenden der Log-Ausgabe mittels Ctrl-C) und das Testen der beiden URLs bis zum Stoppen der Container dokumentiert.

## Stufe 3: Unpacked FatJar

In Projekten, die interpretierte Sprachen wie Python, JavaScript, HTML oder CSS verwenden, kann man mit diesem Stand schon halbwegs vernünftig arbeiten. Natürlich würde man in diesem Fall nicht das Verzeichnis mit den Kompilaten, sondern den Source-Tree im Container bereitstellen. Mit „vernünftig arbeiten" ist in erster Linie gemeint, dass durch Mappen Änderungen an den Sourcen zumindest theoretisch unmittelbar auch im Container sichtbar und damit testbar werden (in der Praxis gibt es einige etwas trickreichere Aufgabenstellungen, je nachdem, welcher Virtualisierer für die Docker-Host-VM zum Einsatz kommt).

Aufgrund der Art und Weise, wie die JVM Klassen lädt, macht eine analoge Arbeitsweise, die anstelle der Sourcen auf den entsprechenden Kompilaten aufbaut, weniger Freude, da relativ häufig, wenn auch nicht immer, die Anwendung zum Laden der Klassen neu gestartet werden muss. Eine Ausnahme bildet unter Umständen das Produkt „JRebel" der Firma ZeroTurnaround, jedoch wurde das im Rahmen der Arbeiten zu diesem Artikel nicht weiter untersucht.

Eine mögliche Realisierung für unser Beispielprojekt kann anhand des Shellskripts „s03.docker-build-and-run-dev.sh" und der „Docker Compose"-Konfigurationsdatei „s03-docker-compose.yaml" nachvollzogen werden. Dazu wird durch das Maven-Profil „unzipFatJar" der Inhalt des Fat-Jar-Files in ein Unterverzeichnis ausgepackt. Beim Bereitstellen der Kompilate im Container werden anstelle des Basis-Ausgabeverzeichnisses „target" die entsprechenden Unterverzeichnisse des ausgepackten Jar-Files und die des inkrementellen Eclipse-Compilers verlinkt,

```
# Änderungen an Sourcen durchführen und speichern
# Service "app" stoppen
docker-compose \
  -f ./src/main/docker/s03-docker-compose.yaml -p dmo stop app
# Service "app" starten
docker-compose \
  -f ./src/main/docker/s03-docker-compose.yaml \
  -p dmo start app
```

Listing 7

```
# Dockerfile für das Image "mapp/demo-helloworld-web04"
FROM java:8
MAINTAINER Bernd Fischer "bfischer@mindapproach.de"
ENV MODIFIED_AT 2015-09-26_1845

COPY demo-helloworld-web.jar /opt/demo-helloworld-web/
```

Listing 8

```
# bitte aus dem Startverzeichnis
# des Beispielprojektes aufrufen
docker build -t mapp/demo-hello-
world-web04:latest \
             -f $(pwd)/target/
workdir-docker/Dockerfile \
             $(pwd)/target/work-
dir-docker
```

Listing 9

```
app:
 image: mapp/demo-helloworld-
web04:latest
  ports:
    - "8080:8080"
  environment:
    - SPRING_PROFILES_
ACTIVE=initdb
  working_dir: /opt/demo-hello-
world-web/
  links:
    - mysql:mysql
  entrypoint: [
"java",
"-Djava.security.egd=file:/
dev/./urandom",
"-jar",
"/opt/demo-helloworld-web/demo-
helloworld-web.jar"
]
```

Listing 10

sodass Änderungen, auch an Java-Sourcen, nach einem Neustart des Containers zur Verfügung stehen *(siehe Listing 7)*.

## Stufe 4: Docker-Image für die Produktion

Für die Installation von Test-/QS- und Produktiv-Umgebungen ist es nicht sinnvoll, das Fat-Jar-File und den Java-Container getrennt voneinander auszuliefern. Besser ist es, so wenig wie möglich Artefakte berücksichtigen zu müssen. Daher packen wir das Jar-File und JVM gemeinsam in ein Docker-Image, dessen Aufbau in einem sogenannten „Dockerfile" beschrieben wird *(siehe Listing 8)*.

Die einzig wichtigen Zeilen sind die erste und die letzte. Mit dem Schlüsselwort „FROM" drücken wir aus, dass das bereits bekannte Image „java:8" verwendet und durch das Kopieren des Fat-Jar-Files erweitert werden soll. Damit das Build des Docker-Image reibungslos funktioniert, speichern wir beide, das Dockerfile und das Jar-File, in einem gemeinsamen und ansonsten leeren Verzeichnis ab. Diese Funktionalität ist durch das Profil „buildDockerWorkDir" bereitgestellt. Im Anschluss verwenden wir „Docker Engine", um ein neues Image zu erstellen *(siehe Listing 9)*.

Um das neue Docker-Image in „Docker Compose" verwenden zu können, passen wir die Konfiguration wie in *Listing 10* an. An die Stelle des Docker-Image „java:8" ist unser eigenes getreten und die Bereitstellung von Artefakten, des Fat-Jar-Files, ist entfallen, da diese bereits im Image enthalten sind. Wer möchte, kann anhand der Skriptdatei „s04.docker-buildPlain-and-run.sh" wieder eine entsprechende Abfolge von Build- und Test-Schritten nachvollziehen.

## Stufe 5: Docker-Image via Maven

In der letzten Stufe wollen wir unser Beispielprojekt noch etwas verschönern, indem wir das Erstellen des Docker-Image in den Maven–basierten, automatischen Build-Zyklus einbinden. Dazu brauchen wir als Erstes ein Maven-Plug-in. Roland Huss hat uns in dankenswerter Weise bereits die meiste Arbeit abgenommen, indem er nicht nur selbst ein nach Ansicht des Autors sehr gutes Maven-Docker-Plug-in geschrieben, sondern auch einen ziemlich objektiven Vergleich mit anderen Plug-ins veröffentlicht hat *[3, 22]*.

Die notwendige Erweiterung der Maven-Konfiguration findet sich im Profil „buildDockerImageWithMaven". Dabei wird im Gegensatz zur Stufe 4 kein separates Dockerfile verwendet, sondern alle Informationen direkt in der Plug-in-Konfiguration definiert. Optional lässt sich durch weitere Einstellungen detailliert steuern, welche Dateien in das Image gepackt werden. Ansonsten besteht der Unterschied zum Dockerfile aus Stufe 4 vor allem in der Anzahl der notwendigen Zeilen und den vielen spitzen Klammern.

Die „Docker Compose"-Konfiguration unterscheidet sich von ihrem Vorgänger nur im Namen des erstellten Docker-Image, für den wir zur Unterscheidung einen etwas anderen Namen wählten. Auch gilt ansonsten das Gleiche wie für Stufe 4: Anhand des Shellskripts „s05.docker-buildAllWithMaven-and-run.sh" können die verschiedenen Schritte zum Bauen des Fat-Jar-Files, das Erstellen des Docker-Image mit Maven sowie das Starten, Testen und Stoppen der „Docker-Composition" nachvollzogen werden.

## Lessons Learned

Aus der beschriebenen „Dockerisierung" einer Java-Web-Anwendung kann aus Sicht des Autors die Schlussfolgerung gezogen werden, dass sich Docker vor allem für folgende Dinge eignet:

- Die Bereitstellung von Tools und Komponenten, wie in unserem Beispiel der MySQL-Datenbank
- Das Verpacken und das Deployment von Anwendungen
- Die relativ einfache Definition und Erstellung von Anwendungsumgebungen

Andere typische Aufgaben eines Entwicklers, speziell auf der Ebene von Unit- oder Module-Tests, bleiben im Wesentlichen der klassischen lokalen Arbeitsweise vorbehalten. Vor allem aufgrund der einfachen Definition von ganzen Anwendungslandschaften könnte der Einsatz von Docker auch außerhalb von Entwicklerarbeitsplätzen, etwa in CI-/CD-Pipelines, durchaus positive Effekte bringen.

## Quellen und weiterführende Links


[1] J. Gray, A Conversation with Werner Vogels: Learning form the Amazon technology platform, 2006: https://queue.acm.org/detail.cfm?id=1142065

[2] Jez Humble, David Farley, Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation, 2010, Addison-Wesley

[3] Git Repository Docker-Maven-Plugin von Ronald Huss: https://github.com/rhuss/docker-maven-plugin

[4] Christoph Arnold, Michel Rode, Jan Sperling, Andreas Steil: KVM Best Practices, 2012, dpunkt.verlag, ISBN 978-3-86491-117-0

[5] Docker Homepage: https://www.docker.com

[6] Docker Documentation: https://docs.docker.com

[7] Docker Engine: https://github.com/docker/docker

[8] Docker Machine: https://github.com/docker/machine

[9] Docker Compose: https://github.com/docker/compose

[10] Docker Toolbox: https://github.com/docker/toolbox

[11] Docker-Repository MySql-Datenbank: https://hub.docker.com/_/mysql

[12] Homepage des Apache Maven Projektes: http://maven.apache.org

[13] Homepage des Projektes „httpie": http://httpie.org

[14] Git-Repository der Sourcen des Beispielprojekte: https://bitbucket.org/mindapproach/demo-helloworld-web

[15] Issue #1085 „Epic: Windows Support" im Docker-Compose-Projekt auf Github: https://github.com/docker/compose/issues/1085

[16] Diskussion zum Thema „How to install docker-compose on Windows" auf Stackoverflow: http://stackoverflow.com/questions/29289785/how-to-install-docker-compose-on-windows

[17] Brian Puglisi, „How to get Docker Toolbox (and Compose) working on Windows 10": http://brianpuglisi.com/how-to-get-docker-compose-working-on-windows-10/

[18] Spring Projekt, Dokumentationen, Tutorials und Guides: https://spring.io/docs

[19] YAML-Format: https://de.wikipedia.org/wiki/YAML

[20] Docker Hub: https://hub.docker.com

[21] Docker-Repository JVM: https://hub.docker.com/_/java

[22] Ronald Huss, Docker Maven Plugin Shootout: https://github.com/rhuss/shootout-docker-maven



*Bernd Fischer*

*bfischer@mindapproach.de*



Bernd Fischer beschäftigt sich seit seinem Studium der Elektrotechnik mit Software-Entwicklung und -Architektur und arbeitet heute als CTO bei der MindApproach GmbH in Dresden. Über Assembler, Fortran, Pascal, C/C++ kam er vor rund fünfzehn Jahren zur Progammiersprache Java und ist ihr seither weitestgehend treu geblieben. Durch seine langjährige Tätigkeit als Entwickler, Architekt und Projektleiter hat er positive wie negative Auswirkungen unterschiedlicher Herangehensweisen in der Software-Entwicklung hautnah kennengelernt und schätzt heute besonders die Vorzügen von DevOps und Continuous Delivery. Neben seiner hauptberuflichen Tätigkeit ist er in der JUG Saxony und seit Neuesten in der Docker-Dresden-Usergroup aktiv.