Adrian Mouat

Docker

Software entwickeln und deployen mit Containern

dpunkt.verlag

Adrian Mouat ist Chief Scientist bei Container Solutions – einem europaweit vertretenen Serviceunternehmen, das sich auf Docker und Mesos spezialisiert hat. Zuvor war er Anwendungsberater bei EPCC, das zur University of Edinburgh gehört.

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.de/plus

Adrian Mouat

Docker

Software entwickeln und deployen mit Containern

dpunkt.verlag

Adrian Mouat

Übersetzung: Thomas Demmig
Überarbeitung und Aktualisierung: Peter Roßbach
Lektorat: René Schönfeldt
Copy-Editing: Annette Schwarz, Ditzingen
Satz: III-satz, www.drei-satz.de
Herstellung: Nadine Thiele
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Buch 978-3-86490-384-7
PDF 978-3-96088-036-3
ePub 978-3-96088-037-0
mobi 978-3-96088-038-7

1. Auflage 2016
Copyright © 2016 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of Using Docker ISBN 9781491915769
© 2016 Adrian Mouat. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.
5 4 3 2 1 0

Für alle, die es versuchen – ob mit oder ohne Erfolg

Geleitwort zur deutschen Übersetzung

Build, Ship and Run,
Any App,
Anywhere

Docker Mantra

Container-Technologie, Clouds, Microservices und die Anforderungen an Zuverlässigkeit und schnelle Reaktion verändern gerade die IT-Welt. Information werden als eigenständiger Wert gehandelt. Diese Entwicklung nimmt seit 20 Jahren durch das Internet ihren Lauf. Erst belächelt, dann immer schneller verwendet, ist es heute essenzieller Bestandteil des Lebens in unserer Gesellschaft. Nur durch die gigantischen Server-Farmen im Hintergrund und die Bereitstellung des Betriebsökosystems, ist das möglich. Immer leistungsfähigere Soft- und Hardware ist im Einsatz.

Ende 2013 entdeckte ich Docker und hatte ein Schlüsselerlebnis: endlich, nach fast 30 Jahren IT, eine simple Lösung, um komplexe Software einfach in kleine Containern zu verpacken, sinnvoll zu kombinieren und dann auch noch zwischen Maschinen austauschbar zu machen. Nach all den vorangegangenen Erfahrungen – manuell, mit Skripten, Packages, Betriebssystem-Images, Konfigurationsmanagement, kryptischer Isolierung von Prozessen – nun eine einfache Lösung! Docker vereint großartig die bestehende Linux-Virtualisierung, Ressourcen-Isolierung, Sicherheits-Features, Netzwerke und Storage-Lösungen in einem einfachen Ansatz.

Die Verbreitung von Docker ist die schnellste Technologie Adaption, die ich je beobachtet habe. Ende 2014 auf der DockerCon in Amsterdam, lernte ich Adrian Mouat und viele andere Docker-Enthusiasten kennen und schätzen. Zu diesem Zeitpunkt war in jedem Gespräch deutlich zu spüren, dass Docker unsere IT-Welt unumkehrbar veränderte und noch mehr zu erwarten war. Seit diesem Zeitpunkt promote ich Container-Technologien des Docker-Ökosystems, gebe Trainings, organisiere Meetings und Konferenzen, schreibe Artikel und lerne jeden Tag etwas Neues über IT-Infrastruktur.

Ein Buch über Docker zu schreiben – eine Technologie, die sich enorm schnell verändert – ist mutig. Adrian ist dies 2015 vorbildlich gelungen. Anschaulich beschreibt er die Grundlagen von Docker, das sich darum entwickelnde Ökosystem sowie aktuelle Trends. Mit einem durchgängigen Beispiel bietet er einen praktischen Einstieg und zeigt, wie Sie erste Schritte mit Docker gehen.

Die Idee einer deutschen Übersetzung betrachtete ich anfangs allerdings skeptisch: Konnte sie überhaupt die zahlreichen Neuerungen des Docker-Ökosystems seit Erscheinen des englischen Originals angemessen darstellen? Zwischen Version 1.8 und der heute aktuellen Version 1.12 von Docker gab es ja einige Erweiterungen und Veränderungen, z.B. was Netzwerke und die Orchestrierung von Containern angeht.

Nachdem ich mich im Auftrag des Verlages dann im Detail mit dem Manuskript beschäftigte, war ich allerdings erst erleichtert und dann begeistert: Adrians Text war so angelegt, dass eigentlich nichts falsch oder komplett umzubauen war. Nur an wenigen Stellen mussten Textstellen leicht umformuliert sowie Hinweise auf Neuerungen hinzugefügt werden, damit das Buch auch weiterhin einen gelungenen praktischen Einstieg in die Technologie von Docker bietet. Die Grundlagen von Docker sind stabile und müssen erlernt werden. Ohne gute Kenntnisse der Docker Basics kann man die großartigen Neuerungen in den Bereichen Netzwerk, Storage, Clustering und Betrieb von Docker nicht wirklich sicher nutzen.

Lieben Dank an den Übersetzer Thomas Demmig sowie René Schönfeldt und sein Verlagsteam, dass sie mir die Chance gegeben haben, Teil dieses Projektes zu sein.

Viel Spaß beim Lesen, Probieren, Übertragen und Einsetzen von Docker!

August 2016
Peter Roßbach

Vorwort

Container sind eine schlanke und portable Möglichkeit, beliebige Anwendungen und ihre Abhängigkeiten zu verpacken und transportabel zu machen.

Wenn man das so aufschreibt, klingt es ziemlich trocken und langweilig. Aber die Verbesserungen im Prozess, die durch Container möglich werden, sind das genaue Gegenteil davon – korrekt eingesetzt können Container wegweisend sein. Die Verlockungen der dadurch möglich werdenden Architekturen und Arbeitsabläufe sind so überzeugend, dass man glauben mag, spätestens in einem Jahr hat sich in jeder größeren IT-Firma der Status von »Docker? Nie gehört!« zu »Klar, schauen wir uns gerade an und setzen wir auch schon probehalber ein« geändert hat.

Der Aufstieg von Docker ist erstaunlich. Ich kann mich an keine Technologie erinnern, die so schnell so tiefgreifende Auswirkungen auf die IT-Branche hatte. Dieses Buch ist mein Versuch, Ihnen zu zeigen, *warum* Container so wichtig sind, *was* es Ihnen bringt, Ihre Umgebung zu containerisieren, und – am wichtigsten – *wie* Sie das erreichen.

Wer dieses Buch lesen sollte

Dieses Buch versucht, Docker ganzheitlich anzugehen, die Gründe für dessen Einsatz zu erklären und zu zeigen, wie man es nutzt und in einen Softwareentwicklungsworkflow integriert. Es behandelt den gesamten Software-Lifecycle – von der Entwicklung über die Produktion bis zur Wartung.

Ich habe versucht, möglichst wenige Annahmen über Sie als Leser zu treffen, und gehe nur davon aus, dass Sie grundlegende Kenntnisse von Linux und der Softwareentwicklung im Allgemeinen haben. Zielgruppe sind vor allem Softwareentwickler, Administratoren und Systemverwalter (insbesondere solche, die einen DevOps-Ansatz verfolgen wollen). Aber auch technisch affine Manager und andere Interessierte sollten ebenfalls Gewinn aus diesem Buch ziehen.

Warum ich dieses Buch geschrieben habe

Ich befand mich in der glücklichen Lage, Docker kennenzulernen und einzusetzen, während es noch ganz am Anfang seines kometenhaften Aufstiegs stand. Als sich die Gelegenheit ergab, dieses Buch zu schreiben, ergriff ich sie mit beiden Händen. Wenn mein Geschreibsel einem von Ihnen dabei hilft, die Containerisierungs-Community zu verstehen und das Beste daraus zu machen, dann habe ich mehr erreicht als in all den Jahren der Softwareentwicklung zuvor.

Ich hoffe wirklich, dass Ihnen das Lesen dieses Buches Spaß macht und es Ihnen dabei hilft, beim Einsatz von Docker in Ihrer Umgebung vorwärtszukommen.

Wie dieses Buch aufgebaut ist

Dieses Buch ist in drei Abschnitte unterteilt:

- Teil I beginnt mit der Erklärung, was Container sind und warum Sie an ihnen interessiert sein sollten. Dann folgt ein Tutorial-Kapitel mit den ersten Schritten beim Einsatz von Docker. Der Abschnitt endet mit einem langen Kapitel mit den zugrunde liegenden Konzepten und Technologien, die hinter Docker stecken, einschließlich eines Überblicks über die diversen Docker-Befehle.
- Teil II erklärt, wie Sie Docker in einem Softwareentwicklungs-Lifecycle einsetzen. Zuerst wird gezeigt, wie Sie eine Entwicklungsumgebung einrichten, bevor Sie eine einfache Webanwendung erstellen, die für den Rest dieses Abschnitts als Beispiel dient. Die Kapitel drehen sich um Entwicklung, Testen und Integration, aber auch darum, wie Sie Container ausrollen und ein Produktivsystem effektiv überwachen und protokollieren.
- In Teil III geht es um die fortgeschrittenen Details und die Tools und Techniken, die Sie brauchen, um Multihost-Cluster mit Docker-Containern sicher und zuverlässig zu betreiben. Haben Sie Docker schon im Einsatz und müssen herausfinden, wie Sie damit wachsen können oder Netzwerk- und Sicherheitsprobleme lösen, ist das Ihr Abschnitt.

Welchen Konventionen dieses Buch folgt

Die folgenden typografischen Konventionen werden in diesem Buch genutzt:

Kursiv
: für neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateierweiterungen

`Nichtproportionalschrift`
: für Programmlistings, aber auch für Codefragmente in Absätzen, wie zum Beispiel Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter

`fette Nichtproportionalschrift`
: für Befehle und anderen Text, der genau so vom Benutzer eingegeben werden sollte

`kursive Nichtproportionalschrift`
: für Text, der vom Benutzer durch eigene Werte ersetzt werden sollte

Dieses Symbol steht für einen Tipp oder Vorschlag.

Dieses Symbol steht für einen allgemeinen Hinweis.

Dieses Symbol steht für eine Warnung oder Vorsichtsmaßnahme.

Codebeispiele

Zusätzliches Material (Codebeispiele, Übungen und so weiter) finden Sie (auf Englisch) zum Herunterladen auf *https://github.com/using-docker*.

Dieses Buch soll Ihnen bei der Arbeit helfen. Die Codebeispiele in diesem Buch können Sie im Allgemeinen in Ihren Programmen und Ihrer Dokumentation nutzen. Sie müssen uns nicht um Erlaubnis fragen, sofern Sie nicht einen signifikanten Anteil des Codes veröffentlichen. So erfordert zum Beispiel das Schreiben eines Programms, das diverse Codeschnipsel aus diesem Buch nutzt, keine Erlaubnis. Das Verkaufen oder Bereitstellen einer CD-ROM mit Beispielen aus diesem Buch benötigt hingegen eine Erlaubnis. Beantworten Sie eine Frage, indem Sie dieses Buch zitieren und Beispielcode daraus nutzen, benötigen Sie keine Erlaubnis. Übernehmen Sie einen deutlichen Teil des Beispielcodes für Ihre Produktdokumentation, müssen Sie dieses Vorhaben von uns genehmigen lassen.

Wir freuen uns über eine Quellenangabe, verlangen sie aber nicht unbedingt. Zu einer Quellenangabe gehören normalerweise Titel, Autor, Verlag und ISBN

(zum Beispiel: »Adrian Mouat: *Docker – Software entwickeln und deployen mit Containern.* dpunkt.verlag 2016, ISBN 978-3-86490-384-7«).

Wenn Sie das Gefühl haben, dass Ihr Einsatz der Codebeispiele über die Grenzen des Erlaubten hinausgeht, können Sie uns über *hallo@dpunkt.de* erreichen.

Danksagungen

Ich bin für all die Hilfe, Ratschläge und Kritik unglaublich dankbar, die ich während des Schreibens dieses Buches erhalten habe. Wenn ich einen Namen in der folgenden Liste vergessen habe, bitte ich um Verzeihung – ich wusste alle Beiträge zu schätzen, auch wenn ich nicht darauf reagiert habe.

Für allgemeines Feedback möchte ich mich bei Ally Hume, Tom Sugden, Lukasz Guminski, Tilaye Alemu, Sebastien Goasguen, Maxim Belooussov, Michael Boelen, Ksenia Burlachenko, Carlos Sanchez, Daniel Bryant, Christoffer Holmstedt, Mike Rathbun, Fabrizio Soppelsa, Yung-Jin Hu, Jouni Miikki und Dale Bewley bedanken.

Mein Dank für technische Diskussionen und Erläuterungen zu bestimmten Technologien geht an Andrew Kennedy, Peter White, Alex Pollitt, Fintan Ryan, Shaun Crampton, Spike Curtis, Alexis Richardson, Ilya Dmitrichenko, Casey Bisson, Thijs Schnitger, Sheng Liang, Timo Derstappen, Puja Abbassi, Alexander Larsson und Kelsey Hightower. Für die Erlaubnis, monsterid.js nutzen zu dürfen, bedanke ich mich bei Kevin Gaudin.

Ein großer Dank für all die Hilfe geht auch an die O'Reilly-Mitarbeiter, insbesondere an meinen Lektor Brian Anderson und an Meghan Blanchette, die das Ganze erst ins Rollen gebracht hat.

Diogo Mónica und Mark Coleman – vielen Dank an euch beide für das Beantworten meines Hilferufs in letzter Sekunde.

Zwei Firmen sollen besonders hervorgehoben werden: Container Solutions und CloudSoft. Jamie Dobson und Container Solutions sorgten dafür, dass ich mit Bloggen und Vorträgen beschäftigt war, und brachten mich mit vielen Leuten zusammen, die Einfluss auf dieses Buch hatten. CloudSoft erlaubten es mir freundlicherweise, in ihren Räumen an diesem Buch zu arbeiten. Zudem organisierten sie das Edinburgh Docker Meetup, das ebenfalls sehr wichtig für mich war.

Dafür, dass Ihr mein Gejammer über dieses Buch und meine Obsession ausgehalten habt, möchte ich mich bei all meinen Freunden und meiner Familie bedanken – Ihr wisst, wen ich meine (und werdet diese Zeilen sehr wahrscheinlich sowieso nicht lesen).

Schließlich möchte ich den DJs von BBC 6 Music danken, die den Soundtrack zu diesem Buch geliefert haben – unter anderem Lauren Laverne, Radcliffe and Maconie, Shaun Keaveny und Iggy Pop.

Inhaltsverzeichnis

Teil I Hintergrund und Grundlagen **1**

1 Was Container sind und warum man sie nutzt **3**

1.1 Container versus VMs 4

1.2 Docker und Container 6

1.3 Eine Geschichte von Docker 9

1.4 Plugins und Plumbing 10

1.5 64-Bit-Linux 11

2 Installation **13**

2.1 Docker auf Linux installieren 13

2.1.1 SELinux im Permissive Mode ausführen 14

2.1.2 Ohne sudo starten 15

2.2 Docker auf Mac OS oder Windows installieren 15

2.3 Ein schneller Check 17

3 Erste Schritte **19**

3.1 Ihr erstes Image ausführen 19

3.2 Die grundlegenden Befehle 20

3.3 Images aus Dockerfiles erstellen 24

3.4 Mit Registries arbeiten 28

3.4.1 Private Repositories 29

3.5 Das offizielle Redis-Image verwenden 30

3.6 Zusammenfassung 34

4 Grundlagen von Docker **35**

4.1 Die Architektur von Docker 35

4.1.1 Zugrunde liegende Technologien 36

4.1.2 Zugehörige Technologien 37

4.1.3 Docker Hosting 40

4.2 Wie Images gebaut werden 40
4.2.1 Der Build Context 41
4.2.2 Imageschichten 42
4.2.3 Caching 44
4.2.4 Basis-Images 45
4.2.5 Anweisungen im Dockerfile 47
4.3 Container mit der Außenwelt verbinden 51
4.4 Container verlinken 52
4.5 Daten mit Volumes und Datencontainern verwalten 53
4.5.1 Daten gemeinsam nutzen 56
4.5.2 Datencontainer 56
4.6 Häufig eingesetzte Docker-Befehle 58
4.6.1 Der Befehl run 59
4.6.2 Container verwalten 62
4.6.3 Docker-Info 64
4.6.4 Container-Info 65
4.6.5 Arbeit mit Images 66
4.6.6 Die Registry verwenden 69
4.7 Zusammenfassung 70

Teil II Der Software-Lebenszyklus mit Docker 71

5 Docker in der Entwicklung einsetzen 73
5.1 Sag »Hallo Welt!« 73
5.2 Mit Compose automatisieren 83
5.2.1 Der Compose-Workflow 84
5.3 Zusammenfassung 86

6 Eine einfache Webanwendung erstellen 87
6.1 Eine einfache Webseite erstellen 88
6.2 Auf vorhandene Images zurückgreifen 90
6.3 Caching ergänzen 95
6.4 Microservices 98
6.5 Zusammenfassung 99

7 Bereitstellen von Images 101
7.1 Namensgebung für Images und Repositories 101
7.2 Der Docker Hub 102

7.3 Automatisierte Builds 104
7.4 Private Distribution 106
7.4.1 Eine eigene Registry betreiben 106
7.4.2 Kommerzielle Registries 113
7.5 Die Imagegröße verringern 114
7.6 Herkunft eines Image 116
7.7 Zusammenfassung 117

8 Continuous Integration und Testen mit Docker 119
8.1 identidock mit Unit-Tests versehen 119
8.2 Einen Jenkins-Container erstellen 124
8.2.1 Builds triggern 132
8.3 Das Image pushen 132
8.3.1 Sinnvolles Taggen. 133
8.3.2 Staging und Produktion 135
8.3.3 Image Sprawl 135
8.3.4 Jenkins Slaves durch Docker betreiben 136
8.4 Backups für Jenkins 136
8.5 Gehostete CI-Lösungen 136
8.6 Testen und Microservices 137
8.6.1 Im Produktivumfeld testen 139
8.7 Zusammenfassung 139

9 Container deployen 141
9.1 Ressourcen mit Docker Machine aufsetzen 142
9.2 Einen Proxy verwenden 145
9.3 Ausführungsoptionen 151
9.3.1 Shell-Skripten 152
9.3.2 Einen Process Manager einsetzen (oder systemd, sie alle zu knechten) 154
9.3.3 Ein Tool zum Configuration Management einsetzen 157
9.4 Host-Konfiguration 161
9.4.1 Ein Betriebssystem wählen 161
9.4.2 Einen Storage-Treiber wählen 162
9.5 Spezialisierte Hosting-Möglichkeiten 165
9.5.1 Triton. 165
9.5.2 Google Container Engine 167
9.5.3 Amazon EC2 Container Service 167
9.5.4 Giant Swarm 170

9.6 Persistente Daten und Produktivcontainer 172
9.7 Gemeinsame Geheimnisse. 172
9.7.1 Geheimnisse im Image ablegen 172
9.7.2 Geheimnisse in Umgebungsvariablen übergeben 173
9.7.3 Geheimnisse in Volumes übergeben. 174
9.7.4 Einen Key/Value-Store einsetzen 174
9.8 Vernetzen 176
9.9 Produktiv-Registry 176
9.10 Continuous Deployment/Delivery. 176
9.11 Zusammenfassung 177

10 Protokollieren und Überwachen 179

10.1 Protokollieren. 180
10.1.1 Standard-Logging von Docker. 180
10.1.2 Logs zusammenfassen 182
10.1.3 Mit ELK loggen. 182
10.1.4 Docker-Logging mit syslog 193
10.1.5 Logs aus Dateien auslesen 199
10.2 Überwachen und Benachrichtigen. 199
10.2.1 Mit den Docker-Tools überwachen. 200
10.2.2 cAdvisor 201
10.2.3 Cluster-Lösungen 203
10.3 Kommerzielle Monitoring- und Logging-Lösungen. 206
10.4 Zusammenfassung 206

Teil III Tools und Techniken 209

11 Vernetzung und Service Discovery 211

11.1 Ambassadors 212
11.2 Service Discovery 216
11.2.1 etcd 217
11.2.2 SkyDNS 221
11.2.3 Consul. 226
11.2.4 Registrieren. 231
11.2.5 Andere Lösungen 232
11.3 Networking-Optionen 234
11.3.1 Bridge 234
11.3.2 Host 235

11.3.3 Container . . . 235
11.3.4 None . . . 236
11.4 Neues Docker-Networking . . . 236
11.4.1 Netzwerktypen und Plugins . . . 238
11.5 Vernetzungslösungen . . . 238
11.5.1 Overlay . . . 239
11.5.2 Weave . . . 241
11.5.3 Flannel . . . 245
11.5.4 Project Calico . . . 251
11.6 Zusammenfassung . . . 256

12 Orchestrieren, Clustering und Verwaltung 259

12.1 Clustering- und Orchestrierungstools . . . 260
12.1.1 Swarm . . . 261
12.1.2 fleet . . . 268
12.1.3 Kubernetes . . . 274
12.1.4 Mesos und Marathon . . . 283
12.2 Container-Management-Plattformen . . . 294
12.2.1 Rancher . . . 295
12.2.2 Clocker . . . 296
12.2.3 Tutum . . . 298
12.3 Zusammenfassung . . . 299

13 Container sichern und beschränken 301

13.1 Worüber Sie sich Gedanken machen sollten . . . 302
13.2 Verteidigung in der Tiefe . . . 304
13.2.1 Least Privilege . . . 304
13.3 identidock absichern . . . 305
13.4 Container nach Host trennen . . . 307
13.5 Updates anwenden . . . 308
13.5.1 Nicht unterstützte Treiber vermeiden . . . 312
13.6 Imageherkunft . . . 312
13.6.1 Docker Digests . . . 313
13.6.2 Docker Content Trust . . . 313
13.6.3 Reproduzierbare und vertrauenswürdige Dockerfiles . . . 318
13.7 Sicherheitstipps . . . 320
13.7.1 Einen Benutzer setzen . . . 321
13.7.2 Netzwerkzugriffe von Containern beschränken . . . 322
13.7.3 setuid/setgid-Binaries entfernen . . . 324

13.7.4 Den Speicher begrenzen 325
13.7.5 Den CPU-Einsatz beschränken 326
13.7.6 Neustarts begrenzen 327
13.7.7 Zugriffe auf die Dateisysteme begrenzen 328
13.7.8 Capabilities einschränken 328
13.7.9 Ressourcenbeschränkungen (ulimits) anwenden 330

13.8 Einen gehärteten Kernel ausführen 332

13.9 Linux Security Modules 333

13.9.1 SELinux 333
13.9.2 AppArmor 336

13.10 Auditing 337

13.11 Reaktion auf Zwischenfälle 338

13.12 Zukünftige Features 339

13.13 Zusammenfassung 339

Index 341

Teil I

Hintergrund und Grundlagen

Im ersten Teil dieses Buches schauen wir uns zunächst an, was Container sind und warum sie sich so großer Beliebtheit erfreuen. Darauf folgt eine Einführung in Docker und die Schlüsselkonzepte, die Sie verstehen müssen, um Container wirklich sinnvoll einzusetzen.

1 Was Container sind und warum man sie nutzt

Container ändern die Art und Weise, wie wir Software entwickeln, verteilen und laufen lassen, grundlegend. Entwickler können Software lokal bauen, weil sie wissen, dass sie auch woanders genauso laufen wird – sei es ein Rack in der IT-Abteilung, der Laptop eines Anwenders oder ein Cluster in der Cloud. Administratoren können sich auf die Netzwerke, Ressourcen und die Uptime konzentrieren und müssen weniger Zeit mit dem Konfigurieren von Umgebungen und dem Kampf mit Systemabhängigkeiten verbringen. Der Einsatz von Containern wächst in der gesamten Branche mit einer erstaunlichen Geschwindigkeit – von den kleinsten Startups bis hin zu großen Unternehmen. Entwickler und Administratoren sollten davon ausgehen, dass sie innerhalb der nächsten Jahre Container regelmäßig einsetzen werden.

Container sind eine Verkapselung einer Anwendung und ihrer Abhängigkeiten. Auf den ersten Blick scheint das nur eine abgespeckte Version einer virtuellen Maschine (VM) zu sein – wie eine VM findet sich in einem Container eine isolierte Instanz eines Betriebssystems (Operating System, OS), mit dem wir Anwendungen laufen lassen können.

Container haben aber eine Reihe von Vorteilen, durch die Anwendungsfälle möglich werden, welche mit klassischen VMs schwierig oder unmöglich zu realisieren wären:

- Container teilen sich Ressourcen mit dem Host-Betriebssystem, wodurch sie um eine wesentliche Größenordnung effizienter sind als virtuelle Maschinen. Container können im Bruchteil einer Sekunde gestartet und gestoppt werden. Anwendungen, die in Containern laufen, verursachen wenig bis gar keinen Overhead im Vergleich zu Anwendungen, die direkt auf dem Host-Betriebssystem gestartet werden.
- Die Portierbarkeit von Containern besitzt das Potenzial, eine ganze Klasse von Bugs auszumerzen, die durch subtile Änderungen in der Laufzeitumgebung entstehen – sie könnte sogar die seit Anbeginn der Softwareentwicklung bestehende Litanei der Entwickler »Aber bei mir auf dem Rechner lief es doch!« beenden.

- Die leichtgewichtige Natur von Containern sorgt dafür, dass Entwickler dutzende davon zur gleichen Zeit laufen lassen können, wodurch das Emulieren eines produktiv nutzbaren, verteilten Systems möglich wird. Administratoren können viel mehr Container auf einer einzelnen Host-Maschine laufen lassen, als dies mit VMs möglich wäre.
- Container haben zudem Vorteile für Endanwender und Entwickler außerhalb des Bereitstellens in der Cloud. Benutzer können komplexe Anwendungen herunterladen und laufen lassen, ohne sich Stunden mit Konfiguration und Installation herumzuschlagen oder über die Änderungen Sorgen machen zu müssen, die am System notwendig wären. Umgekehrt brauchen sich die Entwickler solcher Anwendungen nicht mehr um solche Unterschiede in den Benutzerumgebungen und um eventuelle Abhängigkeiten Gedanken machen.

Wichtiger ist noch, dass sich die grundlegenden Ziele von VMs und Containern unterscheiden – eine VM ist dafür gedacht, eine fremde Umgebung vollständig zu emulieren, während ein Container Anwendungen portabel und in sich abgeschlossen macht.

1.1 Container versus VMs

Obwohl Container und VMs auf den ersten Blick sehr ähnlich wirken, gibt es einige wichtige Unterschiede, die sich am einfachsten über ein Schaubild aufzeigen lassen.

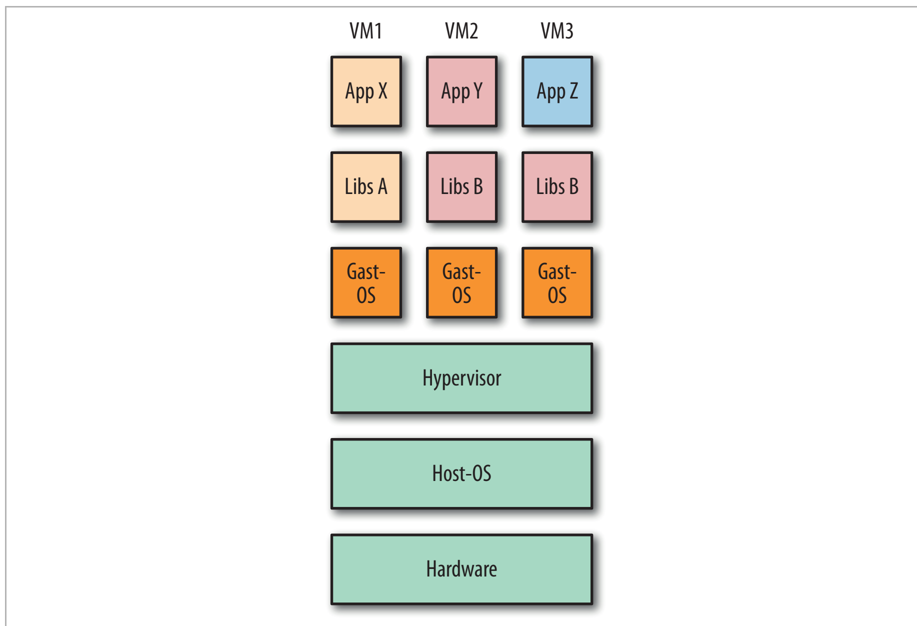

Abb. 1–1 *Drei VMs laufen auf einem Host.*

In Abbildung 1–1 sind drei Anwendungen zu sehen, die auf einem Host in getrennten VMs laufen. Der Hypervisor[1] wird dazu benötigt, VMs zu erstellen und laufen zu lassen, den Zugriff auf das zugrunde liegende Betriebssystem und die Hardware zu steuern und bei Bedarf Systemaufrufe umzusetzen. Jede VM erfordert eine vollständige Kopie des Betriebssystems für sich, dazu die gewünschte Anwendung und alle Bibliotheken, die dafür notwendig sind.

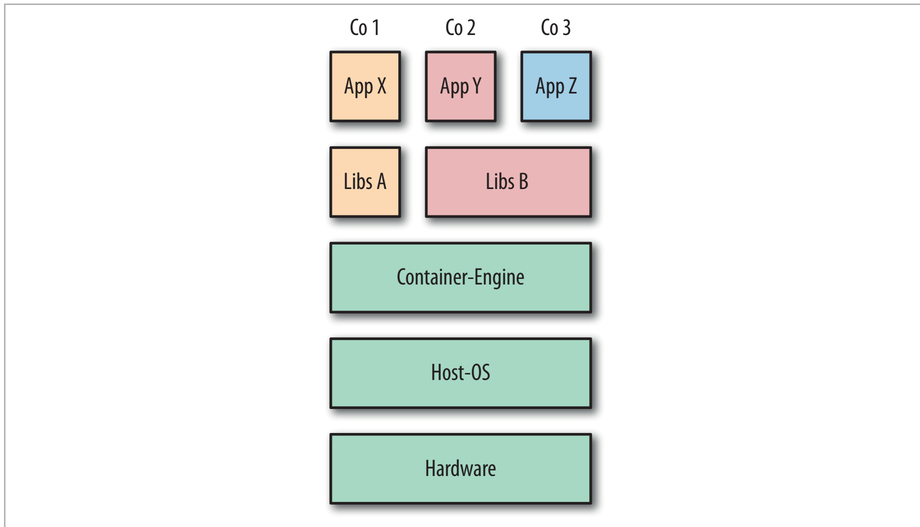

Abb. 1–2 *Drei Container laufen auf einem Host.*

Im Gegensatz dazu sehen Sie in Abbildung 1–2, wie die gleichen drei Anwendungen in einem containerisierten System laufen könnten. Anders als bei VMs wird der Kernel des Host[2] von den laufenden Containern gemeinsam genutzt. Sie sind also immer darauf beschränkt, den gleichen Kernel zu nutzen wie der Host. Die Anwendungen Y und Z verwenden die gleichen Bibliotheken, und sie müssen dafür keine identischen Kopien davon haben, sondern können auf die gleichen Dateien zugreifen. Die Container Engine ist für das Starten und Stoppen von Containern genauso verantwortlich wie der Hypervisor bei einer VM. Aber Prozesse, die innerhalb von Containern laufen, entsprechen nativen Prozessen auf dem Host, und es kommt kein Overhead durch die Ausführung des Hypervisors hinzu.

1. Das Diagramm stellt einen *Typ-2*-Hypervisor dar, wie zum Beispiel Virtualbox oder VMWare Workstation, die auf einem fremden Host-Betriebssystem laufen. Es gibt auch *Typ-1*-Hypervisoren, wie zum Beispiel Xen, bei denen der Hypervisor direkt auf der Hardware läuft.
2. Der Kernel ist die Kernkomponente in einem Betriebssystem. Er ist dafür verantwortlich, Anwendungen die grundlegenden Systemfunktionen rund um Speicher, CPU und den Zugriff auf Geräte zu ermöglichen. Ein vollständiges Betriebssystem besteht aus dem Kernel plus diversen Systemprogrammen, wie zum Beispiel Init-Systemen, Compilern und Window-Managern.

Sowohl VMs wie auch Container können genutzt werden, um Anwendungen von anderen Anwendungen zu isolieren, die auf dem gleichen Host laufen. VMs haben durch den Hypervisor eine weitgehendere Isolation, und es handelt sich bei ihnen um eine vertraute und durch Erfahrung gehärtete Technologie. Container sind verglichen damit recht neu, und viele Firmen scheuen sich, den Isolations-Features von Containern zu trauen, bevor diese ihr Können gezeigt haben. Aus diesem Grund findet man häufig Hybridsysteme mit Containern, die innerhalb von VMs laufen, um die Vorteile beider Technologien vereinen zu können.

1.2 Docker und Container

Container sind ein altes Konzept. Schon seit Jahrzehnten gibt es in UNIX-Systemen den Befehl `chroot`, der eine einfache Form der Dateisystem-Isolation bietet. Seit 1998 gibt es in FreeBSD das Jail-Tool, welches das chroot-Sandboxing auf Prozesse erweitert. Solaris Zones boten 2001 eine recht vollständige Technologie zum Containerisieren, aber diese war auf Solaris OS beschränkt. Ebenfalls 2001 veröffentlichte Parallels Inc. (damals noch SWsoft) die kommerzielle Containertechnologie Virtuozzo für Linux, deren Kern später (im Jahr 2005) als Open Source unter dem Namen OpenVZ bereitgestellt wurde.[3] Dann startete Google die Entwicklung von CGroups für den Linux-Kernel und begann damit, seine Infrastruktur in Container zu verlagern. Das Linux Containers Project (LXC) wurde 2008 initiiert, und in ihm wurden (unter anderem) CGroups, Kernel-Namensräume und die chroot-Technologie zusammengeführt, um eine vollständige Containerisierungslösung zu bieten. 2013 lieferte Docker schließlich die fehlenden Teile für das Containerisierungspuzzle, und die Technologie begann, den Mainstream zu erreichen.

Docker nahm die bestehende Linux-Containertechnologie auf und verpackte und erweiterte sie in vielerlei Hinsicht – vor allem durch portable Images und eine benutzerfreundliche Schnittstelle –, um eine vollständige Lösung für das Erstellen und Verteilen von Containern zu schaffen. Die Docker-Plattform besteht vereinfacht gesagt aus zwei getrennten Komponenten: der Docker Engine, die für das Erstellen und Ausführen von Containern verantwortlich ist, sowie dem Docker Hub, einem Cloud Service, um Container-Images zu verteilen.

Die Docker Engine bietet eine schnelle und bequeme Schnittstelle für das Ausführen von Containern. Zuvor waren für das Laufenlassen eines Containers mit einer Technologie wie LXC umfangreiches Wissen und viel manuelle Arbeit nötig. Auf dem Docker Hub finden sich unglaublich viele frei verfügbare Container-Images zum Herunterladen, so dass Anwender schnell loslegen können und es vermeiden, Arbeit doppelt zu erledigen, die andere schon gemacht hatten.

3. OpenVZ fand nie eine weite Verbreitung, vermutlich weil dafür ein gepatchter Kernel eingesetzt werden musste.

Zu weiteren Tools, die von Docker entwickelt wurden, gehören der Clustering Manager *Swarm*, die GUI *Kitematic* für die Arbeit mit Containern und das Befehlszeilentool *Machine* für die Erzeugung von Docker Hosts.

Durch das Bereitstellen der Docker Engine als Open Source konnte Docker eine große Community aufbauen und auf deren Hilfe bei Bugfixes und Verbesserungen zählen. Das massive Wachstum von Docker hat dazu geführt, dass es ein *De-facto*-Standard wurde, und das wiederum sorgte für Druck aus der Branche, einen unabhängigen, formalen Standard für die Runtime und das Format der Container zu entwickeln. 2015 schließlich wurde dafür die Open Container Initiative[4] gegründet – eine Initiative, die von Docker, Microsoft, CoreOS und vielen weiteren wichtigen Gruppen und Firmen unterstützt wird. Ihre Mission ist das Entwickeln solch eines Standards. Das Containerformat und die Runtime von Docker dienen dabei als Ausgangsbasis Seit der Version 1.11 basiert die Docker Engine auf dem RunC-Kern, der eine Implementierung des Open-Container-Standards ist.

Die wachsende Verbreitung von Containern geht vor allem auf Entwickler zurück, die nun erstmals Tools zur Verfügung hatten, um Container effektiv zu nutzen. Die kurze Startzeit von Docker-Containern hat für die Entwickler einen hohen Stellenwert, weil sie natürlich schnelle und iterative Entwicklungszyklen bevorzugen, in denen sie die Ergebnisse von Codeänderungen möglichst direkt sehen können. Die Portierbarkeits- und Isolationsgarantien von Containern vereinfachen die Zusammenarbeit mit anderen Entwicklern und Administratoren: Entwickler können sicher sein, dass ihr Code in allen Umgebungen laufen wird, während sich die Administratoren auf das Hosten und Orchestrieren von Containern konzentrieren können, statt sich mit dem Code herumschlagen, der darin läuft.

Die Änderungen, die Docker angestoßen hat, sorgen für eine deutlich unterschiedliche Art und Weise, wie wir Software entwickeln. Ohne Docker würden Container noch für eine lange Zeit bei der IT vergessen sein.

Die Fracht-Metapher

Die Docker-Philosophie wird häufig mit einer Frachtcontainer-Metapher erläutert, was auch den Namen »Docker« erklärt. Meist liest sich das wie folgt:

Wenn Güter transportiert werden, sind dafür sehr verschiedene *Hilfsmittel* notwendig, zum Beispiel Lastwagen, Gabelstapler, Kräne, Züge und Schiffe. All diese Hilfsmittel müssen sehr unterschiedliche Güter mit unterschiedlichen Größen und Anforderungen bewegen können (zum Beispiel Kaffeesäcke, Fässer mit gefährlichen Chemikalien, Kisten mit Elektronik, teure Autos oder Rollwagen mit gefrorenem Lammfleisch). Früher war das ein aufwendiger und teurer Prozess, für den viel manuelle Arbeit notwendig war, zum Beispiel durch Dockarbeiter, um die Güter per Hand an jeder Umladestelle aus- und wieder einzuladen (siehe Abbildung 1–3).

4. *https://www.opencontainers.org*

Das Transportgewerbe wurde durch die Einführung der intermodalen Container revolutioniert. Diese Container gibt es in Standardgrößen, und sie sind so entworfen, dass sie mit minimalem manuellen Aufwand zwischen den Verkehrsmitteln umgeladen werden können. Alle entsprechenden Hilfsmittel sind darauf ausgerichtet – Gabelstapler und Kräne, Lastwagen, Züge und Schiffe. Es gibt Kühl- und Isoliercontainer für das Transportieren temperaturempfindlicher Güter, wie zum Beispiel Lebensmittel oder Arzneimittel. Die Vorteile der Standardisierung wurden auch auf zugehörige Systeme ausgeweitet, wie zum Beispiel das Beschriften und Versiegeln der Container. So können sich die Produzenten der Güter um die Inhalte und die Transportunternehmen um das Verschicken und Lagern der Container kümmern.

Abb. 1–3 *Dockarbeiter (»Dockers«) in Bristol im Jahr 1940 (Ministry of Information Photo Division Photographer)*

Das Ziel von Docker ist, die Vorteile der Containerstandardisierung in die IT zu übertragen. In den letzten Jahren ist die Zahl der unterschiedlichen Softwaresysteme massiv gestiegen. Es ist lange her, dass es ausgereicht hat, einen LAMP-Stack[5] auf einem einzelnen Rechner laufen zu lassen. Zu einem typischen modernen System können zum Beispiel JavaScript-Frameworks, NoSQL-Datenbanken, Message Queues, REST-APIs und Backends gehören, die in unterschiedlichsten Programmiersprachen geschrieben wurden. Dieser Stack muss dann teilweise oder vollständig auf einer Vielzahl unterschiedlicher Hardware laufen – vom Laptop des Entwicklers über die Inhouse-Testing-Cluster bis hin zum Cloud Provider für das Produktivsystem. Jede dieser Umgebungen ist anders, auf jeder läuft ein anderes Betriebssystem mit anderen Bibliotheksversionen und anderer Hardware. Kurz gesagt: Wir haben ein ähnliches Szenario wie im Transportgewerbe – fortlaufend ist viel manuelle Arbeit notwendig, um den Code zwischen den Umgebungen zu transportieren.

5. Das stand ursprünglich für Linux, Apache, MySQL und PHP – häufig genutzte Komponenten einer Webanwendung.

So wie die intermodalen Container das Transportieren von Gütern vereinfacht haben, vereinfachen Docker-Container den Transport von Softwareanwendungen. Entwickler können sich darauf konzentrieren, die Anwendung zu bauen und sie in die Test- und Produktivumgebung zu verschieben, ohne sich um Unterschiede in den Umgebungen oder bei den Abhängigkeiten Gedanken machen zu müssen. Die Administratoren können sich auf die wichtigsten Elemente des Ausführens der Container konzentrieren – das Beschaffen von Ressourcen, das Starten und Stoppen von Containern und das Migrieren zwischen den Servern.

1.3 Eine Geschichte von Docker

2008 gründete Solomon Hykes dotCloud, um ein sprachunabhängiges Platform-as-a-Service-(PaaS-)Angebot aufzubauen. Diese Sprachunabhängigkeit war das Alleinstellungsmerkmal für dotCloud – bestehende PaaS waren an bestimmte Sprachen gebunden (so unterstützte Heroku zum Beispiel Ruby und die Google App Engine Java und Python). 2010 beteiligte sich dotCloud am Y Combinator Accelerator Program, wo es mit neuen Partnern zusammengebracht wurde und damit begann, ernsthafte Investitionen einzuwerben. Die entscheidende Wende kam im März 2013, als dotCloud Docker als Open Source bereitstellte – den zentralen Baustein von dotCloud. Während manche Firmen Angst davor gehabt hätten, ihre Geheimnisse preiszugeben, erkannte dotCloud, dass Docker sehr stark davon profitieren würde, wenn es ein durch eine Community unterstütztes Projekt würde.

Frühe Versionen von Docker waren nicht mehr als ein Wrapper für LXC, kombiniert mit einem geschichteten Dateisystem (Union-Dateisystem). Aber die Entwicklung ging ausgesprochen schnell voran: Innerhalb von sechs Monaten gab es mehr als 6700 Stars bei GitHub und 175 Contributors, die keine Mitarbeiter waren. Das führte dazu, dass dotCloud seinen Namen in Docker Inc. änderte und sein Geschäftsmodell anpasste. Docker 1.0 wurde im Juni 2014 veröffentlicht, nur 15 Monate nach der Version 0.1. Docker 1.0 stand für einen großen Schritt in Bezug auf Stabilität und Zuverlässigkeit – und wurde jetzt als »Production Ready« bezeichnet (in vielen Firmen hatte man es sogar schon zuvor tatsächlich produktiveingesetzt, wie zum Beispiel bei Spotify und Baidu). Gleichzeitig eröffnete Docker den Docker Hub – ein öffentliches Repository für Container. Damit war Docker nicht mehr nur eine einfache Container-Engine, sondern begann, sich zu einer vollständigen Plattform zu entwickeln.

Andere Firmen sahen schnell das Potenzial von Docker. Red Hat wurde im September 2013 ein wichtiger Partner und nutzte Docker, um sein OpenShift-Cloud-Angebot zu unterstützen. Google, Amazon und DigitalOcean boten bald Docker-Support für ihre Clouds an, und eine Reihe von Startups spezialisierte sich auf das Docker Hosting, wie zum Beispiel StackDock. Im Oktober 2014 gab

Microsoft bekannt, dass zukünftige Versionen des Windows Server eine Unterstützung für Docker enthalten würden – was eine große Veränderung für eine Firma war, die klassischerweise mit aufgeblasener Firmensoftware in Verbindung gebracht wurde.

Auf der DockerConEU wurde im Dezember 2014 Docker Swarm angekündigt – ein Clustering Manager für Docker – sowie Docker Machine (Letzteres ein CLI-Tool für das Vorbereiten von Docker Hosts). Das war ein deutliches Signal für die Ziele von Docker: eine vollständige und integrierte Lösung für das Ausführen von Containern anzubieten und nicht nur selbst auf die Docker Engine beschränkt zu sein.

Im gleichen Monat kündigte CoreOS an, ihre eigene Container-Runtime rkt zu entwickeln und die appc-Containerspezifikation-S zu erstellen. Im Juni 2015 gaben Solomon Hykes von Docker und Alex Polvi von CoreOS auf der DockerCon in San Francisco die Gründung der Open Container Initiative bekannt (zunächst noch als Open Container Project bezeichnet), um einen gemeinsamen Standard für Containerformate und Runtimes zu entwickeln.

Ebenfalls im Juni 2015 kündigte das FreeBSD-Projekt an, dass Docker nun von FreeBSD unterstützt würde, wobei ZFS und der Linux Compatibility Layer zum Einsatz kommen. Im August 2015 veröffentlichten Docker und Microsoft ein »Tech Preview« der Docker Engine für Windows Server.

Mit dem Release 1.8 wurde von Docker das Content Trust Feature eingeführt, mit dem die Integrität und die Herausgeber von Docker-Images überprüft werden können. Content Trust ist eine entscheidende Komponente für den Aufbau vertrauenswürdiger Arbeitsabläufe, die auf aus Docker Registries stammenden Images basieren.

1.4 Plugins und Plumbing

Docker Inc. war sich schnell bewusst, dass es einen Großteil seines Erfolgs dem Ökosystem um sich herum zu verdanken hatte. Während sich Docker Inc. auf das Bereitstellen einer stabilen, produktiv einsetzbaren Version der Container-Engine konzentrierte, arbeiteten andere Firmen wie CoreOS, WeaveWorks und ClusterHQ an angrenzenden Bereichen, wie zum Beispiel dem Orchestrieren und Vernetzen von Containern. Es wurde aber schnell klar, dass Docker Inc. plante, eine vollständige Plattform zu liefern – mit Networking-, Storage- und Orchestrierungsmöglichkeiten. Um das Ökosystem aber weiter wachsen lassen zu können und um sicherzustellen, dass den Anwendern Lösungen für möglichst viele Szenarien zur Verfügung stehen, verkündete Docker Inc., dass es ein modulares, erweiterbares Framework für Docker schaffen würde, bei dem Komponenten durch entsprechende Elemente von dritter Seite ersetzbar oder erweiterbar wären.

Docker Inc. nannte diese Philosophie »Batteries Included, But Replaceable«: Es würde eine vollständige Lösung angeboten, Teile ließen sich aber austauschen.[6]

Im Laufe der bisherigen Entwicklung hat sich dadurch eine umfangreiche Plugin-Infrastruktur entwickelt. So gab es z.B. schon früh eine Reihe von Plugins, um Container zu vernetzen und die Daten zu verwalten. Stetig wächst das Plugin-Ökosystem weiter und wird mit der Docker-Version 1.12 nun mit dem Befehl `docker plugin` verwaltbar. Der neue Docker Store wird zudem dafür sorgen, dass neben dem Open-Source-Angebot zukünftig auch kommerzielle Erweiterungen bereitstehen.

Docker folgt darüber hinaus dem sogenannten »Infrastructure Plumbing Manifesto«, das sich dazu bekennt, wann immer möglich bestehende Infrastrukturkomponenten wiederzuverwenden und zu verbessern und der Community wiederverwendbare Komponenten bereitzustellen, wenn neue Tools gebraucht werden. Das führte dazu, dass der Low-Level-Code für das Ausführen von Containern in das Projekt *runC* ausgelagert wurde, das vom OCI betreut wird und als Basis für andere Containerplattformen genutzt werden kann.

1.5 64-Bit-Linux

Bis Redaktionsschluss dieser Übersetzung (Docker 1.12) ist die einzige stabile, produktiv nutzbare Plattform für Docker ein 64-Bit-Linux. Auf Ihrem Computer muss also eine 64-Bit-Linux-Distribution laufen, und all Ihre Container werden ebenfalls 64-Bit-Linux nutzen. Sind Sie ein Windows- oder Mac-OS-Anwender, können Sie Docker in einer VM laufen lassen. Übrigens: Eine Version für den Raspberry Pi auf 32-Bit-Basis steht ebenfalls zur Verfügung: *http://blog.hypriot.com/getting-started-with-docker-on-your-arm-device* [7]

Die Unterstützung für andere native Container auf anderen Plattformen, unter anderem BSD, Solaris und Windows Server, steckt in verschiedenen Entwicklungsstadien. Da Docker keine native Virtualisierung bietet, müssen Container immer zum Host-Kernel passen – ein Windows-Server-Container kann nur auf einem Windows-Server-Host laufen und ein 64-Bit-Linux-Container nur auf einem 64-Bit-Linux-Host.

6. Ich persönlich mochte diesen Slogan nie – alle Batterien bieten mehr oder weniger die gleiche Funktionalität und können nur durch Batterien ausgetauscht werden, die die gleiche Größe und Spannung besitzen. Vermutlich kommt er aus der Python-Philosophie »Batteries Included«, wo es darum geht, die umfangreiche Standardbibliothek hervorzuheben, die Python schon mitbringt.
7. Abgerufen am 05.08.2016.

Microservices und Monolithen

Einer der größten Anwendungsfälle und die stärkste treibende Kraft hinter dem Aufstieg von Containern sind *Microservices*.

Microservices sind ein Weg, Softwaresysteme so zu entwickeln und zu kombinieren, dass sie aus kleinen, unabhängigen Komponenten bestehen, die untereinander über das Netz interagieren. Das steht im Gegensatz zum klassischen, *monolithischen* Weg der Softwareentwicklung, bei dem es ein einzelnes, großes Programm gibt, das meist in C++ oder Java geschrieben ist.

Wenn solch ein Monolith dann skaliert werden muss, kann man sich meist nur dazu entscheiden, vertikal zu skalieren *(scale up)*, zusätzliche Anforderungen werden in Form von mehr RAM und mehr Rechenleistung bereitgestellt. Microservices sind dagegen so entworfen, dass sie horizontal skaliert werden können *(scale out)*, indem zusätzliche Anforderungen durch mehrere Rechner verarbeitet werden, auf die die Last verteilt werden kann. In einer Microservices-Architektur ist es möglich, nur die Ressourcen zu skalieren, die für einen bestimmten Service benötigt werden, und sich damit auf die Flaschenhälse des Systems zu beschränken. In einem Monolith wird alles oder gar nichts skaliert, was zu verschwendeten Ressourcen führt.

Bezüglich Komplexität sind Microservices allerdings ein zweischneidiges Schwert. Jeder einzelne Microservice sollte leicht verständlich und einfach zu verändern sein. Aber in einem System, das aus dutzenden oder hunderten solcher Services besteht, steigt die Gesamtkomplexität aufgrund der Interaktion zwischen den einzelnen Komponenten.

Die leichtgewichtige Natur und Geschwindigkeit von Containern bedeutet, dass sie besonders gut dafür geeignet sind, mit ihnen eine Microservices-Architektur zu betreiben.Verglichen mit VMs sind Container deutlich kleiner und schneller ausrollbar, so dass Microservices-Architekturen möglichst wenig Ressourcen nutzen und schnell auf Anforderungsänderungen reagieren können.

Mehr Informationen zu Microservices finden Sie in Microservices – Grundlagen flexibler Softwarearchitekturen von Eberhard Wolff (dpunkt.verlag 2015, ISBN 978-3-86490-313-7, *http://dpunkt.de/buecher/5026/9783864903137-microservices-12181.html*).