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.

Adrian Mouat, Docker, dpunkt.verlag, ISBN 978-3-86490-384-7

- 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.

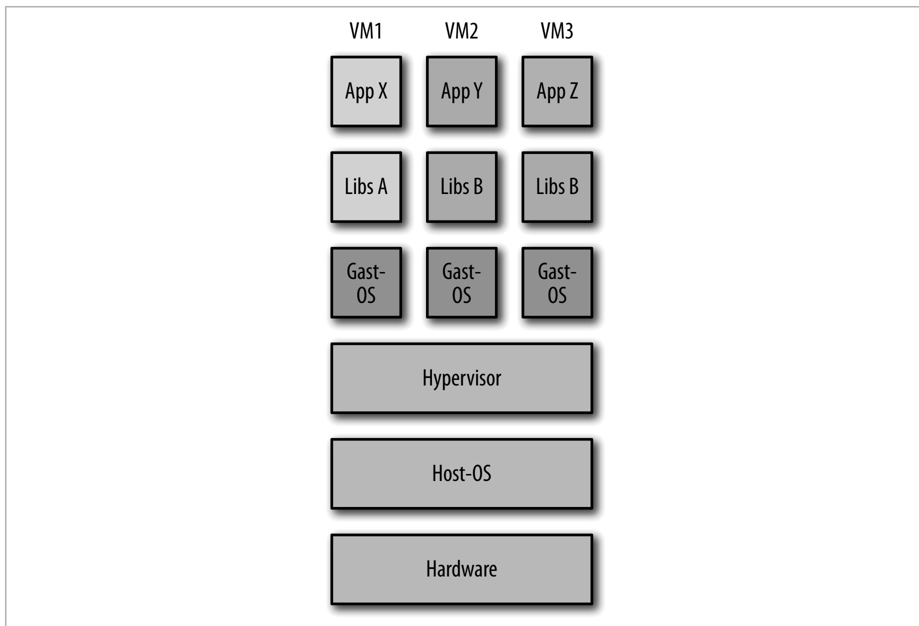

Abb. 1–1 *Drei VMs laufen auf einem Host.*

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.

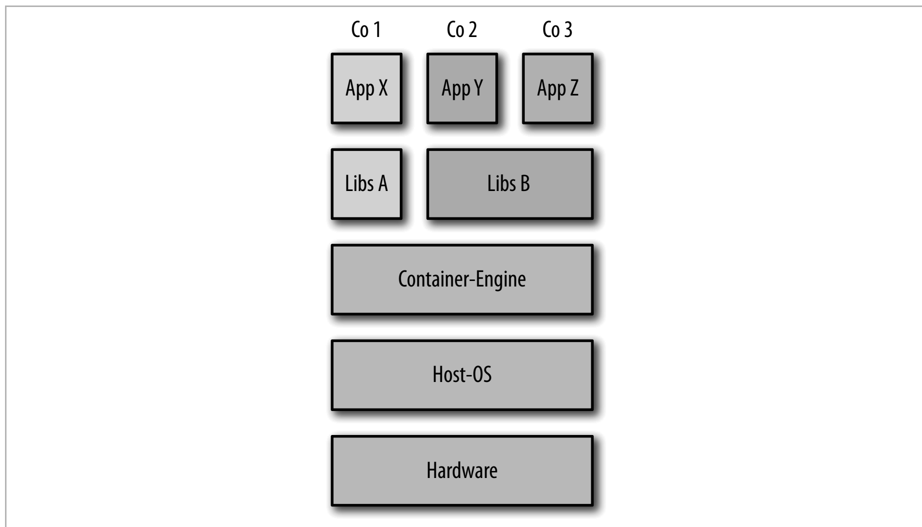

Abb. 1–2 *Drei Container laufen auf einem Host.*

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.

Adrian Mouat, Docker, dpunkt.verlag, ISBN 978-3-86490-384-7

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*

Adrian Mouat, Docker, dpunkt.verlag, ISBN 978-3-86490-384-7

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

Adrian Mouat, Docker, dpunkt.verlag, ISBN 978-3-86490-384-7

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.

Adrian Mouat, Docker, dpunkt.verlag, ISBN 978-3-86490-384-7

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*).