mitp

Karl Matthias

Sean P. Kane

2. Auflage

Docker

Deployment, Testen und Debugging von Containern in Produktivumgebungen

Praxiseinstieg

Inhaltsverzeichnis

Vorwort

Die weitverbreitete technische Evolution, die sich um uns herum vollzieht, konzentriert sich auf ein scheinbar einfaches Tool: den Container. Etwas vom Design her so Kleines und Leichtgewichtiges hat einen gewaltigen Einfluss auf die Softwareentwicklung in allen Branchen. Und das innerhalb kürzester Zeit.

Containerisierung ist allerdings nicht neu und war es auch 2013 nicht, als Docker zum ersten Mal vorgestellt wurde. Allerdings war die Containerisierung vor Docker bei den meisten Softwareentwicklern kaum auf dem Radar. Selbst die Low-Level-Konzepte hinter Containern wurden überwiegend nur von denen verstanden, die ein tiefes Verständnis vom Linux-Kernel hatten oder bei einigen der Tech-Giganten wie Sun oder Google arbeiteten. Windows-Entwickler oder Systemadministratoren wurden generell außen vor gelassen. Heutzutage ist es schwer, ein Gespräch über eine Software zu führen, ohne Docker zu erwähnen. Aber wie sind wir zu diesem Punkt gekommen, und wo geht die Reise noch hin?

Wir nutzen Docker nicht mehr, weil es neu ist, oder nur wegen der reinen Technologie. Es wird hauptsächlich verwendet, weil es den Entwicklungsprozess beschleunigt, die Ausgaben für Infrastruktur und Overhead verringert, das Onboarding neuer Entwickler erleichtert und sogar die Barriere zwischen den Entwicklungs- und Administratorenteams reduziert. Windows-Nutzer können dank der Arbeit von Microsoft, Docker und weiteren zahlreichen Open-Source-Software-(OSS-)Anbietern nun ebenfalls von den Vorteilen von Docker profitieren.

Trotz all seiner Vorteile ist Docker jedoch nicht immer ein einfaches Tool. Man muss es richtig verstanden haben, um Cloud-Native-Anwendungen bauen und administrieren zu können. Cloud-Native-Anwendungen sind hochverfügbare, skalierbare Anwendungen, die auf Managed-Cloud-Infrastrukturen laufen. Um diese Resilienz und Skalierbarkeit zu erreichen, muss man auf Container-Orchestrierungstechnologien wie Kubernetes setzen. Zusätzlich dazu sind Cloud-Native-Anwendungen in der Regel serviceorientiert oder folgen dem Microservice-Architektur-Ansatz.

Ich werde oft gefragt, ob Docker virtuelle Maschinen (VMs) ersetzt, ob Microservice-Architekturen eine Voraussetzung sind oder ob Unternehmen lieber alles über Docker vergessen und stattdessen auf Serverless Computing setzen sollten, was immer populärer wird. Die Antwort lautet immer: Nein! Tools im Cloud-Native-Ökosystem sind ein Zusatzmittel und nicht exklusiv. Docker und VMs ste-

hen nicht in Konkurrenz zueinander, sondern sollten gemeinsam genutzt werden, um den maximalen Nutzen herauszuziehen. Serverless Computing ist dann am sinnvollsten, wenn es mit Containern genutzt wird. Ich würde sogar behaupten, dass Serverless Computing überhaupt nicht so populär wäre, wenn es keine kurzlebigen und leichtgewichtigen Container gäbe. Microservices sind auch keine Voraussetzung für Container. Allerdings ziehen Sie mehr Vorteile aus Containern, wenn Ihre Architektur kleinere Services erlaubt.

Der Einsatz von Docker erlaubt den Entwicklern, ihre Zuständigkeiten in den Bereich der Administration auszudehnen und für das, was sie entwickelt haben, die Verantwortung zu übernehmen. Das kann die Zusammenarbeit über Abteilungsgrenzen hinweg fördern, da Details wie Abhängigkeiten auch in den Verantwortungsbereich des Entwicklungsteams fallen statt ausschließlich in den der Administratoren. Darüber hinaus sind Teams so in der Lage, bessere Artefakte zu generieren, die als Eckpunkte für die Dokumentation genutzt werden können: Ein *Dockerfile* und eine *docker-compose.yml* können zusammen die Rolle eines Leitfadens für die Inbetriebnahme des Projekts übernehmen. Neue Entwickler im Team oder Entwickler in Open-Source-Projekten können in wenigen Schritten produktiv arbeiten, wenn diese Dateien existieren. In der Vergangenheit war es oft eine mehrtägige Aufgabe, eine Entwicklungsumgebung aufzusetzen. Heutzutage können wir das mit einem einfachen, reproduzierbaren Workflow ersetzen: Docker installieren, das Repository klonen und `docker-compose up` laufen lassen.

Ähnlich wie das Cloud-Native-Ökosystem ein Hilfsmittel ist, sind auch viele Docker-eigene Tools praktische Hilfsmittel, und das Beherrschen der Grundlagen wird Ihnen helfen, erfolgreicher zu sein. In diesem Buch sollten Sie Kapitel 4 ganz besondere Aufmerksamkeit widmen. Dieses Kapitel beschäftigt sich mit den Container-Images inklusive der wichtigsten Datei im Projekt: dem Dockerfile. Diese Datei ist, neben den anderen Images, die Sie möglicherweise direkt von einer Image-Registry herunterladen, die Basis für Ihre Anwendung. Jeder weitere Layer, wie Container-Orchestrierung, basiert darauf, dass Sie Ihre Anwendung mittels eines Dockerfiles in einem Image paketiert haben. Sie lernen, dass jeder Anwendungscode, unabhängig von Alter, Framework, Sprache oder Architektur, in ein Container-Image paketiert werden kann.

In diesem Buch teilen Sean und Karl ihr umfangreiches kollektives Wissen, um Ihnen das breite theoretische und taktische Verständnis von Docker und dem dazugehörigen Ökosystem zu vermitteln mit dem Ziel, Ihnen zu einem erfolgreichen Start in die Containerisierung zu verhelfen. Während Docker die Entwicklungsprozesse vereinfachen und optimieren kann, ist es aber auch ein mächtiges Tool mit zahlreichen Komplexitätsschichten. Sean und Karl haben dieses Buch sorgfältig zusammengestellt, um sich auf das Wesentliche zu konzentrieren und Ihnen dabei zu helfen, schnell produktiv zu werden und trotzdem die wichtigen Grundlagen zu verstehen.

Saugen Sie das Wissen und die Erfahrung, die auf diesen Seiten geteilt werden, auf und behalten Sie dieses Buch als Nachschlagewerk. Sie werden es nicht bereuen.

-- Laura Frank Tacho

Docker Captain und Director of Engineering, CloudBees

Twitter: @rhein_wein

Über die Autoren

Sean P. Kane ist zur Zeit Lead Site Reliability Engineer bei New Relic. Er war lange als IT-Techniker tätig und hat in sehr breit gefächerten Industriebranchen (Biotechnologie, Verteidigungswesen, Hightechunternehmen) viele verschiedene Posten bekleidet. Zusätzlich zu seinen Tätigkeiten als Autor, Tutor und Speaker über moderne Systemadministration in produktiven Umgebungen ist er begeisteter Urlauber, Wanderer und Camper. Er lebt mit seiner Frau, seinen Kindern und Hunden im pazifischen Nordwesten der USA.

Karl Matthias ist Director of Cloud and Platform Services bei InVision. Zuvor war er Principal Systems Engineer bei Nitro Software und war in den letzten 20 Jahren als Entwickler, Systemadministrator und Netzwerktechniker für Start-ups und verschiedene Fortune-500-Unternehmen tätig. Nach einigen Jahren in Start-ups in Deutschland und Großbrittannien, mit einem kurzem Aufenthalt in seiner Heimat in Portland, Oregon, wohnt er mit seiner Familie in Dublin in Irland. Wenn er sich nicht gerade mit digitalen Dingen beschäftigt, verbringt er seine Zeit mit seinen zwei Töchtern, mit dem Fotografieren mit Vintage Kameras oder fährt eines seiner Fahrräder.

Einleitung

Dieses Buch richtet sich sowohl an System Engineers als auch an Entwickler. Es weist Ihnen den Weg zu einer funktionierenden Docker-Umgebung und einer vernünftigen Produktivumgebung. Auf dem Weg werden wir erfahren, wie man Docker-Anwendungen baut, testet, deployt und debuggt – und das sowohl in der Entwicklung als auch in der Produktion. Wir behandeln zudem ein paar wichtige Orchestrierungstools aus dem Docker-Ökosystem. Hilfestellungen zu Security und Best Practices für Ihre Containerumgebung runden das Kapitel ab.

Wer sollte dieses Buch lesen?

Das Buch richtet sich an Leser, die nach Lösungen für die verschiedenen mit dem komplexen Workflow bei Entwicklung und Deployment von Anwendungen einhergehenden Problemen suchen. Wenn Sie an Docker, Linux-Containern, DevOps und umfangreichen skalierbaren Softwareinfrastrukturen interessiert sind, ist dieses Buch genau das Richtige für Sie.

Warum überhaupt dieses Buch lesen?

Heutzutage sind jede Menge Foren, Projektbeschreibungen und Artikel zum Thema Docker im Internet verfügbar. Warum also sollten Sie Ihre kostbare Zeit mit dem Lesen dieses Buchs verbringen?

Nun, auch wenn tatsächlich schon viele Informationen bereitstehen, ist Docker doch eine neue Technologie, die sich rasant weiterentwickelt. Während wir die erste Auflage dieses Buchs geschrieben haben, hat Docker, Inc. allein fünf neue Versionen veröffentlicht und sein hauseigenes Ökosystem um eine Reihe bedeutender Tools erweitert. In den drei Jahren zwischen der ersten und zweiten Auflage dieses Buchs hat sich die Docker-Landschaft stark verändert. Docker wurde viel stabiler, und mittlerweile gibt es eine Auswahl an guten Tools für nahezu jeden Aspekt aus dem DevOps-Workflow. Zu verstehen, was mit Docker alles möglich ist und wie es zu Ihrem Workflow passt und darin eingebunden werden kann, ist keine triviale Aufgabe. So haben wir an Aufbau und Betrieb von produktiven Docker-Umgebungen für mehrere Unternehmen über vier Jahre gearbeitet.

Wir haben Docker nur wenige Monate nach seinem Release in einer Produktivumgebung implementiert und möchten in den nachfolgenden Kapiteln einige Erkenntnisse mit Ihnen teilen, die wir in den Jahren 2014 und 2015 im Rahmen der Weiterentwicklung unserer Plattform gewonnen haben. Ziel soll es hierbei sein, Sie von unseren Erfahrungen profitieren zu lassen, sodass Sie den Stolpersteinen, denen wir begegnet sind, soweit möglich aus dem Weg gehen können. Natürlich ist auch die Onlinedokumentation des Docker-Projekts zwar durchaus nützlich, wir möchten Ihnen hier jedoch ein etwas umfassenderes Gesamtbild vermitteln und Ihnen einige der Verfahrensweisen vorstellen, die sich bestens bewährt haben.

Nach der Lektüre dieses Buchs sollten Sie über hinreichende Kenntnisse verfügen, um zu verstehen, was Docker eigentlich leistet, warum es von Bedeutung ist, wie Sie es zum Laufen bekommen, wie Sie Ihre Anwendungen bereitstellen können und was erforderlich ist, um es in einer Produktivumgebung einzusetzen. Lassen Sie sich von diesem Buch auf eine kurze, aufschlussreiche Reise in das Universum einer interessanten Technologie mitnehmen, die einige sehr praktische Anwendungen bietet.

Aufbau des Buchs

Hier ein Überblick über den Inhalt des Buchs:

- Kapitel 1 und Kapitel 2 bieten Ihnen eine Einführung in Docker und erläutern, was genau Docker eigentlich ist und wie Sie es verwenden können.
- Kapitel 3 führt Sie schrittweise durch die Installation von Docker.
- Die Kapitel 4 bis Kapitel 6 sind dem Docker-Client, Images und Containern gewidmet und untersuchen deren Aufgaben und Funktionsweisen.
- Kapitel 7 zeigt auf, wie Sie Images und Container debuggen können.
- Kapitel 8 führt in Docker Compose ein. Sie erfahren, wie signifikante Vereinfachungen in der Softwareentwicklung von komplexen containerbasierten Services mit Docker Compose möglich sind.
- Kapitel 9 behandelt Themen, die wichtig sind, um einen reibungslosen Übergang in die Produktivumgebung zu gewährleisten.
- Kapitel 10 demonstriert das Deployment von Containern in Public und Private Clouds in größerem Maßstab.
- In Kapitel 11 geht es um fortgeschrittene Themen, die einige Erfahrung mit Docker voraussetzen und von Bedeutung sind, wenn Sie anfangen, Docker in Ihrer Produktivumgebung einzusetzen.
- Kapitel 12 untersucht einige der grundlegenden Konzepte, die sich beim Design der nächsten Generation internetweit verfügbarer Produktivsoftware herausgebildet haben.

- Und das Kapitel 13 schnürt die maßgeblichen Inhalte dieses Buchs schließlich zu einem ansehnlichen, mit einer hübschen Schleife dekorierten Paket zusammen: Es enthält eine Zusammenfassung der verfügbaren Tools, die Ihnen dabei helfen sollen, das Deployment und die Skalierung von Softwarediensten zu verbessern.

Natürlich sind wir uns darüber im Klaren, dass kaum jemand technische Fachbücher von vorne bis hinten durchliest und Einleitungen nur allzu leicht übersprungen werden. Wenn Sie es allerdings schon mal bis hierher geschafft haben, finden Sie nachstehend noch einige Hinweise dazu, wie Sie bei der Lektüre des Buchs vorgehen sollten:

- Wenn Linux-Container Neuland für Sie sind, sollten Sie das Buch von Anfang an lesen. Die ersten beiden Kapitel erörtern die Grundlagen von Docker und Linux-Containern und beschreiben, was sie leisten, wie sie funktionieren und warum Sie all dem Beachtung schenken sollten.
- Falls Sie sofort loslegen und Docker auf Ihrem Rechner installieren und ausführen wollen, sollten Sie direkt zu Kapitel 3 und Kapitel 4 springen. Hier erfahren Sie, wie Docker installiert wird, wie Images erstellt oder heruntergeladen werden, wie Sie Container starten können und vieles mehr.
- Sind Sie mit den Grundlagen von Docker vertraut, benötigen aber dennoch Hilfe, um es in der Entwicklung zu nutzen, sollten Sie die Kapitel 5 bis Kapitel 8 lesen. Diese behandeln sehr viele Themen zum täglichen Einsatz von Docker mit Docker Compose, das Ihnen den Alltag erleichtert.
- Wenn Sie Docker bereits zur Entwicklung verwenden, aber Hilfe benötigen, um eine Produktivumgebung einzurichten, sollten Sie die Lektüre ab Kapitel 9 in Betracht ziehen, die sich mit dem Deployment und dem Debugging von Containern sowie weiteren fortgeschrittenen Themen befassen.
- Sie sind Software- oder Plattformarchitekt? Dann dürfte Sie Kapitel 12 interessieren, denn hier werden aktuell gängige Erwägungen zum Design containerisierter Anwendungen und horizontal skalierbarer Services betrachtet.

Konventionen dieses Buchs

In diesem Buch gelten folgende typografische Konventionen:

- Neue Begriffe, Dateinamen und Dateinamenserweiterungen sind *kursiv* gedruckt.
- URLs und E-Mail-Adressen sind im `Hyperlink-Format` dargestellt.
- Für Programm-Listings oder im Fließtext vorkommende Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter wird eine `nicht-proportionale Schrift` verwendet.

- Texte, die vom Benutzer durch eigene Eingaben oder aus dem Kontext ersichtliche Werte ersetzt werden sollen, sind in KAPITÄLCHEN gedruckt.
- Vorschläge, Tipps, Hinweise und Warnungen sind in gesonderten Kästen angegeben.

Danksagungen

Wir möchten den vielen Menschen danken, die jede Auflage dieses Buchs überhaupt erst möglich gemacht haben:

- Nic Benders, Bjorn Freeman-Benson und Dana Lawson (New Relic), die unsere Bemühungen stets unterstützten und uns die nötige Zeit für die erste Auflage verschafften.
- Roland Tritsch und Nitro Software für die Unterstützung von Karl bei der Arbeit an der zweiten Auflage.
- Laurel Ruma (O'Reilly), die uns vorschlug, ein Buch über Docker zu schreiben, und Mike Loukides, der alles Notwendige arrangierte.
- Besonderer Dank gilt unserem Lektor Brian Anderson, der uns klargemacht hat, worauf wir uns einlassen, und uns bei jedem Schritt zur Seite stand.
- Nikki McDonald und Virginia Wilson, die uns durch den Prozess der zweiten Auflage führten.
- Eine neue Leserschaft an eine neue Technologie heranzuführen, benötigt besonderes Talent. Wir sind sehr dankbar, dass Lars Herrmann und Laura Frank Tacho sich die Zeit genommen haben, jeweils ein Vorwort zu schreiben.
- Den Lesern unseres Buchentwurfs, die gewährleisteten, dass wir beim Schreiben nicht vom richtigen Weg abkamen: Ksenia Burlachenko, die eine erste Bewertung und eine vollständige technische Rezension lieferte, sowie Andrew T. Baker, Sébastien Goasguen, Henri Gomez, Chelsey Frank und Rachid Zarouali.
- Besondere Erwähnung verdienen Alice Goldfuss und Tom Offermann, die uns detailliertes und durchweg nützliches Feedback gaben.
- Gillian McGarvey und Melanie Yarbrough für das Redigieren des Manuskripts, damit es so aussieht, als hätten wir in der Schule bei der Rechtschreibung und Zeichensetzung aufgepasst. 517 fehlende Kommata, und die Zählung läuft weiter ...
- Wendy Catalano und Ellen Troutman, die dafür gesorgt hat, dass alle Leser im Stichwortverzeichnis sinnvolle Einträge vorfinden.
- Unseren Kollegen bei New Relic, die uns beim Einsatz von Docker begleiteten und viele der Erfahrungen sammelten, von denen wir hier berichten.

- Grains of Wrath Brewery, World Cup Coffee, McMenamins Ringlers Pub, Old Town Pizza, A Beer at a Time!, Taylor's Three Rock pub und Weitere, die uns freundlicherweise ihre Tische und Strom zur Verfügung stellten, auch wenn unsere Speisen und Getränke schon längst verzehrt waren.
- Unseren Familien für ihre Unterstützung und dafür, dass sie uns die nötige Ruhe gewährten, wenn wir sie brauchten.
- Und schließlich allen, die uns ermutigten, uns Ratschläge gaben oder uns in anderer Weise irgendwie beim Schreiben dieses Buchs unterstützt haben.

Kapitel 1

Einführung

1.1 Die Entstehung von Docker

Es war der 15. März 2013, als Solomon Hykes, der Gründer und Geschäftsführer von dotCloud (jetzt Docker, Inc.), das Projekt Docker erstmals – ohne Vorankündigung und großes Brimborium – in einer blitzartigen Fünf-Minuten-Präsentation (`http://youtu.be/wW9CAH9nSLs`) auf der Python Developers Conference in Santa Clara, Kalifornien, vorstellte. Zu diesem Zeitpunkt hatten lediglich rund 40 Leute außerhalb des Unternehmens dotCloud Gelegenheit gehabt, Docker auszuprobieren.

Schon wenige Wochen nach dieser Präsentation brach ein unerwartet großer Medienrummel über das Projekt herein. Es wurde umgehend auf GitHub (`https://github.com/moby/moby`) als Open Source zur Verfügung gestellt, damit jedermann es herunterladen und eigene Beiträge dazu leisten konnte. Im Laufe der darauffolgenden Monate gewann Docker in der Branche zunehmend an Bekanntheit, und man bescheinigte ihm, die bisherige Art und Weise der Softwareentwicklung, -verteilung und -nutzung revolutionieren zu können. Innerhalb eines Jahres hatte praktisch jeder Branchenkundige zumindest von Docker Notiz genommen, wenngleich vielen Menschen nicht ganz klar war, was es eigentlich genau leistet und warum das so spannend ist.

Docker ist ein Tool, das eine einfache Kapselung des Erstellungsprozesses von Artefakten zur Verteilung beliebiger Anwendungen verspricht. Dabei ist das Deployment für beliebige Umgebungen skalierbar, und der Workflow sowie die Reaktionsfähigkeit der betreffenden agilen Unternehmen werden optimiert.

1.2 Das Docker-Versprechen

Diejenigen, die noch keine Erfahrung mit Docker haben, verstehen Docker vordergründig als Virtualisierungsplattform. Es leistet tatsächlich aber viel mehr. Es umspannt eine Reihe von belebten Branchensegmenten, für die bereits Lösungsansätze in Form individueller Technologien wie KVM, Xen, OpenStack, Mesos, Capistrano, Fabric, Ansible, Chef, Puppet, SaltStack usw. existieren. Wie Sie vielleicht bemerkt haben, ist die Liste der Produkte, mit denen Docker in Konkurrenz tritt, schon ziemlich aufschlussreich. Viele Administratoren würden kaum behaupten, dass Virtualisierungslösungen im Wettbewerb mit Tools für die Konfi-

gurationsverwaltung stünden – dennoch wirkt sich Docker auf beide Technologien disruptiv aus. Dies hängt im Wesentlichen damit zusammen, dass Docker einen großen Einfluss auf die Arbeitsweise hat. Das wiederum wirkt sich stark auf die zuvor genannten traditionell voneinander isolierten Segmente in der DevOps-Pipeline aus.

Zudem werden den vorgenannten Technologien im Allgemeinen produktivitätssteigernde Eigenschaften nachgesagt, und genau deshalb wird ihnen auch so viel Aufmerksamkeit zuteil. Docker befindet sich sozusagen im Zentrum einiger der leistungsfähigsten Technologien des letzten Jahrzehnts und kann zu signifikanten Verbesserungen fast jedes Schritts der Pipeline führen.

Würden Sie Docker allerdings Punkt für Punkt mit den jeweiligen Platzhirschen der verschiedenen Einsatzbereiche vergleichen, stünde es höchstwahrscheinlich bloß wie ein mittelmäßiger Wettbewerber da. In manchen Bereichen kann Docker seine Stärken besser ausspielen als in anderen, insgesamt betrachtet decken die Features, die es mitbringt, aber ein sehr breites Anwendungsspektrum ab. Durch die Zusammenführung der Schlichtheit von Softwareverteilungstools wie Capistrano und Fabric mit der komfortablen Verwaltungshandhabung von Virtualisierungssystemen sowie optionalen Möglichkeiten zur problemlosen Implementierung der Automatisierung von Workflow und Orchestrierung stellt Docker einen sehr leistungsfähigen Satz an Features zur Verfügung.

Viele neue Technologien kommen und gehen, insofern ist eine gewisse Skepsis gegenüber den neuesten Trends durchaus angebracht. Oberflächlich betrachtet, könnte man auch Docker sicherlich einfach als eine weitere neue Technologie abtun, die sich irgendwelcher speziellen Probleme annimmt, mit denen Entwickler und Administratoren konfrontiert sind. Und wenn Sie es lediglich als Pseudo-Virtualisierungs- oder Deployment-Technologie sehen, erscheint es womöglich wirklich nicht besonders reizvoll. Allerdings leistet Docker sehr viel mehr als nur das, was es auf den ersten Blick zu erkennen gibt.

Die Kommunikation und Workflows mehrerer Teams zu koordinieren, ist oftmals selbst in kleineren Unternehmen und Organisationen schwierig und teuer. Dessen ungeachtet kommt dem detaillierten Informationsaustausch zwischen den Teams jedoch immer mehr Bedeutung für ein erfolgreiches Arbeiten zu. Die Entdeckung und Implementierung eines Tools, das diese Kommunikation erleichtert und gleichzeitig dazu beiträgt, stabilere Software zu produzieren, wäre also von enormem Nutzen – und genau deswegen ist Docker einen zweiten Blick wert. Natürlich ist es kein Wundermittel, und seine zielführende Implementierung will wohlüberlegt sein, trotzdem ist Docker ein guter Ansatz, um in der Praxis auftretende organisatorische Probleme zu lösen und einem Unternehmen das zügigere Deployment besserer Software zu ermöglichen. Abgesehen davon kann ein gut geplanter Docker-Workflow auch zufriedenere Teams und spürbare Kosteneinsparungen zur Folge haben.

Wo also treten die größten Schwierigkeiten auf? Software in dem Tempo auszuliefern, das heutzutage gefordert wird, ist nicht leicht – und wenn ein Unternehmen wächst und aus zwei oder drei Entwicklern mehrere Entwicklerteams werden, wird es auch zunehmend schwieriger, die Kommunikation hinsichtlich des Deployments neuer Softwareversionen zu koordinieren. Die Entwickler müssen möglichst weitreichende Kenntnisse von der Umgebung besitzen, in der ihre Software laufen soll, und die Administratoren müssen ihrerseits immer umfassender mit der internen Funktionsweise der ausgelieferten Software vertraut sein. Im Allgemeinen ist es durchaus vernünftig, diese Kenntnisse kontinuierlich weiter zu vertiefen, weil das letztlich zu einem besseren Verständnis der Softwareumgebung insgesamt führt und somit das Design stabilerer Software ermöglicht. Allerdings ist dieses Wissen nur sehr schwer skalierbar, wenn das Wachstum eines Unternehmens zunimmt.

Die spezifischen Details einer Softwareumgebung bedingen oft eine Menge Kommunikation, die für die beteiligten Teams nicht immer unmittelbar von Nutzen ist. Wenn beispielsweise ein Entwicklerteam darauf warten muss, dass ein Administratorenteam irgendeine Library in der Version 1.2.1 deployt, kommt es zunächst nicht weiter – und für das betroffene Unternehmen bedeutet das verlorene Arbeitszeit. Könnten die Entwickler die Version der verwendeten Library hingegen einfach selbst aktualisieren und dann ihren Code schreiben, testen und ausliefern, würde sich die Zeit bis zum Deployment deutlich verkürzen, und es bestünden darüber hinaus geringere Risiken beim Deployment der Änderung. Und wenn umgekehrt die Administratorenteams die Software auf dem Host aktualisieren könnten, ohne sich mit mehreren Entwicklerteams abstimmen zu müssen, ginge es ebenfalls schneller voran. Docker unterstützt die Möglichkeit, die Software so zu isolieren, dass weniger Kommunikation zwischen den beteiligten Teams erforderlich ist.

Docker reduziert aber nicht nur die notwendige Kommunikation, sondern wirkt sich auch hinsichtlich der Softwarearchitektur recht bestimmend aus und begünstigt Anwendungen, die besonders stabil ausgelegt sind. Seine architektonische Philosophie stellt *atomare Container* oder *Wegwerf-Container* in den Mittelpunkt. Beim Deployment wird die gesamte laufende Umgebung der alten Anwendung zusammen mit der Anwendung selbst weggeworfen. Nichts in der Umgebung der alten Anwendung überlebt länger als die Anwendung selbst – eine einfache Idee mit großem Effekt. Auf diese Weise wird es äußerst unwahrscheinlich, dass eine Anwendung versehentlich auf irgendwelche Überbleibsel einer vorhergehenden Version zugreift. Und auch beim Debuggen vorgenommene kurzlebige Änderungen können so keinen Eingang in künftige Versionen finden, indem sie vom lokalen Dateisystem eingelesen werden. Außerdem sind Anwendungen dadurch hochgradig portabel und lassen sich leicht auf einen anderen Server verschieben, denn alle Zustände müssen unveränderlicher Bestandteil des Deployment-Arte-

fakts sein oder an einen externen Speicherort wie eine Datenbank, einen Cache oder einen Dateiserver übergeben werden.

Die Anwendungen sind somit nicht nur besser skalierbar, sondern auch zuverlässiger – und die Instanzen eines Anwendungscontainers können kommen und gehen, ohne dass sich dies großartig auf die Verfügbarkeit des Frontends auswirken würde. Hierbei handelt es sich um erprobte architektonische Entscheidungen, die sich bei Anwendungen bewährt haben, in denen Docker nicht zum Einsatz kommt. Docker hat diese Designentscheidungen übernommen und gewährleistet somit, dass sich Anwendungen, in denen das Projekt genutzt wird, im Bedarfsfall ebenfalls dementsprechend verhalten – was nur vernünftig ist.

1.2.1 Vorteile des Docker-Workflows

Es ist nicht einfach, all die Möglichkeiten, die Docker mitbringt, in sinnvoll zusammenhängende Kategorien einzuordnen. Eine gut vollzogene Implementierung verschafft dem Unternehmen als Ganzes sowie den Teams, den Entwicklern und den Administratoren im Besonderen zahlreiche Vorteile. Sie vereinfacht architektonische Designentscheidungen, weil alle Anwendungen aus der Perspektive des Hostsystems von außen betrachtet im Wesentlichen gleich aussehen. Außerdem fällt es leichter, Toolings zu erstellen, die von mehreren Anwendungen gemeinsam genutzt werden können. Alles auf dieser Welt hat Vor- und Nachteile, im Fall von Docker ist es aber schon erstaunlich, wie sehr die Vorteile überwiegen. Im Folgenden sind einige der Vorteile beschrieben, die Docker Ihnen bietet:

- **Vorhandene Fertigkeiten der Entwickler fließen vollumfänglich in die Erstellung von Softwarepaketen mit ein.**
  In vielen Unternehmen mussten für die Erstellung von Softwarepaketen für die jeweils zu unterstützenden Plattformen eigens Stellen für Release und Build Engineers geschaffen werden, die über die erforderlichen Kenntnisse im Umgang mit den entsprechenden Toolings verfügen. Der Einsatz von Tools wie rpm, mock, dpkg oder pbuilder kann ziemlich kompliziert sein und muss jeweils individuell erlernt werden. Docker schnürt alle von Ihnen benötigten Bestandteile zu einem Paket zusammen, das aus nur einer einzigen Datei besteht.
- **Anwendungssoftware und erforderliche Betriebssystemdateien werden in einem standardisierten Image-Format gebündelt.**
  Früher musste ein Paket nicht nur die eigentliche Anwendung enthalten, sondern darüber hinaus auch viele weitere Dateien, auf die sie angewiesen war, wie z.B. Libraries oder Daemons. Trotzdem konnte man sich nie sicher sein, dass Ausführungs- und Entwicklungsumgebung zu 100 Prozent übereinstimmten. Für nativ kompilierten Code bedeutete dies, dass das Build-System exakt die gleichen Versionen der Shared Libraries haben musste wie die Produktivumgebung. Dies erschwerte die Paketierung und bereitete vielen Unter-

nehmen Probleme, verlässlich funktionierende Pakete zu liefern. Oftmals versuchen Entwickler, die die (auf Red Hat basierende) Linux-Distribution Scientific Linux verwenden, in der Hoffnung, dass beide Distributionen einander ähnlich genug sind, ein unter Red Hat getestetes Paket zum Laufen zu bekommen. Mit Docker wird die Anwendung allerdings bereits inklusive aller für die Ausführung erforderlichen Dateien zur Verfügung gestellt. Dank der mehrschichtigen Images (Overlays), die an dieser Stelle zum Einsatz kommen, handelt es sich hierbei um einen effizienten Vorgang, der gewährleistet, dass Ihre Anwendung in der erwarteten Umgebung läuft.

- **Pakete werden mit exakt gleichen Artefakten zum Testen und Ausliefern für alle Systeme in allen Umgebungen benutzt.**
  Wenn Entwickler über eine Versionsverwaltung Änderungen einchecken, kann ein neues Docker-Image erstellt werden, das den vollständigen Überprüfungsvorgang durchläuft und an die Produktivumgebung ausgeliefert wird, ohne dass dabei eine Neukompilierung oder die Erstellung eines neuen Pakets erforderlich wäre, es sei denn, dies ist speziell gewünscht.
- **Abstraktion von Softwareanwendungen von der Hardware, ohne Ressourcen zu opfern.**
  Wenn ein Abstraktionslayer zwischen der physischen Hardware und der darauf laufenden Softwareanwendung erforderlich ist, werden typischerweise herkömmliche Virtualisierungslösungen wie VMware eingesetzt – zum Preis eines höheren Ressourcenbedarfs: Der Hypervisor, der die VMs verwaltet, sowie alle laufenden VM-Kernel nutzen einen bestimmten Prozentsatz der Hardwareressourcen des Systems, die den Anwendungen dann nicht mehr zur Verfügung stehen. Ein Container hingegen ist einfach nur ein weiterer, direkt mit dem Linux-Kernel kommunizierender Prozess, der daher mehr Ressourcen in Anspruch nehmen kann, bis er an die systemseitig vorgegebene oder ihm zugewiesene Kontingentsgrenze stößt.

Zum Zeitpunkt des ursprünglichen Docker-Releases existierten Linux-Container bereits seit einigen Jahren, und auch viele der anderen Technologien, auf denen Docker beruht, sind nicht völlig neu. Allerdings bildet die einzigartige Mischung der nachhaltigen Designentscheidungen hinsichtlich Architektur und Workflow hier ein großes Ganzes, das erheblich leistungsfähiger ist als die Summe seiner Teile. Docker gewährt dem durchschnittlichen Software-Engineer Zugang zu den seit 2008 verfügbaren Linux-Containern. Es gestattet die relativ einfache Integration von Linux-Containern in den bereits vorhandenen Workflow bzw. die Prozessabläufe eines Unternehmens. Tatsächlich waren offenbar derart viele Menschen von den oben beschriebenen Schwierigkeiten betroffen, dass das Interesse an Docker schneller zunahm, als man vernünftigerweise hätte erwarten dürfen.

Seit seinem Start vor nur wenigen Jahren hat Docker bereits einige Iterationen gesehen, verfügt nun über eine Fülle an Funktionen und läuft weltweit in einer

großen Anzahl von Produktivumgebungen. Es entwickelte sich sehr schnell zu einem der Grundsteine jedes modernen verteilten Systems. Eine Vielzahl von Unternehmen ziehen Docker als Lösung für ernst zu nehmende Probleme in Betracht, mit denen sie sich im Rahmen des Deployments ihrer Anwendungen konfrontiert sehen.

1.3 Was Docker nicht ist

Docker ist für eine breite Palette von Problemstellungen geeignet, für deren Lösung in der Vergangenheit üblicherweise Tools anderer Kategorien herangezogen wurden. Diese Vielseitigkeit bedingt es oftmals aber auch, dass die Funktionalität in bestimmten Bereichen nicht allzu tief greifend ausgeprägt ist. So werden manche Unternehmen feststellen, dass sie komplett auf ihre Tools für die Konfigurationsverwaltung verzichten können, wenn sie Docker einsetzen. Es kann zwar durchaus einige der Aufgaben herkömmlicher Tools übernehmen – seine wahre Stärke zeigt sich jedoch darin, dass es in der Regel mit ihnen kompatibel ist oder sich durch sie ergänzen lässt. Im Folgenden stellen wir einige der Toolkategorien vor, die Docker zwar nicht unmittelbar ersetzt, die sich aber in Kombination verwenden lassen. Auf diese Weise können oft hervorragende Ergebnisse erzielt werden.

- **Enterprise-Virtualisierungsplattform (VMware, KVM usw.)**
  Ein Container ist keine virtuelle Maschine (kurz VM) im herkömmlichen Sinn. Virtuelle Maschinen enthalten ein vollständiges Betriebssystem, das von einem Hypervisor betrieben wird, der innerhalb des Host-Systems ausgeführt wird. Der größte Vorteil gegenüber der Hardwarevirtualisierung besteht darin, dass es problemlos möglich ist, mehrere virtuelle Maschinen mit völlig anderen Betriebssystemen auf einem einzelnen Host auszuführen. Bei einem Container teilen sich der Host und der Container denselben Kernel – das bedeutet, dass der Container weniger Ressourcen belegt als eine VM, aber auf demselben Betriebssystem (z.B. Linux) beruhen muss wie der Host.
- **Cloud-Plattform (OpenStack, CloudStack usw.)**
  Ebenso wie bei Enterprise-Virtualisierungsplattformen haben – oberflächlich gesehen – Container-Workflows und Cloud-Plattformen viele Gemeinsamkeiten. Beide werden typischerweise für die horizontale Skalierung eingesetzt, um wechselndem Leistungsbedarf Rechnung zu tragen. Docker ist allerdings keine Cloud-Plattform, sondern handhabt das Deployment, die Ausführung und die Verwaltung von Containern auf bereits vorhandenen Docker-Hosts. Es ist nicht möglich, neue Hosts (Instanzen), neue Objektspeicher, neuen Speicherplatz oder eine der anderen typischerweise auf einer Cloud-Plattform verfügbaren Ressourcen zu erstellen. Wenn Sie Docker-Tooling ausweiten, werden Sie mehr und mehr von den Vorteilen profitieren, die traditionell mit der Cloud assoziiert werden.

- **Konfigurationsverwaltung (Puppet, Chef usw.)**
  Docker kann zwar die Möglichkeiten zur Administration von Anwendungen und deren Abhängigkeiten erheblich verbessern, ist aber kein direkter Ersatz für eine herkömmliche Konfigurationsverwaltung. Dockerfiles legen fest, wie ein fertiger Container zur Build-Zeit aussehen soll, haben aber keinen Einfluss auf den aktuellen Zustand und können auch nicht verwendet werden, um das Docker-Host-System zu verwalten. Nichtsdestotrotz kann die Nutzung von Docker dazu führen, dass die Komplexität von Konfigurationsverwaltungscode deutlich reduziert wird. Allein dadurch, dass mehr und mehr Server einfach zu Docker-Hosts werden, wird der Code der Konfigurationsverwaltung viel kleiner, und Docker kann anschließend dafür verwendet werden, die komplexen Anforderungen einer Anwendung innerhalb von Docker-Images auszurollen.
- **Deployment-Framework (Capistrano, Fabric usw.)**
  Docker erleichtert viele Aspekte des Deployments, indem es eigenständige Container-Images erstellt, die alle Abhängigkeiten einer Anwendung kapseln und ohne weitere Anpassungen für alle Umgebungen geeignet sind. Es kann jedoch nicht verwendet werden, um die komplexen Deployments selbst zu automatisieren. Für die Aneinanderreihung umfangreicherer automatisierter Workflows werden im Allgemeinen zusätzliche Tools benötigt. Diese Methode verlangt, dass alle deployten Container auf allen Hosts konsistent ausgerollt werden. Ein einziger Deployment-Workflow würde für die meisten – wenn nicht sogar alle – Docker-basierten Anwendungen ausreichen.
- **Workload-Manager (Mesos, Kubernetes, Swarm usw.)**
  Ein Orchestrierungslayer (inklusive des eingebauten Swarm-Modus) muss genutzt werden, wenn man die Workload auf einen Pool von Docker-Hosts intelligent verteilen möchte. Er trackt den aktuellen Status aller Hosts und der genutzten Ressourcen und unterhält ein Inventar von allen laufenden Containern.
- **Entwicklungsumgebung (Vagrant usw.)**
  Vagrant ist ein Developer-Tool zur Verwaltung virtueller Maschinen, das häufig zur Simulation einer Serverumgebung genutzt wird, die der echten Produktivumgebung, in der eine Anwendung laufen soll, sehr ähnlich ist. Unter anderem vereinfacht Vagrant die Ausführung von Linux-Software auf Rechnern mit Windows oder macOS. Docker Machine ist für viele Standard-Docker-Workflows ausreichend. Es bringt allerdings nicht die breite Palette an Funktionen mit, die in Vagrant zu finden sind, da es sehr stark auf den speziellen Docker-Workflow ausgerichtet ist. Wie auch bei vielen der vorherigen Beispiele ist es allerdings nicht notwendig, eine breite Palette von Produktivumgebungen in der Entwicklungsumgebung zu simulieren, wenn Sie Docker einsetzen. Dies hängt damit zusammen, dass die meisten Produktivumgebungen einfache Docker-Server sein werden, die auch leicht lokal gebaut werden können.

1.4 Wichtige Begrifflichkeiten

Hier werden einige Begrifflichkeiten kurz genannt und erläutert, die im weiteren Verlauf des Buchs verwendet werden:

- **Docker-Client**
  Das ist der `docker`-Befehl, der im Docker-Workflow am meisten genutzt wird, um mit Docker-Servern zu arbeiten.
- **Docker-Server**
  Das ist der `dockerd`-Befehl, der genutzt wird, um den Docker-Server-Prozess zu starten, der über einen Client Container baut und startet.
- **Docker-Image**
  Docker-Images bestehen aus einem oder mehreren Dateisystemlayern, angereichert mit einigen wichtigen Metadaten, die alle Dateien repräsentieren, die eine dockerisierte Anwendung benötigt. Ein einzelnes Docker-Image kann auf zahlreiche Hosts kopiert werden. Ein Image hat üblicherweise sowohl einen Namen als auch ein Tag. Das Tag wird generell zur Identifizierung einer spezifizierten Version eines Images nutzt.
- **Docker-Container**
  Ein Docker-Container ist ein Linux-Container, der von einem Docker-Image instanziiert wird. Ein spezifischer Container existiert genau ein einziges Mal. Allerdings kann es mehrere Container von ein und demselben Image geben.
- **Atomic Host**
  Ein Atomic Host ist ein kleines Betriebssystemabbild wie Fedora CoreOS (`https://getfedora.org/en/coreos/`), das das Hosting von Containern erlaubt sowie Image-basierte (atomic) Betriebssystem-Upgrades ermöglicht.

1.5 Zusammenfassung

Sofern Sie nicht den entsprechenden beruflichen Hintergrund besitzen, ist es nicht ganz einfach, Docker zu verstehen. Im folgenden Kapitel geben wir Ihnen einen umfangreichen Überblick darüber, was Docker eigentlich leistet, wie es eingesetzt werden sollte und welche Vorteile Sie erzielen können, wenn Sie all das bei der Implementierung berücksichtigen.

Stichwortverzeichnis