

# Kurzfassung

Softwareentwicklung in Zeiten von Node.js, ASP.NET Core und Microservices impliziert die Verwendung einer Menge an Werkzeugen. Die Hauptaufgabe des Softwareentwicklers ist es allerdings, Software zu entwickeln und nicht den Großteil seiner Zeit mit dem Erlernen und der Konfiguration dieser Werkzeuge zu verbringen.

Mit Docker wurde eine Plattform zur Vereinfachung der Softwareentwicklung auf Basis von Containervirtualisierung geschaffen, deren Ziele eine möglichst einfache Verwendung und Plattformunabhängigkeit sind. Anhand der Geschichte der Containervirtualisierung wird in dieser Arbeit ein Vergleich zu herkömmlichen Virtualisierungsmöglichkeiten gezogen sowie die Funktionsweise und notwendigen Systemvoraussetzungen für die Verwendung von Containern dargestellt. Nach einer Übersicht über bestehende Werkzeuge des Konfigurationsmanagements wird für Softwareentwickler eine Einführung in die Verwendung von Docker zu Entwicklungszwecken gegeben, wobei besonders auf Best-Practices und Integrationsmöglichkeiten in den Entwicklungsprozess Wert gelegt wird.

Am Beispiel verschiedener Anwendungsfälle wird gezeigt, wie der Entwicklungsprozess mithilfe von Docker stabiler und einheitlicher gestaltet werden kann, um das erklärte Ziel, einen produktiveren und angenehmeren Workflow für Softwareentwickler, zu erreichen.

Aktuelle Entwicklungen wie Eclipse Che und Windows Container zeigen, dass Containervirtualisierung im Allgemeinen und Docker im Speziellen eine große Zukunft bevorsteht.

# Abstract

When using technologies like Node.js, ASP.NET Core and Microservices developers have to cope with an immense variety of tools. But investing hours of work studying, tweaking and configuring the tools should not be the primary task of a software developer. It must be the aim to use less time adapting the tools and more time actually developing the product itself.

With the release of Docker in 2014, using containers in software development became a lot easier. Docker's main goals are platform independence and the highest possible ease of use. The thesis targets the difference between container virtualization and conventional virtualization technologies based on their histories as well as explaining the system requirements for Docker and how it works. Commonly used configuration management tools are described, before a short introduction into the usage of Docker for developers is given, highlighting possibilities to integrate Docker into the software development process and focusing on industry-proven best practices.

Various use cases illustrate how a better development experience can be achieved by evolving a consistent and more stable development process.

Latest developments like Eclipse Che and Windows Containers show that container virtualization and Docker in particular are going to affect the way we write software.

# Inhaltsverzeichnis

<b>Kurzfassung</b>	<b>6</b>
<b>Abstract</b>	<b>7</b>
<b>1 Einleitung</b>	<b>10</b>
1.1 Motivation . . . . .	10
1.2 Ziel . . . . .	10
<b>2 Virtualisierung</b>	<b>11</b>
2.1 Beweggründe und Geschichte . . . . .	11
2.2 Vollvirtualisierung . . . . .	13
2.3 Containervirtualisierung . . . . .	15
2.3.1 Implementierungen von Containervirtualisierungslösungen . . . . .	17
2.3.2 Open Container Initiative . . . . .	17
2.3.3 Orchestrierungssysteme . . . . .	18
<b>3 Konfigurationsmanagement</b>	<b>20</b>
3.1 Infrastructure as Code . . . . .	20
3.2 Werkzeuge . . . . .	21
3.2.1 Vagrant . . . . .	21
3.2.2 Chef . . . . .	23
3.2.3 Puppet . . . . .	23
3.2.4 SaltStack . . . . .	24
3.2.5 Ansible . . . . .	24
3.2.6 Packer . . . . .	25
3.3 Kombinationsmöglichkeiten der Werkzeuge . . . . .	25
<b>4 Docker</b>	<b>28</b>
4.1 Geschichte . . . . .	28
4.2 Funktionsweise . . . . .	29
4.3 Plattformen . . . . .	30
4.3.1 Docker for Windows . . . . .	31
4.3.2 Docker for Mac . . . . .	31
4.3.3 Docker Toolbox . . . . .	31
4.4 Produkte . . . . .	32
4.5 Dateisystem . . . . .	33

4.6	Dockerfiles . . . . .	33
4.6.1	Best Practices für das Erstellen von Dockerfiles . . . . .	35
4.6.2	Reduktion der Imagegröße . . . . .	36
4.6.3	Festlegen des Start-Kommandos für den Container . . . . .	38
4.7	Häufig benötigte Docker-Kommandos . . . . .	39
4.7.1	Management-Kommandos . . . . .	39
4.7.2	Images erstellen . . . . .	39
4.7.3	Container starten . . . . .	39
4.7.4	Docker aufräumen . . . . .	40
<b>5</b>	<b>Docker-Anwendungsszenarien für den Softwareentwickler</b>	<b>41</b>
5.1	Softwareevaluierung . . . . .	42
5.2	Plattformunabhängige CLI-Anwendungen . . . . .	44
5.3	Containerbasierte Integrationstests . . . . .	47
5.4	Plattformübergreifende Übersetzung . . . . .	48
5.5	IDE in a Container . . . . .	50
<b>6</b>	<b>Resümee</b>	<b>53</b>
6.1	Zusammenfassung . . . . .	53
6.2	Erkenntnisse . . . . .	53
6.3	Ausblick . . . . .	54
	<b>Literatur</b>	<b>57</b>

# 1 Einleitung

## 1.1 Motivation

Die tägliche Arbeit eines Softwareentwicklers beinhaltet die Verwendung von zahlreichen Werkzeugen. Besonders bei der Entwicklung auf unterschiedlichen Plattformen führt die Kombination der zahlreichen Build-Tools, Compiler, Transpiler, Task-Runner, Paketmanager und Datenbanken schnell zu einer umfangreichen Kombination an Kommandos. Projekte entwickeln sich weiter und neue Technologien erscheinen, schnell wird aus einem `npm run dev` ein `npm start` oder `mvn build` bricht aufgrund einer falschen Java-Version ab. Die große Vielfalt der Werkzeuge, besonders im Web-Bereich, verlangt vom Entwickler, dass er sich für jedes Projekt das Wissen für ein neues Werkzeugset aneignet und dieses auch zu verwenden weiß.

Seit 2014 verspricht Docker mit dem Werbespruch „*Build, Ship, Run!*“ eine Verbesserung des Softwareentwicklungsprozesses mithilfe von Containervirtualisierung. Welche Möglichkeiten der Automatisierung für den Softwareentwickler gegeben sind, inwieweit ein plattformunabhängiger Entwicklungsprozess möglich ist und ob sich durch den Einsatz von Docker die Lernkurve eines Entwicklers beim Einstieg in ein Softwareprojekt verringern lässt, sind motivierende Gründe für diese Bachelorarbeit.

## 1.2 Ziel

Durch exemplarisches Lösen typischer Probleme, mit denen Softwareentwickler häufig konfrontiert sind, soll dem Leser die Verwendung und Flexibilität von Docker gezeigt werden. Weiters sollen Softwareentwickler automatisierbare Probleme erkennen und verstehen, ob und wie Docker in diesen Fällen zu einer zeit- und ressourcensparenden Lösung beitragen könnte.

Zur Verdeutlichung dieser entwicklergesteuerten Sicht auf Docker werden in der Arbeit verschiedene Virtualisierungstechniken verglichen und in der Funktionsweise und dem Einsatzzweck von Docker abgegrenzt. Die grundlegenden, für den Entwickler notwendigen Konzepte von Docker werden ebenso geschildert wie eine Übersicht und Erklärung der in Verbindung mit Docker am häufigsten eingesetzten Werkzeuge.

## 2 Virtualisierung

Die Virtualisierung von IT-Systemen entwickelte sich in den letzten Jahren zur Standardlösung beim Betrieb von Server-Software [BDW16]. Zur Virtualisierung führte die Tatsache, dass leistungsfähigere Systeme, schnellere Technologiewechsel und sich ständig ändernde Voraussetzungen durch die Verwendung von rein hardwarebasierten Systemen zu teuer wurde [VMw11].

2016 stagnierte jedoch der Erwerb von neuen Virtualisierungslizenzen zum ersten Mal [BDW16]. Warum Containertechnologien der nächste Schritt sind, welche Rolle Docker darin einnimmt und wohin diese Technologie noch führen kann, wird in diesem Kapitel erläutert.

### 2.1 Beweggründe und Geschichte

Im Folgenden wird die Geschichte der Virtualisierung auf Basis von [BKL09] kurz dargestellt.

In den 1960er-Jahren begann IBM mit dem Einsatz von Großrechnern, sogenannten Mainframes [Cre81]. Da die Kosten für derartige Computersysteme den Einsatz eines Rechners für ein einzelnes System nicht rechtfertigen konnten, wurden auf einem Mainframe mehrere Systeme parallel virtuell betrieben. Dadurch konnten die Rechenleistung effizienter ausgenutzt sowie Versionsinkompatibilitäten durch den parallelen Betrieb alter und neuer Mainframes vermieden werden. Als im Laufe der 80er-Jahre die x86-Architektur ihren Aufschwung erlebte, stand die Virtualisierung still. Der günstige Preis und die fehlende Virtualisierungsunterstützung machten es unnötig und unmöglich, Endgeräte zu virtualisieren.

Erst mit der steigenden Rechenleistung und dem Erscheinen von Mehrkernprozessoren hat die IT-Virtualisierung wieder an Traktion gewonnen. In den letzten Jahren werden besonders durch Cloud-Angebote viele Teile der IT nur mehr virtuell betrieben. Von der kompletten Netzwerkinfrastruktur bis hin zum Speicher wird in modernen Rechenzentren (Amazon Web Services<sup>1</sup>, Microsoft Azure<sup>2</sup>, ...) alles virtualisiert. Die Analyse-Firma Gartner Inc. spricht von einem Virtualisierungsgrad von 75% im Serverbereich [Gar16]. Aspekte wie die einfache Provisionierung von virtualisierten Systemen führen dazu, dass beispielsweise eine horizontale Skalierung ohne großen Mehraufwand betrieben werden kann.

Auch das Testen von Infrastrukturen wurde durch Virtualisierung ermöglicht. Es ist nicht mehr notwendig, ein gesamtes IT-System temporär lediglich zu Testzwecken zur

---

<sup>1</sup><https://aws.amazon.com/>

<sup>2</sup><https://azure.microsoft.com/>

Verfügung zu stellen oder sogar ein solches eigens anzuschaffen. Stattdessen können die benötigten Ressourcen in der Cloud für den benötigten Zeitraum angemietet werden.

Zahlreiche Vorteile der Virtualisierung führten im Gegensatz zu den wenigen Nachteilen zur großen Verbreitung.

### **Vorteile der Virtualisierung**

Die nachfolgend besprochenen Vorteile der Virtualisierung werden in [BKL09] angeführt. Der kostensparendste Aspekt der Virtualisierung ist die Konsolidierung von Servern.

Server, die nicht unter voller Last laufen, benötigen trotzdem Strom, Kühlung, Wartung und Infrastruktur. Durch Konsolidierung können ungenutzte Ressourcen genutzt werden, die zuvor lediglich Kosten verursachten. Virtualisierte Server sind außerdem erheblich einfacher zu provisionieren. Wird ein neuer Server benötigt, können dazu bestehende, freie Ressourcen genutzt werden, anstatt dass weitere Hardware angeschafft werden muss.

Wartungsaufgaben werden stark vereinfacht, da durch diverse Managementwerkzeuge viele Aufgaben automatisiert sowie ohne Änderung an den physischen Systemen durchgeführt werden können. Dazu zählt auch die Live-Migration von virtuellen Maschinen. Diese können dadurch für den Endanwender unterbrechungsfrei und daher unbemerkt zwischen zwei Rechnern verschoben werden. So wird Wartung an der Hardware ermöglicht, ohne dass sich ein Ausfall für die Benutzer ergibt. Voraussetzung dafür ist allerdings, dass Ressourcen im Rechenzentrum frei sind, auf die Systeme (temporär) migriert werden können.

Durch Virtualisierung wird auch die Dimensionierung der Ressourcen erheblich vereinfacht, da initial lediglich die ungefähr benötigte Rechenleistung zu schätzen ist und diese sich im Betrieb umverteilen lässt. So können Systeme, die ihre Leistungsgrenzen erreicht haben oder bei weitem nicht nützen, im laufenden Betrieb neu dimensioniert werden. Dazu kommt die vorhin beschriebene Live-Migration zum Einsatz. Dies führt zu einer gleichmäßigeren Auslastung der Rechenzeit der Hostsysteme, ohne die virtuellen Maschinen dabei zu beeinflussen. Laut [Han08] können die Rechenzentrumskosten um 50% reduziert werden, wohingegen die Anschaffungskosten neuer Hardware um bis zu 70% verringert werden können. [DSN17] beschreibt Möglichkeiten zur automatischen Skalierung von Ressourcen in der Microsoft-Azure-Cloud. Dabei spielt Virtualisierung eine essenzielle Rolle, da alle bei Azure gemieteten Ressourcen virtualisiert laufen. So kann in diesem Fall eine für den Systemadministrator angenehme Version der Dimensionierung erreicht werden, da die Azure-Cloud automatisch die Leistung der Ressourcen an die aktuelle Auslastung anpasst.

Nicht nur durch Features wie die Live-Migration wird die Flexibilität in virtualisierten Rechenzentren erhöht. Virtuelle Maschinen bestehen aus wenigen Dateien, von denen sehr einfach Backups erstellt werden können. Zusätzlich gibt es die Funktion von Snapshots, dabei wird ein Abbild einer virtuellen Maschine erstellt, das jederzeit gestartet werden kann. Dies ist besonders hilfreich, wenn es Snapshots von fertig konfigurierten, jedoch unbenutzten Systemen gibt, da damit fehlerhafte Maschinen sehr schnell wieder hergestellt werden können.

Durch die starke Isolierung der virtuellen Maschinen kann aufgrund der Virtualisierung

hohe Servicequalität erreicht werden, da der Ausfall einer Maschine lediglich diese betrifft, aber das Hostsystem intakt bleibt. In einem Hochverfügbarkeitsszenario besteht die Möglichkeit, dass ein überwachendes System (Supervisor) den Ausfall erkennt und das betroffene System auf einem anderen Knoten neu startet. Des Weiteren wird durch diese Isolierung die Systemsicherheit erhöht, da sich die virtuellen Maschinen gegenseitig nicht beeinflussen.

## Nachteile der Virtualisierung

Folgende Nachteile der Virtualisierung werden in [BKL09] angeführt.

Da bei virtualisierten Systemen die Instruktionen vom virtuellen auf das Hostsystem übersetzt werden müssen, ergibt sich ein Leistungsverlust. Dieser wird allerdings mit leistungsfähigeren Mehrkernprozessoren und durch Prozessorvirtualisierung wie Intel VT<sup>3</sup> verschwindend gering. Die Prozessoren bieten native Virtualisierungstechniken an, bei denen die Befehle nicht mehr übersetzt werden müssen, wodurch die Verringerung der Leistung auf unter 10% sinkt [Har05].

Ein wesentlich größeres Problem in der Virtualisierung stellen besondere Hardwareanforderungen dar. Authentifizierung über Hardware-Dongles oder die Ansteuerung von Legacy-Hardware ist in virtualisierten Umgebungen durch den fehlenden Direktzugriff auf die Hardware sehr schwierig bis nicht möglich.

Besondere Vorsicht verlangt der Aufbau der Infrastruktur von Rechenzentren, da bei einem Hostausfall in virtualisierten Umgebungen zahlreiche Systeme betroffen sind. Dieses Problem lässt sich durch redundante Installationen und gut durchdachte Ausfallkonzepte lösen, wobei hier die Vermeidung eines *Single Point of Failures* (SPOF) höchste Priorität genießen muss. Die Konzeption, Installation und der Betrieb eines solchen Systems erfordern eine hochwertige Infrastruktur und zusätzliches Detailwissen.

## 2.2 Vollvirtualisierung

[BKL09] stellt dar, dass die Vollvirtualisierung Gastsystemen ein gesamtes virtuelles Betriebssystem zur Verfügung stellt. Weiters wird darin das Zusammenspiel der einzelnen Komponenten folgendermaßen beschrieben. Der Hypervisor läuft auf dem Host-System und ist jene Komponente, die die virtuelle Umgebung zur Verfügung stellt. Dieses vollwertige System beinhaltet ein BIOS sowie eine Abstraktion der gesamten Hardware, wodurch die Gast-Systeme nicht merken, dass sie virtualisiert werden. Zugriffe auf die Hardware werden vom Hypervisor intelligent auf die Hardware des Hosts verteilt oder über Hardware-Emulation zugänglich gemacht. Dieser Aufbau ist in Abb. 2.1 zu sehen. Einerseits zeigt die Abbildung, dass Anwendungen ohne Problem parallel zu den virtualisierten Systemen laufen können. Andererseits wird verdeutlicht, dass jedes virtuelle System ein gesamtes Betriebssystem (Gast-OS) beinhaltet. Jegliche Kommunikation eines Gastsystems mit den Systemressourcen des Hosts läuft über den Hypervisor, wodurch

---

<sup>3</sup><http://www.intel.de/content/www/de/de/virtualization/virtualization-technology/intel-virtualization-technology.html>



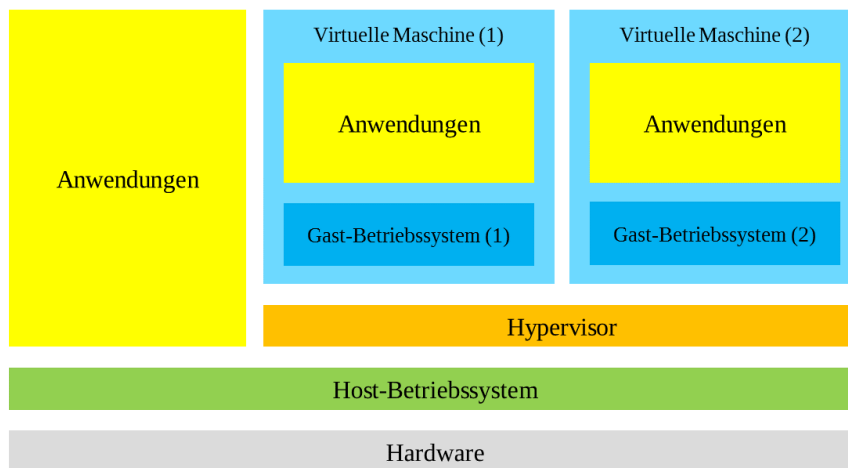


Abbildung 2.1: Architektur der Vollvirtualisierung

gewährleistet wird, dass dieser Zugriff kontrolliert werden kann. Dies stellt sicher, dass die virtuellen Maschinen voneinander abgeschottet sind und ein Ausfall eines Gastsystems die anderen nicht beeinflusst.

Aus dieser hohen Abstraktion folgt, dass am Gastsystem keine Änderungen durchgeführt werden müssen und unterschiedliche Kernels parallel virtualisiert werden können. Lediglich die Prozessorarchitektur muss zusammenpassen. Der Vorteil eines gesamten Betriebssystems pro Gastsystem bringt allerdings den Nachteil mit sich, dass jedes dieser Systeme Speicher für den Kernel und das Betriebssystem belegt. Weiters ist ein Bootvorgang nötig, der die Verwendung zu Entwicklungszwecken verzögert. Allerdings gibt die Vollvirtualisierung dem Entwickler die Möglichkeit, seine Software auf dem lokalen Rechner auf verschiedenen Betriebssystemen zu testen oder Werkzeuge zu verwenden, die auf dem Host-System nicht verfügbar sind. Außerdem können vorgefertigte *Basissysteme* (Snapshots) für die Entwickler bereitgestellt werden, um die Konfigurationszeit zu verkürzen und eine einheitliche Entwicklungsumgebung zu haben. Folgende Probleme treten hingegen in einer virtualisierten Entwicklungsumgebung auf:

- Updates müssen entweder doppelt durchgeführt werden oder die Entwickler werden periodisch mit neuen Snapshots versorgt.
- Für die Daten und Einstellungen der Entwickler muss ein Platz außerhalb der virtuellen Maschine gefunden werden. Besonders bei einem Snapshot-basierten Updateverfahren gehen durch die Verwendung eines neuen Snapshots alle Daten in der VM verloren. Die Konfiguration der Entwicklungsumgebung können idealerweise auf Netzwerklaufwerke verlegt werden und die Quelltexte sollten sich in einem Versionsverwaltungssystem befinden.

- Ein weiteres Problem im Zusammenhang mit den Konfigurationsdateien betrifft die Authentifizierung. Für SSH-Schlüssel für Versionsverwaltungssysteme und andere Zugangsdaten muss ebenfalls eine Möglichkeit geschaffen werden, diese in die virtuellen Maschinen zu bringen.
- Entwicklungsumgebungen gelangen bei virtualisierten Systemen schnell an die Performancegrenzen, wodurch die Produktivität der Mitarbeiter sinkt.
- Bei mehreren parallelen Projekten und damit verbundenen virtuellen Maschinen kann der Speicherplatz auf dem Host-System sehr knapp werden, da für jede VM das gesamte Betriebssystem gespeichert werden muss.

Wie sich die Vollvirtualisierung für Entwickler in der Praxis verwenden lässt, wird in Abschnitt 3.2.1 dargestellt.

## 2.3 Containervirtualisierung

Die Containervirtualisierung, auch manchmal Betriebssystemvirtualisierung genannt, bringt ähnliche Ergebnisse hervor wie die Vollvirtualisierung, hat allerdings eine fundamental andere Funktionsweise. Die folgenden Erläuterungen bauen auf [BAM15] auf, worin ein sehr guter Überblick über die Funktionsweise der Containervirtualisierung gegeben wird. In Abb. 2.2 ist die wesentlich schlankere Architektur der Containervirtualisierung zu sehen. Darin fällt auf, dass es zwischen dem Host-Betriebssystem und den

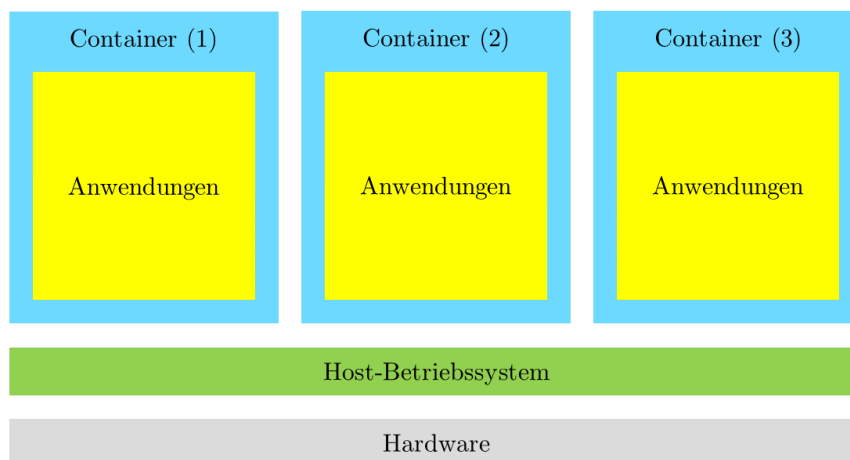


Abbildung 2.2: Architektur der Containervirtualisierung

Containern keine Zwischenschicht gibt. Im Fall der Containervirtualisierung ist der Host gleichzeitig der Hypervisor, wodurch eine Virtualisierung mit einem wesentlich geringeren Overhead entsteht. In manchen Grafiken zur Containervirtualisierung existiert eine

Schicht zwischen dem Host und den Containern, welche allerdings lediglich darstellt, dass die Container irgendwie gestartet werden müssen. Tatsächlich wird eine abgeschottete Instanz des Host-Betriebssystems erzeugt, aus der die Anwendungen nicht ausbrechen können. Sie erhalten lediglich Leserechte auf die mit dem Host-Betriebssystem geteilten Ressourcen wie Libraries und Binaries. Ein weit verbreitetes Synonym ist *Jails*, basierend auf dem Namen dieser Funktion in BSD<sup>4</sup>.

Diese Funktionsweise wird in [Die16] beschrieben. Jede virtualisierte Instanz bekommt eine eigene Sicht auf den Kernel und dessen Funktionen, die aber strikt von den anderen virtualisierten Instanzen getrennt ist. So ist keine Vollvirtualisierung notwendig, da Betriebssystemfunktionen direkt vom Kernel verarbeitet werden. Dies führt zu der bei Containern üblichen extrem schnellen Startzeit. Da die Virtualisierung über eine geteilte Kernel-Instanz abgewickelt wird, besteht bei der Containervirtualisierung die Einschränkung, dass das Gast-System den gleichen Kernel unterstützen muss wie das Hostsystem. Für diese abgeschottete Instanz des Betriebssystems sind folgende zwei Funktionen aus dem Linux-Kernel notwendig:

1. **namespaces:** Namespaces sind die Grundlage für die Abschottung der Prozesse untereinander. Diese Namensräume ermöglichen es Prozessen, dass sie private *Prozess-IDs* (PIDs) verwalten. Diese Prozesse und ihre Kinder können die anderen PIDs nicht sehen, wodurch eine Abstraktion erreicht wird, die einer Virtualisierung ähnelt.
2. **cgroups:** Durch Namensräume wird zwar das Problem der Abschottung gelöst. Allerdings können sich Prozesse gegenseitig stören, da sie keine Limits bei der Verwendung ihrer Ressourcen auferlegt bekommen. *Control Groups* (cgroups) lösen dieses Problem, sie erlauben eine Zuteilung von RAM, CPU-Zeit und Disk-I/O zu einzelnen Prozessen. So können sich Prozesse gegenseitig keine Ressourcen wegnehmen. Es entsteht allerdings kein Overhead, da nativ am Kernel gearbeitet wird.

Das oft erwähnte *LXC*<sup>5</sup> (Linux Containers) [rkt17] verwendet **namespaces** und **cgroups** zum Erstellen und Verwalten von Containern. Es wurde früher vom Container-Marktführer Docker verwendet, zwischenzeitlich allerdings durch **libcontainer** ersetzt und nun durch **runc**<sup>6</sup> abgelöst. **runc** ist nun in der *Open Container Initiative* (OCI, siehe Abschnitt 2.3.2) standardisiert. Die in diesem Kapitel beschriebene Containervirtualisierung trifft lediglich auf die Linux-Container zu. Microsoft arbeitet gerade an einer ersten Version der Windows Container<sup>7</sup>, deren Dokumentation allerdings mit „This is preliminary content and subject to change.“ eingeleitet wird, weshalb diese Technologie aufgrund des Erscheinungsdatums aus dieser Arbeit ausgeklammert wurde.

---

<sup>4</sup>[https://www.freebsd.org/doc/de\\_DE.ISO8859-1/books/handbook/jails.html](https://www.freebsd.org/doc/de_DE.ISO8859-1/books/handbook/jails.html)

<sup>5</sup><https://linuxcontainers.org/>

<sup>6</sup><https://runc.io/>

<sup>7</sup><https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>

### 2.3.1 Implementierungen von Containervirtualisierungslösungen

Für das Verwalten und Starten von Containern gibt es zahlreiche Lösungen, wobei [rkt17] einen Vergleich dieser Ansätze bietet. Im Folgenden werden die wichtigsten dieser Lösungen charakterisiert.

**Docker** Da Docker die in dieser Arbeit verwendete Technologie darstellt, wird sie in Kapitel 4 ausführlich behandelt und die Verwendung exemplarisch gezeigt.

**rkt** **rkt** (ausgesprochen: rocket) ist ein Container-Manager des quelloffenen Unix-Systems CoreOS. Im Gegensatz zu Docker ist es lediglich ein einzelnes ausführbares Programm ohne Hintergrundprozess (Daemon). Die Images der Container werden über HTTPS verteilt. Durch den fehlenden Daemon kann **rkt** sehr einfach in bestehende Systeme wie **systemd** und **Kubernetes** integriert werden. **rkt** baut auf dem standardisierten Containerformat **appc**<sup>8</sup> auf.

**runc** **runc** ist eine sehr systemnahe Möglichkeit, Container zu starten und zu verwalten. Sie bietet allerdings keine Möglichkeit, Images herunterzuladen und erfordert hohe Detailkenntnisse des Betriebssystems und dessen Konfiguration.

**containerd** **containerd** ist ein Daemon für **runc**. Auch hier ist keine Möglichkeit für den Download von Images vorhanden. **containerd** und **runc** bieten die Basis für Docker seit der Version 1.11.0.

**LXC** Der primäre Einsatzzweck von **LXC** ist das Starten einer kompletten Linux-Version parallel zum Host-System. Anwendungscontainer können auch verwaltet werden, erfordern allerdings mehr Wissen als zum Beispiel mit Docker oder **rkt**.

### 2.3.2 Open Container Initiative

Um einen industrieweiten Standard für Container zu erstellen, hat Docker auf der DockerCon 2015 die damals noch *Open Container Project* genannte Open Container Initiative (OCI) unter der Linux Foundation vorgestellt [Gol15]. Folgende zwei Standards der OCI werden in [Lin16] beschrieben:

1. **runtime-spec**:<sup>9</sup> Die Runtime-Specification regelt, wie ein bereits entpackter Container (Filesystem Bundle) ausgeführt wird.
2. **image-spec**:<sup>10</sup> Das Ziel der Image-Specification ist es, ein Format zu definieren, das das Erstellen, Verifizieren, Benennen und Verteilen von Images ermöglicht.

Der OCI sind mittlerweile zahlreiche große Unternehmen beigetreten, um Container-technologien zu standardisieren. Einige davon sind: Amazon, Cisco, Docker, Facebook, Hewlett Packard, IBM, Intel und Microsoft.<sup>11</sup>

---

<sup>8</sup><https://github.com/appc/spec/>

<sup>9</sup><https://github.com/opencontainers/runtime-spec>

<sup>10</sup><https://github.com/opencontainers/image-spec>

<sup>11</sup><https://www.opencontainers.org/about/members>

### 2.3.3 Orchestrierungssysteme

Im Folgenden werden kurz die aktuell wichtigsten Orchestrierungssysteme für Container [Die17] und deren Zweck beschrieben. Softwareentwickler werden diese weder am Entwicklungsrechner benötigen noch aktiv in der Produktion einsetzen. Jedoch sollten die Entwickler zumindest wissen, was Orchestrierung ist und warum sie notwendig ist. Ein Basiswissen dazu genügt, um zu entscheiden, ob das vorgeschlagene Werkzeug im konkreten Problemfall wirklich relevant ist. Die Funktionsweise und Philosophie der Containerorchestrierung sollte schon im Entwicklungsprozess bedacht werden, da die Architektur von Anwendungen davon stark beeinflusst wird. Das Wissen über die Behandlung von Containern verhindert Fehler beim Programmieren und zwingt die Entwickler, auch an den Betrieb der Software zu denken, wodurch die Idee des DevOps in den Entwicklungszyklus integriert wird.

Die Orchestrierung von Containern kann auch als *Besetzung* bezeichnet werden, wobei den jeweiligen Aufgaben in einem Softwaresystem mithilfe von Containern Services zugewiesen werden. Diese Zuteilung geschieht deklarativ, wodurch gleichzeitig eine skalierbare Konfiguration entsteht. Das Orchestrierungssystem weiß, welche Container es für welchen Zweck einsetzen muss und wie diese skaliert werden müssen. An dieser Stelle wird meistens die Cattle-vs.-Pet-Analogie verwendet [Bia16]. In containerbasierten Systemen werden die Serviceinstanzen nicht als Haustiere (Pets) angesehen, um die sich der Administrator händisch kümmern muss und alles versucht, um sie am Leben zu erhalten. Stattdessen werden die Container als Herde (Cattle) betrachtet, bei der der Ausfall eines Services durch die große Menge kompensiert wird. Außerdem ist das System so gebaut, dass das Orchestrierungswerkzeug den Ausfall eines Containers erkennt und einen neuen Container dieser Art startet und konfiguriert. Die Folge dieser Strategie ist, dass das Softwaresystem so gebaut werden muss, dass es den Ausfall und womöglich häufigen Neustart einzelner Services verkraftet.

[Die17] liefert einen Überblick über die aktuell am meisten verwendeten Werkzeuge zur Containerorchestrierung. Diese werden nachfolgend vorgestellt:

**Kubernetes** Wenn es um die Verwaltung von sehr vielen Containern (mehrere tausend) geht, dann ist der Cluster-Manager Kubernetes<sup>12</sup> von Google das Richtige. Die Architektur ist nach dem Master/Slave-Schema aufgebaut. Der Master verteilt die Arbeit auf zahlreiche Knoten, die Slaves. Die Container werden abstrahiert, indem sie zu sogenannten Pods zusammengefasst werden. Kubernetes wird über die Kommandozeile konfiguriert, wodurch es Einsteiger mit einer sehr steilen Lernkurve konfrontiert.

**DC/OS** DC/OS<sup>13</sup> ist ein Cluster-Manager, der nicht nur Container verwalten kann, sondern auch benutzerdefinierte Pakete. Durch diese Flexibilität ist DC/OS sehr komplex, verfügt jedoch über eine automatische Installation und eine grafische Benutzeroberfläche. Zu großer Verwirrung können die verwendeten Namen führen.

---

<sup>12</sup><https://kubernetes.io/>

<sup>13</sup><https://dcos.io/>

DC/OS steht für „Datacenter Operating System“. Ein Teil von DC/OS heißt Mesos und die Software selbst wurde von der Firma Mesosphere entwickelt.

**Rancher** Rancher<sup>14</sup> ist ein Orchestrierungsprodukt der Firma RancherOS. Wie Kubernetes ist die Architektur aufgeteilt in einen Master und mehrere Knoten, hier Agenten genannt. Rancher selbst ist ein Docker-Container, der beim Starten privilegierte Rechte erhält, um selbst wieder Container starten zu können. Rancher erweitert die bestehenden Docker-Componenten, verwendet allerdings dieselbe Syntax, wodurch ein schneller Einstieg ermöglicht wird. Zusätzlich gibt es für die Verwaltung eine grafische Benutzeroberfläche. Wenn die Anwendung über die Möglichkeiten von Rancher hinauswächst, bietet Rancher an, Kubernetes oder Mesos zu integrieren und Rancher lediglich als Verwaltungswerkzeug zu verwenden.

---

<sup>14</sup><http://rancher.com/rancher/>

## 3 Konfigurationsmanagement

Im Zusammenhang mit Docker und einem Softwareentwicklungsprozess werden oftmals zahlreiche weitere Werkzeuge genannt, die ähnliche Einsatzgebiete haben oder in Kombination mit Docker den Prozess erheblich verbessern können. In den folgenden Abschnitten werden die bekanntesten und am weitest verbreiteten Werkzeuge und deren jeweilige Einsatzzwecke vorgestellt. Zusätzlich werden exemplarische Szenarien, in denen Docker gewinnbringend eingesetzt werden könnte, aufgezeigt.

Die vorgestellten Werkzeuge dienen der Konfiguration der Entwicklungs- und Produktionsumgebung. Wie [Wol14, S. 29 f.] beschreibt, ließen sich diese Konfigurationen auch manuell vornehmen, doch damit träten erhebliche Probleme im Sinne der Testbarkeit, Reproduzierbarkeit und Automatisierung auf. Ein noch größeres Problem stellt das implizite Wissen der Entwickler dar, das bei einer händischen Konfiguration nur verbal weitergegeben wird. Selbst eine ausführliche Dokumentation löst dieses Problem nicht, da der Mensch im Gegensatz zum Computer dazu tendiert, fest definierte Abfolgen trotz genauer Spezifikation nicht fehlerlos durchzuführen, weshalb die Idee der Infrastrukturverwaltung in Versionsverwaltungssystemen entstanden ist.

### 3.1 Infrastructure as Code

Da jede Anwendung eine Umgebung zur Ausführung benötigt und gerade in den letzten Jahren die Komplexität der Anwendungen im Hinblick auf die Anzahl und das Zusammenspiel zahlreicher Komponenten stark zugenommen hat, gibt es den Ansatz des *Infrastructure as Code* [Fow16]. Dabei werden, wie zuvor bereits beschrieben, die Vorteile von Versionsverwaltungssystemen auf die Infrastruktur einer Anwendung angewandt. Die Anwendungsinfrasktruktuktur wird nicht mehr händisch aufgebaut, sondern in Quelltexten abgelegt, wodurch das tatsächliche Erstellen der Infrastruktur auf diverse Werkzeuge verteilt wird.

Dadurch können, wie in [Wol14, S. 64 ff.] beschrieben, zahlreiche Vorteile erreicht werden:

- Das System wird reproduzierbar, da manuelle Konfigurationsfehler vermieden werden.
- Bei einer konsequenten Durchführung entstehen idente Test- und Produktionsumgebungen, die sich bis zur Netzwerkebene nicht unterscheiden.
- Inkonsistent gewordene Systeme müssen nicht zwangsläufig repariert werden, wodurch sich Änderungen oder weitere Probleme ergeben können, sondern können entsorgt und sofort wieder neu erstellt werden.

- Die Infrastruktur ist nun reviewfähig, wodurch Probleme frühzeitig entdeckt werden. Unter reviewfähig ist hier zu verstehen, dass die Infrastruktur aufgrund der Darstellung in Quelltext von dritter Seite geprüft und validiert werden kann.
- Wenn zusätzlich zu Infrastructure as Code auch noch verstärkt mit Virtualisierungstechniken gearbeitet wird, lässt sich die Umgebung der Anwendung unter hoher Last aufgrund der replizierbaren Umgebung einfach skalieren. Dies bringt allerdings nur einen Vorteil, wenn die Anwendung auf eine horizontale Skalierung ausgelegt ist.
- Eine Dokumentation der gesamten Infrastruktur entsteht automatisch. Dies geschieht, ohne dass sie aktualisiert werden muss, denn der Infrastruktur-Quelltext ist zugleich die Dokumentation.
- Durch diese Selbstdokumentation wird auch gewährleistet, dass an jede Softwareversion auch die dazu benötigte Infrastrukturversion gebunden wird. Diese Tatsache vereinfacht das Testen und das Ausrollen der Software um ein Vielfaches, da die Anforderungen an die Umgebung bekannt sind. Voraussetzung dafür ist allerdings, dass sich der Anwendungs- und Infrastrukturquelltext in einem gemeinsamen oder zumindest verknüpften Repository der Versionsverwaltung befinden.

Damit dieses System funktioniert, dürfen Änderungen an der Infrastruktur allerdings *nur* in den Konfigurationsdateien geändert werden. Die Infrastruktur muss auf Basis dieser erstellt und darf keinesfalls manuell adaptiert werden.

Da die Infrastruktur auf Basis dieser Konfiguration erstellt wird, gibt es keinen Datenrückfluss aus den Systemen in die Konfiguration.

## 3.2 Werkzeuge

Um die Verwaltung der Infrastruktur zu erleichtern, existieren zahlreiche Werkzeuge, von denen die wichtigsten im Folgenden beschrieben werden.

### 3.2.1 Vagrant

Die folgenden Informationen und Quelltext-Beispiele sind aus [Var16] entnommen.

Vagrant ist ein Verwaltungswerkzeug für virtuelle Maschinen (siehe Abschnitt 2.2). Die Idee von Vagrant ist, bereits bestehende Virtualisierungslösungen wie VMware, VirtualBox oder AWS zu verwenden. Diese werden mit Werkzeugen, die sich um die Bereitstellung von Software kümmern (vgl. Chef in Abschnitt 3.2.2 oder Puppet in Abschnitt 3.2.3) zu fertigen virtuellen Maschinen für Entwickler oder Infrastrukturmanager erweitert. Aufgrund dieses Aufbaus können virtuelle Maschinen sehr einfach erweitert und daher generisch verwendet werden. Um nicht immer jede virtuelle Maschine komplett neu erstellen zu müssen, gibt es in Vagrant bereits vorgefertigte Schablonen für virtuelle Maschinen, sogenannte *Boxen*, die als Basis verwendet werden können.



Vagrant verwendet dazu eine eigene, auf Ruby basierende *Domain Specific Language* (DSL), die in den so genannten *Vagrantfiles* Einsatz findet. Das Vagrantfile wird mit dem Quelltext mitversioniert und bietet den Entwicklern reproduzierbare, portable und plattformunabhängige Entwicklungsgeräte. Das Konzept der Vagrantfiles ist auch in Docker (siehe Abschnitt 4.6) wiederzufinden. Dadurch verliert der bekannte Satz “But it works on my machine!” stark an Bedeutung und Relevanz. Anwendungen, die in der von Vagrant zur Verfügung gestellten Umgebung laufen, laufen immer auch in einer identen Umgebung, die sehr einfach mit Vagrant zur Verfügung gestellt werden kann.

### Beispiel eines Apache-Webservers mit Vagrant

Mit den folgenden Skripten wird ein Apache-Webserver in einer virtuellen Maschine gestartet. Quelltext 3.1 ist ein ausführbares Shell-Skript, welches den Apache-Webserver installiert und vom Vagrantfile verwendet wird.

```
1  #!/usr/bin/env bash
2
3  apt-get update
4  apt-get install -y apache2
5  if ! [ -L /var/www ]; then
6      rm -rf /var/www
7      ln -fs /vagrant /var/www
8  fi
```

Quelltext 3.1: Skript zum Installieren des Apache-Webservers (*bootstrap.sh*)

Der Vagrantfile in Quelltext 3.2 beschreibt, dass die Box *hashicorp/precise64* verwendet wird und zum Erstellen der fertigen virtuellen Maschine ein Shell-Skript namens *bootstrap.sh* ausgeführt werden soll. Der Parameter "2" in der ersten Zeile gibt die Version des Vagrant-Konfigurationsobjekts an. Dadurch kann Vagrant die Rückwärtskompatibilität zu älteren Versionen gewährleisten.

```
1  Vagrant.configure("2") do |config|
2      config.vm.box = "hashicorp/precise64"
3      config.vm.provision :shell, path: "bootstrap.sh"
4  end
```

Quelltext 3.2: Vagrantfile

Das Haupteinsatzgebiet von Vagrant liegt im Verwalten von virtuellen Maschinen für Entwickler, die die Produktionsumgebung spiegeln und ein plattformunabhängiges Entwickeln ermöglichen. Ein besonders nützlicher Anwendungsfall für Vagrant-Boxen sind Legacy-Systeme, für die eine vorkonfigurierte Entwicklungsumgebung benötigt wird, die auf keinem aktuellen Entwicklerrechner mehr verwendet wird. So kann für einfache Fehlerbehebungen eine Vagrant-Entwicklerbox gestartet werden, ohne dass am Entwicklerrechner nicht mehr benötigte Werkzeuge installiert sein/werden müssen.

### 3.2.2 Chef

Wie auf der offiziellen Homepage [Che16] von Chef Software, Inc. beschrieben, ist Chef ein Werkzeug zur Infrastrukturautomatisierung.

[Wol14] liefert eine sehr gute und prägnante Einführung in das Werkzeug Chef. Im Gegensatz zu einer Sammlung von Shell-Skripten bietet Chef ein Framework für die Konfiguration und das Erstellen von Softwaresystemen. Bestehende Funktionalität wird von Chef angeboten und die Konfigurationen werden in einer Meta-Sprache geschrieben. Dies hat den Vorteil, dass Sonderfälle minimiert werden, da sie von Chef übernommen werden. Zum Beispiel muss ein schon vorhandenes veraltetes System nicht komplett entfernt und neu installiert werden, sondern kann durch das Aktualisieren der Anforderungen auf den neuen, aktuellen Zustand gebracht werden.

Chef verwendet ebenso wie Vagrant eine Ruby-DSL, die in den so genannten *Rezepten* (*Recipes*) Verwendung findet. Diese Rezepte definieren die Anforderungen an die einzelnen Teile der Infrastruktur. Darin werden Konfigurationsdateien, Benutzer, Anwendungen, Plug-ins und weitere Abhängigkeiten spezifiziert. Da diese Rezepte mit steigender Komplexität des Systems an Übersichtlichkeit verlieren, gibt es die Gruppierung dieser in *Kochbücher* (*Cookbooks*). Diese Kochbücher fassen mehrere Rezepte zusammen und bieten die Möglichkeit der *Vorlagen* (*Templates*) für Konfigurationsdateien, wodurch zahlreiche Server unterschiedlich parametrisiert werden können. Chef kann entweder in einem „Solo“-Operationsmodus betrieben werden oder als „Chef Zero“ für Testzwecke. Zusätzlich gibt es auch die Option einer Client-Server-Umgebung, in der ein Chef-Server das Management der Konfigurationen übernimmt. Von dort aus können mit geringem Aufwand neue Teile der Infrastruktur gestartet und bestehende gewartet werden. Dazu existieren als Erweiterung zu Chef Werkzeuge wie Knife<sup>1</sup>.

### 3.2.3 Puppet

Der Zweck von Puppet ist laut [Pup16] wie bei Chef die Automatisierung der IT-Infrastruktur. [Wol14] liefert einen Vergleich der beiden Werkzeuge.

Puppet verfolgt ebenso den Ansatz einer Beschreibung des Endzustandes des Systems anstelle von Installationsskripten. Im Gegensatz zu Chef werden bei Puppet die Konfigurationen rein deklarativ beschrieben. Daher reicht als Beschreibungsformat *JSON*. Die möglichen Anweisungen können mit Ruby-Code erweitert werden. Dadurch wird wie bei Chef eine sehr hohe Flexibilität erreicht, viele der Aufgaben sind allerdings durch die deklarative Beschreibung leichter verständlich. Diese Unterschiede in der Syntax reflektieren die ursprüngliche Anwendergruppe. Chef war durch die starke Verwendung von Ruby eher für Entwickler gedacht. Puppet hingegen durch die deklarative Beschreibung der Konfigurationen für Systemadministratoren, die an Konfigurationsdateien und die Kommandozeile gewöhnt sind.

In [UpG14] wird beschrieben, dass der Chef-Ansatz eine höhere Flexibilität ermöglicht, dies allerdings mit einer steileren Lernkurve einhergeht. Der modellgetriebene Ansatz

---

<sup>1</sup><https://docs.chef.io/knife.html>

von Puppet ist zwar simpel, liefert jedoch im Gegensatz zu einer prozeduralen Systembeschreibung Interpretationsspielraum. Beim Erstellen des Systems muss sichergestellt werden, dass die Kommandos jedes Mal gleich erzeugt werden, was durch Änderungen am Algorithmus nicht garantiert werden kann.

[UpG16] liefert eine Einführung in die weiterführenden Systeme und Werkzeuge der Firmen Chef<sup>2</sup> und Puppet<sup>3</sup>.

### 3.2.4 SaltStack

SaltStack ist eine Open-Source-Softwarelösung für die Konfiguration und Automatisierung von Servern. Die folgenden Informationen stammen aus [Sal16b].

SaltStack basiert auf einer Master-Slave-Architektur, bei der sich die Slaves (in SaltStack *Minions* genannt) beim Master registrieren und ab diesem Zeitpunkt steuern lassen. Der einfachste Anwendungsfall ist die Ausführung von Kommandos auf zahlreichen verteilten Servern. Dazu sendet der Master die Kommandos an die Minions, die diese ausführen. Um Reproduzierbarkeit und dadurch Konfigurationsmanagement zu erreichen, werden die Konfigurationen mit YAML in SLS-Dateien (SaLt State) definiert und an die Minions verteilt. Diese States werden für einzelne Gruppen definiert, wobei jeder Minion seine Gruppenzugehörigkeit kennt und die zu ihm passenden States anwendet.

Laut [Sal16a] bietet SaltStack die Möglichkeit von ereignisgesteuerten Infrastrukturen. Alle Aktionen im SaltStack-System führen zu Ereignissen, auf die reagiert werden kann. So entstehen reaktive Infrastrukturen, die sich selbst bereitstellen, verwalten und reparieren können. Dazu lassen sich für Ereignisse und Ereignisgruppen Beobachter und Reaktoren definieren, die wissen, welche Aktionen bei welchen Ereignissen oder Ausnahmen zu tätigen sind. So kann zum Beispiel bei einer zu voll werdenden Festplatte neuer Speicher zur Verfügung gestellt oder der bereits vorhandene Speicher defragmentiert werden. Weiters sind auch Szenarien möglich, bei denen bei steigender Last oder Knotenausfall neue Server bereitgestellt werden.

### 3.2.5 Ansible

Wie die zuvor beschriebenen Werkzeuge bietet auch Ansible die Automatisierung von IT-Systemen. Die folgenden Informationen stammen aus [Ans16].

Die Konfigurationsdateien in Ansible werden *Playbooks* genannt und sind wie bei SaltStack in YAML geschrieben. Im Gegensatz zu anderen Werkzeugen muss bei Ansible am Client kein Service laufen, da Ansible die Kommandos über SSH, WinRM oder Cloud-APIs ausführt. Lediglich auf dem Master ist Ansible installiert. Dieser weiß zusätzlich über ein in Ansible *Inventory* genanntes Verzeichnis Bescheid, welche Clients im System verfügbar sind und wie er diese anzusteuern hat.

Die eben beschriebene CLI-Funktionalität wird durch Ansible Tower um ein grafisches Benutzeroberfläche erweitert. Dadurch können die Ansible-Aufgaben sehr einfach verwaltet und dem gesamten Team zur Verfügung gestellt werden.

---

<sup>2</sup><https://www.chef.io/about/>

<sup>3</sup><https://puppet.com/company>

Mit der Entwicklung von Ansible wurde erst 2012 gestartet, wodurch viele der Probleme aus anderen Systemen behoben wurden [Wol14]. Dieser technisch besseren Basis steht allerdings der Nachteil der kleineren Benutzergruppe gegenüber.

### 3.2.6 Packer

Packer ist ein Werkzeug zum Erstellen von vorkonfigurierten virtuellen Maschinen. In [Has16] werden Anwendungsszenarien und Einsatzmöglichkeiten gezeigt.

Das quelloffene Kommandozeilenwerkzeug verwendet Konfigurationsmanagementwerkzeuge wie Chef oder Puppet zum Konfigurieren und Erstellen von virtuellen Maschinen. Der große Vorteil darin besteht in der Abstraktion des Zielsystems. Packer erstellt virtuelle Maschinen auf Basis von Konfigurationsdateien, wobei die Zielplattform unabhängig von der Konfiguration ist. Dadurch kann eine beinahe idente virtuelle Maschine für die Produktion sowie für die Entwickler zur Verfügung gestellt werden, indem auf Basis einer Konfiguration beispielsweise virtuelle Maschinen für Amazon AWS (Produktion) und Vagrant (Entwicklung) erstellt werden.

Packer abstrahiert die Plattformunterschiede, wodurch bei der Konfiguration von virtuellen Maschinen die Zielplattform nicht feststehen muss. Diese lässt sich bei der Verwendung von Packer im Nachhinein sehr einfach anpassen.

Durch die Verwendung von Packer wird eine zusätzliche Abstraktionsstufe geschaffen, die Vendor-Lock-ins reduziert. Auch Docker-Container können mithilfe von Packer erzeugt werden. Auf diese Weise kann dasselbe Resultat wie bei der Verwendung von Dockerfiles (siehe Abschnitt 4.6) erreicht werden, wobei keine Bindung an Docker entsteht, allerdings ein zusätzliches Werkzeug verwendet werden muss, das nicht in das Docker-Ökosystem integriert ist.

## 3.3 Kombinationsmöglichkeiten der Werkzeuge

Eine sehr durchdachte Kombination der eben vorgestellten Werkzeuge befindet sich bei der Firma npm Inc.<sup>4</sup> im Einsatz. npm ist der Paketmanager für die serverseitige JavaScript-Runtime Node.js. Zusätzlich existieren die freie npm-Registry und Kommandozeilenwerkzeuge zum Verwalten der JavaScript-Pakete. Die Firma npm Inc. entwickelt diese und hostet die Registry. Der starke Open-Source-Ansatz von npm führte zu dem Blogeintrag [npm16], worin beschrieben wird, welche Werkzeuge beim npm-Deployment im Einsatz sind. Daraus entstammen die folgenden Überlegungen.

```
1 # change to service directory
2 cd ~/code/exciting-service
3 # push to production
4 git push origin +master:deploy-production
```

Quelltext 3.3: Deployment-Prozess bei npm

---

<sup>4</sup><https://www.npmjs.com/about>

Die Motivation des Deployment-Prozesses von npm liegt in der höchstmöglichen Simplizität für die Entwickler. Daraus resultiert ein häufigeres Deployment, da nicht darüber nachgedacht werden muss, ob ein Deployment momentan überhaupt möglich ist. In Quelltext 3.3 ist der gesamte Deployment-Aufwand dargestellt. Ein git-Push auf den „production“-Branch genügt, um den neuen Code samt Infrastruktur zu veröffentlichen.

In Abb. 3.1 ist dargestellt, welche Aufgaben die Werkzeuge des Konfigurationsmanagements dabei übernehmen. Die gesamte npm-Infrastruktur läuft auf ca. 120 virtuellen Maschinen in der Amazon-Cloud. Um den Deployment-Prozess zu beschleunigen, wird von Packer in ① ein Basis-VM-Image mit Ubuntu erstellt. Darin befindet sich bereits eine LTS-Version von Node.js als Laufzeitumgebung und das npm-interne Monitoring-System.

*Terraform*<sup>5</sup> kommt als nächstes Werkzeug zum Einsatz. Terraform ist ein weiteres Werkzeug zur Verwaltung von Infrastruktur als Code. Der Haupteinsatzzweck liegt in der deklarativen Verwaltung von Cloud-Ressourcen. Terraform dient dabei als Abstraktionsschicht zu den Cloud-Providern. Bei npm wird nun das von Packer erstellte Basis-Image von Terraform verwendet, um eine virtuelle Maschine in der Amazon-Cloud zu erstellen ①.

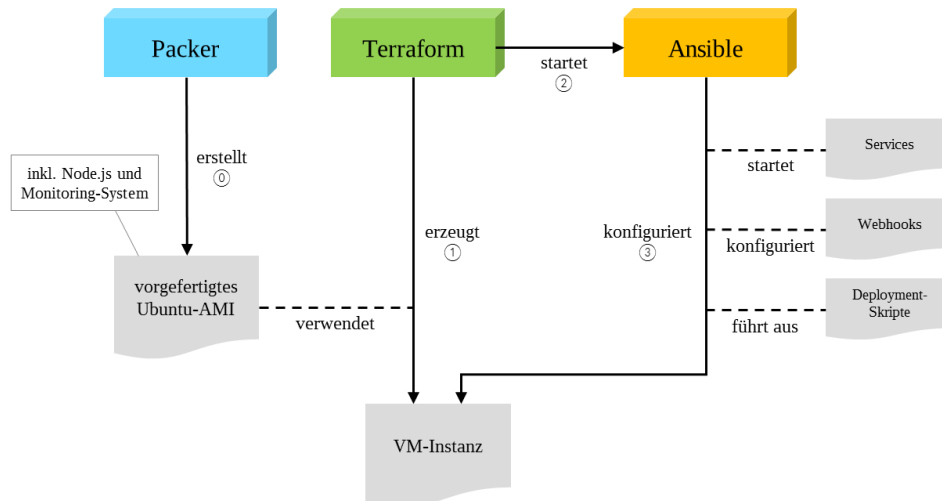


Abbildung 3.1: Packer, Terraform und Ansible im Einsatz bei npm

Danach wird Ansible mit einem Playbook aufgerufen ②, das die benötigten Services installiert und startet, Webhooks konfiguriert und zum Start der npm-Services die Deployment-Skripte ausführt ③. npm verwendet dazu keine externen Werkzeuge, sondern Bash-Skripte, die auch händisch gestartet werden können. Zur Konfiguration wird etcd<sup>6</sup> als Datenbank verwendet. Gegen ein Werkzeug spricht im Fall npm die Integration in den internen Slack-Bot, über den der gesamte Deployment-Prozess beobachtet und angestoßen

<sup>5</sup><https://www.terraform.io/>

<sup>6</sup><https://github.com/coreos/etcd>

werden kann.

Nach dem Start und der Konfiguration der virtuellen Maschine wird diese zuerst in einen Canary-Modus im Load-Balancer geschaltet. Dieser testet unter sehr geringer Produktivlast die Funktionsfähigkeit des Systems, bevor der neue Knoten tatsächlich unter volle Produktionslast gestellt wird.

## 4 Docker

Wie in Abschnitt 2.2 beschrieben, ermöglicht die Virtualisierung von IT-Systemen eine Abstraktion der Hardware. Ressourcen lassen sich effizienter nutzen, die Systeme werden jedoch nicht optimal ausgenutzt, da das gesamte Betriebssystem virtualisiert wird. Dies führt zu langen Startzeiten und einem hohen Speicherverbrauch.

Containervirtualisierung (siehe Abschnitt 2.3) löst diese Probleme durch die Wiederverwendung des Kerns. Dadurch entsteht die Möglichkeit, Anwendungen mitsamt den benötigten Systemressourcen zu verpacken und die Abhängigkeiten für den Betrieb zu reduzieren. Für einen Container wird lediglich die Laufzeitumgebung benötigt. Ist diese installiert, ist garantiert, dass der Container funktioniert. Docker<sup>1</sup> wird in 94% (Stand Juni 2016) der Unternehmen eingesetzt, die auf Containervirtualisierung setzen [DC16].

### 4.1 Geschichte

Der Blogbeitrag [Red15] schildert die Geschichte der Containervirtualisierung. Folgende Darstellung ist eine Zusammenfassung dieses Eintrags.

Im Jahr 2001 ermöglicht Jacques Gélinas mithilfe eines gepatchten Kerns zum ersten Mal das Ausführen mehrerer Linux-Server auf einem einzelnen Rechner, ohne auf Vollvirtualisierung zurückzugreifen.

2006 folgt die Aufnahme von cgroups in den Linux-Kern, worauf 2008 die Implementierung der namespaces folgt. Ebenfalls 2008 beginnt IBM mit der Entwicklung von LXC, wobei auf cgroups und namespaces aufgesetzt wird.

2013 wird Docker als Open-Source-Projekt der Platform-as-a-Service-Umgebung dot-Cloud vorgestellt. Docker führt nichts wesentlich Neues ein, sondern bietet lediglich für die bestehenden Technologien wie LXC eine sehr einfach zu verwendende Benutzerschnittstelle, die Container für eine breite Benutzergruppe zugänglich macht.

2014 erscheint Docker in der Version 1.0, nachdem mit Version 0.9 LXC als Containerumgebung durch *libcontainer*<sup>2</sup> ersetzt wird. *libcontainer* ist eine Eigenentwicklung von Docker, die in der Programmiersprache Go geschrieben ist und die Verwaltung und Ausführung von Containern ermöglicht.

2015 tritt Docker der Open Container Initiative bei, wodurch die Quelltexte und Entwicklungen dem unter der Linux Foundation stehenden Projekt hinzugefügt werden. Dieser Schritt lässt Kritiker verstummen, die eine Monopolstellung von Docker befürchtet haben. Seit 2015 werden hauptsächlich Fehlerbehebungen und Verbesserungen an Docker

---

<sup>1</sup><https://www.docker.com/>

<sup>2</sup><https://github.com/opencontainers/runc/tree/master/libcontainer>

selbst durchgeführt sowie das Docker-Ökosystem durch zahlreiche Werkzeuge erweitert (siehe Abschnitt 4.4).

Die in diesem Kapitel gezeigten Beispiele und verwendeten Funktionen basieren auf der Docker-Version 1.13 vom 19. Jänner 2017.

## 4.2 Funktionsweise

Die Funktionsweise von Docker ist in [Doc17c] beschrieben und wird im folgenden Abschnitt zusammengefasst. Docker basiert auf einer Vielzahl von Komponenten, die erst als Gesamtsystem eine Containervirtualisierung ermöglichen. Ein grundlegendes Wissen über diese Komponenten ist für die Verwendung von Docker unerlässlich. Folgende Komponenten sind essenziell:

**daemon** Im Hintergrund von Docker läuft der Server-Prozess, genannt **daemon**, der sich um das Anlegen und Verwalten von **images**, **container**, **networks** und **data volumes** kümmert. Er läuft auf dem Host-System und nimmt Kommandos der **clients** über eine REST-Schnittstelle entgegen.

**client** Der **client** ist die Benutzeroberfläche von Docker, die in Form einer Kommandozeilenanwendung zur Verfügung steht. Er nimmt Kommandos und Konfigurationen entgegen und sendet diese an einen zuvor festgelegten **daemon**.

**engine** Als **engine** wird die Kombination aus **daemon** und **client** bezeichnet, die den Kern von Docker bildet.

**image** Ein **image** ist eine Vorlage für das Erstellen eines **containers**. **Images** können erweitert und dadurch wiederverwendet werden. Sie bilden die Basis zum Erstellen eines **containers**. **Images** sollten immer deklarativ mithilfe von Dockerfiles (siehe Abschnitt 4.6) erstellt werden.

**container** **container** sind jene Teile, die von Docker ausgeführt werden. Sie sind die ausführbare Instanz eines **images**. **container** werden vom **daemon** gestartet, gestoppt, verschoben oder gelöscht, der die Anweisungen vom **client** enthält. Zusätzlich können **container** zur Inter-Container-Kommunikation Netzwerken zugewiesen oder zur Datenpersistierung mit Speicherbereichen versorgt werden.

**registry** **registries** dienen der Verteilung von **images**. Sie stellen eine Bibliothek dieser dar und können entweder privat oder öffentlich zugänglich sein (siehe Abschnitt 4.4).

**service** Ein **service** ist eine Sammlung von Docker-Containern, welche durch einen Schwarmmanager (siehe Abschnitt 4.4) verwaltet wird und die Basis für Multi-Container-Anwendungen darstellt. Seit Docker 1.12 werden **services** in vollem Ausmaß unterstützt.

In Quelltext 4.1 wird ein neuer Container auf Basis des Ubuntu-Images gestartet, in dem sodann die Bash gestartet wird.



```
1 $ docker run -i -t ubuntu /bin/bash
```

Quelltext 4.1: Ubuntu-Bash in Docker

Dabei erledigt die Docker-Engine folgende Schritte:

1. Das **ubuntu**-Docker-Image wird von der Docker-Registry heruntergeladen, falls dieses lokal noch nicht existiert.
2. Ein neuer Container wird auf Basis des Images erstellt und mit einem zufälligen Namen versehen.
3. Ein schreibfähiges Dateisystem wird zu dem eben erstellten Container hinzugefügt.
4. Für den Container wird eine Netzwerkverbindung geschaffen, die eine Kommunikation mit dem Host ermöglicht. Standardmäßig wird die **bridged**-Schnittstelle verwendet.
5. Der Container erhält eine IP-Adresse.
6. Das angegebene Programm wird ausgeführt. In diesem Fall ist das **/bin/bash**.
7. Die Parameter **-t** und **-i** führen dazu, dass die Standardeingabe und Standardausgabe verbunden werden und der Container in einen interaktiven Modus geschaltet wird. Der Benutzer befindet sich nun in der Bash in Ubuntu in Docker.

In Abb. 4.1 ist dieser Ablauf grafisch dargestellt.

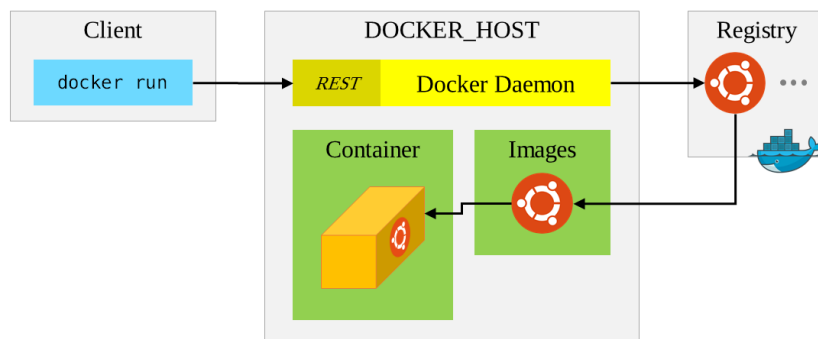


Abbildung 4.1: Docker-Architektur

## 4.3 Plattformen

Docker läuft nur auf Linux-Versionen nativ, deren Kerne zu Docker kompatibel sind. Auf allen anderen Systemen wird durch virtualisierte Umgebungen eine Möglichkeit geschaffen, Docker auszuführen. Die Werkzeuge für diese Systeme wurden im letzten Jahr

stark verbessert, wodurch die Verwendung von Docker unter Windows und macOS stark vereinfacht wurde, derzeit allerdings eine Vielzahl von Werkzeugen existiert. Nachfolgend wird ein kurzer Überblick gegeben.

### 4.3.1 Docker for Windows

Docker for Windows verwendet Hyper-V für die Erstellung einer virtuellen Linux-Maschine zur Ausführung von Docker [Doc17f]. Zusätzlich werden die Docker-Client-Anwendungen installiert. Nicht zu verwechseln ist Docker for Windows mit den Windows-Containern<sup>3</sup>, die sich noch in der Beta-Phase befinden. Zur Verwendung dieser gibt es auf der offiziellen Docker-Homepage eine Anleitung<sup>4</sup>. Eine sehr nützliche Erweiterung für die Verwendung von Docker unter Windows ist `posh-docker`<sup>5</sup>, welches die automatische Vervollständigung der Docker-Kommandos auf der Powershell ermöglicht. Vor allem durch die Integration des Port-Forwarding und der Möglichkeit, auch in Docker auf das gesamte Dateisystem zuzugreifen, fühlt sich die Verwendung von Docker for Windows beinahe so wie unter Linux an.

### 4.3.2 Docker for Mac

Docker for Mac ist das Pendant zu Docker for Windows. Es nutzt die Virtualisierungslösung HyperKit, welche seit macOS 10.10 Yosemite nativ verfügbar ist [Doc17g]. Ebenso wie in der Windows-Version bietet die Mac-Version eine grafische Benutzeroberfläche zur Verwaltung der Anwendung.

### 4.3.3 Docker Toolbox

Die Docker Toolbox war die erste offizielle Möglichkeit Docker auf Windows oder macOS laufen zu lassen [Doc17h]. Sie wird von Docker *nicht* mehr empfohlen, da es mit den nativen Lösungen nun zu wesentlich weniger Problemen und einem geringeren Overhead kommt. Docker Toolbox installiert die Docker-Client-Werkzeuge sowie Oracle VM VirtualBox. Darin wird eine virtuelle Maschine „boot2docker“ angelegt, auf der der Docker-Daemon läuft. Zusätzlich wird eine vorkonfigurierte git-Bash angeboten, die sich automatisch zu diesem Docker-Host verbindet.

Der Hypervisor für die virtuelle Maschine lässt sich mit etwas Mühe austauschen<sup>6</sup>, was aber zu zahlreichen Problemen führt. Wenn Docker for Windows und eine Virtualisierungslösung wie VMware oder VirtualBox gleichzeitig verwendet werden sollen, ergibt sich ein Konflikt zwischen diesen Hypervisoren und Hyper-V. Aufgrund von Problemen wie der fehlenden Möglichkeit, Dateien zwischen dem Host und Docker zu teilen, sollte allerdings anstatt der Docker Toolbox die Lösung von Scott Hanselman [Han14] in Betracht gezogen werden. Er erklärt darin eine Möglichkeit, mehrere Hypervisoren und Hyper-V auf einem System zu verwenden. Dies ist allerdings nur durch einen Neustart

---

<sup>3</sup><https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>

<sup>4</sup><https://docs.docker.com/docker-for-windows/install/>

<sup>5</sup><https://github.com/samneirinck/posh-docker>

<sup>6</sup><https://github.com/pecigonzalo/docker-machine-vmwareworkstation>

des Systems möglich. Diese Lösung ist zwar etwas umständlich, ermöglicht allerdings die Verwendung der aktuellsten Docker-Version und die geringsten Einbußen. Derzeit (Februar 2017) gibt es keine bessere Lösung.

## 4.4 Produkte

Das Docker-Ökosystem besteht mittlerweile aus zahlreichen Komponenten, wobei die Docker-Engine um zusätzliche Werkzeuge erweitert wird. Die folgenden Informationen stammen aus [Doc17e].

### Docker Hub

Der *Docker Hub* ist die offizielle Registry für Docker-Images. Dort werden offizielle Images wie Ubuntu, MySQL, Redis und zahlreiche weitere angeboten. Zusätzlich können im Docker Hub eigene Images hochgeladen werden, für die auch die Möglichkeit des automatischen Erstellens bei Änderungen in Github angeboten wird. Der Docker Hub ist auch eine exzellente Quelle zum Lernen von Best-Practices. Zu den meisten Images existieren Dockerfiles, die gelesen, studiert und nachgebaut werden können.

### Docker Machine

*Docker Machine* ist das Werkzeug zum Verwalten von Docker-Hosts. Dieses Kommandozeilenwerkzeug ermöglicht das Erstellen und Verwalten von virtuellen Hosts, auf denen Docker bereits konfiguriert und einsatzbereit ist. Docker Machine kann sowohl für die Konfiguration des Entwicklungsrechners verwendet werden als auch für die Provisionierung von Docker-Hosts in der Cloud oder in Datenzentren.

### Docker Compose

*Docker Compose* bietet die Grundlage für komplexe Anwendungen. Diese lassen sich oft nicht oder nur schwer in einen Container verpacken, da sie aus zahlreichen Services bestehen. Mithilfe eines eigenen auf YAML basierenden Dateiformates lassen sich diese Services deklarativ beschreiben und mithilfe von `docker-compose` ausführen und zentral verwalten.

### Docker Swarm

*Docker Swarm* ist die offizielle Lösung zum Betrieb von Docker in lastintensiven, kritischen und hochverfügbaren Umgebungen. Mithilfe von Docker Swarm lassen sich mehrere Hosts zu einem virtuellen Host zusammenfassen, der in Folge die Docker-Kommandos ausführt. Da die API ident ist, müssen keine Anpassungen durchgeführt werden, wodurch bestehende Docker-Kommandos weiterverwendet werden können. Allerdings bietet Docker Swarm eine wesentlich höhere Leistung und Ausfallsicherheit.

## 4.5 Dateisystem

Bevor im nächsten Kapitel das Erstellen von Containern mithilfe von Dockerfiles beschrieben wird, ist es wichtig, die Konzepte rund um das von Docker verwendete Dateisystem zu verstehen. Diese werden nachfolgend vorgestellt; sie sind aus [Doc17i] entnommen. In Abb. 4.2 ist dargestellt, wie das Dateisystem von Docker arbeitet.

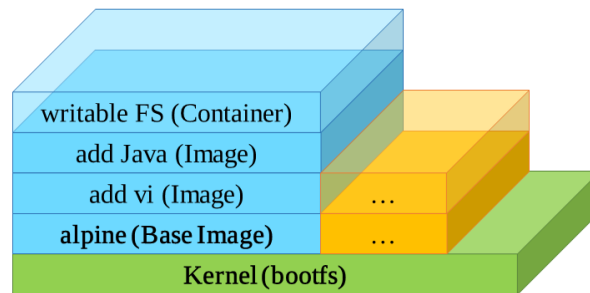


Abbildung 4.2: Docker-Dateisystem

Jedes Image besteht aus mehreren gestapelten Schichten. Die einzelnen Schichten merken dies allerdings nicht. Jede dieser Schichten bekommt eine kombinierte Sicht auf alle darunterliegenden. Sie kann selbst Dateien hinzufügen, ändern oder als gelöscht markieren, ändert allerdings die darunterliegenden Schichten nicht. Dieses Dateisystem-Konzept wird Union File System genannt.

Dadurch wird Docker sehr speichereffizient, da idente Schichten nur einmal existieren. Ein Image, das auf einem anderen aufbaut, teilt sich dieselbe Basisschicht, welche, durch einen eindeutigen Hash identifiziert, wiederverwendet werden kann. Dadurch wird auch beim Start von zahlreichen gleichen Containern beinahe kein Speicherplatz benötigt, denn die schreibgeschützte Image-Schicht wird von jedem Container wiederverwendet.

## 4.6 Dockerfiles

Zum Erstellen von Images gibt es in Docker zwei Ansätze:

1. Die erste Möglichkeit besteht darin, ein Basis-Image mit einer Shell zu starten und in dieser die benötigten Kommandos und Änderungen auszuführen. Nach allen ausgeführten Kommandos wird der Zustand dieses Containers eingefroren und in ein Image verwandelt. Dies führt allerdings dazu, dass dieser Prozess nur sehr schwer reproduzierbar ist.
2. Aus diesem Grund gibt es die sogenannten *Dockerfiles*. Sie stellen Bauanleitungen für Docker-Images dar, die an den Docker-Daemon übergeben werden, der auf Basis dieser das Image erstellt. Daraus resultiert ein reproduzierbarer und plattformunabhängiger Erstellungsprozess.

In Quelltext 4.2 wird ein Apache-Webserver auf einem Ubuntu-Linux aufgesetzt und für den Einsatz vorbereitet. Dieser Dockerfile wird von Kimbro Staken als quelltextoffenes Beispiel<sup>7</sup> geführt und beinhaltet einige der wichtigsten Dockerfile-Instruktionen. Nachfolgend werden die verwendeten Instruktionen beschrieben, eine vollständige Übersicht bietet [Doc17d].

**FROM** Die *FROM*-Anweisung spezifiziert das Basis-Image des Containers. Auf dieser Schicht wird der Container aufgebaut. Meistens wird dies ein offizielles Image des Docker Hubs sein, das weiter parametrisiert und konfiguriert wird.

**LABEL** Die *LABEL*-Anweisung ersetzt ältere Anweisungen wie zum Beispiel *MAINTAINER*. Sie bietet die Möglichkeit, Schlüssel-Wert-Paare zu einem Image zu speichern.

**RUN** *RUN* ermöglicht das Ausführen von Befehlen zum Erstellen des Images. Jedes *RUN* erzeugt eine neue Schicht des Images. In diesem Fall wird *RUN* verwendet, um mit dem Paketmanager **apt-get** Apache zu installieren und gleich im Anschluss das System wieder aufzuräumen.

**ENV** Mit dem *ENV*-Befehl werden Umgebungsvariablen gesetzt.

**EXPOSE** *EXPOSE* kennzeichnet die Netzwerk-Ports, auf denen der Container zur Laufzeit erreichbar ist. Die Ports werden lediglich als verfügbar gekennzeichnet und nicht an den Host gebunden. Ob und auf welchem Port der Container tatsächlich erreichbar ist, bestimmt der Anwender beim Starten dessen.

**CMD** *CMD* spezifiziert jene Anwendung, die standardmäßig beim Starten des Containers ausgeführt wird. Diese kann beim Starten des Containers überschrieben werden. Genauere Informationen sind in Abschnitt 4.6.3 beschrieben.

```
1 # A basic apache server. To use either add or bind mount
   ↪ content under /var/www
2 FROM ubuntu:12.04
3
4 LABEL maintainer="Kimbro Staken"
5 LABEL version="0.1"
6
7 RUN apt-get update && apt-get install -y apache2 && apt-get
   ↪ clean && rm -rf /var/lib/apt/lists/*
8
9 ENV APACHE_RUN_USER www-data
10 ENV APACHE_RUN_GROUP www-data
11 ENV APACHE_LOG_DIR /var/log/apache2
12
```

---

<sup>7</sup><https://github.com/kstaken/dockerfile-examples/blob/master/apache/Dockerfile>

13 EXPOSE 80

14

15 CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]

Quelltext 4.2: Apache-Dockerfile von Kimbro Staken

Um aus diesem Dockerfile ein Image zu erstellen, muss `docker build` ausgeführt werden. Dieses Kommando kann beispielsweise wie in Quelltext 4.3 verwendet werden.

1 \$ docker build -t kstaken/apache .

Quelltext 4.3: Erstellen eines Images auf Basis eines Dockerfiles

Der Parameter `-t` weist dem erstellten Image den Namen `kstaken/apache` zu. Dieser wird verwendet, um einen Container auf Basis dieses Images zu starten. Der Punkt am Ende des Kommandos spezifiziert den Build-Kontext. Alle Dateien in diesem Pfad sind während des Docker-Builds verfügbar und können dem Image hinzugefügt werden. Standardmäßig sucht Docker nach einer Datei namens `Dockerfile` in dem angegebenen Kontext. Dies kann mit dem Parameter `-f` überschrieben werden.

#### 4.6.1 Best Practices für das Erstellen von Dockerfiles

Für das Erstellen von Dockerfiles werden von Docker folgende Best-Practices vorgeschlagen [Doc17b]:

- Container sollen per Definition „flüchtig“ („ephemeral“) gestaltet werden. Damit ist gemeint, dass ein Container mit minimalem Aufwand gestoppt, gelöscht und neu gestartet werden kann. Der Container muss von seinem Zustand unabhängig sein. Dies bedeutet, dass Daten und Konfigurationen außerhalb des Containers existieren müssen.
- Ein Container soll so klein wie möglich sein (siehe Abschnitt 4.6.2). Unnötige Pakete sollen vermieden werden. Ein Datenbankcontainer benötigt beispielsweise keinen Texteditor.
- Jeder Container soll im Gesamtsystem lediglich *eine* Aufgabe haben. Durch diese Aufteilung können Systeme wesentlich besser skalieren. Diese Regel besagt nicht, dass ein Container lediglich einen Prozess ausführen darf. Prozesse können auch Kindprozesse starten, allerdings soll damit nur eine Aufgabe gelöst werden.
- Die Anzahl der Schichten eines Images soll möglichst geringgehalten werden. Alles in einen Befehl zu packen ist allerdings auch nicht der richtige Weg. Hier muss ein guter Mittelweg gefunden werden. Die offiziellen Images im Docker Hub bieten dafür sehr gute Lernressourcen.
- Die Argumente sehr langer Parameter sollen zur besseren Wartbarkeit alphabetisch sortiert werden.

- Durch die Wiederverwendung von Schichten bietet Docker die Möglichkeit eines Build-Caches. Unveränderte Teile des Images müssen nicht neu erzeugt werden. Daher sollen die Schichten so gestaltet werden, dass sich häufig ändernde Module möglichst spät hinzugefügt werden, da dies die Geschwindigkeit beim Erzeugen eines Images erheblich beschleunigt.

#### 4.6.2 Reduktion der Imagegröße

Einer der wichtigsten Punkte beim Erstellen von Docker-Images ist die Größe des resultierenden Images. Diese wirkt sich enorm auf die Startgeschwindigkeit und den Speicherbedarf von Containern aus und spielt daher bei komplexen Systemen mit vielen unterschiedlichen Containern eine große Rolle. Zur Optimierung der Imagegröße existieren zahlreiche Blogbeiträge, beispielsweise [Jon14], [Jan16] und [Cam16].

##### Basis-Image

Zur Demonstration des Einflusses des Basis-Images auf die Imagegröße wird in Quelltext 4.4 eine Shell in Ubuntu und Alpine gestartet.

```
1 $ docker run --rm -it ubuntu sh
2 $ docker run --rm -it alpine sh
```

Quelltext 4.4: Starten einer Shell in unterschiedlichen Linux-Containern

Das Starten der Container aus Quelltext 4.4 bewirkt das Herunterladen und Ausführen der in Quelltext 4.5 angeführten Images. Anhand der Imagegröße ist der immense Unterschied zu erkennen: Während Ubuntu beinahe 130 Megabyte Speicherplatz benötigt, kommt Alpine mit lediglich rund vier Megabyte aus. Alpine<sup>8</sup> ist eine sehr leichtgewichtige Linux-Distribution, die sich aufgrund ihrer Größe ideal für die Verwendung in container-basierten Systemen eignet. Eine vollwertige Linux-Distribution wie Ubuntu oder Debian wird von beinahe keinem Container benötigt, wird allerdings aus geschichtlichen und Komfortgründen selbst in vielen offiziellen Images<sup>9</sup> verwendet.

```
1 $ docker images
2 REPOSITORY          TAG          IMAGE ID      SIZE
3 ubuntu              latest      f49eec89601e  129 MB
4 alpine              latest      88e169ea8f46  3.98 MB
```

Quelltext 4.5: Vergleich der Basis-Imagegröße

##### Zusammenfassen der RUN-Kommandos

In Quelltext 4.6 ist ein Dockerfile abgebildet, der auf den ersten Blick sehr gut geschrieben aussieht. Als Basis-Image wird die platzsparende Alpine-Linux-Distribution verwendet,

<sup>8</sup><https://alpinelinux.org/>

<sup>9</sup>[https://hub.docker.com/\\_/openjdk/](https://hub.docker.com/_/openjdk/)

danach werden `make` und `gcc` installiert, `make` wird getestet und zum Abschluss wird wieder zusammengeräumt, indem `make` und `gcc` wieder deinstalliert werden.

Quelltext 4.8 zeigt allerdings, dass der Aufbau des Dockerfiles alles andere als optimal ist sowie die Vorteile der Schreibweise in Quelltext 4.7. Der dritte `RUN`-Befehl, der sich eigentlich um das Aufräumen kümmern sollte, verursacht noch zusätzlichen Speicherverbrauch des Containers. Dieses Problem ist auf das von Docker verwendete Union-Dateisystem (siehe Abschnitt 4.5) zurückzuführen. Eine zusätzliche Schicht kann Dateien nur als gelöscht markieren, diese existieren allerdings in den unteren Schichten immer noch, wodurch das Image weiterwächst, anstatt kleiner zu werden.

Aus diesem Grund sollte das Aufräumen in einem Container immer im selben `RUN`-Befehl erfolgen. Dieser lässt sich wie in Quelltext 4.7 in mehrere Zeilen aufteilen. Der Nachteil dieser Methode ist, dass kein Build-Caching mehr durchgeführt werden kann, denn sobald sich ein Teil des Kommandos ändert, wird die gesamte Schicht neu erstellt. Der resultierende Unterschied in der Imagegröße rechtfertigt allerdings die etwas längere Erstellungszeit des Images.

```
1 FROM alpine:3.5
2 RUN apk update && apk add make gcc
3 RUN make --help
4 RUN apk del make gcc
```

Quelltext 4.6: Dockerfile mit suboptimalem `RUN`-Befehl

```
1 FROM alpine:3.5
2 RUN apk update && apk add make gcc && \
3     make --help && \
4     apk del make gcc
```

Quelltext 4.7: Dockerfile mit korrigiertem `RUN`-Befehl

```
1 # Building both images
2 $ docker build -t image-large -f .\Dockerfile.image-large .
3 $ docker build -t image-small -f .\Dockerfile.image-small .
4
5 $ docker images
6 REPOSITORY          TAG          IMAGE ID      SIZE
7 alpine              latest      88e169ea8f46  3.98 MB
8 image-large         latest      56038a6ecc87  91.6 MB
9 image-small         latest      114ccaf13bde  5.02 MB
10
11 $ docker history image-large
12 IMAGE              CREATED BY          SIZE
13 56038a6ecc87       /bin/sh -c apk del make gcc  17.3 kB
14 ffe18f5b76d9       /bin/sh -c make --help      0 B
```



```

15 188b3de5ec5d      /bin/sh -c apk update && apk add...      87.6 MB
16 88e169ea8f46      /bin/sh -c (nop) ADD file:92ab74...      3.98 MB
17
18 $ docker history image-small
19 IMAGE              CREATED BY              SIZE
20 114ccaf13bde        /bin/sh -c apk update && apk add...      1.04 MB
21 88e169ea8f46        /bin/sh -c (nop) ADD file:92ab7...      3.98 MB

```

Quelltext 4.8: Vergleich der Imagegröße nach Optimierung des RUN-Befehls

### 4.6.3 Festlegen des Start-Kommandos für den Container

Ein wichtiger Aspekt beim Entwickeln von Docker-Images ist das Start-Kommando. Welche Anwendung der Container ausführt beziehungsweise wie der Benutzer diese konfigurieren kann, wird über die Befehle `ENTRYPOINT` und `CMD` im Dockerfile bestimmt. Ein besonders interessanter Aspekt ist die Kombination der beiden Befehle. Die folgenden Beispiele und Informationen stammen aus [DeH15].

Um dem Benutzer eine möglichst einfache Verwendung des Containers zu ermöglichen, lässt sich mit `ENTRYPOINT` das auszuführende Programm festlegen, mit `CMD` lassen sich Standardparameter an dieses übergeben.

```

1 FROM ubuntu:trusty
2 ENTRYPOINT ["/bin/ping","-c","3"]
3 CMD ["localhost"]

```

Quelltext 4.9: Dockerfile für `ping` mit `ENTRYPOINT` und `CMD`

In Quelltext 4.9 ist ein Beispielcontainer für das Ping-Kommando dargestellt, der die beiden Befehle kombiniert. Als Basis-Image kommt Ubuntu zum Einsatz, danach wird mit `ENTRYPOINT` „ping“ als auszuführendes Programm konfiguriert und mithilfe von `CMD` „localhost“ als Standardwert an `ping` übergeben.

In Quelltext 4.10 ist die Verwendung des Containers dargestellt. Da das Docker `run`-Kommando als letzten Parameter optional den `CMD`-Befehl eines Containers überschreibt, wird nun beim Starten des Containers ohne Parameter der Standardwert *localhost* verwendet. Wenn der Benutzer allerdings das Kommando überschreibt, wird dieses an den `ping`-Befehl übergeben und von diesem als Parameter verwendet. Diese Maßnahme steigert die Benutzerfreundlichkeit von Containern und ermöglicht ein sehr intuitives Arbeiten, da der Container auch ohne Konfiguration ein sinnvolles Standardverhalten aufweist.

```

1 $ docker run ping
2 PING localhost (127.0.0.1) 56(84) bytes of data.
3 ...
4
5 $ docker run ping docker.io

```

```
6 PING docker.io (162.242.195.84) 56(84) bytes of data.  
7 ...
```

Quelltext 4.10: Verwendung des `ping`-Containers

## 4.7 Häufig benötigte Docker-Kommandos

Für das produktive Arbeiten mit Docker genügen lediglich ein paar Kommandos, von denen die wichtigsten in diesem Abschnitt vorgestellt werden.

### 4.7.1 Management-Kommandos

Mit der Weiterentwicklung von Docker beinhaltet die Kommandozeilenschnittstelle mittlerweile über 40 Kommandos. Mit der Version 1.13 wurde die CLI neu strukturiert. In [Doc17a] sind die neuen Management-Kommandos beschrieben. Bei einem Aufruf von Docker ohne Parameter wird eine Übersicht über die Kommandos geliefert. Diese Kommandos sind seit Version 1.13 ihren jeweiligen Komponenten im Docker-System zugewiesen und über diese erreichbar. Die neue Version für das Löschen eines Images schreibt sich nun `docker image rm <image>` und nicht mehr `docker rmi <image>`. Diese Benennung ist in der gesamten Anwendung einheitlich, wobei aus Kompatibilitätsgründen die alten Kommandos weiterhin existieren.

### 4.7.2 Images erstellen

Das Erstellen eines Images bei vorhandenem Dockerfile wird folgendermaßen durchgeführt:

```
docker build [-t <tag>] [-f <dockerfile>] <context>
```

Mit dem Parameter `-t` wird der Name angegeben und mit `-f` kann ein anderer Dockerfile angegeben werden. Standardmäßig wird die Datei namens *Dockerfile* aus dem Kontext-Verzeichnis verwendet. Alle Dateien in diesem Verzeichnis können im Dockerfile verwendet werden, da dies die Umgebung darstellt, in der das Image erstellt wird.

In Quelltext 4.11 ist ein Beispielkommando für das Erstellen des Dockerfiles aus Quelltext 4.2 angegeben. Das erstellte Image erhält den Namen *kstaken/apache*.

```
1 docker build -t kstaken/apache .
```

Quelltext 4.11: Docker-Build-Beispiel

### 4.7.3 Container starten

Die häufigste Aufgabe eines Entwicklers bei der Arbeit mit Docker ist das Starten von Containern. Die wichtigsten Parameter für dieses Kommando sehen wie folgt aus:

```
docker run [--rm] [-it] [-v <volumes>] [-p <ports>] <image> [<command>]
```

Der Parameter `-rm` gibt an, dass der Container gelöscht wird, nachdem er gestoppt wurde. Dadurch wird die Grundidee der flüchtigen Docker-Container unterstützt und das System sauber gehalten. Mit den Parametern `-i` und `-t` wird eine interaktive Konsole mit dem Container verbunden, die zum Beispiel bei der Verwendung einer Shell im Container Einsatz findet. `-v` ermöglicht unter anderem das Einbinden von Verzeichnissen des Hosts in den Container. Dieser Anwendungsfall ist gerade für Entwickler sehr nützlich, da so Quelltext im Container verwendet werden kann. Am Host durchgeführte Änderungen am Quelltext werden auch an den Container weitergeleitet. Vom Container zur Verfügung gestellte Netzwerk-Ports können mit `-p` an den Host gebunden werden, um von dort aus mit *localhost* auf den Container zuzugreifen. Bei beiden Parametern ist die Reihenfolge der angegebenen Werte `<host:container>`. In Quelltext 4.12 ist ein Beispielkommando für das Starten eines Containers auf Basis des in Quelltext 4.11 erstellten Images angegeben.

```
1 docker run -d --rm -v ${pwd}:/var/www -p 8080:80 kstaken/apache
```

Quelltext 4.12: Docker-Run-Beispiel

#### 4.7.4 Docker aufräumen

Im Laufe der Verwendung von Docker fallen gestoppte Container, nicht verwendete Images und übriggebliebene Speicher-Volumes an. Seit Version 1.13 lassen sich diese mit Bordmitteln von Docker sehr einfach aufräumen:

```
docker system df                # Show docker disk usage
docker container|volume|image|system prune  # Clean Resources
```

In Quelltext 4.13 ist das Ergebnis dieser Kommandos dargestellt, nachdem ein gestopp-ter Docker-Container im System vorhanden war.

```
1 $ docker system df
2 TYPE                TOTAL      ACTIVE    SIZE      RECLAIMABLE
3 Images              8          1        523.8 MB   477.1 MB (91%)
4 Containers          1          0         484 B     484 B (100%)
5 Local Volumes       0          0          0 B       0 B
6
7 $ docker system prune
8 WARNING! This will remove:
9     - all stopped containers
10    - all volumes not used by at least one container
11    - all networks not used by at least one container
12    - all dangling images
13 Are you sure you want to continue? [y/N] y
14 Deleted Containers: 7c3387e2f9af442382d140c671619c548f5fd
15 Total reclaimed space: 484 B
```

Quelltext 4.13: Docker-Prune-Beispiel

## 5 Docker-Anwendungsszenarien für den Softwareentwickler

Dieses Kapitel zeigt exemplarisch mögliche Anwendungsfälle für die Verwendung von Docker als Werkzeug für Softwareentwickler und wie diese bei ihren alltäglichen Tätigkeiten unterstützt werden können.

Das Entwickeln von Software erfordert die Installation zahlreicher Werkzeuge. Wenn sich diese auf lediglich ein Projekt beschränkt, stellt dies kein Problem dar, da es zu keinen Versionsinkompatibilitäten kommt und die Werkzeuge nicht laufend gewechselt werden müssen. Werden allerdings mehrere Projekte parallel entwickelt und zusätzlich noch Beiträge zu Open-Source-Projekten geleistet, wird dieses Unterfangen erheblich komplizierter. Unterschiedliche Laufzeitumgebungen, Compiler-Versionen, Build-Werkzeuge und Frameworks führen sehr schnell zu einem unüberschaubaren und instabilen System [Dem16].

In [Dem16] werden bestehende Lösungen für die auftretenden Probleme aufgelistet:

- Anstatt Abhängigkeiten auf Systemebene zu installieren, werden diese von *Paketmanagern* auf Projektebene verwaltet. Vertreter davon sind zum Beispiel npm<sup>1</sup> für JavaScript, Maven<sup>2</sup> für Java oder NuGet<sup>3</sup> für .NET.
- Zusätzlich zu Paketen müssen allerdings auch die Laufzeitumgebungen verwaltet werden. Bei unterschiedlichen Projekten können unterschiedliche Versionen dieser benötigt werden, wodurch Werkzeuge wie der Node Version Manager (nvm<sup>4</sup>) oder der Ruby Version Manager (RVM<sup>5</sup>) entstanden sind. Diese müssen allerdings auf dem Entwicklerrechner verwaltet werden und erschweren den schnellen Wechsel zwischen unterschiedlichen Projekten. Bei alten Legacy-Systemen ist eine Unterstützung durch diese Werkzeuge außerdem nicht garantiert.
- Mit dem Aufschwung der Virtualisierung und Werkzeugen wie Vagrant (siehe Abschnitt 3.2.1) wurden viele dieser Probleme gelöst. Allerdings ist die Notwendigkeit einer virtuellen Maschine pro Projekt sehr ressourcen- und wartungsintensiv. Ein weiterer Nachteil ist, dass Vagrant auf Entwicklerrechner ausgelegt ist und dadurch nicht garantiert werden kann, dass die entwickelte Software auch auf dem Produktionssystem läuft [Lar16].

---

<sup>1</sup><https://www.npmjs.com/>

<sup>2</sup><https://maven.apache.org/>

<sup>3</sup><https://www.nuget.org/>

<sup>4</sup><https://github.com/creationix/nvm>

<sup>5</sup><https://rvm.io/>

Docker bietet eine Abstraktion des Betriebssystems und die Möglichkeit, die benötigten Komponenten einer Anwendung maschinenlesbar zu beschreiben. Dadurch ist es möglich, die Laufzeitumgebung und die benötigten Komponenten der Anwendung gemeinsam mit dieser im Versionsverwaltungssystem aufzubewahren. Danach ist lediglich Docker die Voraussetzung dafür, dass mit einem plattformübergreifenden Kommando mit der Entwicklung von Software begonnen werden kann.

Travis Reeder liefert in seinem Blogeintrag über die Entwicklung mit Docker ([Ree15]) folgende Gründe für die Verwendung von Docker für Softwareentwickler:

- Die Entwicklungsumgebung ist nicht vom Betriebssystem oder dessen Version abhängig. Unterschiedliche Personen im Team können problemlos auf Windows, macOS und Linux gemeinsam an einem Produkt arbeiten. Dieser Aspekt ist besonders bei der Entwicklung von Open-Source-Projekten wichtig, da dort unzählige Entwickler ohne zentrale Verwaltung zu einem Produkt beitragen.
- Die Entwicklungs- und Produktionsumgebung werden vereinheitlicht. Quelltext, der in der Entwicklungsumgebung getestet ist, funktioniert automatisch auch auf dem Produktivsystem.
- Komplizierte Build-Prozesse können abstrahiert werden, sodass die Entwickler diesen nicht verstehen müssen. Außerdem kann dieser Prozess separat entwickelt und getestet oder sogar wiederverwendet werden.
- Unterschiedliche Laufzeitumgebungen haben keinen Einfluss auf das Hostsystem des Entwicklers. Besonders die Entwicklung von Java, Ruby, Python und Node wird dadurch stark vereinfacht. Zusätzlich ist es beispielsweise einfach möglich, die Software mit einer anderen Compiler-Version zu übersetzen, um zu überprüfen, ob diese kompatibel ist.
- Das Veröffentlichen der Software wird einfacher. Es muss lediglich ein Container der Anwendung erstellt werden, der nun auf jedem Docker-Host lauffähig ist.
- Es müssen keine besonderen Entwicklungsumgebungen verwendet oder Kommandos über SSH auf einer virtuellen Maschine ausgeführt werden. Der Container wird in das Host-System integriert und ist beinahe nicht sichtbar.
- Entwicklern wird ein schnellerer und einfacher Einstieg in ein Projekt ermöglicht, da große Teile des Entwicklungsprozesses abstrahiert werden. Diese muss der Entwickler anfangs nicht verstehen, wodurch die Lernkurve und dadurch auch die Einarbeitungszeit sinken.

## 5.1 Softwareevaluierung

Die Evaluierung und Auswahl von Werkzeugen, Frameworks und Komponenten eines Softwaresystems ist eine wichtige Aufgabe des Softwareentwicklers. Ein sehr mühsamer Aspekt dieser Tätigkeit ist das Einarbeiten und Verstehen der Installation und Konfiguration

der zu evaluierenden Werkzeuge. Außerdem werden oftmals unnatürliche Standardpfade (beispielsweise `C:\xampp` bei der Installation der PHP-Umgebung xampp<sup>6</sup>) gewählt, oder die Deinstallation von getesteten Werkzeugen hinterlässt Spuren am Host-System des Softwareentwicklers. Ein weiteres Problem können fehlende Benutzerrechte darstellen, wodurch die Evaluierung bestimmter Werkzeuge gar nicht möglich ist.

### Beispiel: ArangoDB

Docker ermöglicht ein sehr schnelles Testen von Datenbanken, Anwendungsservern und anderen Werkzeugen. Außerdem wird beim Löschen eines Containers das System automatisch wieder auf einen „sauberen“ Ausgangszustand zurückgesetzt. Diese Möglichkeit kann sogar zu Marketingzwecken verwendet werden, wie auf der Homepage von ArangoDB<sup>7</sup> in Abb. 5.1 dargestellt ist.

Das dort verwendete Kommando ist aus Marketinggründen allerdings etwas zu einfach gehalten. Würde der Container auf diese Weise gestartet werden, wäre der Netzwerk-Port auf dem Host-System nicht sichtbar und es kann kein Zugriff auf die Datenbank stattfinden. In Quelltext 5.1 ist das komplette Startkommando dargestellt. Abb. 5.2 zeigt die Administrationsoberfläche der gestarteten Datenbank.

Sobald der Container gestoppt wird, wird er durch `-rm` wieder gelöscht und es verbleiben keine Reste aus der Evaluierungsphase auf dem Host-System. Ein zusätzlicher Vorteil der Verwendung von Docker ist, dass genau derselbe Container auch im Produktivsystem verwendet werden kann. Dies ermöglicht beispielsweise die rasche Weiterentwicklung von Prototypen.

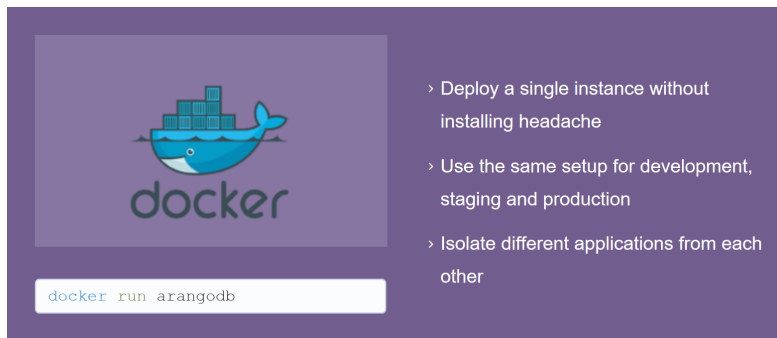


Abbildung 5.1: Docker als Marketinginstrument bei ArangoDB

```
1 docker run --rm -d -e ARANGO_NO_AUTH=1 -p 8529:8529 arangodb
```

Quelltext 5.1: Docker-Kommando zum Starten von ArangoDB

---

<sup>6</sup><https://www.apachefriends.org/>

<sup>7</sup><https://www.arangodb.com/>

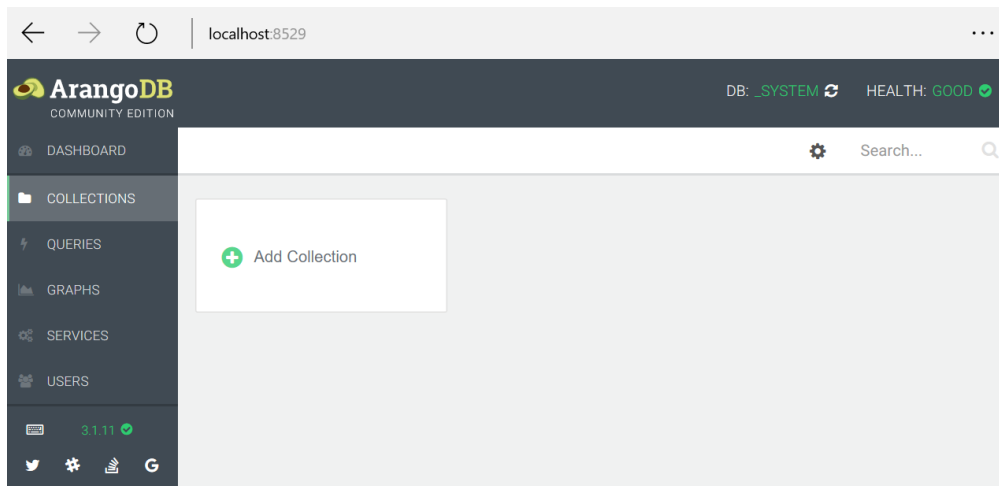


Abbildung 5.2: ArangoDB als Docker-Container

## 5.2 Plattformunabhängige CLI-Anwendungen

In [Eng15] beschreibt Mike English die Möglichkeit, Docker als Laufzeitumgebung für Kommandozeilenwerkzeuge zu verwenden. Durch die Kombination des Werkzeuges mit seinen Abhängigkeiten werden bei der Verwendung von Docker keine Laufzeitsysteme auf dem Host benötigt. Dies führt besonders bei Python und Ruby zur Vermeidung von Versionsproblemen und der einfacheren Verwendung auf Windows. Außerdem bietet Docker mit dem Docker Hub eine sehr angenehme Möglichkeit für die Veröffentlichung und Verbreitung. Dem gegenüber stehen die Nachteile der Notwendigkeit der Installation von Docker und die damit verbundene Imagegröße aufgrund der Docker-Basisimages.

### Beispiel: Jekyll

Jekyll<sup>8</sup> ist ein in Ruby geschriebener Generator für statische Webseiten. In der offiziellen Dokumentation ist beschrieben, dass Windows keine offiziell unterstützte Plattform ist, es aber trotzdem folgende Möglichkeit zur Installation gibt [Jek17]:

1. Installieren eines Paketmanagers wie beispielsweise Chocolatey<sup>9</sup>.
2. Installieren von Ruby mit Chocolatey: `choco install ruby -y`.
3. Installieren von Jekyll: `gem install jekyll`.

Für die Installation eines Werkzeuges werden also zwei weitere Werkzeugen benötigt, wobei die Plattform weiterhin als „nicht offiziell unterstützt“ gilt.

<sup>8</sup><https://jekyllrb.com/>

<sup>9</sup><https://chocolatey.org/>

Allerdings bietet Jekyll auch ein vorgefertigtes offizielles Docker-Image<sup>10</sup> an, welches mit dem in Quelltext 5.2 beschriebenen Kommando Jekyll im aktuellen Ordner startet. Danach ist die Webseite unter `http://localhost:4000` erreichbar.

Die Verwendung von Jekyll kann auf diese Weise anstatt der dreistufigen Installation auf das Starten eines Containers reduziert werden.

```
1 docker run -it --rm -v ${pwd}:/srv/jekyll -p 4000:4000 jekyll/
   ↪ jekyll /usr/local/bin/jekyll serve
```

Quelltext 5.2: Docker-Kommando zum Starten von Jekyll

### Beispiel: mdp

Im vorigen Beispiel (Jekyll) diente ein bereits existierendes Docker-Image als Basis für ein plattformunabhängiges Werkzeug. Da allerdings nicht für jedes Werkzeug ein Docker-Image existiert, ist in Quelltext 5.3 ein Dockerfile abgebildet, der die Portierung eines existierenden Werkzeuges für Docker zeigt.

```
1 FROM alpine:3.5
2 MAINTAINER Bernhard Mayr <bernhard@mayr.io>
3 ENV MDP_VERSION 1.0.9
4
5 RUN apk update && apk add make gcc musl-dev ncurses-dev ca-
   ↪ certificates wget && \
6     update-ca-certificates && mkdir -p /mdp && \
7     wget -P /mdp https://github.com/visit1985/mdp/archive/
   ↪ $MDP_VERSION.tar.gz && \
8     tar -xzf /mdp/$MDP_VERSION.tar.gz -C /mdp && \
9     make -C /mdp/mdp-$MDP_VERSION && make -C /mdp/mdp-
   ↪ $MDP_VERSION install && \
10    cp /mdp/mdp-$MDP_VERSION/sample.md /demo.md && \
11    printf "#!/bin/sh\nsleep 0.1\n\${*}\n" > /start.sh && chmod +
   ↪ x /start.sh && \
12    rm -rf /mdp && apk del make && apk del gcc
13
14 WORKDIR /data
15 ENTRYPOINT ["/start.sh"]
16 CMD ["mdp", "/demo.md"]
```

Quelltext 5.3: Docker-Image für mdp

In Quelltext 5.3 wird ein Docker-Image für mdp<sup>11</sup> erstellt. Dieses Verfahren kann analog auf jedes Linux-Kommandozeilenwerkzeug angewendet werden.

<sup>10</sup><https://hub.docker.com/r/jekyll/jekyll/>

<sup>11</sup><https://github.com/visit1985/mdp>



mdp ist ein C-Programm, das aus Markdown-Dateien Folien für die Kommandozeile erstellt. Im Dockerfile werden, wie in der offiziellen Dokumentation beschrieben, die Übersetzung und Installation mit **make** durchgeführt. Das Startkommando ist auf eine möglichst intuitive Verwendung ausgelegt (vgl. Abschnitt 4.6.3). Wird keine Präsentationsdatei übergeben, startet der Container mit einer Demo-Präsentation, die die Verwendung von mdp erklärt.

### Aufruf aus der Powershell

Die Abstraktion von Kommandozeilenanwendungen mit Docker bietet, wie in den beiden Beispielen gesehen, zahlreiche Vorteile. Doch ein komplettes Docker-Run-Kommando zum Starten einer Anwendung ist nicht nur sehr umständlich und fehleranfällig, sondern verlangt auch bei jedem Start die Konfiguration zahlreicher Parameter.

Dieses Problem lässt sich unter der Verwendung von Shell-Aliasen oder Powershell-Funktionen lösen. In Quelltext 5.4 sind die zwei Funktionen zum vereinfachten Starten von Jekyll und mdp aufgelistet. `-v ${pwd}:<container-path>` ermöglicht dem Container das Arbeiten mit dem aktuellen Ordner, indem es diesen an den Container bindet. Mithilfe von `${args}` werden die Parameter des Funktionsaufrufes an das Docker-Run-Kommando weitergeleitet.

Der Aufruf der Werkzeuge erfolgt nun wie gewohnt, wobei die Ausführung in einem Docker-Container für den Anwender unsichtbar ist.

```
1 $ jekyll
2 function jekyll { docker run -it --rm -v ${pwd}:/srv/jekyll -p
   ↪ 4000:4000 jekyll/jekyll /usr/local/bin/jekyll ${args} }
3 function mdp { docker run -it --rm -v ${pwd}:/data bemayr/mdp
   ↪ mdp ${args} }
```

Quelltext 5.4: Powershell-Funktionen für Docker-Kommandos

Ein besonders angenehmer Vorteil bei dieser Shell-Integration ist in Quelltext 5.5 dargestellt. Das Docker-Run-Kommando lädt nicht vorhandene Images vor dem Starten automatisch herunter, wodurch in diesem Fall eine vollautomatische Installation der benötigten Werkzeuge entsteht. Beim Wechsel oder der Neuinstallation des Host-Systems genügt die Mitnahme der Shell-Konfiguration um das gesamte Werkzeugset auch auf dem neuen System verwenden zu können. Die Verwaltung der benötigten Werkzeuge erfolgt somit auf einfachste Art und Weise.

```
1 $ jekyll
2 Unable to find image 'jekyll/jekyll:latest' locally
3 latest: Pulling from jekyll/jekyll
4 baf40e071063: Pull complete
5 70acac711d95: Downloading [=====>] 11.33 MB/91.37 MB
```

Quelltext 5.5: Automatische Installation der Docker-basierten CLI-Anwendungen

## 5.3 Containerbasierte Integrationstests

Der große Vorteil bei der Verwendung von Containern zum Testen ist ihre Flüchtigkeit. Durch einen Neustart kann der Ursprungszustand schnell und einfach wiederhergestellt werden. Unit-Tests profitieren davon nicht, doch die Verwendung von Containern verschafft ihnen eine reproduzierbare und einheitliche Ausführungsumgebung. Dadurch werden Versionsinkompatibilitäten vermieden und Abhängigkeiten reduziert.

Einen wesentlich größeren Einfluss haben flüchtige Container auf die Ausführung von Integrationstests. Diese haben aufgrund der zu testenden Komponenten sehr viele Abhängigkeiten wie externe Services, Werkzeuge zum Validieren der Tests (CURL) oder Datenbanken und deren Testdaten. Um konsistente Testdaten beizubehalten, müssen die Datenbanken vor jedem Testlauf mit diesen neu initialisiert werden. In containerbasierten Systemen werden die Testdaten automatisch nach jedem Testlauf verworfen, wodurch keine Probleme durch alte Daten auftreten können. Weiters können externe Services sehr einfach durch zusätzliche Container in das System integriert werden.

Das folgende Beispiel stammt aus [Har16], einem Blogeintrag zum Aufbau eines containerbasierten Integrationstestsystems. Die Architektur dieses Systems ist in Abb. 5.3 dargestellt.

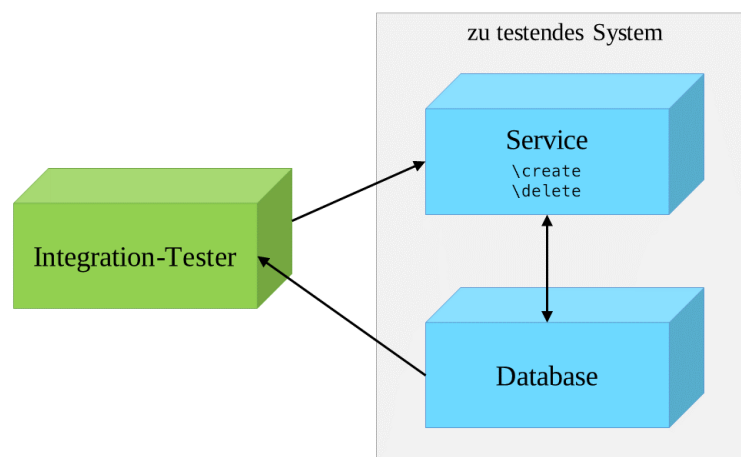


Abbildung 5.3: Architektur des containerbasierten Integrationstestsystems

Das zu testende System wird von einem weiteren Container, dem „Integration-Tester“ gesteuert. Dieser führt die Testfälle am Service durch und überprüft deren Korrektheit mithilfe von Datenbankabfragen. Dadurch kann das gesamte System getestet werden, ohne dass Änderungen an diesem oder dessen Architektur durchgeführt werden müssen. Dazu muss der Integration-Tester Zugriff auf das Service und die Datenbank haben. Für den Aufbau dieser Architektur kommt Docker Compose (siehe Abschnitt 4.4) zum Einsatz. Die Definition der Container ist in Quelltext 5.6 dargestellt.

1 version: '2'

```

2
3 services:
4   integration-tester:
5     build: ./integration-tester
6     links:
7       - my-service
8   my-service:
9     build: ./my-service
10    command: npm start
11    links:
12      - rethinkdb
13    ports:
14      - "8080:8080"
15  rethinkdb:
16    image: rethinkdb
17    expose:
18      - "28015"

```

Quelltext 5.6: Servicedefinition zum Integrationstesten (docker-compose.yml)

Docker-Compose-Dateien definieren im YAML-Format die Parameter, die Docker zum Starten der Container erhält. In diesem Beispiel werden die in Abb. 5.3 definierten Container erstellt und über die Konfigurationsoption `links` miteinander verbunden. Der Start der Integrationstests mit `docker-compose up` führt zu folgendem Ablauf:

1. Die Images *my-service* und *integration-tester* werden erstellt.
2. Die Container *my-service*, *rethinkdb* und *integration-tester* werden gestartet und miteinander verknüpft.
3. Der Integration-Tester führt die Testfälle aus.
4. Nach der Fertigstellung der Integrationstests werden die Container gestoppt.

## 5.4 Plattformübergreifende Übersetzung

Plattformübergreifende Übersetzung ist durch die Idee aus Abschnitt 5.2 möglich. Durch die Möglichkeit, Linux-Anwendungen in Docker-Containern zu starten, können diese nahtlos in Windows eingebunden werden.

### Beispiel: gcc

In Quelltext 5.7 ist ein übliches „Hello-World“-Programm in der Programmiersprache C abgebildet. Um dieses nun auf einem Windows-Host auch unter Linux zu testen, genügt der in Quelltext 5.8 dargestellte Befehl. Darin wird ein gcc<sup>12</sup>-Container auf Basis des

---

<sup>12</sup><https://gcc.gnu.org/>

Alpine-Images zur Übersetzung des C-Programmes verwendet. Mit `-v ${pwd}:/usr/src` wird dem Container das aktuelle Verzeichnis des Host-Systems unter `/usr/src` zur Verfügung gestellt. `-w /usr/src` legt das Arbeitsverzeichnis des Containers fest, sodass das `gcc`-Kommando in diesem Verzeichnis ausgeführt wird.

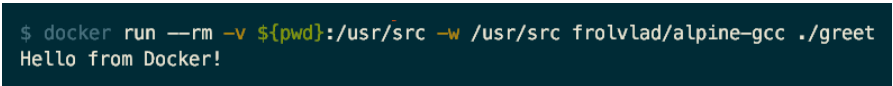
In Abb. 5.4 ist das Resultat des Programmes zu sehen, nachdem es in Linux ausgeführt wurde. Diese Möglichkeit des Cross-Compilings ermöglicht ein schnelles Testen der Plattformunabhängigkeit eines Quelltextes.

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello from Docker!");
5     return 0;
6 }
```

Quelltext 5.7: „Hello-World“-C-Programm

```
1 docker run --rm -v ${pwd}:/usr/src -w /usr/src frolovlad/alpine-
  ↪ gcc gcc -o greet greet.c
```

Quelltext 5.8: Docker-Kommando zum Übersetzen eines C-Programmes mit gcc



```
$ docker run --rm -v ${pwd}:/usr/src -w /usr/src frolovlad/alpine-gcc ./greet
Hello from Docker!
```

Abbildung 5.4: Hello-World-Ausgabe unter Linux

### Beispiel: dockcross

Für die plattformübergreifende Übersetzung existieren bereits zahlreiche fertige Docker-Images wie beispielsweise `dockcross`<sup>13</sup>.

In Quelltext 5.9 sind die notwendigen Kommandos dargestellt, die zum Erstellen des Hello-World-Programms aus Quelltext 5.7 für die `armv7`-Plattform notwendig sind. In `dockcross` werden die Container nicht händisch, sondern von einem aus dem Container erzeugbaren Skript ausgeführt. Dieses Skript wird in Zeile 1 aus dem Container erzeugt, in Zeile 2 werden die notwendigen Berechtigungen gesetzt.

Das Problem dieser Lösung unter Windows ist die fehlende Interpretation der Bash-Kommandos, weshalb die vorgestellten Kommandos unter Docker for Windows in der Powershell nicht ausgeführt werden können. Für die Ausführung dieses Skriptes unter Windows wird ein Werkzeug wie Cygwin<sup>14</sup>, `win-bash`<sup>15</sup> oder Bash on Ubuntu on

<sup>13</sup><https://github.com/dockcross/dockcross>

<sup>14</sup><https://www.cygwin.com/>

<sup>15</sup><http://win-bash.sourceforge.net/>

Windows<sup>16</sup> benötigt.

```
1 docker run --rm dockcross/linux-armv7 > ./dockcross-linux-armv7
2 chmod +x ./dockcross-linux-armv7
3 ./dockcross-linux-armv7 bash -c '$CC test/C/greet.c -o
   ↪ greet_arm'
```

Quelltext 5.9: Kommandos zum Übersetzen mit dockcross

## 5.5 IDE in a Container

Der Google-Entwickler Ray Tsang hat die Idee der Entwicklungsumgebung in einem Container<sup>17</sup> erstmals 2015 vorgestellt [Tsa15]. Der Einsatzzweck seines Containers liegt in der Go-Entwicklung. Seine Voraussetzungen waren Syntax-Highlighting, Auto-Completion sowie der Zugriff auf die Informationen der Framework-Dokumentation. Ray Tsang stellt klar, dass seine Idee lediglich ein Experiment ist, definiert jedoch folgende Vorteile einer containerbasierten Entwicklungsumgebung:

1. Die Installation der Entwicklungsumgebung dokumentiert sich selbst. Jede Konfiguration wird im Dockerfile festgehalten.
2. Abhängigkeiten müssen nicht am Host-System, sondern können im Container installiert werden.
3. Die Entwicklungsumgebung ist portabel. `docker run ...` genügt zum Starten der IDE auf jedem beliebigen System mit installiertem Docker.
4. Jeder Entwickler kann seine Anpassungen selbst vornehmen, indem er die deklarative Beschreibung (Dockerfile) des Images ändert.

Ein weiterer Vorteil ist die Möglichkeit der Versionierung der IDE gemeinsam mit dem Projekt. Dadurch wird Entwicklern ein sehr einfacher Einstieg in ein Projekt ermöglicht. Außerdem muss für Patches und Fehlerbehebungen in Legacy-Systemen keine eigene alte IDE-Version auf dem Entwicklerrechner behalten werden.

### Beispiel: vim (jare/vim-bundle)

Eugene Yaremenko<sup>18</sup> bietet mit seinen IDE-Containern *jare/vim-bundle*, *jare/drop-in* und *jare/spacemacs* vorkonfigurierte Entwicklungsumgebungen in Docker an. Abb. 5.5 zeigt vim<sup>19</sup> in Docker, das mit dem in Quelltext 5.10 dargestellten Kommando gestartet wird. Für Entwickler, die sich auf der Kommandozeile wohlfühlen, bietet diese Möglichkeit eine sehr portable Entwicklungsumgebung.

<sup>16</sup><https://msdn.microsoft.com/en-us/commandline/wsl/about>

<sup>17</sup><https://github.com/saturnism/go-ide>

<sup>18</sup><https://github.com/JAremko>

<sup>19</sup><http://www.vim.org/>



Abbildung 5.5: vim als Docker-Container

- 1 \$ docker run -it --rm -v \${**pwd**}/home/developer/workspace jare/  
↪ vim-bundle

Quelltext 5.10: Docker-Kommando zum Starten von vim

### Beispiel: Eclipse Che

Einen fundamental anderen Ansatz verfolgt die neue Cloud-IDE Eclipse Che<sup>20</sup>. Deren Grundidee ist die Entwicklung von Software ohne der Installation von Werkzeugen. Die IDE selbst läuft als Webanwendung und wird über den Browser bedient. Das gesamte Plug-in-System baut auf Docker-Containern auf, die lediglich kleine Teilaufgaben der Entwicklungsumgebung übernehmen.

In Quelltext 5.11 ist der Startvorgang von Eclipse Che zu sehen. Dem Container wird der Docker-Socket des Host-Systems übergeben, sodass dieser selbst Container starten und verwalten kann. Die gestartete IDE ist in Abb. 5.6 zu sehen.

- 1 \$ docker run --rm -v /var/run/docker.sock\\:/var/run/docker.  
↪ sock -v \${**pwd**}/data eclipse/che start
- 2 WARN: Bound 'eclipse/che' to 'eclipse/che:5.3.1'
- 3 INFO: (che cli): 5.3.1 – using docker 1.13.1 / docker4windows
- 4 INFO: (che start): Booted and reachable
- 5 INFO: (che start): Ver: 5.3.1
- 6 INFO: (che start): Use: http://10.0.75.2:8080

<sup>20</sup><http://www.eclipse.org/che/>

- 7 INFO: (che start): API: <http://10.0.75.2:8080/swagger>  
Quelltext 5.11: Docker-Kommando zum Starten von Eclipse Che

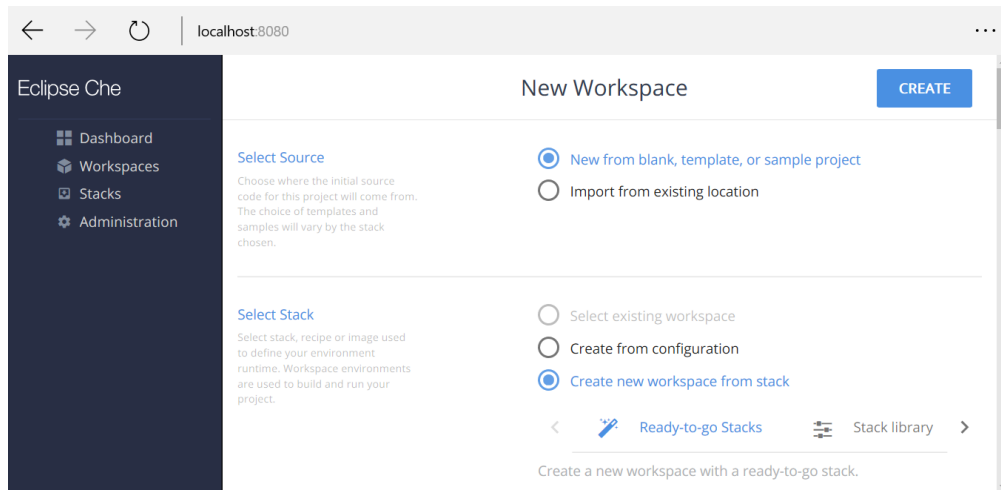


Abbildung 5.6: Eclipse Che als Docker-Container

## 6 Resümee

Um die Möglichkeiten der Containervirtualisierung anhand von Docker zu analysieren, wurde die Geschichte und Funktionsweise dieser in der Arbeit mit dem Konzept der Vollvirtualisierung verglichen. Schwierigkeiten und Probleme im Entwicklungsprozess wurden durch verschiedene Werkzeuge und Lösungsansätze des Konfigurationsmanagements aufgezeigt. Nach einer Einführung in Docker wurden Best-Practices und deren Anwendung gezeigt. Dieses Wissen wurde anschließend verwendet, um verschiedene Anwendungsmöglichkeiten von Docker für die Lösung häufiger Aufgaben eines Softwareentwicklers aufzuzeigen und deren Vorteile zu evaluieren. Im Folgenden wird die Arbeit zusammengefasst und auf Basis der erlangten Erkenntnisse eine Prognose für den zukünftigen Einsatz von Containertechnologien verwendet.

### 6.1 Zusammenfassung

Docker hat es geschafft, Containervirtualisierung soweit zu vereinfachen, dass sie sowohl zum Betrieb von hochperformanten Anwendungen in Cloud-Clustern als auch zur Unterstützung im Entwicklungsprozess verwendet werden kann. Dies wird erst durch die erhebliche Verbesserung der Virtualisierung mittels Container im Vergleich zu traditionellen Virtualisierungsansätzen ermöglicht.

Die Verwendung von Konfigurationsmanagement-Werkzeugen war auch vor Docker bereits die Grundlage für einen stabilen und skalierbaren Entwicklungsprozess. Docker ermöglicht allerdings eine wesentlich elegantere und leichtgewichtigere Lösung vieler dieser Probleme.

Um saubere Docker-Images zu erstellen, ist ein fundiertes Wissen über Docker notwendig, das aufgrund der sehr schnellen Entwicklung ständig aktualisiert werden muss. Durch den großen Anteil an Open-Source-Beispielen und – oft in Blogs zeitnah dokumentierten – Ideen für die Anwendung von Docker entstehen allerdings sehr zeit- und ressourcensparende Problemlösungen in vielen Teilen des Softwareentwicklungsprozesses.

### 6.2 Erkenntnisse

Viele der vorgestellten Probleme können auch mit bestehenden Werkzeugen, wie z. B. Vagrant und Chef, gelöst werden, Docker bietet jedoch eine deutlich modernere und leichtgewichtigere Lösung.

Ohne ein Grundverständnis von Docker ist die Anwendung dieser Technologie für Softwareentwickler eine Herausforderung. Nur durch ein tieferes Verständnis können Container so erstellt und abstrahiert werden, dass dem Anwender kaum auffällt, dass er



eigentlich mit Docker arbeitet. Dadurch ist Docker nicht so leicht einsetzbar, als dass eine Verwendung ohne größeren Mehraufwand für Softwareentwickler ohne Docker-Erfahrung möglich ist.

Werden die Docker-Kommandos allerdings abstrahiert und beinahe unsichtbar in die bestehende Werkzeugkette integriert, wird dank Containervirtualisierung ein plattformübergreifender, stabiler und sich selbst dokumentierender Entwicklungsprozess ermöglicht.

## 6.3 Ausblick

Dem Leser wurde in der Arbeit exemplarisch vermittelt, welche Arten von Problemen sehr einfach mithilfe von Docker lösbar sind. Auf dieser Basis kann der interessierte Leser sich weitere Anwendungsfälle für Docker und Optimierungen der Aufgaben eines Softwareentwicklers überlegen.

Die von Neal Ford und Martin Fowler eingeführten Konzepte des *Polyglot Programming*<sup>1</sup> und der *Polyglot Persistence*<sup>2</sup> sind mithilfe von containerbasierten Lösungen wesentlich einfacher umzusetzen. Auch einer der wohl aktuellsten Trends, *DevOps* (Entwicklung und Betrieb von Anwendungen verschmelzen), wird durch die Verwendung von Containern vereinfacht. Bei DevOps ist es besonders wichtig, bereits zur Entwicklungszeit mit Containern zu arbeiten, da dadurch der Unterschied zwischen Entwicklungs- und Betriebsumgebungen eliminiert werden kann.

Microsoft entwickelt mit den *Windows Containern*<sup>3</sup> eine native Containerlösung für Windows und mit dem *Windows Subsystem for Linux*<sup>4</sup> eine bessere Interoperabilität zu den Linux-Entwicklerwerkzeugen. Diese beiden Projekte werden Containervirtualisierung weiter vorantreiben und völlig neue Möglichkeiten für die Verwendung von Docker auf Windows bieten.

---

<sup>1</sup><http://memeagora.blogspot.co.at/2006/12/polyglot-programming.html>

<sup>2</sup><https://martinfowler.com/bliki/PolyglotPersistence.html>

<sup>3</sup><https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>

<sup>4</sup><https://msdn.microsoft.com/en-us/commandline/wsl/about>

# Abbildungsverzeichnis

2.1	Architektur der Vollvirtualisierung . . . . .	14
2.2	Architektur der Containervirtualisierung . . . . .	15
3.1	Packer, Terraform und Ansible im Einsatz bei npm . . . . .	26
4.1	Docker-Architektur . . . . .	30
4.2	Docker-Dateisystem . . . . .	33
5.1	Docker als Marketinginstrument bei ArangoDB . . . . .	43
5.2	ArangoDB als Docker-Container . . . . .	44
5.3	Architektur des containerbasierten Integrationstestsystems . . . . .	47
5.4	Hello-World-Ausgabe unter Linux . . . . .	49
5.5	vim als Docker-Container . . . . .	51
5.6	Eclipse Che als Docker-Container . . . . .	52

# Quelltextverzeichnis

3.1	Skript zum Installieren des Apache-Webserver ( <i>bootstrap.sh</i> ) . . . . .	22
3.2	Vagrantfile . . . . .	22
3.3	Deployment-Prozess bei npm . . . . .	25
4.1	Ubuntu-Bash in Docker . . . . .	30
4.2	Apache-Dockerfile von Kimbro Staken . . . . .	34
4.3	Erstellen eines Images auf Basis eines Dockerfiles . . . . .	35
4.4	Starten einer Shell in unterschiedlichen Linux-Containern . . . . .	36
4.5	Vergleich der Basis-Imagegröße . . . . .	36
4.6	Dockerfile mit suboptimalem RUN-Befehl . . . . .	37
4.7	Dockerfile mit korrigiertem RUN-Befehl . . . . .	37
4.8	Vergleich der Imagegröße nach Optimierung des RUN-Befehls . . . . .	37
4.9	Dockerfile für <code>ping</code> mit ENTRYPOINT und CMD . . . . .	38
4.10	Verwendung des <code>ping</code> -Containers . . . . .	38
4.11	Docker-Build-Beispiel . . . . .	39
4.12	Docker-Run-Beispiel . . . . .	40
4.13	Docker-Prune-Beispiel . . . . .	40
5.1	Docker-Kommando zum Starten von ArangoDB . . . . .	43
5.2	Docker-Kommando zum Starten von Jekyll . . . . .	45
5.3	Docker-Image für mdp . . . . .	45
5.4	Powershell-Funktionen für Docker-Kommandos . . . . .	46
5.5	Automatische Installation der Docker-basierten CLI-Anwendungen . . . . .	46
5.6	Servicedefinition zum Integrationstesten ( <i>docker-compose.yml</i> ) . . . . .	47
5.7	„Hello-World“-C-Programm . . . . .	49
5.8	Docker-Kommando zum Übersetzen eines C-Programmes mit gcc . . . . .	49
5.9	Kommandos zum Übersetzen mit dockcross . . . . .	50
5.10	Docker-Kommando zum Starten von vim . . . . .	51
5.11	Docker-Kommando zum Starten von Eclipse Che . . . . .	51

# Literatur

- [Ans16] Ansible. *Ansible Getting Started*. 2016. URL: <https://www.ansible.com/quick-start-video> (besucht am 20.02.2017).
- [BAM15] Jürgen Brunk, Matthias Albert und Nils Magnus. *Docker: die Linux-Basics unter der Container-Haube*. 20. März 2015. URL: <https://entwickler.de/online/besuch-im-docker-maschinenraum-126456.html> (besucht am 31.01.2017).
- [BDW16] Thomas J. Bittman, Philip Dawson und Michael Warrilow. *Magic Quadrant for x86 Server Virtualization Infrastructure*. 3. Aug. 2016. URL: <https://www.gartner.com/doc/reprints?ct=160707&id=1-3B9FAM0&st=sb> (besucht am 03.01.2017).
- [Bia16] Randy Bias. *The History of Pets vs Cattle and How to Use the Analogy Properly*. 29. Sep. 2016. URL: <http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/> (besucht am 31.01.2017).
- [BKL09] Christian Baun, Marcel Kunze und Thomas Ludwig. Servervirtualisierung. In: *Informatik-Spektrum* 32.3 (2009), S. 197–205. ISSN: 1432-122X. DOI: 10.1007/s00287-008-0321-6. URL: <http://dx.doi.org/10.1007/s00287-008-0321-6>.
- [Cam16] Marc Campbell. *Refactoring a Dockerfile for Image Size*. 5. Feb. 2016. URL: <https://blog.replicated.com/engineering/refactoring-a-dockerfile-for-image-size/> (besucht am 23.02.2017).
- [Che16] Chef Software, Inc. *Chef – Automate Your Infrastructure / Chef*. 2016. URL: <https://www.chef.io/chef/> (besucht am 01.11.2016).
- [Cre81] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. In: *IBM Journal of Research and Development* 25.5 (Sep. 1981), S. 483–490. ISSN: 0018-8646. DOI: 10.1147/rd.255.0483.
- [DC16] DevOps.com und ClusterHQ. *Container Market Adoption - Survey*. Juni 2016.
- [DeH15] Brian DeHamer. *Dockerfile: ENTRYPOINT vs CMD*. 16. Juli 2015. URL: <https://www.ctl.io/developers/blog/post/dockerfile-entrypoint-vs-cmd/> (besucht am 23.02.2017).
- [Dem16] Daniel Demmel. *Why You Should Stop Installing Your WebDev Environment Locally*. 20. Apr. 2016. URL: <https://www.smashingmagazine.com/2016/04/stop-installing-your-webdev-environment-locally-with-docker> (besucht am 23.02.2017).

- [Die16] Oliver Diedrich. Container: Apps für Server. Wie Docker-Container die IT industrialisieren. In: *c't* 5 (Feb. 2016), S. 108–112.
- [Die17] Oliver Diedrich. Containerverwaltung. Rancher, DC/OS und Kubernetes. In: *iX* 2 (Feb. 2017), S. 32–37.
- [Doc17a] Docker Core Engineering. *Introducing Docker 1.13*. 19. Jan. 2017. URL: <https://blog.docker.com/2017/01/whats-new-in-docker-1-13/> (besucht am 23.02.2017).
- [Doc17b] Docker Inc. *Best practices for writing Dockerfiles*. 22. Feb. 2017. URL: [https://docs.docker.com/engine/userguide/eng-image/dockerfile\\_best-practices/](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) (besucht am 22.02.2017).
- [Doc17c] Docker Inc. *Docker Overview*. 20. Jan. 2017. URL: <https://docs.docker.com/engine/understanding-docker/> (besucht am 22.02.2017).
- [Doc17d] Docker Inc. *Dockerfile reference*. 26. Jan. 2017. URL: <https://docs.docker.com/engine/reference/builder/> (besucht am 22.02.2017).
- [Doc17e] Docker Inc. *Engine User Guide*. 25. Jan. 2017. URL: <https://docs.docker.com/engine/userguide/intro/> (besucht am 23.02.2017).
- [Doc17f] Docker Inc. *Get started with Docker for Windows*. 23. Feb. 2017. URL: <https://docs.docker.com/docker-for-windows/> (besucht am 23.02.2017).
- [Doc17g] Docker Inc. *Install Docker for Mac*. 23. Feb. 2017. URL: <https://docs.docker.com/docker-for-mac/> (besucht am 23.02.2017).
- [Doc17h] Docker Inc. *Install Docker Toolbox on Windows*. 13. Jan. 2017. URL: [https://docs.docker.com/toolbox/toolbox\\_install\\_windows/](https://docs.docker.com/toolbox/toolbox_install_windows/) (besucht am 22.02.2017).
- [Doc17i] Docker Inc. *Understand images, containers, and storage drivers*. 24. Jan. 2017. URL: <https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/> (besucht am 22.02.2017).
- [DSN17] Andy De George, Ralph Squillace und Cynthia Nottingham. *How to Auto Scale a Cloud Service*. 4. Jan. 2017. URL: <https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-how-to-scale> (besucht am 08.01.2017).
- [Eng15] Mike English. *Distributing Command Line Tools with Docker*. 30. Nov. 2015. URL: <https://spin.atomicobject.com/2015/11/30/command-line-tools-docker/> (besucht am 24.02.2017).
- [Fow16] Martin Fowler. *InfrastructureAsCode*. 1. März 2016. URL: <http://martinfowler.com/bliki/InfrastructureAsCode.html> (besucht am 01.11.2016).
- [Gar16] Gartner Inc. *Gartner Says Worldwide Server Virtualization Market Is Reaching Its Peak*. 12. Mai 2016. URL: <http://www.gartner.com/newsroom/id/3315817> (besucht am 03.01.2017).

- [Gol15] Ben Golub. *Docker and Broad Industry Coalition Unite to Create Open Container Project*. 22. Juni 2015. URL: <https://blog.docker.com/2015/06/open-container-project-foundation/> (besucht am 31.01.2017).
- [Han08] Fred Hantelmann. Mit Virtualisierung RZ-Kosten halbieren. In: *iX* 12 (2008), S. 88–91.
- [Han14] Scott Hanselman. *Switch easily between VirtualBox and Hyper-V with a BCDEdit boot Entry in Windows 8.1*. 8. Jan. 2014. URL: <http://www.hanselman.com/blog/SwitchEasilyBetweenVirtualBoxAndHyperVWithABCDEditBootEntryInWindows81.aspx> (besucht am 22.02.2017).
- [Har05] Marcus Hardt. Virtualisation for Grid-Computing. In: *Cracow Grid Workshop* 20 (2005), S. 23.
- [Har16] Harrison Harnisch. *Integration Testing With Docker Compose*. 19. Juni 2016. URL: <https://hharnisc.github.io/2016/06/19/integration-testing-with-docker-compose.html> (besucht am 25.02.2017).
- [Has16] HashiCorp. *Introduction - Packer by HashiCorp*. 12. März 2016. URL: <https://www.packer.io/intro/> (besucht am 10.31.2016).
- [Jan16] Nick Janetakis. *Alpine Based Docker Images Make a Difference in Real World Apps*. 15. Apr. 2016. URL: <https://nickjanetakis.com/blog/alpine-based-docker-images-make-a-difference-in-real-world-apps> (besucht am 23.02.2017).
- [Jek17] Jekyll. *Jekyll on Windows*. 19. Jan. 2017. URL: <https://jekyllrb.com/docs/windows/> (besucht am 24.02.2017).
- [Jon14] Adriaan de Jonge. *Create the smallest possible Docker container*. 4. Juli 2014. URL: <http://blog.xebia.com/create-the-smallest-possible-docker-container/> (besucht am 23.02.2017).
- [Lar16] Laradock. *Laradock Docs*. 2016. URL: <http://laradock.io/> (besucht am 23.02.2017).
- [Lin16] The Linux Foundation. *Open Container Initiative | About*. 2016. URL: <https://www.opencontainers.org/about> (besucht am 31.01.2017).
- [npm16] npm Inc. *How we deploy at npm*. 19. Okt. 2016. URL: <http://blog.npmjs.org/post/152035356435/how-we-deploy-at-npm> (besucht am 21.02.2017).
- [Pup16] Puppet. *Puppet Documentation*. 2016. URL: <https://docs.puppet.com/> (besucht am 20.02.2017).
- [Red15] Red Hat Inc. *The History of Containers*. 28. Aug. 2015. URL: <http://rhelblog.redhat.com/2015/08/28/the-history-of-containers/> (besucht am 22.02.2017).
- [Ree15] Travis Reeder. *Why and How to Use Docker for Development*. 28. Apr. 2015. URL: <https://medium.com/iron-io-blog/why-and-how-to-use-docker-for-development-a156c1de3b24> (besucht am 23.02.2017).

- [rkt17] rkt - CoreOS. *rkt vs Other Projects*. 2017. URL: <https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html> (besucht am 31.01.2017).
- [Sal16a] SaltStack. *Event-Driven Infrastructure*. 2016. URL: <https://docs.saltstack.com/en/getstarted/event/> (besucht am 20.02.2017).
- [Sal16b] SaltStack. *SaltStack Get Started*. 2016. URL: <https://docs.saltstack.com/en/getstarted/> (besucht am 20.02.2017).
- [Tsa15] Ray Tsang. *My Go IDE in a Container*. 3. Nov. 2015. URL: <https://medium.com/google-cloud/my-ide-in-a-container-49d4f177de> (besucht am 24.02.2017).
- [UpG14] UpGuard Inc. *Puppet vs Chef - The Battle Wages On*. Jan. 2014. URL: <https://www.upguard.com/blog/puppet-vs-chef-battle-wages> (besucht am 20.02.2017).
- [UpG16] UpGuard Inc. *Puppet vs Chef Revisited*. Nov. 2016. URL: <https://www.upguard.com/blog/puppet-vs-chef-battle-wages> (besucht am 20.02.2017).
- [Var16] Seth Vargo. *Documentation - Vagrant by HashiCorp*. Jan. 2016. URL: <https://www.vagrantup.com/docs/> (besucht am 31.10.2016).
- [VMw11] VMware Inc. *Grundlagen der Virtualisierung*. abgerufen via <https://web.archive.org/web/20110204170905/http://www.vmware.com/de/overview/history.html>. 2. Apr. 2011. URL: <http://www.vmware.com/de/overview/history.html> (besucht am 03.01.2017).
- [Wol14] Eberhard Wolff. *Continuous Delivery: Der pragmatische Einstieg*. dpunkt.verlag, Okt. 2014. ISBN: 9783864902086.