Jerry Preissler
(gerald.preissler@innoq.com)
ist Senior Consultant bei der innoQ Deutschland GmbH. Er ist Entwickler und Architekt, sein besonderes Interesse gilt im Augenblick verteilten Systemen sowie der Programmierung von reaktiven und event-basierten Systemen.

Oliver Tigges
(oliver.tigges@innoq.com)
ist Principal Consultant bei der innoQ Deutschland GmbH. Er befasst sich seit zehn Jahren mit der Architektur und Realisierung von Webanwendungen und verteilten Unternehmensanwendungen. Sein besonderes Interesse gilt momentan den Themen DevOps, Continuous Delivery und Graphendatenbanken.

Docker – perfekte Verpackung von Microservices

Die Microservice-Bewegung ist mit viel Schwung gestartet und in großen Unternehmen angekommen. Überall werden Monolithen zerschlagen und durch eigenständige, fachlich definierte Microservices ersetzt. Entwicklungsteams können eigenverantwortlicher und autonomer agieren und damit deutlich schneller Ergebnisse ausrollen. Aus Sicht des IT-Betriebs bringen die Microservices aber auch eine Menge Herausforderungen: Statt weniger großer und etablierter Unternehmensanwendungen existieren plötzlich Landschaften mit einer Vielzahl an kleinen, sich schnell ändernden Services, die alle konfiguriert, integriert und überwacht werden wollen. Eine Container-Technologie wie Docker kann das ideale Mittel sein, um diese Services zu verpacken und auszurollen. Aber welche Probleme löst Docker genau und welche neuen Herausforderungen bringt es mit? In diesem Artikel zeigen wir, für welche Softwarearchitekturen Docker geeignet ist und wie ein Einsatz von Docker dabei helfen kann, Microservice-Architekturen zu standardisieren, zu steuern und zu kontrollieren.

Container und Docker

Bevor wir uns mit der Microservice-Architektur beschäftigen, stellen wir kurz Docker an sich vor. Klassische Virtualisierungslösungen, wie z. B. VMware und XEN, emulieren komplette virtuelle Maschinen, in denen jeweils Kernel und alle Prozesse ausgeführt werden. Docker verwendet eine Virtualisierung auf Ebene des Betriebssystems, d. h. die virtuellen Umgebungen laufen direkt auf dem Kernel des Host-Systems und werden mithilfe von verschiedenen Mechanismen, die dieser Kernel zur Verfügung stellt, vom Rest des Systems abgeschottet.

Dieser Ansatz hat gegenüber anderen Virtualisierungstechniken den Vorteil, dass er deutlich leichtgewichtiger ist. Da Docker-Container keinen eigenen Kernel instanziieren, benötigen sie deutlich weniger RAM und auch der Start eines Containers geschieht in einem Bruchteil der Zeit, verglichen mit einer klassischen Virtualisierungslösung. Der Rechenzentrumsbetrieb kann dadurch auf identischer Hardware deutlich mehr Instanzen von virtuellen Umgebungen starten und erschließt sich das Potenzial für den Einsatz hochdynamischer, demand-gesteuerter Deployment-Modelle.

Diese Vorteile werden jedoch mit einigen Einschränkungen erkauft. Die zunächst auffälligste ist dabei, dass Docker nur auf Linux läuft. Wer Windows oder Mac OS X verwenden will, muss den Umweg über eine Linux-VM gehen [boot2docker].

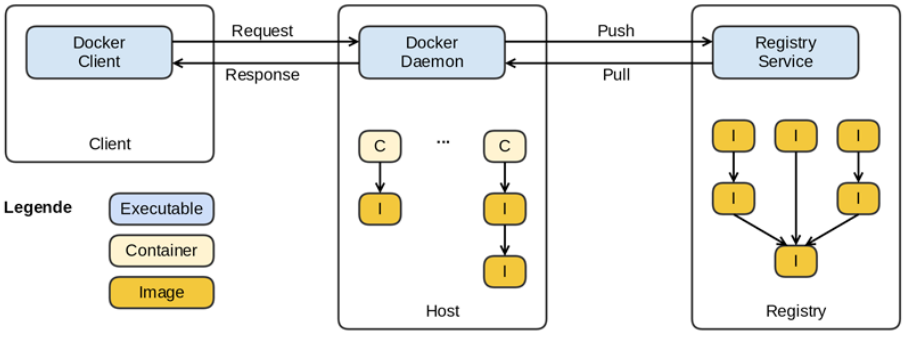

Abb. 1: Die Komponenten von Docker

Komponenten

Zum Einstieg geben wir einen kurzen Überblick darüber, welche Komponenten in einem Docker-System verwendet werden und wie sie zusammenspielen (vgl. Abbildung 1).

Docker Daemon

Die zentrale Komponente einer Docker-Installation ist der Docker Daemon. Er verwaltet die lokalen Container und Images sowie die zugehörigen Netzwerkkomponenten und Dateisysteme.

Image

Ein Image stellt eine Blaupause für eine virtuelle Umgebung zur Verfügung. Es enthält die vom Kernel benötigten Dateien, um die Anwendung(en), die in der Umgebung betrieben werden sollen, ordnungsgemäß auszuführen.

Zusätzlich zum Dateisystem enthält ein Image Metadaten, z. B. das Base-Image und das Binary, das bei der Erstellung einer virtuellen Umgebung gestartet werden soll. Images können auf bereits existierenden Images aufbauen, sodass in mehreren Schritten – üblicherweise ausgehend von einer abgespeckten Linux-Distribution – die gewünschte Umgebung bereitgestellt werden kann.

Container

Ein Container ist eine durch Docker zur Verfügung gestellte virtuelle Umgebung. Beim Start eines Containers wird ein Dateisystem erzeugt, welches die (unveränderlichen) Dateien eines Images einbindet und mit einem read/write Layer kombiniert. Anschließend wird ein vom Rest des Hostsystems isolierter Prozess gestartet, der die für das Image definierte Anwendung ausführt.

Docker Client

Mit dem Docker Client kann der Anwender den Docker Daemon steuern. Dazu stehen ihm Kommandozeilenbefehle zur Verfügung, die der Client über ein REST-API an den Docker Daemon weiterleitet.

Registry

Eine Registry ist in der Docker-Welt ein Web-Dienst, der Docker-Images speichert, verwaltet und Docker-Anwendern zur Verfügung stellt. Das Unternehmen Docker Inc. stellt eine öffentliche Registry unter dem Namen „Docker Hub“ bereit, die eine große Auswahl an Images für die verschiedensten Anwendungsfälle bietet.

Es ist jedoch problemlos möglich, eine eigene Instanz einer Docker-Registry öffentlich oder innerhalb eines abgetrennten Netzes zu betreiben. Für den Betrieb einer eigenen Registry bietet der Docker Hub ein entsprechendes Image an.

Filesysteme im Docker-Container

Docker verwendet für Images und Container ein Dateisystem mit mehreren Schichten. Ausgehend von einem Basisimage referenziert jede Schicht das zugrunde liegende Image und erfasst nur neue, veränderte und gelöschte Dateien und Verzeichnisse. Die darüber liegenden Schichten erhalten eine vereinheitlichte Sicht auf das Resultat. Dabei sind Schichten, die durch Images definiert werden, nur lesend zugänglich.

Docker persistiert alle Änderungen, die während des Betriebs vorgenommen werden, in einer obersten Schicht, sodass Nutzer die Container anhalten und neu starten können, ohne dass ihre Änderungen verloren gehen. Da ein Image von mehreren anderen Images und Containern referenziert werden kann, kann durch diesen Ansatz der Speicherbedarf deutlich reduziert werden.

Aus Sicht des Containers ist das Dateisystem völlig isoliert. Es existiert zunächst keine Möglichkeit aus dem Container heraus auf Verzeichnisse anderer Container oder des Hostsystems zuzugreifen. Diese Beschränkung kann durch die Verwendung von Volumes umgangen werden.

Ein Volume definiert ein Verzeichnis innerhalb des Dateisystems eines Containers. Andere Container oder das Hostsystem können dieses Volume dann an einer Stelle in ihrem eigenen Dateisystem einhängen. Auf diese Weise können Dateien, z. B. zur Datensicherung, Analyse oder zu anderen Zwecken zwischen Container und Host geteilt werden.

Images erstellen

Auf dem Docker Hub steht bereits eine große Anzahl von Images für die verschiedensten Anwendungsfälle zur Verfügung. Die Herausforderung ist oftmals weniger, ein entsprechendes Image zu finden, sondern aus der Vielzahl der verfügbaren Alternativen diejenige auszuwählen, die für den eigenen Anwendungsfall wirklich passt, allen Qualitätsanforderungen genügt und auch weiter gepflegt wird.

Wenn sich ein Nutzer entschieden hat, ein eigenes Image zu erstellen, wählt er zunächst ein Basisimage aus, auf dem alles aufbaut. Auch dazu stehen viele Alternativen zur Verfügung – von einfachen Basisinstallationen einer Linux-Distribution bis zu vorbereiteten Application-Servern. Erhältlich sind diese Images auf dem Docker Hub oder auch von Repositorys der entsprechenden Anwendungsentwickler.

Der einfachste Weg, auf dieser Basis ein neues Image zu erstellen, besteht darin, damit einen neuen Container zu starten und dann mit den Bordmitteln der gewählten Linux-Distribution die gewünschte Software zu installieren und zu konfigurieren. Anschließend kann ein Snapshot des neuen Containers erstellt werden (in der Docker-Terminologie ist dies ein „commit“), der das neue Image darstellt.

Dieser Ansatz ist ohne weitere Vorbereitungen durchführbar und für schnelle Experimente gut geeignet. Er hat jedoch den entscheidenden Nachteil, dass die einzelnen Installationsschritte manuell durchgeführt werden und der Weg zum erreichten Ergebnis nicht dokumentiert wird. Damit ist die Erstellung des Images für Dritte nicht nachvollziehbar und auch nicht automatisiert wiederholbar.

Dieses Problem wird bei einem automatisierten Build unter Verwendung eines Dockerfiles vermieden. Ein Dockerfile fasst alle Anweisungen, die zur Erstellung des neuen Images benötigt werden, sowie die Metadaten über das Basisimage, den Ersteller und exportierte Volumes und Ports in einer Textdatei zusammen. Damit kann durch Aufruf eines entsprechenden Befehls ein neues Image erzeugt werden.

Es ist jedoch zu beachten, dass das Ergebnis – abhängig vom verwendeten Basisimage und den im Dockerfile enthaltenen Anweisungen – auch hier nicht immer exakt wiederholbar ist. Wenn z. B. Pakete installiert werden, die zwischenzeitlich in der Quelldistribution aktualisiert wurden, wird sich das resultierende Image entsprechend unterscheiden.

Ein Container = Ein Prozess

Auch wenn es grundsätzlich möglich wäre, mehrere Prozesse in einem Container auszuführen, z. B. einen Apache-Web-Server, eine Java-Webanwendung und eine Datenbank, unterstützen wir die Empfehlung der Docker-Community, einen Container als einen Prozess zu betrachten. Der Apache-Web-Server, die Java-Webanwendung und die Datenbank würden nach dieser Denkweise in drei verschiedene Images verpackt. Der IT-Betrieb kann

dann pro Image entscheiden, wie viele Container gestartet werden. Das System skaliert also über das Starten weiterer unabhängiger Prozesse, ein sogenanntes horizontales Scale-Out.

Nach diesem kurzen Überblick über die Grundlagen von Docker wollen wir nun ein sehr interessantes Einsatzgebiet betrachten.

Microservice-Architekturen

Das Leitbild für Softwaresysteme hat sich in den letzten Jahren stark verändert. Auch wenn viele Systeme schon früher intern modular oder komponentenorientiert aufgebaut waren, wurden sie oft als monolithisches Artefakt auf die vom IT-Betrieb konfigurierten Applikationsserver [asdead] installiert. Zu diesem Vorgehen hat sich in letzter Zeit eine Alternative etabliert.

Im Rahmen der Microservices-Bewegung gehen Architekten und Entwicklerteams inzwischen immer mehr dazu über, Monolithen zu zerschlagen [brkmlt] und durch fachlich definierte Microservices zu ersetzen. Diese Services können im Idealfall von autonom arbeitenden Teams unabhängig voneinander entwickelt, freigegeben und in Betrieb genommen werden.

In der Regel sind Microservices ausführbare Programme, die einen Großteil der notwendigen Abhängigkeiten und Konfiguration mitbringen und direkt auf dem Betriebssystem eines Servers ausgeführt werden. Die Services starten bei Bedarf z. B. auch selber einen HTTP-Server und reservieren die entsprechenden Ports. Der IT-Betrieb konfiguriert und installiert die Applikationsserver auf den Test-, Abnahme- und Produktionsumgebungen also nicht selbst, sondern stellt den Entwicklungsteams nur noch die notwendigen Plattformen zur Verfügung.

Docker passt mit seinem „Single-Process"-Modell ideal zu diesem Vorgehen. Ein Docker-Container standardisiert die Ablaufumgebung und das Format zur Verpackung eines Microservices mitsamt allen Abhängigkeiten: Es werden nicht nur die notwendigen Software-Bibliotheken – z. B. Jar-, Gem-Dateien oder Npm-Module –, sondern auch Abhängigkeiten auf Betriebssystemebene und die benötigten Ablaufumgebungen definiert, also z. B. eine Java Runtime oder C-Bibliotheken.

Architekturebenen

Beim Entwurf und der Umsetzung von Microservice-Architekturen sind Entscheidungen auf drei verschiedenen Ebenen zu treffen (siehe Abbildung 2).

Abb. 2: Überblick über die relevanten Architekturebenen

Ein Microservice soll eine fachlich definierte Aufgabe übernehmen. Der Schnitt und die Zuständigkeiten der Microservices richten sich also nach der fachlichen **Domänenarchitektur**.

Auf Ebene der **Makroarchitektur** werden zwei Bereiche von der Systemarchitektur definiert:

1. *Die Kommunikation zwischen den Microservices*. Hier beschreibt der Architekt die Integrationsmuster sowie die zu verwenden Protokolle und Datenformate. Typisch für Microservices ist beispielsweise eine Integration über HTTP und Hypermedia-REST-APIs mit JSON als Datenaustauschformat.
2. *Die Integration der Services in die „Plattform", also die Infrastruktur aus Servern und Netzwerken*. Hier gilt es ein übergreifendes Konzept für Paketierung, Installation, Konfiguration und Überwachung der Services zu erarbeiten. Der Architekt muss Fragen dazu beantworten, wie Services sich gegenseitig auffinden, wie Logging und Monitoring funktionieren und wie sie Netzwerk, Load-Balancern und Firewalls bekannt gemacht werden.

Eine funktionierende Makroarchitektur stellt sicher, dass in verschiedenen Programmiersprachen und Technologien implementierte Services über gleichförmige Mechanismen integriert und betrieben werden können. Die Schnittstellen zur Kommunikation und zur Plattform sind also uniform, die Services selber dürfen dadurch heterogen sein.

Die letzte der drei Ebenen ist die **Mikroarchitektur**. Hier werden Entscheidungen getroffen, die nur den internen Aufbau des einzelnen Service betreffen. Dadurch kann das Entwicklungsteam Technologieentscheidungen autonom im Projekt- und Anwendungskontext treffen. So lange die Vorgaben der Makroarchitektur eingehalten werden, spielt es für Außenstehende keine Rolle, ob ein Service z. B. in Java, Scala, Ruby oder mit Node.js implementiert ist oder welche Persistenztechnik er intern verwendet.

Mit Docker ist es möglich, den Entwicklerteams bei der Mikroarchitektur die größtmögliche Freiheit zu gewähren. Trotzdem bieten die als Docker-Images verpackten Microservices dem IT-Betrieb eine uniforme Schnittstelle zum Starten, Überwachen und Skalieren der Services.

Self-Contained-Systems

Eine Implementierungsmöglichkeit für Microservices sind Self-Contained-Systems (auch als „Vertikalen" bekannt). Dabei werden Services in eigenständige Umgebungen verpackt, die alle Abhängigkeiten mitbringen. Ein Docker-Container ist eine ideale Umgebung für diesen Ansatz. Über seine Ablaufumgebung, das Host-Betriebssystem, kann er auf Rechenressourcen und Basisdienste wie IO und Netzwerk zugreifen.

Basisdienste wie z. B. SMTP können durch andere Container zur Verfügung gestellt werden. Alles andere muss der Container selbst mitbringen:

- notwendige Linux-Pakete und C-Bibliotheken,
- Ablaufumgebungen, wie ein JRE, Ruby, Python oder Node.js,
- (Application Server, wenn notwendig),
- die Anwendung selbst mit ihren benötigten Bibliotheken,
- Konfigurationsdateien.

Das Entwicklungsteam nimmt die Konfiguration der Anwendungsinfrastruktur also schon während der Implementierung vor. Das Ergebnis des Build-Prozesses ist nicht mehr ein Software-Artefakt, sondern ein vollständiges und in sich abgeschlossenes Image der Ablaufumgebung. Über einige wenige Parameter können beim Start des Containers umgebungsspezifische Werte gesetzt werden (Ports, Security-Credentials und -Keys, etc.). Ansons-

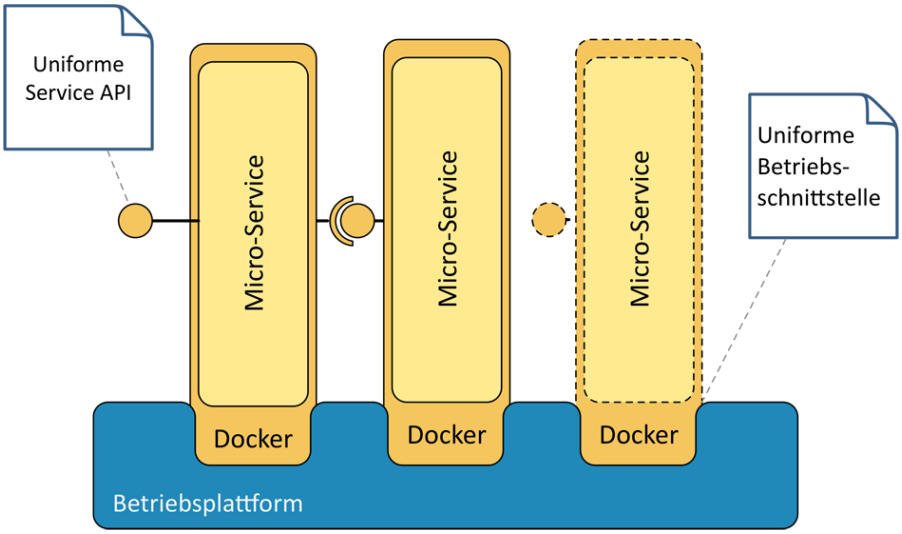

Abb. 3: Uniforme Betriebsplattformen mit Docker

ten muss und sollte beim Starten des Containers nichts mehr getan werden (vgl. Abbildung 3).

Hierdurch wird eine Entwicklung fortgeführt, bei der Softwareentwicklung und IT-Betrieb enger zusammenarbeiten und sich Zeitraum und Inhalt der klassischen „Übergabe an den Betrieb“ gravierend verändern [devops]. Wurden in der Vergangenheit Software-Artefakte mitsamt einer manuellen Installations- und Konfigurationsanleitung an einen Rechenzentrumsbetrieb übergeben, hat sich mittlerweile die Idee von Infrastruktur als Code [iac] etabliert. Serveränderungen sind nur durch Ausführen versionierter, wiederholbarer und idempotenter Scripte erlaubt (z. B. mithilfe von Puppet, Chef oder Ansible).

Durch den Einsatz von Docker wird es in Zukunft möglich, komplette Serverumgebungen versioniert und mit eingefrorenem Konfigurationsstand bereitzustellen. Diese können einfach in Betrieb genommen und einheitlich überwacht werden. Der IT-Betrieb konfiguriert nicht mehr einzelne Systemkomponenten selbst, sondern prüft die ganze Umgebung und nimmt sie ab.

Immutable Server

Ein einmal gestarteter Container wird nicht mehr umkonfiguriert oder aktualisiert. Wenn sich die Anwendung oder ihre Abhängigkeiten eines Microservices ändern, stellt das Entwicklungsteam stattdessen ein neues Image zur Verfügung. Der IT-Betrieb startet dann die neuen Instanzen und stoppt im Anschluss die alte Version.

Auch wenn ein Container oder die darin laufende Software im Betrieb Probleme bereiten, muss der IT-Betrieb nicht langwierig versuchen, die Instanz zu reparieren. Er kann einfach die Logs für eine Post-Mortem-Analyse sichern und dann den Container abschalten und eine neue Instanz hochfahren. Diese Fehlererkennung und -behebung lässt sich grundsätzlich vollständig automatisieren.

Atmende Serverlandschaften

Durch die Möglichkeit, blitzschnell zusätzliche Instanzen eines Services zu starten, ermöglicht Docker dem IT-Betrieb, umgehend auf Änderungen der Systemlast zu reagieren. Wenn z. B. zu bestimmten Tageszeiten besonders viele Anwender mit dem System arbeiten, sich in der Nacht aber nur sehr sporadisch Nutzer einloggen, können sehr bedarfsgerecht Container zu- oder abgeschaltet werden.

Diesen Prozess der Lastanalyse und entsprechender Skalierung kann der IT-Betrieb prinzipiell auch automatisieren. Wirtschaftlich lohnt sich das insbesondere dann, wenn die Kosten für die Infrastruktur sehr stark von der tatsächlichen Ressourcennutzung abhängen. Im Cloud Computing kann diese oft als „Auto Scaling“ oder „Breathing Cloud“ bezeichnete, dynamische Skalierung sehr viel Geld sparen.

Eliminieren von Umgebungsunterschieden

Durch das Verpacken von Microservices in unveränderliche Docker-Images können wir Abweichungen zwischen verschiedenen Entwicklungs-, Test- und Produktionsumgebungen auf ein Minimum reduzieren. Das Docker-Image friert nicht nur die Versionen der Anwendungsbibliotheken ein, sondern auch das gesamte Betriebssystem und alle Konfigurationsdateien.

Im Idealbild konzentriert sich der IT-Betrieb auf die Bereitstellung von Netzwerk, Hardware, Host-Betriebssystemen und die Überwachung der Docker-Prozesse. Von den Konfigurationsdetails einer Vielzahl unterschiedlichster Anwendungen bleibt er weitestgehend verschont.

Zusammenfassung und Ausblick

In diesem Artikel haben wir gezeigt, warum Docker eine interessante Plattform für die Auslieferung und den Betrieb von Microservices ist:

1. *Docker-Images sind Self-Contained*: Microservices bringen also alle ihre Abhängigkeiten selber mit und stellen sie transparent dar.
2. *Horizontales Scale-Out*: Mit Docker werden Microservices durch das Starten weiterer paralleler Prozesse skaliert.
3. *Uniforme Betriebsschnittstelle*: Die Konzepte für Betrieb und Überwachung von Microservices werden standardisiert, unabhängig von Technologie-Stacks oder Ablaufumgebungen.

Docker hat in seiner noch kurzen Lebenszeit viele Fans und Unterstützer gewonnen, darunter auch viele große Unternehmen und Cloud-Anbieter. Grundsätzlich erachten wir Docker und sein Ökosystem als tauglich für den Praxiseinsatz. In vielen Bereichen müssen Anwender heute noch individuelle Strategien für den Betrieb entwickeln und sich aus einer großen Vielfalt von Werkzeugen für Orchestrierung und Überwachung von Docker-Systemen ihr eigenes Toolset definieren. Hier rechnen wir damit, dass sich in den nächsten Jahren noch Standards und allgemeine Best-Practices herausbilden, die auch diese Einstiegshürde senken werden.

Auch ist klar, dass sowohl Entwickler als auch Administratoren für einen erfolgreichen Einsatz von Docker entsprechendes Know-how aufbauen müssen. Sie müssen gemeinsam geeignete Workflows definieren, die Sicherheitsimplikationen verstehen und berücksichtigen und die rapide Entwicklung der Technologie konti-

nuierlich beobachten. Wie wir hoffentlich gezeigt haben, stehen diesem Aufwand jedoch substantielle Vorteile bei der Verwendung der Technologie gegenüber.

Wenn Ihre Infrastruktur auf großen Application-Servern und Datenbanksystemen basiert, ist Docker vermutlich nicht der richtige Startpunkt um eine flexiblere IT-Infrastruktur aufzubauen. Vielleicht ist dann der Einsatz von „Infrastructure as Code“ [iac] ein zielführenderer Schritt.

Wenn Sie aber auf Microservice-Architekturen setzen, können wir eine Evaluierung von Docker auf jeden Fall empfehlen. ■

Ressourcen und Links:

[docker] https://www.docker.com/
[boot2docker] http://boot2docker.io/
[brkmlt] http://blog.matthewskelton.net/2012/03/20/breaking-the-monolith-by-stefan-tilkov-at-qconlondon-2012/
[msjava] Phillip Ghadir, 04/2014 der Zeitschrift JavaSPEKTRUM, „Microservices in Java realisieren“
[devops] Oliver Wolf, 02/2012 der Zeitschrift iX, „Wider Betonköpfe und Freigeister“, iX 02/2012,
https://www.innoq.com/en/articles/2012/10/wider-betonkoepfe-und-freigeister/
[asdead] Eberhard Wolff, „Application Servers are dead!“,
http://jaxenter.com/java-application-servers-dead-1-111928.html
[iac] Martin Eigenbrodt, „Infrastructure as Code“,
https://www.innoq.com/en/articles/2012/04/infrastructure-as-code/