Virtualisierung mit Docker

5.1 Docker für Microservices: Gründe

Kapitel 1 hat Microservices als getrennt deploybare Einheiten definiert. Das getrennte Deployment ergibt nicht nur eine Entkopplung auf Ebene der Architektur, sondern auch bei der Technologiewahl, der Robustheit, der Sicherheit und der Skalierbarkeit.

5.1.1 Prozesse reichen für Microservices nicht aus.

Wenn Microservices tatsächlich alle diese Eigenschaften aufweisen sollen, stellt sich die Frage, wie sie umgesetzt werden können. Microservices müssen unabhängig voneinander skaliert werden. Ein Microservice darf bei einem Absturz andere Microservices nicht mitreißen und so die Robustheit gefährdet. Also müssen Microservices mindestens getrennte Prozesse sein.

Die Skalierbarkeit kann durch mehrere Instanzen eines Prozesses gewährleistet werden. Wenn eine Anwendung gestartet wird, so erzeugt das Betriebssystem einen Prozess und weist ihm Ressourcen wie CPU oder Speicher zu. Für die Skalierbarkeit können in einem Prozess mehrere Threads aktiv sein. Jeder Thread hat einen eigenen Ausführungszustand. Ein Prozess kann jeden Request mit einem Thread abarbeiten.

Für die Skalierung sind Prozesse trotz der Unterstützung von Threads nicht ausreichend: Wenn die Prozesse alle auf einem Server laufen, dann steht nur eine begrenzte Menge an Hardware-Ressourcen bereit. Die Isolation in Prozesse ist also für die Skalierbarkeit nicht ausreichend.

Die Robustheit ist bei Prozessen bis zu einem gewissen Maße gewährleistet, weil der Absturz eines Prozesses die anderen Prozesse nicht beeinflusst. Ein Ausfall eines Servers bringt immer noch eine Vielzahl von Prozessen und damit Microservices zum Absturz. Es gibt aber noch andere Probleme. Alle Prozesse teilen sich ein Betriebssystem. Es muss die Bibliotheken und Werkzeuge für alle Microservices mitbringen. Jeder Microservice muss mit der Version des Betriebssystems kompatibel sein. Es ist schwierig, das Betriebssystem so zu konfigurieren, dass es für alle Microservices passt. Außerdem müssen die Prozesse sich so koordinieren, dass jeder Prozess einen eigenen Netzwerk-Port hat. Man verliert bei solchen Ansätzen sehr schnell den Überblick über die genutzten Ports.

5.1.2 Virtuelle Maschinen sind zu schwergewichtig für Microservices

Statt eines Prozesses kann jeder Microservice in einer eigenen virtuellen Maschine laufen. Virtuelle Maschinen sind simulierte Rechner, die gemeinsam auf derselben Hardware laufen. Für das Betriebssystem und die Anwendung sehen virtuelle Maschinen genauso aus wie ein Hardware-Server. Durch die Virtualisierung hat der Microservice ein eigenes Betriebssystem. So kann die Konfiguration des Betriebssystems auf den Microservice abgestimmt sein und es besteht auch vollkommene Freiheit bei der Wahl des Ports.

Aber eine virtuelle Maschine hat einen erheblichen Overhead:

- Die virtuelle Maschine muss dem Betriebssystem die Illusion geben, direkt auf der Hardware zu laufen. Das führt zu einem Overhead. Daher ist die Performance schlechter als bei physischer Hardware.
- Jeder Microservice hat eine eigene Instanz des Betriebssystems. Das verbraucht viel Speicher im RAM.
- Schließlich hat die virtuelle Maschine virtuelle Festplatten mit einer vollständigen Betriebssysteminstallation. Dadurch belegt der Microservice viel Speicherplatz auf der Festplatte.

Also haben virtuelle Maschinen einen Overhead. Das macht den Betrieb teuer. Außerdem muss der Betrieb eine Vielzahl virtueller Server handhaben. Das ist aufwendig und kompliziert.

Ideal wäre eine leichtgewichtige Alternative zur Virtualisierung, die zwar die Isolation von virtuellen Maschinen hat, aber so wenig Ressourcen verbraucht, wie es Prozesse tun, und auch ähnlich einfach zu betreiben ist.

5.2 Docker-Grundlagen

Docker stellt eine leichtgewichtige Alternative zur Virtualisierung dar. Docker liefert zwar keine so starke Isolation wie eine Virtualisierung, ist dafür aber praktisch genauso leichtgewichtig wie ein Prozess:

- Statt einer vollständigen eigenen virtuellen Maschine *teilen* sich Docker-Container den *Kernel* des Betriebssystems auf dem Docker-Host. Der Docker-Host ist das System, auf dem die Docker-Container laufen. Die Prozesse aus den Containern tauchen daher in der Prozesstabelle des Betriebssystems auf, auf dem die Docker-Container laufen.
- Die Docker-Container haben ein *eigenes Netzwerkinterface*. So kann derselbe Port in jedem Docker-Container neu belegt werden und jeder Container kann beliebig viele Ports nutzen. Das Netzwerkinterface ist in einem Subnetz, in dem alle Docker-Container zugreifbar sind. Das Subnetz ist von außen nicht zugreifbar. Das ist zumindest die Standard-Konfiguration von Docker. Die Docker-Netzwerkkonfiguration bietet noch viele weitere Alternativen. Um dennoch einen Zugriff von außen auf einen Docker-Container zu ermöglichen, können Ports eines Docker-Containers auf Ports auf dem Docker-Host. Beim Binden der Ports der Docker-Container an die Ports des Docker-Hosts muss man vorsichtig sein, weil jeder Port des Docker-Hosts natürlich nur an einen Port eines Docker-Containers gebunden werden kann.

Schließlich ist das Dateisystem optimiert. Es gibt *Schichten im Dateisystem*. Wenn ein Microservice eine Datei liest, geht er die Schichten von oben nach unten durch, bis er die Daten findet. Die Container können sich Schichten teilen. Abbildung 5–1 zeigt das genauer: Eine Dateisystemschicht stellt eine einfache Linux-Installation mit der Alpine-Linux-Distribution dar. Eine weitere Schicht ist die Java-Installation. Beide Anwendungen teilen sich die Schichten. Diese Schichten sind nur einmal auf der Festplatte gespeichert, obwohl beide Microservices sie nutzen. Nur die Anwendungen sind jeweils in Dateisystem-Schichten abgelegt, die exklusiv für einen Container zur Verfügung stehen. Die unteren Schichten sind nicht änderbar. Die Microservices können nur auf die oberste Schicht schreiben. Durch die Wiederverwendung der Schichten sinkt der Speicherbedarf der Docker-Container.

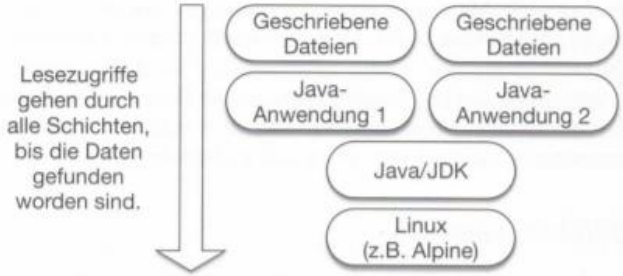

Abb. 5–1 *Dateisystem-Schichten in Docker*

Es ist ohne Weiteres möglich, Hunderte von Containern auf einem Laptop zu starten. Das ist nicht verwunderlich: Schließlich ist es ja auch möglich, Hunderte von Prozessen auf einem Laptop zu starten. Docker hat gegenüber einem Prozess keinen signifikanten Overhead. Im Vergleich zu virtuellen Maschinen sind die Performance-Vorteile aber überragend.

5.2.1 Ein Prozess pro Container

Letztendlich sind Docker-Container durch eigene Netzwerkschnittstelle und Dateisystem stark isolierte Prozesse. Daher sollte in einem Docker-Container nur ein Prozess laufen. Mehr als ein Prozess in einem Docker-Container widerspricht der Idee, durch Docker-Prozesse besonders stark voneinander zu trennen. Weil in einem Docker-Container eigentlich nur ein Prozess laufen soll, gibt es auch keine Hintergrunddienste oder Daemons in Docker-Containern.

5.2.2 Docker-Image und Docker-Registry

Dateisysteme von Docker-Containern können als Docker-Images exportiert werden. Diese Images können als Dateien weitergegeben werden oder in einer Docker-Registry gespeichert werden. Viele Repositories wie Nexus (*https://www.sonatype.com/nexus-repository-sonatype*) und Artifactory (*https://www.jfrog.com/open-source/#artifactory*), die zur Verwaltung von kompilierter Software dienen, können auch Docker-Images speichern und bereitstellen. So ist es ohne Weiteres möglich, für die Installation in der Produktion Docker-Images mit einer Docker-Registry auszutauschen. Die Übertragung der Images von und zu der Registry ist optimiert. Es werden jeweils nur die aktualisierten Schichten übertragen.

5.2.3 Unterstützte Betriebssysteme

Docker ist ursprünglich eine Linux-Technologie. Für Betriebssysteme wie macOS und Windows stehen Docker-Installationen bereit, die es ermöglichen, Linux-Docker-Container zu starten. Dazu läuft im Hintergrund eine virtuelle Maschine mit einer Linux-Installation. Für den Benutzer ist das transparent. Es wirkt so, als würden die Docker-Container direkt auf einem Rechner laufen.

Für Windows gibt es außerdem Windows-Docker-Container. In einem Linux-Docker-Container können Linux-Anwendungen laufen, in einem Windows-Docker-Container Windows-Anwendungen.

5.2.4 Betriebssysteme für Docker

Durch Docker ändern sich die Anforderungen an die Betriebssysteme:

- In einem *Docker-Container* soll nur ein Prozess laufen. Es ist also nur so viel von dem Betriebssystem notwendig, wie man für den Betrieb eines Prozesses benötigt. Bei einer Java-Anwendung ist das die Java Virtual Machine (JVM), die einige Linux-Bibliotheken benötigt, die zur Laufzeit geladen werden. Eine Shell ist beispielsweise nicht notwendig. Daher gibt es Distributionen wie Alpine Linux (*https://alpinelinux.org/*), die nur wenige Megabyte belegen und nur die wichtigsten Werkzeuge mitbringen. Sie sind eine ideale Basis für Docker-Container. Die Programmiersprache Go kann statisch gelinkte Programme erzeugen. Dann muss neben dem Programm selber nichts mehr in dem Docker-Container verfügbar sein. Es ist also auch keine Linux-Distribution notwendig.
- Der *Docker-Host*, auf dem die Docker-Container laufen, muss nur Docker-Container laufen lassen. Viele Linux-Werkzeuge sind überflüssig. CoreOS (*https://coreos.com/*) ist eine Linux-Distribution, die wenig mehr kann als Docker-Container ablaufen zu lassen, aber zum Beispiel Betriebssystem-Updates eines ganzen Clusters wesentlich vereinfacht. CoreOS kann auch als Basis für Kubernetes dienen (siehe auch Kapitel 17). Ein anderes Beispiel ist boot2docker (*http://boot2docker.io/*), das Docker Machine (siehe Abschnitt 5.4) auf Servern installiert, um Docker-Container auf diesen Servern auszuführen. Auch diese Linux-Distribution kann im Wesentlichen nur Docker-Container ausführen.

5.2.5 Überblick

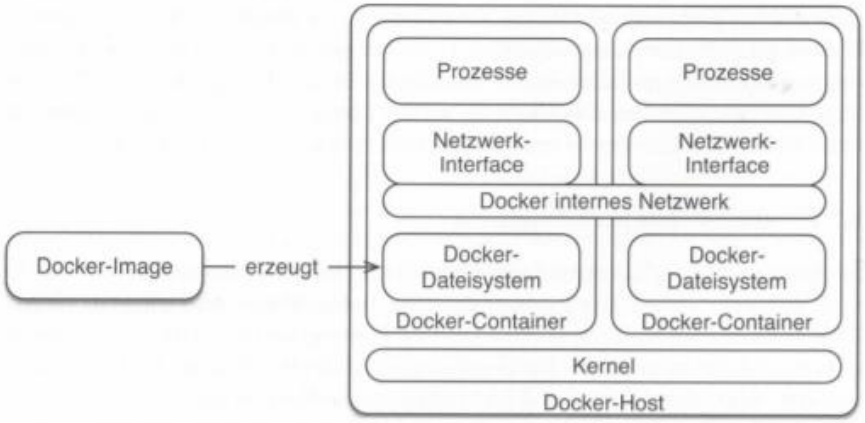

Abb. 5–2 *Überblick über Docker*

Abbildung 5–2 zeigt die Docker-Konzepte im Überblick:

- Der *Docker-Host* ist die Maschine, auf der die Docker-Container laufen. Es kann eine virtuelle Maschine sein oder eine physische.
- Auf dem Docker-Host laufen *Docker-Container*.
- Die Container enthalten typischerweise einen *Prozess*.
- Jeder Container hat ein eigenes *Netzwerk-Interface* mit einer eigenen IP-Adresse. Dieses Netzwerk-Interface ist nur vom Docker-internen Netzwerk aus zugreifbar. Allerdings gibt es Möglichkeiten, Zugriff auch von außerhalb dieses Netzwerks zuzulassen.
- Außerdem hat jeder Container ein eigenes *Dateisystem* (siehe Abbildung 5–1).
- Beim Start eines Containers erzeugt das *Docker-Image* die erste Version des Docker-Dateisystems. Wenn der Container gestartet ist, wird das Image um eine weitere Schicht ergänzt, in die der Container seine eigenen Daten schreiben kann.
- Alle Docker-Container teilen sich den *Kernel* des Docker-Hosts.

5.2.6 Muss es immer Docker sein?

Docker ist eine sehr populäre Möglichkeit, Microservices zu deployen. Aber es gibt Alternativen. Eine Alternative hat Abschnitt 5.1 schon genannt: virtuelle Maschinen oder Prozesse.

Quelle: Das Microservice-Praxisbuch, Wolff, S. 62ff.