ionos.de

Docker-Image

20-24 Minuten

Das Docker-Projekt mit der gleichnamigen Software hat sich als Standard für die Container-Virtualisierung etabliert. Ein zentrales Konzept bei der Nutzung der Docker-Plattform ist das Docker-Image. In diesem Artikel erklären wir, wie Docker-Images aufgebaut sind und wie sie funktionieren.

Inhaltsverzeichnis

Was ist ein Docker-Image?

Der Begriff „**Image**“ ist Ihnen im Zusammenhang mit Virtualisierung möglicherweise von virtuellen Maschinen (VMs) bekannt. Für gewöhnlich handelt es sich bei einem VM-Image um eine Kopie eines Betriebssystems. Ggf. enthält ein VM-Image weitere installierten Komponenten wie Datenbank und Webserver. Der Begriff entstammt einer Zeit, in der Software auf optischen Datenträgern wie CD-ROMs und DVDs verteilt wurde. Wollte man eine lokale Kopie des Datenträgers anlegen, erstellte man mit einer speziellen Software ein Abbild, auf Englisch „Image“.

Bei der Container-Virtualisierung handelt es sich um die konsequente Weiterentwicklung der VM-Virtualisierung. Anstatt einen virtuellen Computer (Maschine) mit eigenem Betriebssystem zu virtualisieren, umfasst ein **Docker-Image in der Regel lediglich eine Anwendung**. Dabei kann es sich um eine einzelne Binärdatei handeln oder um einen Verbund mehrerer Software-Komponenten.

Um die Anwendung auszuführen, wird aus dem Image zunächst ein Container erzeugt. Alle auf einem Docker-Host laufenden Container greifen auf denselben Betriebssystem-Kernel zurück. Dadurch sind Docker-Container und **Docker-Images in der Regel deutlich leichtgewichtiger** als vergleichbare virtuelle Maschinen und deren Images.

Die Konzepte Docker-Container und Docker-Image sind eng miteinander verknüpft. So kann nicht nur ein **Docker-Container aus einem Docker-Image erzeugt** werden, sondern auch aus einem laufenden Container ein neues Image. Docker-Image und Docker-Container stehen also in einem ähnlichen Zusammenhang wie Hühnerei und Huhn:

Docker-Befehl	**Bedeutung**	**Henne-Ei-Analogie**
docker run <image-id>	Docker-Container aus Image erzeugen	Küken schlüpft aus Ei
docker commit <container-id>	Docker-Image aus Container erzeugen	Henne legt neues Ei

Im biologischen Henne-Ei-System wird aus einem Ei genau ein Küken erzeugt. Das Ei geht dabei verloren. Im Gegensatz dazu lassen sich **aus einem Docker-Image beliebig viele gleichartige Container erzeugen**. Aus dieser Reproduzierbarkeit ergibt sich die Eignung von Docker als Plattform für skalierbare Anwendungen und Dienste.

Ein Docker-Image ist eine unveränderliche Vorlage, aus der Docker-Container erzeugt werden. Das Image enthält sämtliche **Informationen und Abhängigkeiten, um einen Container auszuführen**. Dazu gehören grundlegende Programm-Bibliotheken und Nutzer-Schnittstellen. Insbesondere ist normalerweise eine Kommandozeilen-Umgebung („Shell“) und eine Implementierung der C-Standard-Bibliothek mit an Bord. Hier eine Übersicht des offiziellen „Alpine-Linux“ Image:

Linux-Kernel	**C-Standard-Bibliothek**	**Unix-Kommandos**
vom Host	musl libc	BusyBox

Neben diesen grundlegenden Komponenten, die den Linux-Kernel

ergänzen, enthält ein Docker-Image in der Regel weitere Software. Hier einige Beispiele von **Software-Komponenten für verschiedene Einsatzgebiete**. Beachten Sie, dass ein einzelnes Docker-Image für gewöhnlich eine kleine Auswahl der gezeigten Komponenten umfasst:

Einsatzgebiet	Software-Komponenten
Programmiersprachen	PHP, Python, Ruby, Java, JavaScript
Entwicklungs-Tools	node/npm, React, Laravel
Datenbank-Systeme	MySQL, Postgres, MongoDB, Redis
Webserver	Apache, nginx, lighttpd
Caches und Proxys	Varnish, Squid
Content-Management-Systeme	WordPress, Magento, Ruby on Rails

Was unterscheidet ein Docker-Image von einem Docker-Container?

Wie wir gesehen haben, **existieren Docker-Image und Docker-Container im engen Wechselspiel** miteinander. In welchen Merkmalen unterscheiden sich die beiden Konzepte nun?

Zunächst stellen wir fest, dass ein Docker-Image inert ist. Es belegt lediglich etwas Speicherplatz, verbraucht jedoch ansonsten keinerlei Systemressourcen. Ferner ist ein **Docker-Image nach der Erstellung unveränderlich**. Es handelt sich also um ein „read-only“ bzw. schreibgeschütztes Medium. Dazu ein kleiner Hinweis: Ja, es ist möglich, einem existierenden Docker-Image eine Änderung hinzuzufügen. Dabei entsteht jedoch ein neues Image; das ursprüngliche Image liegt weiterhin in der unveränderten Version vor.

Wie erwähnt lassen sich **aus einem Docker-Image beliebig viele, gleichartige Container erzeugen**. Was genau macht nun einen Docker-Container im Vergleich zum -Image aus? Bei einem Docker-Container handelt es sich um eine laufende (soll heißen: in

der Ausführung befindliche) Instanz eines Docker-Images. Wie jede auf einem Computer ausgeführte Software, verbraucht ein laufender Docker-Container die Systemressourcen Arbeitsspeicher und CPU-Zyklen. Ferner ändert sich der Zustand eines Containers über dessen Lebensdauer.

Falls diese Beschreibung Ihnen zu abstrakt erscheint, ziehen Sie **ein einfaches Beispiel aus dem täglichen Leben** zu Hilfe. Stellen Sie sich ein Docker-Image wie eine DVD vor. Die DVD an sich ist inert – sie liegt in ihrer Hülle und macht ansonsten nichts. Dabei belegt sie dauerhaft denselben, begrenzten Platz im Raum. Erst durch Abspielen in einer speziellen Umgebung (DVD-Player) wird der Inhalt „lebendig“.

Wie der beim Abspielen einer DVD erzeugte Film hat ein **laufender Docker-Container einen Zustand**. Beim Film sind dies der aktuelle Wiedergabe-Zeitpunkt, gewählte Sprache, Untertitel und dergleichen. Dieser Zustand ändert sich mit der Zeit, und ein spielender Film verbraucht dabei laufend Elektrizität. Wie aus einem Docker-Image beliebig oft gleichartige Container gestartet werden können, lässt sich der Film einer DVD immer wieder abspielen. Ferner kann der laufende Film gestoppt und wieder gestartet werden – auch dies ist mit einem Docker-Container möglich.

Docker-Konzept	Analogie	Modus	Zustand	Ressourcenverbrauch
Docker-Image	DVD	inert	„read-only“ / unveränderlich	fest
Docker-Container	„lebendig“	spielender Film	ändert sich über die Zeit	variabel nach Nutzung

Wie und wo werden Docker-Images verwendet?

Docker kommt heutzutage in **allen Phasen des Software-Lifecycles** zum Einsatz. Dazu gehören Entwicklung, Test und Betrieb. Das zentrale Konzept im Docker-Ökosystem ist der Container, zu dessen Erzeugung wir immer ein Image benötigen. Docker-Images werden also überall dort verwendet, wo Docker zum Einsatz kommt. Schauen wir uns ein paar Beispiele an:

Docker-Images in der lokalem Entwicklungsumgebung

Entwickelt man Software auf dem eigenen Gerät, möchte man die lokale Entwicklungsumgebung so konsistent wie möglich halten. Meist benötigt man **exakt übereinstimmende Versionen von Programmiersprache, Bibliotheken und weiteren Software-Komponenten**. Sobald sich nur eine der vielen interagierenden Ebenen ändert, ergeben sich schnell Störeffekte. Dann kompiliert der Quelltext nicht oder der Webserver lässt sich nicht starten. Hier wirkt die Unveränderlichkeit der Docker-Images Wunder: Als Entwickler kann man sich darauf verlassen, dass die im Image enthaltene Umgebung konsistent bleibt.

Größere Entwicklungsprojekte werden von Teams geleistet. Hier ist die Nutzung einer über die Zeit stabilen Umgebung **Voraussetzung für Vergleichbarkeit und Reproduzierbarkeit**. Alle Entwickler eines Teams greifen auf dasselbe Image zurück. Kommt ein Entwickler neu ins Team, zieht er sich das passende Docker-Image und kann sofort loslegen. Um Änderungen an der Entwicklungsumgebung vorzunehmen, wird einmalig ein neues Docker-Image erstellt. Die Entwickler beziehen das neue Image und sind damit sofort auf dem aktuellen Stand.

Docker-Images in der serviceorientierten Architektur (SOA)

Docker-Images bilden die Grundlage der modernen serviceorientierten Architektur. Anstatt einer einzigen, monolithischen Anwendung werden **einzelne Dienste mit wohldefinierten Schnittstellen** entwickelt. Jeder Dienst wird in ein eigenes Image gepackt. Die daraus gestarteten Container kommunizieren miteinander über das Netzwerk und stellen somit die Gesamtfunktionalität der Anwendung her. Die Kapselung in eigenen Docker-Images erlaubt die voneinander unabhängige Entwicklung und Wartung der Dienste. Die einzelnen Dienste können sogar in verschiedenen Programmiersprachen geschrieben werden.

Docker-Images für Hosting-Provider / PaaS

Auch im Datencenter finden Docker-Images Verwendung. Jeder Dienst wie Load-Balancer, Webserver, Datenbankserver etc. ist als Docker-Image definiert. Die daraus erzeugten Container können jeweils eine gewisse Last stemmen. Eine **Orchestrator-Software**

überwacht die Container, deren Auslastung und Zustand. Bei steigender Last startet der Orchestrator zusätzliche Container aus dem entsprechenden Image. Dieser Ansatz erlaubt die schnelle Skalierung von Diensten als Reaktion auf sich ändernde Bedingungen.

Wie ist ein Docker-Image aufgebaut?

Anders als bei Images virtueller Maschinen der Fall, ist ein Docker-Image **im Normalzustand keine einzelne Datei**. Stattdessen handelt es sich um einen Verbund mehrerer Komponenten. Hier eine kurze Übersicht; Details folgen im weiteren Verlauf:

- **Image-Layers** enthalten durch Operation auf dem Dateisystem hinzugefügte Daten. Layers werden überlagert und durch ein Union-Dateisystem auf eine konsistente Ebene reduziert.
- Ein **Parent-Image** stellt Grundfunktionen des Images bereit und verankert das Image im Stammbaum des Docker-Ökosystems.
- Ein **Image-Manifest** beschreibt den Verbund und identifiziert die darin enthaltenen Image-Layers.

Was nun, wenn man ein **Docker-Image in eine einzelne Datei überführen** möchte? Auch dies ist mit dem „docker save“-Befehl auf der Kommandozeile möglich. Dabei wird eine tar-Archiv-Datei erzeugt. Diese lässt sich ganz normal zwischen Systemen bewegen. Mit dem folgenden Befehl wird ein Docker-Image mit Namen „busybox“ in eine Datei „busybox.tar“ geschrieben:

```
docker save busybox > busybox.tar
```

Häufig wird die Ausgabe von „docker save“ auf der **Kommandozeile per Pipeline an Gzip übergeben**. So werden die Daten nach Ausgabe in die tar-Datei komprimiert:

```
docker save myimage:latest | gzip >
myimage_latest.tar.gz
```

Eine via „docker save“ erzeugte Image-Datei lässt sich mit „docker load“ als **Docker-Image in den lokalen Docker-Host einspeisen**:

Image-Layers

Ein Docker-Image besteht aus schreibgeschützten Schichten, auf Englisch „Layers“ genannt. Jeder **Layer beschreibt sukzessive Änderungen am Dateisystem des Images**. Für jede Operation,

die zu einer Änderung am Dateisystem des Images führen würde, wird ein neuer Layer angelegt. Der dabei zum Einsatz kommende Ansatz wird generell als „Copy-on-Write“ bezeichnet: Ein Schreibzugriff legt eine veränderte Kopie in einem neuen Layer an; die Ursprungsdaten bleiben unverändert. Falls Ihnen das Prinzip bekannt vorkommt: Die Versionskontroll-Software Git arbeitet nach demselben Muster.

Wir können die **Layers eines Docker-Image anzeigen**. Wir nutzen hier den „docker image inspect“-Befehl auf der Kommandozeile. Der Aufruf gibt ein JSON-Dokument zurück, das wir mit dem Standard-Tool jq verarbeiten:

```
docker image inspect <image-id> | jq -r
'.[].RootFS.Layers[]'
```

Um die in den Layers enthaltenen Änderungen wieder zusammenzuführen, wird ein spezielles Dateisystem verwendet. Dieses **Union-Dateisystem überlagert sämtliche Layers**, sodass sich auf der Oberfläche eine konsistente Ordner- und Dateistruktur ergibt. Historisch kamen verschiedene, als „Storage-Driver“ bezeichnete Technologien zum Einsatz, um das Union-Dateisystem zu implementieren. Heutzutage wird für die meisten Anwendungsfälle der Storage-Driver overlay2 empfohlen:

Storage-Driver	**Kommentar**
overlay2	heutzutage empfohlen
aufs, overlay	in früheren Versionen verwendet

Es ist möglich, den **verwendeten Storage-Driver eines Docker-Image auszugeben**. Wir nutzen hier den „docker image inspect“-Befehl auf der Kommandozeile. Der Aufruf gibt ein JSON-Dokument zurück, das wir mit dem Standard-Tool jq verarbeiten:

```
docker image inspect <image-id> | jq -r
'.[].GraphDriver.Name'
```

Jeder **Image-Layer ist durch einen eindeutigen Hash gekennzeichnet**, der aus den darin enthaltenen Änderungen errechnet wird. Nutzen zwei Images denselben Layer, wird dieser nur einmal lokal gespeichert. Beide Images greifen dann auf denselben Layer zurück. Dies führt zu einer effizienten lokalen Speicherung und reduzierten Transfermengen beim Beziehen von

Images.

Parent-Image

Einem Docker-Image liegt für gewöhnlich ein „Parent-Image“ (zu Deutsch „Eltern-Image“) zugrunde. In den meisten Fällen wird das Parent-Image durch eine FROM-Anweisung in der Dockerfile festgelegt. Das **Parent-Image definiert eine Basis, auf dem davon abgeleitete Images aufbauen**. Dabei werden die existierenden Image-Layers durch zusätzliche Layers überlagert.

Durch das „Erben“ vom Parent-Image ordnet sich ein Docker-Image in einen Stammbaum ein, der sämtliche existierenden Images umfasst. Vielleicht fragen Sie sich bereits, wo der Stammbaum seinen Anfang nimmt. Die **Wurzeln des Stammbaums werden durch wenige, spezielle „Base-Images“ festgelegt**. In den meisten Fällen wird ein Base-Image mit der „FROM scratch“-Anweisung in der Dockerfile definiert. Es gibt jedoch auch andere Wege, ein Base-Image zu erstellen. Dazu mehr im Abschnitt „Woher stammen Docker-Images?“.

Image-Manifest

Wie wir gesehen haben, besteht ein Docker-Image aus mehreren Layers. Nutzt man den „docker image pull“-Befehl, um ein Docker-Image von einer Online-Registry zu beziehen, wird also keine einzelne Image-Datei heruntergeladen. Stattdessen **lädt der lokale Docker-Daemon die einzelnen Layers herunter** und speichert diese. Woher stammt nun die Information über die einzelnen Layers?

Die Information darüber, aus welchen Image-Layers ein Docker-Image besteht, ist im sogenannten **Image-Manifest** enthalten. Das Image-Manifest ist eine JSON-Datei, die ein Docker-Image vollständig beschreibt. Unter anderem enthält ein Image-Manifest:

- Versions-, Schema- und Größenangaben
- kryptografische Hashes der zum Einsatz kommenden Image-Layers
- Angaben zu verfügbaren Prozessorarchitekturen

Um ein Docker-Image eindeutig zu identifizieren, wird ein **kryptografischer Hash des Image-Manifest** errechnet. Beim Aufrufen von „docker image pull“ wird zunächst die Manifest-Datei

heruntergeladen. Im Anschluss bezieht der lokale Docker-Deamon die einzelnen Image-Layers.

Woher stammen Docker-Images?

Wie wir gesehen haben, bilden Docker-Images einen wichtigen Teil des Docker-Ökosystems. Es gibt **vielfältige Möglichkeiten, ein Docker-Image zu erlangen**. Wir unterscheiden zwei grundlegende Wege, deren Ausprägungen wir uns im Folgenden anschauen:

1. **Existierendes Docker-Image** von Registry beziehen
2. **Neues Docker-Image** erzeugen

Existierendes Docker-Image von Registry beziehen

Oft beginnt ein Docker-Projekt mit dem Schritt, ein existierendes Docker-Image von einer sogenannten Registry zu beziehen. Dabei handelt es sich um eine über das Netzwerk erreichbare **Plattform, die Docker-Images bereitstellt**. Der lokale Docker-Host kommuniziert mit der Registry, um in Folge eines „docker image pull"-Befehls ein Docker-Image herunterzuladen.

Zum einen gibt es öffentlich zugängliche Online-Registrys. Diese bieten eine große Auswahl existierender Docker-Image zur Nutzung an. Auf der offiziellen Docker-Registry „Docker-Hub" sind zum Zeitpunkt der Artikelerstellung **mehr als acht Millionen frei verfügbare Docker-Images** gelistet. Microsofts „Azure Container Registry" hält neben Docker-Images weitere Container-Images in verschiedenen Formaten vor. Ferner erlaubt die Plattform eigene, nichtöffentliche Container-Registrys anzulegen.

Neben den genannten Online-Registrys gibt es die Möglichkeit, selbst ein lokales Registry zu hosten. Darauf greifen z. B. größere Organisationen zurück, um den eigenen Teams **geschützten Zugriff auf selbst erstellte Docker-Images** zu geben. Von der Firma Docker wird für diesen Zweck das „Docker Trusted Registry" (DTR) angeboten. Dabei handelt es sich um eine On-Premises-Lösung für das Bereitstellen einer firmeninternen Registry im eigenen Rechenzentrum.

Neues Docker-Image erzeugen

Manchmal möchte man für ein Projekt ein **speziell angepasstes Docker-Image erstellen**. Für gewöhnlich greift man dabei auf ein

existierendes Docker-Image zurück und passt dieses nach Bedarf an. Erinnern wir uns, dass Docker-Images unveränderlich sind: Pflegen wir Änderungen ein, ist das Resultat ein neues Docker-Image. Zur Erzeugung eines neuen Docker-Images gibt es die folgenden Wege:

1. Mit Dockerfile auf **Parent-Image** aufbauen
2. **Aus laufendem Container** erzeugen
3. **Neues Base-Image** erstellen

Der wohl häufigste Ansatz zur Erstellung eines neuen Docker-Image besteht darin, ein Dockerfile zu schreiben. Das Dockerfile enthält spezielle Anweisungen, die das **Parent-Image und sämtliche benötigten Änderungen festlegen**. Mit Aufruf von „docker image build“ wird das neue Docker-Image aus dem Dockerfile erzeugt. Hier ein minimales Beispiel:

```
# Dockerfile auf der Kommandozeile erzeugen
cat <<EOF > ./Dockerfile
FROM busybox
RUN echo "hello world"
EOF

# Docker-Image aus Dockerfile erzeugen
docker image build
```

Historisch stammt der Begriff „Image“ vom „Abbilden“ eines Datenträgers (auf Englisch „to image“). Im Kontext virtueller Maschinen (VM) lässt sich aus einem laufenden VM-Image ein Schnappschuss erstellen. Ähnlich ist dies auch mit Docker möglich. Mit dem „docker commit“-Befehl legen wir ein **Abbild eines laufenden Containers als neues Docker-Image** an. Sämtliche im Container vorgenommen Änderungen werden dabei gespeichert:

```
docker commit <container-id>
```

Ferner haben wir dir Möglichkeit, dem „docker commit“-Befehl **Dockerfile-Anweisungen zu übergeben**. Die mit den Anweisungen kodierten Änderungen werden Teil des neuen Docker-Image:

```
docker commit --change <dockerfile instructions>
<container-id>
```

Um im Nachhinein nachvollziehen zu können, **welche**

Änderungen zu einem Docker-Image geführt haben, nutzen wir den „docker image history“-Befehl:

```
docker image history <image-id>
```

Wie wir gesehen haben, basieren wir ein neues Docker-Image auf einem Parent-Image oder Zustand eines laufenden Containers. Was jedoch, wenn man für **ein neues Docker-Image bei null anfangen** möchte? In diesem Fall bieten sich zwei Wege. Man kann wie oben beschrieben eine Dockerfile mit der speziellen „FROM scratch“-Anweisung nutzen. Heraus kommt dabei ein neues, minimales Base-Image.

Möchte man sogar noch auf Dockers scratch-Image verzichten, nutzt man ein spezielles Tool wie debootstrap und **präpariert damit eine Linux-Distribution**. Diese wird im Anschluss per tar-Befehl in ein „Tarball“-Archiv verpackt und via „docker image import“ in den lokalen Docker-Host importiert.

Die wichtigsten Docker-Image Befehle

Docker-Image Befehl	Erklärung
docker image build	Docker-Image aus Dockerfile erzeugen
docker image history	Schritte zur Erzeugung eines Docker-Image anzeigen
docker image import	Docker-Image aus Tarball-Archiv erzeugen
docker image inspect	Detaillierte Informationen für Docker-Image anzeigen
docker image load	Mit „docker image save“ erzeugtes Image-Archiv laden
docker image ls / docker images	Auf dem Docker-Host verfügbare Images auflisten
docker image prune	Ungenutzte Docker-Images vom Docker-Host entfernen
docker image pull	Docker-Image von Registry beziehen

Docker-Image Befehl	**Erklärung**
docker image push	Docker-Image an Registry senden
docker image rm	Docker-Image vom lokalen Docker-Host entfernen
docker image save	Image-Archiv erzeugen
docker image tag	Docker-Image mit Tag versehen