

Handlungsziele

Modul 169 Services mit Containern bereitstellen

✓ Definiert die erforderliche Umgebung für die automatisierte Bereitstellung von Services. ✓

1. Kennt die Grundprinzipien der monolithischen und der Microservice-Architektur.
2. Kennt die wesentlichen Vor- und Nachteile unterschiedlicher Architekturen und deren Nutzen auf Serviceebene aufzeigen (z.B. Microservice-Architektur).
3. Kennt die Vorgehensweise, wie Services in Container verpackt werden.
4. Kennt die Vorgehensweise, wie Services im Backend bereitgestellt und wie die Services im Frontend von Clients genutzt werden.
5. Kennt die einer Containerumgebung zugrundeliegende Architektur (Daemon, Client/Server, Images, Container, Registry).
6. Kennt die Abhängigkeiten von Services und deren Bereitstellung in einer lokalen Infrastruktur (Beispiele: persistente Datenspeicherung, Vernetzung und andere).

✓ Dokumentiert den logischen und physischen Aufbau der Umgebung in einem Netzwerkschema mit servicespezifischen Angaben. ✓

1. Kennt die erforderliche Dokumentation der Services mit den Standard-Netzwerksymbolen und der Unterscheidung, zwischen logischer und physischer Architektur.
2. Kennt die Netzwerkkonfiguration der Systeme (IP-Adressierung, Subnetzmaske, Standardgateway, DNS, Serviceport-Angaben) und deren Abhängigkeiten.
3. Kennt die unterschiedlichen Darstellungsarten, welche für Dokumentation verwendet werden können (Blockschaltbild, logischer Netzwerkplan, Schichtenmodell für Kommunikation und Systeme) und setzt situativ die sinnvollste Darstellungsart ein.

✓ **Erstellt und dokumentiert den für die Service-Breitstellung erforderlichen Code versioniert.** ✓

1. Kennt Tools mit Syntaxunterstützung für die Code Erstellung.
2. Kennt den Nutzen und Einsatzzweck von Versionsverwaltungssystemen und setzt diese situativ korrekt ein, sodass der erstellte Code versioniert und kommentiert verfügbar ist.
3. Kennt die Vorgehensweise, wie Systeme und Services codebasiert (Infrastructure as Code - IaC) aufgebaut werden.
4. Kennt den Lebenszyklus (Erstellen – Editieren – Testen – Einsetzen – Verwerfen).
5. Kennt die Vorgehensweise, wie die erstellen Services getestet werden können.
6. Kennt Methoden, mit welchen die Codekonsistenz überprüft werden kann.

✓ **Plant und realisiert die servicespezifischen Sicherheitsanforderungen.** ✓

1. Kennt die Sicherheitsmassnahmen und Methoden, um einen sicheren Betrieb der Containerumgebung zu gewährleisten.
2. Kennt die Vorgehensweise, wie definierten Sicherheitsmassnahmen in die Dokumentation transferiert und dargestellt werden kann.
3. Kennt Methoden, mit welchen die Sicherheitsmassnahmen auf die Services angewendet werden können.
4. Kennt die Wichtigkeit der Sicherheitsmassnahmen und überprüft die Wirksamkeit mit Hilfe eines Testkonzepts.

✓ **Erstellt die erforderlichen Datenverbindungen zwischen unterschiedlichen Services.** ✓

1. Kennt die erforderlichen Massnahmen, um die Datenverbindung zwischen Services herzustellen (Beispiel: Portfreigaben, Berechtigungen, Authentifizierung).
2. Kennt die Möglichkeiten des systemübergreifenden Datenaustauschs über definierte Schnittstellen (Beispiele: Synchrone und Asynchrone Kommunikation zwischen Services, Beispiele: REST Paradigma, Message Bus, API Gateway, Monitoring).
3. Kennt die Vorgehensweise und Zusammenhänge der zentralen und persistenten Datenspeicherung.

✓ **Stellt die Services in der definierten Umgebung reproduzierbar bereit.**



1. Kennt die Vorgehensweise, wie Container servicebezogen verwaltet werden (Starten, Funktionsfähigkeit überprüfen Stoppen, Löschen).
2. Kennt die Möglichkeit, Images von Containern in Registries für die weitere Verwendung verfügbar zu machen.

✓ **Administriert und überwacht die bereitgestellten Services.**



1. Kennt die erforderlichen Massnahmen, um die Funktionalität der Services und den zugrundeliegenden Containern automatisiert zu überwachen.
2. Kennt die Möglichkeiten, wie die verfügbaren Ressourcen gemanagt (Beschränkung von Containern) werden können und definiert die nötigen Massnahmen mit Hilfe einer entsprechenden Berechnung aufgrund von konkreten Annahmen.

✓ **Versteht anhand der Dokumentation die Funktionalität der Services und unterstützt bei der Fehlersuche.**



1. Kennt die Zusammenhänge in einer containerbasierten Infrastruktur.
2. Kennt die Grundfunktionalität der Tools, welche die Fehlersuche in Containern unterstützen.

Handlungsziele

Modul 169 Services mit Containern bereitstellen

✓ Definiert die erforderliche Umgebung für die automatisierte Bereitstellung von Services. ✓

1. Kennt die Grundprinzipien der monolithischen und der Microservice-Architektur.
2. Kennt die wesentlichen Vor- und Nachteile unterschiedlicher Architekturen und deren Nutzen auf Serviceebene aufzeigen (z.B. Microservice-Architektur).
3. Kennt die Vorgehensweise, wie Services in Container verpackt werden.
4. Kennt die Vorgehensweise, wie Services im Backend bereitgestellt und wie die Services im Frontend von Clients genutzt werden.
5. Kennt die einer Containerumgebung zugrundeliegende Architektur (Daemon, Client/Server, Images, Container, Registry).
6. Kennt die Abhängigkeiten von Services und deren Bereitstellung in einer lokalen Infrastruktur (Beispiele: persistente Datenspeicherung, Vernetzung und andere).

✓ Dokumentiert den logischen und physischen Aufbau der Umgebung in einem Netzwerkschema mit servicespezifischen Angaben. ✓

1. Kennt die erforderliche Dokumentation der Services mit den Standard-Netzwerksymbolen und der Unterscheidung, zwischen logischer und physischer Architektur.
2. Kennt die Netzwerkkonfiguration der Systeme (IP-Adressierung, Subnetzmaske, Standardgateway, DNS, Serviceport-Angaben) und deren Abhängigkeiten.
3. Kennt die unterschiedlichen Darstellungsarten, welche für Dokumentation verwendet werden können (Blockschaltbild, logischer Netzwerkplan, Schichtenmodell für Kommunikation und Systeme) und setzt situativ die sinnvollste Darstellungsart ein.

✓ **Erstellt und dokumentiert den für die Service-Breitstellung erforderlichen Code versioniert.** ✓

1. Kennt Tools mit Syntaxunterstützung für die Code Erstellung.
2. Kennt den Nutzen und Einsatzzweck von Versionsverwaltungssystemen und setzt diese situativ korrekt ein, sodass der erstellte Code versioniert und kommentiert verfügbar ist.
3. Kennt die Vorgehensweise, wie Systeme und Services codebasiert (Infrastructure as Code - IaC) aufgebaut werden.
4. Kennt den Lebenszyklus (Erstellen – Editieren – Testen – Einsetzen – Verwerfen).
5. Kennt die Vorgehensweise, wie die erstellen Services getestet werden können.
6. Kennt Methoden, mit welchen die Codekonsistenz überprüft werden kann.

✓ **Plant und realisiert die servicespezifischen Sicherheitsanforderungen.** ✓

1. Kennt die Sicherheitsmassnahmen und Methoden, um einen sicheren Betrieb der Containerumgebung zu gewährleisten.
2. Kennt die Vorgehensweise, wie definierten Sicherheitsmassnahmen in die Dokumentation transferiert und dargestellt werden kann.
3. Kennt Methoden, mit welchen die Sicherheitsmassnahmen auf die Services angewendet werden können.
4. Kennt die Wichtigkeit der Sicherheitsmassnahmen und überprüft die Wirksamkeit mit Hilfe eines Testkonzepts.

✓ **Erstellt die erforderlichen Datenverbindungen zwischen unterschiedlichen Services.** ✓

1. Kennt die erforderlichen Massnahmen, um die Datenverbindung zwischen Services herzustellen (Beispiel: Portfreigaben, Berechtigungen, Authentifizierung).
2. Kennt die Möglichkeiten des systemübergreifenden Datenaustauschs über definierte Schnittstellen (Beispiele: Synchrone und Asynchrone Kommunikation zwischen Services, Beispiele: REST Paradigma, Message Bus, API Gateway, Monitoring).
3. Kennt die Vorgehensweise und Zusammenhänge der zentralen und persistenten Datenspeicherung.

✓ **Stellt die Services in der definierten Umgebung reproduzierbar bereit.**



1. Kennt die Vorgehensweise, wie Container servicebezogen verwaltet werden (Starten, Funktionsfähigkeit überprüfen Stoppen, Löschen).
2. Kennt die Möglichkeit, Images von Containern in Registries für die weitere Verwendung verfügbar zu machen.

✓ **Administriert und überwacht die bereitgestellten Services.**



1. Kennt die erforderlichen Massnahmen, um die Funktionalität der Services und den zugrundeliegenden Containern automatisiert zu überwachen.
2. Kennt die Möglichkeiten, wie die verfügbaren Ressourcen gemanagt (Beschränkung von Containern) werden können und definiert die nötigen Massnahmen mit Hilfe einer entsprechenden Berechnung aufgrund von konkreten Annahmen.

✓ **Versteht anhand der Dokumentation die Funktionalität der Services und unterstützt bei der Fehlersuche.**



1. Kennt die Zusammenhänge in einer containerbasierten Infrastruktur.
2. Kennt die Grundfunktionalität der Tools, welche die Fehlersuche in Containern unterstützen.

Begriffe und Konzepte

Virtuelle Maschinen vs. Container

Das folgende Bild veranschaulicht den Unterschied von klassischen virtuellen Maschinen zu Containern

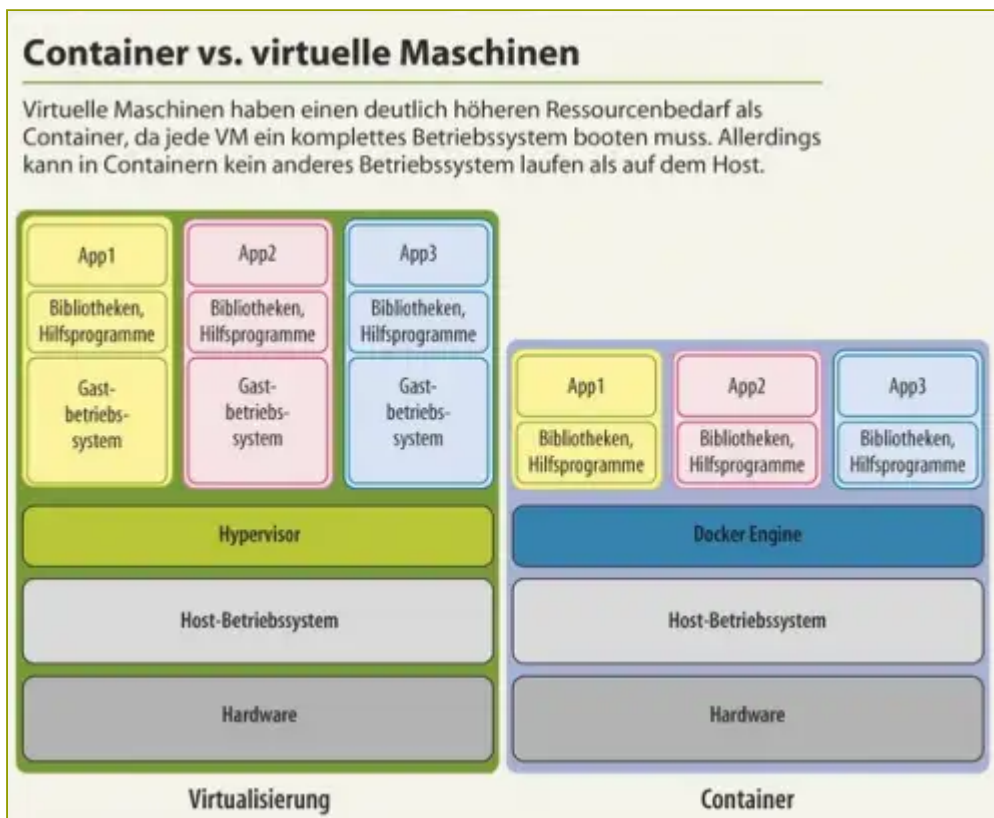


Abb. 1: Virtuelle Maschinen vs. Container

Während in jeder virtuellen Maschine ein vollständiges Betriebssystem läuft, bringt ein Container nur die wirklich benötigten Komponenten ohne Betriebssystem mit. Die Container verwenden alle denselben Kernel, nämlich denjenigen des Host-Rechners. Dies spart natürlich eine Menge Ressourcen, es können sehr viel mehr Container auf einem Host laufen als klassische virtuelle Maschinen.

Images

Ein Image enthält alle benötigten Komponenten, inkl. Bibliotheken, Hilfsprogramme und sonstige Dateien für den Betrieb. Es gibt viele vorgefertigte Images, die aus einer Reihe von Registries heruntergeladen werden, z.B. von Amazon, Google, RedHat oder Microsoft. Die

bedeutendste Registry ist jedoch ohne Zweifel die [Docker Hub Registry](#) selber mit über 100'000 Images. Darunter sind natürlich viele veraltete, nicht mehr unterhaltenes oder zu Testzwecken verwendete, aber genauso viele offizielle und von verifizierten Anbietern gut gewartete Images.

Daneben können aber auch eigene Images an ein konkretes Projekt angepasste Images erstellt werden. Solche Images können wiederum in den Docker Hub oder eine andere Registry hochgeladen werden.

Container

Ein Image bleibt nach dem Download immer unverändert. Das Image ist quasi schreibgeschützt. Ein Image kann auch nicht gestartet werden, sondern es wird aus einem Image heraus einer (oder mehrere) Container gestartet. Schreiboperationen des Containers landen in einem Overlay-Dateisystem, welches "über" dem Dateisystem des Images liegt. Das folgende Bild veranschaulicht diesen Sachverhalt.

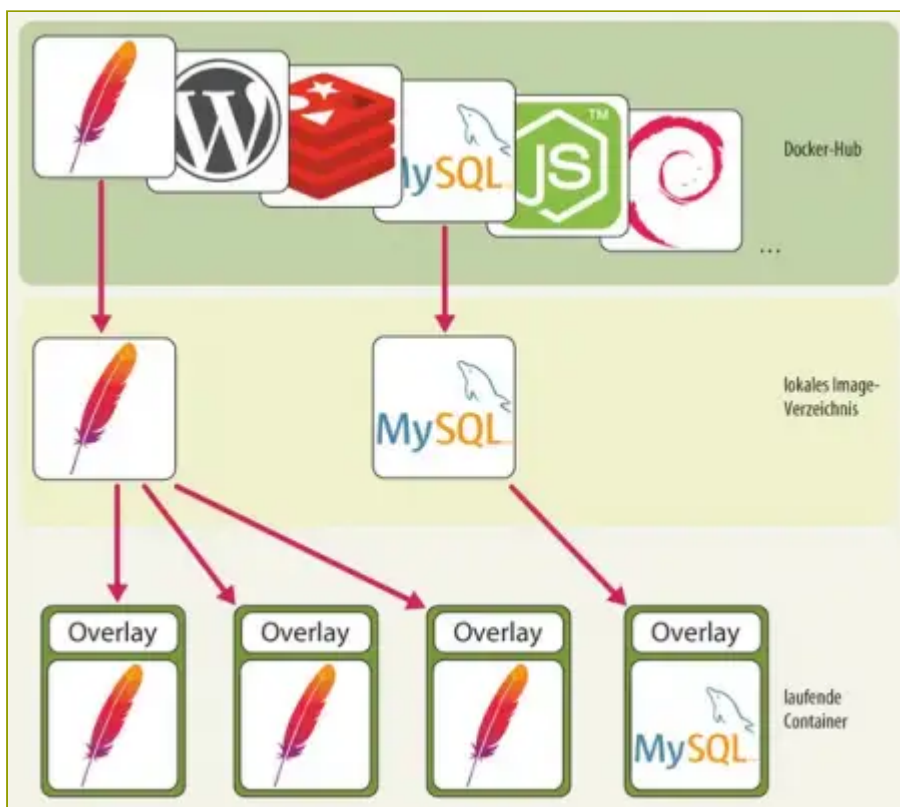


Abb. 2: Image und Container

Microservices

Im Gegensatz zu klassischen Applikationen hat sich im Containerumfeld die Devise "eine Aufgabe - ein Container" etabliert. Jeder Container übernimmt eine spezifische Aufgabe im Gegensatz zu monolithischen Anwendungen. Man spricht dann von Microservices. Die

Microservices können im Idealfall unabhängig voneinander entwickelt und in Betrieb genommen werden. Ein Microservice soll eine fachlich definierte Aufgabe übernehmen. Beispielsweise kann eine Applikation bestehend aus Apache, Java und Datenbank auf drei Container verteilt werden. Die eigentliche Applikation kann aber noch auf weitere funktional unabhängige Bestandteile aufgeteilt werden, z.B. ein Microservice für das Web-Frontend, einer für das Backend, ein weiterer Service für das Forum, den Adminbereich oder ein Service für eine optimierte Website für Mobilgeräte. In der folgenden Abbildung sehen Sie vier Microservices.

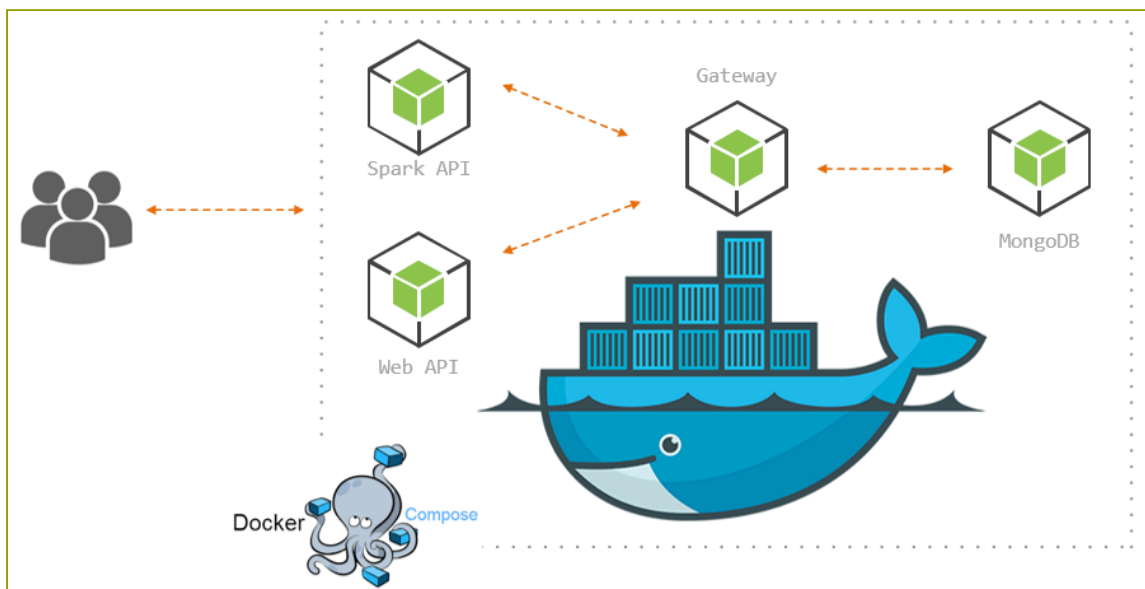


Abb. 3: Microservices

Der koordinierte Start und Stop aller zusammengehöriger Microservices wird von Docker-compose übernommen

Orchestrierung

Ist ein einzelner Microservice auf eine Skalierung, lassen sich bei Bedarf schnell mehrere Instanzen verteilt über mehrere Hosts zuschalten. An die Stelle einzelner Server, die der Admin wie ein Haustier hegt und pflegt, tritt eine Herde von Containern, die man je nach Bedarf vergrößert oder verkleinert. Einen zickenden Container schiesst man einfach ab und startet einen neuen. Dies wird als Orchestrierung bezeichnet. Als Tools kommen hier Docker Swarm oder Kubernetes zu Einsatz.

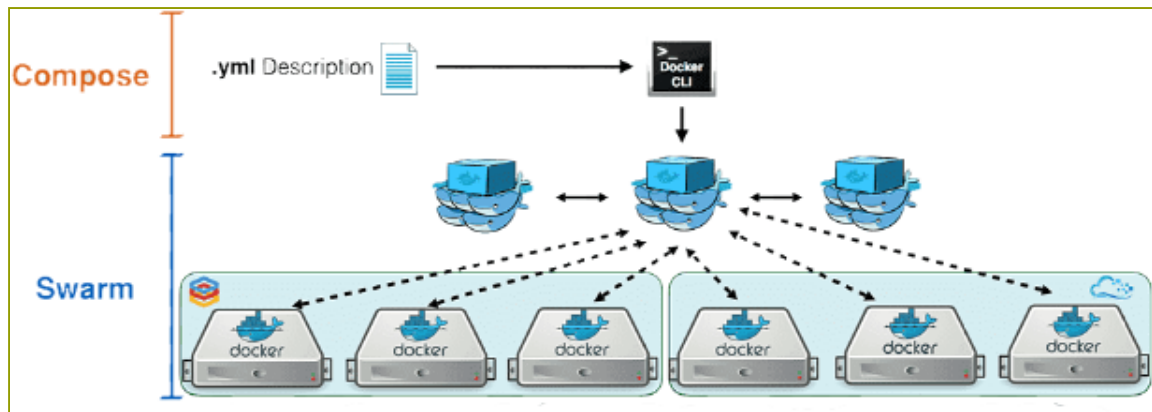


Abb. 4: Orchestrierung

Installation

Im folgenden wird die Installation von docker und docker compose unter Ubuntu beschrieben.

Internetzugang

Überprüfen Sie ob das Netzwerksymbol oben rechts wie abgebildet dargestellt wird. Ist dies der Fall sollte der Internetzugang gewährleistet sein.



Abb. 1: Internetzugang

Sollte dies nicht der Fall sein, starten Sie die virtuelle Maschine LF-2.25

Docker

Die Installation wird am Einfachsten als root durchgeführt. Wechseln deshalb zu einer root-Shell:

```
sudo su
```

Dann müssen die Paketquellen aktualisiert werden

```
apt update
```

Folgende Basispakete werden benötigt und installiert:

```
apt install apt-transport-https ca-certificates curl gpg
```

Docker wird direkt von docker.com installiert, da die üblichen Ubuntu Paketquellen für docker nicht immer aktuell sind. Dazu müssen zuerst die offiziellen Docker Schlüssel hinzugefügt werden:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Anschliessend können die offiziellen Docker Paketquellen hinzugefügt werden:

```
echo "deb [arch=amd64 \
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" > \
/etc/apt/sources.list.d/docker.list
```

Und nochmals

```
apt update
```

Nun kann docker wie gewöhnlich installiert werden

```
apt install docker-ce docker-ce-cli containerd.io
```

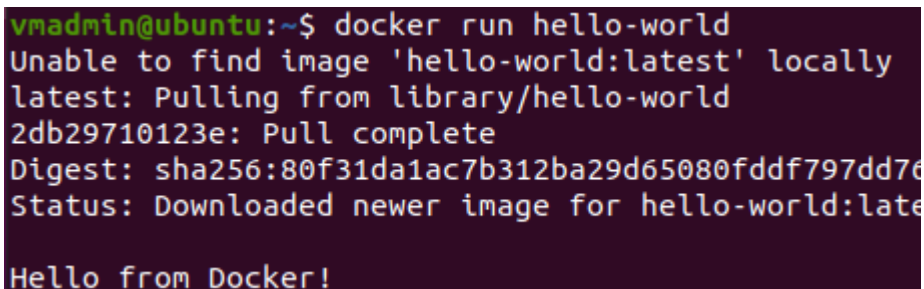
Fügen Sie zum Schluss den Benutzer vmadmin zur Gruppe docker hinzu:

```
usermod -aG docker vmadmin
```

Nun ist ein **Reboot** der Virtuellen Maschine nötig.

Nun können Sie Testen ob die Installation geklappt hat: Starten Sie das offizielle Hello-World Beispiel von Docker als **vmadmin** mit

```
docker run hello-world
```

A terminal window with a dark background. The prompt is 'vmadmin@ubuntu:~\$'. The command 'docker run hello-world' is entered. The output shows that the 'hello-world:latest' image was not found locally, so it was pulled from the library. It shows the pull progress, completion, digest, and status. Finally, it outputs 'Hello from Docker!'.

```
vmadmin@ubuntu:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:80f31da1ac7b312ba29d65080fddf797dd76
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

Abb. 2: Hello World

Visual Code mit Docker Extension

Auch hier wird die Software direkt aus den offiziellen Microsoft Repositories installiert. Dazu zuerst die Schlüssel herunterladen.

```
wget -q https://packages.microsoft.com/keys/microsoft.asc -O- | sudo apt-key
add -
```

dann das MS Repository hinzufügen

```
add-apt-repository "deb [arch=amd64]  
https://packages.microsoft.com/repos/vscode stable main"
```

und danach

```
apt install code
```

Nun können Sie Visual Code starten.

Suchen Sie unter Extensions nach der Microsoft docker extension und installieren Sie diese

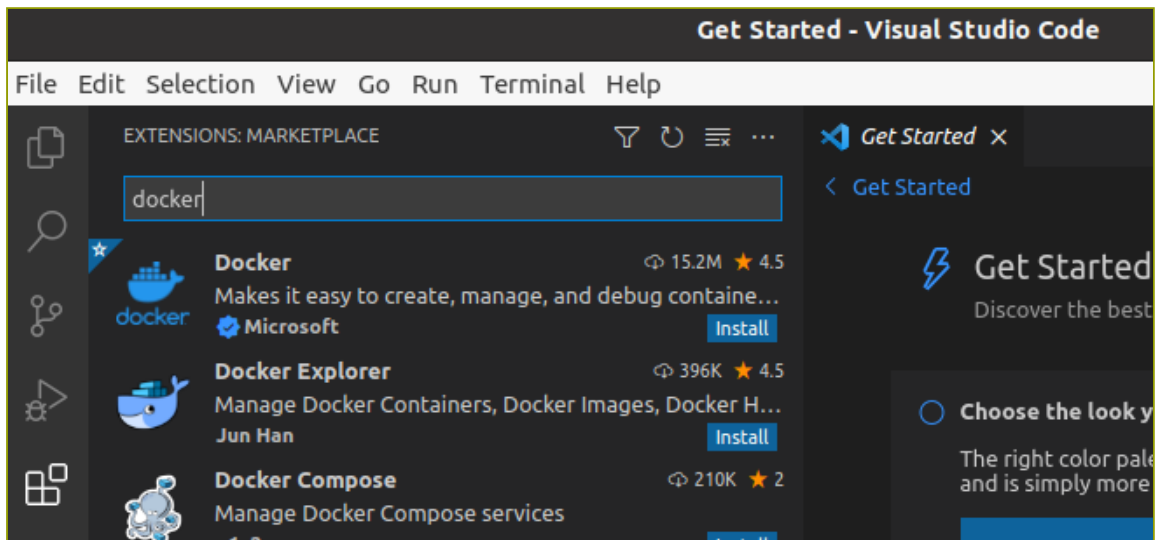
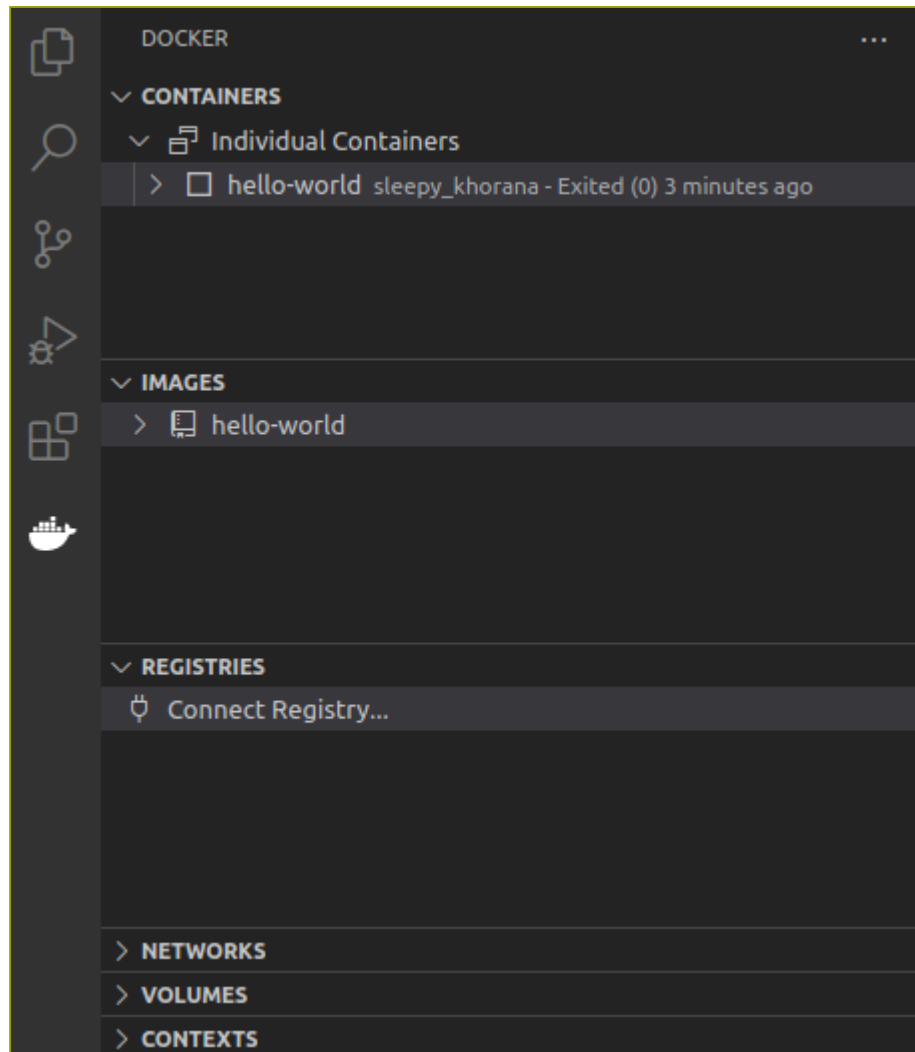


Abb. 3: MS Docker Extension Installation

Danach können Sie die Extension in Betrieb nehmen. Es erscheint eine Liste mit allen Containern,



Images, etc.

Abb. 4: MS Docker Extension Liste

Weitere Hilfe zur Extension finden Sie hier:

<https://code.visualstudio.com/docs/containers/overview>

Ein erstes Beispiel

Bei diesem Beispiel wird ein Image erstellt, welches einen Webserver startet und eine Seite ausliefert mit dem aktuellen Datum und Zeit.

Als Technologie verwenden wir Apache im Zusammenspiel mit php8.

Benötigte Dateien

Erstellen Sie im Homeverzeichnis von vmadmin ein neues Verzeichnis apache-php

```
mkdir bsp-apache-php
```

und wechseln Sie gleich ins neu erstellte Verzeichnis

```
cd bsp-apache-php
```

Erstellen Sie in diesem Verzeichnis eine neue Datei mit Namen Dockerfile und dem abgebildeten Inhalt:

```
# Datei: bsp-apache-php/Dockerfile
FROM php:8-apache
COPY index.php /var/www/html
```

Diese Datei gibt das Rezept an, wie ein Image erstellt wird. **FROM** gibt das verwendete Basisimage an. **COPY** kopiert die Datei index.php ins Verzeichnis /var/www/html im Container. Dieses Verzeichnis ist bei Apache das Standardverzeichnis für Webseiten, d.h. beim Aufruf der Webseite wird index.php aufgerufen von php bearbeitet und an den aufrufenden Browser ausgeliefert.

Die Datei index.php muss natürlich noch erstellt werden und hat folgenden Inhalt:

```
<!DOCTYPE html >
<!-- Datei index.php -->
<html >
<head >
<title >Beispiel</title >
<meta charset ="utf-8" />
</head >
<body >
<h1>Beispiel apache/php</h1>
Serverzeit : <?php echo date("j. F Y, H:i:s, e "); ?>
```

```
</body >
</html >
```

Image builden

Nun wird das Image mit folgendem Befehl erstellt:

```
docker build -t bsp-apache-php .
```

Der Schalter `-t` gibt den Namen des Images an, hier also `bsp-apache-php`. Üblicherweise startet man dieses Kommando aus dem Verzeichnis in dem sich das Dockerfile befindet. Somit gibt der Punkt im Kommando einfach an, dass Sie sich bereits in diesem Verzeichnis befinden. In zwei Schritten wird nun das Basisimage heruntergeladen und extrahiert und die Indexdatei kopiert.

Überprüfen Sie den Erfolg mit dem Kommando

```
docker image ls
```

Hier sollte (unter anderem) nun 2 Images aufgeführt werden:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
bsp-apache-php	latest	c5a049e80c58	About a minute ago	458MB
php	8-apache	af944036d594	2 days ago	458MB

Container starten

Nun kann aus dem Image ein Container gestartet werden:

```
docker run -d --name bsp-apache-php-container -p 8080:80 bsp-apache-php
```

Erklärung:

- ✓ **-d:** gibt an, dass der Container im Hintergrund läuft (daemon)
- ✓ **--name:** gibt en Namen des Containers an, dieser kann anders lauten als das Image
- ✓ **-p:** Verknüpft den Port 80 innerhalb des Containers mit dem Port 8080 des Hosts
- ✓ **bsp-apache-php:** der Name des Image aus dem der Container gestartet wird

Überprüfen Sie ob der Container läuft:

```
docker ps
```


CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
16d2fe2f443b	bsp-apache-php	"docker-php-entrypoi..."	3 minutes ago Up 3 minutes
0.0.0.0:8080->80/tcp, :::8080->80/tcp			bsp-apache-php-container

Nun können Sie die Webseite <http://localhost:8080> aufrufen:

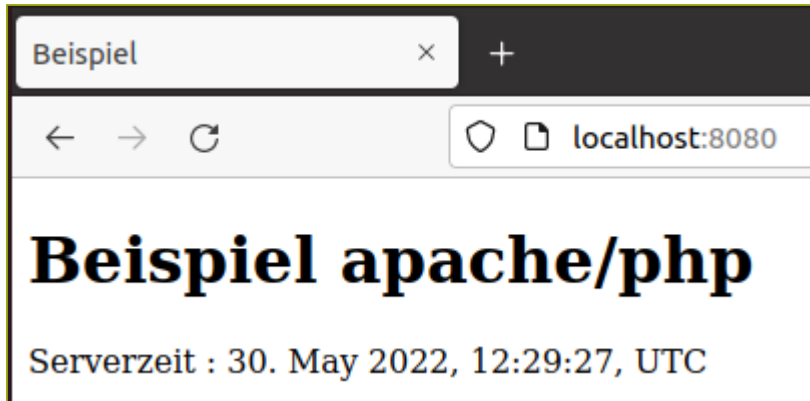


Abb. 1: Docker in Action

Sie können den Container mit stop, start oder restart anhalten und neu starten:

```
docker stop bsp-apache-php-container
```

```
docker start bsp-apache-php-container
```

```
docker restart bsp-apache-php-container
```

Der Container kann gelöscht werden mit

```
docker rm bsp-apache-php-container
```

Ein neuer vom Image abeiteter Container könnte nun wieder mit dem run Kommando gestartet werden.

Lokal vorhandene Images können Sie mit

```
docker images
```

anzeigen lassen.

Ein Image können Sie mit

```
docker rmi bsp-apache-php
```

wieder löschen.

```
docker rmi bsp-apache-php php:8-apache
```

löscht das zu Grunde liegende Basisimage gleich mit.

Grundlagen

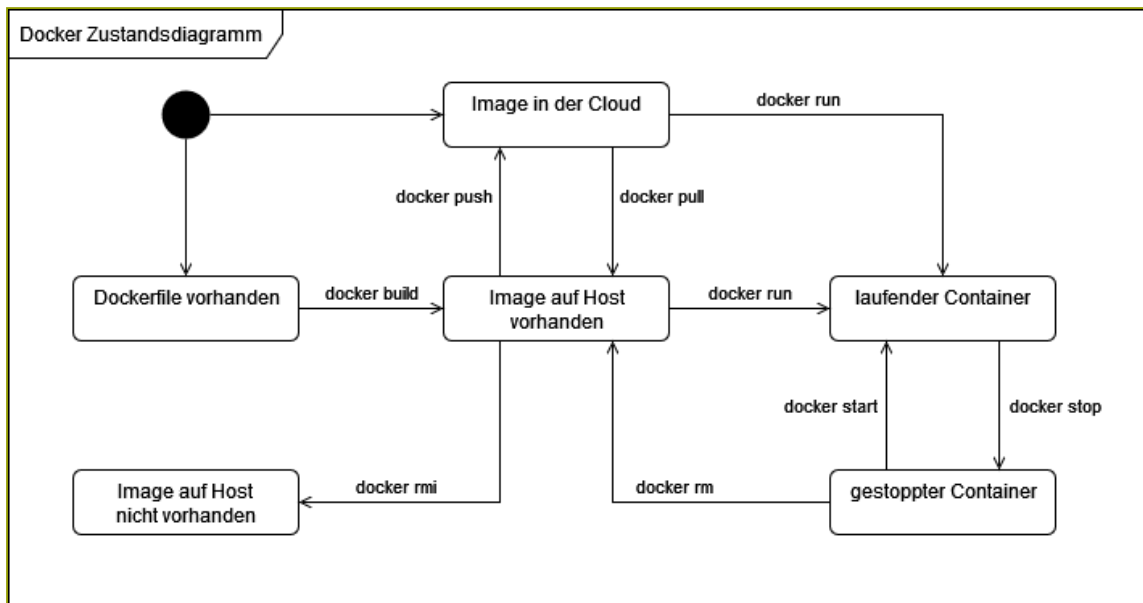


Abb. 1: Zustandsdiagramm für Docker

Portweiterleitung

Stellt ein Container einen Serverdienst über einen bestimmten Port zur Verfügung kann dieser mit einem anderen Port (oder auch dem gleichen) Port auf dem Host verknüpft werden.

Läuft beispielsweise im Container eine Webanwendung auf Port 80 (http), lässt sich dieser Port beim Start des Containers mit dem Parameter -p mit dem Port 80 des Hostrechners so verbinden

```
docker run -p 80:80 <image>
```

Auf dem Host lässt sich somit die Webseite des Containers mit `http://localhost` öffnen

Die Syntax für -p lautet

```
-p hostport:containerport
```

Die erste Zahl bezieht sich also auf die Portnummer des Hostrechners und die zweite der Portnummer im Container drin. Die Syntax folgt der in Docker üblichen Reihenfolge auch bei anderen Zuordnungen, immer zuerst den Host und dann den Container anzugeben.

Sollte der Port 80 des Hostrechners im obigen Beispiel bereits belegt sein, weil z.B. der Host selber ein Webserver ist und damit Port 80 verwendet, kann man einfach eine andere Portnummer verwenden. Für Webdienste üblich ist z.B. Port 8080:

```
docker run -p 8080:80 <image>
```

Die Containerwebseite ist dann unter `http://localhost:8080` erreichbar.

Volumes

Hier geht es darum, wo ein Container seine Daten speichert. Daten die in einem laufenden Container gespeichert sind, bleiben erhalten, wenn der Container gestoppt und wieder gestartet wird, nicht jedoch wenn ein Container gelöscht und wieder neu erzeugt wird.

Um das zu demonstrieren verwenden wir nochmals das Beispiel aus [Kapitel 1.3](#): Wir erzeugen einen Container:

```
docker run -d --name bsp-apache-php-container -p 8080:80 bsp-apache-php
```

und öffnen eine root-Shell innerhalb des Containers mit

```
docker exec -it bsp-apache-php-container /bin/bash
```

dann legen wir mit touch eine neue Datei abc.txt an.

```
vmadmin@ubuntu:~/bsp-apache-php$ docker exec -it bsp-apache-php-container /bin/bash
root@5de9ed33a981:/var/www/html# touch abc.txt
root@5de9ed33a981:/var/www/html# ls
abc.txt  index.php
root@5de9ed33a981:/var/www/html# exit
```

Nach

```
docker stop bsp-apache-php-container
docker start bsp-apache-php-container
```

ist die Datei abc.txt immer noch vorhanden.

```
vmadmin@ubuntu:~/bsp-apache-php$ docker exec -it bsp-apache-php-container /bin/bash
root@5de9ed33a981:/var/www/html# ls
abc.txt  index.php
root@5de9ed33a981:/var/www/html# exit
```

Wird der Container jedoch gelöscht und neu erstellt ist dies nicht mehr der Fall:

```
docker stop bsp-apache-php-container
docker rm bsp-apache-php-container
docker run -d --name bsp-apache-php-container -p 8080:80 bsp-apache-php
```

```
vmadmin@ubuntu:~/bsp-apache-php$ docker exec -it bsp-apache-php-container  
/bin/bash  
root@f84ad2b66b9e:/var/www/html# ls  
index.php  
root@f84ad2b66b9e:/var/www/html# exit
```

Beachten Sie auch die neue Container-ID (f84ad2b66b9e).

Diese Situation ist natürlich nicht brauchbar, wenn Daten das Neuerstellen eines Containers überleben sollen. Beispielsweise sollten Datenbanken ja erhalten bleiben, wenn ein Datenbankcontainer gelöscht werden muss um einer neuen Version des Servers in Betrieb zu nehmen.

Zu diesem Zweck können im Dockerfile Verzeichnisse die erhalten bleiben sollen als VOLUME gekennzeichnet werden. Beispielsweise finden Sie im Dockerfile eines mariadb-Servers den Eintrag `VOLUME /var/lib/mysql`. ([Dockerfile](#)). mariadb speichert die Daten standardmässig in diesem Verzeichnis.

Dies bewirkt nun dass die Daten nicht innerhalb des Containers gespeichert werden sondern ausserhalb auf dem Hostrechner. Wird der Container gelöscht und neu erstellt stehen Sie somit für einen neu erzeugten Container wieder zur Verfügung. Es gibt nun mehrere Möglichkeiten wo sich solche Verzeichnisse auf dem Host befinden und wie sie benannt werden.

1. Unbenannte Volumes

Schauen wir dazu nochmals das Beispiel des mariadb-Servers an. Einen Image können Sie mit dem folgenden Kommando herunterladen und einen Container daraus starten

```
docker run -d --name mariadb-test -e MYSQL_ROOT_PASSWORD=geheim mariadb
```

Um herauszufinden wo nun die Daten aus /var/lib/mysql gelandet sind können Sie das Kommando

```
docker inspect -f '{{.Mounts}}' mariadb-test
```

ergibt:

```
vmadmin@ubuntu:~/bsp-apache-php$ docker inspect -f '{{.Mounts}}' mariadb-test  
[{"volume": "752507751a42f0c781b96adacb4a3d73bdbbf2184ead3bd4874b4b5f065ee4eb",  
  "source": "/var/lib/docker/volumes/752507751a42f0c781b96adacb4a3d73bdbbf2184ead3bd4874b4b5f065ee4eb/_data",  
  "target": "/var/lib/mysql",  
  "type": "local",  
  "readOnly": true}]  
vmadmin@ubuntu:~/bsp-apache-php$
```

Wenn beim Erstellen des Containers also nichts eingestellt wird, liegt das Verzeichnis in einem zufällig benannten Unterverzeichnis von /var/lib/docker/volumes des Hostrechners. Die zufällige Bezeichnung verunmöglicht eine praktikable Verwaltung von Volumes nahezu. Sollte

z.B ein Volume gelöscht werden, müsste der vollständige Verzeichnisname angegeben werden:

```
docker volume rm 7525077...
```

2. Benannte Volumes

Eine bessere Variante ist es deshalb Volumes beim Erstellen eines Containers zu benennen:

```
docker run -d --name mariadb-test2 -v myvolume:/var/lib/mysql -e
MYSQL_ROOT_PASSWORD=geheim mariadb
```

```
vmadmin@ubuntu:~/bsp-apache-php$ docker run -d --name mariadb-test2 -v
myvolume:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=geheim mariadb
2ccb55f675a5867efee008b02e896cf0e0e1b66fe098465d6d7cc3e6a9bc6
vmadmin@ubuntu:~/bsp-apache-php$ docker inspect -f '{{.Mounts}}' mariadb-test2
[{"volume": "myvolume", "source": "/var/lib/docker/volumes/myvolume/_data", "target": "/var/lib/mysql", "type": "local", "read_only": false}]
```

Wie Sie sehen, wird nun der gewählte Name für das Unterverzeichnis verwendet. Die Syntax lautet somit:

```
-v volumename:containerverzeichnis
```

3. Volumes in eigenen Verzeichnissen

Anstelle eines Namens für das Hostvolume kann auch ein Verzeichnis angegeben werden:

```
-v hostverzeichnis:containerverzeichnis
```

Damit wird das Volume ganz aus der Dockerumgebung herausgelöst und kann an einer beliebigen Stelle platziert werden:

```
mkdir /home/vmadmin/databases
docker run -d --name mariadb-test3 -v /home/vmadmin/databases:/var/lib/mysql -e
MYSQL_ROOT_PASSWORD=geheim mariadb
```

Netzwerke

Hier geht es darum wie mehrere Dockercontainer untereinander kommunizieren können. Beispielsweise muss ein Web-Container mit einem Datenbank-Container kommunizieren können und an seine Daten zu kommen.

Der Befehl

```
docker network ls
```

zeigt die vorhandenen Netzwerke an.

```
vmadmin@ubuntu:~$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
65593a9ebb3b	bridge	bridge	local
364521a9eaa2	host	host	local
c69c18f0a974	none	null	local

Standardnetzwerk

Das Netzwerk mit dem Namen bridge ist das Standardnetzwerk und wird verwendet wenn nichts anderes angegeben wird. Die Netzwerkarchitektur lässt sich wie folgt darstellen:

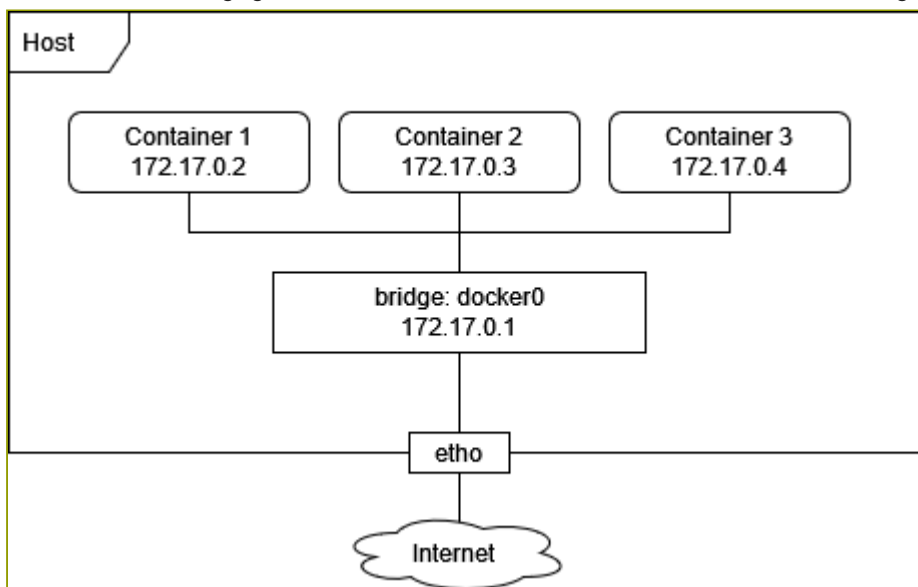


Abb. 1: Bridge Netzwerk

Wir wollen diese Architektur nun nachvollziehen: Die Ausgabe von `ifconfig` auf dem Host zeigt unter anderem das docker0 interface an:

```
vmadmin@ubuntu:~$ ifconfig
...
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:f5ff:fe1c:4929 prefixlen 64 scopeid 0x20<link>
    ether 02:42:f5:1c:49:29 txqueuelen 0 (Ethernet)
...
```

Dort erkennt man das private Klasse B Schnittstelle, mit Namen docker0 und der IP-Adresse 172.17.0.1. Diese Schnittstelle dient den Containern als Router.

Wir starten nun einen Ubuntu-Container. Dabei wird interaktiv (-it) eine Bash-Shell im Container geöffnet:

```
docker run -it --name ubuntu_1 ubuntu:latest
```

```
vmadmin@ubuntu:~$ docker run -it --name ubuntu_1 ubuntu:latest
root@5fe876094647:/#
```

Der Container lässt sich mit `exit` beenden und kann mit

```
vmadmin@ubuntu:~$ docker start -i ubuntu_1
root@5fe876094647:/#
```

auch wieder neu gestartet werden.

In einer zweiten Konsole kann nun das Kommando

```
docker network inspect bridge
```

ausgeführt werden:

```
vmadmin@ubuntu:~$ docker network inspect bridge
[
  {
    "Name": "bridge",
    "Id":
    "65593a9ebb3b25cb368166ff8dc0f3556cd5edf29cbc88286fc38dfa5068594c",
    "Created": "2022-06-25T07:03:44.321727785+02:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
```

```

    }
  ],
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "5fe8760946479f29c3b59470a7a7a23e80ea8c0689f04d1fa4d40c5f668b51c8": {
      "Name": "ubuntu_1",
      "EndpointID":
"194b76a3cd7a3360c054ad752a322822bf122d1544fd3266b49e55e116eda643",
      "MacAddress": "02:42:ac:11:00:02",
      "IPv4Address": "172.17.0.2/16",
      "IPv6Address": ""
    }
  },
  "Options": {
    "com.docker.network.bridge.default_bridge": "true",
    "com.docker.network.bridge.enable_icc": "true",
    "com.docker.network.bridge.enable_ip_masquerade": "true",
    "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
    "com.docker.network.bridge.name": "docker0",
    "com.docker.network.driver.mtu": "1500"
  },
  "Labels": {}
}
]

```

Hier erkennt man die Definition des Klasse B Netzwerkes 172.17.0.0/16 und die IP-Adresse des Containers ubuntu_1

Eigene Netzwerke

Alle Container landen standardmässig im selben Netzwerk, dem bridge-Netzwerk. Dies ist aus sicherheitstechnischen Gründen nicht ideal, wenn unterschiedliche Anwendungen voneinander isoliert sein sollen. Es lassen sich deshalb eigene Netzwerke definieren und diese den Containern zuordnen.

```

docker network create \
--driver=bridge \
--subnet=10.10.10.0/24 \
--gateway=10.10.10.1 \
my_net

```

Überprüfen mit

```
vmadmin@ubuntu:~$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
65593a9ebb3b	bridge	bridge	local
364521a9eaa2	host	host	local
049d1ccb15e7	my_net	bridge	local

ifconfig zeigt die neue Schnittstelle an:

```
vmadmin@ubuntu:~$ ifconfig
...
br-049d1ccb15e7: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 10.10.10.1 netmask 255.255.255.0 broadcast 10.10.10.255
    ether 02:42:88:5d:17:1e txqueuelen 0 (Ethernet)
...
```

Ein Container kann nun beim Start diesem Netzwerk zugeordnet werden:

```
docker run -it --name ubuntu_2 --network=my_net ubuntu:latest
```

und

```
docker network inspect my_net
```

zeigt die IP-Adresse des neuen Containers

```
vmadmin@ubuntu:~$ docker network inspect my_net
...
  "Config": [
    {
      "Subnet": "10.10.10.0/24",
      "Gateway": "10.10.10.1"
    }
  ]
  ...
  "Containers": {
    "8848cacd27a445a7805e34bed6542c21535aa5e7cbb370e4cae7e26cd7d19f15": {
      "Name": "ubuntu_2",
      "EndpointID":
        "1bf43446a8b4a23efb6af2780556fa5566a7e18c09c189b97341dedfa1f474b0",
      "MacAddress": "02:42:0a:0a:0a:02",
      "IPv4Address": "10.10.10.2/24",
      ...
    }
  }
```

Die IP-Adresse für den Container wird dabei von docker via DHCP aus dem definierten Netzwerk vergeben. Alternativ kann eine fixe IP-Adresse beim Start des Containers angegeben werden.

```
docker run -it --name ubuntu_2 --ip="10.10.10.10" --network=my_net
ubuntu:latest
```

Als nächstes soll nun der weiteroben dem Netzwerk bridge zugeordnete Container ubuntu_1 dem Netzwerk my_net zugeordnet werden. Dazu trennen wir ihn zuerst von bridge mit

```
docker network disconnect bridge ubuntu_1
```

anschliessend wird er zu my_net hinzugefügt und neu gestartet

```
docker network connect my_net ubuntu_1
docker start -i ubuntu_1
```

docker inspect zeigt nun beide Container an:

```
vmadmin@ubuntu:~$ docker network inspect my_net
...
  "Containers": {
    "5fe8760946479f29c3b59470a7a7a23e80ea8c0689f04d1fa4d40c5f668b51c8": {
      "Name": "ubuntu_1",
      "EndpointID":
      "9212735dd2214ab7fa18fadc13a7bac348582e1ddd7d0d70ee9f58f308271000",
      "MacAddress": "02:42:0a:0a:0a:03",
      "IPv4Address": "10.10.10.3/24",
      "IPv6Address": ""
    },
    "8848cacd27a445a7805e34bed6542c21535aa5e7cbb370e4cae7e26cd7d19f15": {
      "Name": "ubuntu_2",
      "EndpointID":
      "1bf43446a8b4a23efb6af2780556fa5566a7e18c09c189b97341dedfa1f474b0",
      "MacAddress": "02:42:0a:0a:0a:02",
      "IPv4Address": "10.10.10.2/24",
      "IPv6Address": ""
    }
  }
  ...
```

Um zu überprüfen, ob die beiden Container tatsächlich mit einander kommunizieren können, installieren wir auf ubuntu_1 das Paket iputils-ping:

```
apt update
apt install iputils-ping
```

Anschliessend wird ubuntu_2 angepingt

```
root@5fe876094647:/# ping 10.10.10.2
PING 10.10.10.2 (10.10.10.2) 56(84) bytes of data.
64 bytes from 10.10.10.2: icmp_seq=1 ttl=64 time=0.496 ms
...
```

Dabei ist es nicht einmal nötig die IP-Adresse von ubuntu_2 zu kennen, da man auch den Namen direkt verwenden kann

```
root@5fe876094647:/# ping ubuntu_2
PING ubuntu_2 (10.10.10.2) 56(84) bytes of data.
64 bytes from ubuntu_2.my_net (10.10.10.2): icmp_seq=1 ttl=64 time=0.099 ms
...
```

Nachdem alle Container, die zu einem Netzwerk hinzugefügt wurden, gestoppt oder getrennt wurden, können Sie das Netzwerk mit folgendem Befehl löschen:

```
docker network rm my_net
```

Einführung

Grundgedanke

Eine docker-Applikation besteht in der Regel aus mehreren Containern mit mehr oder weniger komplexen Einstellungen für das Kommando `docker run`. Diese Kommandos liessen sich in einem Skript abspeichern und man könnte einfach dieses Skript speichern um die ganze Applikation zu starten. Es gibt jedoch einen bequemeren Weg mit `docker-compose`.

`docker-compose` wertet die Datei `docker-compose.yml` aus. In dieser Datei sind alle Konfiguration für die einzelnen Container festgehalten. Die ganze Applikation lässt sich dann komfortabel mit `docker-compose up` und `docker-compose down` starten und stoppen.

Die Datei `docker-compose.yml` verwendet eine spezielle Syntax (YAML = **Y**AML **A**in't **M**arkup **L**anguage, rekursives Akronym), die es zu erlernen gilt. Die YAML-Syntax wird in vielen Bereichen für Konfigurationsdateien verwendet.

Installation

Im folgenden wird die Installation von `docker` und `docker compose` unter Ubuntu beschrieben. Die Installation wird am Einfachsten als `root` durchgeführt. Wechseln deshalb zu einer `root`-Shell:

```
sudo su
```

```
curl -L "https://github.com/docker/compose/releases/download/v2.5.1/docker-  
compose-linux-x86_64" \  
-o /usr/local/bin/docker-compose
```

und

```
chmod +x /usr/local/bin/docker-compose
```

Überprüfen Sie die Version

```
docker-compose --version
```


YAML-Syntax

Folgende Syntaxregeln gelten bei YAML-Dateien:

- ✓ `#` Zeilenkommentar
- ✓ Text kann mit einfachen oder doppelten Hochkommas `'`, `"` geklammert werden. Dies ist allerdings nur nötig, wenn der Text Sonderzeichen, z.B. Escape-Sequenz enthält
- ✓ Eine Liste wird mit `-` + Abstand gebildet, z.B.

```
# A list of tasty fruits
- Apple
- Orange
- Strawberry
- Mango
```

- ✓ Key-Value-Paare werden mit `:` + Abstand gebildet, z.B.:

```
name: Hans Martin
```

- ✓ Objekte bestehend aus mehreren Key-Value-Paaren, können durch Zeilenumbrüche gebildet werden

```
vorname: Hans Martin
nachname: Keller
alter: 20
```

- ✓ Ein Value kann seinerseits wiederum rekursiv aus Objekten, Key-Value-Paaren oder Listen bestehen:

```
# Employee records
- martin:
  name: Martin D'vloper
  job: Developer
  skills:
    - python
    - perl
    - pascal
- tabitha:
  name: Tabitha Bitumen
  job: Developer
  skills:
    - lisp
    - fortran
    - erlang
```

- ✔ Die Strukturierung der Elemente wird durch Einrückungen erreicht. Dabei **müssen Leerzeichen** verwendet werden. Tabulatoren auch am Zeilenende führen zu Fehlern.

Einen Online-Validator für die YAML-Syntax finden Sie unter <https://codebeautify.org/yaml-validator>

Weitergehende Syntaxmerkmale, die im Zusammenhang mit docker-compose jedoch kaum gebraucht werden, sind hier <https://en.wikipedia.org/wiki/YAML> festgehalten.

docker-compose.yml

Zusätzlich zur allgemeinen Syntax von YAML-Dateien, gibt es bei docker-compose.yml definierte Schlüsselwörter die als Keys verwendet werden müssen:

- ✓ **version:** zur Zeit "3.8"
- ✓ **services:** Zur Definition der einzelnen Container. Die Dienstnamen (Keys) für die Container können freigewählt werden.
- ✓ **networks:** Zur Definition eigener Netzwerke
- ✓ **volumes:** Angabe von benannten Volumes
- ✓ **secrets:** Angabe von Passwortdateien

```
# Grundsätzlicher Aufbau von docker-compose.yml
version: "3.8"
services:
  dienstname1:
    schlüsselwort1: einstellung1
    schlüsselwort2: einstellung2
    ...
  dienstname2:
    schlüsselwort1: einstellung1
    schlüsselwort2: einstellung2
    ...
volumes:
networks:
secrets:
```

Die wichtigsten Schlüsselwörter unterhalb der Dienstnamen sind:

- ✓ **image:** Das Basisimage eines Service oder
- ✓ **build:** Ein Verzeichnis mit Dockerfile, der Service wird dann nicht aus einem Image gestartet sondern aus einem Dockerfile
- ✓ **restart:** always, on-failure oder unless-stopped
- ✓ **environment:** Liste mit Umgebungsvariablen für einen Container
- ✓ **volumes:** Liste der gemounteten Volumes
- ✓ **ports:** Liste der Portweiterleitungen
- ✓ **expose:** Ports für die Kommunikation zwischen Containern
- ✓ **networks:** Verweis auf ein im Top-Level-Schlüsselwort networks definiertes Netzwerk
- ✓ **secrets:** Verweis auf die im Top-Level-Schlüsselwort secrets definierte Passwortdatei

Hier nun die Schlüsselwörter im Detail:

image

Wenn das Basisimage in Dockerhub verfügbar ist, kann es direkt verwendet werden:

```
services:
  my-service:
    image: ubuntu:latest
    ...
```

build

Wird ein eigenes Image benötigt, kann dieses aus dem angegebenen Dockerfile erstellt werden:

```
services:
  my-custom-app:
    build: /path/to/dockerfile/
    ...
```

restart

Dieser Eintrag definiert die Restart-Policy für einen Container. Folgende Werte sind möglich

- ✔ **no**: Der Standard, Container wird nicht automatisch gestartet
- ✔ **on-failure**: Der Container wird bei einem Fehler (Exit-Code ungleich 0) automatisch neu gestartet
- ✔ **always**: Der Container wird immer neu gestartet, insbesondere bei einem Reboot. Wenn der Container manuell gestoppt wird, startet er neu, wenn der Dockerdaemon neu gestartet wird.
- ✔ **unless-stopped**: Ähnlich wie always, wird der Container manuell gestoppt, startet er nach einem Neustart des Dockerdaemons oder einem Reboot nicht mehr neu.

```
services:
  my-custom-app:
    restart: always
    ...
```

ports

Um einen Service vom Host aus zu erreichen, wird die Angabe der Portweiterleitungen benötigt:

```
services:
  my-custom-app:
    image: myapp:latest
    ports:
      - "8080:3000"
      - "8081:4000"
    ...
```

expose

Angabe der Ports, welche die Container für die Kommunikation untereinander benötigen. Wenn ein Basisimage bereits einen Port exponiert, ist diese Angabe nicht nötig.

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    expose:
      - "80"
```

networks

Hier kann ein unter dem Top-Level-Schlüsselwort definiertes Netzwerk referenziert werden.

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    networks:
      - my_network
    ...
  another-service-in-the-same-network:
    image: alpine:latest
    networks:
      - my_network
    ...

networks:
  my_network:
    ipam:
      config:
        - subnet: 172.17.0.0/24
          gateway: 172.17.0.1
```

In der Regel werden bei der Definition des Netzwerkes keine weiteren Optionen angegeben. Docker kümmert sich dann selber um die konkrete Definition:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
```

```
networks:
  - my_network
  ...

networks:
  my_network:
```

volumes

Gibt an wie Containervolumes auf den Host gemountet werden. Benannte Volumes müssen im Top-Level-Schlüsselwort volumes angegeben werden. Somit können benannte Volumes von mehreren Containern gleichzeitig verwendet werden. Der Zusatz :ro mountet ein Verzeichnis read-only.

```
services:
  volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/my-volumes/named-global-volume
      - /tmp:/my-volumes/host-volume
      - /home:/my-volumes/readonly-host-volume:ro
      ...
  another-volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/another-path/the-same-named-global-volume
      ...
volumes:
  my-named-global-volume:
```

Unbenannte Volumes und Volumes in eigene Verzeichnisse müssen nicht im Top-Level volume aufgeführt werden.

Es können auch einzelnen Dateien gemountet werden.

depends_on

Um Abhängigkeiten zu definieren, kann depends_on verwendet werden. Die betreffenden Container werden dann zuerst geladen:

```
services:
  kafka:
    image: wurstmeister/kafka:2.11-0.11.0.3
    depends_on:
      - zookeeper
    ...
  zookeeper:
    image: wurstmeister/zookeeper
    ...
```

environment

Variablen können sowohl statisch als auch dynamisch mit `${}` definiert werden

```
services:
  database:
    image: "postgres:${POSTGRES_VERSION}"
    environment:
      DB: mydb
      USER: "${USER}"
```

Die dynamischen Variablen können dabei u.a. in einer Datei `.env` im selben Verzeichnis als Key-Value-Paare definiert werden:

```
POSTGRES_VERSION=alpine
USER=foo
```

Die verfügbaren Umgebungsvariablen müssen in der Dokumentation zu einem Image auf Dockerhub nachgeschaut werden

secrets

Sollen Passwörter aus sicherheitstechnischen Gründen nicht in der docker-compose Datei aufgeführt werden, kann man secrets verwenden. Das Top-Level-Schlüsselwort gibt an, wo sich die Passwortdatei befindet. Die Einträge bei den Services referenzieren dann die Top-Level-Definition

```
services:
  db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db_root_password
      MYSQL_PASSWORD_FILE=/run/secrets/db_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
    secrets:
      - db_root_password
      - db_password
  ...
secrets:
  db_password:
    file: db_password.txt
  db_root_password:
    file: db_root_password.txt
```

Die in `MYSQL_ROOT_PASSWORD_FILE` und `MYSQL_PASSWORD_FILE` angegebenen Dateien innerhalb des Containers enthalten dann die in den unter dem Top-Level-Schlüsselwort `secrets`

angegebenen Passwörter. Die verfügbaren Secretsvariablen müssen in der Dokumentation zu einem Image auf Dockerhub nachgeschaut werden

Die vollständige Referenz zum docker-compose.yml Aufbau finden Sie unter <https://docs.docker.com/compose/compose-file>

Anwendung

Wenn die docker-compose.yml Datei angelegt ist, wechseln Sie in dieses Verzeichnis

```
cd myapp
```

und können die ganze Applikation mit

```
docker-compose up -d
```

starten (-d daemon, bewirkt dass die Applikation im Hintergrund gestartet wird) und mit

```
docker-compose down
```

wieder beenden.

Im Idealfall läuft alles gut. Wenn jedoch ein Fehler auftritt empfiehlt es sich `docker-compose up` ohne das -d auszuführen. Die Loggingausgaben aller Container erfolgen dann direkt auf der Konsole

Standardimages

Die folgenden Images aus dem dockerhub dienen vielfach als Basisimages für alle möglichen Anwendungen und sind damit besonders wichtig. Sie werden deshalb in diesem Kapitel kurz beschrieben.

alpine

Alpine Linux ist eine ausserhalb von docker wenig verwendete Linuxdistribution. Sie ist auf den sparsamen Umgang mit Ressourcen optimiert, was sich in erster Linie durch die Grösse des Images ausdrückt. Hier ein Vergleich verschiedener Linuxdistributionen als Dockerimage:

Distribution	Image Grösse
alpine	6 MB
ubuntu	80 MB
debian	125 MB
oracle	250 MB

Die kleine Grösse bewirkt eine grosse Performace beim Starten von Anwendungen, die auf diesem Image beruhen. Die Grösse wird natürlich mit einem Verzicht auf Komfort und bekannte Linux Standardtools erkaufte. Als Paketverwaltung kommt `apk` zum Zug. Standardmässig kommt die rudimentäre `/bin/sh` Shell zum Zug. Bash lässt sich aber nachinstallieren:

```
apk add --update bash bash-completion
```

Das Image lässt sich so ausprobieren

```
docker run -it --rm -h alpine --name alpine alpine
```

Man gelangt in eine interaktive root-Shell und kann z.B. die Version abfragen

```
cat /etc/os-release
```

```
/ # cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.16.0
PRETTY_NAME="Alpine Linux v3.16"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://gitlab.alpinelinux.org/alpine/aports/-/issues"
/ #
```

ubuntu

Wird eine umfangreichere oder bekanntere Linux Distribution benötigt kann man z.B. das ubuntu Image verwenden. Dieses ist deutlich grösser als alpine, jedoch immer noch nicht eine vollwertige Linuxdistribution. Auch hier fehlen bekannte Tools wie ping oder ifconfig, die aber natürlich alle via apt nachinstalliert werden können Das image kann so ausprobiert werden

```
docker run -it --name ubuntu-test ubuntu:latest
```

eine spezifische Version mit

```
docker run -it --name ubuntu-test ubuntu:20.04
```

```
vmadmin@ubuntu:~$ docker run -it --name ubuntu-test ubuntu:latest
root@5fe876094647:/#
```

Der Container lässt sich mit `exit` beenden und kann mit

```
vmadmin@ubuntu:~$ docker start -i ubuntu-test
root@5fe876094647:/#
```

auch wieder neu gestartet werden.

apache

Das offizielle Image enthält nur den apache http Server, also kein php. Um den Server zu testen können Sie dieses Kommando verwenden:

```
docker run -dit --name my-apache-app -p 8080:80 -v
"$PWD":/usr/local/apache2/htdocs/ httpd:2.4
```

Dieses verbindet das lokale Arbeitsverzeichnis (\$PWD) mit dem Serverroot. Die Webseite ist unter <http://localhost:8080> verfügbar.

In der Regel wird dieses Image nicht direkt verwendet, sondern der apache Server wird z.B. auf Basis von alpine hinzuiinstalliert. Man gewinnt dadurch vielmehr Flexibilität in Bezug auf die Konfiguration von zusätzlichen Modulen.

mariadb

mariadb wird von vielen Applikationen als Datenbankserver verwendet. Sie können ein mariadb Container mit

```
mkdir dbvolume
docker run -d --name mariadb -e MYSQL_ROOT_PASSWORD=geheim \
-v $(pwd)/dbvolume:/var/lib/mysql mariadb
```

starten. Dabei wird über eine Umgebungsvariable das root-Passwort gesetzt und das Datenbankverzeichnis im Container /var/lib/mysql auf das aktuelle Arbeitsverzeichnis/dbvolume gemountet. Das Verzeichnis dbvolume muss vorgängig erstellt werden.

Wenn Sie mysql auf dem Host nicht installiert haben, kann der mysql-Client im Container für die Administration des Servers verwendet werden:

```
docker exec -it mariadb mysql -u root -p
```

Das Image stellt weitere Umgebungsvariablen zur Verfügung, die beim Start verwendet werden können:

Variable	Bedeutung
MYSQL_ROOT_PASSWORD	Das mysql root Passwort
MYSQL_DATABASE	Wenn diese Variable gesetzt ist, wird beim ersten Start des Containers eine leere Datenbank mit dem angegebenen Namen erstellt.
MYSQL_USER	Der angegebene Benutzer wird beim ersten Start des Containers erstellt und erhält alle Rechte (GRANT ALL) auf der in MYSQL_DATABASE angegeben Datenbank.
MYSQL_PASSWORD	Das Passwort für MYSQL_USER
MYSQL_ALLOW_EMPTY_PASSWORD	Wenn diese Variable auf yes gesetzt wird, kann bei MYSQL_ROOT_PASSWORD ein leeres Passwort

Variable	Bedeutung
	gesetzt werden. Dies ist nicht zu empfehlen.
MYSQL_RANDOM_ROOT_PASSWORD	Wird diese Variable auf yes gesetzt, wird ein zufällig erzeugtes root Passwort gesetzt.

Das Verzeichnis /docker-entrypoint-initdb.d innerhalb des Containers kann verwendet werden, um bei einer ersten Ausführung des Containers sql-Skripts laufen zu lassen. Sämtliche Skripte in diesem Verzeichnis werden automatisch ausgeführt. Beispielsweise kann dort Code drin sein um die Tabellen in der Datenbank MYSQL_DATABASE zu erstellen. Die Skripte werden nur ausgeführt, falls es noch keine Datenbanken gibt.

Eigene Images

Wenn die vorhandenen Images den Anforderungen nicht genügen, können eigene Images erstellt werden. Dies eignet sich dann, wenn zusätzliche Pakete oder Konfigurationen benötigt werden. Im **Kapitel 1.3 (Erste Schritte)** wurde das Vorgehen bereits vorgestellt. Hier soll es nun genauer untersucht werden.

Kurz gesagt wird in einem Arbeitsverzeichnis die Datei mit Namen **Dockerfile** erstellt. Diese enthält in einer speziellen Syntax das Rezept um das Image anschliessend mit `docker build` zu erstellen. Mit `docker push` kann das Image bei Bedarf in den Dockerhub geladen werden.

Schlüsselwörter

Folgende Schlüsselwörter werden im Dockerfile verwendet

Schlüsselwort	Bedeutung
ADD	Kopiert Dateien in das Dateisystem des Images.
CMD	Führt das angegebene Kommando beim Container-Start aus.
COPY	Kopiert Dateien aus dem Projektverzeichnis in das Image.
ENTRYPOINT	Führt das angegebene Kommando immer beim Container-Start aus.
ENV	Setzt eine Umgebungsvariable.
EXPOSE	Gibt die aktiven Ports des Containers an.
FROM	Gibt das Basis-Image an.
LABEL	Legt eine Zeichenkette fest.
RUN	Führt das angegebene Kommando aus.
USER	Gibt den Account für RUN, CMD und ENTRYPOINT an.

Schlüsselwort	Bedeutung
VOLUME	Gibt Volume-Verzeichnisse an.
WORKDIR	Legt das Arbeitsverzeichnis für RUN, CMD, COPY etc. fest.

Beispiel

Hier sehen Sie ein Beispiel eines Dockerfiles für ein eigenes Webserver Image:

```
# Datei Dockerfile
FROM ubuntu:latest

LABEL maintainer "name@meine-webseite.ch "
LABEL description "Webserver Image für www.meine-webseite.ch"

# Umgebungsvariablen und Zeitzone einstellen
# (erspart interaktive Rückfragen )
ENV TZ="Europe/Zuerich" \
    APACHE_RUN_USER=www-data \
    APACHE_RUN_GROUP=www-data \
    APACHE_LOG_DIR=/var/log/apache2

# Zeitzone einstellen, Apache installieren, unnötige Dateien
# aus dem Paket-Cache gleich wieder entfernen, HTTPS aktivieren
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    echo $TZ > /etc/timezone && \
    apt-get update && \
    apt-get install -y apt-utils apache2 && \
    apt-get -y clean && \
    rm -r /var/cache/apt /var/lib/apt/lists/* && \
    a2ensite default-ssl && \
    a2enmod ssl

# Ports 80 und 443 freigeben
EXPOSE 80 443

# Das Webroot-Verzeichnis als Volume definieren
VOLUME /var/www/html

# Startkommando für den Apache Webserver
CMD ["/usr/sbin/apache2ctl" , "-D" , "FOREGROUND"]
```

Das Image kann nun mit

```
docker build -t meine-webseite-image .
```

erstellt werden. Das Argument Punkt im Kommando gibt an das sich das Dockerfile im aktuellen Verzeichnis befindet. Wechseln Sie also zuerst ins Verzeichnis mit dem Dockerfile.

Liegen keine Fehler im Dockerfile vor wird das Image schrittweise erstellt und mit

```
Successfully built e2a828e60be7
Successfully tagged meine-webseite-image:latest
```

abgeschlossen.

Überprüfen Sie das vorhandene Image mit

```
docker images
```

```
vmadmin@ubuntu:~/meine-webseite$ docker images
REPOSITORY          TAG         IMAGE ID      CREATED        SIZE
meine-webseite-image latest      e2a828e60be7 3 minutes ago 223MB
...
```

Sollte das Image noch nicht den Wünschen entsprechen kann es mit

```
docker rmi meine-webseite-image
```

wieder gelöscht werden.

Ein aus dem Image abgeleiteter Container kann nun mit folgendem Kommando gestartet werden

```
docker run -d --name meine-webseite-container -p 8888:80 \
-v /home/vmadmin/meine-webseite/site:/var/www/html meine-webseite-image
```

Das Verzeichnis site muss vorgängig erstellt werden und wird auf das Webroot-Verzeichnis abgebildet. Wenn Sie darin eine Indexdatei index.html erstellen, wird diese nun beim Aufruf von <http://localhost:8888> angezeigt

FROM

Dieses Schlüsselwort legt das Basisimage fest. Beispiel:

```
FROM ubuntu:20.04
```

Wird der Tag (20.04) nicht verwendet, kommt latest zum Zug:

```
FROM ubuntu:latest
```

ADD, COPY

Beide Kommandos kopieren Dateien oder Verzeichnisse vom Host in das Imagedateisystem

```
COPY samplesite/ /var/www/html
```

```
ADD myfonts.tgz /usr/local/share/texmf
```

Im Gegensatz zu COPY kann ADD auch eine URL als Quelle verwenden, ausserdem entpackt es Archivdateien (tar, gz, ..) automatisch.

Mit `--chown=user:group` kann der Eigentümer und Gruppe im Zielsystem festgelegt werden.

```
COPY --chown=node:node package.json package-lock.json /src/
```

Die Dockerdokumentation empfiehlt COPY anstelle von ADD zu verwenden.

CMD, ENTRYPOINT

Beide Kommandos geben an, welches Programm beim Start eines Containers mit `docker run` oder `docker start` ausgeführt wird. Als erstes wird in eckigen Klammern (Array) und doppelten Hochkommas der vollständige Pfad eines Programmes angegeben, die anschliessenden Arrayelemente sind die Parameter für das Kommando.

```
CMD ["/bin/ls", "/var"]
```

Bei CMD wird das Kommando das bei docker run angegeben wird, anstelle von CMD ausgeführt.

Bei ENTRYPOINT wird das Kommando das bei docker run angegeben wird, als Parameter zu ENTRYPOINT hinzugefügt. Es können auch beide Schlüsselwörter verwendet werden.

Beispielsweise gibt es beim mariadb Image die folgenden Einträge:

```
CMD ["mysqld"]  
ENTRYPOINT ["docker-entrypoint.sh"]
```

RUN

RUN gibt die Kommandos an, die einmalig beim Erstellen des Images mit `docker build` ausgeführt werden. Typischerweise werden damit zusätzlich benötigte Softwarepakete installiert.

```
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
    subversion \
    joe \
    vim \
    less && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Bei diesem Beispiel werden zuerst die Paketquellen aktualisiert und anschliessend die benötigten Pakete (ohne Rückfragen) installiert. Zum Schluss werden alle Cachedateien gelöscht um das Image klein zu halten.

Die Dockerdokumentation empfiehlt möglichst viele Kommandos in einem RUN zusammenzufassen, da jedes RUN einen neuen Layer im Image erstellt, was ineffizient ist.

VOLUME

VOLUME gibt die Verzeichnisse an, die im Hostsystemgemountet werden. Beim folgenden Beispiel werden zwei Verzeichnisse auf den Host abgebildet:

```
VOLUME ["/var/lib/mysql", "/var/log/mysql"]
```

Jenachdem wie der Container gestartet wird, werden die Volumes als unbenannte oder benannte Volumes oder in ein eigenes Verzeichnis eingebunden. Vergleichen Sie dazu [Kapitel 1.6 Volumes](#)

Alle Verzeichnisse die in VOLUME aufgeführt werden, überstehen also das Löschen und Neuerstellen eines Containers.

ENV

ENV gibt die Umgebungsvariablen an die bei docker run mit einem Wert versehen werden können. Beispielsweise kann mit

```
ENV PORT=3000
```

bei `docker run ... -e PORT=8000 ...` ein anderer Port definiert werden (3000 ist also der Defaultwert, wenn nichts angegeben wird)

EXPOSE

EXPOSE gibt die aktiven Ports eines Containers an. Mehrere Container untereinander im selben Netzwerk können darüber kommunizieren. Beispielsweise exponiert eine Redis Container den

Port 6379. Ein anderer Container kann somit über diesen Port mit dem Redis-Container kommunizieren.

```
EXPOSE 6379
```

Ein exponierter Port kann ausserdem bei `docker run ... -p 8080:80 ...` auf einen Hostport weitergeleitet werden (vgl. [Kapitel 1.5](#))

LABEL

Mit LABEL können beliebige Metadaten zu einem Image festgehalten werden. Hier verschiedene Beispiele:

```
FROM ubuntu:latest
LABEL "website.name"="geeksforgeeks website"
LABEL "website.tutorial-name"="docker"
LABEL website="geeksforgeeks"
LABEL desc="This is docker tutorial with \
          geeksforgeeks website"
LABEL tutorial1="Docker" tutorial2="LABEL INSTRUCTION"
```

Diese Labels tauchen dann bei den Informationen (`docker inspect <container-id>`) zu einem laufenden Container wieder auf:

```
...
"Labels": {
  "desc": "This is docker tutorial with geeksforgeeks website",
  "tutorial1": "Docker"
  ...
}
...
```

USER

USER setzt den Benutzer für die folgenden RUN, CMD und ENTRYPOINT Anweisungen

```
USER patrick
```

WORKDIR

WORKDIR setzt das Arbeitsverzeichnis für die folgenden RUN, CMD, ENTRYPOINT, COPY und ADD Anweisungen. Nicht existierende Verzeichnisse werden automatisch erstellt

```
WORKDIR /a  
WORKDIR b  
WORKDIR c  
RUN pwd
```

Die Ausgabe des abschliessenden pwd-Kommandos wäre in diesem Beispiel /a/b/c .

Registries

Docker verwendet standardmässig den Docker Hub als Registry für Images. Es gibt jedoch eine Reihe weiterer Registries, die ähnlich wie Docker Hub funktionieren. Diese Registries sind in der Regel jedoch kostenpflichtig. Deren Nutzung bietet sich dann an, wenn bei diesen Anbietern auch andere Clouddienste verwendet werden. Je nach dem ist dann die Nutzung der Imageregistry im Preis inbegriffen. Die meisten Registries bieten (kostenpflichtig) zusätzlich an, eigene Images auf bekannte Sicherheitslücken zu scannen. Im folgenden Abschnitt werden die wichtigsten Registries kurz vorgestellt.

Docker Hub

<https://hub.docker.com/>

Azure

<https://docs.microsoft.com/en-us/azure/container-registry/>

Bei Azure können Sie den kostenpflichtigen Azure Defender aktivieren, der neben anderen Diensten auch das Scannen von Container Images übernimmt.

Amazon

<https://aws.amazon.com/de/ecr/>

In der Elastic Container Registry von Amazon haben Sie ebenfalls die Möglichkeit, Docker-Images automatisch scannen zu lassen. Amazon verwendet dazu den Open-Source-Scanner Clair

```
docker pull public.ecr.aws/amazonlinux/amazonlinux
```

Google

<https://cloud.google.com/container-registry>

Google hat einen integrierten Dienst, der aktuell nach Sicherheitsproblemen in Images auf der Basis von Ubuntu, Debian, Alpine Linux und Red Hat suchen kann.

RedHat

<https://catalog.redhat.com/software/containers/explore>