

MODUL 347

Dienst mit Container anwenden

Grundlagen der Container- Technologie

Motivation

- Sie haben Erfahrung im Programmieren.
- Ihr Programm läuft in Ihrer Entwicklungsumgebung und eine lauffähige Version funktioniert auf Ihrem Rechner?
- Haben Sie sich schon Gedanken gemacht, wie das Programm bei einem Kunden installiert wird? Welche Dateien müssen Sie mit ausliefern? Welche Bibliotheken? Müssen Sie Pfad-Variable setzen? Ist es wichtig, welches Betriebssystem Ihr Kunde hat?
- Auf welche Probleme sind Sie bisher gestossen?
- Vielleicht sind diese mit der Container-Technologie lösbar!

Inhaltsverzeichnis

- Warum Container?
- Container vs. Virtuelle Maschinen
- Die Architektur von Docker
- Images und Container
- Container Networking
- Dateimanagement in Docker
- Netzwerke in Docker
- Mithilfe von Dockerfiles eigene Images entwickeln

Warum Container?

- Keine fehlenden Dateien
- Keine Versionsprobleme von Software-Komponenten
- Keine unterschiedlichen Konfigurationen
- Keine Betriebssystem-Probleme
- Build, run and ship applications

Container vs. Virtuelle Maschinen

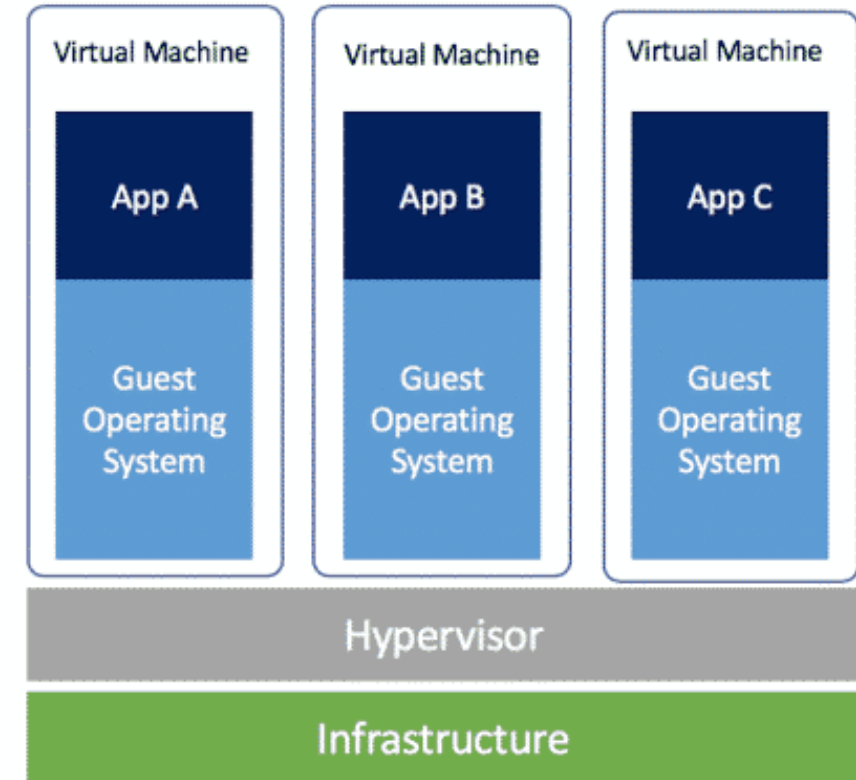
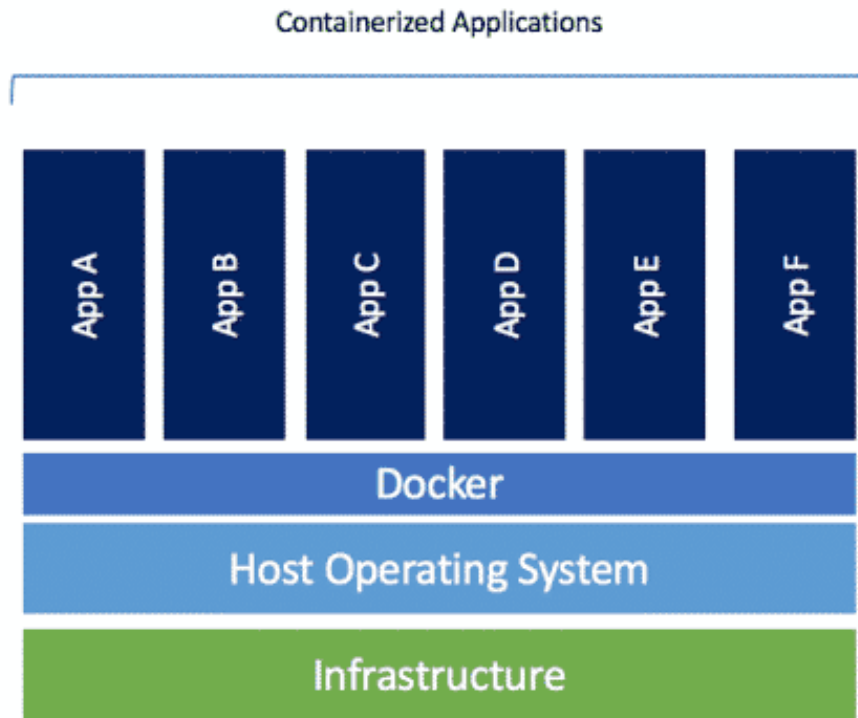
Container

- Kapselt eine Anwendung als Softwarepaket inkl. aller benötigten Dateien und Konfigurationen.
- Bietet eine Umgebung, in der Applikationen isoliert laufen
- Verwendet das Betriebssystem des Hosts
- Startet schnell, benötigt weniger HW-Ressourcen

Virtuelle Maschine

- Abstraktion einer ganzen physischen Maschine
- Läuft auf einem Hypervisor
- Beispiele für Hypervisoren sind VirtualBox, VMWare, Hyper-V
- Jede virtuelle Maschine besitzt ein gesamtes Betriebssystem
- Startet langsamer, benötigt mehr Ressourcen

Container vs. Virtuelle Maschinen



Grundlegende Shell-Befehle

- **cd** (change directory) in einen (Unter-)Ordner wechseln
- **cd ..** in den übergeordneten Ordner wechseln
- **ls** (list) Inhalt des Ordners anzeigen
- **pwd** (print working directory) den Pfad des aktuellen Verzeichnisses ausgeben
- **mkdir** (make directory) einen neuen Ordner anlegen
- **rm** (remove) Dateien und Ordner löschen
- **cat** (concatenate) eine neue Textdatei anlegen oder Inhalt ausgeben
- **touch** eine neue Datei anlegen
- **clear** Konsole leeren
- **exit** (oder **Strg+D**) Shell beenden
- Unter Ubuntu
 - Shell in Ubuntu starten: `docker run -it ubuntu`
 - `apt update`: Paketlisten laden / aktualisieren
 - `apt install -y neofetch`: Das Programm "neofetch" installieren; `apt install nano`: Programm "nano" installieren

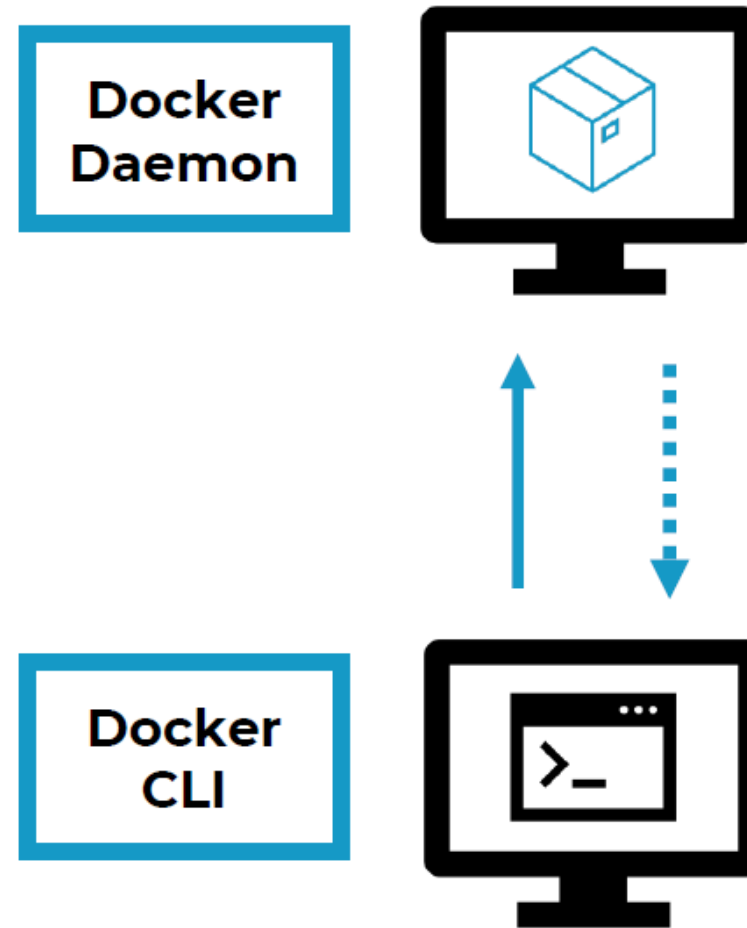
Die Architektur von Docker

- Client-Server: Ein Container ist ein Prozess, der auf einer Engine, läuft und mit ihr über ein REST-API kommuniziert.
- Container «teilen» sich das Host-Betriebssystem über diese Engine (dazu gehören Zugriff auf Dateisystem, Speicher, Netzwerk, CPU).
- Linux- und Windows-Kernel unterstützen die Container-Technologie, macOS benötigt eine zusätzliche Linux VM.



pexels.com (CC0)

Die Client-Server-Architektur von Docker



Zusammenfassung: Einstieg und Architektur

- Die Container-Technologie vereinfacht die Verteilung von Applikationen in Form von Images, welche die Applikationsdateien sowie alle Abhängigkeiten und Konfigurationen beinhalten.
- Images laufen als Container in einem eigenen Prozess auf einer Docker-Engine. Sie haben Zugriff auf das Host-Betriebssystem.
- Gegenüber Virtuellen Maschinen sind Container leichtgewichtiger, da sie nicht eine ganze Maschine sondern nur ein OS virtualisieren.
- Die bekannteste und am weitesten verbreitete Anwendung ist Docker. Docker Desktop beinhaltet eine Docker-Engine sowie weitere Tools. Docker kann über eine Kommandozeile oder auch in einer grafischen Benutzerumgebung bedient werden (z. B. build, run, push und pull).

Was ist ein Image?

- Container werden auf der Basis von **Images** erstellt
- Sie können sich ein Image als Kopiervorlage für Container vorstellen
- **Images** sind Pakete mit allen erforderlichen Daten (Dependencies, Konfigurationen), um einen Container zu bauen
- Images sind nach ihrem Erstellen unveränderlich (read-only-Dateisysteme)

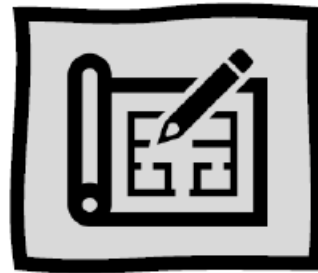
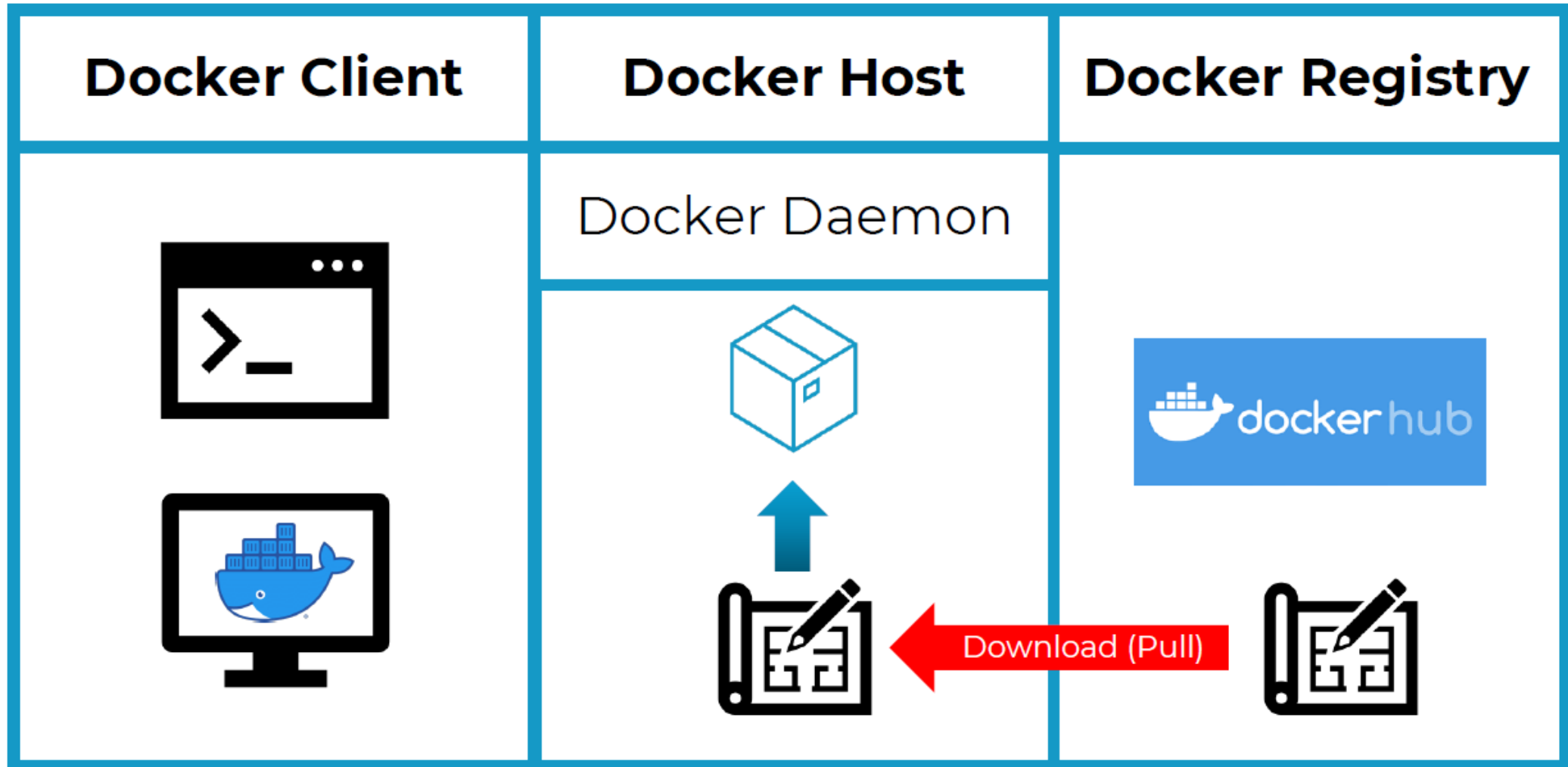


Image („Klasse“)



Container („Objekt“)

Das Kommando docker pull



Wie ist ein Image aufgebaut?

- Jedes Image besteht aus mehreren Schichten (“Layers”)
- Mit jedem Layer entsteht wiederum selbst ein neues Image
- Die Layers bauen aufeinander auf: jeder Layer enthält die Veränderungen zum vorherigen Zustand des Images
 - Informationseffizienz: da jedes Image im Cache von Docker gespeichert wird, brauchen bei einem Update von einem Image nur die aktualisierten Layers heruntergeladen zu werden

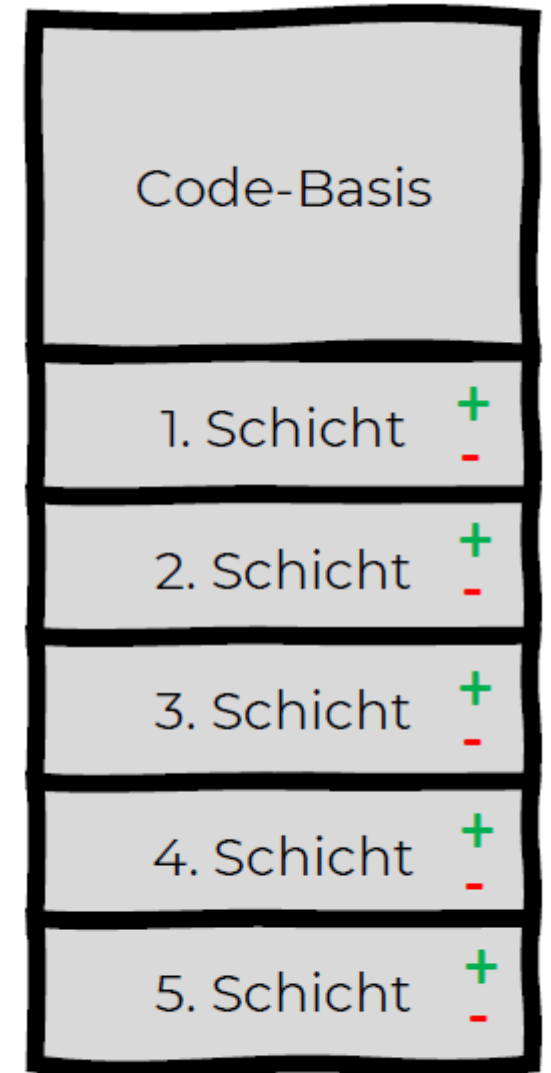


Image:latest

Das Kommando **docker image**

- Eine Übersicht über alle lokalen Images können Sie sich per CLI anzeigen lassen mit dem Befehl
 - **docker image ls** bzw. **docker images**
- Die Historie eines Images abrufen
 - **docker image history [Name vom Image]**
 - **docker image history node**
 - **docker image history --no-trunc node**
- Detaillierte Infos zu einem Image abfragen
 - **docker image inspect [Name vom Image]**
 - **docker image inspect node**

Wie kannst du Images löschen?

- Ein Image löschen
 - **docker image rm [Name vom Image]**
 - **docker image rm node**
- Sie können ein Image nur löschen, wenn bei dir kein Container mehr existiert, der darauf basiert
- Mehrere Images löschen
 - **docker image rm [Name vom Image] [Name vom Image] ...**
 - **docker image rm \$(docker images -a -q)**

Das Kommando **docker create**

- Wie kommt man nun von einem Image zu einem Container?
 - **docker [container] create**
 - **docker create nginx**
 - **docker create -it ubuntu**
 - **docker create --name my_ubuntu -it ubuntu**
 - Wenn Sie den Container im Terminal ausführen möchten, ist es wichtig die Flag **-t** (oder **--tty**) zu setzen
 - Terminal-Treiber hinzufügen, um später beim Betrieb des Containers ein Pseudo-Terminal mit dem Container verbinden zu können
 - Nachträglich nicht mehr möglich!
 - Zudem sollte noch die Flag **-i** (oder **--interactive**) gesetzt sein!



docker create



Das Kommando **docker start**

- Wie starten Sie nun/erneut einen Container?
 - **docker [container] start**
 - Sofern Sie den Container im Terminal ausführen möchten, müssen Sie noch die Flag **-i** (oder **--interactive**) setzen
 - Es muss i.d.R. ein Pseudo-Terminal mit dem Container verbunden sein (Flag **-t** beim Erzeugen des Containers), damit er die Benutzereingaben auch verarbeiten kann
 - **docker [container] start -i [Name oder Container ID]**

Das Kommando **docker run**

docker run

- **docker [image] pull**
Ein Image herunterladen (in der Regel von Docker Hub)
- **docker [container] create**
Einen Container auf Grundlage vom heruntergeladenen Image erstellen
- **docker [container] start**
Den neu gebauten Container ausführen

Die Kommandos **docker stop**, **docker pause** und **docker unpause**

- Wenn Sie einen Container im Terminal ausführen, reicht es aus, das darin ausgeführte Programm zu beenden, um auch den Container zu stoppen
 - z.B. Container mit Ubuntu, Node
- Ansonsten können Sie einen Container mit dem Befehl **docker stop [ID oder Name]** beenden
- Weiter gibt es noch die Befehle **docker pause [ID oder Name]** und **docker unpause [ID oder Name]**, um einen Container vorübergehend anzuhalten bzw. seine Ausführung fortzusetzen

Wie löscht du Container?

- Mit dem Kommando **docker [container] rm [Name oder ID vom Container]** kannst du einen Container auch wieder löschen
- Du kannst nur Container löschen, die nicht gerade aktiv sind
- Beispiel: alle inaktiven Container löschen
 - **docker rm \$(docker container ls -aq)**

Container: Programme starten

- In einem Container wird i.d.R. immer ein hinterlegtes default-Programm gestartet
 - Bei Ubuntu ist es z.B. die bash, bei Node ist es die REPL (Read-Eval-Print-Loop)
- Hierbei ist die Schreibweise wie folgt:
 - **docker container run [OPTIONS] IMAGE [COMMAND]**
 - **docker container create [OPTIONS] IMAGE [COMMAND]**
 - Bsp: **docker container run -it node bash**

Container umbenennen: docker rename und die Flag --name

- Einen Container immer über seinen zufallsgenerierten Namen oder die ID anzusteuern ist ziemlich unpraktisch
- Also sollten wir unseren Container umbenennen, wofür wir verschiedene Möglichkeiten haben
 - Namen nachträglich ändern
 - `docker (container) rename [alter Name] [neuer Name]`
 - `docker rename [alter Name] ubuntu-neu`
 - Eigenen Namen beim Erzeugen festlegen mit der Flag `--name`
 - `docker create --name ubuntu-neu -it ubuntu`
 - `docker run --name ubuntu-neu -it ubuntu`
- Namen von Containern müssen eindeutig sein!

Container: Zusätzliche Programme starten

- Mit Hilfe von folgendem Befehl können Sie (mehrere) Programme innerhalb eines gestarteten oder im Hintergrund laufenden Containers starten:
 - **docker container exec [OPTIONS] CONTAINER COMMAND**

Aufgabe: Images

- Finden Sie auf Docker Hub das offizielle Node.js-Image
 - Welche drei Hauptvarianten werden in der Dokumentation für das Image aufgeführt und was sind die Unterschiede zwischen ihnen?
- Laden Sie das aktuelle **node:alpine**-Image herunter
 - Wie gross ist es im Vergleich zum Standard-Node.js-Image?
- Benennen Sie das heruntergeladene Image in `small_node` um.
- Überzeugen Sie sich, dass bei Ihnen jetzt ein Image namens `small_node` existiert.
- Erzeugen Sie und starten Sie einen Node.js-Container basierend auf dem `small_node`-Image, welcher automatisch wieder gelöscht werden soll.
- Versuchen eine Verbindung zum `small_node`-Container aufzubauen und führen Sie die `bash` aus.
 - Welche Fehlermeldung tritt dabei auf?
- Löschen Sie beide Node.js-Images, die auf Alpine basieren.

Aufgabe 1: Container

- Erzeugen Sie einen neuen Node.js-Container namens `test_container` auf Grundlage des Standard-Node.js-Images und führen Sie die `bash` aus.
 - Dieser Container soll nach dem Beenden NICHT automatisch gelöscht werden.
- Steuern Sie mit der `bash` im Container den Ordner `etc` an und finden Sie die Version der zugrunde liegenden Debian-Distribution heraus (steht in der Datei: „`debian_version`“).
 - Tipp: mit dem `cat`-Befehl können Sie sich Dateien auslesen
- Gehen Sie dann zurück in den `root`-Ordner und von dort zu `/usr/share`
 - Finden Sie in dem Ordner Hinweise zu einer anderen Programmiersprache, die wir schon verwendet haben und die in diesem Container installiert ist?

Aufgabe 2: Container

- Benennen Sie den Container test_container um in my_node-app
- Erstellen Sie dann im umbenannten Container den Ordner /app
- Wechseln Sie anschliessend in diesen Ordner und erstellen Sie eine Datei „main.js“ mit folgendem Inhalt:
 - console.log("Hallo M347");
- Führen Sie anschliessend Ihr Skript via **node main.js** aus.

Container im Hintergrund ausführen

- **Zuerst:**
 - Wenn in einem Container der Hauptprozess beendet wird, wird auch der Container beendet (z.B. wenn wir die Shell beenden)
- Damit ein Container im Hintergrund laufen kann, braucht er einen Prozess, der gestartet wird und am Laufen bleibt.
 - Bei `docker container start` läuft der Container standardmässig bereits im Hintergrund (sofern wir nicht `-i` angeben)
 - Bei **docker run**:
 - Mit der Flag **-d** (detached) kannst du einen Container im Hintergrund laufen lassen
 - Wenn ein Container im Hintergrund läuft, können Sie mit **docker logs** die Ausgabe einsehen.

Container automatisch starten

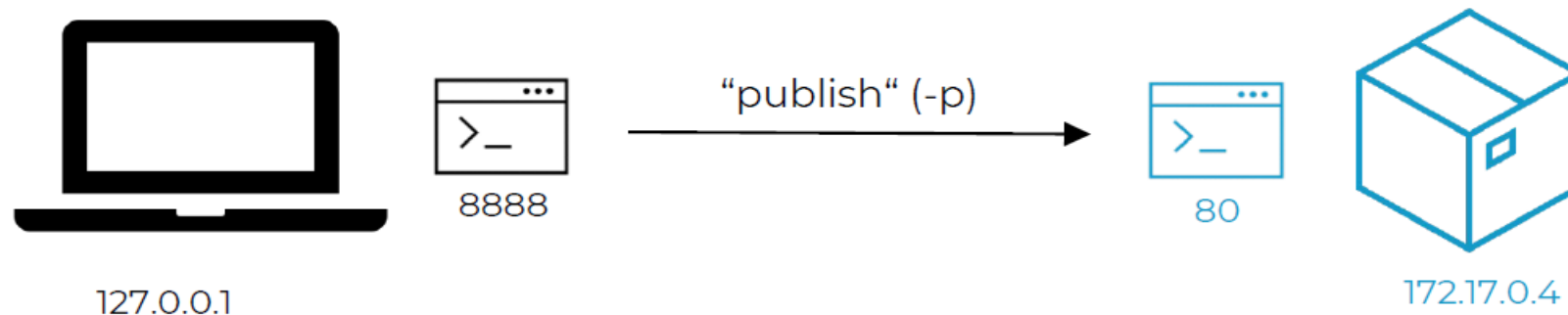
- Standardmässig wird ein Container nicht erneut gestartet.
- Wir können aber die **restart policy** auf folgende Werte setzen:
 - **on-failure[:max-retries]**: Startet den Container neu, wenn es einen Error gibt
 - **always**: Der Container wird immer neu gestartet (es sei denn, wir stoppen ihn, dann wird er erst beim nächsten Start vom Docker Daemon erneut gestartet).
 - **unless-stopped**: Der Container wird immer neu gestartet (und wenn wir ihn stoppen, bleibt er gestoppt)
- Wie setzen wir die restart policy?
 - **docker run -d --restart always [Image-Name]**
 - **docker update --restart always [Container-ID / Name]**

Einen Port mittels Flag -p veröffentlichen

- Wir führen Docker auf localhost (127.0.0.1) aus.
- Jeder Container besitzt eine eigene (virtuelle) Netzwerk-Schnittstelle.
- Standardmässig werden Ports vom Host nicht an den Container weitergeleitet.
- Dadurch können z.B. mehrere Container jeweils Port 80 verwenden, da ja jeder Container eine eigene Netzwerk-Schnittstelle hat.
- Mit der Flag **-p** (oder **--publish**) können Sie den Port eines Containers veröffentlichen, um mit Diensten ausserhalb von Docker (z.B. deinen Browser) auf den Container zuzugreifen
 - Wir können uns das so vorstellen: Wir leiten einen Port von unserem Host zu einem Port auf unserem Container weiter
 - **docker run -p Host-Port:Container-Port Image-Name**
 - Die beiden Ports können natürlich unterschiedlich sein

Einen nginx – Webserver starten

- nginx ist eine weit verbreitete Software für Webserver
- **docker run -p 8088:80 nginx**
- Danach können wir im Browser via `http://localhost:8088` den Webserver auf unserem Port 8088 ansteuern
- Der Container-Port muss allerdings 80 (HTTP) lauten, da nginx standardmässig auf Port 80 lauscht



Docker container create

- Auch beim Erstellen von einem Container kannst du eine Port-Weiterleitung spezifizieren:
 - **docker container create -p Host-Port:Container-Port Image-Name**
- Warum beim Erstellen des Containers, und nicht beim Starten?
- Naja, das Starten kann ja automatisch passieren (je nach restart-policy)
- Daher muss das beim Erstellen eines Containers spezifiziert werden

Aufgabe: (HTML-Seite im nginx-Container anpassen)

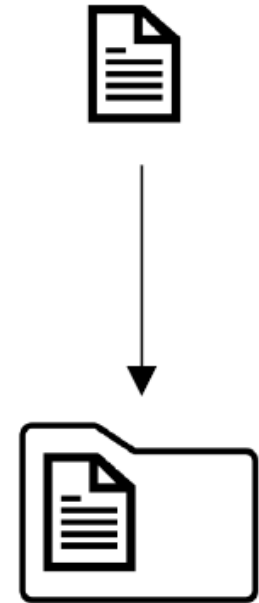
Ziel dieser Aufgabe ist es, die Willkommensseite vom nginx-Container, die standardmässig im Browser angezeigt wird, zu modifizieren

- Starten Sie dazu einen nginx-Container und verbinden Sie einen beliebigen, freien Port zu ihm
 - Erinnerung: Der nginx-Webserver erwartet eingehende Verbindungen auf Port 80!
- Greifen Sie auf die bash von Ihrem nginx-Container zu (während der Webserver läuft)
- Navigieren Sie über die bash in das Verzeichnis `/usr/share/nginx/html`
- Verändern Sie dann den Inhalt der Datei `index.html`, speichern Sie die Änderungen ab und aktualisieren Sie die URL in deinem Browser
- Dazu kann es wie üblich hilfreich sein, den Editor nano nach zu installieren

Das Kommando docker cp

- Mit dem Befehl docker cp können Sie schnell Dateien oder Ordner in einen Container und aus einem Container kopieren
- Du müssen Sie zwei Argumente übergeben:
 - Name bzw. Pfad zu dem Element (Datei oder Ordner), das kopiert werden soll
 - Verzeichnis, in das das Element kopiert werden soll
- Element vom Host-System **in einen Container** hinein kopieren:
 - **docker cp [Element im Host-System bzw. Pfad dahin] [Name/ID des Containers]:[Verzeichnis im Container]**
- Element **aus einem Container** heraus in das Host-System kopieren:
 - **docker cp [Name/ID des Containers]:[Element im Container bzw. Pfad dahin] [Verzeichnis im Host-System]**
- Das Kommando docker cp funktioniert auch dann, wenn der Container gerade nicht aktiv ist.
- Beispiel: [Webseite](#)

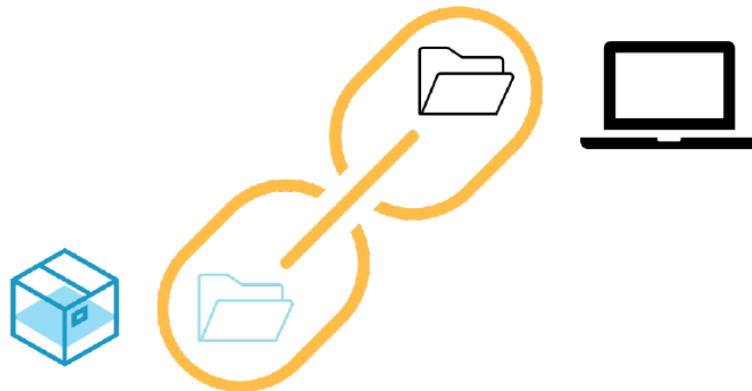
1. Parameter: Speicherort von Datei/Verzeichnis (in Container oder Host-System)



2. Parameter: Neuer Speicherort von Datei/Verzeichnis (in Container oder Host-System)

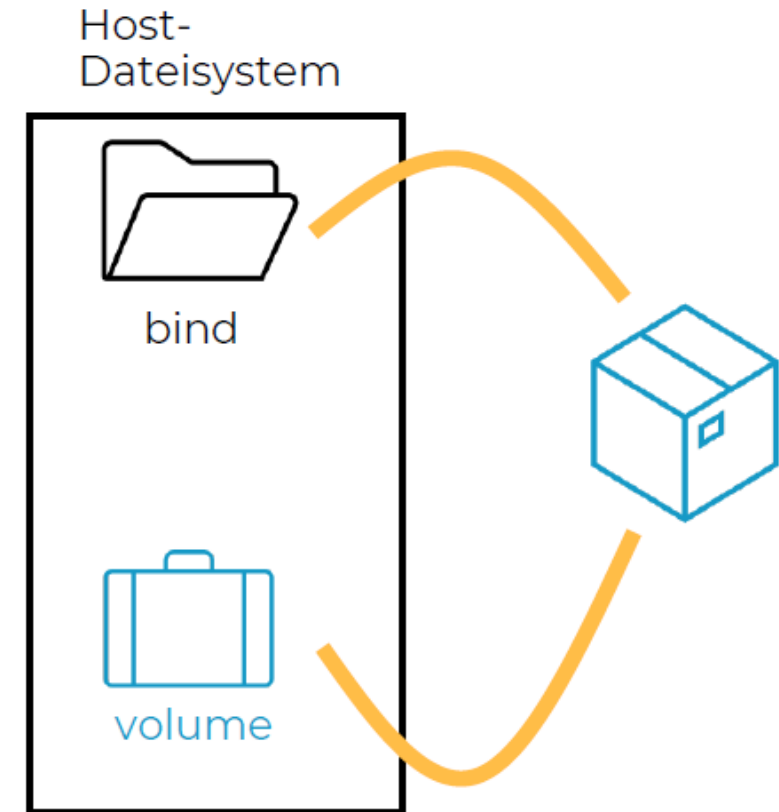
Was sind Mounts?

- Als Mounts bezeichnet man allgemein die Verknüpfung von der Verzeichnisstruktur eines Containers mit Daten, die ausserhalb vom Container liegen.
 - Das erlaubt Ihnen Dateien, die in Containern erzeugt und genutzt werden, **persistent** zu sichern.
 - Dateien bleiben auch nach Löschen des Containers bestehen



Welche Arten von Mounts gibt es?

- Es gibt drei verschiedene Arten von Mounts
 - **bind**
 - Ein Container wird mit deinem lokalen Dateisystem verknüpft
 - Du kannst dann von deinem Container aus in einem Ordner des Host-Systems operieren
 - **volume**
 - Bestimmte Daten in deinem Container werden an einen Docker spezifischen Speicherort ausgelagert
 - Volume's werden (in der Regel) mit Docker verwaltet
 - **tmpfs**
 - nur unter Linux verfügbar!



Wie führst du einen Bind Mount durch?

- Zur Erinnerung, bind mount:
 - Ein Container wird mit deinem lokalen Dateisystem verknüpft
- Es gibt zwei Schreibweisen, um einen *bind mount* durchzuführen
- Schauen wir uns zuerst die ausführlichere Schreibweise an:
 - Flag **--mount** setzen und spezifizieren
 - **type=bind**
 - **source**: enthält den Pfad zum lokalen Verzeichnis, das du mit der Verzeichnisstruktur des Containers verknüpfen möchtest
 - **destination**: enthält den Pfad zum Verzeichnis im Container
- Insgesamt:
- **--mount type=bind,source=[Pfad zu deinem lokalen Verzeichnis],destination=[Pfad zum Container-Verzeichnis]**

Die Flags `--mount` vs. `-v`

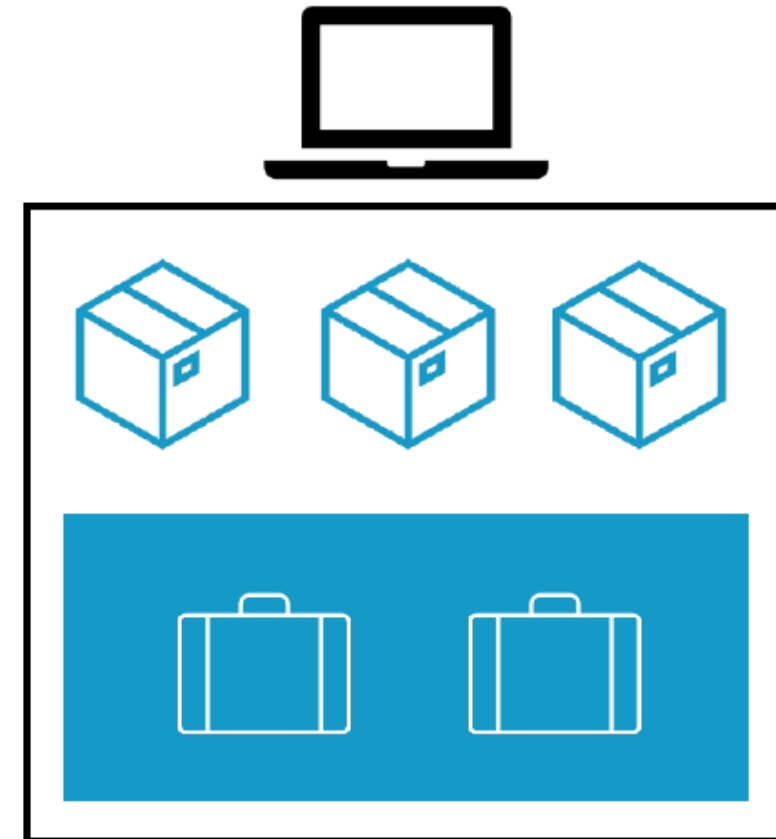
- Es gibt noch eine alternative (und kürzere) Schreibweise für einen bind mount
 - Du kannst auch die Flag `-v` (oder: `--volume`) verwenden:
 - `-v [absoluter Pfad zu deinem lokalen Verzeichnis]:[Pfad zum Container-Verzeichnis]`
 - Du musst immer den absoluten Pfad zu deinem lokalen Verzeichnis angeben!
- Der wesentliche Unterschied zwischen den beiden Schreibweisen besteht darin, dass Docker bei `-v` ein neues Verzeichnis im Host-System anlegen wird, falls es noch nicht existieren sollte, während bei `--mount` das Verzeichnis in *src* tatsächlich existieren muss (sonst: Fehlermeldung)
- Die Syntax mit der Flag `-v` ist übersichtlicher, daher wird sie häufiger verwendet

Aufgabe: Copy & Bind Mount

- Erstellen Sie einen Webserver mit PHP und Apache (Image: php, z.B. **php:8.1-apache**), und liefern Sie darüber eine PHP-Webseite aus.
- Schauen Sie dazu auf Dockerhub nach, in welchem Pfad der Webserver die Daten erwartet.
- Das Projekt kann ein Ordner mit einer "[index.php](#)"-Datei sein, mit folgendem Inhalt:
 - `<?php phpinfo(); ?>`
- **Aufgabe 1)**
 - Kopieren Sie das Projekt in den Container hinein
- **Aufgabe 2)**
 - Erstelle ein Bind-Mount (wahlweise readonly), und liefere darüber das Projekt aus
- Was könnten die Vor- und Nachteile von Copy bzw. Bind-Mount sein?

Was ist ein Volume?

- **Was sind Volume's?**
 - Eine Möglichkeit, um Daten ausserhalb eines Containers zu speichern.
 - Besonders praktisch für z.B. Datenbanken.
- Volumes werden von Docker verwaltet, und sind daher besonders effizient.
- **Wo werden Volumes gespeichert (auf dem Host)?**
 - Unter Linux werden deine Volumes in deinem lokalen System in dem Verzeichnis `/var/lib/docker/volumes/` gespeichert.
 - Unter Windows / macOS wird dieses Verzeichnis im Dateisystem der entsprechenden Linux-VM angelegt.
 - Auf dieses Verzeichnis können (Windows / macOS) oder sollten (Linux) Sie nicht direkt zugreifen, sondern es ausschliesslich über `docker volume` ansteuern.



Wie erzeugst und verwaltest du ein Volume?

- Mit **docker volume create [Name]** können Sie ein neues Volume erzeugen.
 - **docker volume create ubuntu-vol**
- Alle Volumes anzeigen: **docker volume ls**
- Mit **docker volume inspect [Name]** können Sie ein einzelnes Volume untersuchen.
- Sie können auch wie sonst ein Volume mit **docker volume rm [Name]** löschen.
- Weiter können Sie mit **docker volume prune** alle Volumes löschen, die nicht mit mindestens einem Container verbunden sind.
- Es ist nicht möglich, ein Volume umzubenennen.
 - Sie müssten ein neues Volume erstellen, und dort alle Daten hinüberkopieren.

Volumes mit Container verknüpfen (--mount)

- So wie bei bind mounts können Sie auch den Parameter --mount dazu verwenden, um ein Volume zu verknüpfen
 - **type=volume**
 - **source**: der Name von deinem Volume
 - **destination**: enthält den Pfad zum Verzeichnis im Container, das Sie mit dem Volume verbinden möchten
- Insgesamt:
 - **docker container run --mount type=volume,source=[Ihr Volume],destination=[Pfad zum Container-Verzeichnis]**

Volumes mit Container verknüpfen (-v)

- Um ein Volume mit einem Container zu verknüpfen, können wir auch bei `docker run` oder `docker create` den Parameter **-v** bzw. **--volume** setzen.
 - **-v [Name von deinem Volume]:[Verzeichnis im Container]**
 - Achtung! Das ist sehr ähnlich zur Schreibweise von einem *bind mount*!
 - Sie brauchen das Volume zuvor nicht mit *docker volume create* erzeugt zu haben
 - Falls das Volume nicht existiert, wird Docker es automatisch anlegen
 - Das bedeutet auch, dass Sie bei dem Flag -v sehr gut aufpassen müssen, dass Sie nicht ein neues Volume erzeugen, wenn Sie stattdessen eigentlich einen *bind mount* durchführen wollen! (*bind mount* → Pfad), (*volume* → Name)
- Sie können natürlich auch einen Container mit mehreren Volumes verbinden (einfach -v mehrfach angeben)

Automatisch generierte Volumes

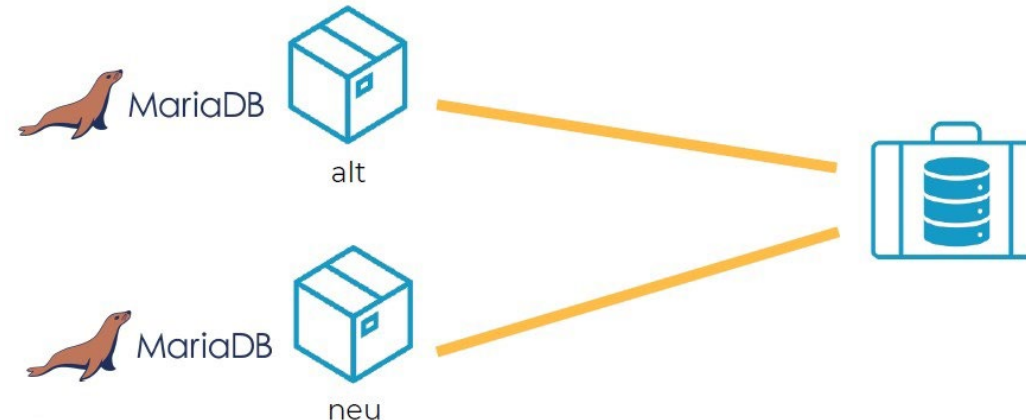
- Bei manchen Containern wird auch ohne explizite Spezifikation ein Volume angelegt:
- Zum Beispiel beim Datenbank-Management-System **mariadb** bzw. **mySQL**
 - **docker run -it -e MYSQL_ROOT_PASSWORD=sml12345 mariadb**
 - Per Flag **-e** bzw. **--env** muss hier eine Umgebungsvariable gesetzt werden
 - **docker container inspect -f "{{.Mounts}}"** [Containername / ID]
 - Das Volume taucht natürlich auch bei *docker volume ls* auf
- Zur besseren Kontrolle sollten Sie aber ein eigenes benanntes Volume einführen
 - **docker run -it -v mariadb-vol:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=sml12345 mariadb**

Aufgabe: Datenbanksoftware ohne Datenverlust aktualisieren

- Sie wollen einen Container, in dem eine MariaDB (mariadb:10.5) läuft, updaten.
 - Erstellen Sie die Ursprungssituation mit dem Begleitmaterial „[Project.sql](#)“.
 - Dazu werden Sie die Datenbank in einem selbst benannten Volume sichern
- Sie updaten den Container, indem Sie einen neuen Container erstellen mit der neusten Version von MariaDB
- **Wichtig:**
 - mariadb ist i.d.R. nicht abwärtskompatibel: Wenn Sie stattdessen die Version herabstufen, dürfte der Container abstürzen!

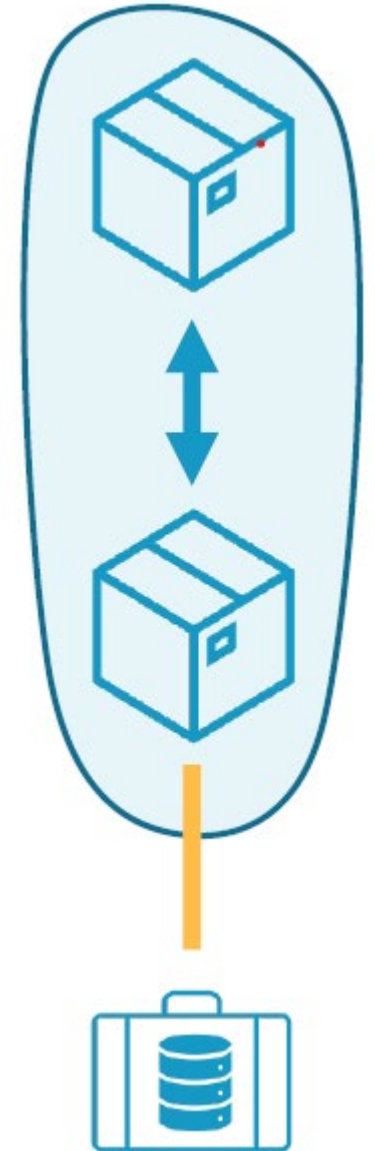
An der DB anmelden:

```
# mysql -u root -p  
(Passwort eingeben)  
> SHOW DATABASES;  
> ...
```



Kommunikation zwischen Containern

- Wir wollen mehrere (eigenständig) laufende Container miteinander verbinden
- In Docker läuft jeder Dienst i.d.R. in einem eigenen Container
- Beispiel:
 - 1 Container für PostgreSQL
 - 1 Container für pgAdmin (Verwaltungssoftware für PostgreSQL)
- Aber wie verbinden wir die Container?
- Lösung: **Wir richten ein Netzwerk ein!**
- Die Interaktion zwischen mehreren Containern ist eine der grössten Stärken von Docker und macht die Container-Technologie so mächtig

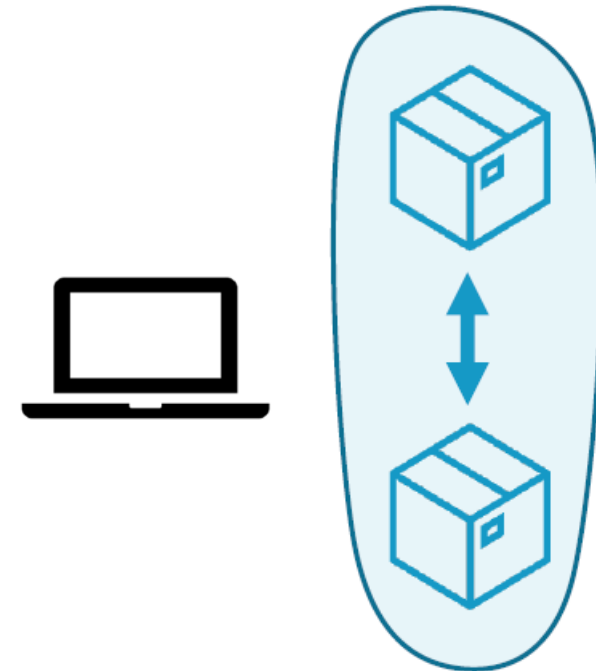


Netzwerk-Treiber in Docker

- Standardmässig laufen bereits einige Netzwerke in docker
 - **docker network ls**
 - Die Standard-Netzwerke heissen: **bridge**, **host** und **none**
 - Später werden noch weitere dazu kommen
 - Jedes **Netzwerk** hat einen eigenen **Namen** sowie eine **Network ID**
 - Genauer gibt es verschiedene Netzwerk-Treiber (Driver) und für jeden davon ein Standard-Netzwerk
 - Diese Treiber fungieren als Vorlagen, die das Verhalten von Containern spezifizieren
 - Wie üblich können wir mit dem Kommando *docker inspect* ein Netzwerk genauer untersuchen
 - **docker network inspect bridge**

Welche Netzwerk-Treiber gibt es in Docker?

- **Bridge:** ermöglicht die Kommunikation zwischen eigenständigen („standalone“) Containern bei Isolation von Containern von ausserhalb
- Genauer wird dabei ein privates Subnetz angelegt mit IP-Adressen der Form 172.???.??
- Container müssen dazu auf demselben Host laufen (bei uns bisher immer der Fall)
- Wenn wir nichts angeben, landet ein Container im bridge-Netzwerk
- Wenn ein Port vom Host aus weitergeleitet werden soll, müssen wir dies manuell angeben (Parameter: -p)



Netzwerk-Treiber in Docker

- **Host:** Container werden nicht vom Host-System isoliert
 - Container mit diesem Treiber sind unter der IP-Adresse des Hosts erreichbar
 - Auf einem Host-System kann es immer nur **ein** Host-Netzwerk geben, das schon besteht; Sie können kein weiteres erzeugen.
 - **Wichtig:** Funktioniert nur auf Linux-Hosts
- **None:** die Netzwerkfunktionalität des Containers wird deaktiviert
 - Es kann kein None-Netzwerk erzeugt werden
 - Beim Erstellen eines Containers die Flag **--network none** setzen

Ein neues (bridge-)Netzwerk erstellen

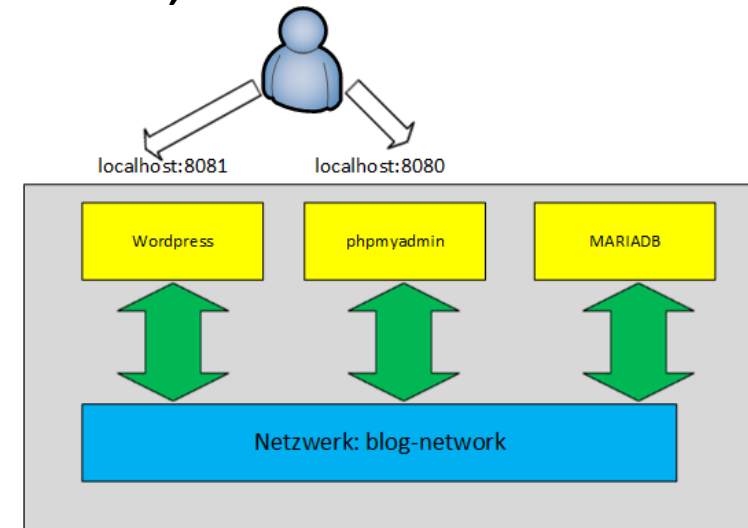
- Sie können ein neues bridge-Netzwerk via **docker network create** erstellen
- Anschliessend können wir einen Container über den Parameter **--network** zu diesem Netzwerk verbinden:
 - **docker network create my-network**
 - **docker container run --network my-network**
- Mit **docker network rm** bzw. **docker network prune** können Sie (ungenutzte) Netzwerke auch wieder löschen
 - Die drei Standard-Netzwerke (bridge, host und none) können Sie nicht löschen
- Warum ein eigenes Netzwerk?
 - Abschottung von anderen Containern
 - Gerade im produktiv-Betrieb sicherer und stabiler

Container nachträglich verbinden

- Sie können einen Container via **docker network connect [Name/ID vom Netzwerk] [Name/ID vom Container]** nachträglich mit einem Netzwerk verbinden
 - **docker network connect bridge my-mariadb**
 - Per **docker network inspect bridge** können Sie dann prüfen, ob der Container dem Netzwerk angehört
- Mit **docker network disconnect** können Sie einen Container auch wieder aus einem Netzwerk entfernen
 - **docker network disconnect bridge my-mariadb**

Vorbereitung zur nächsten Aufgabe: MariaDB & phpmyadmin

- Wir werden ein eigenes Netzwerk anlegen.
- In dem Netzwerk starten wir einen MariaDB-Container...
- ... und einen phpmyadmin-Container.
- Wir werden phpmyadmin so konfigurieren, dass wir uns zu unserer MariaDB verbinden können.



Aufgabe: (MongoDB & MongoExpress)

So ähnlich wie bei der Verbindung von einem mariaDB-Container und einem phpMyAdmin-Container geht es hier darum eine Datenbank (MongoDB) und ein grafisches Webinterface für diese Datenbank (Mongo Express) auf Container-Ebene in einem Netzwerk miteinander zu verbinden.

- Verwendete Images:
 - MongoDB: https://hub.docker.com/_/mongo
 - Mongo Express: https://hub.docker.com/_/mongo-express

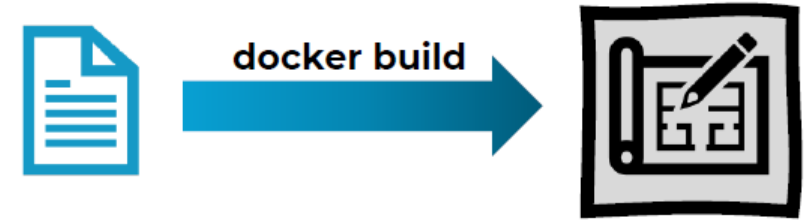
Aufgabe: (MongoDB & MongoExpress)

- Erstellen Sie ein bridge-Netzwerk namens *mongo-net* und starten Sie darin einen Mongo-Container mit dem Namen *my-mongo*. Verbinden Sie im Zuge dessen auch das Verzeichnis */data/db* im Container direkt mit einem Volume *mongo-vol*.
- Tipp: Es macht Sinn den Container im Hintergrund laufen zu lassen.
- Falls Sie den Container erst genauer untersuchen wollen: Auch dieser verwendet ein Debian-Image (bash...)

Aufgabe: (MongoDB & MongoExpress)

- Starten Sie nun, ebenfalls in dem Netzwerk *mongo-net*, einen Mongo Express-Container, den Sie zum MongoDB-Container verbinden. Denken Sie daran, einen Port vom Host auf den Port 8081 des Mongo Express – Containers weiterzuleiten, damit Sie die Mongo-Express-Oberfläche aufrufen können.
- **Zudem:**
 - Damit Mongo Express sich zu MongoDB verbindet, müssen Umgebungsvariablen gesetzt werden.
 - Versuchen Sie erst mithilfe der Dokumentation auf Docker Hub selbst herauszufinden, welche Umgebungsvariable Sie setzen müssen.
 - Öffnen Sie schliesslich die Anwendung im Browser.

Dockerfiles und docker build



- **Problem:**

- CLI-Kommandos können umfangreich und unübersichtlich werden.
- Häufig wollen wir Container verwenden, die auf eine ähnliche Weise konfiguriert sind.
- Zudem können wir manche Befehle nicht automatisieren (z.B. das Nachinstallieren und Konfigurieren von Software).

- **Lösung:**

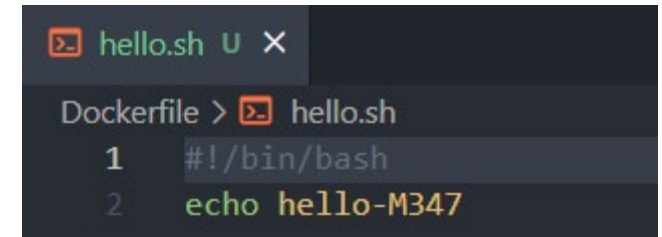
- Wir schreiben unsere Konfigurationen für ein neues Image in eine Datei (genannt **Dockerfile**) und führen dann das Kommando **docker build** aus, um ein für unsere Zwecke massgeschneidertes Image zu erstellen.

Was ist ein Dockerfile?

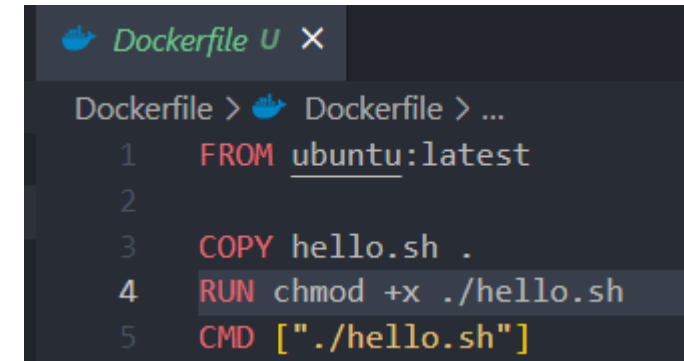
- In einem **Dockerfile** wird ein neues Image anhand von Anweisungen entsprechend einer vorgegebenen Syntax konfiguriert
- Jede Zeile hat die Form
 - **[ANWEISUNG] [Parameter]**
 - Nach Konvention werden die Anweisungen immer gross geschrieben (um sie von den Parametern besser unterscheiden zu können); sie sind aber nicht *case-sensitive*
- Kommentare beginnen mit **#** und werden beim Bauen ignoriert
 - **#** muss am Anfang einer Zeile stehen
- Ein **Dockerfile** hat keine Endung und heisst standardmässig **Dockerfile**

Docker build (Bsp. «hello-docker»)

- Unsere Beispiel-«Applikation» besteht aus einer Bash-Datei (hello.js) mit einem Befehl.
- In die Datei mit Namen `Dockerfile` schreiben wir die Befehle, wie unsere Applikation in ein Image gepackaged wird. Meist startet man mit einem Base-Image, z. B. ubuntu
- Auf hub.docker.com findet man viele Images, auf denen man aufbauen kann.
- Mit dem Befehl
`docker build -t hello-m347 .`
wird unsere Applikation in ein Image gepackaged.
- Sie können auch ein Git-Repository via URL ansteuern.
- Mit dem Flag **-t** bzw. (**--tag**) können Sie dem Image einen neuen *tag* zuweisen, um komfortabler darauf zugreifen zu können.
- Best Practice: Speichere das Dockerfile in einem leeren Verzeichnis, bzw. darin sollten sich nur die Dateien befinden, die notwendig sind, um das Image zu bauen.



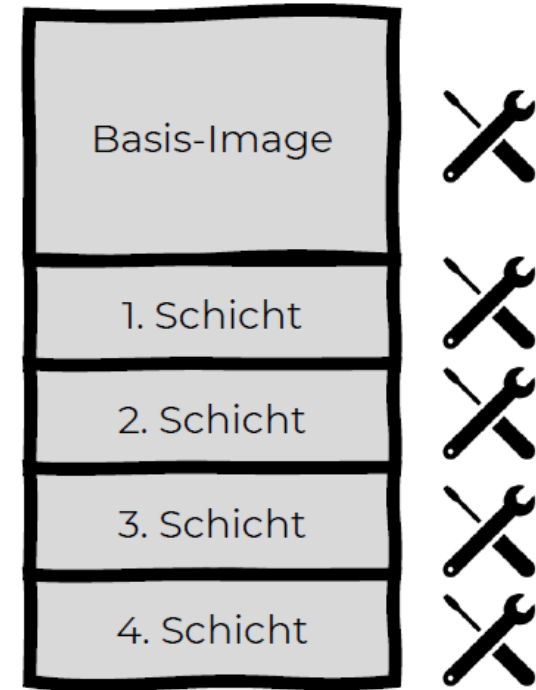
```
hello.sh U X
Dockerfile > hello.sh
1  #!/bin/bash
2  echo hello-M347
```



```
Dockerfile U X
Dockerfile > Dockerfile > ...
1  FROM ubuntu:latest
2
3  COPY hello.sh .
4  RUN chmod +x ./hello.sh
5  CMD [\"./hello.sh\"]
```

Was passiert bei docker build?

- Zunächst wird das gesamte *Dockerfile* validiert: bei einem Syntax-Fehler wird keine Anweisung ausgeführt und Sie erhalten eine Fehlermeldung („[internal]“).
- Falls die Validierung fehlerfrei abgeschlossen wurde, werden danach nacheinander die einzelnen Anweisungen ausgeführt
 - Bestimmte Anweisungen können neue Schichten erzeugen
- Ganz am Ende wird eine ID für das Image generiert
- Docker ist hierbei effizient:
 - Um den Bau-Vorgang zu beschleunigen, nutzt Docker seinen *build-cache* („CACHED“)



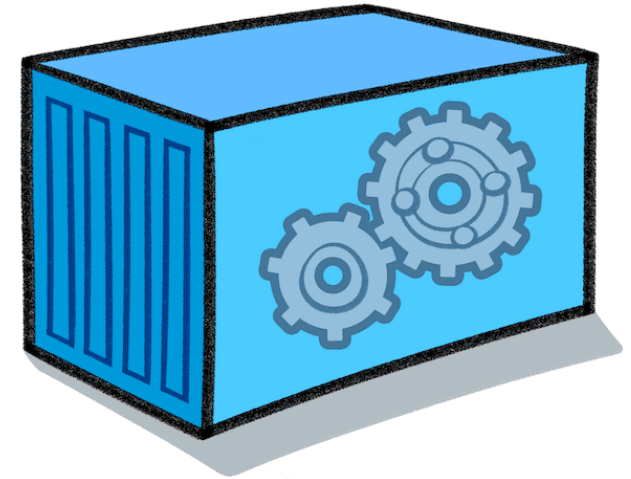
Der Prozess



Docker File



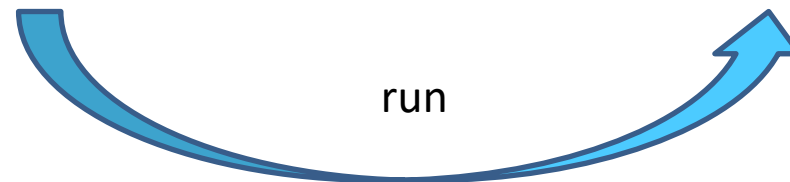
Docker Image



Docker Container



build

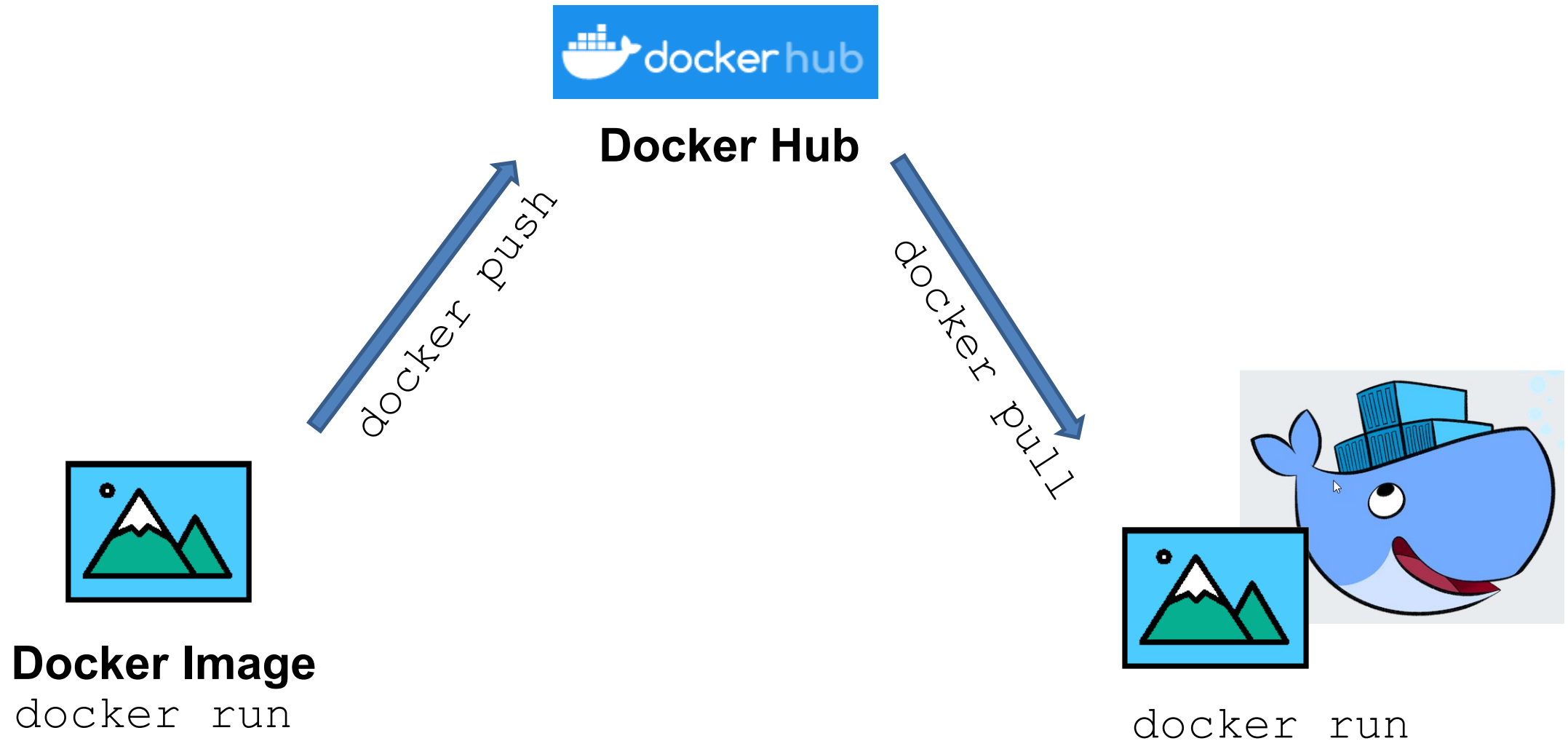


run

Docker run, push und pull

- Aus unsere Beispiel-«Applikation» wurde mit dem build-Befehl ein Image erstellt.
- Mit dem Befehl `docker run <image name>` startet man einen Container auf der eigenen Maschine.
- Das Image kann man z. B. auf eine Plattform wie Docker Hub (registrieren und login) hochladen mit dem Befehl `docker push`.
- Unter <https://labs.play-with-docker.com/> kann man einen anderen Client simulieren: VM starten (mit Linux + Docker Engine) und mit `docker pull` das Image von Docker Hub herunterladen und mit `docker run` starten.

Docker run, push und pull



Dockerfile: Kommandos 1

Anweisung	Bedeutung
FROM	<p>Gibt das Basis-Image an. FROM [Image]:[tag]</p> <p>Es können auch mehrere FROM-Anweisungen in einem Dockerfile stehen.</p> <ul style="list-style-type: none"> <i>Multi-Stage Build</i> <p>Das Image, von dem alle anderen abstammen ist das <i>scratch-Image</i></p> <ul style="list-style-type: none"> <i>FROM scratch</i> (von Grund auf 🎨) https://hub.docker.com/_/scratch/
RUN	<p>Befehle, die beim Bau des neuen Images ausgeführt werden sollen. Sie dienen in der Regel dazu, das Basis-Image durch Installation zusätzlich Pakete zu erweitern.</p> <ul style="list-style-type: none"> Ubuntu/Debian: RUN apt-get update && apt-get install -y [Paket] <p>Jede RUN-Zeile fügt eine neue Schicht zum Image hinzu. → Mit && bündeln</p>
COPY	<p>Kopiert Dateien aus dem Projektverzeichnis in das Image.</p> <p>Jede COPY-Anweisung fügt eine neue Zwischenschicht zum Image hinzu.</p>

Dockerfile: Kommandos 2

Anweisung	Bedeutung
ADD	<p>Kopiert Dateien in das Dateisystem des Images.</p> <ul style="list-style-type: none">• Erlaubt auch per URL, Daten aus dem Internet zu kopieren<ul style="list-style-type: none">• Besser mit RUN und curl oder wget verwenden!• Entpackt Archive automatisch (gzip, tar, bzip2, xz)<ul style="list-style-type: none">• In diesem speziellen Fall ADD statt COPY verwenden
CMD	<p>Führt das angegebene Kommando beim Container-Start aus. Empfohlen: Exec-Form (Auf die Shell-Form gehe ich nicht ein)</p> <ul style="list-style-type: none">• CMD ["Programm / Datei", "Parameter 1", "Parameter 2"] <p>Wichtig: Es wird immer nur das letzte CMD-Kommando beachtet!</p>
ENTRYPOINT	<p>Führt das angegebene Kommando immer beim Container-Start aus. Es können zusätzliche Parameter zum Kommando beim ausführen von <i>docker run ...</i> mitgegeben werden.</p> <p>Gibt es ENTRYPOINT und CMD, so werden die Kommandos aus CMD an den ENTRYPOINT drangehängt</p>

Dockerfile: Kommandos 3

Anweisung	Bedeutung
WORKDIR	Legt das Arbeitsverzeichnis für RUN, CMD, COPY etc. fest.
ENV	Setzt eine Umgebungsvariable.
VOLUME	Gibt Volume-Verzeichnisse an.
EXPOSE	Gibt die aktiven Ports des Containers an.
LABEL	Legt eine Zeichenkette fest.
USER	Gibt den Account für RUN, CMD und ENTRYPOINT an.

Beispiel: Flask-Applikation in Dockerfile

<https://flask.palletsprojects.com/en/2.2.x/quickstart/#a-minimal-application>

Aufgabe: VIM in Docker

- Vim ist ein Texteditor unter Linux, mit dem wir Dateien editieren können (ähnlich nano)
- Der Einarbeitungsaufwand ist aber etwas höher als bei nano
 - Tipp: Über :quit oder :qa können Sie den Editor wieder schliessen.
- **Aufgabe**
 - Erstellen Sie ein Docker Image, welches vim als Editor ausführt, und eine hinterlegte Datei (z.B. text.txt) standardmässig öffnet.
 - Sie können für diese Aufgabe z.B. ein Ubuntu als Ausgangsbasis verwenden, und vim nachinstallieren.
 - Vim soll bei diesem Image immer geöffnet werden (Entrypoint), der Name der Datei soll aber konfigurierbar sein.
 - Kombiniere dafür CMD und ENTRYPOINT

Aufgabe: Express.js-App

So wie wir ein Dockerfile für die Flask-App geschrieben haben, wollen wir nun für eine funktional ähnliche [Node.js-Anwendung](#) (genauer benutzen wir das Framework Express.js) ein Image mit einem Dockerfile erstellen.

Aufgabe:

- Schreiben Sie das Dockerfile, um die Anwendung zu containerisieren.
- Erzeugen Sie ein Image auf Basis von ihrem Dockerfile.
- Starten Sie einen Container, in dem die App ausgeführt wird und öffnen Sie diese im Browser.

Aufgabe: Express.js-App

Tipps:

- Als Basis verwenden Sie ein Node-Image.
- Ein sinnvoller Speicherort in dem Node-Image für die App-Dateien wäre das Verzeichnis: */webApp*
- Denken Sie daran, sowohl die *index.js* als auch die *package.json* zu kopieren.
- Um die Dependencies zu installieren brauchen Sie den Befehl **npm install**
 - Bezieht sich auf das *package.json* → dieses muss bereits im Image vorhanden sein
 - Sinngemäss ist das vergleichbar mit `pip3 install -r requirements.txt` bei unserer Flask-App.
- Den Web-Server können Sie per **node index.js** starten, es sind hier keine zusätzlichen Parameter notwendig.
- Den Container-Port ist 8080 (siehe *index.js*)

Prüfungsvorbereitung: Image manuell bilden 1

- Erstellen Sie ein neues Verzeichnis **flask-manual-build** erstellen und hineinwechseln.
- **hello.py** nach folgender Vorgabe <https://palletsprojects.com/p/flask/> erstellen.

- **start.sh** Skript nach folgender Vorgabe erstellen:

```
#!/bin/bash
cd /app
export FLASK_APP="hello"
export FLASK_ENV="development"
export FLASK_RUN_HOST="0.0.0.0"
flask run
```

- Python Container erstellen und einrichten. Erstellen Sie ein Verzeichnis **/app** und kopieren Sie die oben erstellten Files in das Container-app-Verzeichnis, installieren Flask und führen das start.sh-Skript aus.

Prüfungsvorbereitung: Image manuell bilden 2

- Starten Sie die Webanwendung im Container und überprüfen Sie auf dem Host das Ergebnis.
- Den Container wieder verlassen.
- Aus dem eingerichteten Container ein Image Version 1.0 erstellen und einen neuen Container daraus starten. Tipp: `docker commit ...`
- Auf dem Host das Ergebnis überprüfen.
- Das Image um den CMD-Befehl `/app/start` erweitern und ein neues Image Version 1.1 erstellen. Tipp: `docker commit --change ...`
- Einen neuen Container daraus starten und auf dem Host überprüfen.
- Lösungsvorschlag: https://github.com/IneichenBBZW/Image_manuell_bilden/

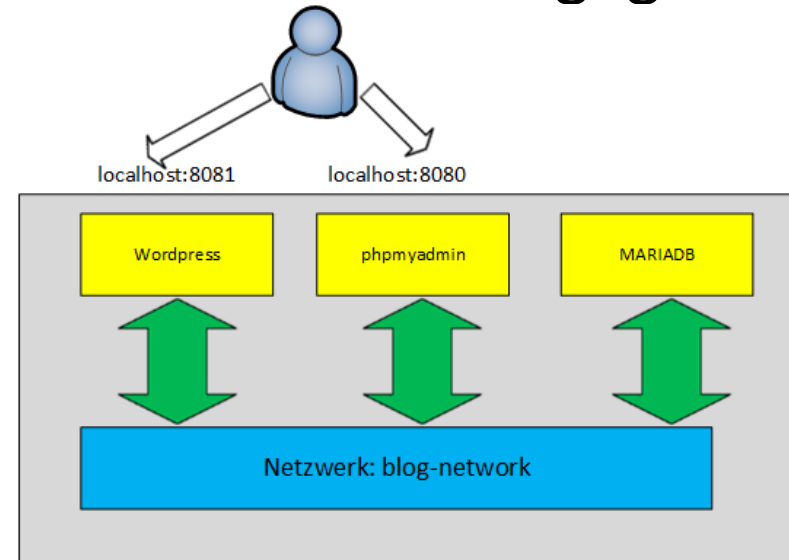
Prüfungsvorbereitung: Image automatisch bilden

- Benutzen Sie [hello.py](#) und [start.sh](#) aus der Übung Image manuell bilden.
- Erstellen Sie ein Dockerfile und automatisieren Sie darin die manuellen Schritte von vorher.
- Bilden Sie ein Image mit der Version 1.0 und Testen Sie es.
- Bauen Sie das Dockerfile um, so das Sie das [start.sh](#) nicht mehr brauchen. Tipp: Umgebungsvariablen
- Bilden Sie ein Image mit der Version 1.1 und Testen Sie es.
- Lösungsvorschlag: https://github.com/IneichenBBZW/Image_automatisch_bilden/

Prüfungsvorbereitung: MariaDB, phpmyadmin und Wordpress

- Erstellen Sie die Container MariaDB, phpmyadmin und Wordpress in einem bridge-network: blog-network
- Richten Sie die Container so ein, dass phpmyadmin über den Port 8080 und Wordpress über den Port 8081 vom Host zugegriffen werden kann.

- Lösungsvorschlag: [hier](#)



Wiederverwendbare Images

Konfigurationsmöglichkeiten:

- Konfigurationsfiles
- Umgebungsvariablen
- Kommandozeilenoptionen und Argumente

Lernen Sie dieses Thema mit folgendem [Beispiel](#).

Erstellungszeit versus Laufzeit-Ausführung

Erstellungsprozesse:

- Verzeichnisse erstellen
- Kopieren von Dateien und Verzeichnissen in das Image
- Metadaten ändern
- Linux-Pakete installieren
- Python-Bibliotheken installieren
- Startbefehl definieren

Studieren Sie die Unterschiede in folgendem [Beispiel](#).

Kleinere Images bilden

Machen Sie folgende Untersuchungen mit kleineren Python-Images!
Siehe: https://hub.docker.com/_/python Image Variants.

- Öffnen Sie den Browser mit: localhost:5000/86907523 und notieren Sie sich die Zeit!
- Notieren Sie die Grösse vom Image factors_flask...!
- Notieren Sie die Grösse von Python...-Image.
- Berechnen Sie die Differenz! Und Werten Sie es aus! (Prozent der Applikation in Bezug auf das Python...-Image.)

Benutzen Sie folgendes [Flask-Beispiel](#).

Entwicklungs- und Produktiv-Image in Frontendentwicklung

- Bei der Entwicklung des Frontend sind Anforderungen für die Entwicklung und Produktivbetrieb unterschiedlich.
 - Multi-Stage-Builds
- Bei Multi-Stage-Builds wird im Dockerfile eine **zweite FROM-Anweisung** hinzugefügt. Damit kann auf die Layers des ersten Teils (FROM-Anweisung) zugegriffen werden.
- Beispiel: Das Vue.js Framework setzt für die Entwicklung auf die Node.js- Runtime
 - Erster Teil: Installation von notwendigen Node.js-Modulen, Quellcode kopieren und Build-Prozess starten. Ergebnis wird in den /src/vue/dist Ordner gestellt.
 - Zweiter Teil: Webserver erstellen (z.B. nginx:alpine) und mit **--from-Anweisung** fertige Webapplikation in den Webserver kopieren.
- Beispielcode: https://github.com/IneichenBBZW/M347_Multi_Stage_Builds.git

Entwicklungs- und Produktiv-Image

Wir haben bereits betrachtet, wie man kleinere Images für unsere Flask-Webanwendung bildet. Unser neues Ziel ist die Faktorisierungsfunktion zu optimieren. Da alle Images, gross oder klein, den gleichen Python-Interpreter brauchen, bleibt die Ausführungszeit in etwa gleich.

Ziele:

- Mit Cython <https://cython.org/> C-Extensions für Python einen nativen C/C++ Code schreiben, der performanter ist.
- Multi-Stage-Builds: Für jede FROM-Anweisung wird ein neuer temporärer Container gebildet. Aber nur der Letzte wird gespeichert und getagged. Die anderen temporären Container werden verworfen.
- Die COPY-Anweisung mit `--from`-Option, ist die Möglichkeit, mit der man vom temporären Container Artefakte in den finalen Container kopieren kann.
- Aufgabe: Bilden Sie einen Multi-Stage-Builds mit dem Dockerfile.cython-multi aus [folgendem Projekt](#).

Aufgabe: Multi-Stage-Build für Go-App

Ein kleines [Go](#)-Projekt wurde [hier](#) erstellt. Sie haben folgende Aufgabe:

- Bilden Sie daraus ein Image und starten den Server. Aus dem Go-Code können Sie herauslesen, dass der Container-Port 8080 ist.
- Stoppen Sie den Server und prüfen Sie die Grösse des Images. → Das müsste kleiner sein!

Aufgabe: Machen Sie daraus ein Multistage-Build.

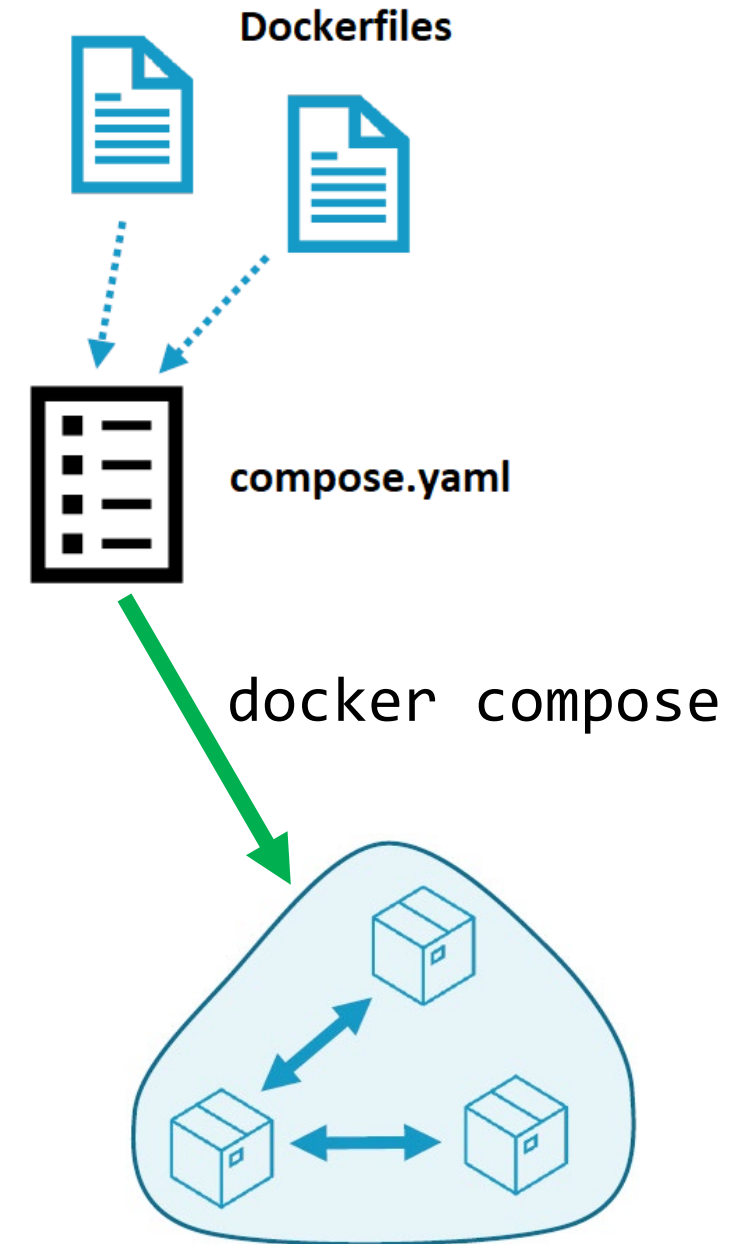
- Benutzen Sie als finales Basisimage «alpine»

Motivation Compose

- Oft benötigt eine Anwendung mehrere andere Anwendungen oder Container. Anstatt diese alle einzeln und nacheinander zu starten, hat man die Möglichkeit, dies einmalig zu definieren und mit dem Befehl `docker compose` die Container-Komposition zusammenstellen zu lassen.
- Die Komposition wird in einer sogenannten yaml-Datei mit Endung `.yaml` oder `.yml` beschrieben.

Was ist docker compose?

- In einer Konfigurationsdatei (üblicherweise `compose.yaml` oder `docker-compose.yaml` genannt) spezifizieren wir mehrere Container und ihre Verbindungen zueinander.
- Dann führen wir das Kommando `docker compose` aus.
- Damit wird automatisch ein **neues Netzwerk** erzeugt und die Container darin gestartet.
- Sie können `docker run` - Befehle als `compose.yaml` Konfigurationen umformulieren.



yaml-Syntax

- Text-Datei mit Unicode Zeichen.
- Struktur wird mit **Einrückung** (Leerschläge, Tabulatoren) gegeben.
- Kommentare starten mit #.
- Ein Eintrag hat die Form key: value. Nach dem Doppelpunkt muss ein Leerschlag stehen.
- Listen-Einträge starten mit einem Strich (-), jeder Eintrag steht auf einer Zeile. Oder man schreibt die Einträge kommagetrennt zwischen eckigen Klammern [entry1, entry2].
- Strings müssen nicht in Anführungszeichen stehen, können aber. In einfachen oder doppelten.
- Vergleich [JSON vs. YAML](https://www.json2yaml.com/): <https://www.json2yaml.com/>

Der Aufbau von compose.yaml

- Siehe: <https://github.com/compose-spec/compose-spec/blob/master/spec.md>
- Spezifikation der Version (version: '3') ganz oben ist veraltet.
- Services spezifizieren
 - Unsere Dienste (jetzt erstmal: Container die gestartet werden) können wir wie folgt definieren:
 - **services:**
 - Erforderlich!
 - Hier werden die einzelnen Container aufgeführt, die Sie konfigurieren und starten (und evtl. replizieren) möchten.
 - Oft: Ein Service entspricht einem Container.
 - Aber: Wir können auch sagen, dass ein Container mehrfach ausgeführt werden soll.
 - z.B. Service "webserver" soll 3x einen nginx-Container starten

Services definieren

- In der YAML wird jeder Service separat konfiguriert, z.B.
 - **image**: das Basis-Image festlegen
 - **build**: falls ein Image basierend auf einem *Dockerfile* gebaut werden soll
 - **volumes**: persistenten Speicherort festlegen
 - **ports**: Verbindung *Docker-Host:Container-Port* herstellen
 - **environment**: Umgebungsvariablen setzen
 - **command**: Befehle in Container ausführen
 - **restart**: Neustart-Verhalten festlegen (*no, always, on-failure, unless-stopped*)
 - **depends_on**: Abhängigkeiten zwischen Services definieren, sodass sie in der richtigen Reihenfolge gestartet und gestoppt werden können
 - **container_name**: Container benennen
 - **expose**: interne Container-Ports freigeben
 - **links**: den Service mit Containern in einem anderen Service aber innerhalb desselben Netzwerks verbinden
 - **secrets**: vertrauliche Daten verwenden, die in einer separaten Datei ausgelagert worden sind
 - Mehr: <https://docs.docker.com/compose/compose-file/#version-top-level-element>

docker compose up

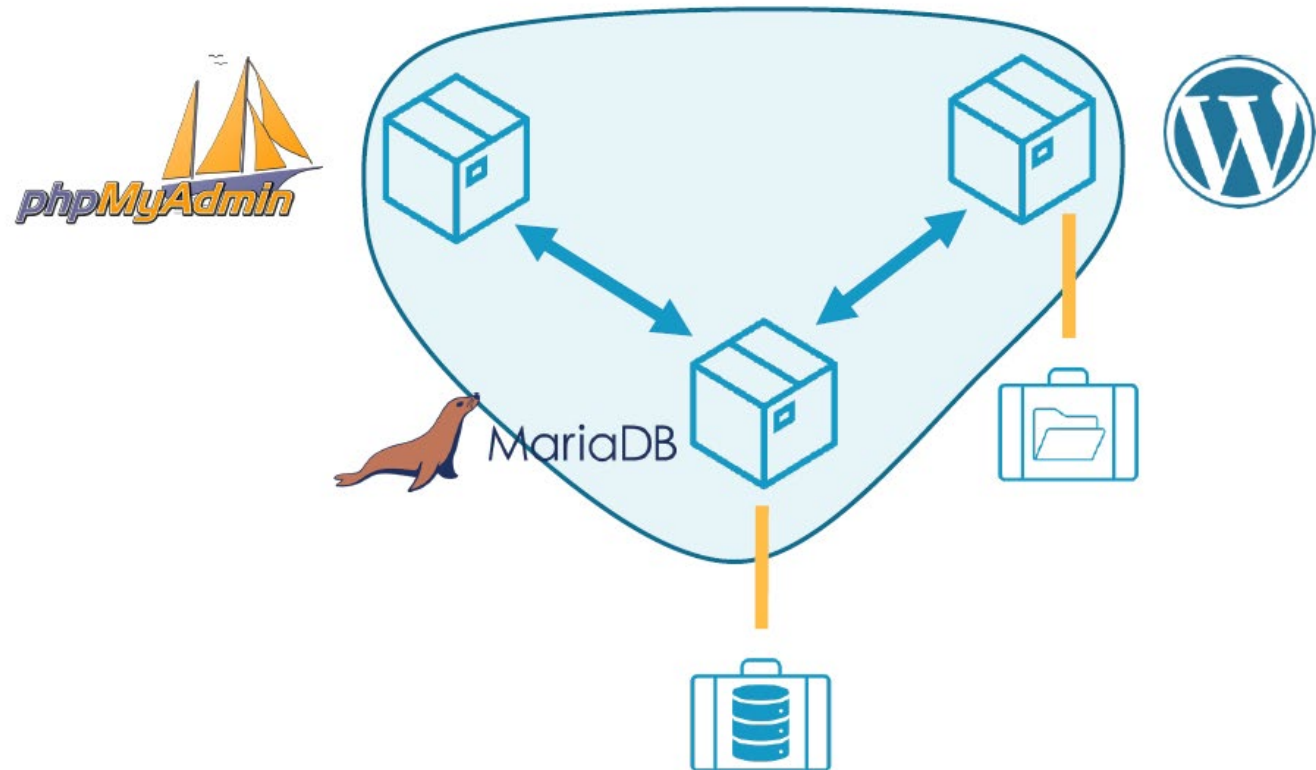
- Wie beim *Dockerfile* sollten Sie ein leeres Projektverzeichnis (Kontext) anlegen. Darin befindet sich eine *compose.yaml*, sowie benötigte Dateien.
- Was passiert beim Kommando ***docker compose up***?
 - Images werden gegebenenfalls heruntergeladen
 - Container erzeugt
 - Netzwerk erzeugt (bridge-Netzwerk, benannt nach dem Projektverzeichnis)
 - Container im Netzwerk gestartet
 - [Startbeispiel](#) + [Material](#)

Das Kommando docker compose

- Mehrere Container starten:
 - `docker compose up`
- Die Container stoppen, die Container und das Netzwerk automatisch löschen, Volumes bleiben natürlich bestehen:
 - `docker compose down`
- Weitere nützliche Sub-Kommandos
 - Container temporär stoppen:
 - `docker compose stop`
 - Container wieder starten:
 - `docker compose start`
 - Container pausieren / fortsetzen:
 - `docker compose pause`
 - `docker compose unpause`

Ein bekanntes Beispiel

- Wir wollen schrittweise dieses frühere Beispiel mit **docker compose** nachbauen und verbessern.
 - Container im Netzwerk per Umgebungsvariabler miteinander verbinden
 - Volumes hinzufügen
 - [Übung](#) + [Material](#)



Volumes konfigurieren

- Sie müssen jedes Volume immer zweimal eintragen.
 - Bei der Konfiguration des Services unter **volumes**:
 - wie bei Verwendung der **-v**-Flag in der Form
 - **my-vol:[Pfad zu Verzeichnis im Container]**
 - Natürlich auch bind mounts möglich
 - Zusätzlich nochmal unter dem Bezeichner volumes (äusserste Einrückungsstufe) in Form einer *Map*
 - **my-vol:**

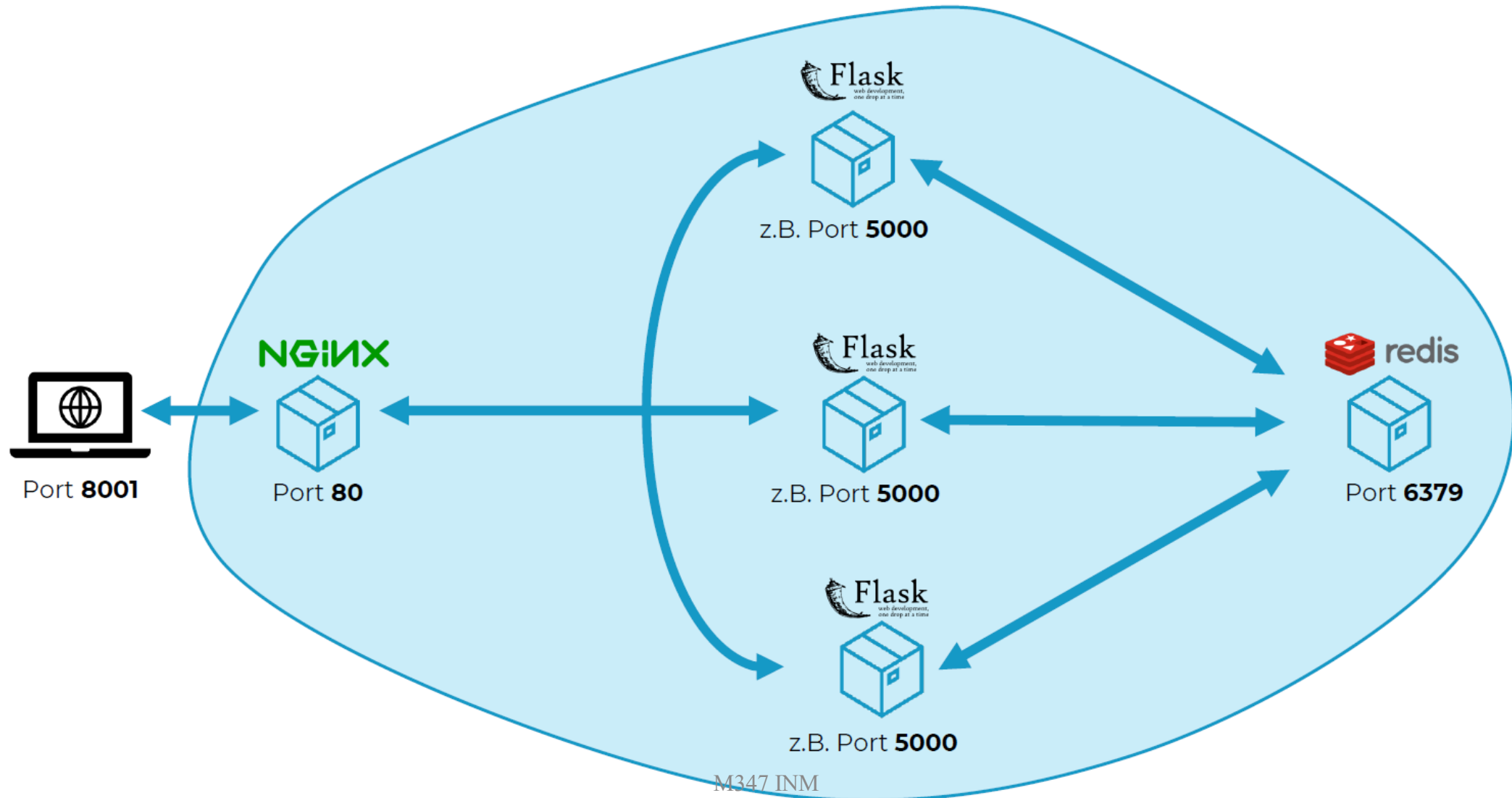
Umgebungsvariablen in .env auslagern

- Um die Konfigurationsdatei noch cleaner zu gestalten, können Sie Umgebungsvariablen in eine .env – Datei auslagern
 - Wegen dem Punkt am Anfang des Dateinamens ist die Datei versteckt
 - In jeder Zeile von der .env steht **variable=wert**
 - Dann können Sie in der *compose.yaml* schreiben:
 - **umgebungsvariable: \${variable}**
 - Standardmässig sucht Docker nach der **.env** im root von Ihrem Projektverzeichnis
 - [Übung](#)
 - [Projekt Wordpress](#)

Aufgabe 1: (Flask-Redis-App)

- Ziel von diesem Projekt ist es eine Webanwendung bestehend aus Frontend (Flask) und Backend (Redis) sowie einem Proxy-Server(Nginx) mithilfe von **docker compose** auszuführen.
 - Die prinzipielle Vorgehensweise können Sie auf eine Vielzahl von ähnlichen Anwendungsfällen übertragen!
- Die Flask-Anwendung ist mit einer Redis-Datenbank verbunden, in der die Seitenaufrufe gezählt werden.
- Die Flask-Anwendung enthält Unterseiten (/visits und /visits/reset), wofür Nginx bereits entsprechend konfiguriert worden ist.

Aufgabe 1: (Flask-Redis-App)



Aufgabe 1: (Flask-Redis-App)

Anleitung:

- Schreiben Sie die `compose.yaml`, in der Sie drei Services definieren: Frontend, Backend und Proxy-Server
 - Für Ihren Backend-Service verwenden Sie das Standard-Redis-Image, bei den anderen beiden Services nutzen Sie Ihre anhand der beiden Dockerfiles gebauten Images
 - Damit sich alle Container richtig verbinden können und Ihre Anwendung auch wirklich läuft, ist es entscheidend Ihren Frontend- und deinen Backend-Service richtig zu benennen:
 - Den Namen vom Frontend-Service können Sie aus der `nginx-proxy/nginx.conf` erschliessen
 - Den Namen vom Backend-Service können Sie aus der `flask-app/app.py` erschliessen

Aufgabe 2: PostgreSQL und pgAdmin

- Informationen:
 - PostgreSQL berücksichtigt Shell-Scripts (*.sh) oder SQL-Dateien (*.sql oder *.sql.gz) im Verzeichnis /docker-entrypoint-initdb.d, die beim Start des Containers ausgeführt beziehungsweise in die Datenbank importiert werden.
 - In Verbindung mit compose können Sie so Datenbankbenutzer oder Stored Procedures anlegen, ohne ein eigenes Docker-Image bauen zu müssen.
- [Übungsmaterial](#)
- Hinweise zur db:
 - In unserem Beispiel verwenden wir *.sql.gz, siehe im Verzeichnis init.
 - Die Umgebungsvariable POSTGRES_DB mit dem Wert uebungsdatenbank wird benötigt, damit die Datenbank erstellt wird.
 - Die Umgebungsvariable POSTGRES_PASSWORD wird benötigt.
 - Importieren Sie den Datenbank-Dump im Service volumes.
 - Der Pfad zum Verzeichnis des Containers für das Volumen ist /var/lib/postgresql/data.
- Hinweise zu pgadmin
 - Der Pfad zum Verzeichnis des Containers für das Volumen ist /var/lib/pgadmin.
 - Der Containerport ist 80
 - Die Umgebungsvariable PGADMIN_DEFAULT_EMAIL wird benötigt.
 - Die Umgebungsvariable PGADMIN_DEFAULT_PASSWORD wird benötigt.

Aufgabe 2: PostgreSQL und pgAdmin

Aufgabe:

- Erstellen Sie ein compose.yaml, dass eine PostgreSQL Datenbank und pgAdmin startet, sowie die Datenbank «uebungsdatenbank» einrichtet.
- Setzen Sie alle notwendigen Umgebungsvariablen und Service.
- Starten Sie den pgAdmin
- Fügen Sie einen neuen Server hinzu (Kurzlinks)
- Auf der ersten Registerkarte (General) geben Sie den Namen m347
- Auf der Registerkarte (Connection) geben Sie folgendes ein:
 - Hostname/-adresse: db (Benutzen Sie den Namen, den Sie für den PostgreSQL-Service benutzt haben. Bei mir db)
 - Port so belassen
 - Wartungsdatenbank so belassen
 - Benutzername: postgres
 - Kerberos authentication? So belassen
 - Passwort: Ihr gewähltes Passwort
 - Passwort speichern? Ja
 - Rolle und Service so belassen
- Öffnen Sie den Tree m347, dann Datenbanken und klicken Sie auf die uebungsdatenbank
- Öffnen Sie in der uebungsdatenbank Schemas \Tabellen\kunde_intern (View/Edit Date > All Rows) Sie finden 12 voreingerichtete Datensätze

Aufgabe 3: Backup PostgreSQL (schwierig)

Eine komfortablere Möglichkeit Backups anzulegen, geht mit `compose`.

Dazu erstellen Sie zusätzlich zu der bestehenden `compose.yaml` eine weitere Konfigurationsdatei.

Wir nennen diese `compose.backup.yaml`.

In dieser Datei befindet sich nur der `backup-Service`. Dieser verwendet das aktuelle PostgreSQL-Image und zeigt mit der `depends_on`-Anweisung an, dass der Datenbank-Server auch gestartet sein muss. Als Kommando für den `backup-Service` soll nicht die PostgreSQL-Datenbank gestartet werden, sondern `pg_dump`.

Speichern Sie die Backup-Datei in `/backup/uebungsdatenbank.dump`, so dass das von Docker verwaltete Volume verwendet wird. Das `pg_dump`-Kommando liest das der Umgebungsvariablen `PGPASSWORD` das Datenbankpasswort aus, das im `environment`-Abschnitt definiert ist.

Der Trick bei diesem Setup ist, dass `compose` mit beiden Konfigurationsdateien und der `run`-Anweisung für den Backup-Service aufgerufen wird:

```
docker compose -f compose.yaml -f compose.backup.yaml run --rm backup
```

Aufgabe 4: Wiederherstellen PostgreSQL

Fügen Sie in der Tabelle kunde_intern zwei Datensätze hinzu.

```
insert into kunde_intern values (12, 'Kundenservice');
```

```
insert into kunde_intern values (13, 'Verkauf');
```

Machen Sie ein Backup! Siehe letzte Aufgabe.

Löschen Sie die ganze Tabelle.

```
DROP TABLE kunde_intern;
```

Machen Sie ein compose.restore.yaml File und stellen Sie die Tabelle damit wieder her!

```
docker compose -f compose.yaml -f compose.restore.yaml  
run --rm restore
```

Aufgabe 5: Telefon-PHP-Applikation

- Erstellen Sie ein `compose.yaml`, welches ein Image für die [Telefon-PHP-Applikation](#) erstellt danach einen Container erstellt und startet.

Kubernetes

- [Einführung Kubernetes](#)
- [Wie werden von Kubernetes Container ausgeführt und verwaltet](#)
- [Pod mit Controllern betreiben](#)

Quellen

- Docker docs: <https://docs.docker.com/>
- Kubernetes Documentation: <https://kubernetes.io/docs/home/>
- Kubernetes API: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.20/>
- Kubernetes Konzepte: <https://kubernetes.io/docs/concepts/>
- Heise online: <https://www.heise.de/>
- Docker für Dummies, Frank Geisler und Benjamin Kettner, Wiley-VCH Verlag, 2019
- Docker, Bernd Öggli und Michael Kofler, Reinwerk Verlag, 2021
- Kubernetes, Brendan Burns – Joe Beda – Kelsey Hightower, dpunkt Verlag, 2021
- Kubernetes in Action, Marko Lukša, Manning Verlag, 2017
- Skalierbare Container-Infrastrukturen, Oliver Liebel, Rheinwerk-Verlag, 2021

Anhang

- [Lösung Seite 76](#)
- [Lösung Seite 77](#)
- [Lösung Seite 88](#)
- [Lösung Seite 91](#)
- [Lösung Seite 95](#)