

Docker Bootcamp

Einführung in diesen Kurs

Vielen Dank

► Vielen Dank, dass ich dir Docker beibringen darf!

Warum du Docker lernen solltest

► Motivation:

- ▶ Oft musst du sicherstellen, dass deine Anwendung mit einer bestimmten Version (z.B. von Python) kompatibel ist
 - ▶ Du möchtest definieren, wie die Software-Umgebung für deine Anwendung eingerichtet werden soll
 - ▶ Du möchtest Tools verwenden, deren Einrichtung aufwendig ist
-
- ▶ **Zudem:** Mit Docker Swarm bzw. Kubernetes kannst du ein Cluster komfortabel administrieren

Wie ist dieser Kurs aufgebaut?

- ▶ **Einführung in Docker**

- ▶ **Erste Schritte mit Docker:**

- ▶ Grundlegendes Arbeiten mit der Docker-Kommandozeile
- ▶ Datei-Management (Volumes, Binds, ...)
- ▶ Networking (Ports freigeben)

- ▶ **Dockerfiles:**

- ▶ Beschreibe deine Container in einer Datei und erstelle eigene Images

- ▶ **Docker Compose:**

- ▶ Verwalte mehrere Container gleichzeitig (z.B. MongoDB für die Datenbank, und Node.js als Webserver)

- ▶ **Docker Swarm:**

- ▶ Cluster-Management in Docker eingebaut

- ▶ **Kubernetes "Crashkurs":**

- ▶ Umfangreicheres Cluster-Management

Docker Bootcamp

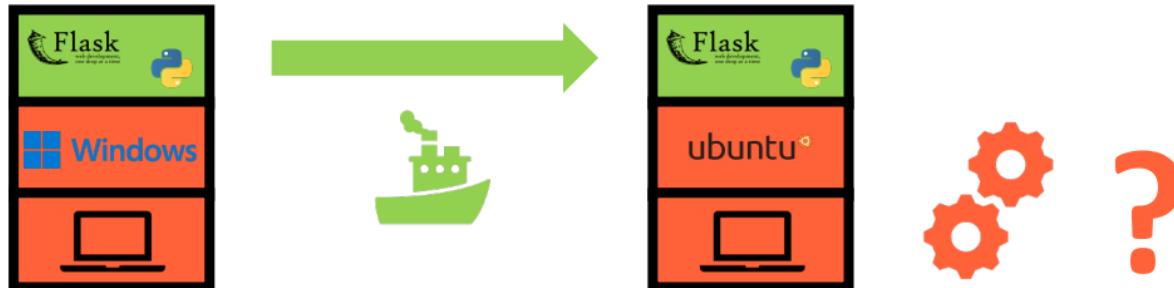
Einführung: Docker

Was lernst du in diesem Abschnitt?

- ▶ Welche typischen Probleme bei der Softwareentwicklung lassen sich mithilfe von *Docker* lösen?
- ▶ Was genau ist ein „Container“?
- ▶ Worin unterscheiden sich Container von *virtuellen Maschinen* und wie von *virtuellen Umgebungen*?
- ▶ Was ist *Docker*?
- ▶ Wie installiere ich *Docker Desktop*?
- ▶ Erste Beispiele mit dem **docker run** – Kommando
 - ▶ Einen Ubuntu-Container ausführen
 - ▶ Verschiedene Python-Container parallel ausführen

Problem: Replizierbarkeit von Anwendungen

- ▶ Beispiel: Wir wollen eine Webanwendung unter **Windows** entwickeln
 - ▶ Hier: eine Webseite auf Basis von **Flask** (ein Python-Framework)
- ▶ Später im Produktivbetrieb soll die Anwendung jedoch auf einem **Ubuntu**-Server laufen
- ▶ Wie richten wir die Entwicklungsumgebung ein, damit der Code später im Produktivbetrieb möglichst reibungslos funktioniert?



Problem: Replizierbarkeit von Anwendungen

- ▶ **Mögliche Probleme (in unserer Anwendung):**

- ▶ Verschiedene Versionen von Python
- ▶ Unterschiedliche Versionen von zusätzlichen Modulen (z.B. Flask, Numpy,...)
- ▶ Python unterschiedlich kompiliert (z.B. ohne SSL-Modul für https-Verbindungen)
- ▶ Unterschiedliche Namen für Zeitzonen

- ▶ **Mögliche Probleme (generell):**

- ▶ Unterschiedliche Versionen von externen Tools
- ▶ Unterschiedliche Behandlung von Groß- und Kleinschreibung im Dateisystem
- ▶ Unterschiedliche, maximale Länge von Dateipfaden

- ▶ **Zusätzlich:**

- ▶ Wir müssen ausführlich beschreiben, wie der Ubuntu-Server eingerichtet werden soll
- ▶ Dieses Dokument muss dann von unserem IT-Department interpretiert werden, um den korrekten Zustand vom Ubuntu-Server einrichten zu können

Docker Bootcamp

Idee 1: Virtuelle Maschine

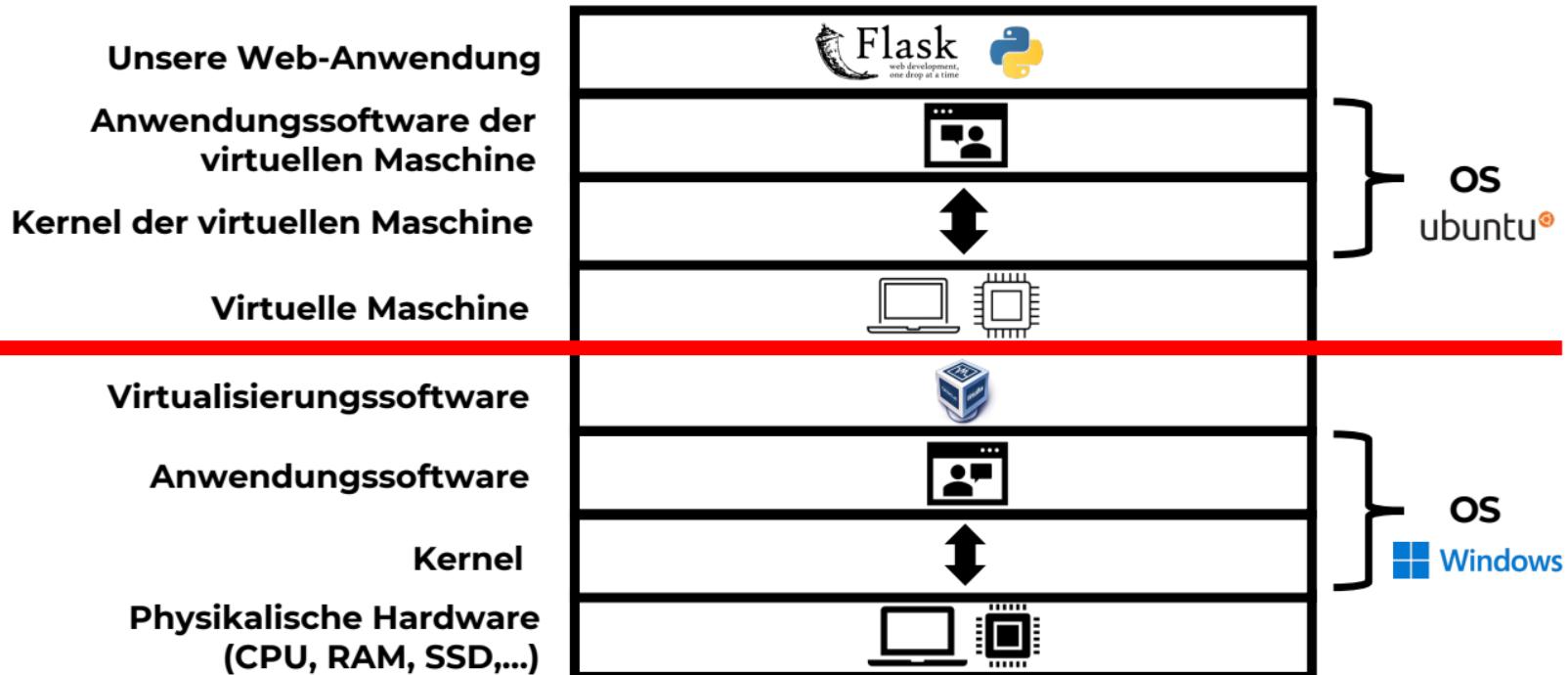
Idee 1: Virtuelle Maschine

► Virtuelle Maschine?

- ▶ Technologie, die seit den 1960ern besteht
- ▶ Virtuelle Maschinen simulieren die komplette Architektur eines Rechners und sind vom Host-System abgekapselt
- ▶ Erste Idee: wir könnten eine **virtuelle Maschine** auf dem Entwicklungssystem verwenden
 - ▶ Diese ist dann unabhängig...
 - ▶ und einfach portierbar
- ▶ Bekannte Virtualisierungssoftwares:

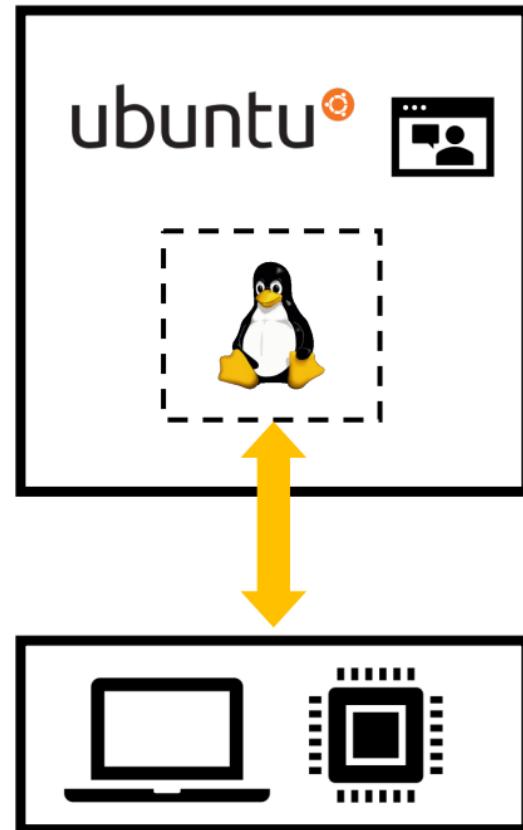


Was ist eine virtuelle Maschine?



Was ist ein Kernel?

- ▶ Kernel: "Kern eines Betriebssystems"
- ▶ Das Programm, das die unterste Schicht deiner Betriebssystem-Software darstellt
- ▶ Das erste Programm, das nach dem Systemstart in den Arbeitsspeicher geladen wird
- ▶ Fungiert als Schnittstelle zwischen der Hardware und der Anwendungssoftware
- ▶ Verwaltet die CPU-Ressourcen für die Prozesse
- ▶ Steuert die Zugriffe auf den Speicher und die Geräte (Grafikkarte, Tastatur,...)



Was sind die Nachteile von VMs?

- ▶ Nachteil von virtuellen Maschinen: es wird eine komplette Rechnerarchitektur virtualisiert mit Anwendungssoftware und Kernel
 - ▶ Tendenziell langsamer als ohne VM
 - ▶ Hohe Rechenleistung (Arbeitsspeicher, CPU)
 - ▶ Größerer Speicherplatz-Verbrauch
- ▶ Oft brauchen wir aber gar keine komplette Virtuelle Maschine
 - ▶ **Idee für mehr Effizienz:**
 - ▶ Wir virtualisieren nur die Anwendungssoftware, und nicht den Kernel
 - ▶ Dann würde auch der Overhead wegfallen
 - ▶ Technische Details zu der Art der Virtualisierung am Ende dieses Abschnitts
 - ▶ Nächster Schritt:
 - ▶ **Container**

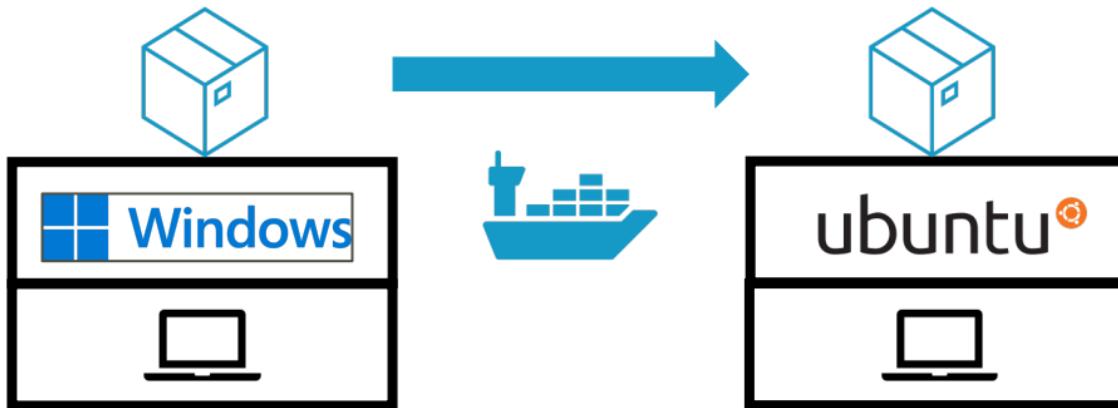
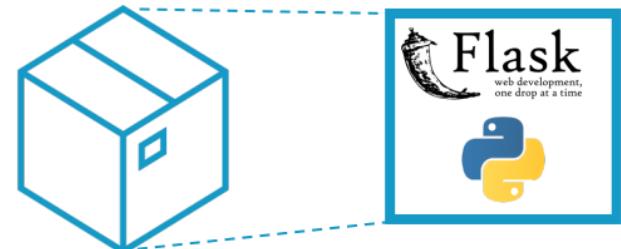


Docker Bootcamp

Von der Virtuellen Maschine zu Containern mit Docker

Was ist ein Container?

- ▶ Idee:
 - ▶ Eine separate „Umgebung“ auf deinem Rechner, die von deinem restlichen System isoliert ist
 - ▶ Enthält als Paket den gesamten Code und alle notwendigen *Dependencies*, um eine Anwendung auszuführen
 - ▶ Kann auf anderen Betriebssystemen ausgeführt werden
 - ▶ Läuft getrennt von anderen Containern



Docker Bootcamp

Installation von Docker

Was ist Docker?

- ▶ Die Firma **Docker, Inc** hat einen Industrie-Standard für solche Container etabliert
- ▶ **Docker** ist die Standard-Container-Software
- ▶ Docker selbst ist freie Software, und als Open-Source verfügbar
- ▶ Docker Desktop (die Benutzeroberfläche und ggf. der Zugang zu Dockerhub) ist aber nicht frei, ggf. sind Lizenzgebühren fällig (z.B. für größere Unternehmen)
- ▶ Wir werden jetzt Docker Desktop bei dir installieren



Docker Bootcamp

Crashkurs: Linux-Shell

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
- ▶ **touch** eine neue Datei anlegen
- ▶ **clear** Konsole leeren
- ▶ **exit** (oder **Strg+D**) Shell beenden
- ▶ **Unter Ubuntu:**
 - ▶ apt-get update: Paketlisten laden / aktualisieren
 - ▶ apt-get install -y neofetch: Das Programm "neofetch" installieren

Erstes Beispiel für einen Container: Ubuntu

- ▶ Wir beschäftigen uns jetzt kurz mit Shell-Grundlagen, die im weiteren Kursverlauf noch nützlich sein werden
- ▶ Informationen über das System per *neofetch* anzeigen:
 - ▶ **apt update**
 - ▶ **apt install neofetch**
 - ▶ **neofetch**

Erstes Beispiel für einen Container: Ubuntu

- ▶ Editor nachinstallieren
 - ▶ **apt install nano**
 - ▶ Oder: **apt install -y nano**
- ▶ Textdatei mit Inhalt anlegen:
 - ▶ **nano Beispiel.txt**
 - ▶ Reinschreiben & abspeichern: *Ich lerne den Umgang mit der Shell*

Docker Bootcamp

Die Architektur von Docker

Die Architektur von Docker

Unsere Web-Anwendung



Anwendungssoftware der
virtuellen Maschine



Kernel



Physikalische Hardware
(CPU, RAM, SSD,...)



Die Architektur von Docker

- ▶ Docker braucht einen Linux-Kernel, um Linux-Container auszuführen zu können*
- ▶ unter macOS & Windows wird im Hintergrund eine virtuelle Maschine mit einer sehr kleinen und optimierten Linux-Distribution ausgeführt
 - ▶ Overhead bei der Entwicklung vernachlässigbar wegen Deployment auf Linux (in der Regel)
 - ▶ Unter Windows ist es dank der Virtualisierungssoftware WSL 2 (Windows Subsystem for Linux) möglich, Windows- und Linux-Container auf dem gleichen Rechner auszuführen
- ▶ Unter Linux kann Docker den Kernel nutzen und somit im Vergleich zu virtuellen Maschinen Redundanzen sparen
- ▶ *Anmerkung, Randfall: Windows-Container können nur unter einem Windows-Host ausgeführt werden (Serverlizenz nötig)

Konzeption des Kurses

- ▶ Kommandozeile (CLI)
- ▶ Docker Desktop
- ▶ Dockerfiles
- ▶ Docker Compose
- ▶ Docker Swarm
- ▶ Kubernetes

Docker Bootcamp

Übungen: Erste Schritte mit Docker

Aufgabe 1: JavaScript-Befehle

- ▶ Starte in Docker einen **Node.js**-Container
 - ▶ Node.js ist eine Laufzeitumgebung für JavaScript, um JavaScript außerhalb eines Webbrowsers ausführen zu können
 - ▶ Der Docker-Befehl ist sehr ähnlich zu dem Beispiel mit dem Python-Container: du brauchst in dem Docker-Befehl nur Python durch **node** auszutauschen, beachte dabei aber die Flag!
 - ▶ Mit **Strg + C** oder **.exit** kannst du den Prozess im Terminal auch wieder beenden

Aufgabe 1:

- ▶ Nachdem du den Node.js-Container gestartet hast, überprüfe die Ausgaben von folgenden JavaScript-Befehlen:
 - a) **3 > 2 > 1;**
 - b) **typeof NaN;**
 - c) **[1, 2, 3] + [4, 5, 6];**

Aufgabe 2:

- ▶ Starte in Docker einen (neuen) Ubuntu-Container
- a) Lerne das Ubuntu-Programm **tree** kennen
 - ▶ Aktualisiere den Paketmanager apt und installiere dann das Programm **tree**
 - ▶ Ganz analog wie du den Texteditor nano installiert hast
 - ▶ Wechsel anschließend in das Stammverzeichnis:
 - ▶ **cd /**
 - ▶ Führe dann den Befehl **tree** aus – was passiert, welchen Zweck erfüllt das Programm **tree**?

Aufgabe 2:

- ▶ Im gleichen Ubuntu-Container:
 - b) Installiere auch das Programm **cmatrix**
 - ▶ Hier kannst du die Flag **-y** setzen, um eine manuelle Bestätigung zu überspringen
 - ▶ Während der Installation wirst du mehrmals aufgefordert Zahlen zur Konfiguration einzutragen: es ist spielt inhaltlich keine Rolle, welche der angegebenen Optionen du dich entscheidest
 - ▶ Was passiert dann, wenn du das Programm **cmatrix** ausführst?
 - ▶ Mit **Strg + C** kannst du das Programm auch wieder beenden

Lösung zu Aufgabe 2b):

- b) Installiere auch das Programm **cmatrix**

- ▶ **apt-get install cmatrix -y**
- ▶ Was passiert dann, wenn du dann das Programm **cmatrix** ausführst?
- ▶ "*Die Matrix ist allgegenwärtig, sie umgibt uns, selbst hier ist sie, in diesem Zimmer.*"

Docker Bootcamp

Docker Grundlagen (Containers & Images)

Lernziele für diesen Abschnitt

- ▶ Leitfrage: Was passiert eigentlich genau, wenn du den **docker run** – Befehl ausführst?
- ▶ Aus welchen Komponenten besteht Docker?
 - ▶ Das *Client-Server-Modell*
- ▶ Was ist ein *Image*?
- ▶ Wie kannst du Images von *Docker Hub* herunterladen?
 - ▶ Der **docker pull** – Befehl
- ▶ Wie startest du einen Container (erneut)?
 - ▶ Der **docker start** – Befehl
- ▶ Wie benennst du einen Container um?
- ▶ Wie kannst du einen Container debuggen?
 - ▶ Wie rufst du Metadaten zu einem Container ab?

Docker Bootcamp

Client-Server-Architektur

Die Client-Server-Architektur von Docker

► Docker Host

- ▶ Führt den *Docker Daemon* aus
- ▶ Wenn du die Anwendung Docker Desktop startest, startest du damit automatisch den *Docker Daemon*
- ▶ Daemon: ein Prozess, der im Hintergrund arbeitet und Dienste bereitstellt
- ▶ Für das Bauen, Starten und Managen von Containern zuständig
- ▶ Du kannst dir mit dem Kommando **docker info** einige Daten zu diesem Prozess anzeigen lassen

Die Client-Server-Architektur von Docker

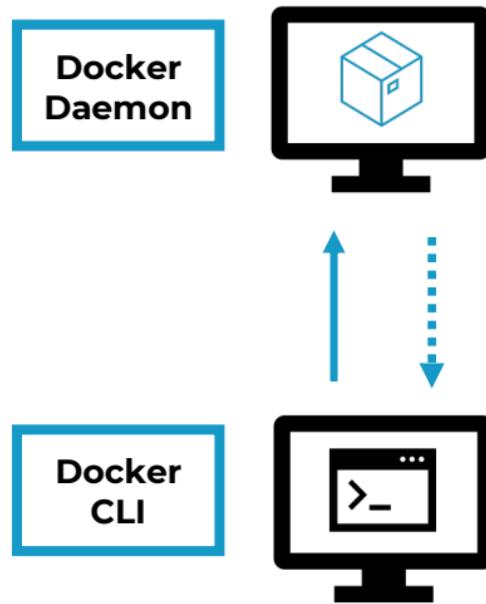
► Docker Client

- ▶ Kommuniziert mit dem Docker Deamon über die Docker API
- ▶ Wir können darüber Docker über die Kommandozeile ansteuern
- ▶ Bei Docker Host und Docker Client kann es sich um dasselbe System handeln – so wie in unserem Fall

► Warum ist Docker so aufgeteilt?

- ▶ Der eigentliche Clou von Docker liegt darin, Aufgaben auf mehrere Hosts aufteilen zu können
- ▶ Später dazu mehr (*docker swarm, Kubernetes*)

Die Client-Server-Architektur von Docker



Docker Bootcamp

Images in Docker

Was ist ein Image?

- ▶ Container werden auf der Basis von **Images** erstellt
- ▶ Du kannst dir ein Image als Bauplan 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“)

Wie kommst du an Images?

- ▶ **Container Registries** sind Server, die Images bereitstellen
 - ▶ Beispiel **Docker Hub**: Plattform mit einer Vielzahl öffentlicher Images
 - ▶ <https://hub.docker.com/>
 - ▶ Plattform wird von Docker, Inc. betrieben
 - ▶ Images sind kostenlos und werden von der Community bereitgestellt
 - ▶ **Auf Official Image achten!**
 - ▶ **Auch Anzahl der Downloads berücksichtigen**
 - ▶ Du brauchst dich NICHT anmelden, solange du nicht selbst Images veröffentlichen möchtest!
 - ▶ Jeder angemeldete Nutzer darf eigene Images auf Docker Hub veröffentlichen



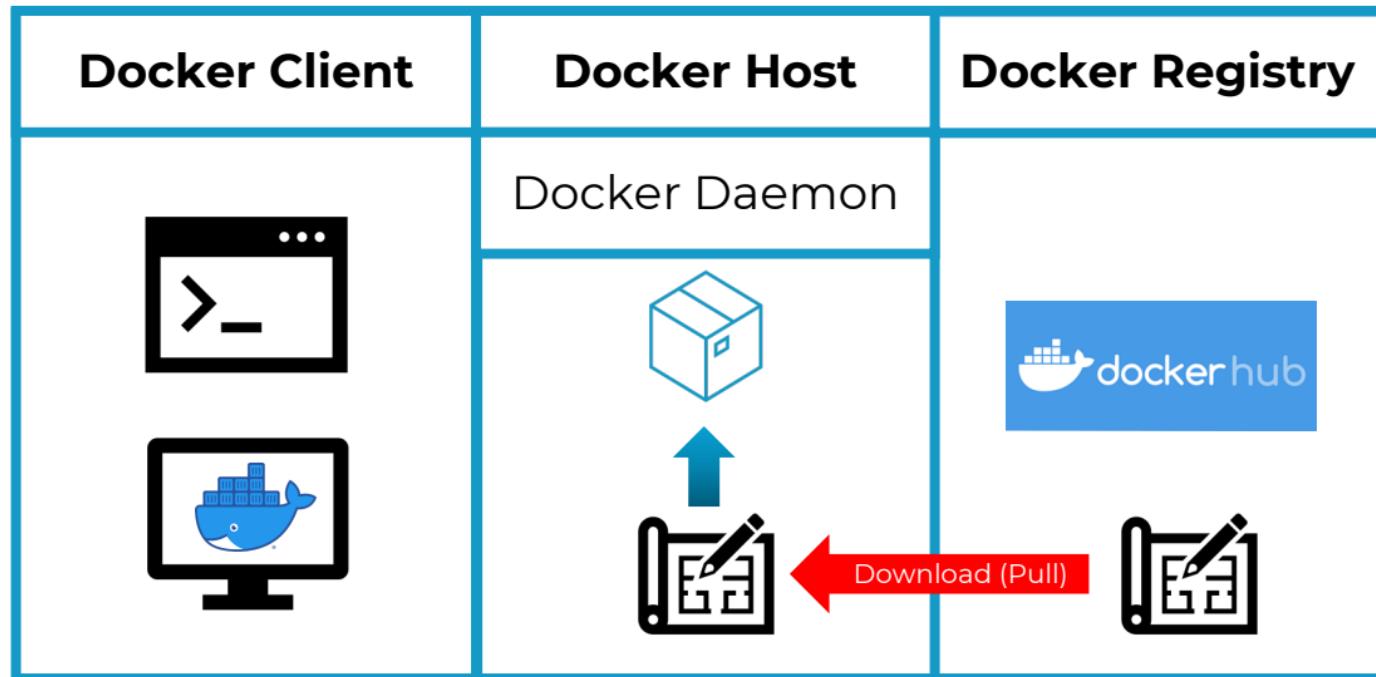
Docker Bootcamp

Images herunterladen: Docker pull

Das Kommando `docker pull`

- ▶ Mit dem Pull-Befehl können wir ein Image herunterladen:
 - ▶ **`docker pull [Quelle]/[Name vom Image]`**
 - ▶ Die Quelle brauchst du nur spezifizieren, wenn du eine andere Container Registry als Docker verwenden möchtest
 - ▶ Sonst wird standardmäßig auf Docker Hub gesucht und du kannst die Quelle somit auch weglassen
- ▶ Wir müssen das Image auf <https://hub.docker.com/> finden
 - ▶ Auf der Übersichtsseite ist der benötigte Pull-Befehl angegeben, hier:
 - ▶ **`docker pull ubuntu`**
 - ▶ Wir sehen: ist bereits heruntergeladen worden
 - ▶ **`docker pull alpine`**
 - ▶ Alpine ist eine besonders kleine Linux-Distribution, die häufig als Basis für Docker-Container verwendet wird

Das Kommando `docker pull`



Docker-Kommandos via CLI

- ▶ Wenn der Docker Daemon läuft, kannst du deine Eingabeaufforderung (Windows) bzw dein Terminal (macOS) öffnen, um Docker per CLI zu nutzen
- ▶ In der Regel besteht ein Docker-Kommando aus 4 Bestandteilen
 - ▶ Das Schlüsselwort **docker**
 - ▶ Auf welche Art von Objekt du dich beziehst: z.B. **container**, **image**,...
 - ▶ Kann auch manchmal weggelassen werden
 - ▶ Was du tun möchtest: z.B. **pull** (ein Image herunterladen)
 - ▶ Parameter für den Befehl: z.B. welches Image du herunterladen möchtest (**ubuntu**, **python**)
 - ▶ Zusätzlich können oder müssen manchmal noch Flags gesetzt werden

Docker Bootcamp

Aufbau eines Images

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 (prinzipiell wie ein diff)
- ▶ 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 kannst du dir 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 ubuntu`
 - ▶ `docker image history --no-trunc ubuntu`
- ▶ Detaillierte Infos zu einem Image abfragen (z.B. „Created“)
 - ▶ **`docker image inspect [Name vom Image]`**
 - ▶ `docker image inspect ubuntu`
- ▶ Du kannst auch gezielt eine Eigenschaft abfragen
 - ▶ `docker image inspect ubuntu --format="{{.Created}}"`
 - ▶ Die Flag `-f` (oder auch `--format`) steht für Formatierungsoptionen einer Ausgabe
 - ▶ Im Wert wird *Go Template Syntax* verwendet, um eine Eigenschaft gezielt anzusteuern

Docker Bootcamp

Tags von Images

Verschiedene Tags ansteuern

- ▶ Tags erlauben es dir verschiedene Versionen von Images anzusteuern
 - ▶ Doppelpunktschreibweise **image:tag**
 - ▶ Das kennen wir schon vom Python-Beispiel!
 - ▶ Wenn du keinen tag spezifizierst, wird standardmäßig das Tag *latest* für die aktuellste Version gesetzt

Docker Bootcamp

Images taggen

Wie kannst du ein Image taggen?

- ▶ Ein Image kann NICHT umbenannt werden, aber du kannst es mit einem **tag** versehen, um das Image dann darüber anzusteuern
 - ▶ **docker image tag [Name vom Image] [neuer tag für Image]**
 - ▶ **docker image tag ubuntu ubuntu-new**
 - ▶ Die tags tauchen dann als Metadaten auch bei **docker image inspect** für das Image auf:
 - ▶ **docker image inspect -f "{{ .RepoTags }}" ubuntu**

Docker Bootcamp

Images löschen

Wie kannst du Images löschen?

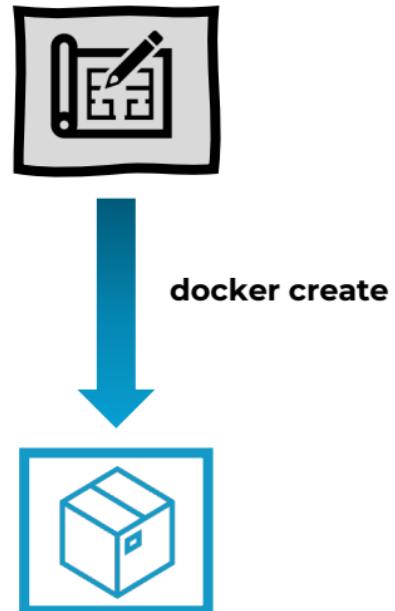
- ▶ Einen tag bzw. ein Image löschen
 - ▶ **docker image rm [Name vom Image bzw tag]**
 - ▶ **docker image rm ubuntu-new**
 - ▶ dabei wird erstmal nur der als Parameter verwendete tag gelöscht
 - ▶ deswegen erhält man als Feedback eine *untagged*-Info!
 - ▶ Du kannst ein Image nur löschen, wenn bei dir kein Container mehr existiert, der darauf basiert
 - ▶ Mehrere Images löschen
 - ▶ **docker image rm [Name vom Image bzw tag] [Name vom Image bzw tag] ...**
 - ▶ **docker image rm \$(docker images -a -q)**

Docker Bootcamp

Docker: Container erstellen & manuell starten

Das Kommando `docker create`

- ▶ Wie kommt man nun von einem Image zu einem Container?
 - ▶ **docker [container] create**
 - ▶ **docker create nginx**
 - ▶ **docker create -it ubuntu**
 - ▶ Wenn du den Container im Terminal ausführen möchtest, 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!
 - ▶ Speicherplatzoptimierung: Ein Container enthält nur die geänderten Daten im Bezug zum Image, aus dem er erstellt wurde



Das Kommando `docker start`

- ▶ Wie startest du nun einen Container?
 - ▶ `docker [container] start`
 - ▶ Sofern du den Container im Terminal ausführen möchtest, musst du 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 Bootcamp

Abgrenzung: docker run vs. docker create und start

Das Kommando `docker run`

`docker run`

`docker [image] pull`

Ein Image herunterladen (in der Regel von Docker Hub)

`docker [container] create`

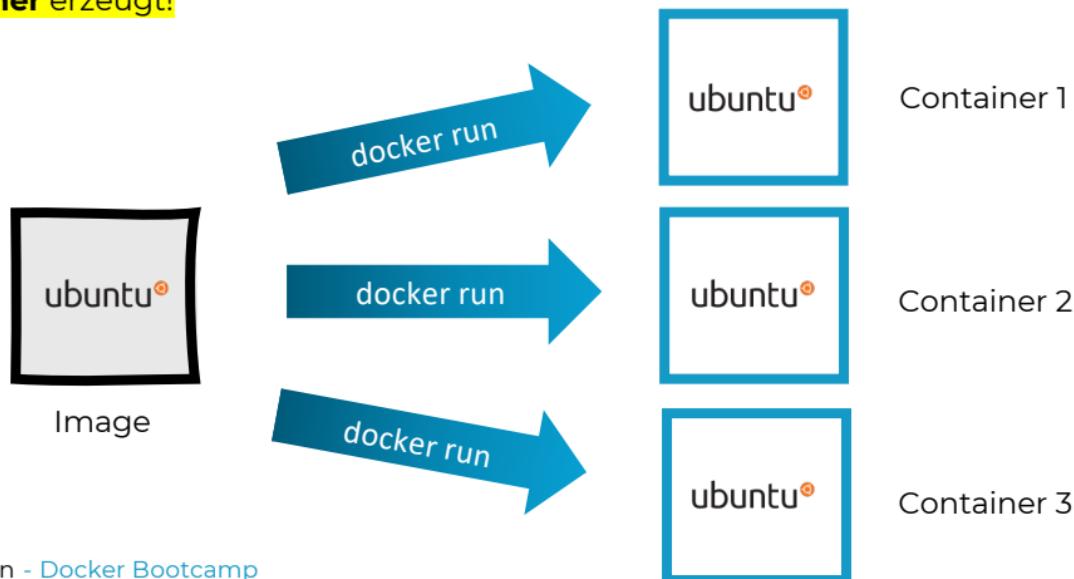
Einen Container auf Grundlage vom heruntergeladenen Image erstellen

`docker [container] start`

Den neu gebauten Container ausführen

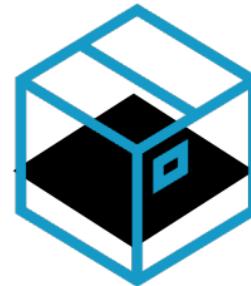
Das Kommando `docker run`

- ▶ Beachte:
 - ▶ `docker run` bezieht sich auf *Images*
 - ▶ `docker start` bezieht sich auf *Container*
- ▶ Bei der Ausführung von `docker run` wird immer ein **neuer Container** erzeugt!



Einen Container erneut starten

- ▶ Nun können wir einen Container mit dem Kommando **docker start [Name des Containers]** deutlich komfortabler ansteuern und neu starten
 - ▶ **docker start [Name/ID musst du bei dir nachschauen]**
- ▶ Dateien bleiben im Container erhalten
 - ▶ Jeder Container besitzt einen eigenen Speicher
 - ▶ Daten, die in einem Container erzeugt werden, werden in einen speziellen Container-Layer geschrieben: **writable layer**
 - ▶ Details zum Dateimanagement später
- ▶ **Blackbox:** Auf diesen *writable layer* eines Containers können wir von außen nicht zugreifen



Docker Bootcamp

Infos zu Containern abrufen

Infos zu Containern abrufen

- ▶ Auch bei **docker container** können wir uns Informationen zu allen Containern anzeigen lassen
- ▶ **docker container ls -a -s**
 - ▶ Mit der Flag **-a** (oder **--all**) werden alle Container aufgelistet (sonst nur die gerade aktiven!)
 - ▶ Mit der Flag **-s** (oder **--size**) wird zu jedem Container noch die Größe angegeben
 - ▶ alternativ (aber älteres Kommando): **docker ps** (Flags beachten)
 - ▶ Es fällt auf, dass jeder Container einen zufallsgenerierten Namen trägt!

Infos zu Containern abrufen (Teil 2)

- ▶ **docker container stats**: Zeige dir z.B. Arbeitsspeichernutzung von den Containern an
- ▶ **docker [container] inspect**
 - ▶ Du kannst einen Container über seine ID oder seinen Namen ansteuern

Docker Bootcamp

Container benennen

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-test**
 - ▶ Eigenen Namen beim Erzeugen festlegen mit der Flag **--name**
 - ▶ **docker create --name ubuntu-test -it ubuntu**
 - ▶ **docker run --name ubuntu-test -it ubuntu**
 - ▶ Namen von Containern müssen eindeutig sein!

Docker Bootcamp

Container stoppen / erneut starten / ...

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

- ▶ Wenn du einen Container im Terminal ausführst, reicht es aus, das darin ausgeführte Programm zu beenden, um auch den Container zu stoppen
 - ▶ Z.B. Container mit Ubuntu, Python
- ▶ Ansonsten kannst du einen Container mit dem Befehl **`docker stop [ID / Name]`** beenden
- ▶ Weiter gibt es noch die Befehle **`docker pause [ID / Name]`** und **`docker unpause [ID / Name]`**, um einen Container vorübergehend anzuhalten bzw. seine Ausführung fortzusetzen

Docker Bootcamp

Container löschen

Wie löscht du Container?

- ▶ Mit dem Kommando **docker [container] rm [Name/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)**

Wie löscht du Container?

- ▶ **Tipp:** Bei docker run
 - ▶ Mit der Flag **--rm** kannst du spezifizieren, dass ein Container direkt nach seinem Beenden automatisch wieder gelöscht wird
 - ▶ **docker run -it --rm ubuntu**
 - ▶ Praktisch, wenn du kurz mal was ausprobieren willst
 - ▶ Wenn ein Container gelöscht wird, gehen auch alle Dateien aus seinem internen Speicher (writable layer) verloren!

Docker Bootcamp

Container: Programme starten

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 Python ist es eine Python-Shell
- ▶ Hierbei ist die Schreibweise wie folgt:
 - ▶ **docker container run [OPTIONS] IMAGE [COMMAND]**
 - ▶ **docker container create [OPTIONS] IMAGE [COMMAND]**
- ▶ Wir können hier das Kommando austauschen!

Container: Zusätzliche Programme starten

- ▶ Mit Hilfe von folgendem Befehl kannst du mehrere Programme innerhalb des gleichen Containers starten:
 - ▶ **docker container exec [OPTIONS] CONTAINER COMMAND**
- ▶ Dies kann praktisch sein, wenn du z.B. in einem Python-Container bist, aber z.B.:
 - ▶ Du möchtest eine zusätzliche Datei im Container platzieren
 - ▶ Du möchtest Programme nachinstallieren

Docker Bootcamp

Aufgaben: Grundlagen von Docker

Aufgabe 1: (Images)

- ▶ Finde 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?
- ▶ Lade das aktuelle **node:alpine**-Image herunter
 - ▶ Wie groß ist es im Vergleich zum Standard-Node.js-Image?
- ▶ Benenne das heruntergeladene Image in *small-node* um
- ▶ Überzeuge dich, dass bei dir jetzt ein Image namens *small-node* existiert
- ▶ Erzeuge und starte einen Node.js-Container basierend auf dem *small-node*-Image, welcher automatisch wieder gelöscht werden soll. Starte dann per docker exec in diesem Container eine Bash
 - ▶ Welche Fehlermeldung tritt dabei auf?
- ▶ Lösche beide Node.js-Images, die auf Alpine basieren

Aufgabe 2: (Container)

- ▶ Erzeuge einen neuen Node.js-Container namens `test-container` auf Grundlage des Standard-Node.js-Images und führe darin die `bash` aus
 - ▶ Dieser Container soll nach Beenden NICHT automatisch gelöscht werden
- ▶ Steuere mit der `bash` im Container den Ordner `etc` an und finde die Version der zugrunde liegenden Debian-Distribution heraus (steht in der Datei: "debian_version").
 - ▶ Tipp: mit dem **cat**-Befehl kannst du Dateien auslesen
- ▶ Gehe dann zurück in den root-Ordner und von dort zu `/usr/share`
 - ▶ Findest du in dem Ordner Hinweise zu einer anderen Programmiersprache, die wir schon verwendet haben und die in diesem Container installiert ist?

Aufgabe 2: (Container)

- ▶ Benenne dann den Container *test-container* um in *my-node-app*
- ▶ Erstelle dann dort dann den Ordner /app (Befehl: mkdir /app)
- ▶ Wechsel anschließend in diesen Ordner und erstelle eine Datei "main.js" mit folgendem Inhalt:
 - ▶ console.log("Hallo Welt")
- ▶ Führe anschließend dein Skript via **node main.js** aus

Docker Bootcamp

Docker Grundlagen, Teil 2 (inkl. Networking)

Was du in diesem Abschnitt lernen wirst

- ▶ Leitfrage: *Wie kannst du einen Webserver in einem Container starten und dann im Browser nutzen?*
- ▶ Bisher haben wir Container immer per Terminal gesteuert
- ▶ Wenn wir jetzt aber einen Webserver betreiben möchten...
 - ▶ Wie lassen wir diesen im Hintergrund laufen (sodass er immer aktiv ist)?
 - ▶ Wie geben wir einen Port frei, damit wir von außen auf unseren Container zugreifen können?
- ▶ Beispiele in diesem Abschnitt:
 - ▶ Wir betreiben einen Webserver (nginx) in Docker
 - ▶ Wir betreiben einen Webserver (apache mit php) in Docker
 - ▶ Wir starten eine PySpark-Umgebung für Data Science

Docker Bootcamp

Container im Hintergrund laufen lassen

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, kannst du mit **docker logs** die Ausgabe einsehen

Docker Bootcamp

Container automatisch starten

Container automatisch starten

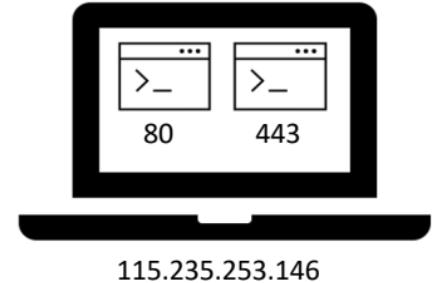
- ▶ Standardmäßig wird ein Container nicht erneut gestartet
- ▶ Wir können aber die **restart policy** auf folgende Werte setzen:
 - ▶ **on-failure[:max-retries]**: Startet den Container neu, wenn es einen Error gibt
 - ▶ **always**: Der Container wird immer neu gestartet (es sei denn, wir stoppen ihn, 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 unless-stopped [Image-Name]**
 - ▶ **docker update --restart unless-stopped [Container-ID / Name]**

Docker Bootcamp

Crashkurs: Ports

Was sind Ports?

- ▶ Unterschied zwischen einer IP-Adresse und einem Port:
 - ▶ **IP-Adresse:** bezieht sich auf ein Gerät in einem Netzwerk
 - ▶ IPv4, z.B. 115.235.253.146 (4x Dezimalzahlen, 0-255)
 - ▶ IPv6, z.B. 2001:0db8:85a3:0000:0000:8a2e:0370:7334 (8x Hexadezimalzahlen)
 - ▶ **Port:** bezieht sich auf einen Prozess auf einem Gerät in einem Netzwerk
 - ▶ Insgesamt gibt es 65.535 Portnummern, standartmäßig werden z.B. folgende Ports für folgende Protokolle verwendet:
 - ▶ Port 22: SSH
 - ▶ Port 80: HTTP
 - ▶ Port 443: HTTPS
 - ▶ Oft werden IP-Adresse und Port kombiniert geschrieben (z.B. 115.235.253.146:80)
 - ▶ Wenn du versuchst einen weiteren Prozess auf einen Port zu starten, auf dem bereits ein Prozess läuft, kommt es zu einer **port collision**
 - ▶ Lösung: einen anderen Port wählen
 - ▶ *Manchmal können auch Berechtigungen fehlen, um bestimmte Ports öffnen zu können*



Docker Bootcamp

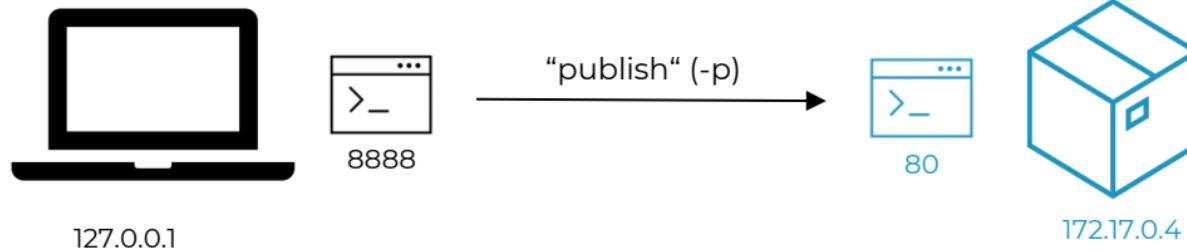
Webservice in Container starten

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
- ▶ Mit der Flag **-p** (oder **--publish**) kannst du 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 (Marktanteil:
ca. 44% unter den 10.000 meistbesuchten Webseiten)
- ▶ **docker run -p 8888:80 --rm nginx**
- ▶ Danach können wir im Browser via <http://localhost:8888> den
Webserver auf unserem Port 8888 ansteuern
- ▶ Du kannst auch andere Ports bei dir freigeben, z.B. 8080
- ▶ Der Container-Port muss allerdings 80 (HTTP) lauten, da nginx
standardmäßig auf Port 80 lauscht



Ports setzen (docker container create)

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

Docker Bootcamp

Nächstes Beispiel: Webserver mit PHP

Docker Bootcamp

Nächstes Beispiel: PySpark für Data Science

Nächstes Beispiel: PySpark

- ▶ **PySpark:** Eine Python-Schnittstelle für Apache Spark
- ▶ **Apache Spark:** Framework für die Verarbeitung von Big Data in Rechner-Clustern
- ▶ Ohne Docker: nicht so leicht einzurichten
- ▶ Wir werden dieses Image verwenden:
<https://hub.docker.com/r/jupyter/pyspark-notebook>
- ▶ Wir werden einen Notebook-Server im Browser benutzen



Code für PySpark

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('test').getOrCreate()
spark

-----
from pyspark.sql.types import StructType, StructField, StringType

schema = StructType([
    StructField('user_name', StringType()),
    StructField('language', StringType())
])

data = [('elastic_newton', 'english'), ('kind_einstein', 'german')]

df=spark.createDataFrame(data=data, schema=schema)
df.show()

-----
df.write.option('header', True).csv('datacsv')

-----
df = spark.read.option('header', True).csv("datacsv")
df.show()
```

Docker Bootcamp

Übung: Container im Browser nutzen

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

- ▶ Starte dazu einen *nginx*-Container und verbinde einen beliebigen, freien Port zu ihm
 - ▶ **Erinnerung:** Der nginx-Webserver erwartet eingehende Verbindungen auf Port 80!
- ▶ Greife auf die bash von deinem *nginx*-Container zu (während der Webserver läuft)
- ▶ Navigiere über die bash in das Verzeichnis `/usr/share/nginx/html`
- ▶ Verändere dann den Inhalt der Datei `index.html`, speichere die Änderungen ab und aktualisiere die URL in deinem Browser
 - ▶ Dazu kann es wie üblich hilfreich sein, den Editor `nano` nachzuinstallieren

Docker Bootcamp

Daten in Containern verwalten

Lernziele für diesen Abschnitt

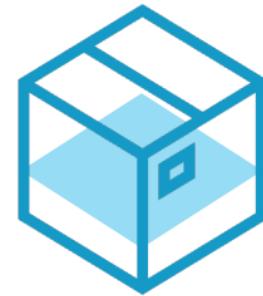
- ▶ Wie werden Daten in einem Docker-Container gespeichert?
- ▶ Wie kannst du Daten in einen Container kopieren und aus einem Container herauskopieren?
 - ▶ Das Kommando **docker cp**
- ▶ Wie kannst du Daten eines Containers persistent speichern?
 - ▶ **Mounts**
 - ▶ Welche Arten von Mounts stellt Docker bereit?
 - ▶ **Volumes**
 - ▶ **Binds**
 - ▶ **(tmpfs)**
 - ▶ Was bedeutet die Flag -v im einführenden Beispiel?
 - ▶ **docker run -it -p 3000:80 -v \$(pwd):/usr/share/nginx/html nginx**

Docker Bootcamp

Dateien im Container speichern

Wie werden Daten in Docker gespeichert? (was du bislang weißt)

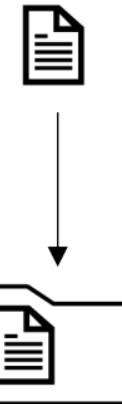
- ▶ Images sind read-only, und können nicht verändert werden
- ▶ Dateien, die in einem Docker-Container erzeugt oder verändert werden, befinden sich in einem speziellen *writable layer* des Containers
- ▶ Wenn du einen Container löscht, dann gehen auch alle seine Dateien verloren



Das Kommando `docker cp`

- ▶ Mit dem Befehl **docker cp** kannst du schnell Dateien oder Ordner in einen Container und aus einem Container kopieren
- ▶ Du musst 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

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



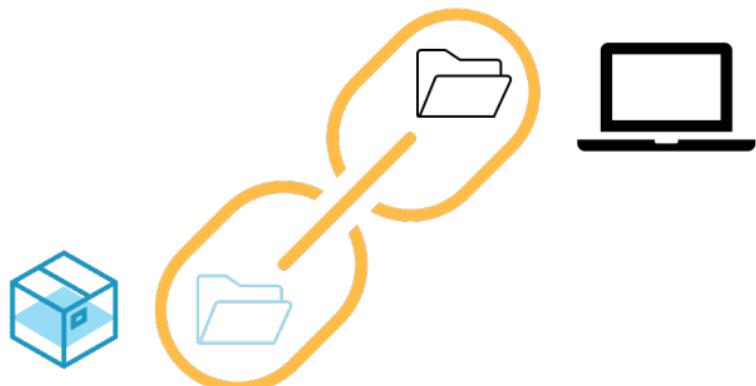
2. Parameter: Neuer Speicherort von Datei/Verzeichnis (in Container oder Host-System)

Docker Bootcamp

(Bind-)Mounts in Docker

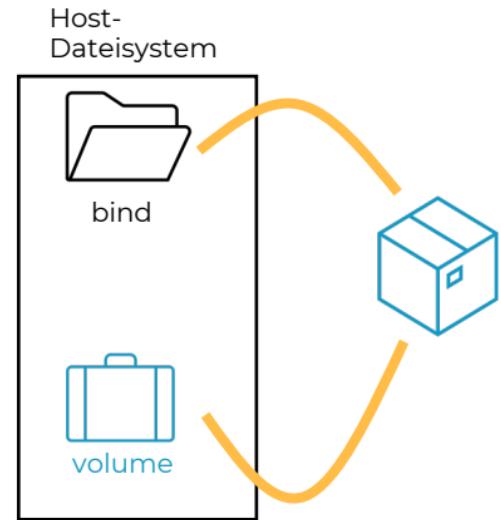
Was sind Mounts?

- ▶ Als Mounts bezeichnet man allgemein die Verknüpfung von der Verzeichnisstruktur eines Containers mit Daten, die außerhalb vom Container liegen
- ▶ Das erlaubt dir 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
 - ▶ Volumes werden (in der Regel) mit Docker verwaltet
 - ▶ **tmpfs**
 - ▶ nur unter Linux verfügbar!
 - ▶ Daten werden in den Arbeitsspeicher deines lokalen Systems geschrieben
 - ▶ Daten werden sofort gelöscht, sobald der Container anhält
 - ▶ Use Case: sensible Daten



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

Docker Bootcamp

Readonly Bind-Mounts

Readonly Bind Mounts

- ▶ Wir können einen Bind Mount auch auf Readonly schalten
- ▶ Dann darf der Container auf diesen Ordner nur lesend zugreifen
- ▶ Wenn wir dann aus dem Container heraus doch versuchen, eine Datei zu schreiben, bekommen wir einen Fehler:
 - ▶ Read-only file system
- ▶ Änderungen vom Host werden natürlich weiterhin direkt in den Container übernommen
- ▶ Wie setzen wir einen readonly bind mount?
 - ▶ **-v [absoluter Pfad zu deinem lokalen Verzeichnis]:[Pfad zum Container-Verzeichnis]:ro**
 - ▶ **--mount type=bind,source=[Pfad zu deinem lokalen Verzeichnis],destination=[Pfad zum Container-Verzeichnis],readonly**

Docker Bootcamp

Beispiel: Bind Mount, nginx

Docker Bootcamp

Beispiel: Bind Mount, Jupyter

Docker Bootcamp

Aufgabe: Copy & Bind Mount

Aufgabe: Copy & Bind Mount

- ▶ Erstelle einen Webserver mit PHP und Apache (Image: php, z.B. php:8.1-apache), und liefere darüber eine PHP-Webseite aus
- ▶ Schau 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)**
 - ▶ Kopiere 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?

Docker Bootcamp

Copy vs. Bind-Mount

Copy vs. Bind-Mount: Funktionsweise

▶ **Copy:**

- ▶ Ein docker cp kopiert die entsprechenden Dateien in den "writable layer" von unserem Container
- ▶ Dies ist performant, da Zugriffe direkt von Docker optimiert werden können

▶ **Bind-Mount:**

- ▶ Ein bind mount erzeugt einen gewissen Overhead
- ▶ Warum?
 - ▶ Änderungen an Dateien müssen direkt im Container sichtbar sein
 - ▶ Ggf. müssen auch Änderungen aus dem Container auf den Host zurückgeschrieben werden
- ▶ Dies kostet zusätzliche Performance

Copy vs. Bind-Mount: Wann nutzen?

- ▶ Manchmal müssen wir für einen Anwendungszweck z.B. ein bind mount benutzen
- ▶ Wenn wir aber die Wahl haben:
 - ▶ **Copy:**
 - ▶ Ein Copy ist ideal im Produktivbetrieb, wo es auf Performance ankommt
 - ▶ Wenn der Container einmal eingerichtet ist, dann bleibt er so, und Änderungen müssen manuell übernommen werden (oft gut!)
 - ▶ **Bind-Mount:**
 - ▶ Ein bind mount ist praktisch während der Entwicklung
 - ▶ Änderungen müssen nicht in den Container übertragen werden

Docker Bootcamp

Bonus-Beispiel: Markdown nach PDF umwandeln

Beispiel für eine Markdown-Datei

```
# Das Pandoc/LaTeX-Image

## Quelle
[Docker Hub](https://hub.docker.com/r/pandoc/latex)

## Download
Du kannst das Image mit dem Befehl
```bash
docker pull pandoc/latex
```
herunterladen.

## Inhalte
Dieses Image enthält

- den Parser *pandoc*,
- eine minimale *LaTeX*-Installation.

## Use Case
Mit *pandoc* und *LaTeX* können wir Markdown-Dateien (wie diese hier) in PDF-Dateien umwandeln.
```

Beispiel: Mit Pandoc eine Markdown-Datei in eine PDF-Datei umwandeln

- ▶ Ein weiteres praktisches Beispiel ist die Umwandlung einer Markdown-Datei in eine PDF-Datei mithilfe von **pandoc** und **LaTeX**
 - ▶ pandoc: Programm zur Konvertierung verschiedener Dokumentenformate („parsen“)
 - ▶ LaTeX: Programm zur komfortableren Verwendung des Textsatzsystems TeX, mit dem du Dokumente sehr präzise und grundlegend formatieren kannst
- ▶ Wir werden dieses Image verwenden:
 - ▶ <https://hub.docker.com/r/pandoc/latex>

Beispiel: Mit Pandoc eine Markdown-Datei in eine PDF umwandeln

- ▶ Lege auf deinem Desktop einen Ordner `pandoc-conv` an
- ▶ Speichere in diesem Ordner eine `Readme.md`-Datei mit dem Inhalt von der vorherigen Slide
- ▶ Navigiere in deinem Terminal in diesen Ordner
- ▶ Führe dann das folgende Kommando aus
 - ▶ **`docker run --rm -v $(pwd):/data pandoc/latex README.md -o outfile.pdf`**
 - ▶ Die Flag `-o` gehört nicht zu Docker, sondern zu pandoc, und ist notwendig, um eine Ausgabe (eine pdf-Datei) zu generieren
 - ▶ Natürlich darauf achten, wo sich gerade dein Arbeitsverzeichnis befindet (hier: der Ordner, in dem auch die md-Datei liegt)
 - ▶ Unter ARM wird eine Warnung angezeigt, aber die Funktionsweise sollte dadurch nicht beeinträchtigt sein
 - ▶ Die Konvertierung kann schon einige Sekunden dauern...

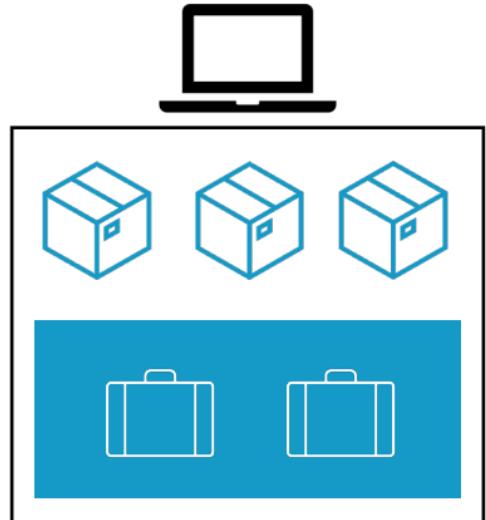
Docker Bootcamp

Volumes in Docker

Was ist ein Volume?

► Was sind Volumes?

- ▶ 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 kannst (Windows / macOS) oder solltest (Linux) du nicht direkt zugreifen, sondern es ausschließlich über docker volume ansteuern



Wie erzeugst und verwaltest du ein Volume?

- ▶ Mit **docker volume create [Name]** kannst du ein neues Volume erzeugen
 - ▶ **docker volume create ubu-vol**
- ▶ Alle Volumes anzeigen: **docker volume ls**
- ▶ Mit **docker volume inspect [Name]** kannst du ein einzelnes Volume untersuchen
- ▶ Du kannst auch wie sonst ein Volume mit **docker volume rm [Name]** löschen
- ▶ Weiter kannst du mit **docker volume prune** alle Volumes löschen, die nicht mit mindestens einem Container verbunden sind
- ▶ Es ist nicht möglich, ein Volume umzubennnen
 - ▶ Du müsstest ein neues Volume erstellen, und dort alle Daten hinüberkopieren

Docker Bootcamp

Volumes mit Container verknüpfen

Volumes mit Container verknüpfen (`--mount`)

- ▶ So wie bei bind mounts kannst du 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 du mit dem Volume verbinden möchtest
- ▶ Insgesamt:
 - ▶ **--mount type=volume,source=[dein 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!
 - ▶ Du brauchst 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 du bei der Flag **-v** sehr gut aufpassen musst, dass du nicht ein neues Volume erzeugst, wenn du stattdessen eigentlich einen *bind mount* durchführen willst!
- ▶ Du kannst natürlich auch einen Container mit mehreren Volumes verbinden (einfach **-v** mehrfach angeben)

Docker Bootcamp

Automatisch generierte Volumes

Automatisch generierte Volumes



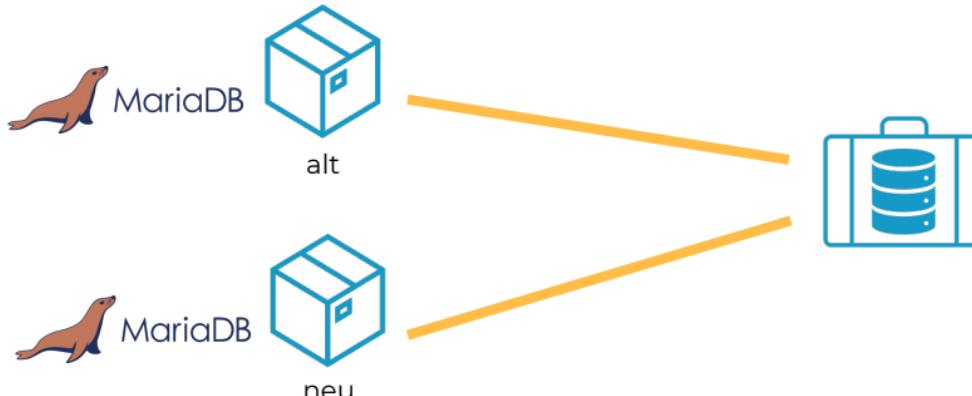
- ▶ Bei manchen Containern wird auch ohne explizite Spezifikation ein Volume angelegt:
- ▶ Zum Beispiel beim Datenbank-Management-System **mariadb** bzw. **mySQL**
 - ▶ **docker run -it -e MYSQL_ROOT_PASSWORD=password123 mariadb**
 - ▶ Per Flag **-e** bzw. **--env** muss hier eine Umgebungsvariable gesetzt werden
 - ▶ **docker inspect -f "{{.Mounts}}" [Name / ID]**
 - ▶ Das Volume taucht natürlich auch bei *docker volume ls* auf
 - ▶ Zur besseren Kontrolle solltest du aber ein eigenes benanntes Volume einführen
 - ▶ **docker run -it -v mariadb-vol:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=password123 mariadb**

Docker Bootcamp

Beispiel: Datenbanksoftware aktualisieren

Datenbanksoftware ohne Datenverlust aktualisieren

- ▶ Wir wollen einen Container, in dem eine MariaDB läuft, updaten
 - ▶ Dazu werden wir die Datenbank in einem Volume sichern
- ▶ Wir updaten den Container, indem wir einen neuen Container erstellen mit einer neuen Version von MariaDB
- ▶ **Wichtig:**
 - ▶ mariadb ist i.d.R. nicht abwärtskompatibel: Wenn du stattdessen die Version herabstufst, dürfte der Container abstürzen!



Docker Bootcamp

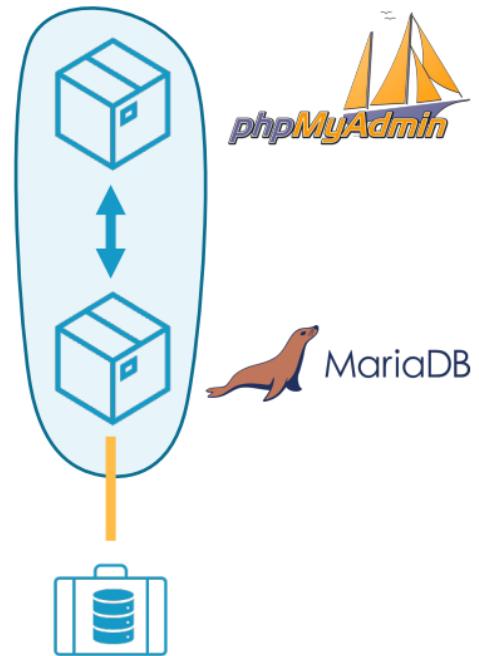
Networking in Docker

Lernziele für diesen Abschnitt

- ▶ Wie kannst du mit Docker mehrere Container miteinander verbinden?
- ▶ Wierichtest du ein Netzwerk ein, in dem mehrere Container miteinander kommunizieren können?
- ▶ Was sind *bridge*-Netzwerke und welche anderen Netzwerk-Arten gibt es noch?

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 MySQL bzw. MariaDB
 - ▶ 1 Container für phpmyadmin (Verwaltungssoftware für MySQL / MariaDB)
 - ▶ 1 Container für Wordpress
- ▶ 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

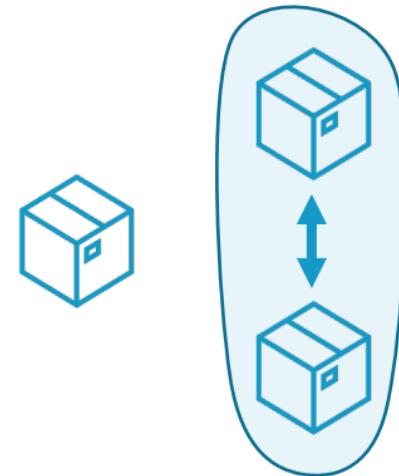


Netzwerk-Treiber in Docker

- ▶ 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)



Docker Bootcamp

Networking in Docker

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; du kannst kein weiteres erzeugen
- ▶ **Wichtig:** Funktioniert nur auf Linux-Hosts

Netzwerk-Treiber in Docker

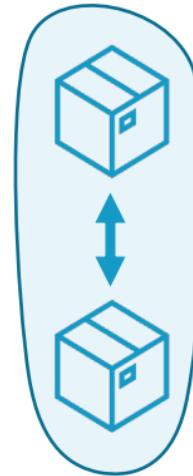
- ▶ **Null:** die Netzwerkfunktionalität des Containers wird deaktiviert
 - ▶ Es kann kein *None*-Netzwerk erzeugt werden
 - ▶ Beim Erstellen eines Containers die Flag **--network none** setzen
- ▶ Später werden wir noch weitere Netzwerk-Treiber kennen lernen
 - ▶ Z.B. **Overlay** bei *docker swarm*, wenn Container auf verschiedenen Hosts laufen

Docker Bootcamp

Eigene Netzwerke anlegen

Ein neues (bridge-)Netzwerk erstellen

- ▶ Du kannst 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** kannst du (ungenutzte) Netzwerke auch wieder löschen
 - ▶ Die drei Standard-Netzwerke (*bridge*, *host* und *none*) kannst du nicht löschen
- ▶ Warum ein eigenes Netzwerk?
 - ▶ Abschottung von anderen Containern
 - ▶ Gerade im produktiv-Betrieb sicherer und stabiler



Container nachträglich verbinden

- ▶ Du kannst 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** kannst du dann prüfen, ob der Container dem Netzwerk angehört
- ▶ Mit **docker network disconnect** kannst du einen Container auch wieder aus einem Netzwerk entfernen
 - ▶ **docker network disconnect bridge my-mariadb**

Docker Bootcamp

Beispiel: MariaDB und phpMyAdmin

Beispiel: 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

Docker Bootcamp

Aufgabe: Container in Netzwerken miteinander verbinden

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 eine 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)

- ▶ Erstelle ein bridge-Netzwerk namens *mongo-net* und starte darin einen Mongo-Container mit dem Namen *my-mongo*. Verbinde 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 du den Container erst genauer untersuchen willst: auch er verwendet ein Debian-Image (bash...)

Aufgabe: (MongoDB & MongoExpress)

- ▶ Starte nun, ebenfalls in dem Netzwerk *mongo-net*, einen Mongo Express – Container, den du zum MongoDB-Container verbindest. Denke daran, einen Port vom Host auf den Port 8081 des Mongo Express – Containers weiterzuleiten, damit du die Mongo-Express-Oberfläche aufrufen kannst
- ▶ **Zudem:**
 - ▶ Damit Mongo Express sich zu MongoDB verbindet, müssen Umgebungsvariablen gesetzt werden
 - ▶ Versuche erst mithilfe der Dokumentation auf Docker Hub selbst herauszufinden, welche Umgebungsvariable du setzen musst
 - ▶ Öffne schließlich die Anwendung im Browser

- ▶

```
docker network create my-new-network
```
- ▶

```
docker network ls
```
- ▶

```
docker run -dit --network my-new-network -v mariadb-vol:/var/lib/mysql -e  
MYSQL_ROOT_PASSWORD=password123 --name my-mariadb mariadb
```
- ▶
- ▶

```
docker run -d -p 8080:80 --network my-new-network -e PMA_HOST=my-  
mariadb --name my-phpmyadmin phpmyadmin/phpmyadmin/phpmyadmin
```
- ▶
- ▶
- ▶
- ▶ <http://localhost:8080/>
- ▶
- ▶
- ▶

Docker Bootcamp

Dockerfiles: Images automatisch bauen

Lernziele für diesen Abschnitt

- ▶ Was sind **Dockerfiles**?
- ▶ Wie sieht die Syntax von einem *Dockerfile* aus?
 - ▶ Wie lauten die wichtigsten Anweisungen?
- ▶ Wie funktioniert das Kommando **docker build**?
- ▶ Was sind **Multi-Stage Builds**?

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**
- ▶ Du kannst einen beliebigen Texteditor verwenden, um *Dockerfiles* zu erstellen

Das Kommando `docker build`

- ▶ Der Builder erzeugt aus einem *Dockerfile* ein Image
 - ▶ **`docker build [Pfad zum Dockerfile]`**
- ▶ Falls sich im aktuellen Arbeitsverzeichnis das *Dockerfile* befinden sollte, brauchst du nur auszuführen:
 - ▶ **`docker build .`**
 - ▶ der Punkt `.` verweist auf das aktuelle Verzeichnis
 - ▶ Wenn dein *Dockerfile* nicht **Dockerfile** heißen sollte, kannst du es mit der Flag **-f** (bzw. **--file**) ansteuern
 - ▶ **`docker build -f mydockerfile .`**
 - ▶ Du kannst auch ein Git-Repository via URL ansteuern
 - ▶ Mit der Flag **-t** bzw. (**--tag**) kannst du dem Image einen neuen tag zuweisen, um komfortabler darauf zugreifen zu können

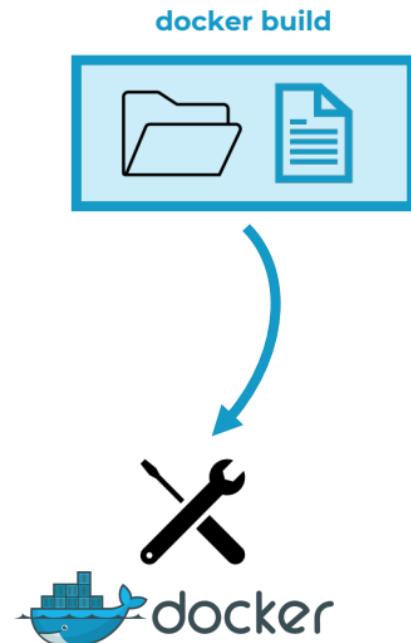


Docker Bootcamp

Dockerfiles: Der build-Prozess genauer beleuchtet

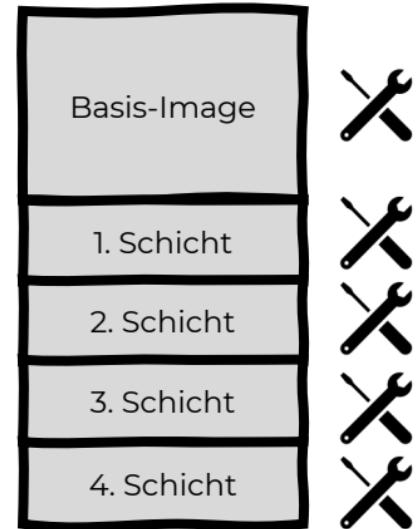
Was solltest du bei der Ordner-Struktur beachten?

- ▶ Der Pfad der **docker build** [Pfad] übergeben wird, wird zum Kontext für den Bau deines Images
- ▶ Mit Kontext ist die Gesamtheit aller Daten gemeint, auf die der *docker daemon* beim Abhandeln der Anweisungen im *Dockerfile* zugreifen darf
- ▶ Pass also auf, worauf du **docker build** ausführst!
- ▶ Best Practice:
 - ▶ speichere das *Dockerfile* in einem leeren Verzeichnis bzw. darin sollten sich nur die Dateien befinden, die nötig sind, um das Image zu bauen
 - ▶ Dann führst du **docker build** auf dieses Verzeichnis aus
 - ▶ Wenn du **docker build** auf dein root-Verzeichnis ausführst (z.B. weil du darin dein Dockerfile gespeichert hast), kann über das Dockerfile auf alle Daten zugegriffen werden



Was passiert bei docker build?

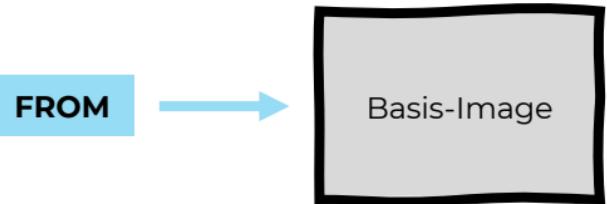
- ▶ Zunächst wird das gesamte *Dockerfile* validiert: bei einem Syntax-Fehler wird keine Anweisung ausgeführt und du erhältst eine Fehlermeldung („[internal]“)
- ▶ Falls die Validierung fehlerfrei abgeschlossen wurde, werden danach nacheinander die einzelnen Anweisungen ausgeführt
 - ▶ Bestimmte Anweisungen 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“)



Docker Bootcamp

Das Kommando: FROM

Das Kommando FROM



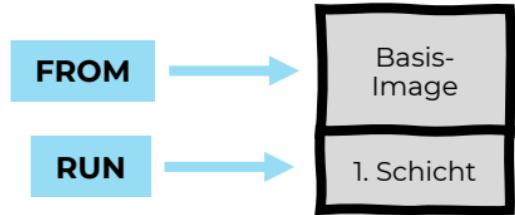
- ▶ Die Anweisung FROM sieht wie folgt aus:
 - ▶ **FROM [Image]:[tag]**
 - ▶ Als Basis-Image solltest du ein offizielles Image verwenden
 - ▶ Sehr oft wird das Image *Alpine* oder *Debian* verwendet
- ▶ Die erste Anweisung in einem *Dockerfile* muss **FROM [Name von einem Image]** lauten
- ▶ Es können auch mehrere **FROM**-Anweisungen in einem *Dockerfile* stehen
 - ▶ *Multi-Stage Builds* (später)
- ▶ Fun Fact: das Image, von dem alle anderen abstammen ist das scratch-Image (Wortspiel: „from scratch“)
 - ▶ https://hub.docker.com/_/scratch/

Docker Bootcamp

Das Kommando: RUN

Das Kommando **RUN**

- ▶ Befehle, die beim Bau des neuen Images ausgeführt werden sollen
- ▶ Setup: dienen in der Regel dazu das Basis-Image durch Installation zusätzlicher Pakete zu erweitern
- ▶ Typischer Use Case:
 - ▶ Alpine: **RUN apk update && apk add [Paket]**
 - ▶ Ubuntu / Debian: **RUN apt-get update && apt-get install -y [Paket]**
- ▶ Jede **RUN**-Zeile fügt eine neue Schicht zum neuen Image hinzu
- ▶ Deswegen: einzelne Befehle mit **&&** bündeln

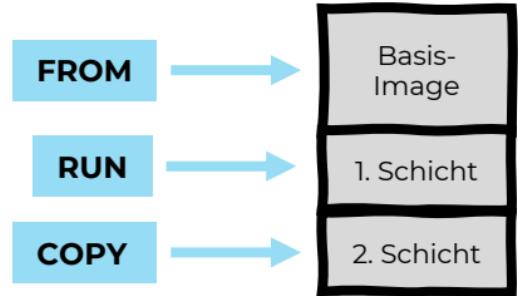


Docker Bootcamp

Die Kommandos: ADD & COPY

Die Kommandos ADD & COPY

- ▶ **COPY**: kopiert lokale Daten aus dem Projektverzeichnis in das Image
 - ▶ Sollte im Regelfall verwendet werden!
- ▶ **ADD**: wie **COPY** aber mit einigen (nicht-offensichtlichen) Zusatzfeatures
 - ▶ Erlaubt auch per URL, Daten aus dem Internet zu kopieren
 - ▶ Allerdings, best practice: stattdessen **curl** oder **wget** in einer **RUN**-Anweisung nutzen
 - ▶ Das ist oft sehr viel übersichtlicher, als den ADD-Befehl zu verwenden
 - ▶ Entpackt Archive automatisch (gzip, tar, bzip2, xz)
 - ▶ Einziger Fall, in dem du **ADD** statt **COPY** verwenden solltest!
 - ▶ Bei beiden Anweisungen wird jeweils eine neue Zwischenschicht zum neuen Image hinzugefügt



Docker Bootcamp

Das Kommando: CMD

Das Kommando: CMD

- ▶ Mit CMD kannst du ein Kommando definieren, welches standardmäßig ausgeführt wird
- ▶ Hierbei ist die Schreibweise wie folgt:
 - ▶ Exec-Form (offiziell empfohlen):
 - ▶ Das Programm wird direkt gestartet
 - ▶ `CMD ["Programm / Datei", "Parameter 1", "Parameter 2"]`
 - ▶ Shell-Form:
 - ▶ Hier wird eine Shell gestartet, die das Programm dann aufruft
 - ▶ Vorteil: Platzhalter wie z.B. `"$"`, und Variablen können aufgelöst werden
 - ▶ `CMD Programm/Datei Parameter 1 Parameter 2`
- ▶ **Wichtig:**
 - ▶ Es wird immer nur das letzte CMD-Kommando beachtet!

Docker Bootcamp

Das Kommando: ENTRYPOINT

Das Kommando: ENTRYPOINT

- ▶ **ENTRYPOINT**: Ähnlich wie CMD, allerdings mit ein paar

Unterschieden:

- ▶ Zusätzliche Befehle, die bei **docker run** als Parameter mitgegeben werden, werden als Parameter interpretiert
- ▶ Ein solcher Hauptbefehls ermöglicht es dir den Container wie eine ausführbare Datei zu nutzen („service-based image“)
- ▶ Gibt es ENTRYPOINT und CMD, so werden die Kommandos aus CMD an den Entrypoint drangehängt
- ▶ Mit dem Parameter **--entrypoint** bei **docker run** kannst du den **ENTRYPOINT** manuell überschreiben

Wann was?

- ▶ CMD:
 - ▶ Grundsätzlich zu bevorzugen
- ▶ ENTRYPOINT:
 - ▶ Ausschließlich für Service-based Images

Docker Bootcamp

Das Kommando: WORKDIR

Das Kommando: Workdir

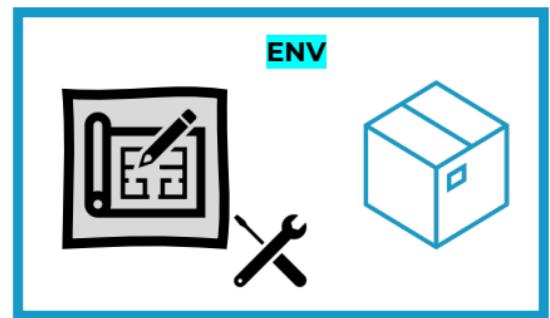
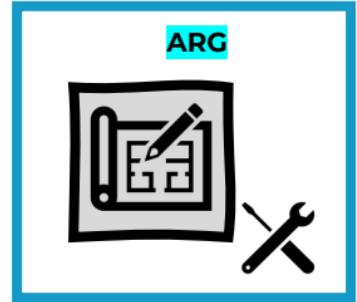
- ▶ **WORKDIR**: den angegebenen Pfad als aktuelles Arbeitsverzeichnis setzen
- ▶ Anstatt **RUN cd [Pfad]** verwenden (besser zu lesen und zu warten)

Docker Bootcamp

Build- und Umgebungsvariablen

Build-Variablen & Umgebungsvariablen

- ▶ Der Scope von **ARG**-Variablen beschränkt sich auf die Bauphase des neuen Images: danach kannst du **ARG**-Variablen nicht mehr nutzen!
- ▶ insbesondere nicht zur Laufzeit in einem Container (**ENTRYPOINT** oder **CMD**-Anweisungen)
- ▶ Mit der **ARG**-Anweisung kannst du Parameter vom **docker build** - Befehl in Variablen speichern
 - ▶ **ARG build_variable=default_wert**
 - ▶ **docker build --build-arg build_variable=neuer_wert .**
- ▶ Mit **ENV** kannst du Umgebungsvariablen definieren, die sowohl in der Bauphase als auch für Container verfügbar sind
 - ▶ also auch in **ENTRYPOINT** oder **CMD**-Anweisungen
 - ▶ Diese können wir beim Erstellen eines Containers setzen (über -e)
 - ▶ Viele Images (die du als Basis benutzt) verfügen über bereits definierte Umgebungsvariablen
 - ▶ Beispiel: **MARIADB_ROOT_PASSWORD** (MariaDB-Image)
 - ▶ Dokumentationen lesen



Docker Bootcamp

Das Kommando: VOLUME

Das Kommando **VOLUME**

- ▶ **VOLUME** spezifiziert, wo später in einem Container ein Volume erzeugt werden soll
- ▶ Nimmt nur einen Parameter entgegen: Pfad innerhalb des Containers zum Verzeichnis, zu dem ein Volume erstellt werden soll
- ▶ Enthält keinen Pfad in das Host-System, um die Portierbarkeit des Images zu gewährleisten
- ▶ Wenn wir das Volume benennen möchten, können wir dies über –v zum Zeitpunkt der Container-Erstellung tun
- ▶ **Use Case:** Sicherstellen, dass eine Datenbank die Datenbasis immer in ein Volume auslagert – selbst wenn der Container ohne den –v-Parameter erstellt wurde

Docker Bootcamp

Das Kommando: EXPOSE

Das Kommando **EXPOSE**

- ▶ Mit EXPOSE [PORT] bzw. EXPOSE [PORT]/[PROTOCOL] können wir spezifizieren, dass dieser Container z.B. auf einem Port lauschen soll
- ▶ So wirklich viel macht das Kommando aber nicht:
 - ▶ Wir können auch so einen Port auf unserem Container öffnen
 - ▶ Wir können auch so einen Port über -p vom Host zum Container weiterleiten
 - ▶ Allerdings: Wenn wir bei docker run alle Ports automatisch mappen möchten (Option: -P), dann werden nur Ports verwendet, die mit EXPOSE angegeben worden sind
- ▶ **Beispiele:**
 - ▶ EXPOSE 80/tcp
 - ▶ EXPOSE 443/tcp
 - ▶ EXPOSE 80
 - ▶ EXPOSE 443

Docker Bootcamp

Sicherheitshinweis: ENV und ARG

Sicherheitshinweis: ENV und ARG

- ▶ Wenn wir über ENV oder ARG Parameter definieren, sind diese später immer noch einsehbar (z.B. über die docker image history)
- ▶ Wir sollten diese also nur für unsensible Informationen verwenden
- ▶ Wir sollten uns dem also bewusst sein, wenn wir z.B. unser Image später an Dritte übertragen

Beispiel: Flask-Anwendung in Dockerfile

Docker Bootcamp

Aufgabe: VIM-Editor zum Üben

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 kannst du den Editor wieder schließen
- ▶ **Aufgabe:**
 - ▶ Erstelle ein Docker Image, welches vim als Editor ausführt, und eine hinterlegte Datei (z.B. text.txt) standardmäßig öffnet
 - ▶ Du kannst hier 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

Docker Bootcamp

Dockerfiles (Teil 2)

Dockerfiles – Teil 2

► In diesem Teil:

- ▶ LABEL (Weitere Metadaten hinzufügen)
- ▶ Dateien ignorieren (.dockerignore)
- ▶ Alpine Linux (glibc vs. musl)
- ▶ CPU-Architekturen & Dockerfiles
- ▶ Fortgeschrittene Dockerfiles:
 - ▶ Multi-Stage-Builds

Docker Bootcamp

Dockerfiles: LABEL

Das Kommando: LABEL

- ▶ **LABEL**: Metadaten zum Image hinzufügen, um ein Projekt zu organisieren
- ▶ **LABEL [Schlüssel]=[Wert]**
- ▶ Darf erst nach einer **FROM**-Anweisung stehen
- ▶ Labels können mit **docker image inspect** eingesehen werden

Docker Bootcamp

Dateien ignorieren: `.dockerignore`

.dockerignore: Daten nicht berücksichtigen

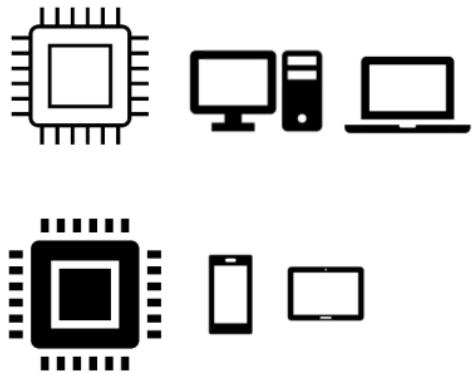
- ▶ In der **.dockerignore** - Datei führst du Dateien und Verzeichnisse auf, die beim Bau des Images nicht berücksichtigt werden
 - ▶ Achtung: Versteckte Datei (Punkt am Anfang vom Dateinamen)
- ▶ Die *.dockerignore* muss in dem Verzeichnis liegen, für das wir docker build ausführen
- ▶ Du kannst Namen explizit in die *.dockerignore* schreiben, aber auch *Patterns* mithilfe der Wildcard ***** vorgeben:
 - ▶ **data***: alle Dateien und Verzeichnisse (im root-Verzeichnis), die mit der Zeichenfolge „data“ beginnen, werden ignoriert (z.B. eine Datei data.txt)
 - ▶ ***data**: alle Dateien und Verzeichnisse (im root-Verzeichnis), die mit der Zeichenfolge „data“ aufhören, werden ignoriert (z.B. ein Ordner more-data)
 - ▶ ****/data**: Ordner in allen Unterverzeichnissen, die *data* heißen, werden ignoriert
 - ▶ Hierbei darf der Ordner "data" auch in einem tiefen Unterverzeichnis liegen (daher hier die mehreren ******). Beispielsweise würde auch folgender Ordner ignoriert:
 - ▶ a/b/data

Docker Bootcamp

Sonstiges: CPU-Architekturen

CPU-Architekturen: ARM vs. x86

- ▶ **x86**: eine Mikroprozessor-/Befehlssatz-Architektur, die bislang standardmäßig bei CPUs in Laptops und Desktoprechnern verwendet wird
 - ▶ insbesondere stellen Intel und AMD x86-kompatible CPUs her
- ▶ **ARM**: eine alternative Mikroprozessor-/Befehlssatz-Architektur, die vorwiegend in Smartphones und Tablets zum Einsatz kommt
 - ▶ Aber auch die Apple-Chips für MacBooks basieren auf ARM
 - ▶ Es scheint einen Trend hin zu ARM-basierten CPUs zu geben



CPU-Architekturen und Docker

- ▶ Docker verwendet den Kernel des Host-Betriebssystems mit
- ▶ Ein Ändern der CPU-Architektur ist daher nicht so ohne weiteres möglich
- ▶ **Das bedeutet:**
 - ▶ Ein ARM-Host führt i.d.R. einen Container mit ARM-Architektur aus (linux/arm64)
 - ▶ Ein x86_64-Host führt den Container mit der x86_64-Architektur aus (linux/amd64)
- ▶ Auf bestimmten Systemen werden aber auch andere Platformen unterstützt
- ▶ **Beispiel:**
 - ▶ `docker run -it --platform=linux/amd64 ubuntu`
 - ▶ Führt auf einem modernen Mac den Ubuntu-Container als 64-bit x86-Container aus
 - ▶ Achtung: Führt zu signifikanten Performance-Overhead, da ein gesamter Prozessor emuliert werden muss!
- ▶ **In der Praxis:**
 - ▶ Für jede CPU-Architektur funktioniert unser Image u.U. leicht unterschiedlich
 - ▶ Wir müssen das als Nutzer also wissen und beachten!

Docker Bootcamp

Vorstellung: Alpine Linux

Alpine Linux

- ▶ Alpine Linux: sehr kleine (wenige MB groß) Linux-Distribution
 - ▶ Wird häufig als Basis-Image von Containern verwendet
(alternativ Debian)
 - ▶ Verwendet als Standard-C-Bibliothek **musl** statt **glibc (GNU C Library)**
 - ▶ Alpine nutzt zur Paketverwaltung das Tool **apk** statt das *Advanced Packing Tool (apt)* wie z.B. Ubuntu



Docker Bootcamp

Vorstellung: Alpine Linux (Teil 2)

musl vs. glibc

- ▶ Was ist musl bzw. glibc?
 - ▶ Beides sind Implementierungen der C-Standard-Bibliothek
 - ▶ Stellt diverse grundlegende Funktionen zur Verfügung:
 - ▶ Einlesen und Öffnen von Dateien
 - ▶ Ein- und Ausgaben von Programmen (z.B. printf)
 - ▶ Umwandlung von Kodierungen (z.B. iconv für Unicode / UTF-8)
 - ▶ Formatierung von Datumsangaben
 - ▶ Reguläre Ausdrücke
 - ▶ Grundlegende Netzwerkkommunikation
 - ▶ glibc:
 - ▶ Der defacto-Standard unter Linux
 - ▶ Musl:
 - ▶ Kleinere, modernere Implementierung der C-Standard-Bibliothek

Problem 1: Software funktioniert u.U. anders

- ▶ Code ist oft auf glibc optimiert
- ▶ Und wir als Programmierer gehen oft nicht davon aus, dass die C-Standard-Bibliothek anders funktioniert
- ▶ **Beispiel:**
 - ▶ Performance-Unterschiede:
 - ▶ <https://news.ycombinator.com/item?id=27379482>
 - ▶ strftime hat z.B. in Python anders funktioniert unter musl vs. Glibc
 - ▶ <https://bugs.python.org/issue34672>

Das Problem: musl vs. glibc

- ▶ Wenn wir Alpine Linux verwenden (musl):
 - ▶ Code, der auf der C-Standard-Bibliothek aufbaut, muss für musl kompiliert sein
 - ▶ **Beispiel:**
 - ▶ pip install pandas
 - ▶ Damit wird das Python-Tool "pandas" nachinstalliert (Python und pip muss natürlich installiert sein im Container)
 - ▶ Das pip-Repository stellt folgendes zur Verfügung:
 - ▶ Den Quellcode, um das Tool "pandas" zu bauen
 - ▶ Und ggf. eine fertig kompilierte Version von Pandas
 - ▶ Oft steht dort aber nur eine Version für glibc-Betriebssysteme zur Verfügung
 - ▶ Auf Alpine Linux muss der Code also manuell kompiliert werden
 - ▶ ... Unter Ubuntu kann das Paket fertig kompiliert (als .whl) heruntergeladen werden

Docker Bootcamp

Multi-Stage-Builds

Was ist ein Multi-Stage Build?

► Problem:

- ▶ Images sind oft unnötig groß, weil sie z.B. einen Compiler enthalten

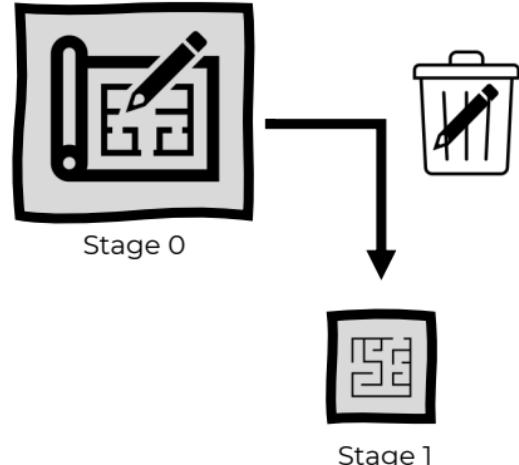
- ▶ Dieser wird später zur Laufzeit aber gar nicht mehr benötigt

▶ Lösung:

- ▶ **Multi-Stage Build:** Build-Image(s) und Produktivbetriebs-Image

- ▶ Wir erzeugen erst die für unsere Anwendung benötigten Artefakte (z.B. Binärdateien) und kopieren sie dann in ein kleineres Image (in der Regel die Linux-Distribution *Alpine*)

- ▶ Dieses Image für den Produktivbetrieb enthält nur noch die Komponenten, die zur Ausführung notwendig sind (aber z.B. keinen *Compiler* mehr)



Was ist ein Multi-Stage Build?

- ▶ Praxis:
 - ▶ Wir verwenden in einem *Dockerfile* mehrere **FROM**-Anweisungen
 - ▶ Du kannst dann z.B. mit **COPY --from=0 [Ort alt] [Ort neu]** Artefakte aus vorherigen Bauphasen in die aktuelle Phase kopieren und nutzen
 - ▶ Die Phase sind aufsteigend nummeriert, beginnend bei 0
 - ▶ Du solltest die Bauphasen mit **AS** benennen und sie über ihre Namen ansteuern
 - ▶ **FROM golang:1.15.1-alpine AS builder**
 - ▶ **...**
 - ▶ **COPY --from=builder [Ort im alten Image] [Ort im neuen Image]**

Docker Bootcamp

Multi-Stage-Builds, Projekt

Docker Bootcamp

Übungsaufgabe: Dockerfiles

Aufgabe: Express.js-App

So wie wir das für die Flask-App getan haben, wollen wir hier für eine funktional ähnliche Node.js-Anwendung (genauer benutzen wir das Framework Express.js) ein Image mit einem Dockerfile erstellen. Die App liegt als Anhang an diese Lektion vor.

Aufgabe:

- ▶ Schreibe das Dockerfile, um die Anwendung zu containerisieren
- ▶ Erzeuge ein Image auf Basis von deinem Dockerfile
- ▶ Starte einen Container, in dem die App ausgeführt wird und öffne sie im Browser

Aufgabe: Express.js-App

Tipps:

- ▶ Als Basis verwendest du ein Node-Image
- ▶ Ein sinnvoller Speicherort in dem Node-Image für die App-Dateien wäre etwa das Verzeichnis /web-app
- ▶ Denke daran, sowohl die *app.js* als auch die *package.json* zu kopieren
- ▶ Um die Dependencies zu installieren brauchst du hier den Befehl **npm install**
 - ▶ Bezieht sich auf die *package.json* -> die muss dann schon im Image vorhanden sein
 - ▶ Sinngemäß vergleichbar mit **pip3 install -r requirements.txt** bei der Flask-App
- ▶ Den Web-Server kannst du per **node app.js** starten, es sind hier keine zusätzlichen Parameter notwendig!
- ▶ Den Container-Port ist 8080 (siehe *app.js*)

Docker Bootcamp

Docker Compose: Mehrere Container

Lernziele für diesen Abschnitt

- ▶ Was ist ein **Service**?
- ▶ Wie ist eine **docker-compose.yaml**-Datei aufgebaut?
 - ▶ Wofür ist das Format **YAML** gut?
- ▶ Wie funktioniert der Befehl **docker compose**?

Was ist docker compose?

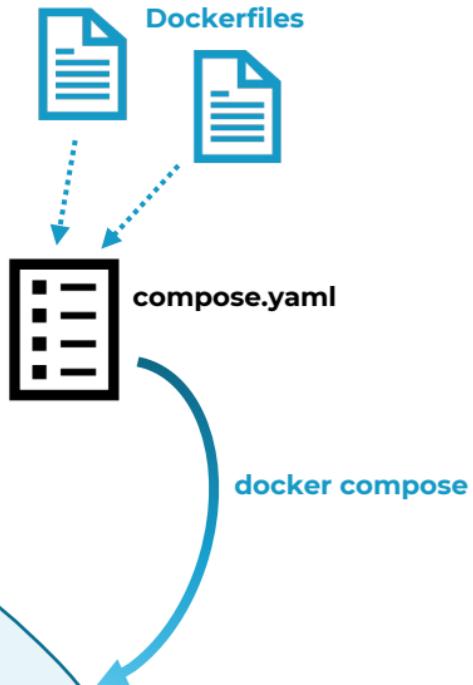
► Problem:

- ▶ Bislang ist es aufwändig Anwendungen aufzusetzen, die aus mehreren Containern bestehen
- ▶ Wir müssen für jeden Container separat **docker run** ausführen
- ▶ Mit *Dockerfiles* können wir zwar einzelne Images konfigurieren, aber keinen Verbund von Containern
 - ▶ z.B. können wir Images per *Dockerfile* nicht miteinander verbinden

Was ist docker compose?

► Lösung:

- ▶ 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
- ▶ Du kannst **docker run** - Befehle als **compose.yaml (bzw. docker-compose.yaml)** Konfigurationen umformulieren



Docker Bootcamp

Exkurs: YAML

Was ist YAML?

- ▶ Ziel: Daten in für Menschen gut lesbarer Form darstellen
- ▶ **YAML** steht für **YAML ain't Markup Language**
- ▶ Vereinfachte Auszeichnungssprache ("markup language")
 - ▶ ähnlich **XML** oder **JSON**
 - ▶ Jedes JSON-Dokument ist ein gültiges YAML-Dokument
- ▶ Endungen: **.yml** oder **.yaml**
- ▶ Kannst du prinzipiell in allen Programmiersprachen nutzen,
sofern ein passender Parser vorhanden ist



YAML Basics

- ▶ Wir brauchen nur wenige Elemente von YAML
 - ▶ **Einrückungen** legen Strukturen fest!
 - ▶ Die wichtigsten Datenstrukturen für uns sind:
 - ▶ **Arrays**
 - ▶ **Maps**
 - ▶ **Datentypen sind quasi wie in JSON:**
 - ▶ Zahlen
 - ▶ Booleans
 - ▶ Strings

Docker Bootcamp

Aufbau der compose.yaml

Der Aufbau von `compose.yaml`

- ▶ Üblicherweise wird die Konfigurationsdatei für `docker compose` **compose.yaml (empfohlen)** oder **docker-compose.yaml** genannt
- ▶ Du kannst sie aber auch **compose.yml** oder **docker-compose.yml** nennen
- ▶ Ganz oben sieht man sehr häufig die Spezifikation der Version
 - ▶ **version: '3'**
 - ▶ Aktuell: 3
 - ▶ Legacy: 2
 - ▶ Ist aber veraltet: <https://github.com/compose-spec/compose-spec/blob/master/spec.md>

Services spezifizieren: `compose.yaml`

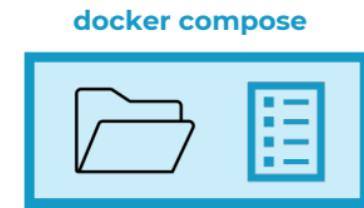
- ▶ Unsere Dienste (jetzt erstmal: Container die gestartet werden) können wir wie folgt definieren:
 - ▶ **services:**
 - ▶ Erforderlich!
 - ▶ Hier werden die einzelnen Container aufgeführt, die du konfigurieren und starten (und evtl. replizieren) möchtest
 - ▶ Oft: Ein Service entspricht einem Container
 - ▶ 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*)
- ▶ **expose**: Ports freischalten
- ▶ **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>

Das Kommando `docker compose`

- ▶ Ähnlich wie bei einem `Dockerfile` solltest du ein leeres Projektverzeichnis anlegen und darin nur eine **`docker-compose.yaml`** sowie benötigte Daten speichern (Kontext!)
- ▶ Besonders praktisch ist das Kommando **`docker compose up`**
 - ▶ Images ggf. herunterladen
 - ▶ Container erzeugen
 - ▶ Netzwerk erzeugen
 - ▶ Container im Netzwerk starten
- ▶ anders als bei **`docker build`** brauchst du keinen Pfad als Parameter zu übergeben, der Standardpfad ist nämlich auf **`./compose.yaml`** bzw. **`./docker-compose.yaml`** gesetzt
- ▶ Du siehst via **`docker network ls`**, dass standardmäßig ein *bridge*-Netzwerk eingerichtet wird, benannt nach dem Projektverzeichnis



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`**

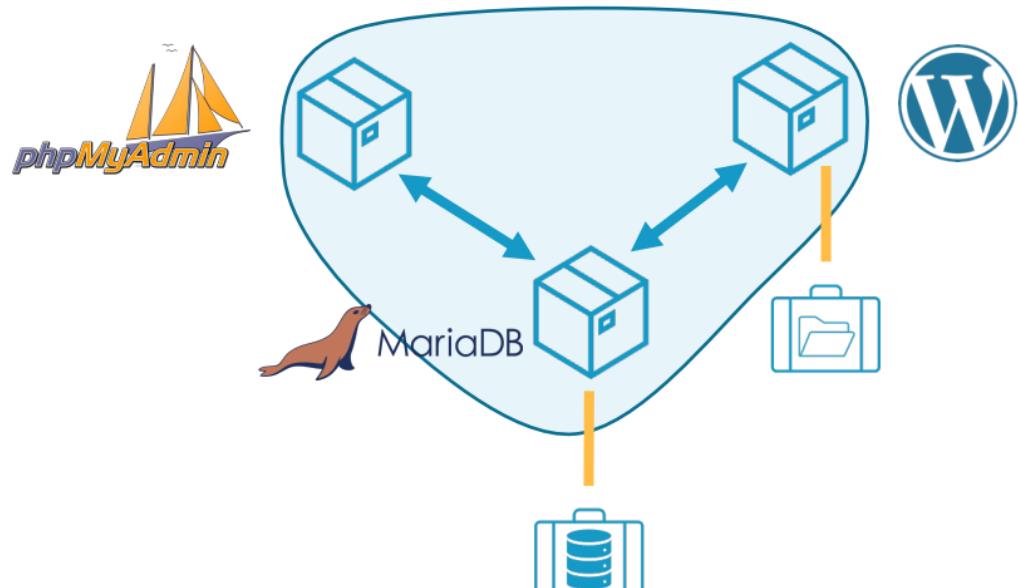
- ▶ Wir werden zwei Services (Flask & express.js) in mehreren Containern starten
- ▶ Ordner: **scaling_services**
- ▶ Im Rahmen von einem Service können wir die Anzahl der Container skalieren
 - ▶ **docker compose up --scale flask-app=2**
 - ▶ Kommentare vor **deploy: replicas: 3** löschen
 - ▶ Wir haben immer noch zwei Services, aber für jeden mehrere Container erzeugt
- ▶ Hier ist das Problem, dass jeder replizierte Container einem anderen Host-Port zugewiesen werden muss!
 - ▶ Indem wir beim Service nur einen Port setzen, wird dieser als Container-Port interpretiert und Docker sucht automatisch freie Host-Ports
 - ▶ Die Ports findest du per **docker container ls** bzw. **Docker Desktop** heraus

Services skalieren

- ▶ Du kannst die Flag **--scale** setzen, um mehrere Container in einem Service zu erzeugen
 - ▶ Dazu musst du bei der Einstellung **ports** nur einen Port (den Container-Port) setzen
 - ▶ Die Host-Ports werden dann automatisch vergeben
 - ▶ Eine Port-Range in Verbindung mit **--scale** zu setzen führt regelmäßig zu Problemen
 - ▶ <https://github.com/docker/compose/issues/7188>

Ein bekanntes Beispiel

- ▶ Wir wollen schrittweise dieses frühere Beispiel mit **docker** **compose** nachbauen und verbessern
- ▶ Container im Netzwerk per Umgebungsvariablen miteinander verbinden
- ▶ Volumes hinzufügen



Volumes konfigurieren

- ▶ Du musst 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

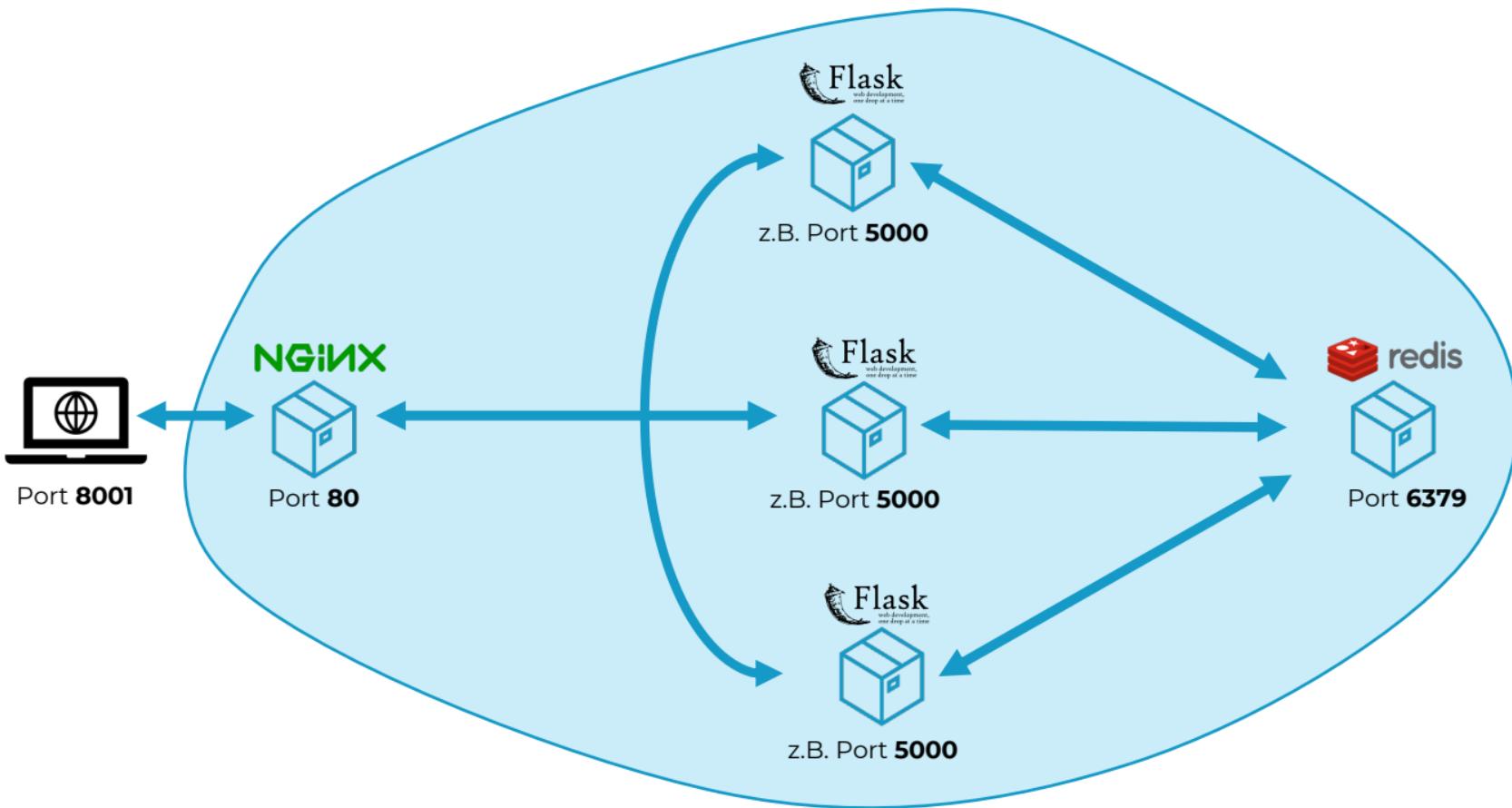
- ▶ Um die Konfigurationsdatei noch cleaner zu gestalten, kannst du Umgebungsvariablen in eine **.env** – Datei auslagern
- ▶ Wegen dem Punkt am Anfang des Dateinamens ist sie versteckt
- ▶ In jeder Zeile von der .env steht **variable=wert**
- ▶ Dann kannst du in der *compose.yaml* schreiben:
 - ▶ **umgebungsvARIABLE: "\$variable"**
- ▶ Standardmäßig sucht Docker nach der **.env** im root von deinem Projektverzeichnis
- ▶ Alternative Dateinamen kannst du per Flag **--env-file** manuell setzen oder via **env_file** direkt in die *docker-compose.yaml* eintragen

Docker Bootcamp

Übungsaufgabe: Docker Compose

Aufgabe 1: (Flask-Redis-App)

- ▶ Ziel von diesem Projekt ist es eine Webanwendung bestehend aus Frontend (Flask) und Backend (Redis) sowie einem Proxy-Server mit Hilfe von **docker compose** auszuführen
- ▶ Die prinzipielle Vorgehensweise kannst du auf eine Vielzahl von ähnlichen Anwendungsfällen übertragen!
- ▶ Das Projekt wird dir in den Kursmaterialien zur Verfügung gestellt.
- ▶ 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)

Anleitung:

- ▶ Schreibe die `compose.yaml`, in der du drei Services definierst: Frontend, Backend und Proxy-Server
- ▶ Für deinen Backend-Service verwendest du das Standard-Redis-Image, bei den anderen beiden Services nutzt du deine anhand der beiden Dockerfiles gebauten Images
- ▶ Damit sich alle Container richtig verbinden können und deine Anwendung auch wirklich läuft, ist es entscheidend deinen Frontend- und deinen Backend-Service richtig zu benennen:
 - ▶ Den Namen vom Frontend-Service kannst du aus der `nginx-proxy/nginx.conf` erschließen
 - ▶ Den Namen vom Backend-Service kannst du aus der `flask-app/app.py` erschließen

Docker Bootcamp

Docker Swarm

Worum geht es in diesem Abschnitt?

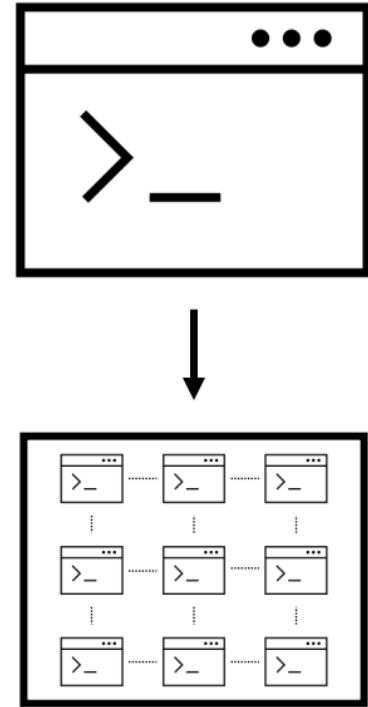
- ▶ Bislang haben wir uns mit der **Entwicklung** von Anwendungen beschäftigt (Dockerfiles, Docker Compose,...)
- ▶ In diesem Abschnitt geht es um das Deployment auf ein Cluster mit Docker Swarm
- ▶ Wir schauen uns hier Themen an wie:
 - ▶ Was sind **Microservices**?
 - ▶ Welche Funktion erfüllt der **Swarm-Mode** von Docker?
 - ▶ Wie benutzt du den Swarm-Mode?

Docker Bootcamp

Zuerst: Was sind Microservices?

Was sind Microservices?

- ▶ Paradigmenwechsel im Software-Engineering:
 - ▶ Von "monolithischen Architekturen" zu **Microservices**
 - ▶ **Monolithisch:** alle Komponenten in einer Code-Basis
 - ▶ **Microservices:** Funktional eng begrenzte und eigenständige Software-Komponenten, die über Schnittstellen miteinander kommunizieren
- ▶ **Vorteile von Microservices:**
 - ▶ Leichtere Aktualisierung/Weiterentwicklung durch gezielten Austausch von Komponenten möglich
 - ▶ Verschiedene Technologien lassen sich besser miteinander zu kombinieren
 - ▶ Mehrere Entwicklerteams können parallel arbeiten
 - ▶ Skalierbarkeit: flexible Verteilung von Rechenlast auf mehrere Hosts
 - ▶ Ausfallsicherheit
- ▶ **Nachteile:**
 - ▶ Zunahme an Komplexität beim Entwickeln, Testen & Deployen
 - ▶ Kompliziertere Kommunikation zwischen Services
 - ▶ Z.B. um die Konsistenz von Datenbanken zu gewährleisten
 - ▶ Architektur insgesamt unübersichtlicher



Wichtiger Hinweis: Persistenz bei Microservices

- ▶ Microservices sollen oft automatisch skaliert werden können

- ▶ **Beispiel:**

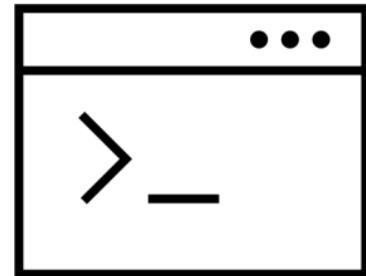
- ▶ Wenn mehr Anfragen an eine Webseite gestellt werden, können automatisch weitere Server mit Webseiten gestartet werden (z.B. in der Cloud)
- ▶ Oft sind Microservices daher "stateless" (zustandlos)

- ▶ **Was bedeutet stateless?**

- ▶ Sie speichern keine Daten
- ▶ Die gesamte Datenspeicherung ist ausgelagert
- ▶ Beispielsweise an eine große Datenbank, oder an eine Database-as-a-Service (z.B. von Google Cloud oder Amazon AWS)

- ▶ **Wichtig:** Es gibt natürlich auch stateful Microservices!

- ▶ Wir schauen uns in diesem Abschnitt an, wie wir zustandlose Microservices mit Docker Swarm skalieren können

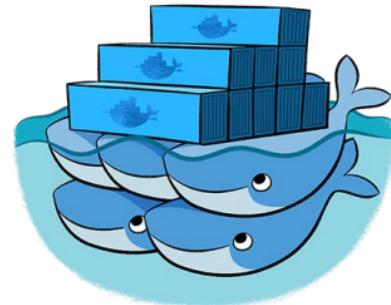


Docker Bootcamp

Warum Docker Swarm?

Warum brauchst du ein Tool zur Container-Orchestrierung?

- ▶ Unser Ziel: Anwendungen **steuern und verwalten**, die aus mehreren Containern bestehen (deployment)
- ▶ **Die Anwendung hier:**
 - ▶ Die Container sollen sich auf verschiedenen physischen Servern in einem Netzwerk befinden und ausgeführt werden (Cloud-Architektur)
 - ▶ In der professionellen Praxis: Cluster bestehen aus hundert bis tausend separaten Containern!
- ▶ Docker hat dazu bereits ein Tool integriert: **Docker Swarm**
 - ▶ Im Prinzip: *docker compose* aber auf mehreren Host-Systemen
 - ▶ Ermöglicht den Produktivbetrieb ("Deployment") von Multi-Container-Anwendungen in der Cloud
- ▶ **Ein Docker Swarm ist also ein Cluster von Docker Engines**



Was liefert uns ein solches Tool?

- ▶ **Durchgängige Verfügbarkeit:**

- ▶ Vermeidung von Downtime

- ▶ **Skalierbarkeit von Anwendungen:**

- ▶ Performante Ausführung sorgt für geringe Ladezeiten, schnelle Reaktionszeiten
 - ▶ Wir können komfortabel auch nachträglich noch weitere Server hinzufügen

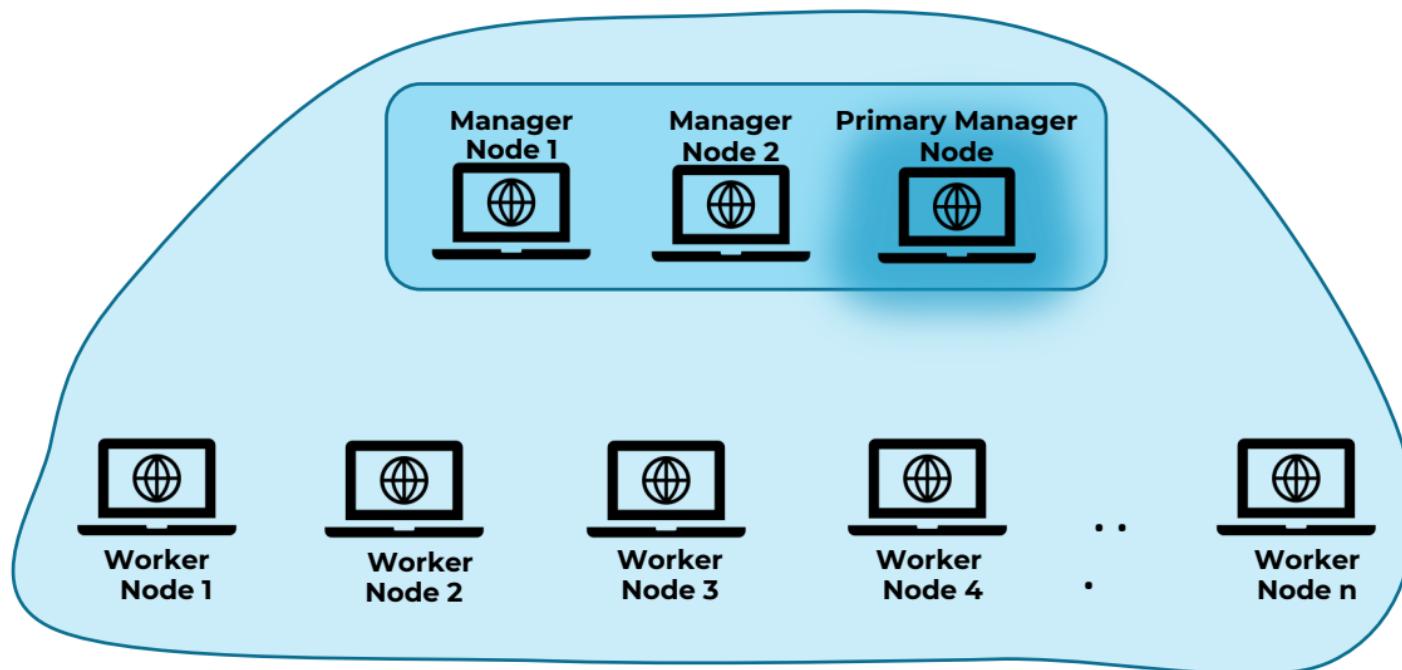
- ▶ **Absicherung vor Infrastruktur-Schäden:**

- ▶ Bereitstellung von Backups und Wiederherstellung von früheren Zuständen

Docker Bootcamp

Wie ist Docker Swarm aufgebaut?

Wie ist ein Swarm aufgebaut?



Was sind Nodes in einem Swarm?

- ▶ Alle Server in einem Swarm werden als **Nodes** bezeichnet
- ▶ Es gibt zwei Arten von Nodes:
 - ▶ **Manager Nodes**
 - ▶ Kommunizieren mit den Worker Nodes
 - ▶ Verteilen und starten Prozesse auf den Nodes
 - ▶ Es kann mehrere Manager Nodes geben, aber nur einen
 - ▶ **Primary Manager Node**
 - ▶ **Worker Nodes**
 - ▶ Führen Prozesse aus, die von Manager Nodes zugewiesen wurden
 - ▶ Kommunizieren ihren Status zum Manager Node

Docker Swarm testen mit einem Node

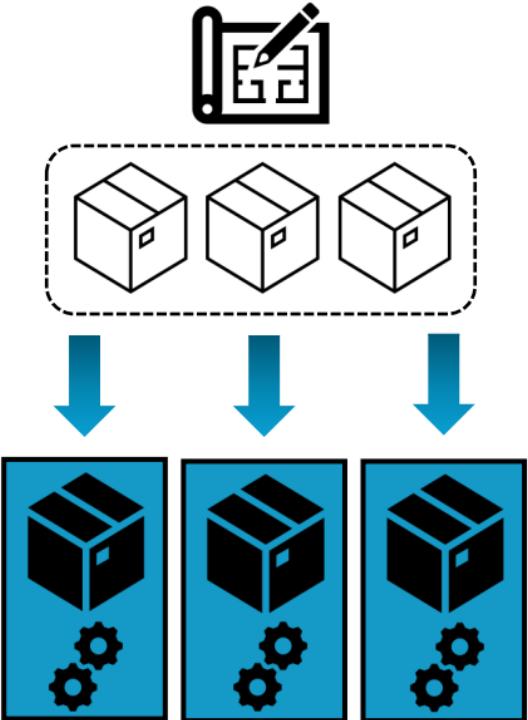
- ▶ Nachschauen, ob Docker Swarm aktiv ist
 - ▶ **docker system info**
- ▶ Einen Swarm starten:
 - ▶ **docker swarm init**
 - ▶ Erzeugt in unserem Fall einen Swarm, der nur aus einem Manager Node besteht
 - ▶ Einen Swarm wieder verlassen
 - ▶ **docker swarm leave --force**
 - ▶ Infos über Nodes anzeigen (auf einem Swarm-Manager):
 - ▶ **docker node ls**
 - ▶ Ein Node sollte angezeigt werden mit *Manager Status: Leader*

Docker Bootcamp

Wie starten wir etwas im Swarm?

Services vs. Tasks

- ▶ Docker Swarm basiert auf dem Prinzip eines "deklarativen Modells":
 - ▶ Wir legen Zielzustand fest, und Swarm versucht diesen zu erreichen
- ▶ **Service:** Beschreibung des Zielzustands
 - ▶ Beispiel: bestimmte Anzahl von Replicas von einem Container
- ▶ **Task:** Durchführung von Arbeitsstufen, um diesen Zustand herzustellen
- ▶ Im Rahmen von Docker Swarm kann ein einzelner Container als Instanzierung eines Tasks betrachtet werden
- ▶ Tasks werden also von / durch Services gestartet



Wie managen wir einen Service?

- ▶ Hierzu haben wir verschiedene Befehle zur Verfügung:
 - ▶ **docker service create:** Hiermit erstellen wir einen Service
 - ▶ **docker service ls:** Services anzeigen
 - ▶ **docker service update:** Service-Konfiguration aktualisieren
 - ▶ **docker service remove:** Services entfernen
 - ▶ **docker service ps:** Alle Tasks auflisten

Docker Bootcamp

Beispiel: Last verteilt?

Wird Load-Balancing ausgeführt?

- ▶ Hierzu haben wir verschiedene Befehle zur Verfügung:
 - ▶ **docker service create**: Hiermit erstellen wir einen Service
 - ▶ **docker service ls**: Services anzeigen
 - ▶ **docker service update**: Service-Konfiguration aktualisieren
 - ▶ **docker service remove**: Services entfernen
 - ▶ **docker service ps**: Alle Tasks auflisten

Docker Bootcamp

Routing-Mesh

Routing-Mesh

- ▶ Standardmäßig erzeugt Swarm für uns ein Routing-Mesh:
 - ▶ Jeder Node nimmt auf einem Port eine Anfrage entgegen, und schickt sie an ein beliebiges Replica im Cluster
 - ▶ Das funktioniert auch, wenn auf einer Node kein Replica dieses Services läuft
 - ▶ Das erzeugt natürlich internen Netzwerk-Traffic, da quasi alle Anfragen weitergeleitet werden an eine andere Node
 - ▶ Wenn wir das nicht möchten, können wir den Port auch direkt auf unseren physischen Servern öffnen
- ▶ **Das geht über:**
 - ▶ --mode global
- ▶ **Wichtig:**
 - ▶ Wir müssen uns dann selbst um das Load-Balancing kümmern!

Docker Bootcamp

Docker Swarm vs. Docker

Beispiel: compose.yaml zu Swarm deployen

Docker Bootcamp

Secrets

Secrets

- ▶ Oft möchten wir Passwörter in unserem Cluster verteilen (z.B. sollen alle Webserver das MySQL-Passwort kennen, etc.)
- ▶ Dafür gibt es in Docker Swarm einen eingebauten Mechanismus:
 - ▶ Die sogenannten Secrets
- ▶ Das sind Dateien, die uns dann in unserem Swarm zur Verfügung stehen
- ▶ Die Secrets werden dann in unserem Swarm (auf unserem Manager) gespeichert und verwaltet
- ▶ Wie verwenden wir Secrets?

Secrets: Verwendung

- ▶ Zuerst legen wir in der Shell ein Secret an:
 - ▶ `printf "This is a secret" | docker secret create top_secret -`
- ▶ Anschließend können wir das Secret in unserem Container verwenden:
 - ▶ `docker service create --secret top_secret php:8.1-apache`
- ▶ Das Secret liegt dann unter `/run/secrets/`

Docker Bootcamp

Configs in Docker Swarm

Configs

- ▶ Oft möchten wir auch Konfigurationsdateien im Cluster verteilen
- ▶ Das funktioniert über Configs
- ▶ Diese funktionieren ähnlich wie Secrets:
 - ▶ docker config create [Dateiname]
- ▶ Weitere Befehle:
 - ▶ docker config ls
 - ▶ docker config rm
- ▶ Können bis zu 500kb groß sein
- ▶ Wo ist der Unterschied zwischen Secrets und Configs?
 - ▶ Secrets: Werden verschlüsselt gespeichert, und intern als RAM-Disk gemounted
 - ▶ Configs: Werden unverschlüsselt gespeichert und als ohne RAM-Disk gemounted

Docker Bootcamp

Templated Configs in Docker Swarm

Templated Configs in Docker Swarm

- ▶ Docker Swarm kann auch eine Template-Engine auf unsere Konfigurationsdatei anwenden

- ▶ Beispiel:

```
▶ <html lang="en">
  <head>
    <title>Hello Docker</title>
  </head>
  <body>
    <p>Hallo mit Umgebungsvariable {{ env "HALLO" }}!. Ich bin Service {{ .Service.Name }}, mit Task-ID: {{ .Task.ID }}.</p>
  </body>
</html>
```

- ▶ Diese kann dann wie folgt hinzugefügt werden:

```
▶ docker config create --template-driver golang [config-name]
  index.html.tpl
```

Docker Bootcamp

Docker stack deploy

Docker Bootcamp

Ein echtes Cluster

Ein echtes Cluster

- ▶ In diesem Projekt werden wir ein echtes Cluster in Betrieb nehmen
- ▶ **Setup bei mir:**
 - ▶ 3x Raspberry Pi (armhf, raspbian Linux)
 - ▶ Alle im gleichen Netzwerk
- ▶ **Generell, was ist wichtig:**
 - ▶ Alle Computer müssen untereinander im Netzwerk kommunizieren können (u.A. auch fürs Routing Mesh)
 - ▶ Alle Computer müssen Zugriff auf die Images haben, die wir auf dem Manager ansteuern wollen
 - ▶ Alle Computer müssen Linux verwenden
- ▶ **Bei mir anscheinend auch:**
 - ▶ Docker muss immer mit root-Rechten gestartet werden, rootless-Docker hat bei mir nicht funktioniert

Ein echtes Cluster

- ▶ Worauf solltest du noch achten?
- ▶ Die Verbindung zu den Nodes sollte möglichst stabil sein
(idealerweise Ethernet, bei mir jetzt 5Ghz WLAN)
- ▶ Die Manager-Nodes sollten statische IPs haben
 - ▶ Das sorgt bei einem Reboot eines Managers dafür, dass sich das Cluster wieder besser neu aufbauen kann
- ▶ Das Netzwerk sollte hinreichend schnell sein

Docker Bootcamp

Bonus / Crash-Kurs: Kubernetes

Lernziele für diesen Abschnitt

- ▶ Was ist **K8s**?
- ▶ Wie ist die **K8s-Architektur** aufgebaut?
- ▶ Was ist **Minikube** und wie gehst du damit um?
- ▶ Wie kannst du **kubectl** verwenden?
- ▶ Was sind **Pods**?
- ▶ Was ist ein **Deployment**?
- ▶ Wie kannst du Anwendungen mit K8s "deployen"?
- ▶ Was sind die wichtigsten Objekte in K8s?

Wichtiger Hinweis

Wichtiger Hinweis

- ▶ Kubernetes ist ein unglaublich umfangreiches Thema
- ▶ Im Rahmen dieses Kurses kann ich dir daher nur einen Kurzeinstieg bieten
- ▶ **Das Ziel:**
 - ▶ Du kannst dich mit einem Cloud-Administrator über Kubernetes austauschen, und verstehst, was dieser von dir möchte

Docker Bootcamp

Was ist Kubernetes?

Was ist Kubernetes?

- ▶ Eine Plattform zur Steuerung und Verwaltung von Container-basierten Anwendungen
 - ▶ "kubernetes" griechisch für "Steuermann" / "Pilot"
- ▶ Deutlicher größerer Funktionsumfang als Docker Swarm
- ▶ Plattform nicht nur für Docker-basierte Container, sondern allgemeine Container-Engines
 - ▶ Insbesondere: **containerd**, **Mirantis Container Runtime**, **CRI-O**
 - ▶ Auch: **CoreOS rkt**, **runC** (Open Container Initiative)
- ▶ Häufig abgekürzt als **K8s** (8 steht für die Anzahl der ausgelassenen Buchstaben)
- ▶ Von Google in Go entwickelt
- ▶ 2015 als Open Source veröffentlicht
- ▶ Mit der Prüfung zum *Certified Kubernetes Administrator* (CKA exam) kannst du dich zertifizieren lassen



Was ist Kubernetes nicht?

- ▶ Es deployt weder unseren Quellcode, noch baut es unsere Anwendung
 - ▶ Wir müssen uns darum selbst kümmern (z.B. mit Docker)
- ▶ Es kümmert sich nicht um unsere physischen Maschinen - die müssen wir für Kubernetes bereitstellen
- ▶ Kubernetes bietet keine Dienste auf Anwendungsebene

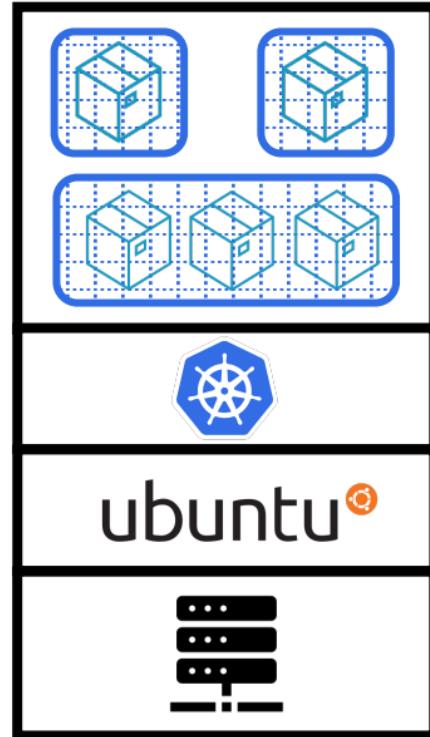
Docker Bootcamp

Kubernetes Grundlagen: Was ist ein Pod?

Was ist ein Pod?



- ▶ In K8s interagieren wir nicht mit der zugrunde liegenden Container-Engine
- ▶ Wir könnten also irgendwann mal die Container-Technologie wechseln, ohne auf Ebene von K8s Änderungen vornehmen zu müssen
- ▶ Diese "Austauschbarkeit" wird u.A. durch die Einführung einer neuen Abstraktionsebene erreicht
- ▶ Die kleinste Einheit in K8s stellen **Pods** dar:
 - ▶ High-level-Perspektive auf container-basierte Anwendungen
 - ▶ **Pod:** eine Umgebung, in der ein Container oder eine "logische" Gruppe aus mehreren Containern ausgeführt wird
 - ▶ Intuition: zusätzliche "Isolationsschicht" um Container
 - ▶ Oft, aber nicht immer: Ein Container in einem Pod
 - ▶ Alle Container in einem Pod laufen auf der gleichen physischen Maschine
 - ▶ Pods sind "Wegwerf"-Objekte (ähnlich wie Container): bei neuen Konfigurationen werden Pods gelöscht und neue erstellt

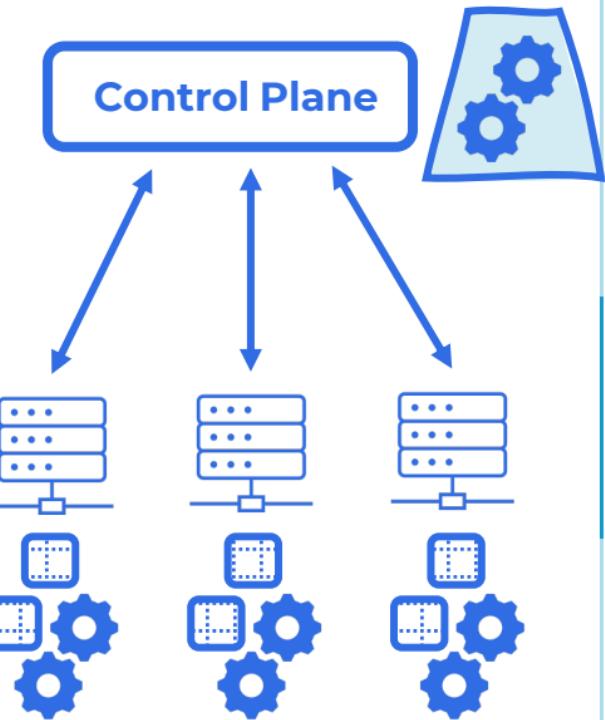


Docker Bootcamp

Wie ist ein Kubernetes-Netzwerk aufgebaut?

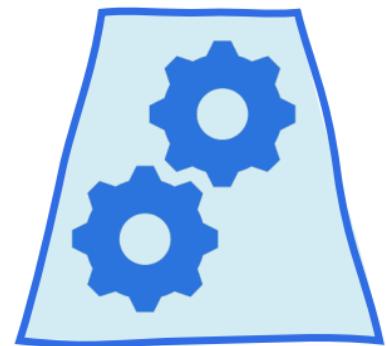
Was sind die Komponenten in einem K8s-Netzwerk?

- ▶ Ein K8s-Cluster besteht aus mindestens einem Node
- ▶ **Node**: physikalische oder virtuelle Maschine
 - ▶ Nodes, auf denen Pods ausgeführt werden, nennt man manchmal auch *worker nodes*
- ▶ In der Praxis laufen auf jedem Node in einem Cluster mehrere Pods
- ▶ Die Steuerung des Clusters erfolgt über die sogenannte **control plane** (Steuerungsebene), deren Komponenten prinzipiell auf jedem Nodes des Clusters ausgeführt werden können
- ▶ Cluster lassen sich beliebig erweitern durch neue Server, auf denen die entsprechenden Prozesse installiert sind



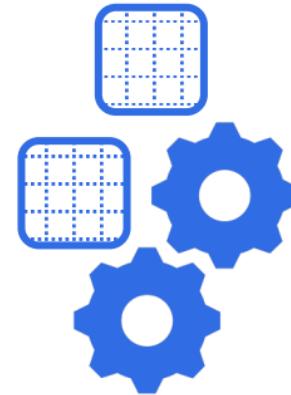
Was sind die Komponenten in einem K8s-Netzwerk?

- ▶ Die Steuerungsebene eines Clusters setzt sich aus folgenden Komponenten zusammen:
 - ▶ **kube-apiserver**: Frontend zur Kommunikation zwischen Nodes, stellt K8s-API zur Verfügung, nimmt *requests* von Client an
 - ▶ Einziger *entrypoint* zum Cluster (Sicherheit)
 - ▶ **etcd**: Speicher für Konfigurationsdateien, die Zustände der ausgeführten Anwendungen und Metadaten des Clusters
 - ▶ **kube-scheduler**: Verteilung von Pods auf Nodes entsprechend verfügbarer Ressourcen
 - ▶ **kube-controller-manager**: Überwachungsschleifen für die Nodes im Cluster, um auf Zustandsveränderungen reagieren (z.B. Ausfälle)
 - ▶ **cloud-controller-manager**: Cloud-Anbieter-spezifische Überwachungsschleifen
- ▶ Die Steuerungsebene kann auf mehrere Nodes aufgeteilt werden
 - ▶ => Ausfallsicherheit!

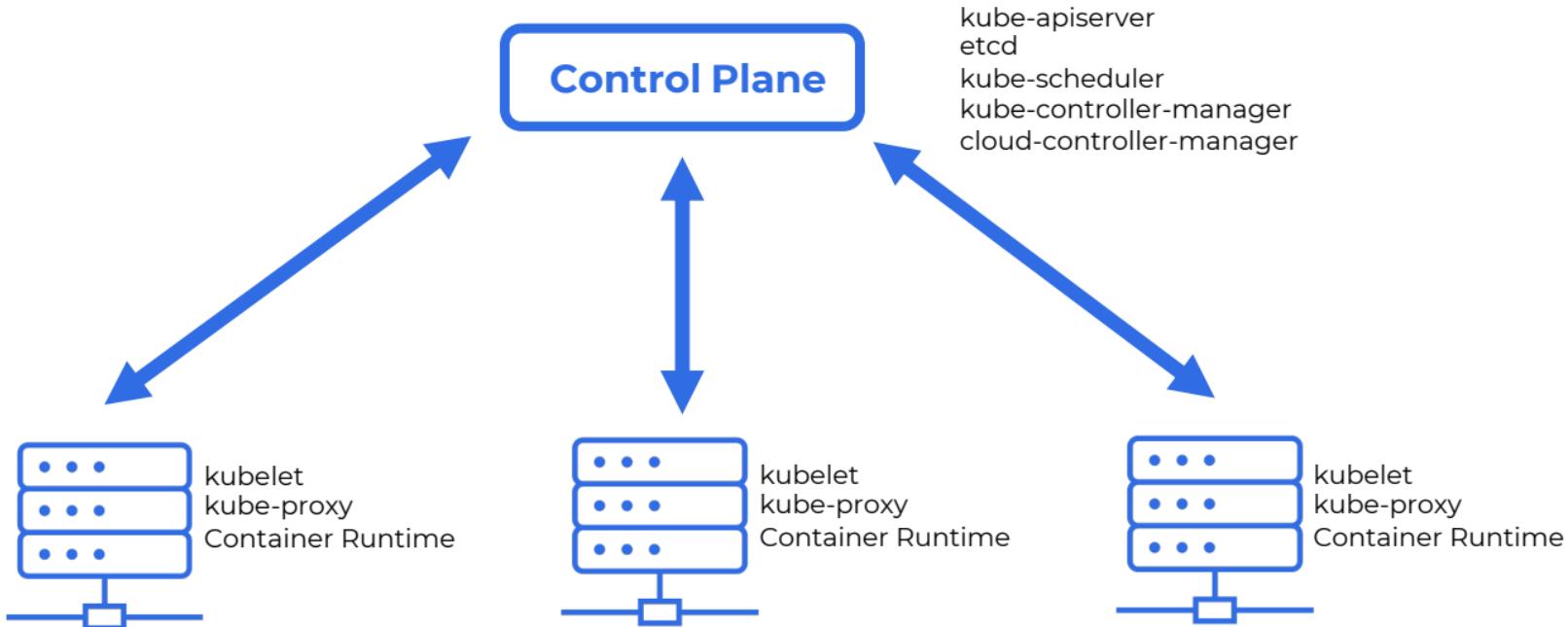


Was sind die Komponenten in einem K8s-Netzwerk?

- ▶ Auf jedem Node laufen immer diese Komponenten:
 - ▶ **kubelet**: Ausführung von Pods, in denen Container laufen
 - ▶ gemäß *request* von *kube-scheduler*
 - ▶ **kube-proxy**: Umsetzung von Netzwerkregeln und der Verbindungsweiterleitung zum Host; leitet *requests* zwischen Pods weiter und gewährleistet performante Kommunikation
 - ▶ **Container Runtime**: z.B. Docker
- ▶ Zusätzliche Addons (Wissen für Fortgeschrittene):
 - ▶ <https://kubernetes.io/docs/concepts/cluster-administration/addons/>



Cluster als Schaubild



Docker Bootcamp

Abgrenzung zu Docker Swarm

Was sind die Unterschied zwischen Docker Swarm und K8s?

- ▶ K8s ist komplexer und mächtiger
 - ▶ professioneller Standard für Container-Orchestrierung gemäß **Cloud Native Computing Foundation** (CNCF, gehört zur Linux-Foundation)
 - ▶ Umfangreicheres Ökosystem
 - ▶ Installation sehr umfangreich
 - ▶ K8s braucht aber eigentlich nicht lokal eingerichtet zu werden
 - ▶ Wird i.d.R. vom Cloud-Anbieter zur Verfügung gestellt (AWS, Digital Ocean, Google Cloud,...)
 - ▶ Kann lokal per Minikube ausgeführt werden -> gleich
 - ▶ **Trend:** Größere Community und institutioneller

Was sind die Unterschied zwischen Docker Swarm und K8s?

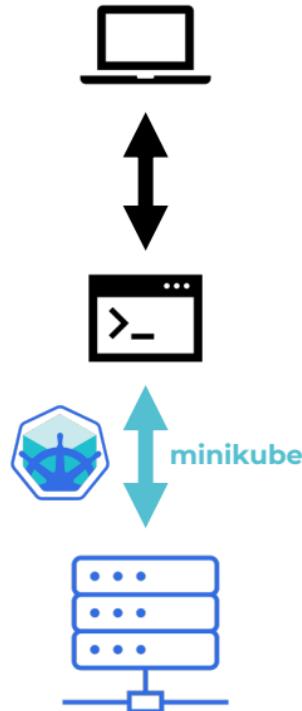
| | docker swarm | K8s |
|----------------|---|--|
| CLI | Integration von Docker CLI | kubectl |
| UI-Monitoring | zusätzliches Tool notwendig, z.B. Portainer | Dashboard integriert |
| Load Balancing | automatisch | Viele manuelle Konfigurationen möglich |
| Auto-Scaling | nur manuelle Skalierung | Ja |

Docker Bootcamp

Was ist Minikube?

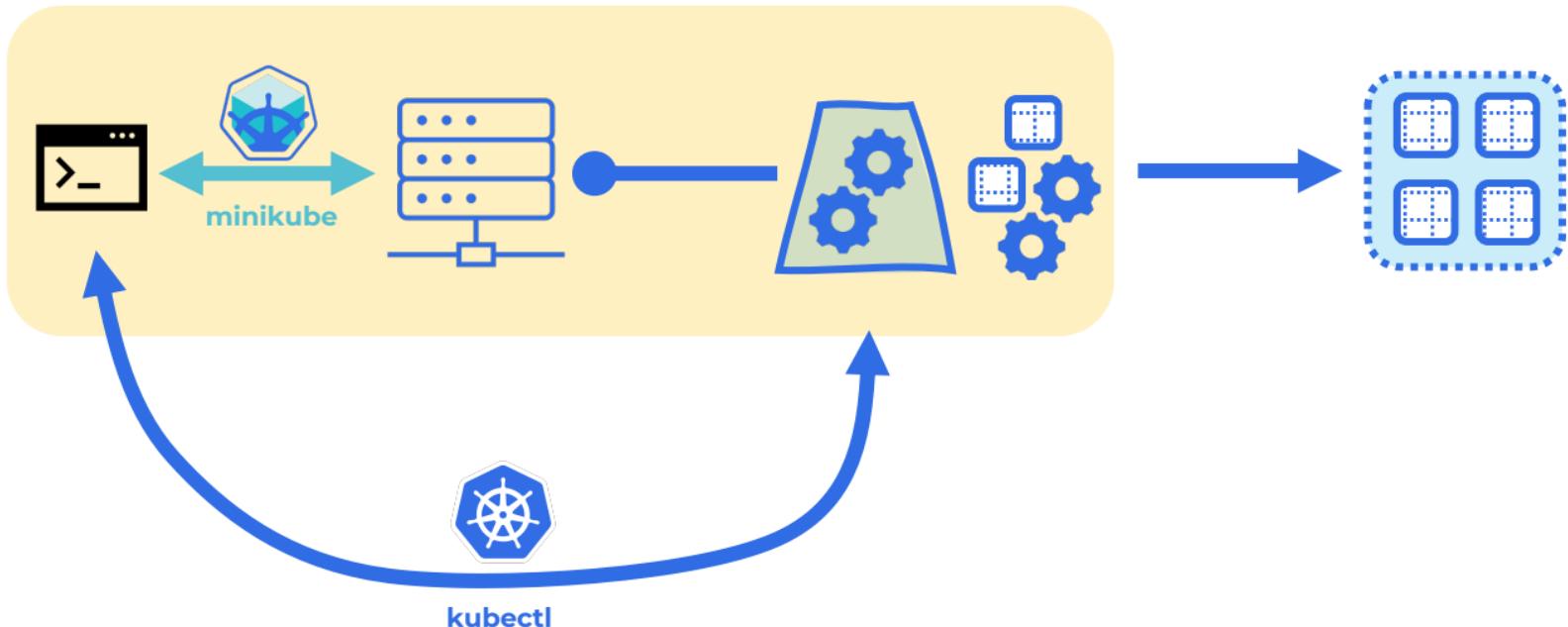
Was ist minikube?

- ▶ Problem:
 - ▶ Zum Entwickeln und Testen ist es aufwendig ein Cluster bestehend aus verschiedenen Nodes so wie im Produktivbetrieb einzurichten
- ▶ Lösung: **minikube**
 - ▶ CLI-Tool (Open Source)
 - ▶ Erzeugt eine virtuelle Umgebung, in dem ein K8s-Cluster bestehend aus einem Node läuft
 - ▶ Wichtig: wir werden minikube nur für die Einrichtung des Clusters verwenden!
 - ▶ Für die Verwaltung des Clusters brauchen wir das K8s CLI-Tool **kubectl**



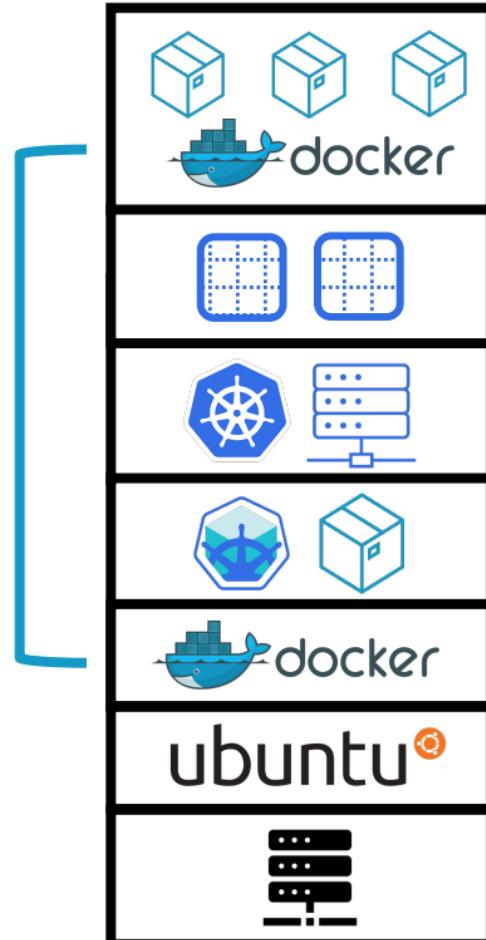
Unser lokales Setup

Dieser Abschnitt



Minikube installieren

- ▶ Wir werden Minikube als Docker-Container ausführen
 - ▶ Praktisch, da wir Docker bereits installiert haben
 - ▶ Weitere mögliche Treiber:
 - ▶ <https://minikube.sigs.k8s.io/docs/drivers/>
- ▶ Systemanforderungen beachten
 - ▶ <https://minikube.sigs.k8s.io/docs/start/>
- ▶ Windows: Installer verwenden
 - ▶ *minikube-installer.exe* herunterladen und ausführen
- ▶ MacOS: Homebrew Package Manager (<https://brew.sh>)
 - ▶ **brew install minikube**
- ▶ Nach der Installation führst du bei dir im Terminal
minikube start aus
 - ▶ Kann beim ersten Mal etwas länger dauern, da sehr viel heruntergeladen werden muss
 - ▶ **minikube** hat **kubectl** als dependency; wird hier automatisch installiert



Troubleshooting

- ▶ Bei **minikube start** kann es gerade unter Windows zu Problemen kommen
- ▶ Mögliche Lösungen:
 - ▶ Internetverbindung checken
 - ▶ Aktuelles Terminal schließen und neues Terminal öffnen
 - ▶ nacheinander die folgenden Befehle ausführen:
minikube config set driver docker
minikube delete
minikube start --driver=docker
- ▶ Nur im Notfall, wenn gar nichts anderes funktioniert:
 - ▶ Anderen Treiber verwenden
 - ▶ <https://minikube.sigs.k8s.io/docs/drivers/>
 - ▶ Einen der Online-Playgrounds nutzen (erfordern User-Accounts):
 - ▶ [Killercoda](#)
 - ▶ [Play with Kubernetes](#)

Die Minikube CLI

- ▶ Mit der minikube können wir ein lokales K8s-Cluster bestehend aus einem Node aufsetzen
 - ▶ Lokales K8s-Cluster starten
 - ▶ **minikube start**
 - ▶ Zustand der Prozesse auf der Steuerungsebene ("control plane") abfragen
 - ▶ **minikube status**
 - ▶ K8s pausieren
 - ▶ **minikube pause**
 - ▶ K8s weiter ausführen
 - ▶ **minikube unpause**
 - ▶ Lokales K8s-Cluster beenden
 - ▶ **minikube stop**
 - ▶ Lokales K8s-Cluster löschen
 - ▶ **minikube delete**
- ▶ Sobald das Cluster läuft, können wir es via **kubectl** konfigurieren

Docker Bootcamp

Das Kubernetes Dashboard

Das K8s-Dashboard

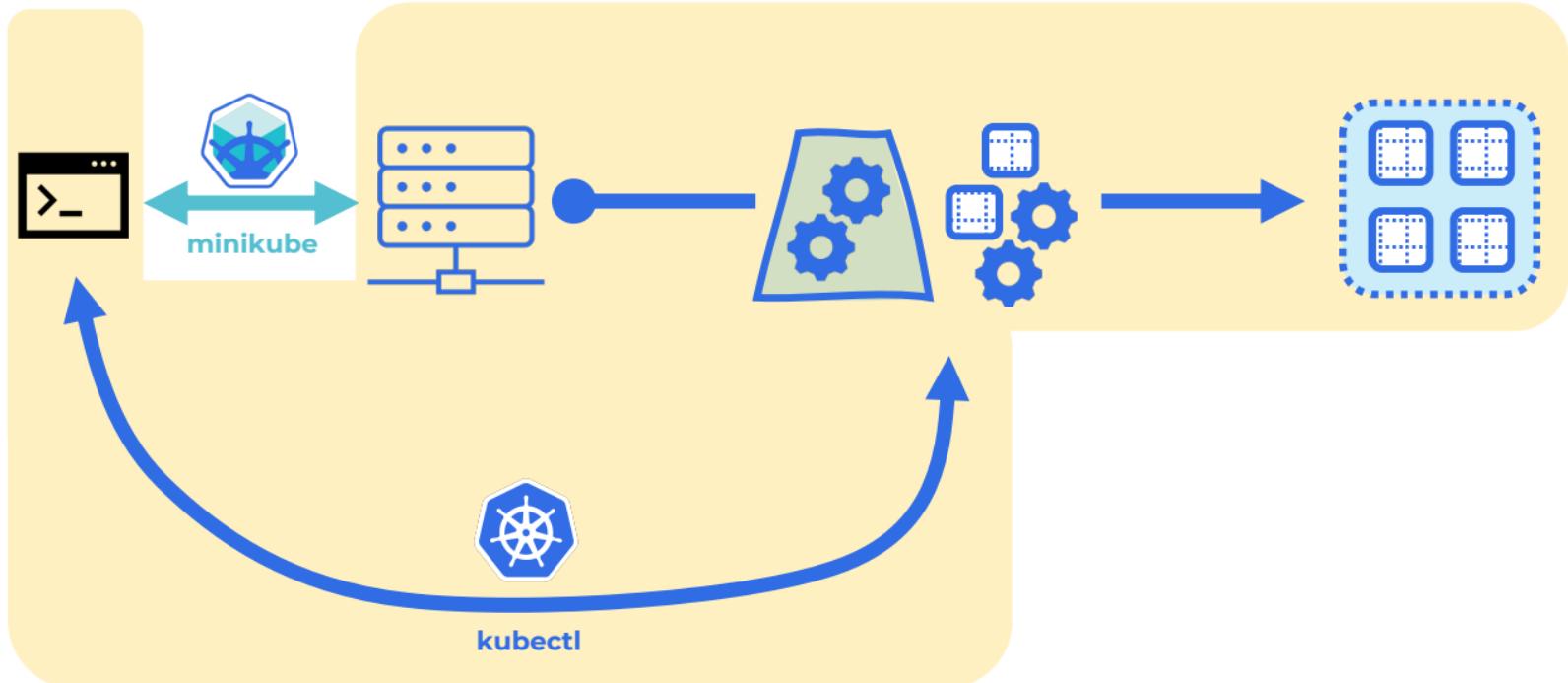
- ▶ **K8s Dashboard:** integriertes grafisches User-Interface
 - ▶ So ähnlich wie *Docker Desktop*, aber für K8s-Cluster
 - ▶ Bietet visuelle Übersicht über den Zustand deines Clusters
 - ▶ Ermöglicht Verwaltung des Clusters per Buttons
 - ▶ Entsprechende **kubectl**-Befehle werden dann eingeblendet
- ▶ Dashboard aufrufen:
 - ▶ **minikube dashboard**
 - ▶ Dashboard öffnet sich automatisch im Browser
 - ▶ Du musst das Terminal, in dem du das Kommando ausgeführt hast, geöffnet halten

Docker Bootcamp

Das Kommando: Kubectl

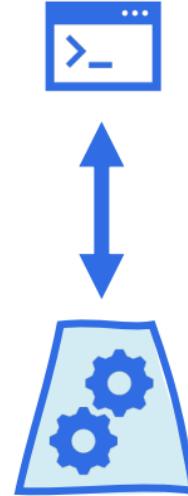
Unser lokales Setup

Dieser Abschnitt



Was ist **kubectl**?

- ▶ CLI-Tool für K8s-Cluster
 - ▶ Minikube-Cluster oder Cloud-Cluster
- ▶ Ermöglicht Kommunikation mit der Komponente *kube-apiserver*
- ▶ Wir werden **kubectl** nutzen, um mit dem Minikube-Cluster zu interagieren und es konfigurieren
- ▶ So ähnlich wie bei Docker kannst du das Cluster mit Befehlen in der Konsole verwalten oder komplexere Konfigurationen in Dateien (im yaml-Format) schreiben und dann anwenden



Erste Schritte mit kubectl

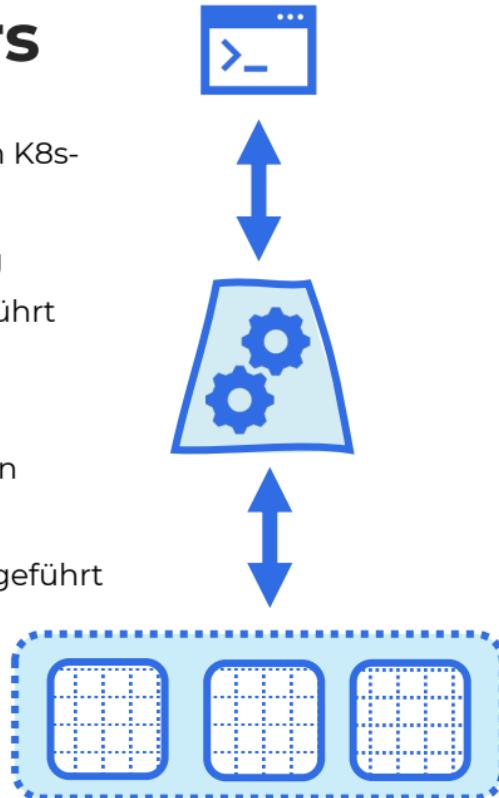
- ▶ **kubectl version --output=yaml**
 - ▶ Ohne die Flag gibt es eine deprecated-Meldung
 - ▶ Client und Server-Versionen
- ▶ Mit der Flag **--help** lässt du dir Infos zu einem **kubectl** – Kommando anzeigen
- ▶ Mit **kubectl get [K8s-Ressource]** gibst du Infos zu einer K8s-Ressource aus
- ▶ **kubectl get nodes**
 - ▶ Zustand von unserem Node abfragen (sollte Status: *ready* sein)
- ▶ **kubectl get pods**
 - ▶ Noch keine Pods vorhanden
- ▶ **kubectl get pods -A**
 - ▶ Alle Prozesse auf Steuerungsebene sollten laufen
- ▶ **kubectl get all**

Docker Bootcamp

Kubectl: Konfiguration im Cluster

Konfiguration des Clusters

- ▶ **Workload:** eine containerisierte Anwendung, die auf dem K8s-Cluster ausgeführt wird
- ▶ K8s stellt diverse verschiedene **Workloads** zur Verfügung
- ▶ Diese bestimmen, wie die Anwendung im Cluster ausgeführt und skaliert wird
- ▶ **Beispiel:**
 - ▶ ReplicaSet bzw. Deployment für stateless Anwendungen
 - ▶ StatefulSet für stateful Anwendungen
 - ▶ DaemonSet für Pods, die auf jeder Node im Cluster ausgeführt werden sollen
 - ▶ Job bzw. CronJob für Cronjobs



ReplicaSets und Deployments

- ▶ Hier in diesem Kurs:

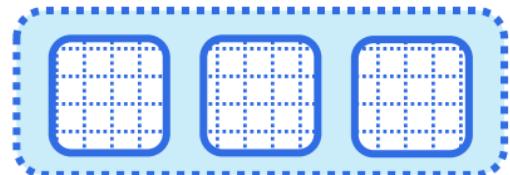
- ▶ Wir möchten eine stateless-Anwendung im Cluster ausführen
- ▶ Dafür haben wir 2 Möglichkeiten:
 - ▶ ReplicaSet
 - ▶ Deployment

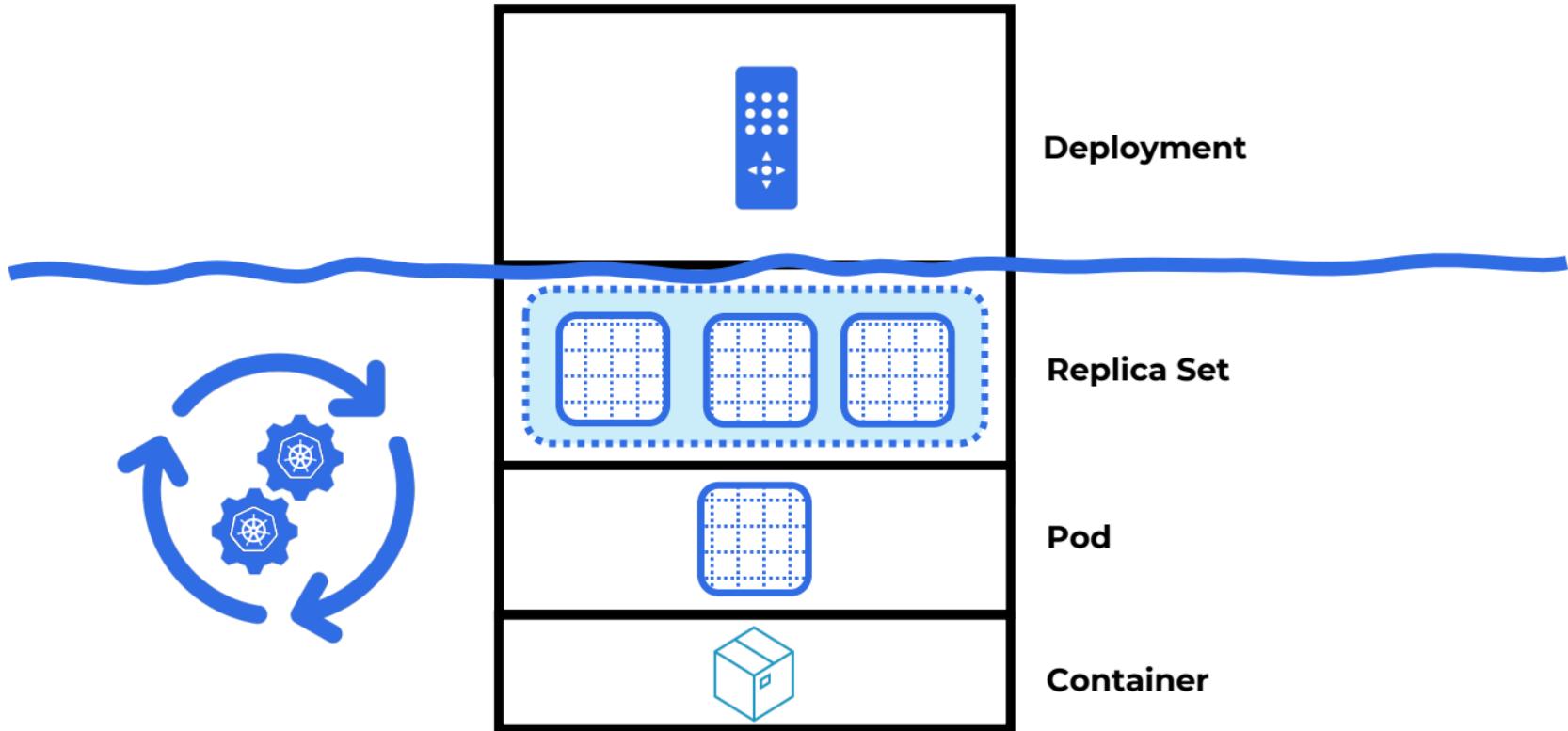
- ▶ **ReplicaSet:**

- ▶ Stellt sicher, dass eine bestimmte Anzahl an Kopien unseres Pods im Cluster ausgeführt wird

- ▶ **Deployment:**

- ▶ Verwaltet ein ReplicaSet, aber erlaubt einfachen, nachträglichen Austausch auf eine neuere Version





Beispiel: Deployment für nginx!

- ▶ Schauen wir uns jetzt mal an, wie wir einen nginx-Webserver deployen können
- ▶ Dafür können wir ein Deployment über kubectl anlegen:
 - ▶ **kubectl create deployment [Name deines Deployments] --image=[Docker-Image]**
 - ▶ Das Image muss hier auf Docker Hub abrufbar sein!
 - ▶ **kubectl create deployment nginx-deployment --image=nginx --replicas=3 --port=80**
 - ▶ Mit dem Parameter **--replicas** können wir die Anzahl der Replicas konfigurieren
 - ▶ Mit dem Parameter **--port** können wir dokumentieren, welcher Port vom Container geöffnet wird (ähnlich wie EXPOSE in Docker, nur Dokumentation)
 - ▶ Die Konfiguration können wir anschließend einsehen:
 - ▶ **kubectl edit deployment/nginx-deployment**



NGINX

Docker Bootcamp

Konfiguration als .yaml schreiben

Beispiel: Deployment für nginx!

- ▶ Wir können die Konfiguration auch selbst per .yaml schreiben
- ▶ Warum würden wir das tun wollen?
 - ▶ Wir können die Konfiguration für unsere Software z.B. in einem git-Repository speichern!
 - ▶ Dadurch können wir unseren Cloud-Setup reproduzierbar deployen
 - ▶ Und wir können einfach Updates durchführen
- ▶ Wir können eine solche .yaml-Datei deployen:
 - ▶ **kubectl apply -f deployment.yaml**



NGINX

Aufbau von K8s-Konfigurationsdateien

- ▶ Wichtigste Eigenschaften:
 - ▶ **apiVersion**: bezieht sich auf die Version der K8s-API
 - ▶ **kind**: Art des Objektes, z.B. Deployment
 - ▶ **metadata**: hier stehen der "name" und die "labels", unter denen das Objekt angesteuert werden kann
 - ▶ **spec** (specification): beschreibt den gewünschten Zielzustand für dieses Objekt; entsprechende Eigenschaften hängen von der Art der Objektes ab
 - ▶ **status**: beschreibt den aktuellen Zustand des Objektes im System

Beispiel: deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```



NGINX

Docker Bootcamp

Was genau machen die Labels?

Was machen die "labels"?

- ▶ Bei unserem Deployment haben wir metadata.labels angegeben
- ▶ Aber was machen die genau?
 - ▶ Damit können wir später das Deployment komfortabel finden
 - ▶ Wir können dann z.B. sagen:
 - ▶ **kubectl get deployments -l app=nginx**
 - ▶ Welche Labels wir vergeben, bleibt komplett uns überlassen
 - ▶ Wir dürfen auch mehrere Labels vergeben, und können dann nach mehreren filtern:
 - ▶ **kubectl get deployments -l app=nginx,version=1.0**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
    version: 1.0
  ...
  
```

Docker Bootcamp

Was genau machen die labels? (Teil 2)

Warum mehrere Labels?

- ▶ Unter `spec.template.metadata.labels` können wir Labels definieren, die dann auf den jeweiligen Pod angewendet werden
- ▶ `metadata.labels` sind also die Labels für das Deployment, `spec.template.metadata.labels` sind die Labels für die Pods die erstellt werden
- ▶ Natürlich können wir auch die Pods mit einem bestimmten Label auflisten:
 - ▶ **kubectl get pods -l app=nginx**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:latest
        ports:
          - containerPort: 80
```

Aber warum dann matchLabels?

- ▶ Wenn ein Deployment die zugehörigen Pods finden möchte, wird dazu spec.selector.matchLabels verwendet
- ▶ Diese matchLabels sind immutable!
- ▶ Dieser Selector muss alle Pods von spec.template.metadata.labels finden!
- ▶ In der Regel gibt man hier also 1:1 die gleichen Labels an
- ▶ Aber warum dann doppelt?
- ▶ **Grund:**
 - ▶ Wenn wir ein Update unserer Konfiguration herausbringen, können sich die Labels der Pods unter spec.template.metadata.labels geändert haben!
 - ▶ spec.selector kann so entwickelt sein, dass es immer noch die zukünftigen Pods findet
 - ▶ Übrigends: Statt spec.selector.matchLabels können auch komplexere Abfragen mit spec.selector.matchExpression angegeben werden (nicht Teil dieses Kurses)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:latest
    ports:
    - containerPort: 80
```

Docker Bootcamp

Mehrere Deployments in einer Datei

Mehrere Deployments

- ▶ Wir können in einer Datei mehrere Deployments spezifizieren
- ▶ Das kann nützlich sein, wenn wir mehrere Deployments dann gleichzeitig deployen möchten
- ▶ **Und:**
 - ▶ Später möchten wir oft ein Deployment und einen Service zusammen deployen
 - ▶ Service ist quasi der Netzwerk-Layer, der die Verbindung von außen zu den Pods erlaubt
 - ▶ Wir werden später dann Deployment + Service in eine Datei schreiben!

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  [...]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd-deployment
  labels:
    app: httpd
spec:
  [...]
```

Docker Bootcamp

Deployment debuggen

Deployment debuggen?

- ▶ Neuen Zustand des Clusters untersuchen:
 - ▶ **kubectl get deployment**
 - ▶ **kubectl get replicases**
 - ▶ **kubectl get pods**
 - ▶ erweiterte Ansicht
 - ▶ **kubectl get pods -o wide**
 - ▶ falls es mal länger dauert, bis ein Pod startet
 - ▶ **kubectl get pods --watch**
 - ▶ Logs anzeigen: **kubectl logs [Name vom Pod]**
 - ▶ Zusätzliche Infos anzeigen: **kubectl describe pod [Name vom Pod]**
 - ▶ Auf das Terminal in einem Pod zugreifen:
 - ▶ **kubectl exec -it [Name vom Pod] -- /bin/bash**
 - ▶ Mit **kubectl delete deployment [Name vom Deployment]** kannst du das deployment auch wieder löschen
 - ▶ **kubectl delete deployment nginx-deployment**

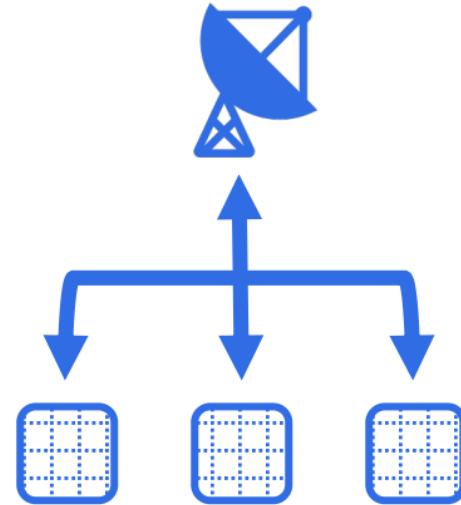
Docker Bootcamp

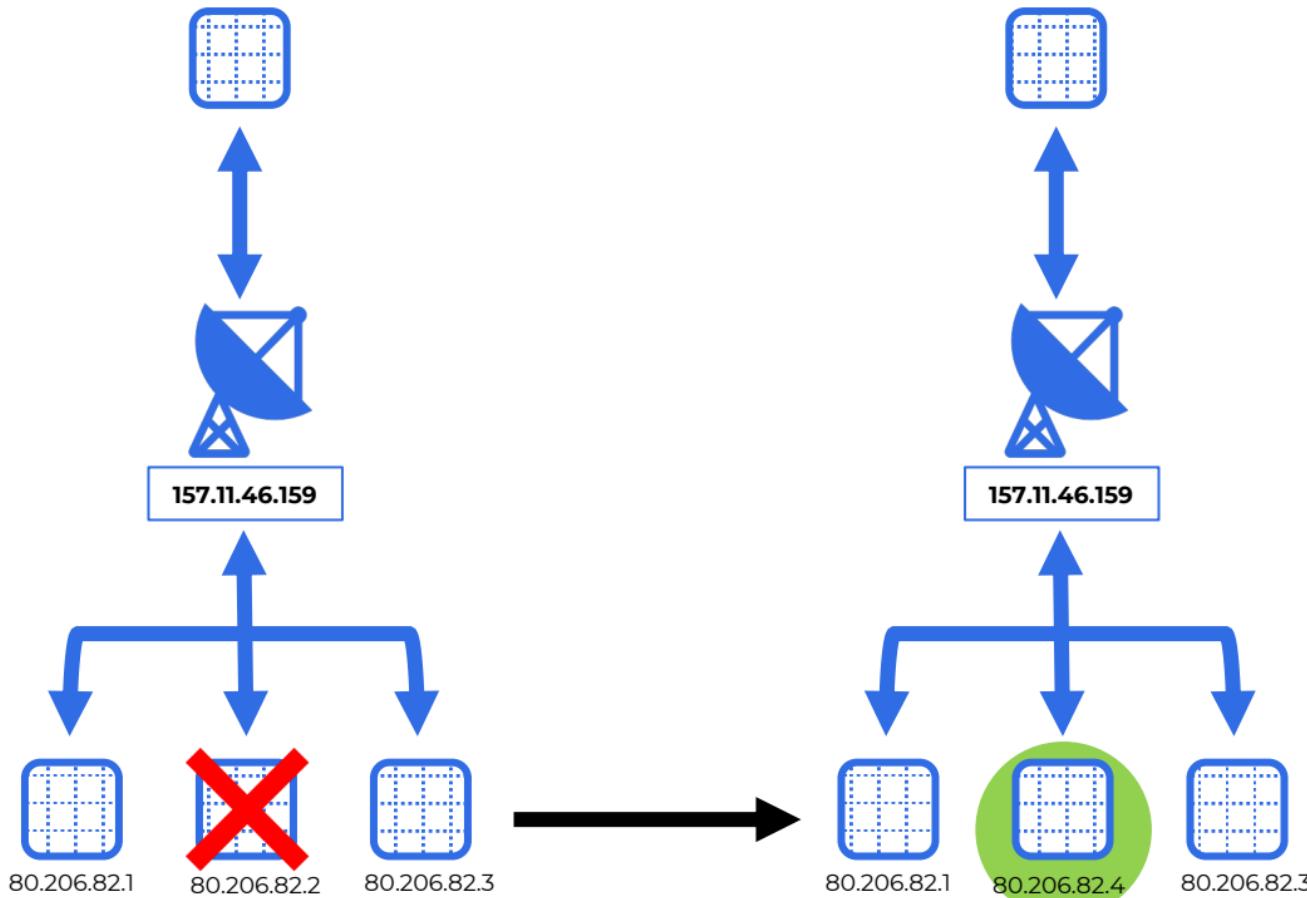
Auf unsere Deployments zugreifen: Services

Networking konfigurieren: Services

► Networking in Kubernetes:

- ▶ **Services**: Mechanismus, um Anwendungen in einem Pod in einem Netzwerk zugänglich zu machen
- ▶ Eine logische Gruppierung von Pods und einer Policy, wie darauf zugegriffen werden kann
- ▶ Intuition:
 - ▶ Ein Service gewährt für eine logische Gruppe von Pods eine permanente IP-Adresse
 - ▶ Dadurch kann diese Gruppe komfortabel angesteuert werden
- ▶ Beachte: jeder Pod hat eine eigene IP-Adresse und eine logische Gruppe von Pods trägt einen einzigen DNS-Namen
- ▶ <https://kubernetes.io/docs/concepts/services-networking/service/>





Arten von Services

- ▶ Es gibt verschiedene Service-Arten, u.a.
 - ▶ **ClusterIP:** Cluster-interne Verbindungen
 - ▶ Standard-Service
 - ▶ **NodePort:** Verbindung auf einem statischen Port auf unseren Nodes öffnen
 - ▶ Leitet intern auf eine ClusterIP weiter
 - ▶ i.d.R. im Port-Bereich 30.000-32767
 - ▶ **LoadBalancer:** Verbindung nach außen (unter Verwendung eines Load Balancers deines Cloud-Anbieters)

Docker Bootcamp

Einen ersten Service erstellen

Einen neuen Service erstellen

- ▶ Der schnellste Weg einen passenden Service aufzusetzen ist das Kommando
- ▶ **kubectl expose deployment [Name von deinem Deployment] --port=[PORT]**
 - ▶ Bei uns: **kubectl expose deployment nginx-deployment --port=80**
- ▶ Nun können wir im Browser auf nginx mittels minikube(!) zugreifen:
 - ▶ **minikube service nginx-deployment**
 - ▶ Am besten in einem neuen Terminal

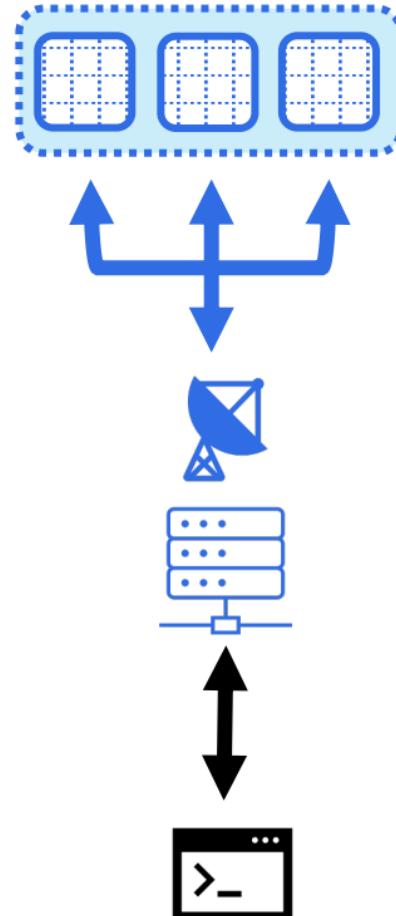


Docker Bootcamp

Services verwalten

Services verwalten

- ▶ Services einsehen:
 - ▶ Mit **kubectl get services** können wir alle Services einsehen
- ▶ Service löschen:
 - ▶ Mit **kubectl delete service [Name von deinem Service]** kannst du einen Service auch wieder löschen
- ▶ Ausführlicherer Befehl, um Service zu erstellen:
 - ▶ **kubectl create service [Art des Service] [Name von deinem Service] --tcp=[port]:[targetPort]**
 - ▶ In unserem nginx-Beispiel verwenden wir
 - ▶ **kubectl create service nodeport nginx-service --tcp=80:80**
 - ▶ Anschließend müssen wir dem Service sagen, wie die Pods gefunden werden:
 - ▶ **kubectl set selector service [Name] [key]=[value]**
 - ▶ **kubectl set selector service nginx-service app=nginx**



Docker Bootcamp

Services als .yaml

Services als .yaml

- Wir können Services auch in einer .yaml-Datei spezifizieren:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: nginx
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
```

Docker Bootcamp

Projekt: Webserver in Kubernetes

Projekt: Webserver in Kubernetes

- ▶ Wir möchten jetzt einen Webserver in Kubernetes laufen lassen
- ▶ Dieser soll unsere eigene Webseite ausliefern
- ▶ **Wie bekommen wir das hin?**
 - ▶ Schritt 1: Wir erstellen ein Image (per Dockerfile)
 - ▶ Schritt 2: Wir stellen dieses Image dem Docker in Kubernetes zur Verfügung
 - ▶ Schritt 3: Wir erstellen Deployment und Service



Docker Bootcamp

Projekt: Webserver in Kubernetes

Schritt 1: Image bauen und verifizieren (Dockerfile)

Docker Bootcamp

Projekt: Webserver in Kubernetes

Schritt 2: Image für Kubernetes zur Verfügung stellen

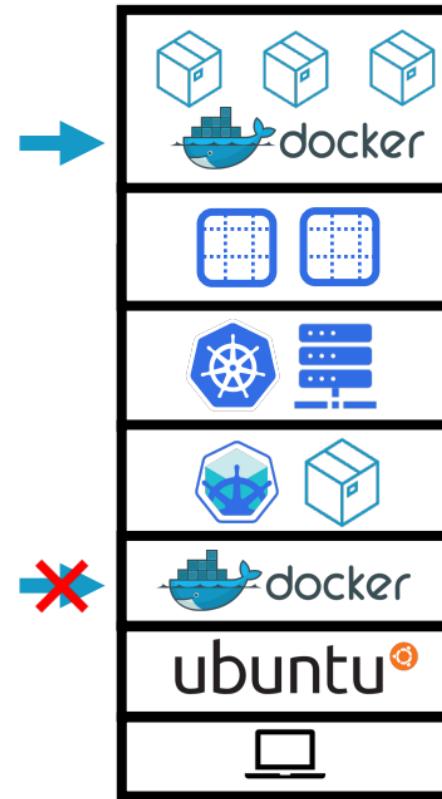
Eigene Images verwenden

▶ Problem:

- ▶ Wir können nicht ohne weiteres auf eigene Images zugreifen, die lokal bei uns gespeichert sind
- ▶ Denn unser Standard *Docker Daemon* läuft außerhalb des Clusters, beim Erstellen eines Pods wird aber der innere *Docker Daemon* verwendet, um das Image auszuführen

▶ Lösung:

- ▶ Wir konfigurieren die aktuelle Shell, dass sie nicht mehr den lokalen Docker Daemon nutzt, sondern zu dem von minikube verbindet
- ▶ **eval \$(minikube docker-env)**
- ▶ Oder: **eval \$(minikube -p minikube docker-env)**
- ▶ nur innerhalb der aktuellen Shell-Session gültig
- ▶ Dann baust du dein Image via **docker build -t [gewünschter Name von deinem Image]**.
- ▶ Du kannst anschließend (am besten in einer weiteren Shell) unter **minikube ssh** und **docker images** nachschauen, ob dein neues Image im Docker Daemon von minikube existiert



Eigene Images verwenden: In der Praxis

► In der Praxis:

- ▶ Wir möchten nicht auf allen Nodes vom Kubernetes-Cluster alle Images bauen wollen
- ▶ Cloud-Provider stellen daher i.d.R. ein Cloud-interne, privates Repository (quasi ein cloud-internes "DockerHub") zur Verfügung
- ▶ Die Nodes sind so eingerichtet, dass Images von dort heruntergeladen werden können
- ▶ Alle Nodes haben also Zugriff auf alle Images
- ▶ Ansonsten kann man sich auch bei DockerHub registrieren, und Images dort hochladen

Docker Bootcamp

Projekt: Webserver in Kubernetes

Schritt 3: Deployment & Service erstellen (als .yaml-Datei)

Docker Bootcamp

Crashkurs: DNS

Crashkurs: DNS

- ▶ **DNS?**
 - ▶ **Domain Name System**
- ▶ **Sorgt dafür, dass ein Domainname zu einer IP-Adresse aufgelöst wird**
- ▶ **Wenn ich also eine Webseite aufrufe (z.B. google.de):**
 - ▶ Dann wird der DNS-Server abgefragt, zu welcher IP sich mein Browser verbinden soll
 - ▶ Und mein Browser stellt dann an diese IP die entsprechende Anfrage

Docker Bootcamp

Networking innerhalb Kubernetes: DNS

DNS in Kubernetes

- ▶ **Das Ziel (generell):**
 - ▶ **Wir möchten, dass mehrere Pods untereinander kommunizieren können**
 - ▶ **Beispielsweise:**
 - ▶ **1x Frontend (z.B. Flask)**
 - ▶ **Kommuniziert zu einem Backend (z.B. einem Redis)**
- ▶ **Wie geht das?**
- ▶ **Ein Service:**
 - ▶ Ist steht innerhalb vom Kubernetes-Cluster unter seinem DNS-Namen zur Verfügung
 - ▶ Das ermöglicht, dass ein Pod auf einen Service (und damit auf andere Pods) zugreifen kann!
- ▶ **Das schauen wir uns am Besten am Beispiel an ☺**

Docker Bootcamp

Umgebungsvariablen

Umgebungsvariablen

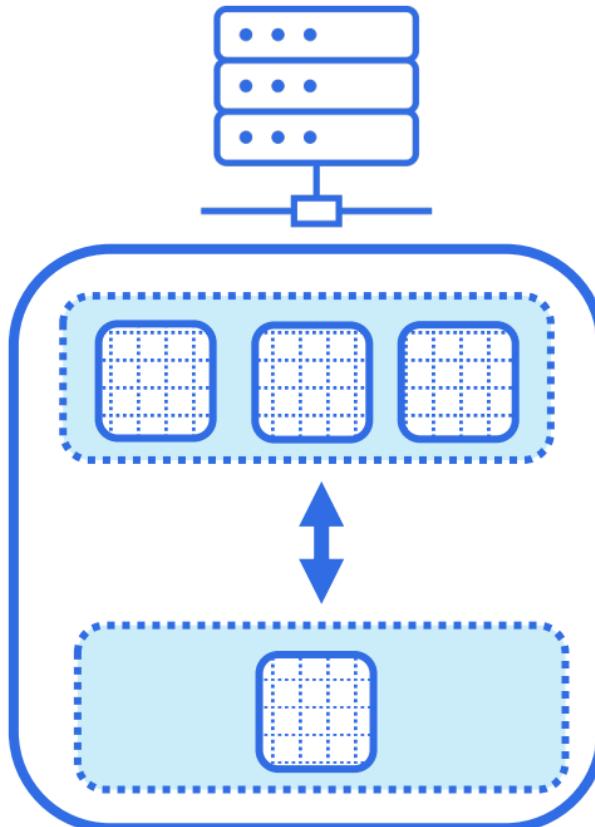
- ▶ Wir möchten Umgebungsvariablen für unsere Container definieren
- ▶ Quasi wie ein ENV aus dem Dockerfile, aber wir möchten es über die Kubernetes-Konfiguration steuern
- ▶ **Warum?**
 - ▶ Dinge, die sich ändern können wir dann in diese Umgebungsvariablen auslagern
 - ▶ Oder z.B. später den DNS-Namen von einem anderen Service aus Kubernetes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
      env:
        - name: ENV_VAR
          value: "Der Wert"
```

Docker Bootcamp

Projekt: Flask und Redis in Kubernetes

- Wir möchten eine Flask <-> Redis - Anwendung in Kubernetes deployen



Docker Bootcamp

Kubernetes: Weiterer Ausblick

Docker Bootcamp

Schlussworte

Schlussworte

- ▶ Du beherrschst jetzt Docker:
 - ▶ Docker CLI
 - ▶ Dockerfile
 - ▶ Docker Compose
 - ▶ Docker Swarm
 - ▶ Und Grundlagen von Kubernetes
- ▶ Damit kannst du jetzt Docker für deine Problemfälle einsetzen ☺

Vielen Dank!

- ▶ Vielen Dank, dass ich dir Docker (inkl. Kubernetes Exkurs) beibringen durfte
- ▶ Das bedeutet mir sehr viel
- ▶ Lass gerne eine ehrliche Bewertung da, darüber freue ich mich sehr!