Docker Container Grundlagen

- Konzepte der Container-Virtualisierung
- Eigenschaften von Containern
- Container vs. Virtuelle Maschinen (VM)
- Docker auf Linux und Windows
- Docker und DevOps

Konzepte der Container-Visualisierung

Container

Ein Container ist eine isolierte Laufzeitumgebung für einen oder mehrere Prozesse. Der Container bringt dabei alle Abhängigkeiten und Pakete mit, die für den laufenden Prozess notwendig sind.

Images

Container werden aus `Images` erzeugt. Ein Image ist dabei eine Vorlage für einen Container. Es können mehrere Container aus einem Image erzeugt werden.

Images sind in `Layers` geschichtet, wobei die Schichten Änderungen an dem vorherige Zustand darstellen und aufeinander aufgebaut, das fertige Image bilden. Diese Schritte können wir uns für ein Image visualisieren lassen:

```
$ docker image history nginx
IMAGE          CREATED        CREATED BY
SIZE      COMMENT
195245f0c792   2 months ago   CMD ["nginx" "-g" "daemon off;"]
0B        buildkit.dockerfile.v0
<missing>      2 months ago   STOPSIGNAL SIGQUIT
0B        buildkit.dockerfile.v0
<missing>      2 months ago   EXPOSE map[80/tcp:{}]
0B        buildkit.dockerfile.v0
<missing>      2 months ago   ENTRYPOINT ["/docker-entrypoint.sh"]
0B        buildkit.dockerfile.v0
<missing>      2 months ago   COPY 30-tune-worker-processes.sh /docker-ent…
4.62kB    buildkit.dockerfile.v0
<missing>      2 months ago   COPY 20-envsubst-on-templates.sh /docker-ent…
3.02kB    buildkit.dockerfile.v0
<missing>      2 months ago   COPY 15-local-resolvers.envsh /docker-entryp…
336B      buildkit.dockerfile.v0
<missing>      2 months ago   COPY 10-listen-on-ipv6-by-default.sh /docker…
2.12kB    buildkit.dockerfile.v0
<missing>      2 months ago   COPY docker-entrypoint.sh / # buildkit
1.62kB    buildkit.dockerfile.v0
<missing>      2 months ago   RUN /bin/sh -c set -x     && groupadd --syst…
95.9MB    buildkit.dockerfile.v0
<missing>      2 months ago   ENV DYNPKG_RELEASE=2~bookworm
0B        buildkit.dockerfile.v0
<missing>      2 months ago   ENV PKG_RELEASE=1~bookworm
0B        buildkit.dockerfile.v0
<missing>      2 months ago   ENV NJS_RELEASE=1~bookworm
0B        buildkit.dockerfile.v0
<missing>      2 months ago   ENV NJS_VERSION=0.8.5
0B        buildkit.dockerfile.v0
<missing>      2 months ago   ENV NGINX_VERSION=1.27.1
0B        buildkit.dockerfile.v0
<missing>      2 months ago   LABEL maintainer=NGINX Docker Maintainers <d…
0B        buildkit.dockerfile.v0
<missing>      2 months ago   /bin/sh -c #(nop)  CMD ["bash"]
0B
<missing>      2 months ago   /bin/sh -c #(nop) ADD file:06a1877f1e100122a…
97.1MB
```

Es gibt auf der einen Seite fertige Anwendungsimages, in denen die entsprechende Anwendung schon fertig vorbereitet ist. Zusätzlich existieren Basis-Images für die meisten Programmiersprachen, Technologien und Betriebssysteme, aus denen eigene Images erstellt werden können.

Die Images werden in `Registries` gespeichert und vorgehalten. Öffentliche Registries sind beispielsweise:

- hub.docker.com
- gitlab.com

Zusätzlich gibt es viele Tools und open Source Anwendungen, mit denen eigene, private Registries betrieben werden können. Eine Zugriffskontrolle mittels Benutzerauthentifizierung ist ebenfalls in vielen Fällen möglich.

Tags

Docker images mit gleichem Namen können sich noch mit Tags unterscheiden. Damit werden oft verschiedene Versionen der Inhalte abgebildet, was der implizite Standard-Tag `latest` suggeriert. Es ist jedoch auch Möglich, unterschiedliche varianten des Images zu beschreiben, wie zum Beispiel unterschiedliche Basis-Betriebssysteme:

- `rust:1.82.0-bookworm`
- `rust:1.82.0-bullseye`
- `python:3.12.7-bullseye`
- `python:3.12.7-alpine`

Tags werden mit einem `:` vom Namen des Images getrennt.

Networks

Docker kann private Netzwerke zwischen containern verwalten, sodass die Kommunikation zwischen den Containern nicht nach außen dringen kann. Es gibt verschiedene Netzwerkmodi, die wir uns später genauer anschauen werden.

Volumes

Container sind volatil, während der Laufzeit gespeicherte Daten gehen bei einem Neustart des Containers verloren. Wenn persistente Daten abgelegt werden sollen, müssen dafür Volumes definiert werden. Auch hier gibt es verschiedene Arten von Volumes mit eigenen Vor- und Nachteilen, die wir uns später im Detail ansehen.

Logs

Docker sammelt den Stdout der Container in einer von Docker verwalteten Logdatei, die bei

neustart des Containers überschrieben wird. Für langfristiges Logging ist daher ein eigener Ansatz notwendig.

Struktur

Container Engine Docker

Docker wie wir es benutzen ist ein Frontend für andere Komponenten, die von der Docker Installation mitgebracht werden.

Daemon Containerd

Containerd ist der Daemon die von Docker als Open Soure veröffentlicht wurde. Docker ist ein komfortables Frontend für Containerd, das leichter zu benutzen ist als Containerd. Jedoch lassen sich Container auch mit Containerd direkt verwalten.

Containerd ist ein Linux Tool, das spezielle Fähigkeiten des Linux Kernels nutzt um die Prozesse in Containern voneinander zu isolieren. Unter Macos und Windows wird Docker Desktop genutzt, was im Hintergrund eine Virutelle Linux Maschine mit Docker installiert und startet, damit Docker auch auf diesen Systemen genutzt weren kann.

Dabei gibt es kleine Unterschiede zu dem Verhalten unter Linux. Beispielsweise kann ein Linux User nicht in ein Volume Mount schreiben, der im Container `root` gehört. Ein Unser unter MacOs kann dies jedoch ohne Probleme tun.

Runtime runC

Docker nutzt `runC` als low-level Runtime um Container zu verwalten. Es gibt auch andere Runtimes, die mit Docker genutzt werden können, falls sie sich an den OCI Standard halten.

Andere Containerlösungen

Neben Docker gibt es noch weitere Projekte, die auf Containerd aufsetzen um die Container zu verwalten. Eine beliebtes Projekt ist `Podman` , die sich durch den verzicht auf eien Background Deamon auszeichnet und keine Rootrechte benötigt. Das macht Podman

besonders für den Betrieb von Containeranwendungen interessant.

Eigenschaften von Containern

Container sind grundsätzlich voneinander unabhängig, jedoch werden aus praktischen Gründen oft Abhängigkeiten zwische zwei Containern hergestellt, die unterschiedliche Teile eine Anwendung darstellen, wie beispielsweise den Anwendungsserver und die Datenbank.

Container haben ein standardisiertes Format und verhalten sich auf unterschiedlichen Systemen stets gleich.

Oft läuft nur ein Prozess in einem Container, eine berühmte Ausnahme dazu ist der Gitlab Container. Dort finden wir eine Datenbank, die Gitlab Anwendung, Sidekiq, Grafana, eine Image Registry und vieles mehr. Der Grund dafür ist dass Gitlab so umfangreich ist, dass das Aufsetzen einer eigenen Instanz zu kompliziert wäre, hätten die Entwickler ein einzelnes Image für jeden Prozess bereitgestellt.

Container vs. virtuelle Maschine

Eine virtuelle Maschine besteht aus einem kompletten Betriebssystem mit Kernel, Hardwaredrivern, Programmen und Anwendungen. Virtuelle Maschinen benötigen in der Regel eine signifikante Zeit um zu starten. Für das Betriebssystem wird in der Regel signifikanter Arbeitsspeicher und Festplattenplatz benötigt.

Für das Einfassen einer einzelnen Anwendung sind virutelle Maschinen oft zu groß.

Container hingegen bilden nur das nötigste ab, um einen Prozess laufen zu lassen. Mehrere Container teilen sich den Kernel des Host Betriebssystems. Mit Containern lassen sich daher mehr Anwendungen auf der gleichen Hardware betreiben, als es mit virtuellen Maschinen der Fall wäre.

Docker Einführung

- Architektur und Konzepte verstehen
- Installation und erste Schritte
- Docker konfigurieren

Docker installieren

Docker Desktop für Mac, Windows und Linux

Unter MacOs und Windows wird docker als Docker Desktop installiert, da diese Plattformen von Docker nativ nicht unterstützt werden. Docker Desktop startet im Hintergrund eine virutelle Linux Maschine, in der die Container nativ laufen.

Für Linux ist Docker Desktop ebenfalls verfügbar und startet ebenfalls eine virutelle Maschine im Hintergrund. Die Installation von Docker Desktop und der Docker Eingine auf einem Linux System ist möglich, allerdings ist in diesem Fall eine erweiterte Konfiguration notwendig, damit sich beide Instanzen nicht gegenseitig beeinflussen. Es wird empfohlen, bei der Verwendung von Docker Desktop die Docker Engine zu beenden.

Docker Engine für Linux

Docker Engine verwaltet Linux Container direkt auf dem Betriebssystem und ist nur für Linux verfügbar. Die Installation erfolgt in der Regel direkt aus den Archiven der verwendeten Linux distribution. Für `debian` sind einige Schritte notwendig, um das `apt`-Repository in die Paketverwaltung des Systems einzubinden. https://docs.docker.com/engine/install/debian/#install-using-the-repository

Anschließend müssen die folgenden Pakete installiert werden:

```
apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

Der Docker Daemon läuft als Rootprozess. Damit Docker ohne die Eingabe von `sudo` bedienbar ist, sollte, falls nicht schon geschehen, eine Gruppe namens `docker` angelegt werden und den entsprechenden Usern zugeordnet werden. Dann ist der Zugriff auch ohne Rootrechte möglich.

> ‼️ Da Docker mit Rootrechten betrieben wird, erhält jeder Benutzer der Docker bedienen darf die Möglichkeit, sich Rootrechte auf dem System zu verschaffen.

Ist die Docker Engine installiert kann mit diesem Befehl überprüft werden, ob alles

funktioniert:

```
docker run hello-world
```

Docker konfigurieren

Garbage Collection

Docker fährt einen sehr konservativen Ansatz im Aufräumen von Daten. Daher ist es notwendig, selber Hand anzulegen. Besonders die heruntergeladenen Docker Images und die abgeschalteten Containerinstanzen sowie deren `Volumes` werden nicht automatisch gelöscht. Dies kann durch das `prune` Kommando manuell ausgelöst werden.

Images:

```
docker image prune
docker image prune -a
```

Container:

```
docker container prune
```

Volumes:

```
docker volume prune
```

Network:

```
docker network prune
```

Alles:

```
docker system prune --volumes
docker system prune --volumes -a
```

Beim *pruning* werden keine Ressourcen gelöscht, die gerade in Benutzung sind. Daher kann auf Servern, die eine Produktionsumgebung aus Docker Containern betreiben, der System Prune-Befehl beispielsweise via Cron jede Nacht durchgeführt werden. Je nach containerisierter Technoligie sind Images sehr groß (Python z.B. 1GB), daher kann bei ausbleibender Garbage Collection ein Server schnell voll laufen.

Aufgaben

Docker installieren

Installiere Docker (Engine) für die Linux Distribution. https://docs.docker.com/engine/install/debian/

Lasse den Helloworld Container laufen.

System aufräumen

Wie viel Speicherplatz verbraucht das Image?

Lösche das Image und den Container wieder vom System.

Images und Container

- Docker-Registries einrichten
- Docker-Images erstellen, verwalten und verteilen
- Mit Dockerfiles Images erstellen und anpassen
- Steuerung von Containern und Befehle

Registries

Viele Docker Images werden über öffentliche, zentrale Registries wie github, gitlab, Red Hat Container Registry, Codeberg Container Library und viele mehr bereitgestellt.

Zusätzlich gibt es viele open und closed Source Container Registry Anwendungen, die auf eigener Infrastruktur in privaten Netzwerken betrieben werden können. Ein Beispiel dafür ist die **Distribution Registry** https://hub.docker.com/_/registry. Manche Versionsverwaltungsserver bieten auch Container Registries als Teil ihres Leistungsumfangs an, wie beispielswesie **Gitlab Community Edition** und **Forgejo**.

Standardmäßig geht Docker davon aus, mit der eigenen Registry unter hub.docker.com zu kommunizieren.

```
docker pull myproject/myimage:latest
```

Nicht-Standard Registry

Im Namen des Images kann jedoch auch eine Domain angegben werden, um abweichende Regsitries anzugeben:

```
docker pull myregistry.mycompany.com/myproject/myimage:latest
```

Eine Änderung der Standard Registry auf eine andere als hub.docker.com ist nicht vorgesehen, da dies dazu führen würde, dass die Image Bezeichnungen auf unterschiedlichen Servern auf unterschiedliche Images auflösen würden. Somit muss eine Abweichung immer explizit sein. Eine Ausnahme zu der Regel ist die Red Hat Variante von Docker, die abweichende Standard-Registries erlaubt. Das Problem der nicht-Eindeutigkeit besteht dann jedoch weiterhin.

Authentifizierung

Manche Registries oder Images erzwingen eine Authentifizierung. Ein Computer kann mittels folgendem Befehl authentifiziert werden:

```
docker login registry.mycompany.com -u myusername
```

Images erstellen

In der Regel werden für die eigenen Anwendungsfälle eigene Images erstellt. Dazu wird oft auf ein bestehendes Image aufgebaut. Als Startpunkte eignen sich Betriebssystem-Images wie `debian` oder `ubuntu`, oder je nach verwendeter Technoligie auch Images mit vorinstallierten Abhängigkeiten, zum Beispiel `python:3.12`, `php:8.3.12` oder `rust:1.82.0`.

Je nach Anforderung können sich auch Docker Images mit installierten Anwendungen eignen, wie beispielsweise `nginx`, `redis` oder `postgres`.

Sollen möglichst kleine Images erstellt werden gibt es mit `scratch` eine Möglichkeit, von einem leeren Image aus zu starten. Dort lassen sich allerdings nur kompilierte, direkt unter Linux ausführbare Dateien als Anwendung hinterlegen, da eine Runtime nicht vorhanden ist.

Dockerfile und Images bauen

Die Definition eines Images ist ein `Dockerfile`. Dort ist spezifiziert, welche Schritte unternommen werden müssen, um das Image zu erstellen. Gehen wir davon aus, dass eine ausführbare Datei namens `hello` im aktuellen Verzeichnis liegt, können wir sie so in ein Image einbetten:

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

Diese Zeilen stehen in einer Datei namens `Dockerfile`. Um das Image zu erstellen nutzen wir im Ordner des Dockerfiles folgenden Befehl:

```
docker build -t registry.mycompany.com/myproject/hello:v0.0.1 .
```

Wichtig ist hier der `.` am Ende des Befehls, da beim bauen des Images ein Pfad angegeben werden muss. Mit `-t` wird ein Imagenamen angegeben, mit dem wir das Image später referenzieren können. Geben wir keinen Namen an wird Docker einen mehr oder weniger lustigen Namen für uns auswählen.

Im Dockerfile geben wir die Schritte vor, um das Image nach unseren Wünschen aufzubauen. Jeder Schritt entspricht dabei einem `Layer` im fertigen Image.

1. `FROM scratch` Hier geben wir das Image an, auf das wir aufsetzen wollen.

2. `COPY hello /` Wir kopieren die Datei `hello` in das Dateisystem des Images
3. `CMD ["/hello"]` Wir definieren den Standardbefehl, der im Container ausgeführt wird, sobald er startet.

Ein komplexeres Dockerfile ergibt sich beispielsweise in Python Projekten:

```
FROM python:3.12

ADD . /usr/src/app
WORKDIR /usr/src/app
COPY requirements.txt ./

RUN mkdir /usr/src/app/static

VOLUME /usr/src/app/static

RUN pip install --no-cache-dir -r requirements.txt

RUN ln -sf /dev/stdout /var/log/application.log

RUN chown -R 1000:1000 /usr/src/app
USER 1000
CMD  bash -c "python3 manage.py runserver &> /var/log/application.log"
```

Neben den bereits diskutierten Punkten definieren wir hier Docker Volumes, ändern den Zustand des Images mit einigen Konsolenbefehlen, in denen wir beispielsweise die Abhängigkeiten der Anwendung mit `pip` installieren, setzen Symlinks, sodass die Logfiles unserer Anwendung an `stdout` geschickt werden und somit für Docker sichtbar sind. Zuletzt starten wir unsere Anwendung mit dem Standardkommando.

Images und Registries

Damit wir unser Image zwischen mehreren Servern komfortabel transferieren können laden wir es in eine Registry hoch. Das geschieht mit dem folgenden Kommando:

```
docker push registry.mycompany.com/myproject/hello:v0.0.1
```

Hier müssen wir wieder den Namen unseres Containers nennen, wobei wir auch direkt angeben, in welche Registry wir ihn hochladen möchten.

Images Verwalten

```
docker images
docker pull nginx
docker inspect nginx
docker image rm nginx
```

Container erstellen

Aus Images können wir laufende Container erstellen. Dazu nutzen wir einen der folgenden Befehle:

Startet einen Nginx Container im Hintergrund (`-d`), definiert den Namen als `mywebserver` und verbindet den internen Port 80 auf den Host Port 8080.

```
docker run -p 8080:80 -d --name mywebserver nginx
```

Zeigt die erstellten Container auf dem Host an.

```
docker ps
```

Einen Befehl im Container ausführen:

```
docker exec mywebserver ls
```

Eine Shell im Container öffnen:

```
docker exec -it mywebserver bash
```

Aufgaben

Statische Website

Erstelle eine `index.html` Datei mit folgendem Inhalt:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>HTML5 Boilerplate</title>
    <link rel="stylesheet" href="styles.css" />
  </head>

  <body>
    <h1>Page Title</h1>
    <script src="script.js"></script>
  </body>
</html>
```

und `styles.css`:

```
body {
  background: aliceblue;
}
```

und `script.js`:

```
console.log("The script works");
```

Schreibe dann ein Dockerfile das ein Image produziert, dass die Website beinhaltet und auf Port 80 ohne SSL ausliefert. Dazu kann `nginx` mit folgender Konfiguration als basis genutzt werden:

```
server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

Diese Konfiguration muss im Nginx Container unter `/etc/nginx/conf.d/default.conf` abgelegt werden.

Starte einen Container aus dem Image und verbinde den Host Port 8080 (oder einen anderen) auf 80 des Containers.

Verifiziere:

- Das Image wird fehlerfrei gebaut
- Der Container startet
- Unter Port 8080 ist die Website sichtbar
- Das CSS wird geladen (der Hintergrund ist hellblau)
- Das Script wird geladen (in der Entwicklerkonsole im Browser steht "The script works")
- Öffne eine Shell im Container und verifiziere mit curl, dass der Nginx Server auf Port 80 läuft.

Docker Networking

- Netzwerkoptionen in Docker
- Netzwerke erstellen und verwalten

Netzwerkoptionen

In Docker gibt es mehrer Driver für Netzwerke, die sich für unterschiedlche Einsatzzwecke eignen:

Driver	Beschreibung
bridge	Der Standarddriver. Er wird verwendet, falls kein Driver angegeben wird
host	Die Container sind direkt im Host Netzwerk eingebunden
none	Der Container ist von der Außenwelt komplett abgeschnitten
overlay	Hiermit lassen sich mehrere Docker Daemons verbinden
ipvlan	Ermöglichen volle Kontrolle über IPv4 und IPv6 Adressvergabe
macvlan	Vergibt MAC Adressen an die container

bridge

Üblicherweise wird der `bridge` Driver benutzt, wenn mehrere Container untereinander auf dem gleichen Host kommunizieren müssen. Die Container sind von allen anderen Containern auf dem Host isoliert. Docker setzt automatisch Firewallregeln auf dem Host um diese Kommunikationseinschränkungen sicherzustellen.

> 💡Das bedeutet, docker kann mit der externen Konfiguration von `ufw` interferieren. Es ist also besondere vorsicht geboten, wenn UFW mit Docker gemeinsam auf einem System betrieben werden soll.

Wenn der Docker Daemon startet wird automatisch ein bridge Netzwerk namens `bridge` angelegt, in das alle Container eingebunden werden, außer wir spezifizieren etwas anders. In diesem Netzwerk können sich die Container untereinander nur mit ihrer IP Adresse erreichen. Definieren wir ein eigenes Netzwerk sind die Container auch mit ihrem Namen über DNS auflösbar.

host

Ein Docker Container in einem `host` Netzwerk verhält sich so, als ob die Anwendung direkt

auf dem Host laufen würde. Üblicherweise findet dieser Netzwerkdriver in leistungskritischen Szenarien Anwendung, oder wenn ein Container sehr viele Ports öffnen soll.

Ein praktischer Anwendungsfall ist der Betrieb eines DHCP Servers in einem Docker container. Dieser muss aus den Paketen Informationen auslesen, die in `bridge` Netzwerken nicht mehr verfügbar sind.

Für die meisten Anwendungen ist das `bridge` Netzwerk jedoch ausreichend.

overlay

`overlay` Netzwerke erlauben die Kommunikation zwischen verschiedenen Docker Daemon Hostsystemem. In der Regel werden sie genutzt, um verschiedene Systeme in einen Docker **swarm** einzubinden, sie können jedoch auch dazu benutzt werden, manuell Container auf verschiedenen Hostsysteme zu verbinden.

Damit zwei Hosts in einem overlay Netzwerk verbunden werden können, müssen sie untereinander über die Ports 2377/tcp, 4789/udp, 7946/tcp und 7946/udp erreichbar sein.

Aufgrund von Grenzen in den Fähikgeiten des Linux Kernels werden Netzwerke ab etwa 1000 Containern auf einem Host instabil.

Die Kommunikation zwischen den Docker Daemon Hosts kann verschlüsselt erfolgen, allerdings geht dies mit teils signifikanten Leistungseinbußen daher, sodass im Einzelfall überprüft werden muss, ob die geplante Applikation unter Last auch mit Verschlüsselung noch wie gewüsncht funktioniert.

ipvlan

Netzwerke in diesem Modus geben den Anwendern volle Kontrolle über die Kommunikation mittels IPv4 und IPv6 Adressen und eignen sich so für Anwendungen mit speziellen Netzwerkanforderungen, die über `bridge` und `overlay` Netzwerke hinausgehen.

Durch das Vermeiden der Linux Bridge ist ein ipvlan Netzwerk besonders performant gegenüber den anderen Netzwerktypen. Besonders für Services die extern erreichbar sein sollen, eignet sich dieser Netzwerktyp.

macvlan

Einige Anwendungen erwarten, dass sie eine direkte Netzwerkverbindung erhalten. Dies ist oft bei alten Anwendungen der Fall. In diesem Fall kann über ein `macvlan` Netzwerk einzelen Containern eine MAC Adresse zugewiesen werden, die sich wie ein physischer Netzwerkanschluss verhält. Jedoch muss diesem Container dann auch ein physischer Port des Hostsystems zugewiesen werden. Zusätzlich hat dieser Netzwerkmodus noch einige Anforderungen an das Netzwerk, in dem der Docker Host betrieben wird. Generell gilt daher, dass die Verwendung eines `bridge` Netzwerkes, falls möglich, langfristig oft die bessere Lösung ist.

Netzwerke Erstellen

Netzwerke können so erstellt werden:

```
docker network create --driver bridge mynetwork
```

wobei, statt `bridge` auch ein anderer Netzwerktreiber genutzt werden kann.

Mit

```
docker network ls
```

können die vorhandenen Netzwerke angezeigt werden,

```
docker network inspect
```

zeigt Details zu dem Netzwerk an, unter anderem die verbundenen Container.

Container und Netzwerke

Container mit Netzwerk starten.

```
docker run --network mynetwork nginx
```

Laufenden Container in ein Netzwerk einbinden.

```
docker network connect <network> <containername>
```

Aufgaben

Bridge Netzewrk

Erstelle zwei `nginx` Container mit eindeutigen Namen und verbinde sie mit einem `bridge` Netzwerk.

Verifiziere, dass die Container untereinander mittels ihrem Namen im Netzwerk aufgelöst werden und die Nginx Standardseite angezeigt wird. Funktioniert das auch mit `debian` Containern?

Interne Kommunikation

Erstelle ein Docker Image, das sowohl Nginx als auch einen weiteren Webserver beinhaltet. Eine einfache Variante ist beispielsweise `python3 -m http.server 81` Nginx soll auf Port 80, der andere Webserver auf Port 81 gerichtet sein.

Öffne eine Shell im Container und verifiziere, dass beide Services über ihre Ports erreichbar sind.

Speichermöglichkeiten in Docker

Arten von Volumes

Mit Volumes können wir den Containern eine Möglichkeit geben, Daten auf dem Host zu speichern und sie nach dem neustart wieder verwenden zu können.

Bind Mounts

Streng genommen sind Bind Mounts keine Volumes, dennoch können sie in vielen Fällen das gleiche Problem wie Volumes lösen. Bei einem Bind Mount wird eine Datei oder ein Ordner unter einem absoluten Pfad in den Container eingebunden. Dieser Speicherort ist in der Regel offensichtlich und so auch für Personen verständlich, die sich wenig mit Docker auskennen. Außerdem sind Bind Mounts recht performant.

Jedoch kann die Tatsache, dass Bind Mounts vom Betriebssystem und absoluten Pfaden im Dateisystem abhängen, bei komplexeren Anwendungen oder besonderen Anforderungen problematisch werden.

So lässt sich ein Container mit einem Bind Mount starten, der einen Ordner `target` im aktuellen Verzeichnis nach /app im Container einbindet:

```
docker run -d --name w1 --mount type=bind,source="$(pwd)"/target,target=/app
nginx:latest
```

Volumes

Volumes werden komplett von Docker verwaltet und durch verschiedene Volume Treiber, sind unterschiedliche Speichermöglichkeiten verfügbar. Einige Treiber sind als Plugin für Docker erhältlich.

Volumes werden von Docker in `/var/lib/docker/volumes` mit ihrem Hash gespeichert.

```
docker volume create myvolume
```

```
docker run -d --name w1 --mount type=volume,source=myvolume,target=/app
nginx:latest
```

Volumes können genau wie Bind Mounts genutzt werden, Daten zwischen den Containern auszutauschen.

tmpfs Mounts

tmpfs Mounts sind nur unter Linux verfügbar. Sie sind nur im Arbeitsspeicher persistiert und werden beim Neustart des Containers wieder gelöscht.

Mit tmpfs mounts können keine Daten zwischen Containern übertragen werden.

Daher eignen sich diese Mounts besonders für die performante und kurzfristige Ablage von Dateien, zum Beispiel den Inhalt einer Message Queue.

```
docker run -d --name w1 --mount type=tmpfs,destination=/app nginx:latest
```

Volumes Verwalten

Zum Verwalten der Volumes sind folgende Befehle hilfreich.

```
docker volume create myvolume
docker volume inspect myvolume
docker volume rm myvolume
docker volume prune
docker volume ls
```

Aufgaben

Volumes Verwalten

- Erstelle ein neues Volume mit dem Namen `foo` .
- Wo wird das Volume gespeichert.
- Starte einen Nginx Container und binde das Volume `foo` so ein, dass der Nginx Container den Inhalt des Volumes ausliefert. Du kannst dafür die folgende Konfiguration verwenden:

```
server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    autoindex on;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

Diese Konfiguration muss im Nginx Container unter `/etc/nginx/conf.d/default.conf` abgelegt werden.

- starte einen `debian` Container, binde das `foo` Volume ein und installiere einen Texteditor.
- Editiere im Debian Container den Inhalt des Volumes.
- Beobachte, dass sich die Dateien die Nginx ausliefert im anderen Container ändern.

Bind Mounts

- Ersetze das Volume mit einem Bind Mount.
- Nutze einen Texteditor im Host Bestriebssystem um die Dateien zu verändern, die Nginx ausliefert.
- Verifiziere, dass sich deine Änderungen auf Nginx auswirken.

Docker Compose

- Orchestrierung mehrerer Container auf einem System

Grundlagen

In einer YAML Datei, auch **Compose file** genannt, namens `docker-compose.yml` definieren wir mehrere Container, Netzwerke, Volumes, Ports und Umgebungsvariablen für die Container. Das erlaubt uns, besonders mit Bind Mounts, komplexere Anwendungen in der Konfigurationsdatei zu definieren (Infrastructure as Code) und sie mittels `docker compose` zu verwalten.

> Die YAML Datei muss nicht unbedingt docker-compose.yml heißen. Nennen wir sie anders so müssen wir Docker Compose mit dem `-f myfile.yml` Switch sagen, welche Datei im aktuellen Befehl genutzt werden soll.

Eine `docker-compose.yml` Datei kann beispielsweise so aussehen:

```yaml
services:
  web:
    image: nginx
    ports:
      - "8080:80"
    volumes:
      - "./www:/usr/share/nginx/html:ro"
      - "./default.conf:/etc/nginx/conf.d/default.conf:ro"
```

Hier definieren wir einen Service namens `web`, der auf dem Image `nginx` basiert. Der Container Port 80 wird auf den Host Port 8080 verbunden und es werden zwei Bind Mounts definiert: Die Konfigurationsdatei und der Ordner, den Nginx ausliefern soll.

> `Volumes` würden hier mit ihrem Namen angegben werden, und nicht mit einem absoluten oder relativen Pfad

---

Befehle

Doker compose unterstützt ähnliche Befehle wie `docker`:

```
docker compose up
docker compose up -d
docker compose pull
docker compose restart
docker compose down
docker compose ps
docker compose logs
```

Einsatzzwecke von Docker Compose

Docker Compose kann dann gut eingesetzt werden, wenn eine Anwendung aus mehreren Containern besteht und diese gemeinsam orchestriert werden müssen. Insbesondere für kleinere Anwendungen und Hostingumgebungen, die den kostenspieligen Einsatz von Kubernetes nicht rechtfertigen, ergibt Docker Compose Sinn. Dies ist beispielswesie der Fall für Anwendungen, die nur im internen Firmennetzwerk erreichbar sind, da die Zahl der Nutzer nicht spontan ansteigen wird.

Ein anderer Anwendungsfall ist das Hosting einer öffentlichen SaaS Lösung, die sich noch im Wachstum befindet.

Docker und Docker Compose abstrahieren das Hosting von Webanwendungen, sodass mehrere Anwendungen auf einem Server betrieben werden können, wobei die Unterschiede zwischen den Anwendungen für die Administratoren minimal sind. Auch unbekannte Technologien können so leichter zugänglich werden, ohne sich mit der verwendeten Programmiersprache im Detail beschäftigen zu müssen.

Durch diese Vereinfachung können solche Hostingumgebungen sehr leicht automatisiert eingerichtet und verwaltet werden.

Netzwerke

Mittels `docker compose` können auch Netzwerke zwischen meheren Containern definiert werden:

```
services:
  backend:
    image: mycompany/mybackendimage:latest
    restart: unless-stopped
    environment:
      DB_HOST: db
      DB_USER: myapplication
      DB_PASSWORD: mysecretpassword
    ports:
      - "8080:80"
    networks:
      - internal
    depends_on:
      db:
        condition: service_healthy
        restart: true

  db:
    image: postgres
    environment:
      - POSTGRES_USER=myapplication
      - POSTGRES_PASSWORD=mysecretpassword
      - POSTGRES_DB=myapplication
    volumes:
      - ./postgres-data:/var/lib/postgresql/data
    networks:
      - internal

networks:
  internal:
```

Aus dem `internal` Netzwerk wird werden keine Ports an die Außenwelt exponiert, nur Port 80 des `backend` Containers ist von außen zugänglich. Die Datenbank ist von außen nicht erreichbar. Das Volume der Datenbank sorgt dafür, dass die Daten auch nach einem Neustart erhalten bleiben.

Diese Anwendung lässt sich leicht durch Microservices, Message Broker und Caching Services erweitern.

Umgebungsvariablen

Wie wir im letzten Beispiel gesehen haben, können sich Umgebungsvariablen bereits bei zwei Containern oft wiederholen. Daher unterstützt docker compose auch `.env` Dateien.

Dazu erstellen wir eine `.env` Datei im gleichen Ordner wie die `docker-compose.yml`:

```
# DB Connection Information:
DB_HOST=db
DB_USER=myuser
DB_PASSWORD=mypassword
DB_DATABASE_NAME=mydatabase
```

Diese Variablen können dann im `doker-compose.yml` verwendet werden:

```
services:
  backend:
    image: mycompany/mybackendimage:latest
    environment:
      DB_HOST: DB_HOST
      DB_USER: DB_USER
      DB_PASSWORD: DB_PASSWORD
      DB_DATABASE_NAME: DB_DATABASE_NAME
    ports:
      - 8080:80
    networks:
      - internal

  db:
    image: postgres
    environment:
      - POSTGRES_USER=DB_USER
      - POSTGRES_PASSWORD=DB_PASSWORD
      - POSTGRES_DB=DB_DATABASE_NAME
    volumes:
      - ./postgres-data:/var/lib/postgresql/data
    networks:
      - internal

networks:
  internal:
```

Damit ist das Ändern der Konfiguration ein leichtes.

Skalierung

Starten wir folgenden service, so beobachten wir, dass die Container durchnummeriert sind:

```
services:
  database:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=hackmd
      - POSTGRES_PASSWORD=hackmdpass
      - POSTGRES_DB=hackmd
    volumes:
      - ./codimd-database:/var/lib/postgresql/data
    networks:
      backend:
    restart: unless-stopped

  app:
    image: quay.io/hedgedoc/hedgedoc:1.9.9
    environment:
      - CMD_DOMAIN=localhost
      - CMD_PROTOCOL_USESSL=false
      - CMD_URL_ADDPORT=false
      - CMD_ALLOW_ANONYMOUS=false
      - CMD_ALLOW_ANONYMOUS_EDITS=true
      - CMD_ALLOW_EMAIL_REGISTER=false

    restart: unless-stopped
    depends_on:
      - database

    networks:
      - default
      - backend

networks:
  backend:
```

```
docker compose ps
```

```
NAME                            IMAGE                                  COMMAND
SERVICE    CREATED          STATUS                     PORTS
hedgedoc_example-app-1          quay.io/hedgedoc/hedgedoc:1.9.9   "/usr/local/
bin/dock…"   app        6 minutes ago    Up 6 minutes (healthy)   3000/tcp
hedgedoc_example-database-1   postgres:15-alpine                "docker-
entrypoint.s…"   database   15 minutes ago   Up 15 minutes            5432/tcp
```

Folgendermaßen können wir den Backend Service hochskalieren:

```
services:
  database:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=hackmd
      - POSTGRES_PASSWORD=hackmdpass
      - POSTGRES_DB=hackmd
    volumes:
      - ./codimd-database:/var/lib/postgresql/data
    networks:
      backend:
    restart: unless-stopped

  app:
    image: quay.io/hedgedoc/hedgedoc:1.9.9
    deploy:
      replicas: 2
    environment:
      - CMD_DOMAIN=localhost
      - CMD_PROTOCOL_USESSL=false
      - CMD_URL_ADDPORT=false
      - CMD_ALLOW_ANONYMOUS=false
      - CMD_ALLOW_ANONYMOUS_EDITS=true
      - CMD_ALLOW_EMAIL_REGISTER=false

    restart: unless-stopped
    depends_on:
      - database

    networks:
      - default
      - backend

networks:
  backend:
```

Alternativ können wir dies auch über die Kommandozeile durchführen:

```
docker compose scale app=2
```

Daraufhin startet Docker Compose zwei Instanzen des App Containers:

```
NAME                              IMAGE                                  COMMAND
SERVICE    CREATED              STATUS                          PORTS
hedgedoc_example-app-1            quay.io/hedgedoc/hedgedoc:1.9.9   "/usr/local/
bin/dock…"   app        About a minute ago   Up About a minute (healthy)
3000/tcp
hedgedoc_example-app-2            quay.io/hedgedoc/hedgedoc:1.9.9   "/usr/local/
bin/dock…"   app        About a minute ago   Up About a minute (healthy)
3000/tcp
hedgedoc_example-database-1   postgres:15-alpine                 "docker-
entrypoint.s…"   database   57 minutes ago       Up 57 minutes
5432/tcp
```

Damit dies funktioniert dürfen die Container nicht an einen Host Port gebunden sein, da der zweite Container sich nicht mehr an den gleichen Port binden kann wie der erste Container.

Stattdessen exponieren wir die Ports nur im `backend` Netzwerk und schalten beispielsweise einen Nginx als Reverseproxy vor. Nginx leitet die Anfrage dann an `app` Service weiter, der über die interne Docker Networking DNS Auflösung zwischen den beiden app-Containern verteilt wird.

Aufgaben

Webcontainer

Betreibe mittels Docker Compose einen Nginx Webserver, der einen Ordner aus dem lokalen Dateisystem ausliefert. Der Ordner soll sich neben der `docker-compose.yml` befinden.

Hosting einer Anwendung

Betreibe mittels Docker Compose eine Wordpress Instanz mit Datenbank.

Unternehmensanwendungen

Betreibe eine kleine Unternehmens IT bestehend aus:

- Dokuwiki
- Forgejo
- Hedgedoc
- Nextcloud

Alle Services sollen in eigenen Docker Compose Dateien beschrieben sein. Die Services sollen auf unterschiedlichen Ports, z.B. 90-93 verfügbar sein. Nach einem Neustart der Container darf der Service keine Daten verlieren. Definiere individuelle Konfigurationen in `.env` Dateien.

Docker Swarm Mode

- Orchestrierung mehrerer Container auf mehreren Systemen

!!Achtung, es gab vor einiger Zeit auch Docker Classic Swarm, was nicht mehr weiterentwickelt wird. Es handelt sich dabei um zwei verschiedene Projekte.

Die folgenden Schritte werden wir gemeinsam durchführen. Dazu stoppt bitte alle Container, die gerade auf den Systemen laufen.

Übersicht

Docker Swarm Mode ermöglicht es uns, mittels der Docker Engine CLI Anwendungen über mehrere Docker Daemons verteilt zu betreiben.

Ein Docker Swarm besteht dabei aus mehreren Systemen oder Nodes. Die Nodes müssen untereinander über feste IP Adressen erreichbar sein. Die Nodes unterteilen sich in **Manager** und **Worker**.

Manager Nodes

Manager Nodes verwalten den Swarm. Hier definieren wir die Services, die wir gerne betreiben möchten. Die Manager Node leitet aus unserer Service Definition Aufgaben ab, die als **Tasks** an die Worker Nodes verteilt werden. Standardmäßig erfüllen Manager Nodes auch die Aufgabe einer Worker Node.

Sind mehrere Manager Nodes implementiert wählen sie untereinander eine primäre Node, die die Aufgaben an die Worker verteilt.

Worker Nodes

Im Umkehrschluss empfangen Worker Nodes die Tasks von den Managern und setzen sie um. Dabei kommunizieren sie ihren Stand an die Manager Nodes. Die Container im Swarm laufen auf den Worker Nodes.

Service

Ein Service ist eine Definition von Anweisungen an die Worker und die primäre Art und Weise wie User mit dem Swarm kommunizieren.

Es wird zwischen zwei Arten von Services unterschieden:

Replicated Services

Hierbei verteilt der Manager eine festgelegte Anzahl an replicas eines Containers über den Swarm.

Global Services

Es wird auf jeder Worker Node ein Container dieser Art gestartet.

Load Balancing

Wie auch schon Docker Compose bringt der Swarm Mode DNS und ein einfaches round-robin Loadbalancing mit. Es ist jedoch auch möglich externe Loadbalancer oder Nginx/ HAProxy-Container als Loadbalancer zu verwenden, um andere Loadbalancing Strategien einzusetzen.

Installation

Vorbereitungen

Zuerst muss Docker Engine auf den Servern installiert sein.

Die Server müssen sich gegenseitig über ihre IP Adresse erreichen können und die folgenden Ports müssen erreichbar sein:

- 2377 TCP Kommunikation zwischen Manager und Worker
- 7946 TCP/UDP Overlay Network Discovery
- 4789 UDP Overlay Network

Wichtig ist dass 4789 nicht öffentlich zugänglich ist. Wenn dem Netzwerk der Swarm Hosts nicht vertraut wird, sollte ein Verschlüsseltes Overlay Netzwerk benutzt werden.

Installation der Manager Node

Auf der Manager Node initialisieren wir den Swarm:

```
docker swarm init --advertise-addr <MANAGER-IP>
```

Installation auf den Worker Nodes

Aus dem `init` Befehl resultieren Anweisungen, wie wir die Worker dem Swarm hinzufügen können:

```
docker swarm join --token xxxx <MANAGER-IP>:2377
```

Sollte das Token abhanden gekommen sein oder sollten weitere Nodes (als Worker oder Manager) später hinzugefügt werden, kann so das Token wieder generiert werden:

```
docker swarm join-token worker
docker swarm join-token manager
```

Status

Mit

```
docker info
```

erhalten wir Informationen über den aktuellen Zustand des Swarms:

```
...
Swarm: active
 NodeID: xxxx
 Is Manager: true
 ClusterID: xxxx
 Managers: 1
 Nodes: 2
...
```

Mit

```
docker node ls
```

erhalten wir informationen über die angeschlossenen Server:

```
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER
STATUS   ENGINE VERSION
u1saxm5vs9zi81u8b68ammkq4 *   host0      Ready     Active         Leader
27.3.1
ij58v11ws23jtjzepmf6kd2et     host1      Ready     Active
27.3.1
```

Der `*` Zeigt an, dass wir gerade auf diesem Host eingeloggt sind.

Deinstallation

Nodes in einem Docker Swarm können mit

```
docker swarm leave
```

wieder aus dem Swarm entfernt werden. Da Docker Swarm Mode Teil der gewöhnlichen Docker Engine Installation ist, müssen keine Komponenten deinstalliert werden.

Einen Service ausrollen

Ein sehr einfacher Service kann so ausgerollt werden:

```
docker service create --replicas 1 --name helloworld alpine ping docker.com
```

Services anzeigen

Wir können verifizieren dass der Service läuft:

```
docker service ls
```

```
ID             NAME         MODE         REPLICAS   IMAGE           PORTS
n5v6tt5cy2iw   helloworld   replicated   1/1        alpine:latest
```

Skalieren

Die Skalierung funktioniert ähnlich wie in Docker Compose:

```
docker service scale helloworld=2
```

Details eines Services anzeigen

```
docker service inspect --pretty helloworld
```

```
ID:             n5v6tt5cy2iwnkepl00m31sue
Name:           helloworld
Service Mode:   Replicated
 Replicas:      2
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:
alpine:latest@sha256:beefdbd8a1da6d2915566fde36db9db0b524eb737fc57cd1367effd16d
c0d06d
 Args:          ping docker.com
 Init:          false
Resources:
Endpoint Mode:  vip
```

Auflistung, welche Container des Services auf welchem Host laufen.

```
docker service ps
```

```
ID             NAME            IMAGE           NODE      DESIRED STATE   CURRENT
STATE           ERROR     PORTS
kot9fh2yppbn   helloworld.1    alpine:latest   host0     Running         Running
8 minutes ago
uddark7c7ar3   helloworld.2    alpine:latest   host1     Running         Running
6 minutes ago
```

Rollende Updates

Mittels `--update-delay` kann die Pause zwischen den erfolgreichen Updates der einzelnen Container konfiguriert werden. Sollte ein Container nicht aktualisiert werden können, wird das Update pausiert. Durch einen erneuten Aufruf von `update` kann das Update erneut angestoßen werden.

```
docker service create --replicas 3 --name redis --update-delay 10s redis:7.2.6
docker service update --image redis:7.4.1 redis
```

Jetzt können wir beobachten, dass die Services mit einer Pause von 10s nacheinander aktualisiert werden.

Dieser Updateprozess wird auch visualisiert:

```
docker service ps redis
ID             NAME          IMAGE         NODE      DESIRED STATE   CURRENT
STATE            ERROR     PORTS
rfdzlz71iuwd   redis.1       redis:7.4.1   host0     Running         Running 2
minutes ago
4rleewf8mvqu    \_ redis.1   redis:7.2.6   host1     Shutdown        Shutdown 2
minutes ago
u2scco4bcxsz   redis.2       redis:7.4.1   host1     Running         Running 2
minutes ago
i6k8jdfttzlf    \_ redis.2   redis:7.2.6   host1     Shutdown        Shutdown 2
minutes ago
tis7q8k9v8q6   redis.3       redis:7.4.1   host0     Running         Running 2
minutes ago
4leea3a08266    \_ redis.3   redis:7.2.6   host0     Shutdown        Shutdown 2
minutes ago
```

Node Verfügbarkeit

Eine Node kann auf `drain` gesetzt werden, sodass alle Container von ihr entfernt werden. Dies geschieht sofort.

Wird anschließend die Verfügbarkeit auf `active` gesetzt, können ab jetzt Container auf dieser Node gestartet werden. Zunächst bleibt die Node jedoch leer. Erst wenn ein Service skaliert oder aktualisiert wird, oder eine andere Node auf `drain` gesetzt wird, werden wieder Container auf der aktuellen Node gestartet.

```
docker node update --availability drain xxx
docker node update --availability active xxx
```

Logs

Mittels

```
docker service logs <servicename>
```

können die Logs aller Container eines Services angezeigt werden.

Portfreigaben

Docker Swarm bringt ein `routing mesh` mit, das uns ermöglicht einzelne Ports eines Services zu veröffentlichen.

Dabei wird der Port auf jedem Host geöffnet:

```
docker service create --publish published=8080,target=80   --replicas 3   --
name nginx   --update-delay 10s   nginx:1.26.2
```

```
docker service update \
  --publish-add published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  SERVICE
```

Intern routet Docker Swarm dabei die Anfrage auf einen der möglichen Container mittels round robin. Dabei muss der Container nicht auf dem Host sein, der die Anfrage von außen erhalten hat.

Aufgaben

Einrichtung von Swarm

Richte Docker Swarm Mode auf den beiden Debian VMs ein. Verifiziere durch das Ausrollen des `helloworld` service wie gezeigt.

Verteilter Webserver

- Erstelle ein Docker Image mit einem Webserver und einer Website, die Teil des Images ist.
- Verteile dieses Image auf deinen beiden Debian Maschinen.
- Starte den Webserver mit vier Replicas und binde einen Port an die Container, sodass er extern erreichbar ist.
- Wie sind die Container über die Infrastruktur verteilt?
- Wie wird die Last auf die Contaier verteilt? Probiere es mit `curl` aus.
- Skaliere den Webserver herunter auf eine Instanz. Ist er auf beiden Hosts immer noch erreichbar?

Rollende Updates

- Skaliere auf zehn Instanzen.
- Erstelle eine neue Version deiner Website und rolle sie auf das Cluster aus.

Kubernetes Einführung

- Orchestrierung mehrere Containergruppen über mehrere Systeme hinweg

Wir haben **Docker Compose** kennengelernt, was uns ermöglicht ganze Anwendungen aus mehreren Containern mit mehreren Bedürfnissen in einer Datei zu definieren. Mit **Docker Swarm** haben wir gesehen, wie ein Container Image über mehrere Hosts skaliert werden kann.

Kubernetes (K8s) vereint diese beiden Ansätze. Hier definieren wir `Pods , zusammenhängende Strukturen aus mehreren Containern, die eine Anwendung darstellen können und skalieren mehrere Replicas davon auf mehrere Systeme.

Vorbereitung

Bevor wir unser Kubernetes einrichten sollte zunächst Swarm deaktiviert werden:

```
# Worker node
docker swarm leave

# Manager node
docker swarm leave --force
```

‼️💡 wir werden versuchen Kubernetes auf unseren zwei Virtuellen Maschinen zu installieren. Sollten dabei Probleme auftreten können wir stattdessen Microk8s installieren: https://microk8s.io/docs/getting-started

---

Komponenten

Nodes

Wie auch bereits in Docker Swarm Mode sind in Kubernetes mehreree Node Typen vorgesehen

Master Nodes

Steuern das Cluster und sollten in ungerader Zahl auf dedizierten Hosts installiert werden.

Komponenten

- **API Server** steuert das Cluster und wird vom Frontend angesprochen
- **ETCD**: Key Value Store, im Grunde die einzige Komponente von Kubernetes die unbedingt gesichert werden muss.
- **Scheduler** Sendet aufgaben an die Worker Nodes. K8s bringt einen standard Scheduler mit, es gibt aber auch alternativen
- **Controller Manager** Controller sind Plugins für K8s die seine Fähigkeiten erweitern und es auf spezielle Anforderungen anpassen können. Es gibt standard Controller wie `Node Controller` (Node Verfügbarkeit, CPU und RAM Status) und `Replication Controller` (CRUD von Pods)
- **Cloud Controller Manager** Abstraktionsschicht für die zugrundeliegene Cloudlösung, z.B. vmWare, Azure und AWS mit teils unterschiedlichen Fähigkeiten. Sie organisieren Storage und können auch Netzwerke bereitstellen.

Worker Nodes

Hier wird die Arbeitslast des Clusters betrieben. Worker können beliebig entfernt und neu dazugenommen werden.

Komponenten

- **Kubelet** Agent der auf jeder Node läuft, aufgaben vom Scheduler annimmt und metriken meldet. Er verwaltet Pods und Container.

- **Kube Proxy** Abstraktionsschicht, die Interaktionen mit dem Cluster von außen ermöglicht. Er kann HTTP-Anfragen an die korrekte Node weiterleiten, eine Fähigkeit die aus Performancegründen nicht im Loadbalancer implementiert wird, da sich die Pods zwischen den Workern frei bewegen können. ETCD weiß, welcher Pod wo läuft.
- **Container Runtime** z.B. Docker, rkt oder runc.

Storage Nodes

In K8s sind sie streng genommen nicht vorgesehen, jedoch benötigen die meisten Anwendung die ein oder andere Form von persistentem Speicher.

Storage Nodes sind dabei ein Weg, mit Storage umzugehen. Dabei wird beispielswesie eine Datenbank auf der Storage Node im Container gestartet, die eine lokale, persistente Speichermöglichkeit hat.

Die Container verbinden sich über das Netzwerk mit der Datenbank und können somit leichter und schneller zwischen unterschiedlichen Hosts transferiert werden, da die Storage nicht mit umziehen muss.

Infrastrucure Nodes

Diese unterstützen Aufgaben wie Logging und Monitoring. Wiederum handelt es sich hier um keinen festen K8s Begriff, sondern um eine Mögliche Stragegie, mit Logging und Monitoring umzugehen.

Soft Namespaces

Eine leiche multi-tenant Abstraktion auf dem Cluster, die nicht für die IT Sicherheit geschaffen wurde, sondern als Organisationsschicht. Namespaces trennen Objekte voneinander, sodas ein Namespace beispielsweise eine anwendung von einer anderen trennen kann. Zusätzlich können Ressourcengrenzen (2Cores, 4GB RAM) pro Namespace definiert werden. Namespaces können untereinander kommunizieren.

Service Accounts können einem oder mehreren Namespaces zugewiesen werden, in denen sie Änderungen durchführen dürfen. Dadurch kann verhindert werden, dass ein User das ganze Cluster und alle Anwendungen löscht.

Pods

Pods sind Mengen von einem oder mehr Containern, die gemeinsam Skalieren sollen, zum Beispiel eine Anwendungscontainer und ein Logging Container. Beim Skalieren werden identische Kopien gestartet.

Services

Normalerweise kommunizieren Pods nicht diret mit der Außenwelt, sondern über einen Service. Im Grunde sind Services ein internes cluster DNS, da eine von außen erreichbare URL, z.B. `NAME.NAMESPACE.srv.cluster.local` and mehrere IP Adressen und Ports im Cluster weitergeleitet werden kann, auf denen die entsprechenden Pods erreichbar sind.

Services sollten den Usern hinter einem Load Balancer angeboten werden, da diese oft ein besseres Featureset mitbringen als das Quasi-loadbalancing was über den Service alleine möglich ist.

Selector

Ein Label an K8s Objekten das benutzt weden kann, um mögliche Ziele von Anfragen einzuschränken. Hier muss jedoch besonders darauf geachtet werden, dass die Datenbank nicht das Label vom Anwendungspod bekommt.

Replica Set / Deployment

Definiert die Menge an identischen Pods die generiert werden soll. Deployments können skaliert werden, was die Menge an Pods verändert. Bei Problemen sollte immer auf 0 Skaliert werden statt das Deployment zu löschen.

Daemon set

Definiert einen Pod, der auf jeder Node ein mal gestartet wird. Diese werden in der Regel für Monitoring oder Logging genutzt, z.B. mit `Prometheus node_exporter` oder dem `Zabbix`

`Agent`.

Kubernetes Betrieb

Einige hilfreiche Befehle:

```
kubectl get nodes
kubectl get pods -n <namespace>
kubectl get services --all-namespaces
```

Installation von Kubernetes

Um Kubernetes zu installieren wird in der Regel eines der folgenden Tools/Stragetien benutzt:

- **Kubeadm** Ohne cloud provider Interaktion und eher kompliziert, nicht die beste Wahl für Produktionsumgebungen
- **Kubespray** Sehr vielseitig, aber nicht für AWS geeignet
- **Kops** Sehr gut für Amazon
- **From source**

Tech Stack

Wir werden folgende Komponenten installieren:

- Kubeadm
- Docker
- Kubelet
- CNI Controller Network Interface

Vorbereitungen

Swap deaktivieren

```
swapoff -a
```

Entferne die Zeile `swap` von `/etc/fstab`

Basiskomponenten Installieren

```
sudo apt-get update
# apt-transport-https may be a dummy package; if so, you can skip that package
sudo apt-get install -y apt-transport-https ca-certificates curl gpg

# If the directory `/etc/apt/keyrings` does not exist, it should be created
before the curl command, read the note below.
# sudo mkdir -p -m 755 /etc/apt/keyrings
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg
--dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

# This overwrites any existing configuration in /etc/apt/sources.list.d/
kubernetes.list
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://
pkgs.k8s.io/core:/stable:/v1.31/deb/ /' | sudo tee /etc/apt/sources.list.d/
kubernetes.list

sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl

sudo systemctl enable --now kubelet
```

Ein Cluster erstellen

💡Gegebenfalls ist es notwendig `/etc/containerd/config.toml` zu editieren und sicherzustellen, dass `cri` nicht in den `disabled_plugins` ist. In der Regel kann die Zeile auskommentiert werden. Danach muss `containerd` neu gestartet werden `systemctl restart containerd`.

```
kubeadm init
```

```
...
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each
as root:

kubeadm join <ip>:<port> --token xxx.xxx \
    --discovery-token-ca-cert-hash sha256:xxx
```

Der Befehl kann auch mit `kubeadm token create --print-join-command` bei bedarf generiert werden.

```
kubectl get nodes
NAME    STATUS     ROLES           AGE     VERSION
host0   NotReady   control-plane   10m     v1.31.2
host1   NotReady   <none>          2m24s   v1.31.2
```

Die Hosts sind `NotReady` da das Networking noch fehlt.

Networking hinzufügen

Wie uns der Installationsbericht schon anzeigt müssen wir noch ein Netzwerklayer installieren. Wir nutzen `calico`:

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Falls hier Probleme auftreten kann das daran liegen, dass etwas mit den Rechten nicht stimmt. Dazu kann als Root die Umgebungsvariable `KUBECONFIG` wie oben beschrieben gesetzt werden.

Nach der installation von Calico haben wir ein funktionierendes Cluster:

```
kubectl get nodes
NAME    STATUS   ROLES           AGE     VERSION
host0   Ready    control-plane   4m55s   v1.31.2
host1   Ready    <none>          43s     v1.31.2
```

Service Account

Zuerst legen wir einen Namespace an:

```
kubectl create namespace entenhausen
```

Anschließend einen User:

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: donald
  namespace: entenhausen
```

Und vergeben mit einem `ClusterRoleBinding` die Reche an den user `donald` im Namespace `entenhausen`:

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: donaldAdminBinding
subjects:
  - kind: ServiceAccount
    name: donald
    namespace: entenhausen
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
```

Kubernetes in Docker Desktop

https://docs.docker.com/desktop/kubernetes/

Deployments und Replica Sets

Wir definieren ein Deployment mit einem Nginx Container pro Pod mit fünf Replicas:

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: entenhausen
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Und ein zweites Deployment mit Ubuntu containern:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu2-deployment
  namespace: entenhausen
  labels:
    app: ubuntu2
spec:
  replicas: 5
  selector:
    matchLabels:
      app: ubuntu2
  template:
    metadata:
      labels:
        app: ubuntu2
    spec:
      containers:
        - name: ubuntu
          image: ubuntu
          command: ["/bin/bash", "-c", "--"]
          args: ["while true; do sleep 30; done;"]
          ports:
            - containerPort: 80
```

Um die Nginx Container zu erreichen benötigen wir einen Service. Mit dem Typ `NodePort` öffnen wir einen Port auf jeder Node.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: entenhausen
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30080
```

Und Netzwerkregeln, eine um den Traffic standardmäßig zu verweigern:

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: entenhausen
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

Und eine Netzwerkregel die unseren Nginx erreichbar macht:

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-pod
  namespace: entenhausen
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: ubuntu2
      ports:
        - protocol: TCP
          port: 80
```

Schlussendlich müssen die Dateien einzeln oder auch als eine Datei an Kubernetes übergeben werden:

```
kubectl create -f nginxDeployment.yml
kubectl create -f ubuntu1Deployment.yml
kubectl create -f ubuntu2Deployment.yml
kubectl create -f nginxService.yml
kubectl create -f createDefaultDenyNetworkPolicy.yml
kubectl create -f createAllowPodNetworkPolicy.yml
```