



**GU**

**GEDOPLAN**

## Docker und Kubernetes für Java-Entwickler

Kompakte Einführung in die Entwicklung und den Betrieb  
containerbasierter Java-Anwendungen

Enterprise Java Team, GEDOPLAN GmbH

Dieses Handout wird Ihnen persönlich zur Teilnahme am GFU-Seminar am 29.03.2022 überreicht.  
Eine Weitergabe oder Präsentation außerhalb des Seminars ist nicht erlaubt.





## Inhalt

- Docker
  - Motivation für den Einsatz von Container-Images
  - Gegenüberstellung virtuelle Maschine vs. Container
  - Begrifflichkeiten
    - Image, Registry, Container
  - Docker-Kommandos
  - Image-Erstellung
  - Volumes einrichten und nutzen
  - Registries einsetzen



## Inhalt

### ☰ Kubernetes

- ☰ Motivation für die Orchestrierung von Containern mit Kubernetes
- ☰ Grundsätzlicher Aufbau der Plattform
- ☰ Deklarativer Ansatz
- ☰ YAML-Files zur Beschreibung von Kubernetes-Objekten
- ☰ Building Blocks von Kubernetes
  - Deployments, Services, Volumes, Secrets, Namespaces, etc.

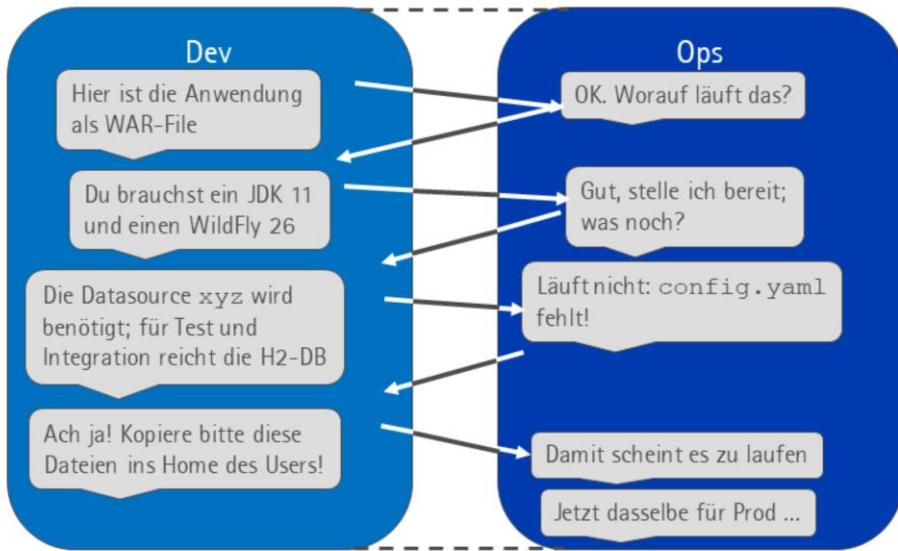


## Containerisierung mit Docker

Überblick über das Paketieren von Anwendungen mit Docker



## Problemstellung: DevOps





## Problemstellung: DevOps

- Entwickler immer häufiger zuständig für:
  - Entwicklung
  - Auslieferung
  - Betrieb



- Automatisieren des gesamten Prozesses gewünscht
  - Weniger fehleranfällig
  - Reproduzierbar



## Problemstellung: Unterschiedliche Umgebungen

- Anwendung läuft auf
  - Produktion, Test, Integration, Entwicklerrechner, ...
- Umgebungen unterschiedlich konfiguriert
- Aufsetzen neuer Umgebungen schwierig
- => Schlecht für Automatisierung und Nachvollziehbarkeit



## Problemstellung: Ausliefern/Verteilen von Software

- Kein einheitliches Format
  - Java-Anwendung (jar, war) oder Executable
- Kein Standardverfahren für Verteilung
- Vorbereiten der Server im Vorfeld erforderlich
- => Viele Probleme für Dev und Ops



## Virtualisierung mit VMs

- Verpacken der Anwendung mit kompletten OS
  - + Einstellungen, Tools und Bibliotheken
- Automatisierbar mit Werkzeugen wie Ansible
  - Mit Aufwand verbunden
- Schwergewichtig
  - Startzeiten
  - Ressourcenverbrauch
  - Große Images



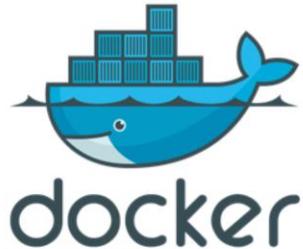
Microsoft  
Hyper-V





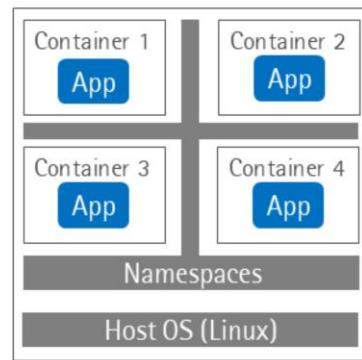
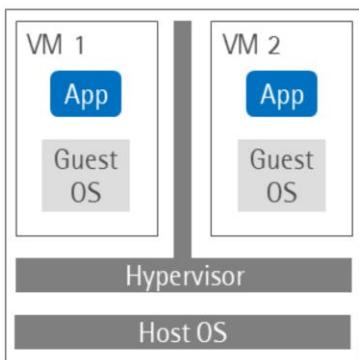
## Docker

- Offene Container-Plattform
- Leichtgewichtige Alternative zu VMs
  - Funktioniert mit Prozessisolierung
- Anwendungen werden in Images verpackt
  - Inklusive aller Abhängigkeiten
  - Immutable, wiederverwendbar
- Community und Enterprise Edition verfügbar





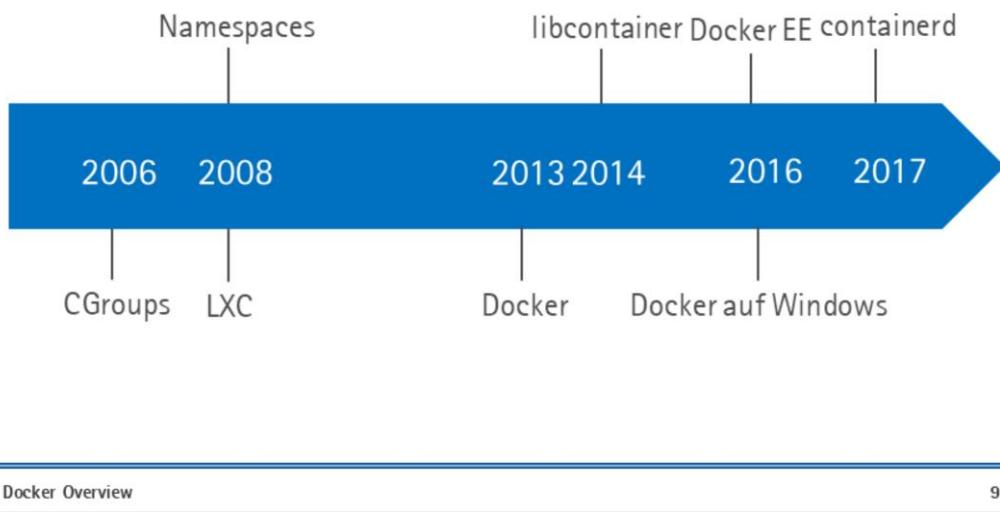
## Unterschied: Virtuelle Maschine <-> Container



- VM = „kompletter Rechner“
- Hypervisor teilt HW zu
- Isolation durch Hypervisor
- Container = Prozess auf Host OS
- Isolation durch Namespaces



## Historie – Linux Container und Docker





## Docker Versionen und Installation

### ☰ Docker Community Edition

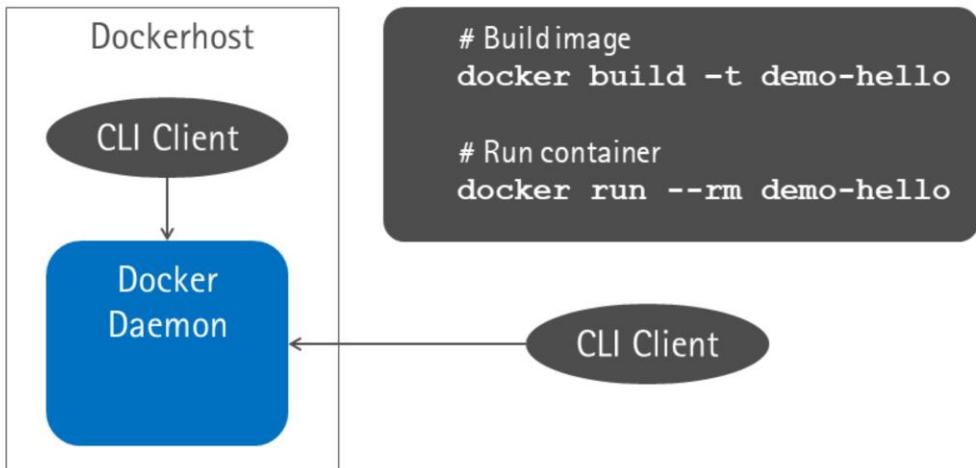
- ☰ Linux: docker-ce per Package Manager
- ☰ Windows: Docker Desktop for Windows
- ☰ Mac: Docker Desktop for Mac

### ☰ Docker Enterprise Edition

- ☰ Linux: docker-ee per Package Manager
- ☰ Windows Server: Docker Engine



## Docker





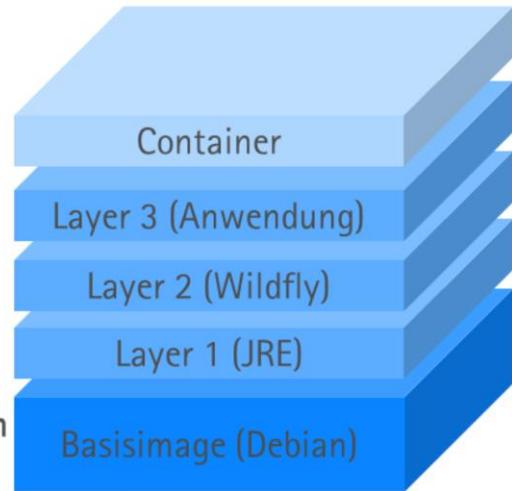
## Images und Container

### ■ Image

- Layered Dateisystem
- Immutable
- Identifizierbar über Tag

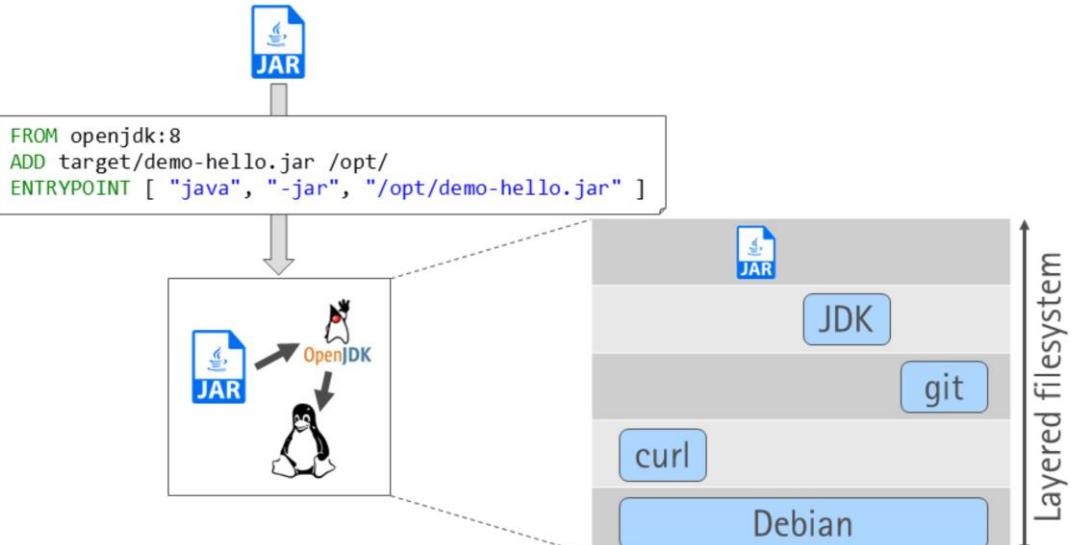
### ■ Container

- Prozess
- Filesystem von Image
- Eigener Layer für Schreiben





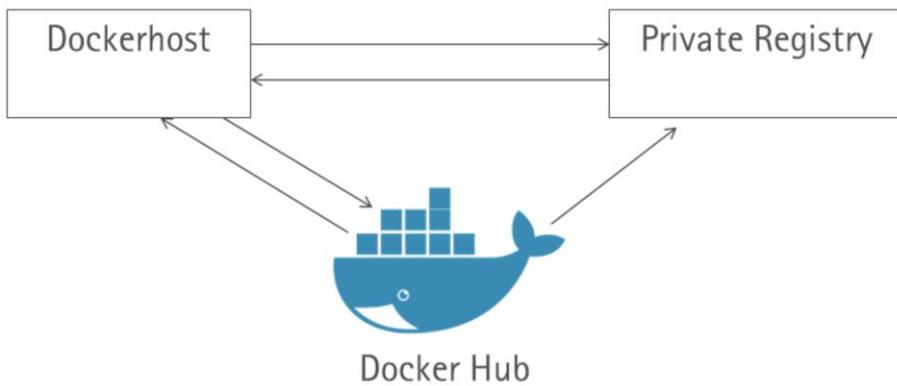
## Images erstellen





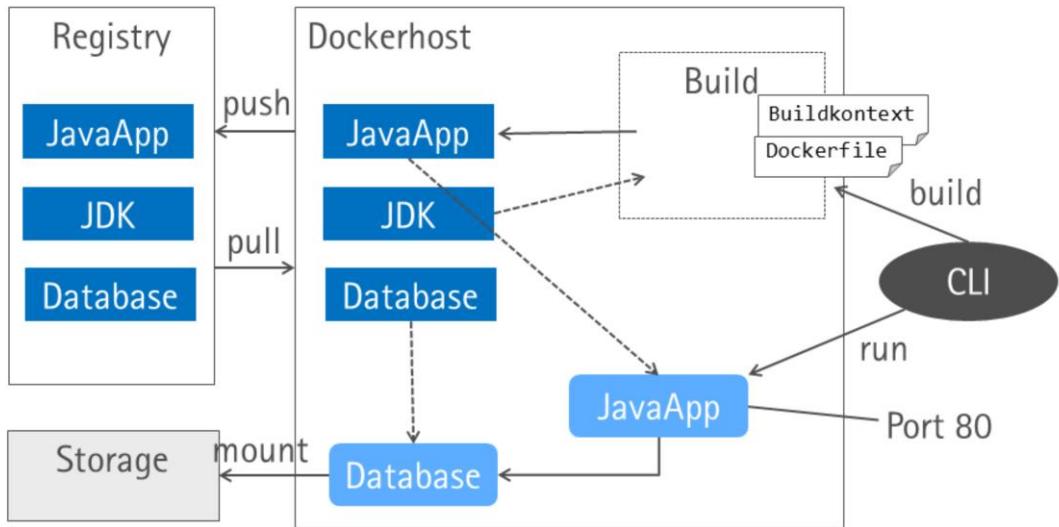
## Images verteilen

- ☰ Austausch der Images über Registries
- ☰ Docker Hub als zentrale Default Registry





## Überblick





## Docker für Paketieren von (Java-)Anwendungen

- Dockerfile eingecheckt in Projekt im SCM
- Image der Anwendung bauen
  - JDK + AppServer + Tools + Jar
- Ausgeliefert wird ein Docker Image
  - Ergebnis des (CI-)Builds
  - Verteilt auf Maschinen über Registries
  - Starten der Anwendung mit docker



## Open Container Initiative

- Projekt der Linux Foundation
- Definiert Standards im Bereich der Containerisierung
- Derzeit zwei Spezifikationen
  - runtime-spec
  - image-spec
- Docker implementiert, nutzt diese Spezifikationen





## Orchestrierung von Containern

- Container brauchen Laufzeitplattform
  - Start, Stopp, Überwachung
  - Verteilung (On-prem / Cloud)
  - Networking / Discovery
  - Skalierung
  - Persistenz

- => Lösung: Orchestrierungsplattform





## Vorteile von Docker für DevOps

- Gleiche Images in allen Umgebungen
  - Fehler nachvollziehbar
- Konfiguration der Anwendung in Image
  - Einfaches Ausrollen in neuen Umgebungen
- Einfaches Verteilen der Anwendungen
- Einfach automatisierbar
  - => Gut für Continuous-Integration/Delivery



## Vorteile von Docker für Administratoren

- Generische Schnittstelle für Anwendungen
- Immutable Images, keine persistenten Daten in Container
  - Neustart im Originalzustand möglich
- Verwaltung einfacher als VMs
- Infrastruktur unabhängig von Anwendungen

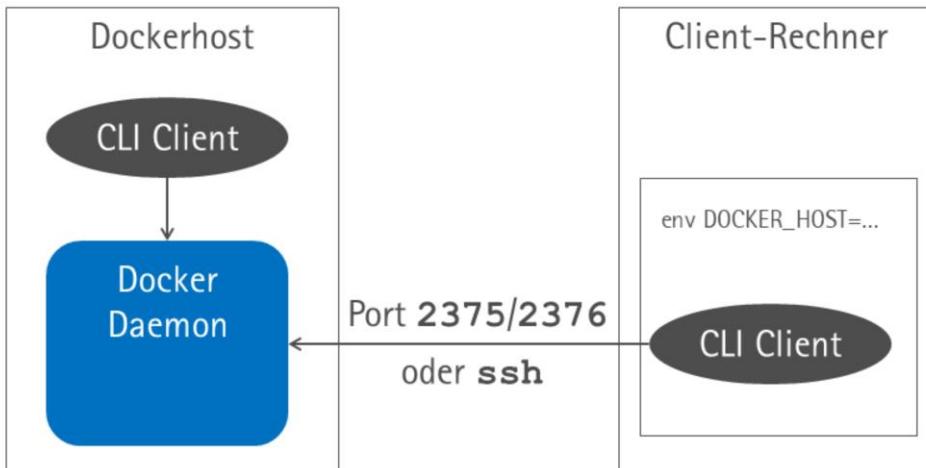


## Docker Basics

Docker Grundlagen für Java Entwickler



## Dockerhost und Client



Standardmäßig steht der Docker-Daemon auf einer Linux-Maschine auf dem Socket `/var/run/docker.sock` lokal zur Verfügung. Es besteht aber natürlich auch die Möglichkeit, einen Remote-Client zu verwenden. In diesem Fall wird standardmäßig der Port 2375 oder 2376 (`https`) verwendet.

Um einen Docker-Client mit einem Remote-Daemon zu verbinden, muss entweder beim Befehl der Parameter `-H host:port` mitgegeben werden oder die Umgebungsvariable `DOCKER_HOST` gesetzt sein.

Diese Vorgehensweise ist allerdings nicht empfehlenswert, weil das Freigeben des Docker-Daemons natürlich ein weiteres potentielles Risiko darstellt. Da auf den meisten Servern `ssh` eingerichtet ist, gibt es seit der Version 18.09 die Möglichkeit, dass sich Docker selbst über `ssh` verbindet. Dafür muss dann auf dem Client-Rechner eine `ssh-config` eingerichtet sein und für Docker der Parameter oder die Umgebungsvariable mit der entsprechenden `ssh`-URI belegt werden (`-H ssh://user@host`).

Informationen über die Client- und Daemon-Version können über den Befehl `docker version` angezeigt werden.

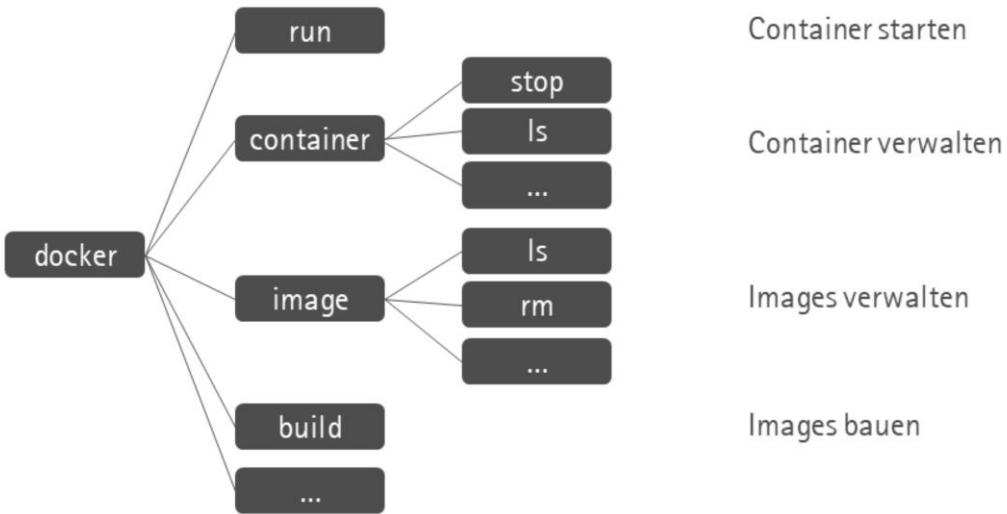


### Docker-Befehle:

- `docker version`



## Docker CLI



Das Kommandozeilenwerkzeug `docker` ist in eine ganze Reihe von Unterbefehlen eingeteilt. Hierbei sind mittlerweile seit der Version 1.13 die Kommandos in einer hierarchischen Struktur gegliedert worden. Die Befehle, welche weitere Kommandos zusammenfassen, werden als Management Commands bezeichnet.

Alle Befehle, die mit dem Verwalten von Images zu tun haben, sind unterhalb von `image` zu finden, alle mit Containern arbeitende unterhalb von `container` usw.

Es gibt aber zum Teil auch noch ältere Befehle direkt unterhalb von `docker`. Hier entspricht z. B. `docker ps` dem neueren Befehl `docker container ls`, welcher dafür zuständig ist, Container aufzulisten.

Diese neue Einteilung des CLI dient der Übersichtlichkeit und es empfiehlt sich, die neueren Management Commands zu nehmen, weil die Befehle dadurch sprechender wirken und natürlich auch ein zukünftiges Wegfallen der alten Kommandos nicht auszuschließen ist.



## Starten von Containern

- Run-Befehl erzeugt und startet einen Container

```
docker run <image-tag>
```

- Image wird geladen, falls nicht vorhanden
  - Defaultmäßig von Docker Hub



`docker run` startet einen neuen Docker-Container zu dem angegebenen Image-Tag. Wenn das Image noch nicht vorhanden ist, wird dieses noch von Docker Hub bzw. der Registry im Tag-Namen heruntergeladen. Der daraus entstehende Container bekommt eine ID und einen Namen zugewiesen. Der Name kann über den Parameter `--name` auch durch den Aufrufer festgelegt werden.



### Docker-Befehle:

- `docker run hello-world`
- `docker run --name hello-world hello-world`



## Kommando bei Start in Container ausführen

- Auszuführenden Befehl anhängen
  - Funktioniert, wenn Container keinen Startprozess hat

```
docker run <image-tag> <command>
```

- Interaktive Shell öffnen mit
  - -t: Anmelden einer TTY Pseudo-Shell
  - -i: Interaktiv, leitet Input in Container

```
docker run -ti <image-tag> /bin/sh
```

Falls ein Image keinen Prozess definiert hat, der beim Starten ausgeführt wird, so wie das in der Regel bei Images von Linux-Distributionen der Fall ist, kann einfach beim `docker run` ein Befehl mitgegeben werden. Dieser Befehl wird nun bei Containerstart im Container aufgerufen.

Falls eine interaktive Shell ausgeführt werden soll, so muss zum einen als Befehl die Shell im Container also z. B. `bash` oder `sh` mitgegeben werden. Des Weiteren ist das Reservieren eines Pseudo-Terminals für den Container-Prozess über `--tty` (kurz `-t`) notwendig sowie das Offenhalten der Standardeingabe über `-i`.

Das Starten des Container mit `-it` führt dazu, dass auch ein im Hintergrund mit `-d` gestarteter Container bei Prozessende nicht beendet wird, da die Shell am Leben gehalten wird.

Achtung: Sollten Sie eine Windows-Version der Bash zum Absetzen der `docker`-Kommandos nutzen (z. B. die `Git-bash`), werden Pfadnamen (wie oben `/bin/sh`) vor der Ausführung in Windows-Pfade konvertiert (z. B. `C:\Program Files\Git\bin\sh`), was für das im Container ausgeführte Kommando unsinnig ist. Dieses Verhalten können Sie unterdrücken, indem Sie in den ersten Schrägstrich des Pfads verdoppeln (also: `docker run -ti <image-tag> //bin/sh`).

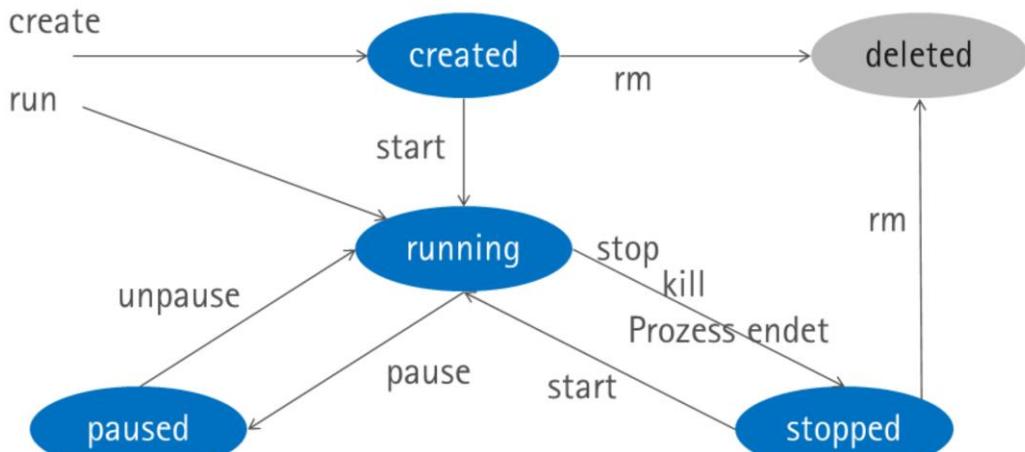


### Docker-Befehle:

- `docker run alpine cat /etc/passwd`
- `docker run -ti alpine /bin/sh`



## Container Lifecycle



Laufende Container können gestoppt und anschließend wieder gestartet werden. Alle Dateien, die in dem Container-Layer geschrieben wurden, bleiben dabei erhalten. `stop` und `kill` unterscheiden sich darin, mit welchem Befehl der Prozess beendet wird. Im Falle von `pause` wird der Prozess eingefroren und kann über `unpause` fortgesetzt werden. Auch ein Beenden des Hauptprozesses führt dazu, dass der Container gestoppt wird.

Befehle und damit verbundene Signale:

- `pause`: SIGSTOP
- `stop`: SIGTERM und nach Grace Period SIGKILL
- `kill`: SIGKILL

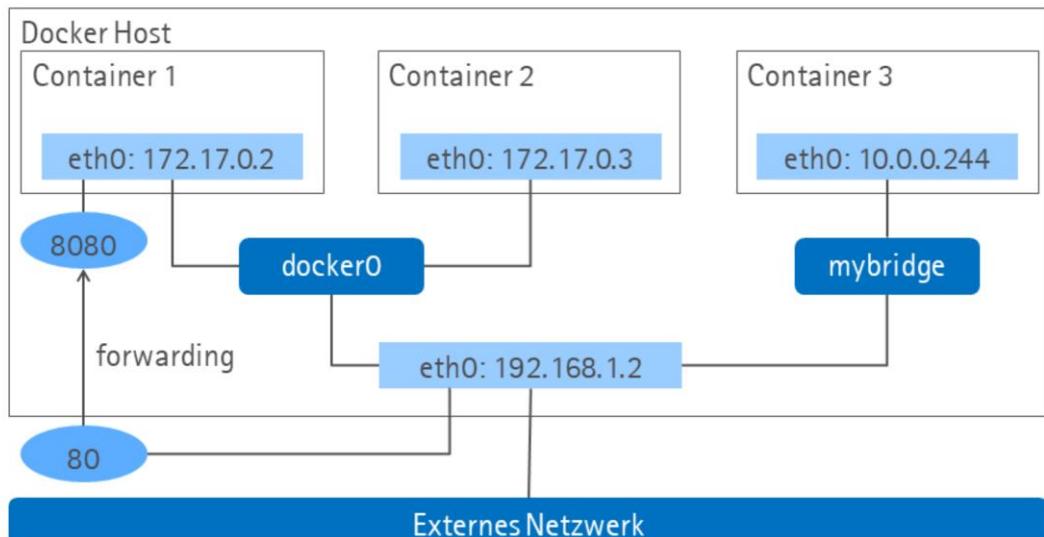


### Docker-Befehle:

- `docker run -ti -d --name background alpine /bin/sh`
- `docker container stop background`
- `docker container start background`
- `docker container rm background`



## Netzwerk-Zugriff auf Anwendung



Es gibt in Docker unterschiedliche Arten von Netzwerken, welche durch entsprechende Networkdriver implementiert werden.

- Host-Network
  - Nutzt den Networkstack des Dockerhosts
- Bridge-Network (Default)
  - Virtuelles Netzwerk auf dem Dockerhost
  - Neue Netzwerke können angelegt und Container diesen zugeordnet werden
  - Defaultmäßig wird docker0 als Bridge-Network verwendet
- Overlay-Network:
  - Verteiltes Netzwerk über mehrere Hosts hinweg

Dargestellt auf der Folie ist ein Networking mit dem Bridge-Driver. Dabei sind zwei Container dem Default-Netzwerk docker0 zugeordnet und einer dem zusätzlich angelegtem mybridge. Die Container im selben Netzwerk können direkt miteinander kommunizieren. Um auf den Container von außerhalb zugreifen zu können, kann mit einem Portforwarding auf dem Host gearbeitet werden:

```
docker run -p hostPort:containerPort image
```



### Docker-Befehle:

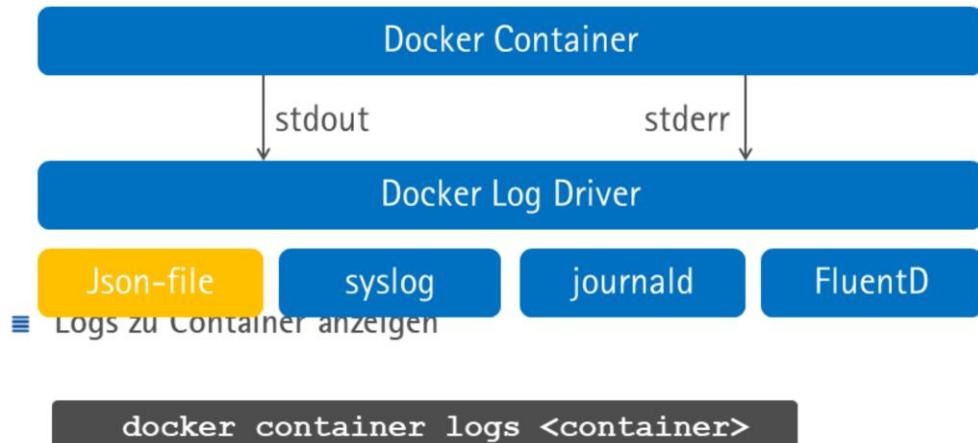
- `docker run -d -p 8080:80 \ --name nginx-hello nginxdemos/hello:plain-text`

Hiermit wird ein NGINX-Server gestartet, der auf dem Port 8080 einige Request-Daten ausliefert (z. B. mit `curl localhost:8080` abrufbar)



## Docker Container Logs

### Logging Architektur



Anwendungen in Docker-Containern loggen ihre Ausgaben in der Regel nicht in eine Datei, sondern geben diese auf stdout/stderr aus. Von dort werden die Ausgaben von der Docker Runtime mit einem Logging-Driver abgeholt und weitergeleitet. Logdriver können in der Datei `daemon.json` konfiguriert werden.

Der Default Logdriver ist `json-file`, welcher die Logeinträge in Json-Form in einem Logfile auf dem Dockerhost ablegt. Die Einträge enthalten neben der Logausgabe der Anwendung selber noch die Quelle (stdout/stderr) und einen Timestamp:

```
{ "log": "Log line is here\n",
  "stream": "stdout",
  "time": "2019-01-01T11:11:11.11111111Z" }
```

Logs können über den Befehl `docker container logs containerName` abgerufen werden. Mit dem Parameter `-f` (bzw. `--follow`) kann man einem Log folgen und so neue Einträge direkt sehen, wenn sie erzeugt werden. Mit `-tail` kann eingegrenzt werden, wie viele Logeinträge, vom Ende des Logs aus gerechnet, angezeigt werden sollen.



### Docker Befehle:

- `docker container logs nginx-hello`
- `docker container logs -f nginx-hello`



## Übung DOCKER\_BASICS\_01

- Starten Sie einen WildFly-Server mit Docker
  - Image-Tag:  
`quay.io/wildfly/wildfly:26.0.0.Final`
  - Im Hintergrund starten
- Im Browser sollte die Startseite aufrufbar sein
  - Im Container öffnet der WildFly den Port 8080
- Schauen Sie sich die Logs zu dem Container an

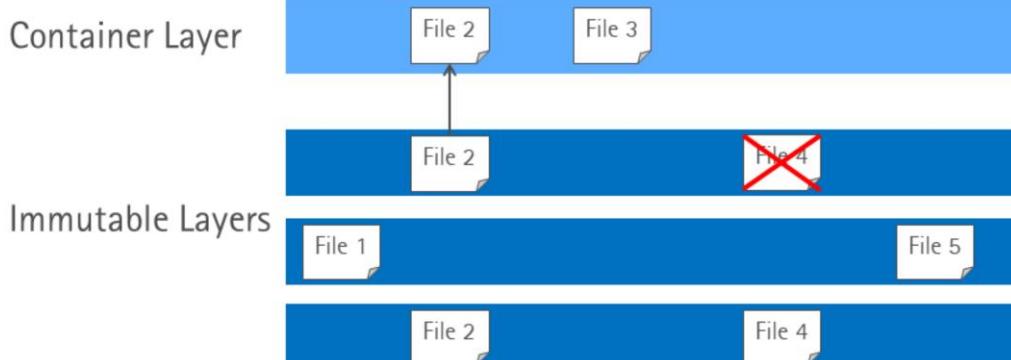


1. Nutzen Sie `docker run`, um ein WildFly-Image zu starten
  - a) Auf `quay.io` sind WildFly-Images namens `wildfly/wildfly` verfügbar. Nutzen Sie die Version `26.0.0.Final`.
  - b) Geben Sie `docker run` die Option für den Start im Hintergrund mit.
  - c) Geben Sie eine passende Option für den exponierten Port 8080 mit.
2. Nachdem der Container gestartet wurde, können Sie seine Startseite mit einem Browser auf `localhost` unter dem von Ihnen genutzten Port erreichen.
3. Nutzen Sie `docker logs` zum Betrachten des Server-Logs.
4. Stoppen und Löschen Sie den Container.



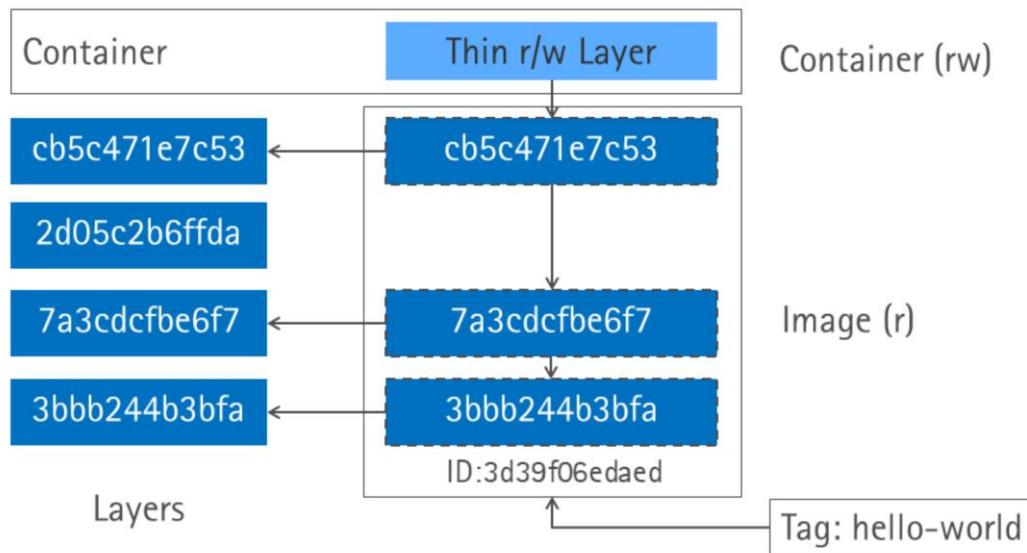
## Union File System

- ☰ Zusammengesetzte Sicht durch Folge von Layers





## Images und Layers



Images stellen ein Dateisystem dar, welches sich aus verschiedenen Schichten zusammensetzt.

Einzelne Layer enthalten dabei immer nur den Diff, der auf den jeweils vorhergehenden Layer angewendet wird. Insgesamt kommt so eine unifizierte Sicht heraus, welche sich aus dem Anwenden der Diffs in der entsprechenden Reihenfolge ergibt.

Seit der Version 1.10 sind Layer und Images nicht mehr das gleiche. Für die Layer wird nun ein eindeutiger inhaltsbasierter Hash generiert, was die Wiederverwendbarkeit der Layer deutlich erhöht. Images referenzieren jetzt nicht mehr direkt Basisimages, sondern eine Reihe von Layern. Dadurch müssen nun auch nicht mehr die Metadaten von Basisimages vorhanden sein. Mehrere Images können dieselben Layer verwenden, falls inhaltsgleiche Diffs enthalten sind.

Die Layer, die von den Images verwendet werden, sind nicht veränderbar, können also nur gelesen werden. Jeder Container, der auf Basis eines Images gestartet wird, bekommt seinen eigenen RW-Layer, in welchem er auch Dateiänderungen vornehmen kann. Dadurch ist es möglich, dass sich viele Container ein einziges Image als Dateisystem teilen.

Images haben ebenso einen Hash als ID, der auch auf Basis des Inhalts (enthaltene Layer) ermittelt wird. Um Images sprechend zu benennen, können diesen Tag-Namen zugeordnet werden.



### Docker-Befehle:

- docker image ls
- docker image history *imageTagName*
- docker image inspect *imageTagName*



## Dockerfile

- Definiert Buildschritte für Erstellung eines Images

FROM <image-tag>	Basisimage
COPY <context-src-path> <target-ctr-path>	Datei auf Image kopieren
ADD <context-src-path/url> <target-ctr-path>	Datei dem Image hinzufügen
RUN [<command>]	Befehl ausführen
ENTRYPOINT [<command>, <param>...]	Command für Containerstart
CMD [<command>, <param>...]	Angehängt an Entrypoint

Dockerfiles definieren die Buildschritte, die für das Erstellen eines neuen Images notwendig sind. Alle Images müssen ein Basisimage angeben und diesem dann weitere Dateien hinzufügen bzw. mit Kommandozeilenbefehlen konfigurieren.

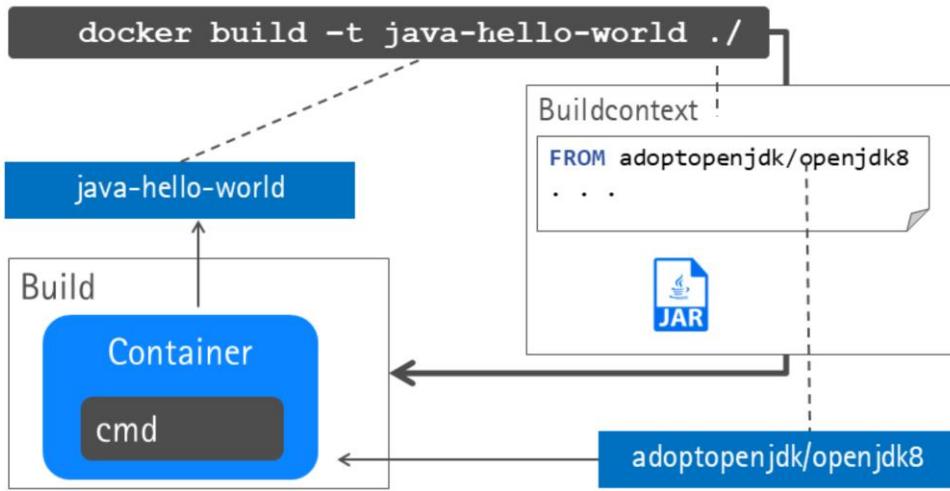
Als Basisimage werden in der Regel Images von Linux-Distributionen verwendet oder auch „leere“ Images wie scratch oder distroless.

Grundlegende Befehle:

- FROM: Definiert das Basisimage
- COPY: Kopiert Dateien auf das Image
- ADD: Ähnlich wie COPY aber noch etwas weiter gefasst, kann z. B. auch Dateien von URLs herunterladen und auf das Image kopieren. Empfehlung ist, für das einfache Kopieren von Dateien eher COPY zu verwenden.
- RUN: Konfiguriert das Image durch Ausführen eines Kommandozeilenbefehls. Hier können z. B. Pakete installiert, Dateien heruntergeladen oder User angelegt werden.
- ENTRYPOINT: Definiert den Einstiegspunkt für den Hauptprozess, der bei Starten des Containers ausgeführt werden soll. Kann über --entrypoint bei docker run übersteuert werden.
- CMD: Wird dem Befehl in ENTRYPOINT angehängt. Falls ENTRYPOINT leer ist, ist dies also der Einstiegspunkt. Wird bei docker run durch die zum Schluss angehängten Parameter überschrieben. In der Regel sollte ENTRYPOINT für den Hauptprozess genutzt werden und CMD für Defaultparameter, welche dann einfach beim run überschrieben werden können.



## Docker Build



Der Build eines neuen Images wird über den Befehl `docker build` ausgelöst. Hier können ein Dockerfile und ein Buildcontext angegeben werden. Per Default wird für das Dockerfile eine Datei namens `Dockerfile` im angegebenen Buildcontext gesucht. Der Name und Ort des Dockerfiles können über den Parameter `-f` übersteuert werden.

Der Tag des neuen Images wird über den Parameter `-t` festgelegt. Tags setzen sich in der Regel folgendermaßen zusammen:

`[registry/] [library/] name[:version]`

Während des Builds werden temporäre Docker Container auf Basis des `FROM`-Images gestartet, in welchen dann die Anweisungen aus dem Dockerfile ausgeführt werden. Daraus entstehen dann die neuen Layer, welche, über die Layer des Basisimages gelegt, das neue Image ergeben. Dabei ist zu beachten, dass jeder Befehl im Dockerfile zur Erzeugung eines neuen Layers führt.

Projekt `ctr-demo-hello`

Dockerfile:

- `Dockerfile-01-public`

Befehle:

- `mvn package`
  - `docker build -t gedoplan-seminar/ctr-demo-hello \ -f Dockerfile-01-public .`
  - `docker run --rm gedoplan-seminar/ctr-demo-hello`
- (Die Docker-Befehle befinden sich im Projektverzeichnis `src/test/script`)



## Übung DOCKER\_BASICS\_02

- Erstellen Sie ein **Dockerfile** für die Webanwendung **ctr-exercise-rest**
  - Basisimage:  
**quay.io/wildfly/wildfly:26.0.0.Final**
    - Enthält bereits Entrypoint zum Start des Servers
    - Deploy folder: /opt/jboss/wildfly/standalone/deployments
  - Bauen Sie das Anwendungsimage mit dem **build**-Befehl
  - Starten Sie die Anwendung als Container



Projekt **ctr-exercise-rest**

1. Das Übungsprojekt **ctr-exercise-rest** erstellt die gleichnamige Webanwendung, d. h. nach einem `mvn clean package` befindet sich im Folder `target` die gebaute Anwendung als `.war`-File.

Die Anwendung läuft auf einem JEE-Server – wir nutzen hier WildFly.

Achtung: Die Anwendung referenziert in `src/main/resources/META-INF/persistence.xml` eine Datasource mit Hilfe eines Ausdrucks, der eine spezielle Konfiguration des WildFly benötigt, die erst in einer der nächsten Übungen eingestellt wird. Kommentieren Sie bis dahin bitte die Zeile `<jta-data-source>...</jta-data-source>` aus.

2. Das Projekt enthält noch kein Dockerfile. Erstellen Sie es:
  - Nutzen Sie als Basisimage `quay.io/wildfly/wildfly:26.0.0.Final`.
  - Lassen Sie die gebaute Anwendung in das Deployment-Verzeichnis `/opt/jboss/wildfly/standalone/deployments` kopieren.
3. Bauen Sie das Image und starten Sie es als Container
  - im Hintergrund,
  - mit Mapping für den Port 8080.
4. Prüfen Sie, ob die Anwendung funktioniert, z. B. mit einem Request auf den Endpunkt `/resources/hello`.
5. Stoppen Sie den Container.

Musterlösung: Dockerfile-01-public sowie Skripte in `src/test/script`



## Basisimages

- Basisimages von Docker Hub bequem, aber:
  - Keine beliebige Kombination
    - z. B. Debian + AdoptOpenJDK 8ux+ WildFly 15
  - Evtl. nicht Projektanforderungen entsprechend



- Eigenes Basisimage definieren

Basisimages von Docker Hub sind ideal, um schnell eine Anwendung mit Docker an den Start zu bringen. Dennoch kann häufig das Erstellen eines eigenen Basisimages sinnvoll sein. Hier hat man die Möglichkeit, das Image nach den eigenen Anforderungen zu gestalten und zum Beispiel die gewünschte Kombination aus Distribution, Tools, JDK und Application Server herzustellen.

Auch was Sicherheit anbelangt, sind nicht alle Images auf Docker Hub ideal konfiguriert – auch aus diesem Grund bietet sich oft ein eigenes Basisimage an.

Es wäre natürlich auch möglich, auf ein eigenes Basisimage zu verzichten und alles im Image der Anwendung zu konfigurieren, allerdings ist dieser Ansatz nicht zu empfehlen, da hier der Build der Anwendung komplexer und langsamer wird und natürlich auch die Wiederverwendung von Images über Anwendungen hinweg nicht möglich ist.



Projekte ctr-demo-jdk und ctr-demo-hello

### Dockerfile:

- In ctr-demo-jdk: Dockerfile-01-simple
- In ctr-demo-hello: Dockerfile-02-custom

### Befehle:

- In ctr-demo-jdk:

```
docker build -f Dockerfile-01-simple \
             -t gedoplan-seminar/ctr-demo-jdk .
```
- In ctr-demo-hello:

```
docker build -f Dockerfile-02-custom \
             -t gedoplan-seminar/ctr-demo-hello .
docker run --rm gedoplan-seminar/ctr-demo-hello
```

(Die Docker-Befehle befinden sich in den Projektverzeichnissen src/test/script)



## Übung DOCKER\_BASICS\_03

- Erstellen Sie ein eigenes Wildfly-Basisimage
  - Tag: gedoplan-seminar/ctr-exercise-wildfly
  - Basis: adoptopenjdk/openjdk11:jdk-11.0.3.7
- Stellen Sie die Anwendung **ctr-exercise-rest** auf das neue Basisimage um



Projekte **ctr-exercise-wildfly** und **ctr-exercise-rest**

### Anleitung:

1. Erstellen Sie im Projekt **ctr-exercise-wildfly** ein Dockerfile:
  - a) Nutzen Sie das Basisimage `adoptopenjdk/openjdk11:jdk-11.0.3.7`.
  - b) Für das Herunterladen des Wildfly wird das Werkzeug `curl` benötigt. Installieren Sie dies mit dem Befehl: `apt-get install -y curl`.

Sie finden diesen und die nächsten Befehle im File

`src/main/snippets/download_wildfly.txt` des Projektes **ctr-exercise-wildfly**.

- c) Der Wildfly in der Version 26.0.0.Final sollte als `.tar.gz` heruntergeladen werden (der Link ist auf der Webseite [wildfly.org](http://wildfly.org) zu finden). Entpacken Sie den WildFly in den Ordner `/opt/jboss`. Sie können eine Befehlssequenz wie diese nutzen (Download-URL anpassen):

```
mkdir -p /opt/jboss
cd /opt/jboss
curl url-to-tar.gz | tar xzf -
mv wildfly-26.0.0.Final wildfly
```

Fortsetzung auf der nächsten Seite



Fortsetzung:

- d) Definieren Sie einen Entrypoint, der den Wildfly startet und auf Interface 0.0.0.0 bindet:  
`/opt/jboss/wildfly/bin/standalone.sh -b 0.0.0.0`
2. Bauen Sie das neue Basisimage mit dem Tag  
`gedoplan-seminar/ctr-exercise-wildfly`
  3. Setzen Sie im Projekt `ctr-exercise-rest` das neue Basisimage ein.
  4. Bauen Sie das Anwendungs-Image neu und starten Sie es, um zu prüfen, ob seine Funktionalität erhalten geblieben ist. Der Rest-Endpoint `/resources/systemInfo` sollte jetzt als Java-Vendor AdoptOpenJDK anzeigen und als Os-User root.

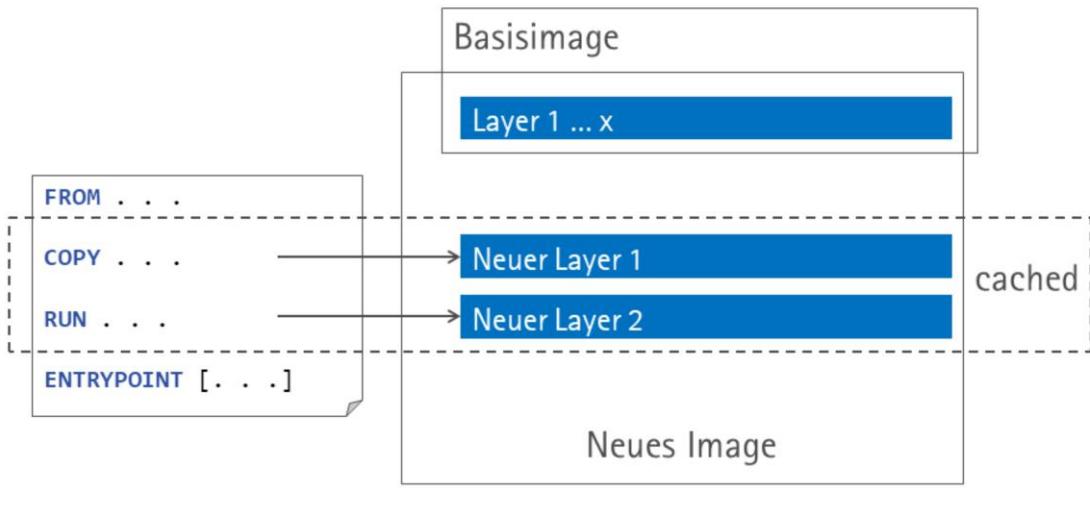
Stoppen Sie den Container.

Musterlösung:

- In `ctr-exercise-wildfly`: `Dockerfile-02`
- In `ctr-exercise-rest`: `Dockerfile-02-custom`
- Docker-Befehle in beiden Projekten in `src/test/script`



## Docker Build Layer und Caching



Die Anweisungen im Dockerfile, welche das Dateisystem verändern, wie `COPY`, `ADD` und `RUN` führen jeweils zu der Erstellung eines neuen Layers.

Es wird beim Bauen des Images ein Cache aufgebaut, welcher die Befehle (bei `RUN`) oder die Inhalte (bei `ADD/COPY`) als Key den erzeugten Layern zuordnet. Dadurch ist es möglich, dass bei einem erneuten Build nur die Layer neu erzeugt werden müssen, deren Bauanweisungen sich verändert haben und die darauf folgen. Somit laufen wiederholte Builds in der Regel schneller ab, weil unveränderte Layer aus dem Cache bezogen werden.

Dies bedeutet aber auch, dass es nicht möglich ist, die Image-Größe zu reduzieren, indem in einem nachgelagertem `RUN` Dateien gelöscht werden, die nicht mehr gebraucht werden. Das liegt daran, dass dies ja nur einen neuen Layer erzeugt, welcher die Informationen über das Entfernen enthält; im darunterliegendem Layer sind die Files noch vorhanden.

Es besteht die Möglichkeit, mehrere Befehle zu verketteten, sodass man diese in einem `RUN` im Dockerfile definieren kann, dies führt dann entsprechend nur zu einem einzigen Layer für die verketteten Befehle:

```
RUN cmd1 && cmd2 && \
cmd3
```



## Optimieren von Images

- ☰ Manche Tools nur für Build benötigt

**curl**//

**Maven™**

**yum**  
yellowdog updater modified



- ☰ Unnötig große Images + Sicherheitsrisiko



- ☰ Minimales Basisimage + Entfernen der Buildwerkzeuge

Viele Basisimages sind unnötig groß, enthalten viele Werkzeuge, die nicht (mehr) benötigt werden und haben ein entsprechend großes Dateisystem. Darüber hinaus besteht auch noch das Problem, dass manche Werkzeuge zwar gebraucht werden, aber nur für den Build. Dies trifft zum Beispiel auf Package Manager, Programme wie `curl` oder auch Buildtools wie `Maven` zu.

Diese Werkzeuge machen das Image nicht nur unnötig groß, sondern stellen auch ein Sicherheitsrisiko dar, da nun ein Angreifer, der sich zu dem Container Zugriff verschafft hat (z. B. über eine darin laufende Webanwendung) nun diese Werkzeuge nutzen kann.

Daher empfiehlt es sich, die Images so schlank wie möglich zu halten, sodass neben der Anwendung nur die benötigten Bibliotheken und Tools darin enthalten sind.

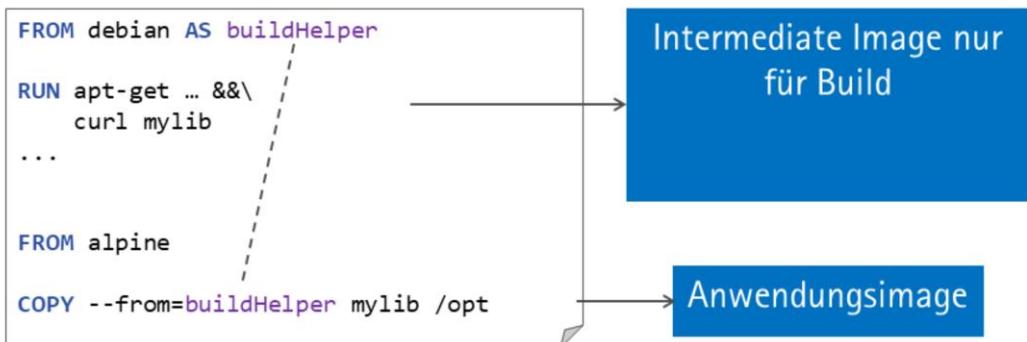
Dafür können zum einen minimale Distributionen als Basisimage genutzt werden wie z. B. `busybox` oder `alpine` oder sogar Images ohne Distribution wie `scratch` oder `distroless` von Google.

Darüber hinaus sollte beim Build aufgeräumt werden, wo man aber bedenken muss, dass dies nicht in nachgelagerten Schritten geschehen kann. Hier könnte man dann versuchen, die Befehle für das Entfernen mit an einen bestehenden `RUN` anzuhängen. Allerdings ist diese Vorgehensweise problematisch, weil man dazu ja erstmal analysieren müsste, was überhaupt alles zu entfernen ist.



## Multi-Stage-Build

- ☰ Zwischenimages durch mehrere **FROM**
  - ☰ Files können von Images kopiert werden



Mit Hilfe von Multistage-Builds können Zwischenimages für den Build definiert werden. Diese können Ergebnisse produzieren (Build, Download usw.), welche dann anschließend auf das Anwendungsimage kopiert werden können. Dies ermöglicht es, das eigentliche Ergebnis des Dockerbuilds, also das Anwendungsimage, entsprechend schlank zu gestalten.

Referenziert werden Zwischenimages beim `COPY` mit `--from` entweder über die Nummer des Images (Reihenfolge im Dockerfile beginnend mit 0) oder über einen mit `AS` vergebenen Bezeichner.

Es ist auch möglich, Dateien nicht von Zwischenimages des gleichen Builds, sondern von beliebigen anderen Images zu kopieren. Dafür muss im `--from` einfach das entsprechende Tag des gewünschten Images angegeben werden. Dies kann sinnvoll sein, da bei Dateien, die direkt aus dem Internet heruntergeladen werden, immer die Gefahr besteht, dass sich URLs im Laufe der Zeit ändern.

Projekt `ctr-demo-jdk`

Dockerfile:

- `Dockerfile-02-multistage`

Befehle:

- `docker build -f Dockerfile-02-multistage \ -t gedoplan-seminar/ctr-demo-jdk .`

Zum Test kann nun das Image `ctr-demo-hello` neu gebaut und gestartet werden. Seine Funktionalität ist unverändert, aber es basiert nun auf einem kleineren Basisimage.



## Übung DOCKER\_BASICS\_04

- Optimieren Sie den Build für das WildFly-Image
  - Schlankes Basisimage – `adoptopenjdk11 alpine`
  - Galleon zum Provisionieren des WildFlys
  - Multistage-Build
    - Provisionieren in Buildhelper
    - Ergebnis auf echtes Image kopieren
- Zusatzaufgabe:
  - Erstellen Sie für den Builder ein separates Galleon-Image
    - Wildfly-Abhängigkeiten bereits laden

Projekte `ctr-exercise-wildfly` und `ctr-exercise-rest`

### Anleitung:

1. Galleon ist ein Tool, mit dem WildFly-Servere provisioniert werden können. Schreiben Sie das Dockerfile im Projekt `ctr-exercise-wildfly` so um, dass es in einem Multistage-Build zunächst Galleon herunterlädt und damit einen WildFly-Server provisioniert, um dann im zweiten Step ebendiesen WildFly-Server mit einem Basisimage zu kombinieren:
  - a) Nutzen Sie für den Build Helper das Basisimage `adoptopenjdk/openjdk11:jdk-11.0.3.7`.
  - b) Lassen Sie mit `apt-get install -y curl unzip` die benötigten Zusatztools installieren. Zuvor wird ein `apt-get update` benötigt.

Die Befehle können Sie wieder kopieren, und zwar aus  
`src/main/snippets/download_galleon.txt` des Projekts `ctr-exercise-wildfly`.

- c) Lassen Sie Galleon von Github herunterladen:

```
curl -Ls https://github.com/wildfly/
       galleon/releases/download/
           4.2.5.Final/galleon-4.2.5.Final.zip
       -o /tmp/galleon.zip
```
- d) Entpacken Sie Galleon nach `/galleon`:

```
unzip /tmp/galleon.zip
       mv /galleon-4.2.5.Final /galleon
```

Fortsetzung auf der nächsten Seite



## Anleitung - Fortsetzung:

- e) Zur Provisionierung des WildFly-Servers kann Galleon eine Parameterdatei mitgegeben werden. Diese ist im Projekt bereits vorhanden: `provisioning.xml`. Lassen Sie diese Datei in das Image kopieren (z. B. nach `/tmp/provisioning.xml`).
- f) Nutzen Sie Galleon mit der Parameterdatei, um einen WildFly-Server in `/tmp/wildfly` abzulegen:  

```
/galleon/bin/galleon.sh provision \
/tmp/provisioning.xml --dir=/tmp/wildfly
```
- g) In einem zweiten Build-Schritt nutzen Sie das kleine Basisimage  
`adoptopenjdk/openjdk11:jdk-11.0.3_7-alpine-slim`.
- h) Legen Sie das Verzeichnis `/opt/jboss` an und kopieren Sie den WildFly-Server aus dem ersten Build-Schritt dort hinein.
- i) Bauen Sie das Image `ctr-exercise-wildfly`, dann `ctr-exercise-rest`, und testen Sie die Anwendung.

## Musterlösung:

- Dockerfile-03 im Projekt `ctr-exercise-wildfly`.
- Dockerfile-02-custom im Projekt `ctr-exercise-rest` (unverändert).

2. Zusatzaufgabe: Galleon lädt bei einem Build stets den gesamten WildFly herunter. Dabei wird zwar ein Cache aufgebaut, der aber wieder verloren geht, wenn der entsprechende Container – der erste des Multistage-Builds – terminiert. Um dies zu vermeiden, kann ein separates Basisimage für den ersten Build-Schritt erstellt werden, das Galleon inkl. des bereits heruntergeladenen WildFly-Servers enthält:

- a) Das Übungsprojekt `ctr-exercise-galleon` existiert bereits. Ergänzen Sie ein Dockerfile, das die ersten Anweisungen des ersten Build-Schrittes des oben gebauten Dockerfiles enthält: Basisimage, Installation von `curl` und `unzip`, Herunterladen und Entpacken von Galleon.
- b) Lassen Sie nun Galleon einen vollständigen WildFly-Server ohne spezielle Provisionierung installieren:  

```
/galleon/bin/galleon.sh \
install wildfly:current#26.0.0.Final \
--dir=/tmp/download
```

Den heruntergeladenen Server können Sie gerne direkt wieder löschen (`rm -fr /tmp/download`), denn wichtig ist nur der nebenbei gefüllte Download-Cache.
- c) Nutzen Sie das soeben erstellte Image als Basisimage des ersten Build-Schrittes. Es ersetzt die Anweisungen bis zum Entpacken von Galleon.

## Musterlösung:

- Dockerfile im Projekt `ctr-exercise-galleon`.
- Dockerfile-04 im Projekt `ctr-exercise-wildfly`.
- Dockerfile-02-custom im Projekt `ctr-exercise-rest` (unverändert).



## Images mit Nicht-Root-User

- Root-User im Container birgt Gefahren



- Images mit eingeschränktem User erstellen

```
FROM ...
RUN adduser -h /opt/java -D java
...
USER java
```

User mit Gruppe anlegen

User für Prozess festlegen

Standardmäßig laufen Container-Prozesse mit `root` als User. Dies bringt eine Reihe von Sicherheitsrisiken mit sich. Zum einen könnte nun jemand, der Zugriff auf den Container hat, innerhalb des Containers beliebige Kommandos ausführen, zum anderen ist es auch so, dass der `root` im Container dieselbe User Id hat wie `root` auf dem Host. Es wäre also möglich, innerhalb des Containers gemountete Hostverzeichnisse zu sehen, auch wenn diese auf dem Host nur für `root` zugänglich sind.

Soll ein anderer User verwendet werden, so muss dieser zunächst angelegt werden. Dafür können Befehle wie `adduser` oder `useradd` verwendet werden. In Alpine kann z. B. ein User `java` mit der dazugehörigen Gruppe `java` und einem Homeverzeichnis `/opt/java` mit folgendem Befehl angelegt werden:

```
adduser -h /opt/java -D java.
```

Über die Anweisung `USER` kann der User festgelegt werden, unter dem weitere Befehle im Dockercontainer ausgeführt werden sollen. Es besteht zudem die Möglichkeit, einen User direkt im `run`-Befehl über den Parameter `--user` mitzugeben.

Für Befehle wie `COPY`, die Dateien kopieren, gibt es die Möglichkeit, einen User anzugeben, dem die kopierten Dateien zugeordnet werden sollen.



Projekt `ctr-demo-jdk`

Dockerfile:

- Dockerfile-03-user

Befehle:

- docker build -f Dockerfile-03-user \
-t gedoplan-seminar/ctr-demo-jdk .

Zum Test kann nun erneut das Image `ctr-demo-hello` gebaut und gestartet werden. Der interne User ist nun `java`.



## Übung DOCKER\_BASICS\_05

- Der WildFly soll unter einem eigenen User `jboss` laufen



Projekt `ctr-exercise-wildfly`

### Anleitung:

- User können unter Alpine mit `adduser` angelegt werden. Die Option `-h` legt ein Homeverzeichnis an und `-D` bedeutet "ohne Passwort", sodass der user `jboss` mit seinem Homeverzeichnis `/opt/jboss` so angelegt werden kann:  
`adduser -h /opt/jboss -D jboss`
- Würde man das Image nun mit diesem User starten, so stellt man fest, dass der WildFly keine Berechtigung hat, Log-Dateien in seinem eigenen Verzeichnis zu schreiben. Dies liegt daran, dass die vom Buildhelper kopierte Verzeichnisstruktur keine Berechtigung für den User oder die Gruppe `jboss` konfiguriert hat. Um das kopierte Verzeichnis dem User zuzuordnen kann ein entsprechender Parameter beim `COPY`-Befehl mitangegeben werden: `--chown=jboss:jboss`
- Sorgen Sie dafür, dass der Container zur Laufzeit ebenfalls den User `jboss` nutzt.
- Bauen Sie mit diesen Änderungen das Image `ctr-exercise-wildfly`, danach erneut `ctr-exercise-rest`, und testen Sie die Anwendung. Ein Rest Call auf `/resources/systemInfo` muss nun den User `jboss` ausweisen.

### Musterlösung:

- Dockerfile-05 im Projekt `ctr-exercise-wildfly`.
- Dockerfile-02-custom im Projekt `ctr-exercise-rest` (unverändert).



## Fabric8 Docker Maven Plugin

- Ermöglicht Anstoßen des Builds über Maven
- Mit Verwendung eines Dockerfiles



```
<image>
  <name>gedoplan-seminar/ctr-demo-rest</name>
  <build>
    <contextDir>${project.basedir}</contextDir>
```

```
<build>
  <assembly>...</assembly>
  <cmd>java -jar /opt/java/myapp.jar</cmd>
```

Im Falle von Java-Anwendungen, die mit Maven gebaut werden, kann der Docker-Build auch über Maven heraus aufgerufen werden.

Mit dem `fabric8-maven-plugin` ist es möglich, entweder nur einen normalen Build mit `Dockerfile` aufzurufen oder den Build komplett in der `pom.xml` zu definieren.

Falls die Variante mit `Dockerfile` gewählt wird, so muss eventuell vorher noch ein Fat Jar gebaut oder Bibliotheken in einem Verzeichnis bereitgestellt werden. Für das Sammeln von Bibliotheken kann das `maven-dependency-plugin` verwendet werden.

Wird ein Build ohne `Dockerfile` durchgeführt, so kopiert das Plugin standardmäßig das erzeugte Artefakt (`jar/war`) sowie die Jars, die sich aus den Maven-Dependencies ergeben, auf das Image. Über das `assembly`-Element kann dies umkonfiguriert werden.

Projekt `ctr-demo-rest`

Plugins:

- In `pom.xml` sind die Profile `docker-f8-inline` und `docker` enthalten
- Befehle:
  - `mvn package -Pdocker-f8-inline` für den Build ohne `Dockerfile`
  - `mvn package -Pdocker` für den Build mit `Dockerfile`  
(das zu nutzende `Dockerfile` kann bspw. mit `-DdockerFile=Dockerfile-02-custom` angegeben werden)



## Jib Docker Maven Plugin

- Baut Images ohne Docker-Daemon
- Flat-Classpath-Anwendungen
- Schnelle Builds
- Separiert Anwendung von Bibliotheken



Containerize your Java application.

```
<configuration>
  <to>
    <image>gedoplan-seminar/ctr-demo-rest</image>
  </to>
  <container>
    <mainClass>com.kumuluz.ee.EeApplication</mainClass>
```

Mit dem Jib Plugin ist es möglich, Images per Maven zu bauen, ohne dabei Zugriff auf einen Docker-Deamon zu haben. Dies ist vor allem für CI-Umgebungen interessant. Die gebauten Images können entweder zum Schluss direkt in eine Registry gepusht oder an einen Dockerhost übertragen werden.

Mit Jib können keine beliebigen Anwendungen gebaut werden. Anfänglich war es nur möglich, Flat-Classpath-Anwendungen zu bauen, mittlerweile wird auch ein Build von war-Archiven unterstützt.

Jib ermöglicht schnelle Buildzeiten, da die Anwendung von den Abhängigkeiten separiert wird. Da keine gebauten Java-Archive, sondern nur die geänderten .class-Dateien und Ressourcen übertragen werden, erfolgt ein Rebuild des Images in entsprechend kurzer Zeit.

Standardmäßig nutzt Jib das distroless-java-Image als Basis, was allerdings bei Bedarf umkonfiguriert werden kann.



Projekt ctr-demo-rest

Plugins:

- In pom.xml ist das Profile docker-jib enthalten

Befehle:

- mvn package -Pdocker-jib



## Build in Container

### Problemstellung:

Dockerhost



Jenkins



### Lösungen

- Daemon des Hosts verwenden
- 

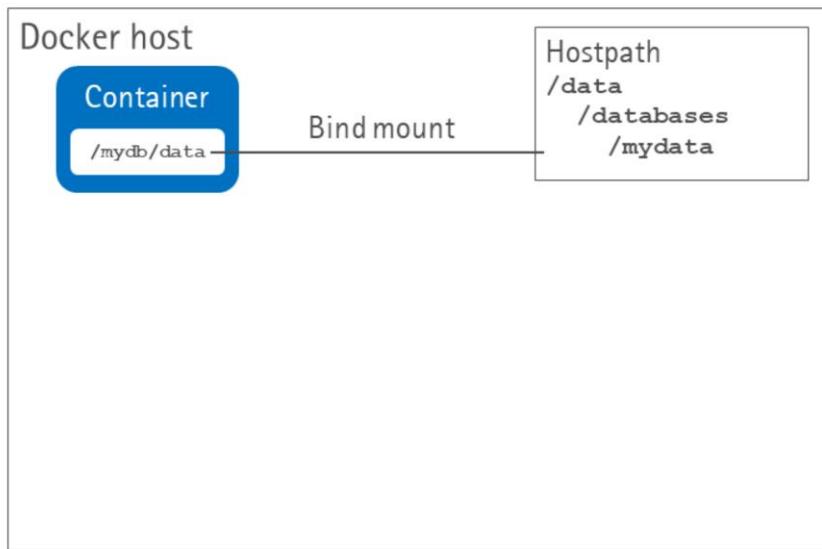


In einer CI-Umgebung kann es vorkommen, dass ein Build durchgeführt werden muss, ohne dass direkt auf einem Dockerhost gebaut wird. Dies ist z. B. dann der Fall, wenn die Schritte der CI-Pipeline selbst in Dockercontainern ausgeführt werden. Ein Build innerhalb eines Containers ist nicht ohne weiteres möglich, denn dieser müsste entweder mit privilegierten Rechten ausgeführt werden oder direkt so eingestellt werden, dass er den Daemon des Hosts verwendet (z. B. über ein Mounting des Sockets `/var/run/docker.sock`). Das liegt daran, dass die einzelnen Befehle ja innerhalb von Containern ausgeführt werden.

Soll der Build komplett unabhängig von einem Dockerhost innerhalb eines Containers erfolgen, so können hierfür Werkzeuge wie Kaniko, Buildah oder Jib verwendet werden.



## Persistente Daten: Bind Mounts



Daten, die in den Containerlayer geschrieben werden, gehen nach dem Entfernen des Containers verloren. Sollen diese erhalten bleiben, so können die Dateien bestimmter Containerpfade im Dateisystem des Docker Hosts abgelegt werden.

Ein sog. Bind Mount ermöglicht das Ablegen von Dateien in einem Verzeichnis auf dem Docker Host:

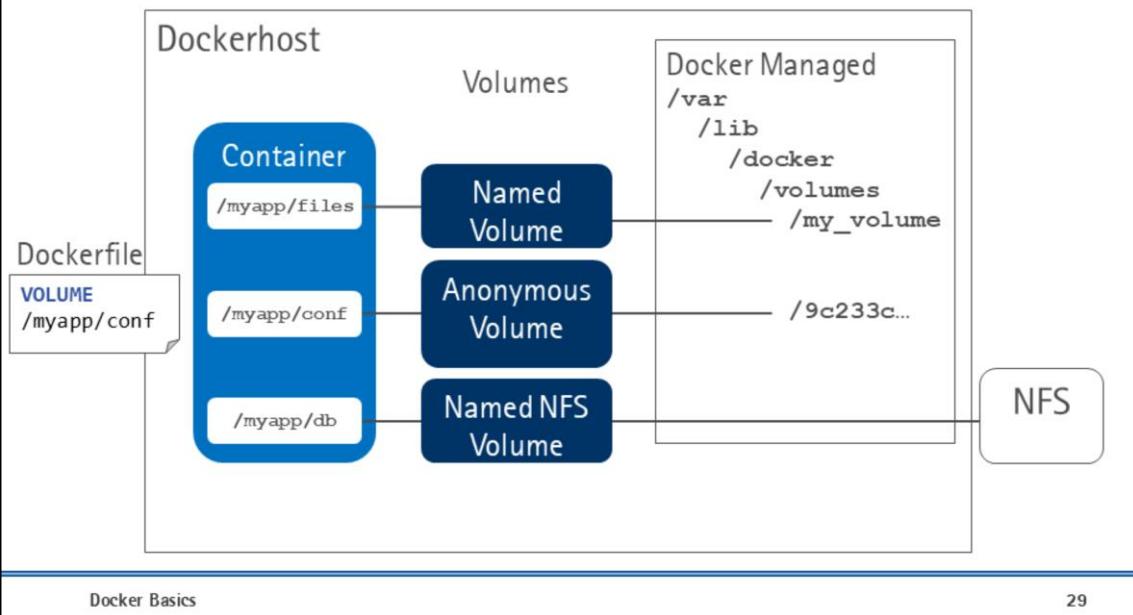
```
docker run -v hostpath:containerpath ...
```

Falls im Container schon Dateien im genutzten Verzeichnispfad liegen, werden diese durch den Bind Mount überdeckt.

Die Dateizugriffsrechte des Hostverzeichnisses müssen passend sein.



## Persistente Daten: Volumes



Statt Bind Mounts können auch ein sog. Volume genutzt werden. Sie werden über einen Namen angesprochen und können ebenfalls Verzeichnisse auf dem Host sein.

Über sog. Driver sind andere Speicherorte wie NFS oder diverse Cloudstorages möglich.

Volumes werden zunächst separat angelegt und dann über einen Namen referenziert:

```
docker volume create volume-name  
docker run -v volume-name:container-path
```

Die Verzeichnisse auf dem Host werden in diesem Fall von Docker verwaltet und z. B. unter Linux unterhalb von /var/lib/docker/volumes abgelegt. Dateien, die bereits im Dateisystem des Containers in dem Pfad liegen, werden in das Volume kopiert. Volumes sind besser wiederverwendbar, da die Referenzierung über den Namen erfolgt und der genaue Speicherort bei der Bindung nicht bekannt sein muss.

In Dockerfiles kann ein Volume über das Schlüsselwort `VOLUME` definiert werden. Wird der Container zu einem solchen Image gestartet ohne explizit ein Volume an diesen Pfad zu binden, so wird dafür ein anonymes Volume angelegt.



## 1. Container ohne Bind Mount / Volume starten:

```
docker run --rm -p 8080:8080 -d gedoplan-seminar/ctr-demo-rest
```

Neue Person anlegen:

```
curl -X POST http://localhost:8080/resources/personen \
-d '{"name":"hugo"}' -H 'content-type: application/json'
(Kommando ist in Projektdatei src/test/curl/postPerson.sh enthalten)
```

Alle Personen anzeigen:

```
curl http://localhost:8080/resources/personen
(Kommando ist in Projektdatei src/test/curl/getPersonen.sh enthalten)
```

Die Anwendung speichert ihre Daten im Ordner /opt/java/h2, der nach einem Stopp und Neustart des Containers wieder leer ist.

## 2. Ordner anlegen für Bind Mount, z. B. /tmp/ctr-demo-rest-h2

Achtung: Wenn Sie Docker Desktop für Windows nutzen, muss der Ordner in einem der lokalen Laufwerke liegen, die in Settings -> Resources -> File Sharing freigaben wurden. Das Laufwerk C: ist vom Docker Host aus als /c erreichbar.

Container mit Bind Mount starten:

```
docker run --rm -p 8080:8080 -d \
-v /tmp/ctr-demo-rest-h2:/opt/java/h2 \
gedoplan-seminar/ctr-demo-rest
```

Hinzugefügte Personen bleiben nun auch bei einem Neustart des Containers erhalten.

## 3. Volume anlegen: docker volume create ctr-demo-rest-h2

Achtung: Vielfach sind Volumes im Docker Host nur für root zugreifbar. Da unsere Demo-Anwendung mit der User java läuft, ist dann kein Zugriff erlaubt. Als Workaround können in einem solchen Fall mit dem folgenden Befehl die Zugriffsrechte für das Volume-Verzeichnis modifiziert werden:

```
docker run --rm -v ctr-demo-rest-h2:/opt/dummy \
alpine chmod 777 /opt/dummy
```

Container mit Volume starten:

```
docker run --rm -p 8080:8080 -d \
-v ctr-demo-rest-h2:/opt/java/h2 \
gedoplan-seminar/ctr-demo-rest
```

Hinzugefügte Personen bleiben auch jetzt bei einem Neustart des Containers erhalten.



## Übung DOCKER\_BASICS\_06

- ☰ Sorgen Sie dafür, dass die Einträge in der Datenbank auf dem Host persistiert werden.



Projekt `ctr-exercise-rest`

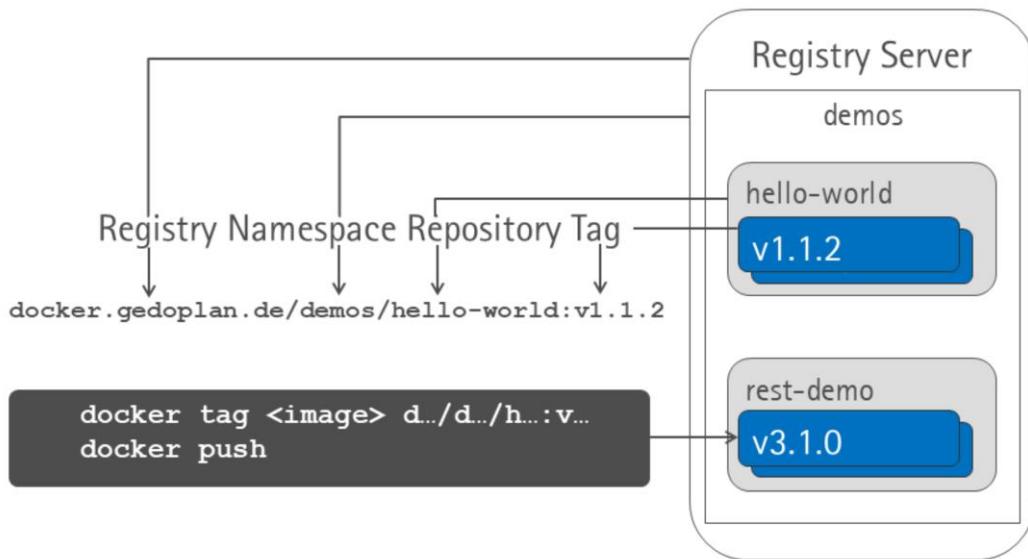
### Anleitung:

1. Die Anwendung `ctr-exercise-rest` bietet wie `ctr-demo-rest` ein Rest API zur Verwaltung von Personen an:
  - `curl http://localhost:8080/resources/personen` liefert alle Personen
  - `curl http://localhost:8080/resources/personen \ -X POST -d '{"name":"hugo"}' \ -H 'content-type: application/json'` fügt eine neue Person hinzu.  
(Die Kommandos sind in `src/test/curl` des Projekts `ctr-demo-rest` enthalten)
2. Die Personen werden in einer Datenbank gespeichert. Wir nutzen hier eine H2-Datenbank. Dazu muss beim Start des Containers eine Environment-Variable gesetzt werden:  
`docker run --env DATASOURCE=exerciseH2 ...`  
Achtung: Die Datasource-Referenz im Deskriptor `persistence.xml` darf nun nicht mehr auskommentiert sein, sonst wirkt die Environment-Variable nicht!
3. Die H2-Datenbank speichert ihre Daten im Verzeichnis `/opt/jboss/seminar` innerhalb des Containers. Verbinden Sie dieses Verzeichnis beim Containerstart mit einem Verzeichnis des Docker Hosts (z. B. `/tmp/ctr-exercise-rest-h2` für Linux oder `/c/seminar/ctr-exercise-rest-h2` für Windows).
4. Prüfen Sie, ob die Daten auch nach einem Neustart des Containers noch verfügbar sind.

Musterlösung: `src/test/script/docker_run_with_bindmount.sh`



## Images verteilen mit Registries



Docker Basics

32

Docker Images können über Registries zur Verfügung gestellt werden. Neben der Default-Registry Dockerhub (`hub.docker.com`) im Internet gibt es auch die Möglichkeit, eine eigene private Registry zu betreiben. Dafür gibt es eine ganze Reihe unterschiedlicher Softwareprodukte, z. B. mit Hilfe von Nexus der Fa. Sonatype.

Docker Registries können zugriffsbeschränkt sein. In diesem Fall ist ein Anmelden vom Dockerhost aus notwendig: `docker login registry-url`.

Das Zugangstoken wird auf dem Rechner in Json-Form abgelegt und erst bei einem expliziten `docker logout` wieder entfernt. Um weitere Hosts anzumelden, würde also ein Kopieren der Zugangstokens auf diese Rechner genügen.

Sofern der User entsprechende Rechte hat, können nun Images in die Registry gepusht werden. Der Imagename muss dazu so zusammengesetzt sein:

`registryurl/namespace/repository:tag`.

- `registryurl`: Adresse der Registry, Default `hub.docker.com`
- `namespace`: Kann beliebig oft vorkommen (bei einige Registries max. 1 mal)
- `repository`: Eigentlicher Name des Images
- `tag`: Version des Images, Default `latest`

Mit dem Befehl `docker push imagename` kann das Image dann an die Registry übertragen werden. Über `docker pull imagename` kann es nun auf einem anderen Dockerhost heruntergeladen werden.



Projekt ctr-demo-rest

Demo siehe nächste Seite!



## Übung DOCKER\_BASICS\_07

- Veröffentlichen Sie das Anwendungsimage in eine Registry



Projekt `ctr-exercise-rest`

Anleitung:

Gehen Sie für Ihr Image `ctr-exercise-rest` grundsätzlich so vor, wie unten für `ctr-demo-rest` gezeigt.

Damit sich die hochgeladenen Images verschiedener Seminarteilnehmer nicht überdecken, nutzen Sie in Abstimmung mit Ihrem Trainer und den anderen Kursteilnehmern einen neuen Namespace anstelle von `trainer`.



Projekt `ctr-demo-rest`

Demo zur vorigen Seite:

- Zuvor gebautes Image mit einem Tag passend zur Registry versehen:  
`docker tag gedoplan-seminar/ctr-demo-rest  
semidock.gedoplan.de/trainer/ctr-demo-rest`  
(natürlich hätte das Image auch direkt mit diesem Namen gebaut werden können)
- Anmeldung an der Registry:  
`docker login semidock.gedoplan.de  
(User: seminar, Passwort: seminar)`
- Image hochladen:  
`docker push semidock.gedoplan.de/trainer/ctr-demo-rest`
- Image lokal löschen und erneut herunterladen:  
`docker image rm semidock.gedoplan.de/trainer/ctr-demo-rest  
docker pull semidock.gedoplan.de/trainer/ctr-demo-rest`



## Aufräumen des Dockerhosts

### ☰ Images

**docker image prune**

### ☰ Container

**docker container prune**

### ☰ Images + Container + Buildcache + Netzwerke + Volumes

**docker system prune**

Standardmäßig werden auf dem Dockerhost alle Objekte wie Images, Container usw. aufbewahrt, auch wenn diese nicht mehr verwendet werden. Um solche ungenutzten Objekte aufzuräumen, kann mit dem Befehl `prune` gearbeitet werden:

`docker image prune`

Entfernt alle dangling (ungetaggten) Images;  
kann mit dem Parameter `-a` dazu gebracht werden, alle nicht von einem  
Container verwendeten Images zu entfernen.

`docker container prune`

Entfernt alle gestoppten Container, die nicht mit `--rm` gestartet wurden.

`docker system prune`

Entfernt alle Arten von Objekten, also Images, Container, Buildcache und  
ungenutzte Netzwerke. Über den Parameter `--volume` können auch  
ungenutzte Volumes entfernt werden.



## Orchestrierung mit Kubernetes

Überblick über das Orchestrieren von Containern mit Kubernetes



## Container Orchestrierung

- Container (~Prozesse) brauchen Laufzeitplattform (~Betriebssystem)
  - Start, Stopp, Überwachung
  - Verteilung (On-prem / Cloud)
  - Networking / Discovery
  - Skalierung
  - Persistenz



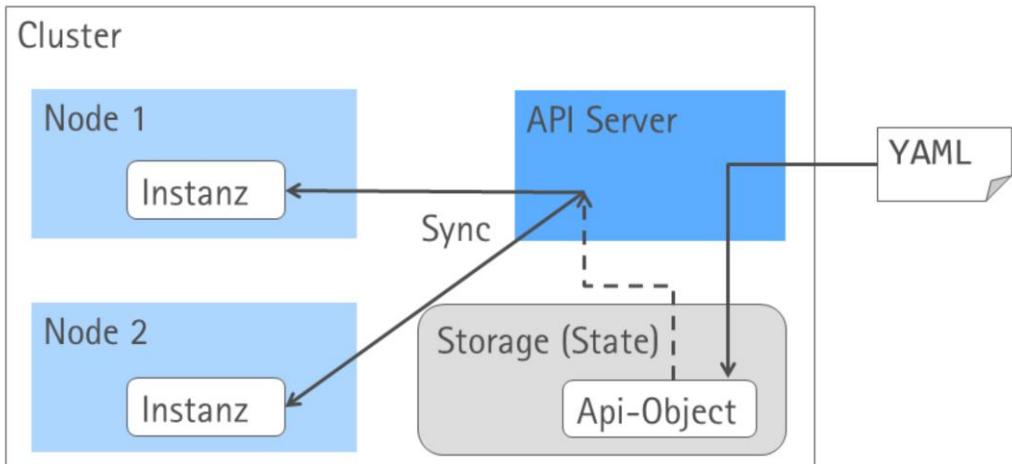
## Kubernetes (Abgekürzt: K8s)

- Orchestrierungs-Plattform für Container
- Open-Source (Apache-Lizenz 2.0)
- Projekt der Cloud Native Computing Foundation
- Ursprünglich von Google entwickelt
- Altgriechisch: κυβερνήτης = Steuermann



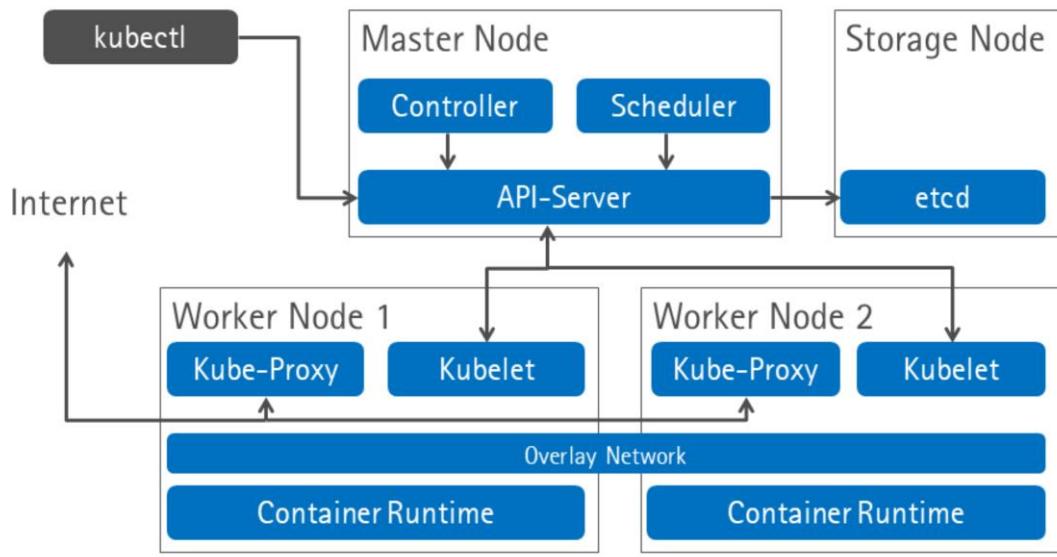


## Deklarativer Ansatz



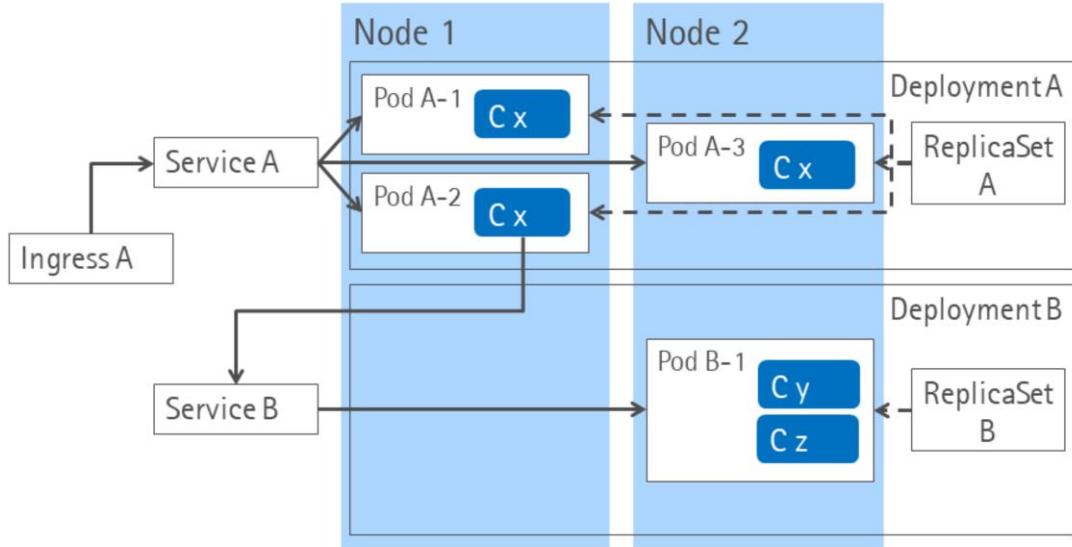


## Cluster-Architektur





## Überblick



Kubernetes Overview

Dieses Handout wird Ihnen persönlich zur Teilnahme am GFU-Seminar am 29.03.2022 überreicht.  
Eine Weitergabe oder Präsentation außerhalb des Seminars ist nicht erlaubt.



## Cloud

- Auf Kubernetes basierende Cloud-Plattformen
  - Managed Cluster
  - Einfach zu nutzen
  - Zahlen für genutzte Ressourcen
  - Mittlerweile viele Anbieter



Google Kubernetes Engine



Amazon EKS



Azure



DigitalOcean



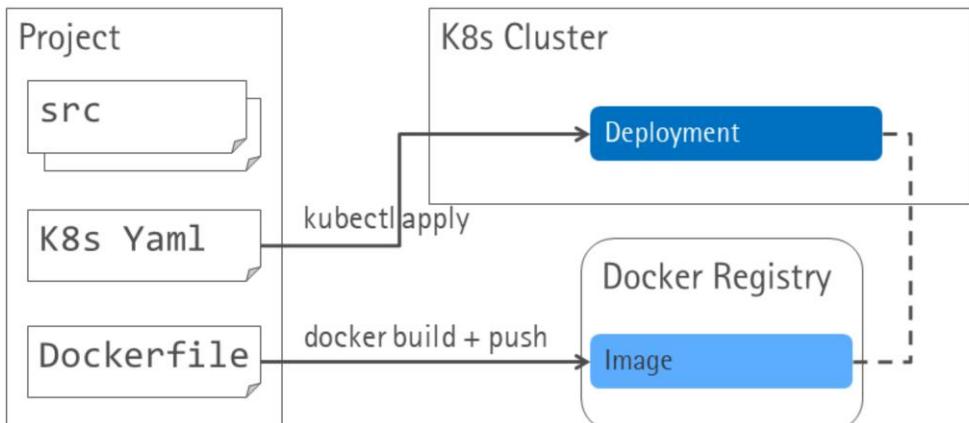
## Installation Bare-Metal

- Nicht ganz trivial
- Verschiedene Werkzeuge für Ausrollen eines Clusters
  - Kubespray
  - Rancher rke
  - ...





## K8s als Plattform für (Java-)Anwendungen





## Vorteile

- Deklarativer Ansatz
- Einfaches Skalieren
- No Vendor Lock-in
- On Premises oder in Cloud nutzbar
- Derzeit de facto Standard für Orchestrierung

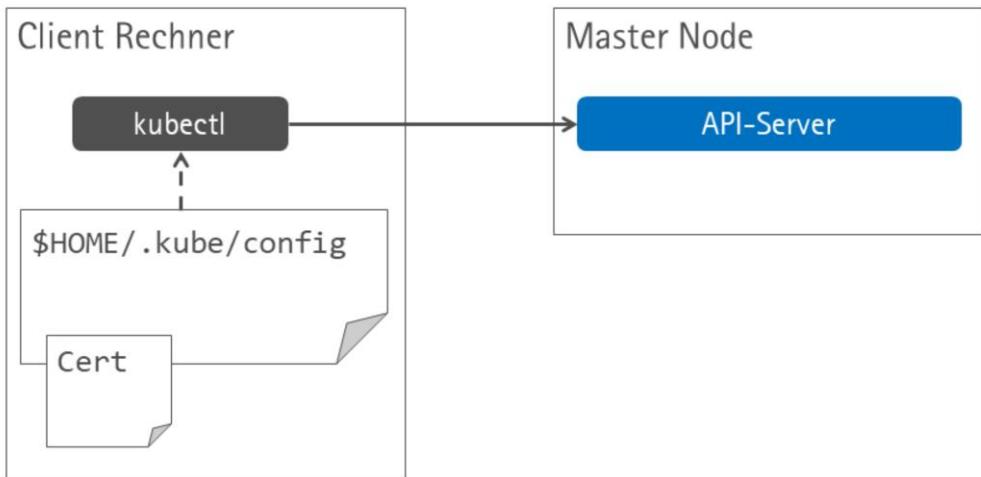


## Kubernetes Basics

Grundlagen des Orchestrierens von Containern mit Kubernetes



## Kubernetes



Kommunikation mit dem API-Server des Clusters findet über das Kommandozeilenwerkzeug `kubectl` statt, über welches alle Befehle abgesetzt werden können. Die Verbindung zum Cluster ist in einer Konfigurationsdatei eingetragen, welche sich standardmäßig unter `$home/.kube/config` befindet. Über die Umgebungsvariable `KUBECONFIG` oder das Flag `--kubeconfig` kann auch ein alternatives Configfile angegeben werden.

Es können Verbindungen zu mehreren Clustern in einer Konfigurationsdatei verwaltet werden. Ausgewählt werden kann die gewünschte Konfiguration dann über das Setzen des entsprechenden Contexts: `kubectl config use-context`

Authentifizierung kann entweder über Zertifikate erfolgen, falls von der Administration entsprechende Konfigurationen für die Nutzer bereitgestellt werden oder über ein Token, welches sich die User vom Cluster beziehen können.



## Lokale Kubernetes Umgebungen

- Docker Desktop
  - Windows und Mac
  - Systemeigene Virtualisierung
  
- Minikube
  - Windows, Mac und Linux
  - Unterschiedliche Virtualisierungsprovider
    - VirtualBox, Hyper-V, KVM, hyperkit, none

Häufig ist es erforderlich, zu Testzwecken einen Kubernetes Cluster auf dem lokalen Entwicklerrechner zu betreiben.

Hier gibt es unter Windows und Mac die Möglichkeit, Docker Desktop zu verwenden, welches Kubernetes direkt mitbringt. Kubernetes läuft in diesem Fall als Ein-Knoten-Cluster in derselben VM, wo auch der Docker-Daemon betrieben wird. Als Virtualisierungstechnologie kommt hierbei zwangsläufig Hyper-V bzw. hyperkit zum Einsatz. Auf Windows oder Mac ist dies vermutlich die einfachste Variante.

Alternativ ist es auch möglich, das Werkzeug minikube zu verwenden. Hiermit können über ein Kommandozeilentool Virtuelle Maschinen mit einem Docker + Ein-Knoten-Kubernetes-Cluster verwaltet werden. Es werden Windows, Mac und Linux als Betriebssystem und eine Reihe von Virtualisierungsprovidern unterstützt. Auf Linux ist es auch möglich, minikube ohne Virtualisierung direkt mit Docker zu verwenden. Vorteil von minikube ist, dass hier einfach mehrere Maschinen verwaltet werden können und auch die Kubernetes Version des Cluster definiert werden kann.



## YAML

- ☰ YAML Ain't Markup Language
- ☰ Sprache zur Datenserialisierung und Konfiguration
- ☰ Strukturiert durch Einrückungen
- ☰ Implementierungen für alle gängigen Programmiersprachen

YAML (YAML Ain't Markup Language) ist eine für den Menschen gut lesbare Sprache zur Datenserialisierung. Verglichen mit XML ist YAML deutlich minimalistischer und definiert die Struktur der Daten mit Hilfe von Einrückungen. Seit YAML 1.2 kann es als Superset zu JSON angesehen werden, es können in YAML entsprechend auch die Klammern wie in JSON für das Strukturieren verwendet werden. Theoretisch sollte also ein YAML-Parser dazu in der Lage sein, auch JSON zu verarbeiten.

Da YAML Dateien gut lesbar und aufgrund der einfachen Syntax schnell in einem Texteditor zu schreiben sind, findet YAML häufig Verwendung für Konfigurationsdateien.

Implementierungen für YAML stehen für alle gängigen Programmiersprachen zur Verfügung.



## YAML

<code>kurstitel: Kubernetes für Java-Entwickler</code>	Einfacher Wert
<code>adresse:</code> <code>strasse: Kantstraße 164</code> <code>plz: 10623</code> <code>ort: Berlin</code>	Assoziative Collection (Map)
<code>themen:</code> <ul style="list-style-type: none"><li>- <code>tag: 1</code> <code>thema: docker</code></li><li>- <code>tag: 2</code> <code>thema: kubernetes</code></li></ul>	Liste von Maps
<code>stichworte: [docker, kubernetes, java]</code>	Liste in Array-Schreibweise
<code>kommentar:  </code> <code>Kompakter Kurs mit allem Wesentlichen über</code> <code>Docker und Kubernetes aus Sicht der ...</code>	Mehrzeiliger Value

Kubernetes Basics

5

### Einfache Werte

Definiert in der Form

`key: value`

Als Basisdatentypen gibt es: `integer`, `float`, `string`, `date` und `boolean`  
`string`-Werte können in Hochkommas oder Anführungszeichen eingeschlossen werden (nur nötig, falls Sonder-/Meta-Zeichen im Text vorkommen).

### Listen / Arrays

Liste von einfachen Werten oder von Collections, definiert in der Form

`liste:`

- `item1`
- `item2`

Die Einrückungstiefe ist – ungewohnt für Java-Entwickler(innen) – signifikant!

Alternativ kann die aus JSON bekannte Array-Schreibweise genutzt werden:

`liste: [item1, item2]`

## Assoziative Collections / Maps

Fasst eine Reihe von Schlüssel-Wert-Paaren zusammen:

*map:*

```
key1: value1  
key2: value2
```

Alternativ die JSON Schreibweise:

```
map: {key1: value1, key2: value2}
```

## Mehrzeilige Text-Blöcke

Längere Texte können auf mehreren Zeilen definiert werden. Wird der Block mit einem > eingeleitet, werden beim Parsen des Textes Zeilenwechsel sowie White Space am Beginn und am Ende der Zeilen entfernt.:

```
text1: >  
    Dieser Text  
    wird ohne Blanks  
    und Zeilenwechsel gelesen
```

Mit einem | eingeleitet, bleiben die Zeilenwechsel dagegen erhalten:

```
text2: |  
    Dieser Text  
    besteht aus drei  
    Zeilen
```

## Mehrere Dokumente in einer Datei

In einer Datei können mehrere YAML-Dokumente hintereinander abgelegt werden:

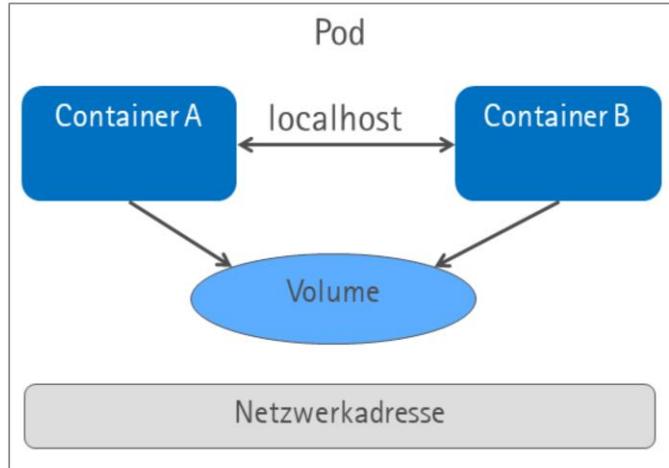
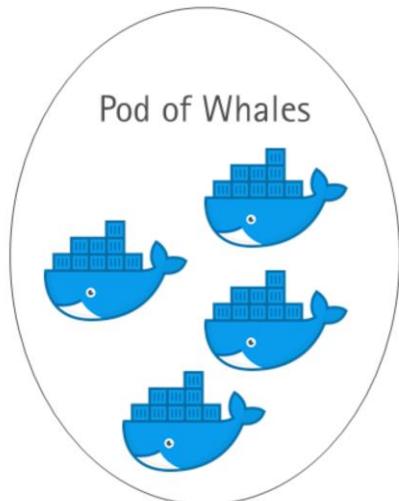
```
doc1
```

```
---
```

```
doc2
```



## Pods



Ein Pod fasst in Kubernetes eine Menge von Containern zusammen, welche immer gemeinsam als Einheit auf einem Node gestartet werden. Ein Pod stellt somit die kleinste verteilbare Einheit dar, die auf den Nodes gescheduled werden kann.

### **Pod = Pod of Whales**

Eine Gruppe von Meerestieren wie z. B. Walen wird im Englischen als Pod bezeichnet (Pod of Whales) und das Symbol von Docker ist ein Wal.

### **Pod = Rechner**

Einen Pod kann man sich auch als Abstraktion eines Rechners vorstellen. Die Container, welche in einem Pod laufen, befinden sich in einem Namespace und können auch Dateien austauschen. Volumes, die im Pod definiert werden, können in die Container gemounted werden, wodurch ein einfacher Dateiaustausch möglich ist. Die Container eines Pods können sich über localhost als Netzwerkadresse ansprechen und können auch Verfahren zur Inter-Prozesskommunikation wie System V Semaphores oder POSIX Shared Memory verwenden.

### **Pod = Anwendung**

In einem Pod werden Container zusammengefasst, die eigentlich eine eigenständige Anwendung ausmachen. Daher sieht man in der Praxis auch häufig, dass in einem Pod nur ein einziger Container definiert wurde. Zusätzliche Container würde man nur für Hilfswerzele zu verwenden. Etwas größeres, wie eine zur Anwendung gehörende Datenbank, würde normalerweise eigenständig in einem separaten Pod gestartet. Man könnte auch mehrere Prozesse in einem Container betreiben, was aber den Nachteil hätte, dass diese Information in Kubernetes nicht sichtbar wäre.



## Pod Konfiguration

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: rest-demo
```

```
spec:
```

```
  containers:
```

```
    - name: rest-demo
```

```
      image: ctr-demo-rest
```

```
      imagePullPolicy: IfNotPresent
```

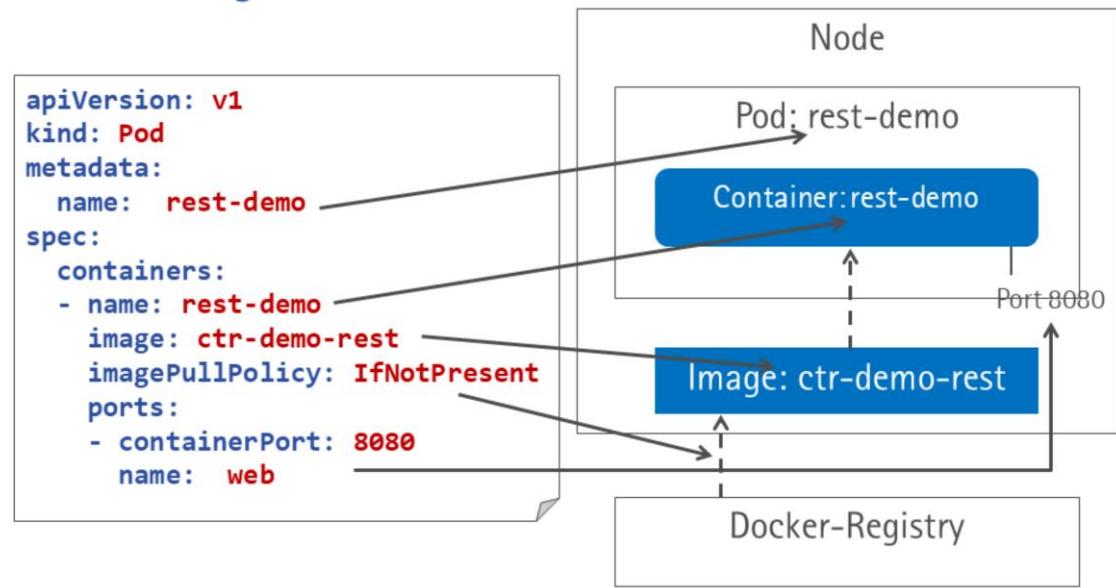
```
      ports:
```

```
        - containerPort: 8080
```

```
        name: web
```

Kubernetes Basics

8



Diese Felder sind für alle Kubernetes-Objekte anzugeben:

apiVersion: Gibt die Version der Kubernetes API an, die für dieses Objekt verwendet werden soll, Pods sind ab der Version v1 definiert.

kind: Gibt die Art des Objektes an, in diesem Fall: Pod

Alle Objekte können Metadaten definieren, z. B.:

metadata: Metadaten für den Pod

metadata.name: Name des Pods

Die Konfiguration des Pods erfolgt im Attribut spec:

spec.containers: Liste mit Containern

spec.containers[0].name: Name des Containers

spec.containers[0].image: Das zu verwendene Dockerimage

spec.containers[0].imagePullPolicy: Legt fest, wann das Image gepullt werden soll, mögliche Ausprägungen: Never, Always, IfNotPresent. Default ist Always für latest-Images, IfNotPresent sonst.

Eine vollständige Auflistung aller Parameter würde den Rahmen sprengen. Nutzen Sie bitte die Kubernetes API Reference:

<https://kubernetes.io/docs/reference/kubernetes-api/>



## kubectl apply

- ☰ Objekte verwalten
  - ☰ `kubectl create/replace/delete`

```
kubectl create -f <YAML-file>
```

- ☰ Kombi-Befehl: `apply` ≈ `create` oder `replace`
  - ☰ `kubectl apply`

```
kubectl apply -f <YAML-file>
```

- ☰ Konfiguration über Kustomization-file möglich

Es gibt zwei Wege, um Objekte über `kubectl` im Cluster zu verwalten, einen imperativen und einen deklarativen.

### Imperativ

Über `create` können neue Objekte angelegt, durch `replace` geupdated und über `delete` entfernt werden. Diese Vorgehensweise erfordert entsprechend, dass man sich Gedanken darüber machen muss, welche Befehle erforderlich sind, um jetzt einen gewünschten Zustand herzustellen.

### Deklarativ

Mit `apply` werden die notwendigen Operationen wie `create`, `replace` und `delete` automatisch passend ausgeführt, um einen Zustand herzustellen. Dem Befehl kann eine YAML-Datei, ein Verzeichnis mit YAML-Dateien oder eine Liste mit Dateien/Verzeichnissen übergeben werden. Im Falle eines Updates werden Felder mit alten Werten überschrieben, neue, im Cluster noch nicht vorhandene Felder gesetzt und nicht mehr vorhandene Felder entfernt.

Zum Löschen von Einstellungen, die in den Files nicht mehr vorhanden sind, gibt es den Parameter `--prune`. Er befindet sich aber noch im Alpha-Zustand (Version 1.15).

Seit der Version 1.14 gibt es die Möglichkeit, mit einem Kustomization-File zu arbeiten. Dies ist später unter dem Punkt `Kustomize` näher beschrieben.



## kubectl

☰ Pods anzeigen

```
kubectl get pods -o wide
```

☰ Pod als YAML

```
kubectl get pod <podname> -o YAML
```

☰ Logs anzeigen

```
kubectl logs -f <podname>
```

☰ Port forwarding

```
kubectl port-forward <pod-name> \
<local-port>:<pod-port>
```

☰ Pod entfernen

```
kubectl delete pod <podname>
```



### Objekte anzeigen

Über das Kommando `get` können Ressourcen angezeigt werden. Um alle Objekte eines Typs anzuzeigen: `kubectl get <objecttype>`. Eine erweiterte Darstellung kann über den Parameter `-o wide` angefordert werden.

Es ist auch möglich, sich einzelne Objekte anzuzeigen `kubectl get <objecttyp> <objectname>`. Um die YAML-Präsentation des Objektes zu erhalten, kann für den Output YAML als Typ gesetzt werden: `-o YAML`.

### Objekte entfernen

Objekte können auch wieder entfernt werden, und zwar mit Hilfe des `delete` Befehls: `kubectl delete <objecttyp> <objectname>`.

### Logs anzeigen

Um Container-Logs aus dem Pod anzuzeigen, kann der `logs` Befehl verwendet werden. Möchte man die Logs streamen, gibt es die Möglichkeit, dies über den Parameter `-f` festzulegen: `kubectl logs -f <objectname>`.

### Port Forwarding

Ein lokaler Port (auf dem Rechner, wo `kubectl` genutzt wird) kann auf einen Containerport weitergeleitet werden, sodass der Dienst auf `localhost` erreichbar ist: `kubectl port-forward <podname> <localhost-port>:<container-port>`.

---

💻 (Projekt `ctr-demo-rest`):

Pod einspielen und Port-Forwarding:

- `kubectl apply -f src/main/k8s/demo_01`
- `kubectl port-forward ctr-demo-rest 8080:8080`

The screenshot shows the Kubernetes Dashboard interface. On the left, there's a sidebar with navigation links like Nodes, Persistent Volumes, Roles, Storage Classes, Namespace (set to kube-system), Overview, Workloads, Cron Jobs, Daemon Sets, Deployments, Jobs, and Pods (which is selected). The main area has two charts: 'CPU usage' and 'Memory usage', both showing trends over time from 11:10 to 11:24. Below the charts is a table titled 'Pods' listing six entries:

Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
kubernetes-dashboard-7bfc7b	minikube	Running	0	27 minutes	0	19.746 Mi
heptio-ophelia	minikube	Running	0	27 minutes	0	18.004 Mi
infidult-granite-77c7p	minikube	Running	0	27 minutes	0.01	43.926 Mi
kube-scheduler-minikube	minikube	Running	0	20 hours	0.01	11.930 Mi
etcd-minikube	minikube	Running	0	20 hours	0.015	58.445 Mi

At the bottom left is a 'Kubernetes Basics' link, and at the bottom right are page numbers 11 and 10.

Ein Weboberfläche für Kubernetes, auf der Informationen zum Cluster und den Objekten angezeigt und Objekte angelegt bzw. geändert werden können.

Je nachdem, wie der Cluster aufgesetzt wurde, ist ein Dashboard vorhanden oder kann zumindest einfach deployed werden. Im Falle von Minikube erhält man das Dashboard durch den Kommandozeilenbefehl `minikube dashboard`. Falls das Dashboard in einem Cluster nachträglich eingerichtet werden soll, so reicht es, die dafür notwendigen YAML-Objekte einzuspielen:

```
kubectl apply -f \
  https://raw.githubusercontent.com/kubernetes/\
  dashboard/master/aio/deploy/recommended/\
  kubernetes-dashboard.yaml
```

In einem Produktionscluster sollte das Dashboard nicht ungesichert betrieben werden. Hier empfiehlt es sich, im Cluster RBAC zu aktivieren und dann eine Anmeldung über einen Authentifizierungsproxy, vor dem Dashboard, vorzunehmen, da das Dashboard selber nicht über die Möglichkeit verfügt, direkt einen Dienst wie Keycloak einzubinden.

---

(Projekt ctr-demo-rest):

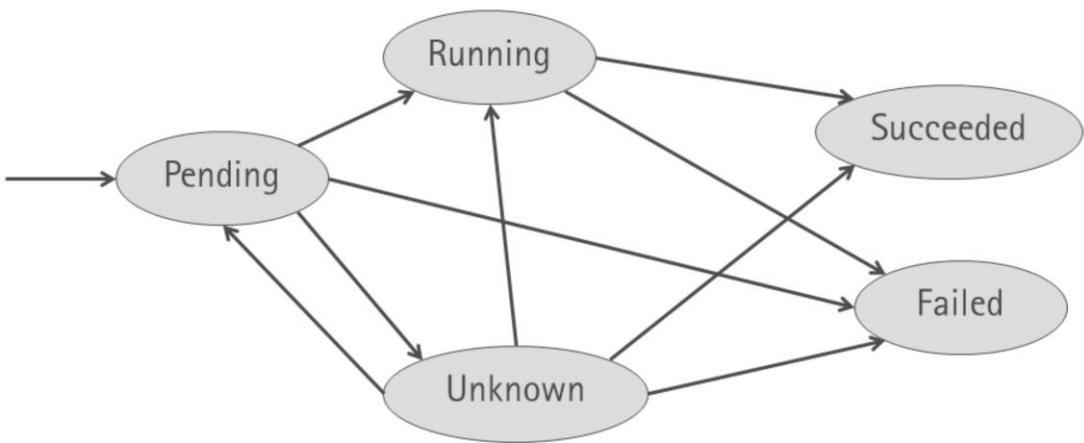
Dashboard:

- `kubectl apply -f src/main/k8s/dashboard`

Das Dashboard ist dann erreichbar auf dem Port 30001



## Pod Lifecycle



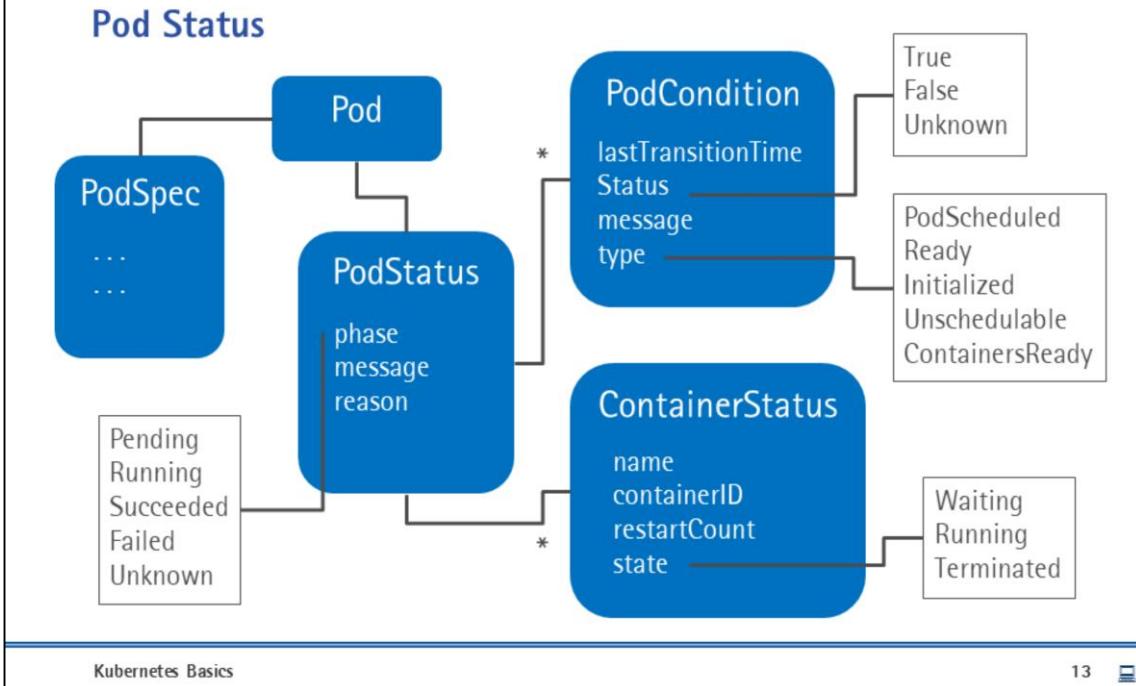
Ein Pod wird nach Anlage des API Projektes auf einem passendem Node gescheduled, was bedeutet, dass auf diesem Node die entsprechenden Container gestartet werden müssen. Solange der Pod noch nicht zugewiesen wurde bzw. die Container auf dem Node noch nicht gestartet wurden, befindet er sich in der Phase Pending.

Sind alle Container des Pods gestartet und läuft davon noch mindestens einer, so befindet sich der Pod in der Phase Running. Die Container im Pod laufen unter einer `RestartPolicy`, welche definiert, wann ein Container vom Pod neugestartet werden soll. `Always` bedeutet "immer nach Beenden des Containers", `OnFailure` "im Fehlerfalle" und `Never` "niemals". Der Default für die `RestartPolicy` ist `Always`.

Die Phase Succeeded wird erreicht, wenn alle Container des Pods gestartet wurden und erfolgreich beendet sind und auch nicht restarted werden.

Als Failed wird der Pod angesehen, wenn alle Container terminiert wurden und mindestens einer davon mit einem Fehler.

Falls es zu Problemen bei der Kommunikation zwischen Node und Master kommt, kann es sein, dass aktuelle Podzustände nicht für den API-Server bekannt sind. In diesem Fall wird als Wert für die Phase Unknown angegeben.



Pod-Objekte bekommen neben der Spec, die den Sollzustand beschreibt, auch ein Status-Objekt zugeordnet, in welchem der aktuelle Zustand des Pods beschrieben ist. Der Status wird automatisch von der Ablaufumgebung gepflegt.

#### Phase

Im Statusfeld `phase` wird der aktuelle Zustand des Pods beschrieben (s. vorige Seite).

#### PodCondition

Eine Liste von Statuszuständen, die der Pod im Laufe seines Lebens angenommen hat, mit dem dazugehörigen Zeitstempel, wann dieser Status erreicht wurde. Das Feld `status` gibt dabei jeweils an, ob dieser Zustand erreicht wurde, das Feld `type` gibt die Statusart an.

#### ContainerStatus

Liste der aktuellen Statuswerte für jeden der Container in diesem Pod. Enthält neben Informationen über den Container ein Feld `state` für den Zustand.

---

💻 (Projekt `ctr-demo-rest`):

Das File `src/main/k8s/demo_01/pod.yaml` zeigt die Definition eines Pods, der das zuvor gebaute Image `gedoplan-ctr-demo-rest` ausführt. Die `imagePullPolicy` wird darin explizit auf `IfNotPresent` gestellt, damit das Image lokal genutzt werden kann, denn als `latest-Image` würde es sonst stets aus einer Registry geladen.



## Übung KUBERNETES\_BASICS\_01

- ☰ Starten Sie die Anwendung `ctr-exercise-rest` als Pod im Cluster
- ☰ Lassen Sie sich die Logs anzeigen
- ☰ Nutzen Sie die Anwendung über Port Forwarding

💻 (Projekt `ctr-exercise-rest`):

1. Wir gehen hier davon aus, dass in einer früheren Übung bereits ein Docker Image namens `gedoplan-seminar/ctr-exercise-rest` erstellt wurde.

Schreiben Sie ein YAML-File zur Definition eines Pods, der dieses Image als Container laufen lässt. Nutzen Sie als `imagePullPolicy` `IfNotPresent`.

Musterlösung: `src/main/k8s/exercise_01/app-pod.yaml`

2. Spielen Sie den Pod in Ihren Kubernetes-Cluster ein (`kubectl apply ...`). Prüfen Sie, ob der Pod läuft (`kubectl get pod ...`)
3. Zeigen Sie die Log-Ausgaben an (`kubectl logs ...`).
4. Richten Sie ein Port Forwarding ein (`kubectl port-forward ...`). Der Container nutzt intern den Port 8080.  
Prüfen Sie, ob Sie nun bspw. mit `curl` auf die Anwendung zugreifen können.
5. Entfernen Sie den Pod wieder (`kubectl delete pod ...`).



## Metadaten

- Labels
  - Identifizierend
  - Verwendet für Auffinden von Objekten
  
- Annotations
  - Nicht identifizierend
  - Erlaubt Zeichen, die in Label nicht möglich
  - Auch für strukturierte Inhalte geeignet

```
apiVersion: v1
kind: Pod
metadata:
  name: rest-demo
  labels:
    key: value
  annotations:
    key: value
```

Den Objekten in Kubernetes können Metadaten in Form von Labels und Annotationen zugeordnet werden.

### Labels

Labels sind zusätzliche Informationen an den Objekten, welche der Identifizierung dienen. Damit kann man die Objekte z. B. einer bestimmten Kategorie zuordnen oder sie als Bestandteile einer Anwendung markieren. Labels können dann dafür genutzt werden, um Objekte im Cluster aufzusuchen. Die Länge der Values ist auf 63 Zeichen begrenzt.

### Annotations

Annotations können für zusätzliche Metainformationen genutzt werden, welche nicht identifizierend sind. In Annotation-Values sind längere Inhalte mit weniger Beschränkungen in Hinblick auf verwendete Zeichen möglich. Hier könnten Informationen über die Versionshistorie abgelegt werden oder vielleicht auch Informationen über Log-Formate und Metrics Endpoints. Diese Informationen könnten dann von einem Automatismus ausgewertet werden, um für die Anwendung eine entsprechende Konfiguration vorzunehmen.



## Selektoren

```
kubectl get pods --selector=key=value
```

Selektor: apptype=backend

Pod 1

labels:  
name: myapp1  
apptype: backend

Pod 2

labels:  
name: myapp1  
apptype: backend

Pod 3

labels:  
name: myapp1  
apptype: frontend

Selektoren können verwendet werden, um Objekte nach deren Labels auszuwählen. Damit ist es dann z. B. möglich, alle Pods anzuzeigen, welche über bestimmte Labels verfügen oder auch alle Objekte zu entfernen, die mit den entsprechenden Labels versehen sind.

Dieser Mechanismus wird in Kubernetes von einigen Komponenten verwendet: Unter anderem identifizieren ReplicaSets ihre Pods durch einen Selector, Services selektieren so die zugeordneten Pods und Pods können damit auf bestimmte Nodes zugewiesen werden.



(Projekt ctr-demo-rest):

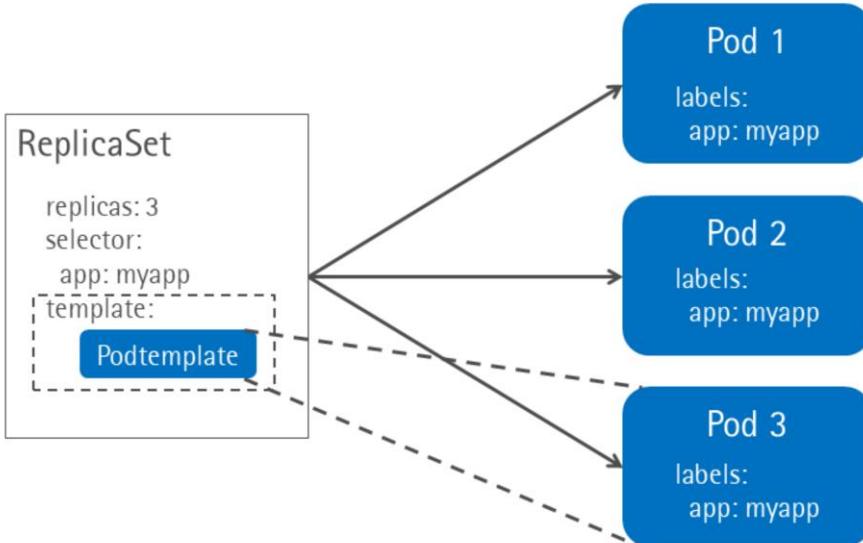
K8s-Manifeste:

- src/main/k8s/demo\_02

Pods selektieren über: kubectl get pod --selector=apptype=backend



## ReplicaSets



Ein ReplicaSet hat die Aufgabe, dafür zu sorgen, dass immer eine bestimmte Anzahl von Pods im Cluster läuft. Die Pods werden mit Hilfe eines Selectors zugeordnet und können entweder vom ReplicaSet mit einem Template erzeugt oder auch separat eingespielt werden. Das ReplicaSet wird also, falls zu wenig Pods da sind, neue anlegen und falls zu viele laufen, wieder welche entfernen, um auf die gewünschte Anzahl zu kommen.

Das Pod-Template eines ReplicaSets wird in Form einer gewöhnlichen Pod-Definition im YAML angegeben. Die Anzahl der Instanzen wird in dem Feld `spec.replicas` angegeben. Mit Hilfe des Befehls

```
kubectl scale --replicas=3 rs/<rsname>
```

kann die Anzahl der Replicas über `kubectl` erhöht werden.

Ein ReplicaSet kann als Target für einen Horizontal Pod Autoscaler verwendet werden, welcher dann die Anzahl Replicas kontrolliert.

ReplicaSets werden in der Regel nicht direkt verwendet, da ein Deployment eine komfortablere Variante darstellt, um ReplicaSets zu verwalten.

---

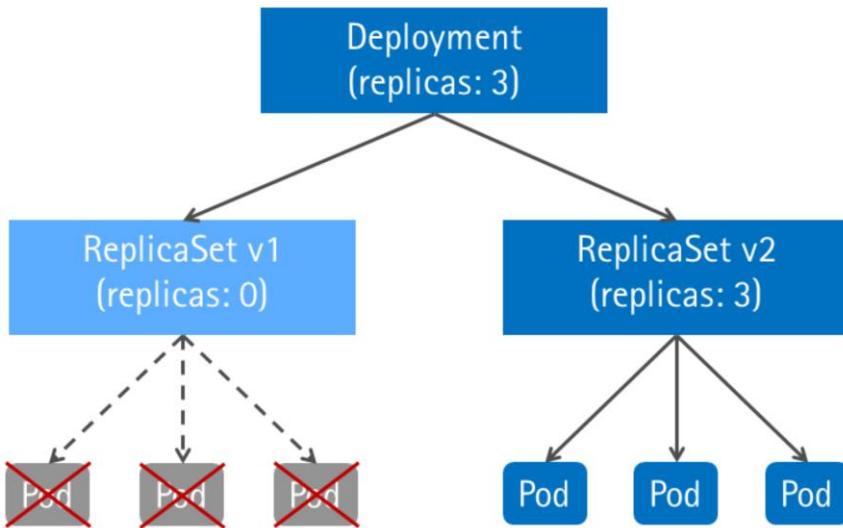
(Projekt ctr-demo-rest):

K8s-Manifeste:

- `src/main/k8s/demo_03`



## Deployments



Ein Deployment ermöglicht ein deklaratives Update von ReplicaSets und den davon verwalteten Pods. Ein Deployment legt ein ReplicaSet mit der konfigurierten Anzahl Replicas an, welches dann wiederum für das Hochfahren der Pods sorgt. Bei Änderung an der Deployment-Konfiguration wird eine neue Version des ReplicaSets angelegt.

Für die Updates gibt es verschiedene Verfahren, welche über das Feld `spec.strategy.type` definiert werden können:

Die Default-Update-Strategie `RollingUpdate` sorgt dafür, dass immer eine gewünschte Anzahl Pods verfügbar ist, indem schrittweise neue Pods hinzugefügt und alte entfernt werden. Die max. Abweichungen vom Soll werden über `spec.strategy.rollingUpdate.maxUnavailable` und `spec.strategy.rollingUpdate.maxSurge` gesteuert.

Die Strategie `Recreate` entfernt alle alten Pods, bevor neue hochgefahren werden.

Alte Versionen der ReplicaSets werden aufbewahrt und ermöglichen somit auch ein Rollback auf einer älteren Version. Die Anzahl an aufbewahrten Revisionen kann über das Feld `spec.revisionHistoryLimit` festgelegt werden, Default ist hier 10. Zu einer alten Version kehrt man mit folgendem Befehl zurück:

```
kubectl rollout undo deployment.v1.apps/mydeployment
```

Für das Zurücksetzen auf eine bestimmte Revision kann der zusätzliche Parameter `--to-revision=revision` verwendet werden.

---

(Projekt ctr-demo-rest):

K8s-Manifeste:

- `src/main/k8s/demo_04`



## Übung KUBERNETES\_BASICS\_02

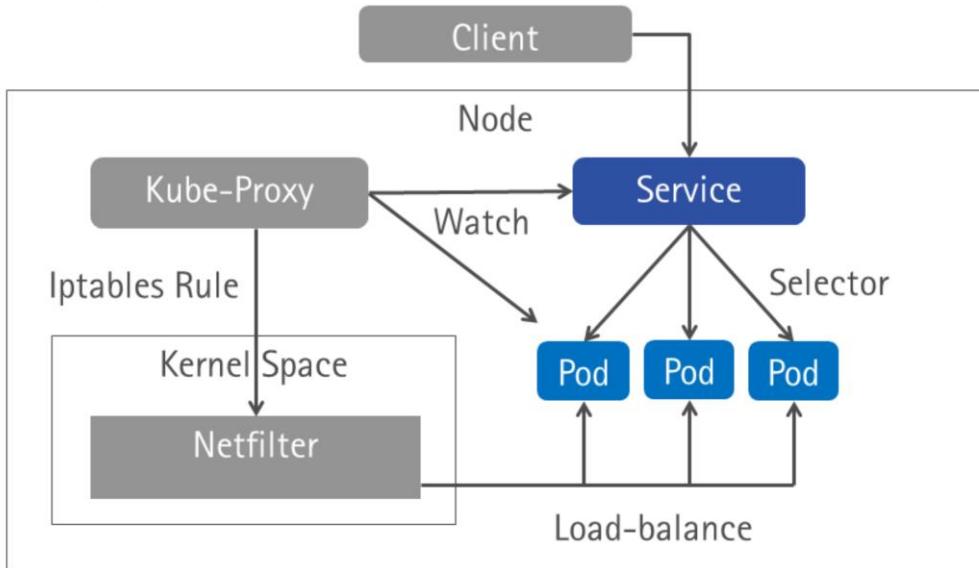
- ☰ Definieren Sie die Anwendung `ctr-exercise-rest` als Deployment im Cluster

### 💻 (Projekt `ctr-exercise-rest`):

1. Schreiben Sie ein YAML-File zur Definition eines Deployments für die Übungsanwendung.
  - Nutzen Sie die Pod-Definition der vorigen Übungsaufgabe nun im Template der Deployment-Definition.
  - Verknüpfen Sie Deployment und Pod-Template mit einem passenden Selektor.
2. Spielen Sie das Deployment in Ihren Kubernetes-Cluster ein (`kubectl apply ...`). Prüfen Sie, ob die gewünschten Objekte vorhanden sind (`kubectl get all ...`)
3. Zusatzaufgabe: Ändern Sie die Replica-Anzahl.
4. Entfernen Sie das Deployment wieder (`kubectl delete ...`).

Musterlösung: `src/main/k8s/exercise_02/app-deployment.yaml`

## Services



Da Pods in der Regel automatisch erzeugt werden und somit auch immer unterschiedliche clusterinterne IP-Adressen bekommen, können diese Adressen nicht genutzt werden, um auf die Anwendungen zuzugreifen. Des Weiteren besteht im Falle mehrerer Replicas der Bedarf nach einem Loadbalancing.

Diese Probleme werden durch Services gelöst, mit denen zentrale Schnittstellen im Cluster definiert werden. Ein Service wird standardmäßig unter einer clusterinternen IP-Adresse bereitgestellt. Services werden über eine Selector-Query mit den Pods verknüpft. Auf jedem Node im Cluster läuft ein kube-proxy, welcher die Services und Pods überwacht und entsprechende Regeln für das Routing einrichtet, sodass Anfragen an die Service-IP auf die damit verknüpften Pod-Adressen verteilt werden.

## Proxy-Modes

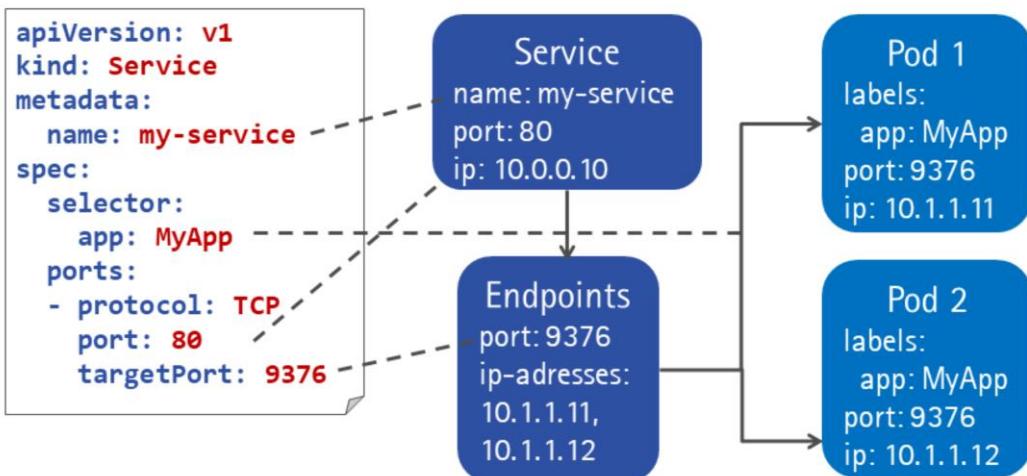
Für clusterinterne IPs gibt es verschiedene Verfahren, wie die Anfragen an die Pods weitergeleitet werden: userspace, iptables (default) und ipvs.

## Load Balancing

Wer genau das Load-Balancing übernimmt, hängt vom verwendeten Proxy-Mode ab. Das Verfahren ist dabei standardmäßig Round-Robin, kann im Falle von Proxy-Mode ipvs aber auch umgestellt werden. Stickiness wird dabei anhand der Client-Adresse unterstützt.



## Services – Konfiguration



### Selectoren und Endpoints

Über einen im Feld `spec.selector` definierten Selector wird ein Endpoints Objekt kontinuierlich mit den Adressen der Pods geupdated. Wird keine Selectorquery angegeben, so wird kein Endpoints-Objekt automatisch angelegt. Dies kann genutzt werden, um mit einem manuell angelegten Endpoints-Objekt z. B. Adressen, die nicht im Cluster liegen, als Service bereitzustellen.

### Ports

Unter `spec.ports` können Ports für den Service definiert werden, wobei `port` den Port des Services angibt und `targetPort` den Port des Endpoints. Sind diese gleich, so kann `targetPort` auch weggelassen werden. Als `protocol` kann für den Port TCP oder UDP gewählt werden.

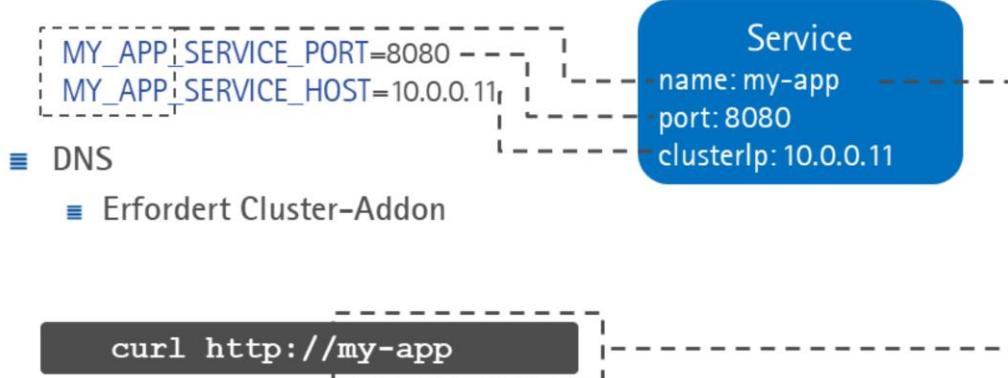
### ServiceTypes

Wie der Service bereitgestellt wird, kann über das Feld `type` festgelegt werden. Der Default ist hierfür `ClusterIp`, welches den Service unter einer clusterinternen Adresse anbietet mit dem als proxy-mode gewählten Verfahren. Es kann auch eine feste IP konfiguriert werden. `NodePort` stellt den Service auf allen Nodes auf einem bestimmten Port zur Verfügung und ermöglicht ein externes Load Balancing (außerhalb von Kubernetes). Der Typ `LoadBalancer` erfordert eine entsprechende Load-Balancer-Komponente, welche vom Cloudprovider bereitgestellt werden muss.



## Service Discovery

- Environment Variablen
  - In Pods automatisch gesetzt



### Environment Variables

In allen Pods werden von Kubernetes automatisch eine Reihe von Umgebungsvariablen gesetzt. Diese sind dann für die Pods im Container entsprechend verfügbar:  
{ SVCNAME }\_SERVICE\_HOST und { SVCNAME }\_SERVICE\_PORT.

### DNS

In der Regel verfügen Kubernetes-Installationen über ein Cluster-Addon für DNS wie z. B. CoreDNS. Für Services werden dann DNS-Namen bereitgestellt:

service-name.namespace.svc.cluster.local

Das Suffix svc.cluster.local ist konfigurierbar, kann bei einem Lookup allerdings weggelassen werden.

Der Namespace für unsere Services ist derzeit immer default. Innerhalb eines Namespaces kann auch dieser Name beim Lookup entfallen.

Der Beispielservice ist somit von anderen Pods aus erreichbar unter

my-app.default.svc.cluster.local  
my-app.default  
my-app, falls im gleichen Namespace



## Umgebungsvariablen

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - name: A
      env:
        - name: DATABASE_USER
          value: my-user
        - name: DATABASE_SCHEMA
          value: ${DATABASE_USER}
        - name: DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              ...

```

Definition mit festem Wert

Definition mit Wert aus Variable

Definition mit Wert aus:  
ConfigMap, Secret oder PodField



### Definition

Umgebungsvariablen können in den Containern eines Pods gesetzt werden, indem diese in der Liste unter `spec.containers.env` eingetragen werden. Als Werte besteht hier die Möglichkeit, mit `value` direkt einen Wert einzutragen, mit `valueFrom` einen Wert aus einer ConfigMap, einem Secret oder einem PodField zu referenzieren. Diese Objekte werden später noch erläutert.

### Verwendung in Pod-Config

Umgebungsvariablen eines Containers können in der Pod-Konfiguration des Containers an einigen Stellen verwendet werden, um Werte einzusetzen. Dies ist möglich bei Befehlen/Parametern wie `cmd` und `exec` oder auch in der Wertbelegung anderer Umgebungsvariablen. Für das Einsetzen eines Wertes aus einer im Container definierten Umgebungsvariablen wird die folgende Schreibweise verwendet: `$ (MY_VAR)`

---

💻 (Projekt ctr-demo-rest):

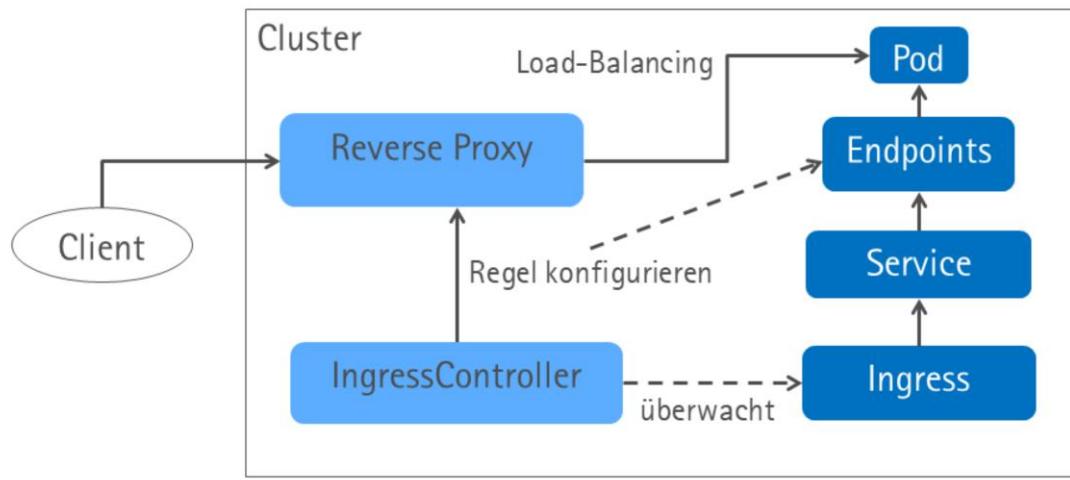
K8s-Manifeste in `src/main/k8s/demo_05:`

- `db-deployment.yaml` Deployment für eine Postgres-DB
- `db-service.yaml` passender Service dazu
- `app-deployment.yaml` Deployment für die Demo-Anwendung; sie wird darin mit Hilfe von Umgebungsvariablen so konfiguriert, dass sie auf die Postgres-DB zugreift (statt der ursprünglichen H2-DB)
- `app-service.yaml` Service dazu; er definiert u. a. den Nodeport 30101, d. h. die Anwendung ist dann unter `localhost:30101` erreichbar

Die Manifeste können auf einmal mit `kubectl apply -f src/main/k8s/demo_05` eingespielt (und später mit `kubectl delete -f ... entfernt`) werden.



## Ingresses



Über Ingress-Regeln können Services nach außerhalb des Clusters hin erreichbar gemacht werden. Für das Definieren einer Ingress-Regel muss ein entsprechendes Ingress-Objekt angelegt werden, welches den Service referenziert. Ein Ingresscontroller überwacht diese Objekte und legt in einem Reverse-Proxy entsprechende Regeln an. Der Reverse Proxy kann sich dabei im Cluster befinden, muss dann allerdings auf Nodes laufen, wo er von außen auf einem Port z. B. 80 erreichbar ist.

### Virtual Host basiertes Routing

In der Ingress-Regel kann der Service unter einem virtuellen Hostnamen bereitgestellt werden. In diesem Fall muss sichergestellt sein, dass eine Anfrage an diesen Hostnamen beim Reverse-Proxy landet, d. h. dass die IP-Adresse des Nodes im DNS entsprechend eingetragen ist.

### Path basiertes Routing

Anstelle oder zusätzlich zu Host basiertem Routing kann auch ein Pfad verwendet werden. Dies führt allerdings erfahrungsgemäß zu einer ganzen Reihe von Schwierigkeiten wegen Path Rewriting.

### SSL

Es kann SSL-Passthrough aktiviert werden, falls das Backend bereits mit SSL konfiguriert wurde. Ansonsten kann auch über Angabe eines Zertifikat-Secrets eine Regel für SSL-Termination im Reverse-Proxy konfiguriert werden.



## Nginx Ingress Controller

- IngressController auf Basis von Nginx

- Einfach im Cluster zu deployen
  - Enthält Nginx als Reverse-Proxy
  - Muss nach außen verfügbar sein



- Nginx: Open-Source Webserver
  - Reverse-Proxy, Loadbalancer, Cache, etc.



Nginx ist ein Open-Source-Webserver, welcher auch als Reverse Proxy, Load Balancer und HTTP Cache verwendet werden kann.

Der Nginx-IngressController ist ein IngressController mit einem Nginx-Reverse-Proxy, welcher einfach im Cluster deployed werden kann. Der Nginx Service muss dafür auf den Nodes, wo er läuft, von außen erreichbar sein, z. B. über einen Hostport.

Virtual Hosts oder Pfade, welche vom Nginx nicht aufgelöst werden können, werden an einen Backendservice mit einer Fehlerseite weitergeleitet. Ein Default Backend hierfür, welches ein "not found" (Code 404) zurückmeldet, ist bereits vorkonfiguriert.

Über spezielle Annotationen an den Ingress-Objekten können die Regeln, die im Nginx angelegt werden, noch genauer bestimmt werden.

Die Installation eines Nginx-Ingress-Controllers hängt stark vom verwendeten System ab. Weitere Infos dazu finden Sie in <https://kubernetes.github.io/ingress-nginx/deploy/>.



(Projekt `ctr-demo-rest`):

Je nach Zielumgebung muss zunächst ein Ingress-Controller eingerichtet werden. Für Docker Desktop können die YAML-Dateien im Ordner `src/main/k8s/nginx-ingress` des Projekts `ctr-infra` genutzt werden.

Die YAMLS im Projektverzeichnis `src/main/k8s/demo_06` unterscheiden so von der vorigen Demo:

- `app-service.yaml` definiert keinen NodePort mehr
- `app-ingress.yaml` definiert eine Ingress-Regel für den Virtual Host `rest-demo.localtest.me`. `localtest.me` ist eine Pseudo-Domäne, die stets auf `127.0.0.1` abbildet (also `localhost`).  
Sollte die Auflösung von `*.localtest.me` in der jeweiligen Zielumgebung nicht funktionieren, kann so getestet werden:  
`curl localhost -H 'host: rest-demo.localtest.me'`



## Übung KUBERNETES\_BASICS\_03

- Mysql Datenbank als Datenquelle für die Anwendung
  - Deployment mit Mysql-DB anlegen
  - Service für die Datenbank definieren
  - Service-Adresse in Datasource-Konfiguration verwenden
- Anwendung `ctr-exercise-rest` über Service und Ingress nach außen bereitstellen

🔗(Projekt `ctr-exercise-rest`):

Anleitung:

1. Erzeugen Sie ein Deployment für eine MySQL-Datenbank:
  - Nutzen Sie das Image `mysql:8.0.3`
  - Eine Datenbank mit Benutzer wird beim Starten angelegt und ist über Umgebungsvariablen konfigurierbar: `MYSQL_DATABASE`, `MYSQL_USER`, `MYSQL_PASSWORD`. Ein `root`-Passwort muss ebenfalls vergeben werden über `MYSQL_ROOT_PASSWORD`.
  - Definieren Sie den Port 3306 (Standard-Port von MySQL) als Container-Port.
2. Erzeugen Sie einen passenden Service dazu.
3. Ergänzen Sie das Deployment der Anwendung um Environment-Variablen:
  - `MYSQL_SERVICE_HOST` und `MYSQL_SERVICE_PORT` adressieren Ihren DB-Service.
  - `MYSQL_USER` und `MYSQL_PASSWORD` wie oben.
  - `MYSQL_DATABASE` wie oben, aber ergänzt um `?useSSL=false`.
  - `DATASOURCE=MySQLDS`
4. Ergänzen Sie einen passenden Service.
5. Schreiben Sie einen Ingress bspw. für `exercise.localtest.me`.

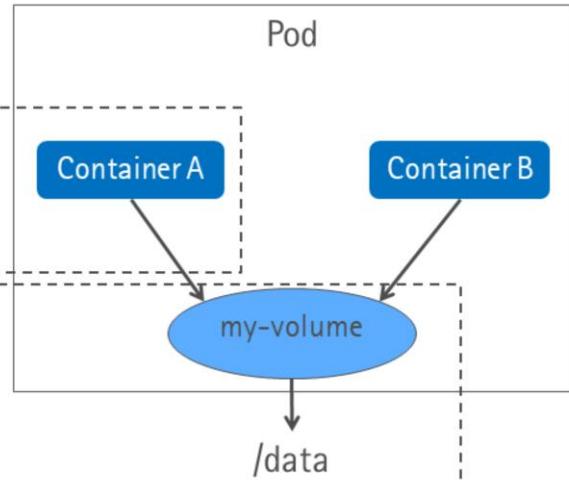
Musterlösung: `src/main/k8s/exercise_03`

Achtung: Die DB muss laufen, bevor die Anwendung gestartet wird!



## Volumes

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - name: A
      volumeMounts:
        - mountPath: /mydata
          name: my-volume
  volumes:
    - name: my-volume
      hostPath:
        path: /data
```



Da Container von der Laufzeitumgebung automatisch neugestartet werden, falls diese z. B. abstürzen, gehen Dateien in den Containern verloren. Sollen Dateien über Neustarts hinweg bestehen bleiben, so muss hier mit Volumes gearbeitet werden.

Volumes werden auf Pod-Ebene definiert und dann in den Containern gemountet. Dabei ist es möglich, dass mehrere Container in dem Pod sich ein Volume mounten, um darüber Dateien auszutauschen.

Es werden eine ganze Reihe unterschiedlicher Typen von Volumes unterstützt, darunter z. B. hostPath, nfs, iscsi und diverse Cloudspeicher wie GlusterFs, CephFs oder Storageos.

Ist in einem Dockerfile ein Pfad als Volume definiert worden, aber kein expliziter mount für diesen Pfad konfiguriert, so wird ein Volume vom Typ emptyDir angelegt. Dabei handelt es sich um ein temporäres Dateisystem, welches nur so lange lebt wie der Pod.

---

(Projekt ctr-demo-rest):

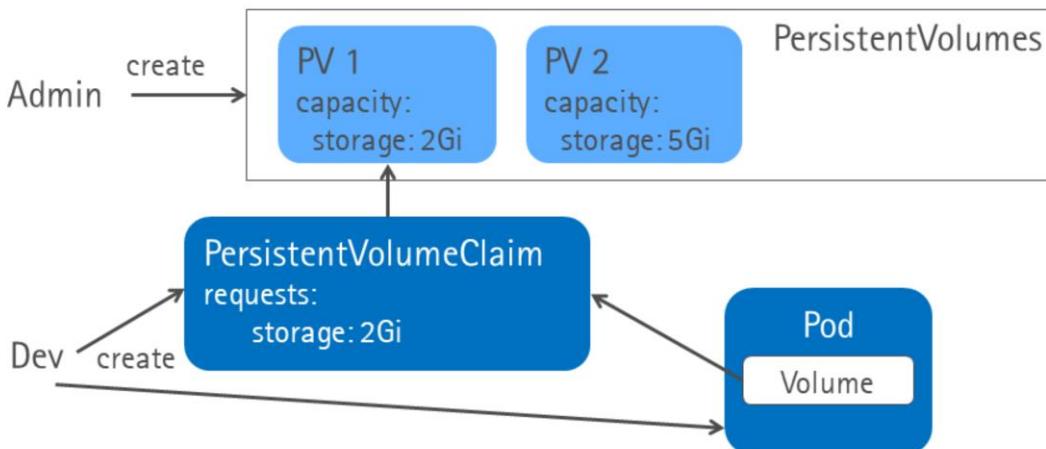
Mit dem Manifesten in `src/main/k8s/demo_07` wird die Demo-Anwendung diesmal ohne DB, aber mit zwei Containern im Pod gestartet. Beide Container nutzen das selbe Volume und tauschen darüber einen Text aus, der unter `rest-demo.localtest.me/resources/textFile` angeschaut werden kann.

Diese Demo hat u. a. zwei Schwächen, die noch behandelt werden:

- Statt der direkten Nutzung von Volumes empfehlen sich PersistentVolumeClaims
- Der zweite Container sollte besser ein Init-Container sein.



## PersistentVolumeClaims



In der Regel sollten sich die Entwickler, welche Anwendungen im Cluster betreiben wollen, sich nicht direkt damit auseinandersetzen müssen, wo die Daten abgelegt werden. Daher gibt es die Abstraktionen PersistentVolume und PersistentVolumeClaim.

Es ist Aufgabe des Cluster-Administrators, PersistentVolume Objekte anzulegen, welche dann jeweils einen echten Speicherbereich (hostpath, nfs, ceph, glusterfs, usw.) referenzieren. Wird nun für eine Anwendung persistenter Speicher benötigt, so muss nur ein PersistentVolumeClaim Objekt angelegt werden, welches PersistentVolumes nach Kriterien wie z. B. benötigte Speicherkapazität anfragt. Wird ein entsprechendes Volume gefunden, so ist der Claim danach als 'bound' gekennzeichnet und kann nun in einem Pod als Volume verwendet werden.

### ReclaimPolicy

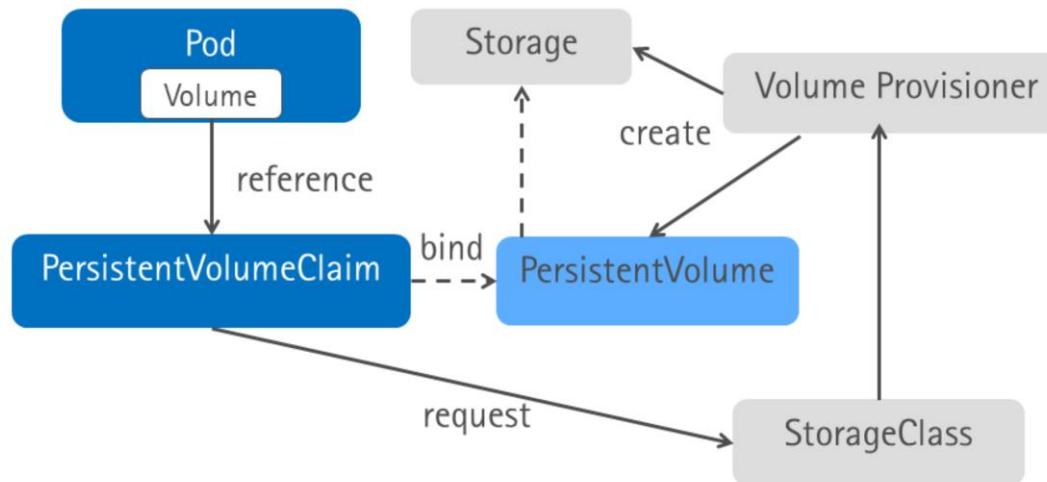
Die ReclaimPolicy legt fest, was mit einem wieder freigegebenen Volume (nach Entfernen eines Claims) geschieht. 'Retain' belässt das Volume und erfordert ein manuelles Aufräumen, falls gewünscht, 'Recycle' entfernt den Inhalt, lässt das Volume aber zur Wiederverwendung bestehen und 'Delete' sorgt für ein Entfernen des Volumes.

### Access Modes

Der AccessMode eines PVs legt fest, ob das Volume von mehreren Stellen aus parallel für Lesen oder Schreiben gemountet werden kann. Die verfügbaren Modes sind: `ReadWriteOnce`, `ReadOnlyMany` und `ReadWriteMany`.



## StorageClasses – Volume Provisioner



Für das dynamische Bereitstellen von Volumes im Cluster kann ein entsprechender Provisionierer konfiguriert werden, welcher Volumes mit Hilfe eines entsprechenden Storage-Backends erzeugt.

Die Konfiguration erfolgt hier über StorageClass-Objekt, welches im Cluster angelegt werden muss. Die PersistentVolumeClaims können dann eine StorageClass angeben, aus der das Volume stammen soll oder verwenden bei Nichtangabe die Default-StorageClass.

Wird nun ein PVC zu einer StorageClass angelegt, so wird über den in der StorageClass definierten Provisionierer über ein Storage-Plugin ein Volume mit den gewünschten Eigenschaften angelegt. Zum einen wird hier ein tatsächliches Volume vom Storage-Backend erzeugt (GlusterFs, Ceph, Storageos, usw.) und auch ein PersistentVolume Objekt dazu angelegt.

Auf diese Weise ist es möglich, Volumes automatisch bei Bedarf zu erzeugen, ohne dass ein Administrator manuell tätig werden muss. Hier ist nur einmal zu Beginn das Konfigurieren der StorageClass notwendig.



(Projekt `ctr-demo-rest`):

Mit den K8s-Manifesten in `src/main/k8s/demo_09` nutzt die DB nun ein Persistent Volume.

Die DB-Inhalte bleiben jetzt erhalten, auch wenn der DB-Pod gelöscht und dann automatisch neu gestartet wird. Wird allerdings das PersistentVolumeClaim (kurz PVC) entfernt, gehen die Daten verloren.



## Übung KUBERNETES\_BASICS\_04

- ☰ Sorgen Sie in `ctr-exercise-rest` für den Erhalt der Daten in der MySQL-Datenbank
  - ☰ Anlegen eines PersistentVolumeClaims
  - ☰ Einbinden des PVCs als Volume im Pod
    - Mountpath für Mysql-Container: `/var/lib/mysql`

💻(Projekt `ctr-exercise-rest`):

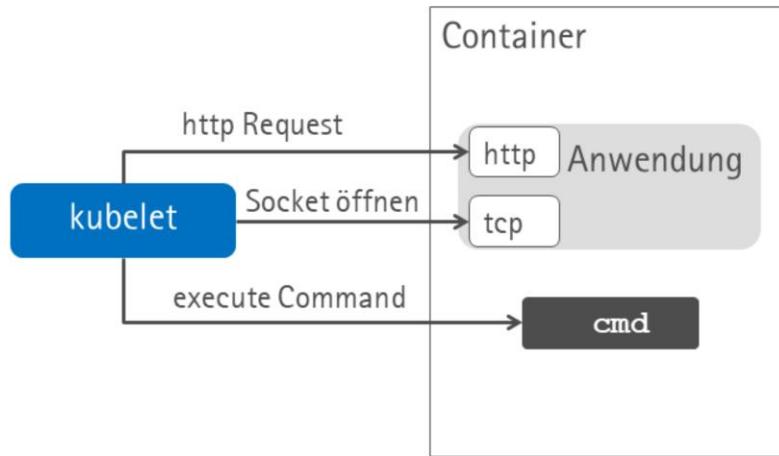
Anleitung:

1. Ändern Sie das Deployment der MySQL-DB:
  - Die MySQL-DB speichert ihre Daten im Pfad `/var/lib/mysql`. Verbinden Sie diesen Pfad mit einem Volume.
  - Lassen Sie das Volume ein PVC referenzieren.
2. Definieren Sie das PVC in einem neuen YAML-File (Access Mode `ReadWriteOnce`, Storage-Größe `1Gi`).

Musterlösung: `src/main/k8s/exercise_04`



## Probes



Für die Container eines Pods können Probes definiert werden, welche der Kubernetes-Laufzeitumgebung Informationen über den Zustand des Containers geben. Es gibt drei unterschiedliche Möglichkeiten für das Definieren von Probes.

### Command

Ein Befehl, der im Container ausgeführt wird. Sein Returncodes bestimmt das Ergebnis. In dem Feld `exec.command` wird dazu ein Array aus Befehl und Parametern angegeben.

### http

Nutzt einen http-GET-Request auf einem Port und Pfad, welcher anhand des http-Response-Codes festlegt, ob der Test erfolgreich war. Codes zwischen  $\geq 200$  und  $< 400$  bedeuten Erfolg, Codes  $\geq 400$  stehen für Fehler.

### Tcp

Versucht ein Socket auf dem angegebenen Tcp-Port zu öffnen.

### Konfigurierbare Parameter

`initialDelaySeconds`: Wartezeit vor erstem Check

`periodSeconds`: Häufigkeit der Tests (default 10)

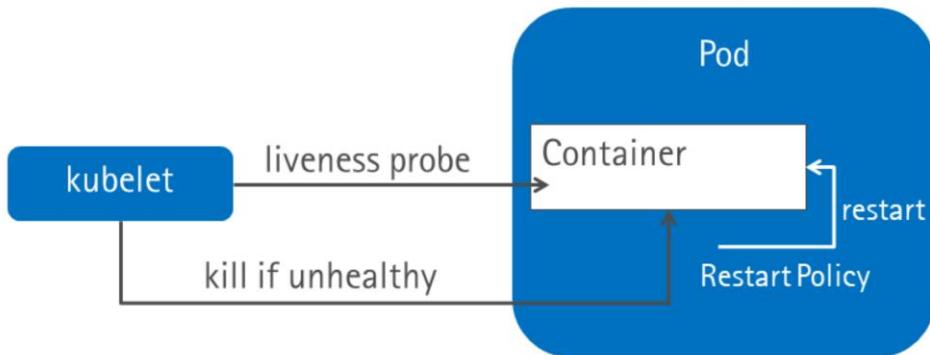
`timeoutSeconds`: Timeout für Warten auf Beenden einer Anfrage (default 1)

`successThreshold`: Nach wie vielen Erfolgen gilt Probe wieder als erfolgreich (default 1)

`failureThreshold`: Nach wie vielen Fehlschlägen gilt Probe als gescheitert (default 3)



## Liveness Probes



Ein Container läuft normalerweise so lange wie sein Hauptprozess. Wird der Hauptprozess beendet, so ist danach auch der Container gestoppt und wird je nach Policy im Pod automatisch neugestartet.

Es kann aber zu Situationen kommen, wo der Prozess der Anwendung noch läuft, aber in einem nichtfunktionalen Zustand festsetzt. In diesem Fall kann über eine fehlgeschlagene LivenessProbe der Container als gescheitert markiert werden, was dann zu einem Beenden des Containers durch die Laufzeitumgebung führt. Auch hier würde der Container je nach RestartPolicy im Pod wieder neu gestartet werden.

Mit diesem Mittel ist es möglich, dass Container, die nicht richtig arbeiten können, weil sie z. B. von einem anderen, gerade nicht verfügbaren Dienst abhängen, wieder automatisch neugestartet werden, bis die Bedingungen irgendwann wieder erfüllt sind. Diese System bezeichnet man als self-healing.

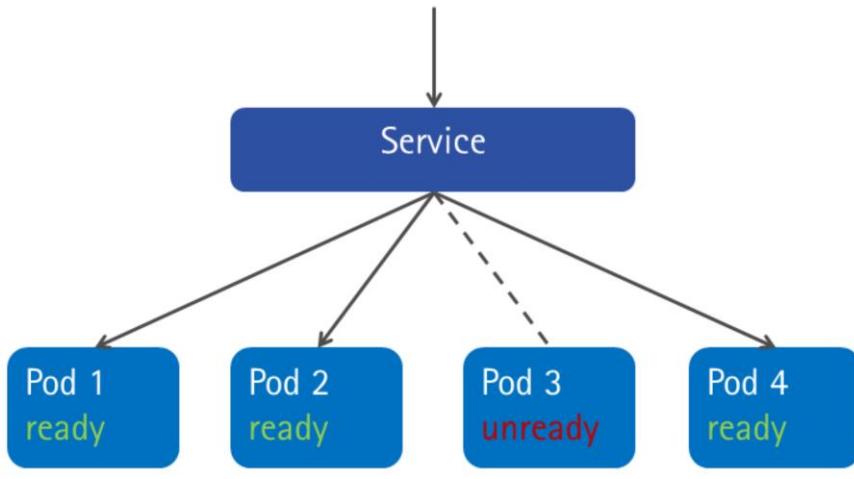


(Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_09/app-deployment.yaml` ist eine Liveness Probe konfiguriert. Die Demo-Anwendung meldet "live", wenn sie ordnungsgemäß gestartet ist. Trägt man für die Liveness Probe eine falsche Adresse ein – z. B. `/xyz/live` – resultiert dies in regelmäßigen Restarts des Containers.



## Readiness Probes



Gibt an, ob die Container eines Pods bereit sind, um Anfragen zu erhalten. Pods, die zwar laufen, aber nicht den Status Ready haben, werden bei der Verteilung von Anfragen über einen Service nicht miteinbezogen. Beim Rolling Update eines Deployments werden alte Pods erst gestoppt, wenn die neuen Ready sind. Wenn für einen Container keine Readiness Probe definiert wurde, so gilt der Container bereits als Ready, wenn er erfolgreich gestartet wurde und noch läuft.

Readiness Probes können also genutzt werden, um Anfragen erst dann an Pods zu leiten, wenn diese bereit sind, die Anfragen zu verarbeiten. Sie können auch genutzt werden, um einen Pod zeitweise unerreichbar zu machen, wenn dort gerade irgendwelche Prozesse laufen, die dies erfordern.



(Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_09/app-deployment.yaml` ist auch eine Readiness Probe konfiguriert. Sie prüft die Funktionsfähigkeit der DB-Verbindung. Nach der anfänglichen Wartezeit wird der Anwendungs-Pod "ready", was mit `kubectl get pod ...` beobachtet werden kann. Wird dann bspw. temporär der DB-Service entfernt, wird der Anwendungs-Pod mit einer gewissen Verzögerung "unready". Dann liefert der Anwendung-Service bei jedem Request den Status 503.



## Übung KUBERNETES\_BASICS\_05

- Fügen Sie der Anwendung `ctr-exercise-rest` Liveness- und Readiness-Probes hinzu
  - Rest-Endpoints:
    - `/health/live`  
liefert Response 200, wenn Anwendung läuft
    - `/health/ready`  
liefert Response 200, wenn DB-Zugriff funktioniert

💻(Projekt `ctr-exercise-rest`):

Anleitung:

1. Fügen Sie im Deployment der Anwendung Liveness und Readiness Probes hinzu.

Die Anwendung nutzt MicroProfile Health mit den Endpoints `/health/live` und `/health/ready`. :

- `/health/live` liefert dann ein positives Ergebnis (Status 200), wenn der Application Server gestartet und die Anwendung deployt werden konnte. Sollte zum Startzeitpunkt der Anwendung die DB noch nicht verfügbar sein, schlägt das Deployment fehl.
- `/health/ready` liefert ein positives Ergebnis, wenn die Anwendung auf die DB zugreifen kann.

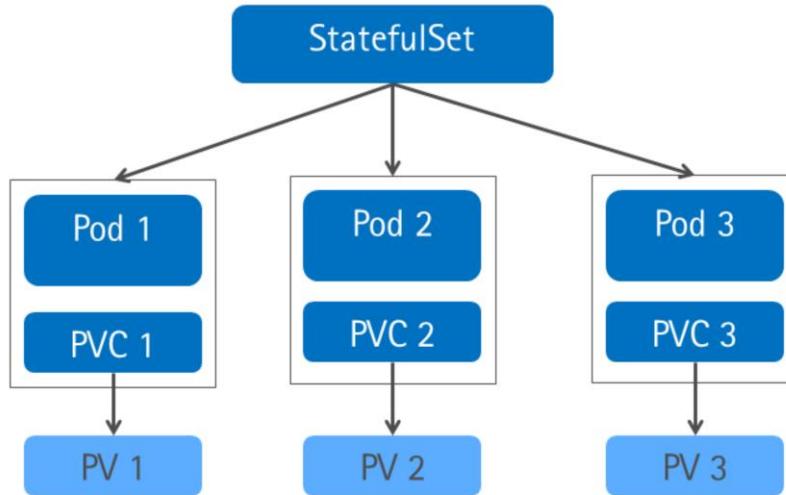
Achtung: WildFly veröffentlicht die Health-Endpoints auf dem Port 9990.

Musterlösung: `src/main/k8s/exercise_05`

Sollte nun bei Anwendungsstart die DB noch nicht verfügbar sein, wird der Anwendungscontainer neu gestartet. Wird die DB später unerreichbar, wechselt der Anwendungspod nach unready.



## StatefulSets



Ein StatefulSet ist ein Controller für zustandsbehaftete Anwendungen, wo jeder Pod eine feste Identität hat und eigene Volumes zugewiesen bekommt. Pods werden wie beim Deployment (bzw. ReplicaSet) mit Hilfe eines Templates erzeugt.

Anwendungsfälle sind clusterfähige Anwendungen wie Apache Kafka, deren Instanzen sich untereinander über (feste) IP-Namen koordinieren und die pro Instanz einen eigenen persistenten Speicher benötigen.

### Pod Identity

Die Pods erhalten einen Namen, der aus dem Namen des StatefulSets und einer angehängten Ordnungszahl besteht (`setName-0`, `setName-1`, ...). Wird für das Set auch ein Service erstellt, so haben die Pods sogar fest zugewiesene DNS-Namen (`serviceName.setName-0`, `serviceName.setName-1`, ...).

### Volume Claims

Über ein im StatefulSet definiertes Template werden PVCs für jeden Pod angelegt. Die Pods sind jeweils mit dem an ihre Identität geknüpften PVCs gebunden. Beim Löschen des StatefulSets gehen die PVCs und PVs nicht verloren, sondern werden bei einem erneuten Start des StatefulSets bzw. der Pods wieder gebunden.

---

#### (Projekt ctr-demo-rest):

Die K8s-Manifeste in `src/main/k8s/demo_10` definieren ein StatefulSet für die Demoanwendung mit zwei Pods. Man kann beim Start beobachten:

- Die Pods werden nacheinander gestartet und "durchnummeriert"
- Pro Pod wird ein PVC erzeugt und ein PV gebunden

Unter `rest-demo.localtest.me/resources/modifiableFile` kann der Inhalt einer Datei im gebundenen Volume abgefragt werden. Sie wird beim ersten Aufruf erzeugt und mit einem Text gefüllt, der Hostname und IP-Adresse enthält. Diese Werte sind pro Pod unterschiedlich und bleiben auch nach einem Neustart erhalten.



## Übung KUBERNETES\_BASICS\_06

- Nutzen Sie für die MySQL-Datenbank in `ctr-exercise-rest` ein Stateful Set anstelle eines Deployments
  - `volumeClaimTemplate` statt eigenständigem `PersistentVolumeClaim`
  - `replicas: 1`

💻 (Projekt `ctr-exercise-rest`):

Anleitung:

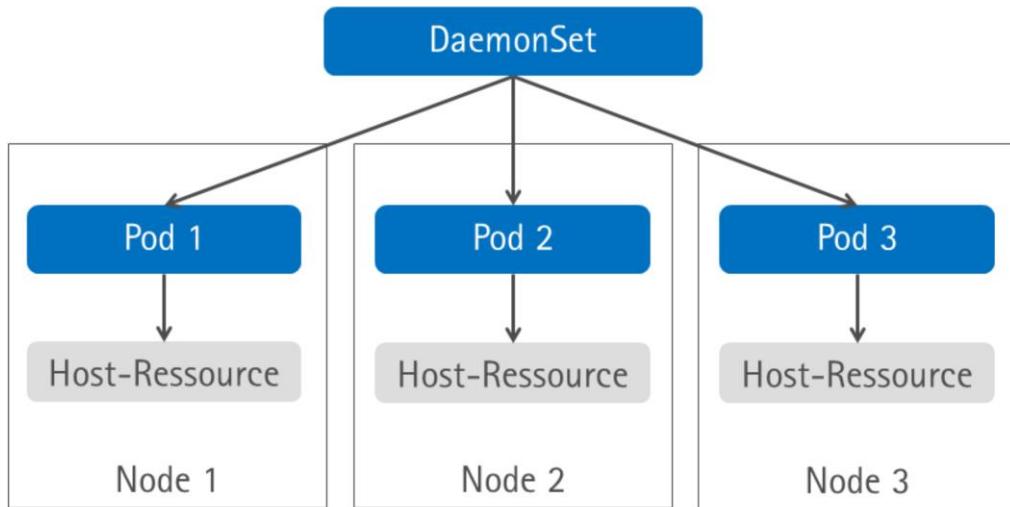
Wird in der bisherigen Übungsanwendung das Persistent Volume Claim (versehentlich) gelöscht, sind die in der DB gespeicherten Daten verloren. Ein Stateful Set, das statt eines eigenständigen PVCs ein Template nutzt, lässt PVs beim Löschen intakt und ist daher für DB-Anwendungen i. A. besser geeignet als ein Deployment:

1. Ändern Sie das Deployment der MySQL-DB in ein Stateful Set:
  - Definieren Sie ein `volumeClaimTemplate` analog zum bisherigen PVC.
  - Referenzieren Sie den dafür genutzten Namen direkt als `name` in `volumeMounts` (d. h. die separate Angabe von `volumes` entfällt).
  - Referenzieren Sie den DB-Service (`spec.serviceName`).
  - Die MySQL kann nicht im Cluster betrieben werden. Starten Sie also nur eine Instanz.
2. Entfernen Sie das bisherige eigenständige PVC.

Musterlösung: `src/main/k8s/exercise_06`



## DaemonSets

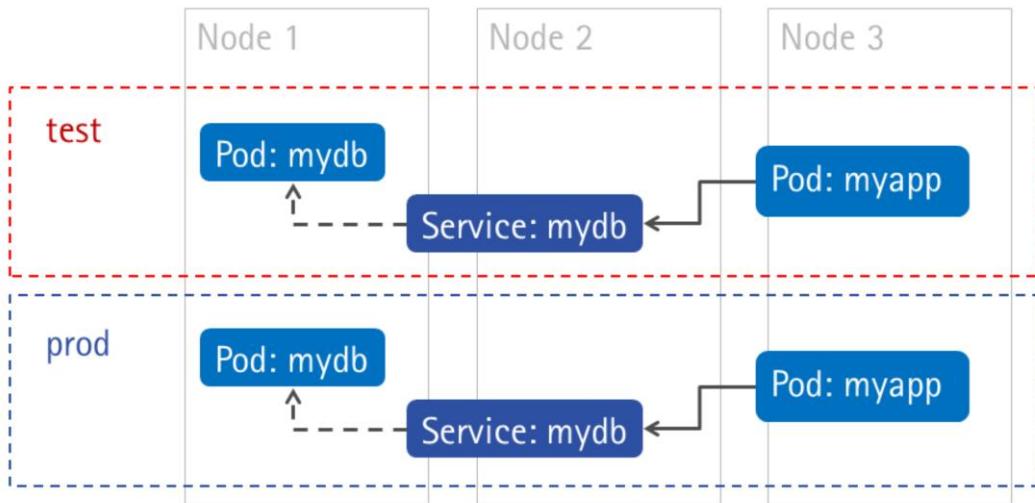


Ein DaemonSet sorgt dafür, dass eine Pod-Kopie auf jedem Node bzw. auf bestimmten Nodes im Cluster läuft. Pods werden erzeugt oder entfernt, wenn Knoten im Cluster hinzugefügt oder herausgenommen werden. Per Default wird auf jedem Knoten ein Pod gestartet; soll dies auf bestimmte Nodes beschränkt werden, so kann dafür ein NodeSelector im DaemonSet definiert werden.

Der Einsatz eines Daemonsets macht Sinn, wenn man einen Dienst auf jedem Knoten laufen lassen möchte. Dies ist zum Beispiel für das Einsammeln von Docker Logfiles auf den Nodes notwendig oder für das Weiterleiten von Daten für das Monitoring der Nodes. Auch ein verteiltes Storage-System könnte man so im Cluster ausrollen. Häufig wird von den Containern aus direkt auf Host-Ressourcen zugegriffen wie z. B. Verzeichnisse oder den Netzwerkstack.



## Namespaces



Namespaces definieren voneinander isolierte Namensräume/Bereiche, mit denen es möglich ist, virtuelle Cluster im tatsächlichen Cluster aufzubauen.

Die meisten Kubernetes-Objekte werden immer in einem Namespace angelegt, ausgenommen hiervon sind Low-Level-Objekte wie Namespaces selber, Nodes und PersistentVolumes. Um Objekte in einem Namespace abzulegen oder daraus abzufragen, kann der Parameter `--namespace (-n)` beim `kubectl`-Befehl angegeben werden:

```
kubectl apply --namespace=<namespace> -f ....
```

Wird kein Namespace angegeben, so wird der Namespace `default` verwendet. Es ist möglich, einen Namespace für alle folgenden Befehle des aktuellen Kubernetes-Kontextes zu setzen:

```
kubectl config set-context \
  $(kubectl config current-context) -- \
  namespace=<namespace>
```

Namespaces können wie alle Kubernetes Objekte als YAML-Files eingespielt oder über `kubectl create namespace <name>` angelegt werden.

Die DNS-Namen für Services stehen innerhalb eines Namespaces mit dem einfachen Namen zur Verfügung: Ein Hostname `myservice` würde innerhalb des Namespaces, wo er angesprochen wird, aufgelöst. Dadurch ist es möglich, bei Verwendung der einfachen Hostnamen die gleiche Anwendungsgruppe in unterschiedlichen Namespaces einzuspielen, weil die Pods jeweils die Services aus ihrem Namespace referenzieren. Um einen Service aus einem anderen Namespace anzusprechen, muss der Namespace mit im Hostnamen angegeben werden: `myservice.mynamespace`.



## Namespaces – Vorgehensweisen

- Namespace pro Team
- Namespace pro Umgebung (`prod`, `test`, ...)
- Namespace pro Projekt und Umgebung

Es sind verschiedene Vorgehensweisen für die Verwendung von Namespaces denkbar.

### Namespace pro Team / Organisation

Verwendung der Namespaces als virtuelle Cluster innerhalb von einem zentralen Cluster im Unternehmen. Teams bekommen dann Zugriff nur auf die ihnen zugeteilten Namespaces und können in diesen ihre Anwendungen betreiben.

### Namespace pro Umgebung

Namespaces für Umgebungen wie Produktion oder Test machen es einfach, Anwendungen, die untereinander kommunizieren müssen, in unterschiedlichen Stages zu betreiben. Aufgrund der Namensauflösung werden immer die restlichen Services aus derselben Umgebung gefunden. Auch kombinierbar mit der Organisation nach Teams, wenn es für jedes Team die entsprechenden Umgebungen gibt.

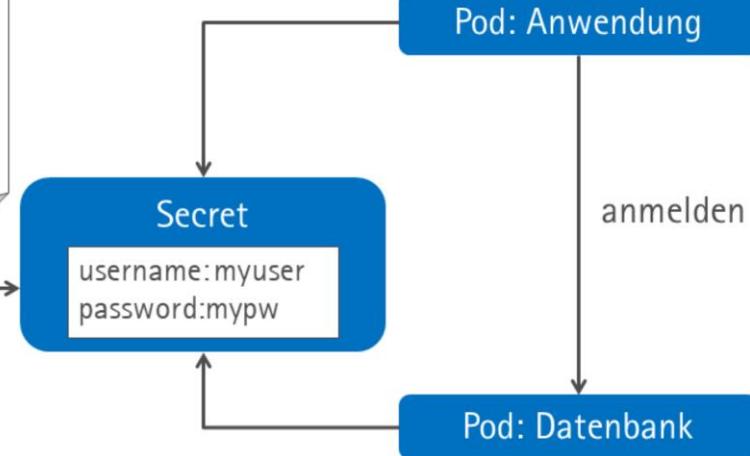
### Namespace pro Projekt (und Umgebung)

Zu einem Projekt/einer Anwendung gehören eine Reihe von Kubernetes-Objekten. Diese könnten in einem Anwendungsnamespace zusammengefasst werden. Für die Umgebungen wie Prod und Test müssten dann jeweils für jedes Projekt Namespaces angelegt werden. Bietet den Vorteil, dass Anwendungen isoliert voneinander sind und ermöglicht einfach anwendungsspezifische Demo-Umgebungen bereitzustellen. Führt zu dem Nachteil, dass Anwendungen sich gegenseitig außerhalb ihres Namespaces aufrufen müssen.



## Secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  username: myuser
  password: mypw
```



Daten wie Passwörter und Zertifikate sollten nicht direkt im Anwendungsquellcode bzw. im Dockerimage eingebaut werden. In Kubernetes gibt es hierfür Secrets als Objekte, welche solche Daten in Form von Schlüssel-und-Wert-Paaren enthalten. Diese Secrets können dann von Pods eingebunden werden, um die gewünschten Werte daraus zu beziehen.

### Secrets erstellen

Secrets können entweder mit dem `kubectl create secret` Befehl erzeugt oder per `kubectl apply` aus einer YAML Datei heraus eingespielt werden. Die Werte eines Secrets sind beim Typ `Opaque` Base64-encodiert, müssen also, wenn ein YAML-File genutzt wird, schon dort in codierter Form eingetragen sein. Encodieren unter Linux: `echo -n <myval> | base64`. Soll der Wert in der Datei in nicht codierter Form stehen und erst beim Anlegen/Updateen encoded werden, so muss das Feld `'stringData'` anstatt `'data'` verwendet werden. Auch ein mehrzeiliger Inhalt ist in diesem Falle über die entsprechende YAML-Schreibweise möglich.

### Secrets-Sicherheit

Secrets sind für jeden, der Zugriff auf die Objekte hat, im Klartext sichtbar, dieser Zugriff kann natürlich über RBAC-Regeln festgelegt werden. Standardmäßig werden Secrets zudem unverschlüsselt im etcd abgelegt, sind also für jeden, der darauf Zugriff hat, sichtbar. Sollen die Secrets verschlüsselt im etcd abgelegt werden, so kann dies mit einer `EncryptionConfiguration` festgelegt werden (seit v1.13).



## Secrets einbinden

### Als Volume



```
containers:
  volumeMounts:
    - name: dbsecret
      mountPath: "/etc/dbsecret"
volumes:
  - name: dbsecret
    secret:
      secretName: mysecret
```

### Als Umgebungsvariable

```
USERNAME=myuser
PASSWORD=mypw
```

```
env:
  - name: USERNAME
    valueFrom:
      secretKeyRef:
        name: mysecret
        key: username
```



Secrets können auf zwei unterschiedliche Weisen eingebunden werden.

### Volume

Secrets können als Volumes in den Pods konfiguriert und dann über ein mounting in die Container unter einem Pfad eingehängt werden. Dabei wird für jeden Schlüssel in dem Secret eine Datei mit demselben Namen angelegt, welche den Wert als Inhalt enthält. Die Namen der Dateien können bei Bedarf auch für jeden Schlüssel umkonfiguriert werden. Diese Vorgehensweise wird vor allem für Zertifikate oder Konfigurationsdateien verwendet.

### Umgebungsvariable

Werte eines Secrets können auch als Umgebungsvariablen in den Containern bereitgestellt werden. Hierfür muss bei der Definition einer Environment-Variablen anstelle des values direkt, über valueFrom.secretKeyRef.name ein Secret referenziert und mit valueFrom.secretKeyRef.key ein Wert daraus ausgewählt werden. Es gibt auch die Möglichkeit, alle Werte des Secrets auf einmal als Umgebungsvariablen mit dem Schlüsselnamen bereitzustellen.

---

💻 (Projekt ctr-demo-rest):

In src/main/k8s/demo\_11 sind gegenüber der Demo 09 geändert:

- DB-Name, User und Passwort sind als Secret definiert (db-secret.yaml)
- app-deployment.yaml und db-deployment.yaml enthalten diese Werte nicht mehr



## Alternativen für Verwalten von Secrets

- Cloud
  - AWS Secrets Manager
  - Cloud KMS
  - Azure Key Vault
- HashiCorp Vault
- SealedSecrets

Kubernetes Secrets sind einfach in der Handhabung, haben aber auch einige Sicherheitsmängel. Zum einen müssen die YAMLs irgendwo verwaltet werden (z. B. Git), wo die Daten dann im Klartext (oder Base64-encoded) vorliegen würden und zum anderen werden sie einfach als Umgebungsvariable in den Container eingebunden, wo sie von jedem mit den entsprechenden Rechten gesehen werden können.

Als Alternative oder Ergänzung können zum einen die Lösungen der Cloudanbieter herangezogen werden oder, wenn die Anwendung nicht in einer entsprechenden Cloud läuft, Open-Source-Produkte genutzt werden.

### HashiCorp Vault

Vault ist eine Software für das Verwalten von Secrets, welche es unter anderem auch ermöglicht, Werte direkt aus den Anwendungen heraus aufzulösen, ohne diese in Dateien zu schreiben oder als UmgebungsvARIABLEN zu setzen. Es gibt für Vault Hilfsmittel für die Integration mit Kubernetes.

### Bitnami SealedSecrets

Ermöglicht es, Secrets im Git zu verwalten in Form von verschlüsselten SealedSecret-Objekten. Dafür wird im Cluster ein Dienst betrieben, welcher einen öffentlichen Schlüssel für das Generieren eines SealedSecrets erzeugt und einen internen privaten Schlüssel für das Entschlüsseln. Ein Controller sorgt dafür, dass bei Einspielen eines SealedSecret die Werte entschlüsselt und ein normales Secret angelegt werden.



## Übung KUBERNETES\_BASICS\_07

- Lagern Sie die Zugangsdaten für die DB in ein Secret aus
  - Base64 Strings erzeugen:  
`echo -n '<text>' | base64`

💻 (Projekt `ctr-exercise-rest`):

Anleitung:

1. Definieren Sie ein Secret, das die bislang im Klartext im DB-Deployment enthaltenen DB-Parameter enthält.  
Zum Erzeugen der notwendigen Base64-kodierten Strings können Sie den Befehl  
`echo -n "zuKodierenderText" | base64` nutzen.
2. Ändern Sie das DB-Deployment so ab, dass die Angaben im Secret statt der Klartext-Angaben genutzt werden.
3. Verfahren Sie entsprechend mit dem Anwendungsdeployment.

Achtung: Der Wert der Variablen `MYSQL_DATABASE` unterscheidet sich zwischen Anwendung (Client-Sicht) und DB bzw. Secret (Server-Sicht) durch den angefügten Parameter `?useSSL=false`. Holen Sie sich daher den Wert aus dem Secret zunächst in eine zusätzliche Variable (z. B. `DATABASE`) und definieren Sie `MYSQL_DATABASE` mit deren Hilfe (also: `$(DATABASE) ?useSSL=false`).

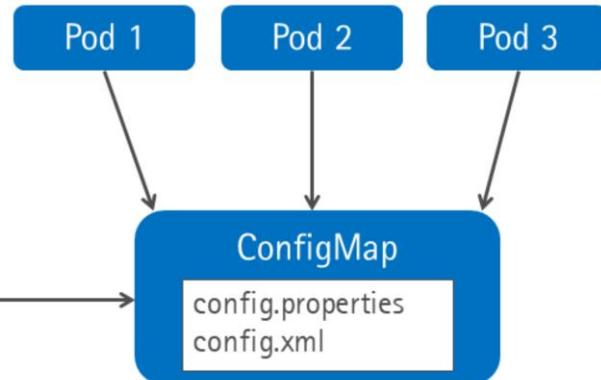
Musterlösung: `src/main/k8s/exercise_07`



## ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfig
data:
  config.properties: |
    myprop1=myval1
    ...
  config.xml | 
    <xml>
    ...

```



ConfigMaps sind dafür gedacht, um Konfiguration aus den Container-Images auszulagern. Sie erlauben es, Konfiguration zentral für mehrere Pods abzulegen oder im Zusammenspiel mit Namespaces unterschiedliche Konfiguration für die Umgebungen bereitzustellen.

### ConfigMaps erstellen

ConfigMaps funktionieren ähnlich wie Secrets, können aber nicht verschlüsselt abgelegt werden. Unterhalb von dem Feld `data` können Schlüssel-Wert-Paare definiert werden, es ist hier auch möglich, mehrzeiligen Text anzugeben, womit es möglich wird, komplett Konfigurationsdateien in der ConfigMap abzulegen.

Für das Erstellen von ConfigMaps stehen verschiedene `kubectl`-Befehle bereit, um eine Map aus Literalen, Dateien oder Verzeichnissen zu erstellen.

### ConfigMaps einbinden

Für das Einbinden in Pods gibt es hier auch die Möglichkeiten wie von den Secrets bekannt. Im Falle von Dateien, die in der Map abgelegt sind, kann die Map als Volume eingebunden werden. Ansonsten ist es auch möglich, Umgebungsvariablen mit Werten aus der Map zu belegen.

Änderungen an der ConfigMap sind bei Einbinden per Volume in einem laufenden Pod nach einiger Zeit verfügbar, für ein Update der Umgebungsvariablen ist aber ein Neustart erforderlich.

---

(Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_12` wird eine ConfigMap definiert und im Anwendungsdeployment referenziert – einmal als Umgebungsvariable und einmal als Volume. Unter `rest-demo.localtest.me/resources/textFile` wird der Dateiinhalt aus der CM angezeigt.



## Zugriff auf Daten des Pod im Container

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - name: A
      env:
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
```

Name des Nodes, auf dem der Pod läuft

Dem Pod zugewiesene IP



Informationen über den Pod können den Containerprozessen in Form von Umgebungsvariablen bereitgestellt werden. Hierfür wird im `valueFrom` mit `fieldRef` ein Feld aus dem Pod-Objekt referenziert.

Hierüber ist es unter anderem möglich, im Container zu erfahren, auf welchem Node der Pod läuft, welche IP-Adresse er im Cluster hat, in welchem Namespace er läuft usw.

Auch Informationen über den Container selbst können so als Umgebungsvariable zur Verfügung gestellt werden. Hierfür muss mit `resourceFieldRef` anstatt `fieldRef` gearbeitet werden und bei `container-name` der Name des gewünschten Containers aus dem Pod angegeben werden.



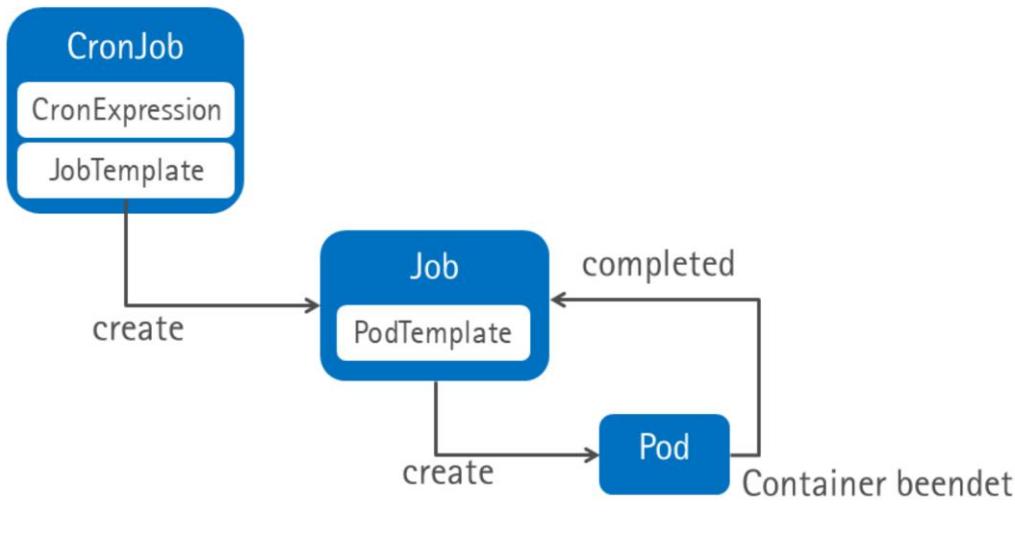
(Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_13` wird die Demo-Anwendung mit einigen Infos aus der Laufzeitumgebung versorgt.

Sie können unter `rest-demo.localtest.me/resources/podInfo` abgefragt werden.



## Jobs und CronJobs



## Jobs

Jobs definieren Aufgaben, die im Cluster ausgeführt werden sollen und nach dem Abschließen dieser Aufgabe als erledigt gelten. Technisch erzeugt ein Job einen Pod mit einem Container, welcher die Arbeit erledigt. Wenn der Containerprozess erfolgreich beendet ist, dann gilt auch der Job als erledigt. Im Fehlerfalle wird der Job nochmals ausgeführt (also der Pod neu erzeugt), solange bis ein Erfolg eintritt oder das `backoffLimit` erreicht wurde. Jobs werden direkt durch Anlegen des Objektes ausgeführt.

## CronJobs

CronJobs erlauben das zeitgesteuerte Anlegen von Jobs. Die Jobs werden mit Hilfe eines JobTemplates zum durch eine CronExpression definierten Zeitpunkt erzeugt.

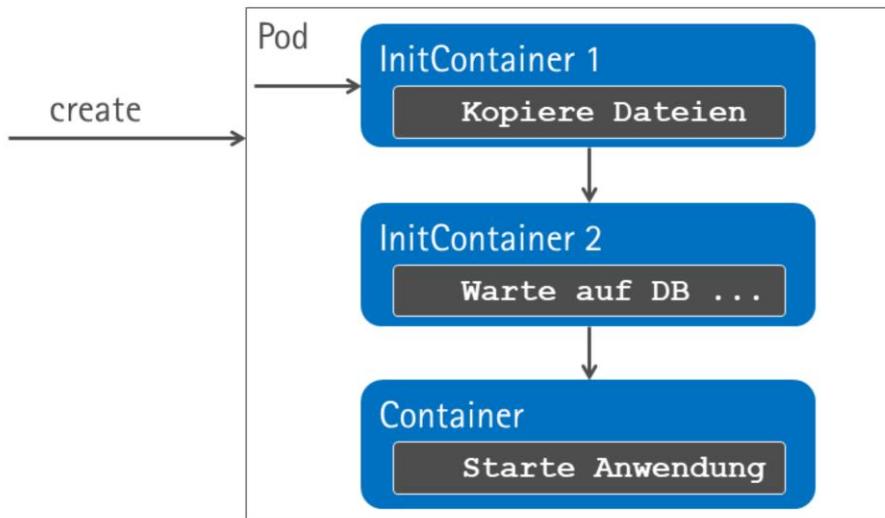
---

💻 (Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_14` wird ein CronJob definiert, der einmal pro Minute einen PUT-Request auf den Endpoint `resources/modifiableFile` der Demo-Anwendung ausführt und damit dort einen Text ablegt, der u. a. den aktuellen Zeitstempel enthält.



## InitContainers



Neben den normalen Containern des Pods besteht auch die Möglichkeit, eine Reihe von InitContainern zu definieren. Diese InitContainer sind dafür gedacht, um darin Aufgaben für die Initialisierung der Anwendung bzw. auf das Warten auf Bedingungen zu erledigen.

Die InitContainer werden in der Reihenfolge durchlaufen, wie sie in der Pod spec definiert sind. Endet der Hauptprozess eines InitContainers, so wird mit dem nächsten fortgefahrt, bis alle InitContainer erfolgreich beendet wurden, dann werden die eigentlichen Container der Anwendung gestartet.

InitContainer können auch auf Volumes des Pods zugreifen und sind daher verwendbar, um z. B. Dateien oder Verzeichnisse vorzubereiten.

### Warten auf andere Anwendungen im Cluster

Ein InitContainer kann genutzt werden, um auf eine andere Anwendung oder auch eine Jobcompletion zu warten. Hierfür kann der Befehl `kubectl wait` verwendet werden. Dieser benötigt allerdings entsprechende Rechte im Cluster, um Pods usw. abfragen zu dürfen.

---

(Projekt `ctr-demo-rest`):

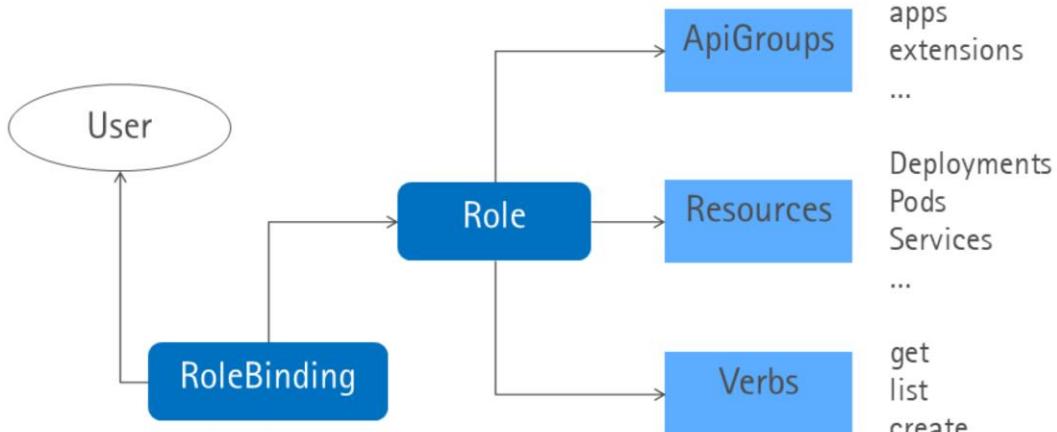
Änderungen in `src/main/k8s/demo_15` gegenüber der Demo 11:

- Das Anwendungsdeployment definiert einen Init-Container, der wartet, bis der DB-Pod ready ist.
- Um dem Init-Container Zugriffsrechte einzuräumen, sind ein Service-Account sowie Rollendefinitionen und Zuordnungen ergänzt worden (Erklärung folgt).

Beim Start der Anwendung kann man nun beobachten, dass der Hauptcontainer der Anwendung erst gestartet wird, wenn die DB ready ist.



## RBAC



Es ist möglich, in einem Kubernetes-Cluster über Aktivierung von Role Based Access Control (kurz: RBAC) den Zugriff auf die Ressourcen zu beschränken. Die Berechtigungen werden über Objekte des Typs Role und RoleBinding festgelegt.

### Role und ClusterRole

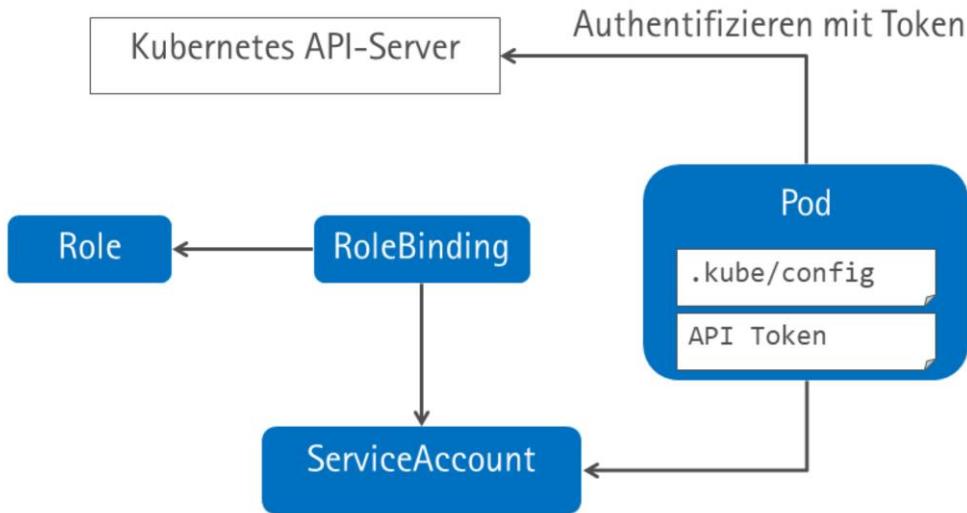
Rollen definieren eine Reihe von Aktionen auf Typen von Kubernetes-Objekten. Normale Rollen Roles gelten innerhalb eines Namespaces, clusterweite Rollen ClusterRoles können Aktionen auf Clusterebene festlegen.

### RoleBinding und ClusterRoleBinding

Um die Rollen einem Benutzer oder einer Gruppe zuzuordnen, wird ein Role Binding benötigt, welches die Zuordnung definiert. Auch hier gibt es wieder Zuweisungen innerhalb eines Namespaces RoleBinding und auf Clusterebene ClusterRoleBinding.



## ServiceAccounts



ServiceAccounts ermöglichen es Prozessen im Container, sich am API-Server zu authentifizieren und somit Aufrufe an diesen durchzuführen. Einem Service Account werden mit Hilfe eines RoleBindings Rollen zugewiesen und der ServiceAccount wird wiederum in der Pod Spec referenziert. Wird kein ServiceAccount angegeben, so wird der Default-Account des Namespaces dafür verwendet.

In den Containern des Pods werden die Zugangstokens des ServiceAccounts für den API-Server sowie eine .kube/config-Konfiguration, welche auf den Cluster zeigt, in dem der Pod läuft, eingebunden. Prozesse in den Containern können somit z. B. über kubectl einfach gemäß ihren Berechtigungen Aktionen im Cluster ausführen.



(Projekte ctr-infra und ctr-demo-rest):

In der Demo 15 benötigt der Init-Container erweiterte Rechte, um auf den Status eines anderen Pods zugreifen zu dürfen. Dazu wird mittels src/main/k8s/service-accounts/seminar-account-and-roles.yaml im Projekt ctr-infra ein Service Account und eine passende Role definiert und miteinander verknüpft.

Dieser Service Account wird in src/main/k8s/demo\_15 des Projektes ctr-demo-rest für den Applikations-Pod verwendet.



## Übung KUBERNETES\_BASICS\_08 (nur wenn ausreichend Zeit)

- Warten Sie auf die Datenbank mit einem InitContainer

### 💻 (Projekt `ctr-exercise-rest`):

Anleitung:

1. Ergänzen Sie das Anwendungsdeployment um einen Init-Container:

- Image: `lachlanevenson/k8s-kubectl:v1.14.1`
- Befehl zum Warten:  
`kubectl wait --timeout=60s --for=condition=ready \`  
`pod -l label-expression -n namespace`

Für die `label-expression` nutzen Sie ein passendes Label des DB-Deployments.

Als `namespace` können Sie zunächst `default` eintragen, aber besser ist es, den eigenen Namespace aus einer `FieldRef` zu ermitteln (vgl. Demo).

Nutzen Sie für den Pod den Service-Account aus der Demo (s. vorige Seite).

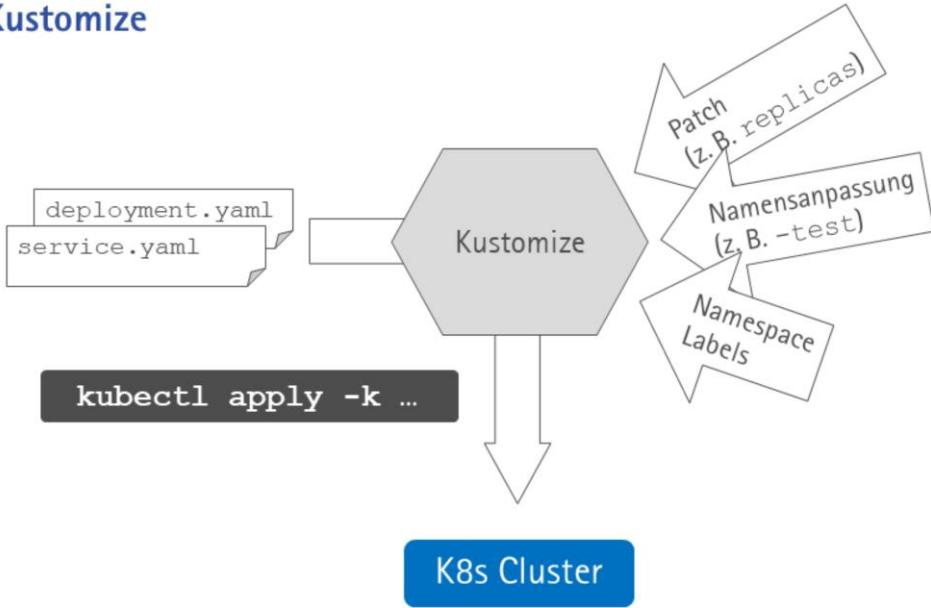
2. Nun gibt es noch das Problem, dass die MySQL-Datenbank zu früh als ready angesehen wird, was vor allem beim Neueinspielen der Anwendung auffällt, wenn erst noch das Schema eingerichtet und initialisiert werden muss. Hier kann mit einer entsprechenden Readiness-Probe für die Datenbank dafür gesorgt werden, dass diese erst nach der Initialisierung verfügbar ist:

```
mysql -h 127.0.0.1 -u <user> -p<password> \
-D <database> -e "SELECT 1"
```

Musterlösung: `src/main/k8s/exercise_07`



## Kustomize



Kubernetes hat keinen scharf definierten Anwendungsbegriff. Es ist nur üblich, dass man die zu einer Anwendung gehörenden K8s-Manifeste in einem Ordner sammelt, den man mit `kubectl apply -f` in den Cluster einspielt.

In der Praxis reicht diese Gruppierung der YAML-Files nicht aus. Soll bspw. eine Anwendung in der Testumgebung und später in Produktion laufen, bleibt sie zwar im Kern gleich, bekommt aber meist andere Parameter mitgegeben, z. B.

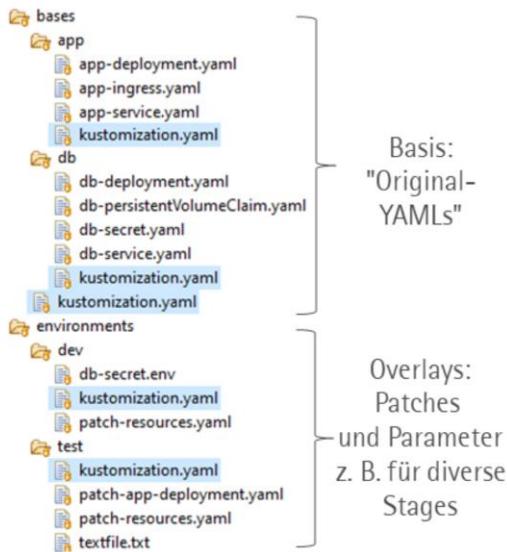
- Im Test werden nur 2 Replicas benötigt, in Prod 10.
- die Namen von Deployments, Services etc. sollen in Test um `-test` ergänzt werden,
- für Test wird ein anderer Namespace genutzt
- alle Objekte sollen zusätzliche einheitliche Labels erhalten

Die YAMLs müssen somit beim Einspielen in den K8s-Cluster passend manipuliert werden. Für diese Aufgabe wird häufig Helm genutzt, wobei in den YAMLs Template-Ausdrücke genutzt werden, die Helm beim Deployment durch Werte aus einer Parameterdatei ersetzt. Durch das Templating werden die YAMLs allerdings äußerst unübersichtlich, was den Aufwand für Erstellung und Wartung in die Höhe treibt.

Seit Kubernetes 1.14 ist eine Alternative – Kustomize – in `kubectl` integriert. Kustomize hat ebenfalls Parameterdateien (`kustomization.yaml`), nutzt aber anstelle von Templating sog. Overlays, die strukturell Ausschnitte von K8s-Manifesten sind und in der Kombination die entsprechenden Werte der Ausgangs-YAMLs ersetzen. Sie sind leicht verständlich und können mit üblichen Werkzeugen bearbeitet werden.



## Kustomize



```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- app-deployment.yaml
- app-ingress.yaml
- app-service.yaml

commonLabels:
  app.kubernetes.io/name: rest-demo
  app.kubernetes.io/component: server
```

```
kubectl apply -k \
.../environments/dev
```

```
kustomize build \
.../environments/dev
```

Kustomize arbeitet mit Verzeichnissen, die neben den K8s-Manifesten und Patchfiles jeweils eine Datei namens `kustomization.yaml` enthalten. Eine genaue Ordnerstruktur ist nicht vorgeschrieben, wobei die oben gezeigte recht üblich ist.

Unter `bases` finden sich die Original-K8s-Manifeste. Man kann dabei durchaus mehrere Basisverzeichnisse kombinieren, um die Gesamtanwendung übersichtlicher zu strukturieren – oben aufgeteilt in Anwendung und Datenbank.

Unter `environments` liegen nun Verzeichnisse mit Patches für die K8s-Manifeste und Parameters für bestimmte Zielumgebungen wie bspw. `dev`, `test`, `prod`, ...

Jedes Verzeichnis enthält eine `kustomization.yaml`, die die zugehörigen Ressourcendateien referenziert und ergänzende Parameter definiert.

Die mit Hilfe von Kustomize angepassten Objekte können mit

`kubectl apply -k ...`  
in den K8s-Cluster eingespielt werden.

Möchte man zunächst das Ergebnis der Kustomization einsehen, ohne es direkt einzuspielen, kann man dies mit

`kustomize build ...`  
tun. Das Ergebnis erscheint dann auf der Standardausgabe. Das Kommandozeilen-Tool `kustomize` ist allerdings nicht Teil üblicher K8s-Distributionen und muss meist separat installiert werden.



## Kustomize

### ☰ Dokumentation: [kustomize.io](https://kustomize.io)

- ☰ Ressourcen lokal oder per Web-URL referenzierbar
- ☰ Overlays (~Patches für K8s-Manifeste)
- ☰ Gemeinsame Label und Annotationen
- ☰ Generatoren für ConfigMaps und Secrets
- ☰ Bereitstellung von Variablen

Kustomize hat die u. a. folgenden Features:

- Die `kustomization.yaml` können Ressourcen auch per Web-URL referenzieren.
- Die effektiven Objektdefinitionen werden aus Basis und Overlays kombiniert.
- Es können gemeinsame Labels und Annotationen für alle Objekte definiert werden.
- ConfigMaps und Secrets haben häufig als Ausgangspunkt eine Liste von Key/Value-Paaren oder eine Datei. Kustomize kann daraus automatisiert ConfigMaps und Secrets erstellen.
- Werte aus Ressourcen-Definitionen können als Variablen bereitgestellt werden, die in anderen Ressourcen wiederum genutzt werden können.

Die Beschreibung sämtlicher Features von Kustomize würde hier den Rahmen sprengen. Bitte nutzen Sie die Dokumentation unter <https://kustomize.io/>

---

### (Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_16` finden Sie die auf der vorigen Seite abgebildete Dateistruktur. Im Vergleich zu den bisherigen Demos sind nun die Aspekte (Haupt-)Anwendung und Datenbank übersichtlich unter `bases` aufgeteilt. Die Overlays unter `environment` patchen und konfigurieren die Anwendung nun für die Entwicklungsumgebung und die Test-Stage. Darin finden sich auch Beispiele für ConfigMap- und Secret-Generatoren. Ein Deployment in der Entwicklungsumgebung erfolgt mit dem folgenden Befehl:

```
kubectl apply -k src/main/k8s/demo_16/environments/dev
```



## Empfohlene Labels

### metadata:

#### labels:

```
app.kubernetes.io/name: rest-demo-app  
app.kubernetes.io/instance: rest-demo-1  
app.kubernetes.io/version: "1.2.3"  
app.kubernetes.io/managed-by: tool  
app.kubernetes.io/component: server  
app.kubernetes.io/part-of: rest-demo
```

In Kubernetes gibt es kein direktes Konzept einer Anwendung, es ist allerdings möglich, über Labels Ressourcen entsprechend als Teil einer Anwendung zu markieren. Diese Metainformationen könnten dann von Werkzeugen ausgewertet werden.

Es gibt einige empfohlene Labels, welche man nutzen kann, um Ressourcen als Teil einer Anwendung zu kennzeichnen.

app.kubernetes.io/name: Name der Anwendung

app.kubernetes.io/instance: Eindeutiger Name für bestimmte Instanz

app.kubernetes.io/version: Version der Anwendung

app.kubernetes.io/managed-by: Werkzeug zur Verwaltung der Anwendung

app.kubernetes.io/component: Aufgabe der Anwendung in der Architektur

app.kubernetes.io/part-of: Übergeordnete Anwendung



## Übung KUBERNETES\_BASICS\_09 (nur wenn ausreichend Zeit)

- Erstellen Sie für die Anwendung Kustomization-Files
  - Datenbank und Anwendung als separate Anwendungen
  - Kustomization-File, welches diese zusammenfasst
    - Hinzufügen eines gemeinsamen Labels: app=exercise

💻(Projekt ctr-exercise-rest):

Anleitung:

1. Bauen Sie eine Dateistruktur ähnlich der gezeigten Demo auf. Sehen Sie zwei Bases vor für die eigentliche Anwendung und für die Datenbank.
2. Verschieben Sie die bereits vorhandenen Manifeste der (eigentlichen) Anwendung in das dafür erzeugte Basisverzeichnis. Erzeugen Sie dort auch ein kustomization.yaml, das die Manifeste referenziert.
3. Verfahren Sie analog mit dem Datenbank-Teil der Anwendung.
4. Erzeugen Sie ein kustomization.yaml im Basisverzeichnis, das die beiden Unterverzeichnisse referenziert. Fügen Sie dort als gemeinsames Label app=exercise hinzu.
5. Prüfen Sie, ob Sie die Anwendung nun über kubectl apply -k ... deployen können.
6. Zusatzaufgaben:
  - Erzeugen Sie zwei Overlays für test und prod und lassen Sie in prod den app-Anteil mit mehreren Replicas laufen.
  - Lassen Sie das DB-Secret mit Hilfe des Secret-Generators erstellen.

Musterlösung: src/main/k8s/exercise\_08



## Images von privaten Registries

- Zugriffstoken auf Nodes
  - Ermöglicht jedem, Images von Registry zu pullen
- ImagePull-Secret
  - Kann in Pods referenziert werden

```
apiVersion: v1
kind: Secret
metadata:
  name: my-pull-secret
data:
  .dockerconfigjson: eyJodHRw... JoQUI6RTIifXo=
type: kubernetes.io/dockerconfigjson
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - image: <your-private-image>
      imagePullSecrets:
        - name: my-pull-secret
```

Wenn Images von einer nicht öffentlichen Registry im Cluster gestartet werden sollen, so müssen Kubernetes die Daten für die Anmeldung an der Registry bekannt sein.

### Konfiguration auf Nodes

Bei Login an einer Registry über Docker wird ein File \$HOME/.docker/config.json angelegt. Es enthält die Zugangstokens und kann nun entsprechend verteilt werden. Auf den Nodes kann es dann z. B. unter /var/lib/kubelet/config.json abgelegt werden, damit diese Registry im Cluster ohne explizite Anmeldung genutzt werden kann.

### Image Pull Secrets

Über Image Pull Secrets können Zugangsdaten zu einer Registry als Secret im Cluster abgelegt werden. Dieses Secret kann dann in einem Pod unter der Liste spec.imagePullSecrets referenziert werden.

Ein Secret kann von einer Json-Config generiert werden:

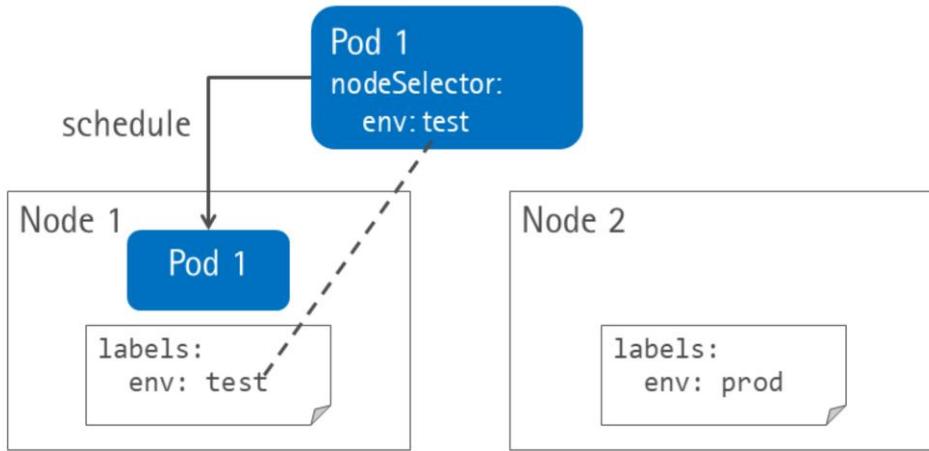
```
kubectl create secret generic my-ipull-secret \
--from-file=<path/to/.docker/config.json> \
--type=kubernetes.io/dockerconfigjson.
```

Oder über Angabe der Credentials:

```
kubectl create secret docker-registry regcred \
--docker-server=<registry> \
--docker-username=<name> --docker-password=<pword> \
--docker-email=<email>'
```



## Pods auf Nodes zuweisen



### NodeSelector

Über einen Node Selector im Pod ist es möglich, die Cluster-Knoten, auf denen dieser Pod laufen soll, einzuschränken. In dem Selektor können Labels angegeben werden, welche dann mit den Labels der Nodes übereinstimmen müssen. Hierüber ist es möglich z. B. Informationen über die Hardware/Leistung der Knoten in den Labels zu beschreiben, welche dann als Kriterien für das Scheduling der Pods genutzt werden können.

### Node affinity

Über Nodeaffinities ist es möglich, noch umfangreichere Regeln für die Pods anzugeben, es kann z. B. auch ausgedrückt werden, ob eine Regel wirklich zwingend erforderlich erfüllt sein muss oder nur eine Preferenz ausdrückt. Es können hierbei auch mehr als nur genaue Übereinstimmungen der Labels für die Selektion genutzt werden.



## Resource Requests

Node 1

cpu (1)



Memory (8Gb)



Node 2

cpu (1)



Memory (8Gb)



Pod 1  
cpu: 400m  
memory: 5Gi

Pod 3  
cpu: 400m  
memory: 2Gi

Pod 2  
cpu: 700m  
memory: 3Gi

Ein Container hat die Möglichkeit, Resource-Requests zu definieren, welche beim Scheduling auf die Nodes berücksichtigt werden, um sicherzustellen, dass der Pod ausreichend Ressourcen zur Verfügung hat. Ist kein Knoten mit ausreichend CPU und Memory vorhanden, so kann der Pod nicht gescheduled werden. Zur Laufzeit wird sichergestellt, dass die Container die von ihnen angefragten Ressourcen auch wirklich bekommen. Sind auf einem Node mehr Ressourcen vorhanden als gegenwärtig angefragt, so können die darauf laufenden Container auch mehr Ressourcen nutzen.

### Memory

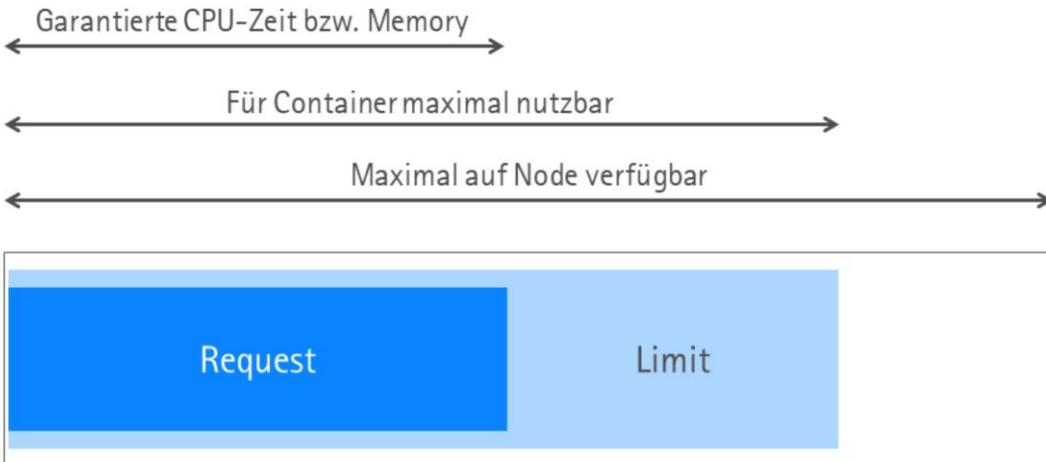
Arbeitsspeicher wird in Bytes angegeben, wobei verschiedene Schreibweisen für Megabyte, Gigabyte usw. bereitstehen.

### CPU

CPU wird in Cores angegeben, dies entspricht entweder einer Cloud-CPU oder ein Hyperthread auf Bare-Metal-Prozessoren mit Hyperthreading. Es sind auch Teile einer CPU in 1/1000 möglich, 500m würde 1/2 CPU entsprechen. In diesem Fall werden nur eine bestimmte Menge Slices der CPU-Zeit für die Anwendung angefragt.



## Resource Limits



Über Limits ist es möglich festzulegen, dass Container zwar, falls verfügbar, mehr Ressourcen nutzen dürfen, aber nur bis zu einem bestimmten Limit. Damit kann verhindert werden, dass Anwendungen zu viel Ressourcen auf einem Node für sich beanspruchen.

Wird das Limit für Memory von der Anwendung im Container überschritten, so wird dies zu einem Abstürzen des Containers führen. Im Falle der CPU bedeutet es, dass die maximal nutzbare CPU-Zeit pro Sekunde beschränkt wird, sodass der Prozess entsprechend gedrosselt wird.

Wird nur ein Limit angegeben und kein Request, so wird automatisch ein Request von der Höhe des Limits angenommen. Wird nur ein Request und kein Limit angegeben, dann kann der Container beliebig viel der noch freien Ressourcen nutzen.



(Projekt `ctr-demo-rest`):

In `src/main/k8s/demo_16` sind in der Grundkonfiguration (im Unterverzeichnis `bases`) Requests und Limits für `app` und `db` eingetragen. In den Overlays für `dev` und `test` werden exemplarisch andere Werte genutzt.

Macht man die Memory-Limits zu klein (z. B. `128Mi` für `app`), werden die Container aufgrund von OOM (Out-of-Memory) abgebrochen.

Bei kleinen CPU-Werten bemerkt man deutlich eine entsprechende Verlangsamung der Anwendungen.



## Ausblick: CustomResourceDefinitions

- Erweiterungen der Kubernetes API

```
apiVersion: gedoplan.de/v1
kind: WebSite
metadata:
  name: my-new-website
spec:
  gitUrl: http://github...
```

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: website.gedoplan.de
spec:
  group: gedoplan.de
  versions:
    - name: v1
  scope: Namespaced
  names:
    plural: websites
    singular: website
    kind: WebSite
    shortNames:
      - ws
```

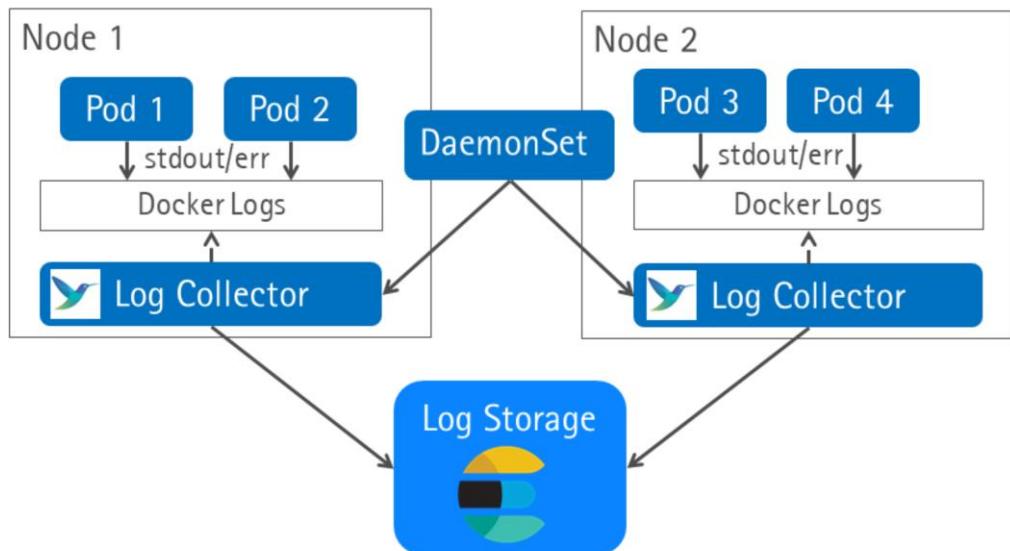
Über CustomResourceDefinitions ist es möglich, die Kubernetes-API zu erweitern und eigene Objekte zu definieren, welche dann ebenfalls im etcd abgelegt werden. Diese Objekte können über die gängigen Werkzeuge wie kubectl verwaltet werden.

Ein Controller kann dann diese Objekte überwachen z. B. mit kubectl --watch ... und auf Änderungen reagieren.

Somit wäre z. B. das auf der Folie angedeutete Beispiel möglich, wo im Cluster eine einfache Möglichkeit bereitgestellt werden soll, um Webseiten aus einem Git-Repo zu deployen, ohne dafür Deployments, Services usw. definieren zu müssen.



## Ausblick: Logging (EFK)



In einer Kubernetes Umgebung, wo viele Anwendungen verteilt auf einer Reihe von Nodes laufen, wird das Thema der Logverwaltung entsprechend wichtig. Damit Logs nicht verlorengehen, wenn die Container beendet sind, und damit Logs einer Anwendung, die mit mehreren Instanzen läuft, gemeinsam durchsucht werden können, sollten die Logs eingesammelt und an zentraler Stelle abgelegt werden.

### Log Collection

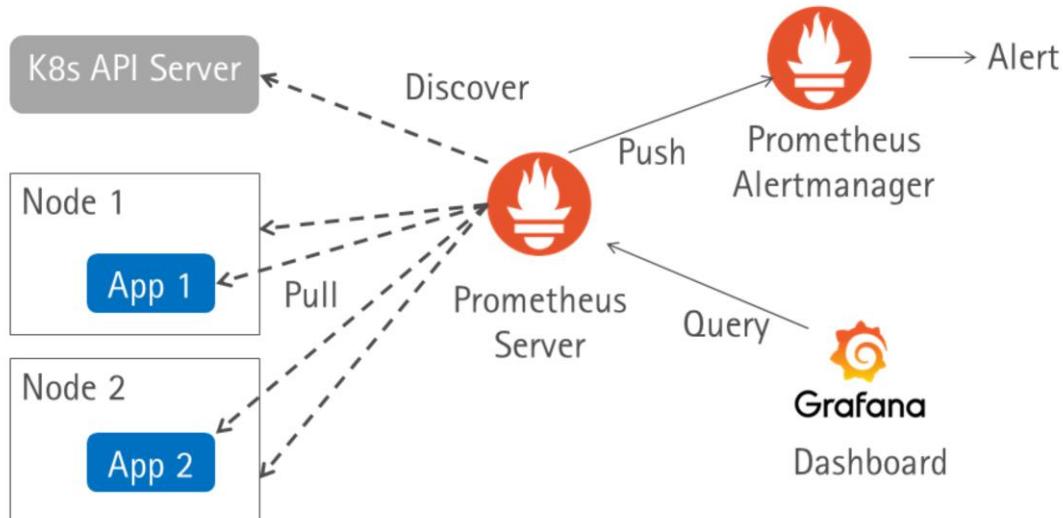
In der Regel wird im Docker/Kubernetes Umfeld so gearbeitet, dass die Anwendungen in den Containern auf stdout/-err rauschreiben, dies wird dann von einem der LogProvider auf dem Dockerhost abgelegt z. B. in Form von Json-Log-Files. Um die Logs zu sammeln, kann nun auf jedem Knoten im Cluster ein Dienst laufen, welcher diese Logs einliest und weiterleitet. Um einen Dienst auf jedem Node zu starten, kann ein DaemonSet verwendet werden. Als Log-Collector können dann z. B. Fluentd, FluentBit oder Filebeat eingesetzt werden.

### Log Storage

Logs müssen irgendwo persistent abgelegt werden mit der Möglichkeit, diese komfortabel zu durchsuchen. Die gängigste Open-Source-Software, die hierfür zum Einsatz kommt, ist Elasticsearch. Für Elasticsearch gibt es eine Webconsole Kibana, mit welcher es möglich ist, komfortabel die Logeinträge anzuzeigen bzw. diese zu durchsuchen.



## Ausblick: Monitoring mit Prometheus



Ein häufig eingesetztes Werkzeug für das Monitoring im Kubernetes-Umfeld ist Prometheus. Prometheus ist ein Open-Source-Werkzeug, welches Metriken in Echtzeit einsammelt und in einer Time-Series-Database ablegt. Über eine Query-Language können diese Daten dann komfortabel abgefragt werden.

### Sammeln der Metrics

Prometheus holt sich die Daten von den Anwendungen z. B. über Rest-Schnittstellen, welche innerhalb des Clusters automatisch mit Hilfe des API-Servers aufgefunden werden können. Ebenso wie von den Anwendungen können auch Daten der Nodes selber abgeholt werden, indem z. B. ein NodeExporter als DaemonSet betrieben wird.

### Alertmanager

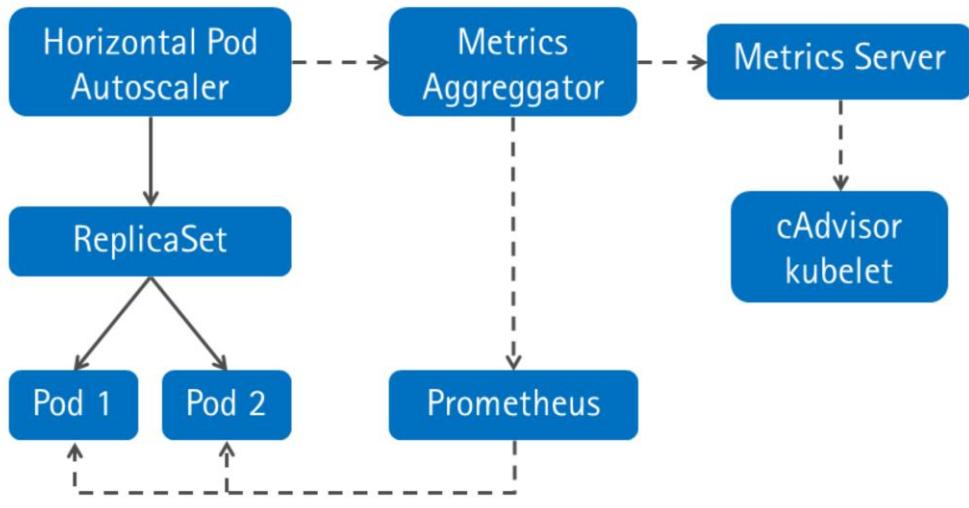
Prometheus bietet auch einen Alertmanager, welcher es ermöglicht, Regeln zu definieren, welche dann zu entsprechenden Benachrichtigungen an Fremdsysteme führen.

### Grafana

Als Software für Dashboards wird in der Regel Grafana eingesetzt, hier gibt es die Möglichkeit, über eine Weboberfläche bequem Dashboards anzulegen oder zu verwalten, welche ihre Daten aus verschiedenen Datenquellen (darunter auch Prometheus) beziehen können.



## Ausblick: Autoscaling



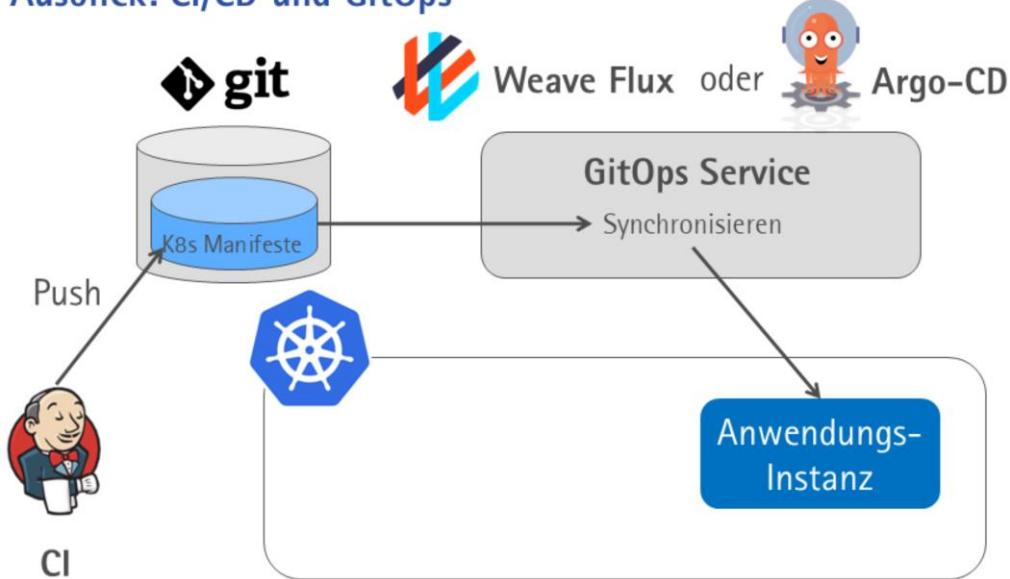
Mit Hilfe des Horizontal-Pod-Autoscalers ist es möglich, die Anzahl Replikas eines ReplicaSets automatisch anhand von Metriken anzupassen.

Zum einen muss hierfür das Metrics-Server Addon im Cluster hinzugefügt werden, hierüber wird der Autoscaler bereits mit grundlegenden Metriken versorgt. Sollen zusätzliche eigene Anwendungsmetriken mit berücksichtigt werden, so besteht die Möglichkeit, Prometheus über ein entsprechendes Plugin zu integrieren.

Ein Autoscaler wird als normales Kubernetes-Objekt über den API Server eingespielt und definiert die Regeln für das Skalieren bezogen auf ein Deployment/ReplicaSet.



## Ausblick: CI/CD und GitOps



Ein Kubernetes Cluster eignet sich als Plattform für Continuous-Integration/Delivery.

### Jenkins

Jenkins ist ein Werkzeug für CI/CD, welches im Java-Umfeld häufig eingesetzt wird. Jobs können hier mit deklarativen Dateien beschrieben und die Buildschritte innerhalb von Containern im Cluster ausgeführt werden. Über die Kubernetes-Manifeste in Form von YAML-Dateien können die Anwendungen dann einfach automatisch ausgerollt werden.

### GitOps

Ein Ansatz, bei dem die Anwendungen nicht direkt über ein CLI-Werkzeug wie kubectl oder helm eingespielt werden, sondern in ein Git-Repo gepusht und dann anschließend von einem Controller, der im Cluster läuft, automatisch synchronisiert werden. Vorteil dieser Vorgehensweise ist, dass der Zustand der deployten Anwendungen immer im Git dokumentiert ist und ein Rollback auf einen alten Stand darüber immer einfach möglich ist. Darüber hinaus entfällt auch das Verteilen von Zugangsdaten für den Kubernetes API-Server, da hier nur der Controller diese Berechtigungen braucht, andere Anwendungen oder User können durch ein Einchecken in ein Git-Repo deployen.