

Verteilte Systeme und Komponenten

Containervirtualisierung

Docker Container in der (Java-)Entwicklung

Roland Gisler



Inhalt

- Grundlagen – Was ist Docker?
- Installation von Docker
- Einsatzszenarien
- Nutzung bestehender Images
- Erstellen eigener Images (Basis)
- Docker im Java-Umfeld
- Zusammenfassung

Lernziele

- Sie kennen die fundamentalen Konzepte von Docker.
- Sie können Docker Container auf Basis bestehender Images starten und nutzen.
- Sie können eigene, einfache Images erstellen und Deployen.
- Sie kennen das Potential, das Docker für Java-Anwendungen bietet.
- Sie können Java-Anwendungen in Docker-Images deployen.

Grundlagen – Was ist Docker?

Docker – Was ist das?

- Sehr stark vereinfacht erklärt: Leichtgewichtige, virtuelle Maschinen, die zwar vollständig isoliert, aber trotzdem direkt auf dem Host ausgeführt werden → schlanker und schneller.
- Motivation und Idee: Lauffähige Applikationen inklusive ihrer Laufzeitumgebung in einer standardisierten Form verteilen können, ohne komplizierte Installation. Persistente Nutzdaten in getrennten, virtuellen Dateisystemen auslagern. Einfache Updates.
- Zentrale, wichtige Konzepte:
 - Container, Images, Volumes, Netzwerke, Registry, Repository
- Docker ist klar in der Linux-Welt zuhause (2013 dort entstanden), kann aber auch auf Windows und Mac genutzt werden.
- **Wichtig:** Unix-Kenntnisse sind für Docker unabdingbar!

Funktionsweise von Docker

- Verwendet Techniken aus dem Linux-Kernel (z.B. «cgroups» und «namespaces») um Prozesse und Speicher vollständig zu trennen.
 - Prozesse laufen «isoliert» auf dem selben Betriebssystemkern (Scheduling), nutzen das selbe Memory (aber isoliert), können auf ein (virtuelles) Dateisystem zugreifen und auch auf Netzwerkdienste.
- Die verschiedenen Basis-Techniken wurden später von Docker in einer Schnittstelle «libcontainer» zusammengefasst.
- Docker kann selber problemlos in einer virtuellen Maschine (z.B. VirtualBox, VMWare etc.) ausgeführt werden. Und seit einiger Zeit auch auf dem Hyper-V (Microsoft, Basis für WSL) auf Windows.
 - siehe Hinweis zu ➔ Installation.

Wichtige Begriffe und Konzepte von Docker

- **Container:** Ist eine laufende Instanz einer Anwendung in Docker. Ein Container basiert immer auf einem →Image.
- **Image:** Enthält die im Docker-Container ausführbare Anwendung in Form eines virtuellen, geschichteten Dateisystems. Images sind read-only und werden in →Repositories auf →Registries abgelegt.
- **Tags:** Versionslabels für Images (nicht analog zu VCS!)
- **Volume:** Virtuelles Dateisystem, das für die persistente (Nutzdaten-)Speicherung in Containern genutzt werden kann.
- **Registry:** Ein Zugangspunkt (Server) für eine Anzahl von →Repositories für die Verteilung von Docker Images.
- **Repository:** Identifikation einer Gruppe von →Images in verschiedenen Versionen (Tags) in einer →Registry.

Erste Demo

Docker Desktop

Containers

Images

Volumes

Dev Environments BETA

Extensions BETA

Disk usage

Logs Explorer

Portainer

Snyk

Add Extensions

RAM 5.17GB CPU 1.47% Connected

Containers Give Feedback

A container packages up code and its dependencies so that apps can run anywhere. [Learn more](#)

Showing 1 items

NAME

IMAGE

STATUS

PORT(S)

STARTED

ACTIONS

☐

clever_franklin

df2db0d4e96a

rgisler/fiboperfl

Running

0 seconds ago

rog@SF-STUDIO: ~

ctop - 08:32:10 CEST 7 containers

NAME	CID	CPU	MEM	NET RX/TX	IO R/W	PIDS
cool_haibt	f0f3b8109ea6	97%	657M / 24.92G	586B / 0B	0B / 0B	27
determined_haibt	4d3acaa032ce	98%	632M / 24.92G	1K / 0B	0B / 0B	27
eager_swirles	d8c961bc2965	99%	754M / 24.92G	586B / 0B	0B / 0B	27
ecstatic_gates	b105a214ade6	98%	723M / 24.92G	586B / 0B	0B / 0B	27
loving_burnell	b3991f405711	99%	575M / 24.92G	516B / 0B	0B / 0B	27
mystifying_boyd	a7f8ff69e79e	98%	556M / 24.92G	586B / 0B	0B / 0B	27
sharp_sammet	fd317da35bf2	98%	674M / 24.92G	516B / 0B	0B / 0B	27

Windows PowerShell

2022-10-19 06:33:05,954 INFO - Fibonacci f(20) = 6765 in 108ns rekursiv mit Array-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,349 INFO - Fibonacci f(20) = 6765 in 493ns rekursiv mit Map-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,350 INFO - fibo(20) = 6765 in 0.021811ms rekursiv berechnet mit 10945 Additionen.
2022-10-19 06:33:06,350 INFO - === fibo(20) - 2. Messung (warm) ===
2022-10-19 06:33:06,356 INFO - Fibonacci f(20) = 6765 in 7ns iterativ berechnet mit 19 Additionen.
2022-10-19 06:33:06,389 INFO - Fibonacci f(20) = 6765 in 40ns nach Binet berechnet.
2022-10-19 06:33:06,473 INFO - Fibonacci f(20) = 6765 in 104ns rekursiv mit Array-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,863 INFO - Fibonacci f(20) = 6765 in 487ns rekursiv mit Map-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,864 INFO - Fibonacci f(20) = 6765 in 0.030539ms rekursiv berechnet mit 10945 Additionen.
2022-10-19 06:33:06,864 INFO - === fibo(40) - 1. Messung (warm) ===
2022-10-19 06:33:06,871 INFO - Fibonacci f(40) = 102334155 in 8ns iterativ berechnet mit 39 Additionen.
2022-10-19 06:33:06,903 INFO - Fibonacci f(40) = 102334155 in 39ns nach Binet berechnet.
2022-10-19 06:33:07,043 INFO - Fibonacci f(40) = 102334155 in 174ns rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:07,871 INFO - Fibonacci f(40) = 102334155 in 1034ns rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:08,203 INFO - Fibonacci f(40) = 102334155 in 332.518ms rekursiv berechnet mit 165580140 Additionen.
2022-10-19 06:33:08,204 INFO - === fibo(40) - 2. Messung (warm) ===
2022-10-19 06:33:08,210 INFO - Fibonacci f(40) = 102334155 in 7ns iterativ berechnet mit 39 Additionen.
2022-10-19 06:33:08,240 INFO - Fibonacci f(40) = 102334155 in 37ns nach Binet berechnet.
2022-10-19 06:33:08,375 INFO - Fibonacci f(40) = 102334155 in 168ns rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:09,139 INFO - Fibonacci f(40) = 102334155 in 954ns rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:09,477 INFO - Fibonacci f(40) = 102334155 in 338.3276ms rekursiv berechnet mit 165580140 Additionen.
2022-10-19 06:33:09,478 INFO - === fibo(80) - 1. Messung (warm) ===
2022-10-19 06:33:09,478 INFO - Fibonacci f(80) = 23416728348467685 in 17ns iterativ berechnet mit 79 Additionen.
2022-10-19 06:33:09,492 INFO - Fibonacci f(80) = 23416728348467744 in 38ns nach Binet berechnet.
2022-10-19 06:33:09,523 INFO - Fibonacci f(80) = 23416728348467685 in 324ns rekursiv mit Array-Cache berechnet mit 79 Additionen.
2022-10-19 06:33:09,782 INFO - Fibonacci f(80) = 23416728348467685 in 2085ns rekursiv mit Map-Cache berechnet mit 79 Additionen.
2022-10-19 06:33:11,451 INFO - NOPE - Rekursiver Fibonacci ist fuer n=80 viel zu langsam.
2022-10-19 06:33:11,451 INFO - === fibo(80) - 2. Messung (warm) ===
2022-10-19 06:33:11,473 INFO - Fibonacci f(80) = 23416728348467685 in 27ns iterativ berechnet mit 79 Additionen.
2022-10-19 06:33:11,504 INFO - Fibonacci f(80) = 23416728348467744 in 38ns nach Binet berechnet.

Installation von Docker

Wichtig: Account auf <https://hub.docker.com/>

- **WICHTIG:** Auch wenn Sie im Schulnetz nur wenig mit Docker arbeiten wollen, sind wir **ALLE** darauf angewiesen, dass Sie noch vor den ersten Versuchen bitte einen **kostenfreien Account (Personal Plan \$0)** auf <https://hub.docker.com/> **eröffnen**.
 - Danach können Sie u.a. auch eigene Registries nutzen.
- Grund: Ohne Login greifen Sie «anonym» auf die öffentlich Docker-Registries zu, und Docker **beschränkt** die Anzahl Pulls auf das Netzwerk der Organisation in der Sie arbeiten → HSLU-I.
 - Darum immer mit eigenem Account arbeiten!
- Login (**nach** Docker-Installation) in der Shell nicht vergessen:

```
docker login [repository-url]
```

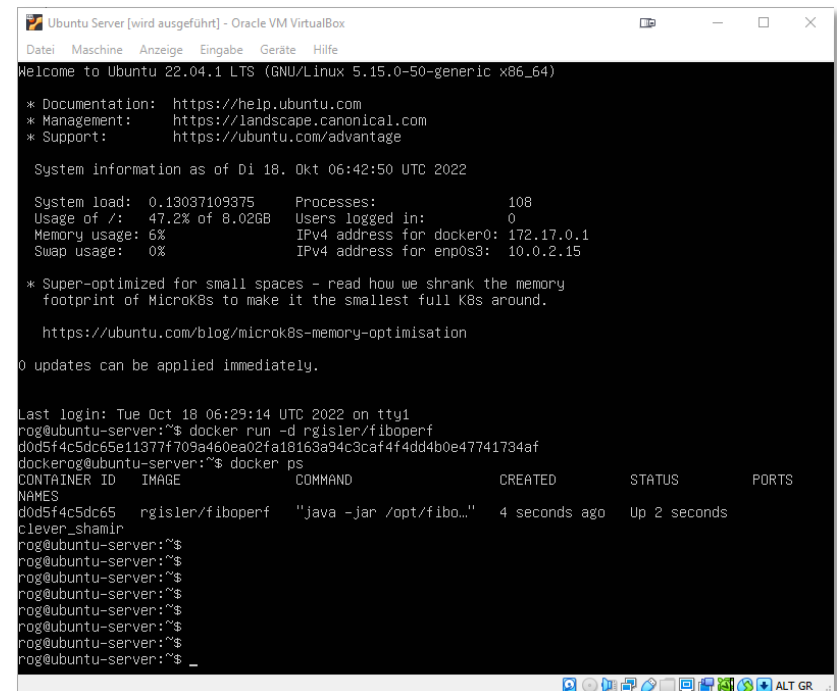
Hinweis: Die Credentials werden in ihrem \$HOME gespeichert.

Docker auf Linux (oder Mac)

- Docker ist auf Linux zu Hause. Just do it! 😊
- Installation gemäss Anleitung von «Docker Desktop»:
 - Linux: <https://docs.docker.com/desktop/install/linux-install/>
 - Mac: <https://www.docker.com/>
- Viel Spass!

Docker auf Windows – Virtuelle Maschine (alt)

- Das ist quasi die «alte» Form der Installation. Wenn Sie nur wenig Experimente in einem völlig isolierten Umfeld machen wollen.
- Nutzen Sie dazu eine Software wie VMWare oder VirtualBox.
- Installieren Sie darin z.B. einen Ubuntu Server 22.04 und ergänzen Sie über `sudo apt install docker.io` die Docker-Installation.
- Das ist vergleichbar mit einem entfernten Docker-Server, wie er z.B. produktiv in einer Cloud betrieben würden.
 - Nachteil: Eingeschränkte Möglichkeiten in unserer Rolle als Entwickler*in. (Integration)



```
Ubuntu Server [wird ausgeführt] - Oracle VM VirtualBox
Datei Maschine Anzeige Eingabe Geräte Hilfe
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-50-generic x86_64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage

System information as of Di 18. Okt 06:42:50 UTC 2022

System load:  0.13037109375   Processes:            108
Usage of /:   47.2% of 8.02GB   Users logged in:      0
Memory usage: 6%              IPv4 address for docker0: 172.17.0.1
Swap usage:   0%              IPv4 address for enp0s3:  10.0.2.15

* Super-optimized for small spaces - read how we shrank the memory
  footprint of MicroK8s to make it the smallest full K8s around.

https://ubuntu.com/blog/microk8s-memory-optimisation

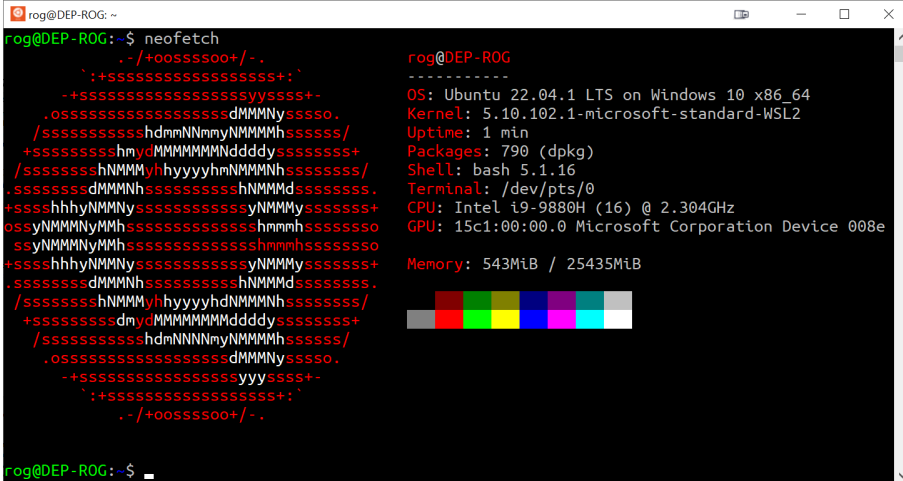
0 updates can be applied immediately.

Last login: Tue Oct 18 06:29:14 UTC 2022 on tty1
rog@ubuntu-server:~$ docker run -d rgisler/fiboperf
d0d5f4c5dc65e11377f709a460ea02fa18163a94c3caf4f4dd4b0e47741734af
rog@ubuntu-server:~$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED    STATUS    PORTS
NAME          rgisler/fiboperf  "java -jar /opt/fibo..."  4 seconds ago    Up 2 seconds
clever_shamir

rog@ubuntu-server:~$
rog@ubuntu-server:~$
rog@ubuntu-server:~$
rog@ubuntu-server:~$
rog@ubuntu-server:~$
rog@ubuntu-server:~$
rog@ubuntu-server:~$
rog@ubuntu-server:~$
```

Docker auf Windows – auf/mit WSL (aktuell)

- Es lohnt sich, zuerst ein virtuelles (Ubuntu-)Linux auf Windows zu haben, so dass das **WSL** (**W**indows **S**ubsystem **L**inux) bereits installiert ist (und in der notwendigen Version **2**) vorliegt.
- Windows Store: «**Ubuntu 22.04**» installieren. Dabei werden Sie von Microsoft wenn nötig zu weiteren Schritten angeleitet.
- Ubuntu steht danach (über Hyper-V virtualisiert) in einer Shell zur Verfügung.
- **Wichtig:** Unter `/mnt` sind alle lokalen Laufwerke gemounted!
- So können Sie sich ganz nebenbei auch ein paar grundlegende Linux-Kenntnisse aneignen, die für Docker unabdingbar sind.

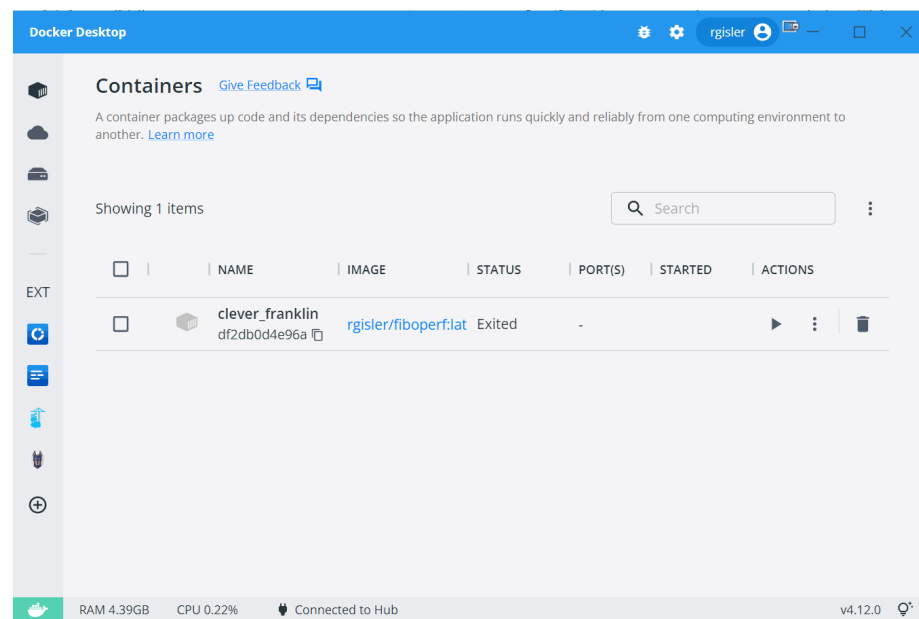


```
rog@DEP-ROG: ~  
rog@DEP-ROG:~$ neofetch  
      .-/+oosssso+/-.  
      `:+ssssssssssss++`  
      .-ssssssssssssssyyss+-.  
      /sssssssssssssssdMMMNysssso.  
      /ssssssssshdmmNNmmyNMMMNhssssso/  
      +ssssssssshmydMMMNMMNdDdDySSssss+  
      /ssssssshNMMMyhhyyyhNMMMNh+ssssso/  
      .sssssssdMMMNhssssssshNMMMdssssss+  
      +ssssshhhyNMMNysssssssssyNMMMyssssss+  
      ossyNMMMNyMMhssssssssshmmhssssssso  
      ssyNMMMNyMMhssssssssshmmhssssssso  
      +ssssshhhyNMMNysssssssssyNMMMyssssss+  
      .sssssssdMMMNhssssssshNMMMdssssss+  
      /ssssssshNMMMyhhyyyhNMMMNhssssss+  
      +sssssssdmydMMMNMMNdDdDySSssss+  
      /ssssssshdmmNNmmyNMMMNhssssso/  
      .ossssssssssssssdMMMNysssso.  
      -+ssssssssssssssyyss+-.  
      `:+ssssssssssss++`  
      .-/+oosssso+/-.  
      rog@DEP-ROG
```

OS: Ubuntu 22.04.1 LTS on Windows 10 x86_64
Kernel: 5.10.102.1-microsoft-standard-WSL2
Uptime: 1 min
Packages: 790 (dpkg)
Shell: bash 5.1.16
Terminal: /dev/pts/0
CPU: Intel i9-9880H (16) @ 2.304GHz
GPU: 15c1:00:00.0 Microsoft Corporation Device 008e
Memory: 543MiB / 25435MiB

Docker auf Windows – Installation

- Erst jetzt installieren Sie **Docker Desktop** von <https://www.docker.com/> und aktivieren dabei die Option, dass Docker auf Basis von **WSL2** (nutzt Hyper-V Virtualisierung) arbeiten soll. Vorteil: Docker läuft dann im Linux Subsystem und steht auch dort zur Verfügung. So haben Sie eine vernünftige Shell zur Hand!
- **Wichtig:** Nach der Installation als erstes ein **docker login** in der Shell ausführen.
 - Speichert Ihre Login-Daten verschlüsselt in Ihrem \$HOME-Verzeichnis.
 - Erlaubt ihnen mehr pulls zu machen, und andere nicht zu blockieren.



Einsatzszenarien von Docker für/in der Entwicklung

Docker – Einsatzszenarien in der Entwicklung

- Spontanes, einfache ausprobieren von (Server-)Anwendungen.
 - Ohne aufwändige Installation, einfach «out-of-the-box».
- Nutzung als Buildumgebung für unterschiedliche, fremde, auf dem Host gar nicht installierte Plattformen.
 - Oder sogar für die komplette Entwicklungsumgebung (!)
- Deployment und Betrieb (eigener) produktiver Anwendungen
 - Komplett konfigurierte, lauffähige Docker-Images.
 - Verteilung und Archivierung der Images in Repositories.
- Schnelle und flexible Testumgebungen.
 - Integration in eigene Testfälle möglich (➔Testcontainer)!
 - Wird im ➔Testing-Input noch vertieft.

Installation und Test

Aus Modul AD: Performancemessung bei Fibonacci

- Im Modul AD haben wir mal vergleichende Performancemessungen für verschiedene Implementationen zur Berechnung von Fibonacci-Zahlen gemacht.
 - Iterativ, Berechnet, Cache mit Array oder Map und Rekursiv.
- Haben Sie Lust, das mal schnell auszuprobieren? Docker installiert?

```
docker run --rm rgisler/fiboperf
```

- Hinweis: `--rm` löscht den Container nach dessen Ende sofort wieder.

18

Docker Images als Build-Umgebung

Docker Images als Buildumgebung

- Haben Sie auf jedem Rechner immer das gewünschte JDK in der richtigen Version und auch das richtige Buildtool verfügbar?
 - Dafür ist Docker extrem praktisch!

- Viele Projekte bieten für Ihre Produkte fertige Images an. Seien Sie aber vorsichtig, achten Sie auf seriöse Quellen!

Auf Docker Hub:



DOCKER OFFICIAL IMAGE

- Beispiel: Apache bietet verschiedene Versionen von Maven, und diese auch auf unterschiedlichen (!) JDK's an.
- Für uns sind folgende Tags interessant:
 - **3.8.6-eclipse-temurin-17**
 - **3.8.6-amazoncorretto-17**
 - **3.8.6-openjdk-18** (kein LTS!)

Supported tags and respective Dockerfile links

```
• 3.8.6-openjdk-18, 3.8-openjdk-18, 3-openjdk-18
• 3.8.6-openjdk-18-slim, 3.8-openjdk-18-slim, 3-openjdk-18-slim
• 3.8.6-eclipse-temurin-11, 3.8-eclipse-temurin-11, 3-eclipse-temurin-11
• 3.8.6-eclipse-temurin-11-alpine, 3.8-eclipse-temurin-11-alpine, 3-eclipse-temurin-11-alpine
• 3.8.6-eclipse-temurin-11-focal, 3.8-eclipse-temurin-11-focal, 3-eclipse-temurin-11-focal
• 3.8.6-eclipse-temurin-17, 3.8.6, 3.8.6-eclipse-temurin, 3.8-eclipse-temurin-17, 3.8, 3.8-eclipse-temurin, 3-eclipse-temurin-17, 3, latest, 3-eclipse-temurin, eclipse-temurin
• 3.8.6-eclipse-temurin-17-alpine, 3.8-eclipse-temurin-17-alpine, 3-eclipse-temurin-17-alpine
• 3.8.6-eclipse-temurin-17-focal, 3.8-eclipse-temurin-17-focal, 3-eclipse-temurin-17-focal
• 3.8.6-eclipse-temurin-19, 3.8-eclipse-temurin-19, 3-eclipse-temurin-19
• 3.8.6-eclipse-temurin-19-alpine, 3.8-eclipse-temurin-19-alpine, 3-eclipse-temurin-19-alpine
• 3.8.6-eclipse-temurin-19-focal, 3.8-eclipse-temurin-19-focal, 3-eclipse-temurin-19-focal
• 3.8.6-eclipse-temurin-8, 3.8-eclipse-temurin-8, 3-eclipse-temurin-8
• 3.8.6-eclipse-temurin-8-alpine, 3.8-eclipse-temurin-8-alpine, 3-eclipse-temurin-8-alpine
• 3.8.6-eclipse-temurin-8-focal, 3.8-eclipse-temurin-8-focal, 3-eclipse-temurin-8-focal
• 3.8.6-ibmjava-8, 3.8.6-ibmjava, 3.8-ibmjava-8, 3.8-ibmjava, 3-ibmjava-8, 3-ibmjava, ibmjava
• 3.8.6-ibm-semeru-11-focal, 3.8-ibm-semeru-11-focal, 3-ibm-semeru-11-focal
• 3.8.6-ibm-semeru-17-focal, 3.8-ibm-semeru-17-focal, 3-ibm-semeru-17-focal
• 3.8.6-amazoncorretto-11, 3.8.6-amazoncorretto, 3.8-amazoncorretto-11, 3.8-amazoncorretto, 3-amazoncorretto-11, 3-amazoncorretto, amazoncorretto
• 3.8.6-amazoncorretto-17, 3.8-amazoncorretto-17, 3-amazoncorretto-17
• 3.8.6-amazoncorretto-18, 3.8-amazoncorretto-18, 3-amazoncorretto-18
• 3.8.6-amazoncorretto-19, 3.8-amazoncorretto-19, 3-amazoncorretto-19
• 3.8.6-amazoncorretto-8, 3.8-amazoncorretto-8, 3-amazoncorretto-8
• 3.8.6-sapmachine-11, 3.8-sapmachine-11, 3-sapmachine-11
• 3.8.6-sapmachine-17, 3.8.6-sapmachine, 3.8-sapmachine-17, 3.8-sapmachine, 3-sapmachine-17, 3-sapmachine, sapmachine
```

Beispiel: JDK 17 «Eclipse Temurin» mit Maven 3.8.6

- Starten Sie doch einfach mal das folgende Image:

```
docker pull maven:3.8.6-eclipse-temurin-17
```

```
docker run -it --rm maven:3.8.6-eclipse-temurin-17 bash
```

- Prüfen Sie mit `mvn -version` was wirklich drauf ist! 😊
- Wir könnten nun mit **git** ein Projekt klonen und bauen. Aber es geht noch einfacher: Wir können auch Teile unseres Host-Dateisystems und ein ➔**Volume** in den Container mounten!
- Somit kann der Container transparent auf Bereiche unseres Host-Dateisystems zugreifen, und das Maven-Repository zur Wiederverwendung persistieren.
 - Das Volume ist für das Maven-Repository, und kann so in verschiedenen Containern wiederverwendet werden!
(als Beispiel, nur ein möglicher Ansatz)

Build auf JDK 17 «Eclipse Temurin» mit Maven 3.8.6

- Hinweis: Pfade an Ihre eigene Umgebung anpassen!

- Beispiel für ein Volume (Erstellung, einmalig):

```
docker volume create "maven-repo"
```

- Container mit gemountetem Verzeichnis und Volume starten:

```
docker run -it --rm  
  -v "C:\path\project\:/work"  
  -v "maven-repo:/root/.m2"  
maven:3.8.6-amazoncorretto-17 bash
```

- Im Container einfach ins Verzeichnis `/work` wechseln (`cd work`) und den Maven-Build starten: `mvn package`
 - Hinweis: Beim ersten Mal ist das Repo natürlich noch leer.
- **Achtung:** Nicht vergessen, den Container mit `exit` zu verlassen.

Docker Image als Grundlage für Buildprozess

- Ist im Arbeitsalltag eine enorme Erleichterung!
- Wir müssen nicht (mehr) verschiedene Java-Laufzeitumgebungen installieren (und umschalten). Deutlich weniger Installationen.
- Wir können relativ einfach auf/mit/für die effektive Zielumgebung bauen und natürlich auch **testen**! Isolation!
- Mit ausgewählten IDE's (IntelliJ und Visual Studio Code) ist es sogar möglich, sich mit einem entsprechenden (laufenden) Container zu verbinden, und direkt darin zu entwickeln!
 - Für diese Fälle lohnt es sich, nach eigenen Bedürfnissen erstellte und konfigurierte Images als Basis zu verwenden!
- ➔ Potential: Standardisierte Entwicklungs(-basis)-Umgebungen für alle Entwickler*innen in einem Team bzw. Organisation!

Erstellen eigener Docker-Images

Wie werden Images erstellt?

- Images werden in → Layers erstellt, wobei jeder Layer eine Anzahl Dateien ergänzt oder Befehle ausführt (die Dateien verändern). Images basieren typisch auf bereits existierenden Images!
- Die einzelnen Schritte werden über Befehle in einem → **Dockerfile** beschrieben (Text-Datei mit dem Namen **Dockerfile**), das ist quasi der «Quellcode» für die Image-Erstellung.
 - Images werden somit reproduzierbar erstellt, vergleichbar mit einem Buildprozess – **infrastructure as code (IaC)**.
- Images werden typisch in einer Registry abgelegt, wo sie zur Verwendung zur Verfügung gestellt werden (Deployment).
 - Default: Docker Hub (<https://hub.docker.com/search?q=>)
- Die Dockerfiles (und alle weiteren Quellartefakte) stellt man typisch unter Versionskontrolle (VCS).

Beispiel: Ein (sehr) einfaches Dockerfile

- Das folgende Beispiel steht Ihnen im bekannten Projekttemplate «oop_maven_template» (seit Version 3.2.0) zur Verfügung!
- Inhalt des **Dockerfile**:

```
FROM amazoncorretto:17.0.4-alpine
COPY target/oop_maven_template.jar /opt/demo/oop_maven_template.jar
CMD ["java", "-jar", "/opt/demo/oop_maven_template.jar"]
```

- Erklärung der einzelnen Zeilen:
 - **FROM**: Das Image basiert auf einem Alpine Linux (klein) mit einer JRE-Distribution «Amazon Corretto», LTS-Version 17.0.4.
 - **COPY**: Aus dem **target**-Verzeichnis des Projektes wird die direkt ausführbare JAR-Datei (→ Shade-JAR mit Manifest, hier eine wichtige Voraussetzung!) in das Image kopiert.
 - **CMD**: Der Startbefehl des Images (für **run**) wird hinterlegt.

Beispiel: Bau und Push eines Images

- Befehl zum Bauen (mit docker):

```
docker build . -t "rgisler/oop-demo:latest"
```

- Das Image steht danach im lokalen Repository zur Verfügung:
 - **rgisler** – Namespace des Images.
 - **oop-demo** – Name des Images.
 - **latest** – Neuste Version (vgl. SNAPSHOT in Apache Maven)
- Vorausgesetzt Sie haben ein Docker-Login (sollten Sie!) kann das Image danach direkt in das Docker-Repository gepushed werden:

```
docker push "rgisler/oop-demo:latest"
```

- Wird dann sichtbar auf <https://hub.docker.com/repositories>



Bau von Images - Herausforderungen

- Die Images müssen natürlich der jeweiligen Plattform und Distribution entsprechend konfiguriert werden. Das setzt relativ gute Kenntnisse des Zielsystems voraus (oder viel try&error ☹).
- Man kann viel gutes (aber auch schlechtes) von bestehenden Images (bzw. deren Dockerfiles) abgucken.
- Die Referenz für die Dockerfile-Syntax:
<https://docs.docker.com/engine/reference/builder/>
- Es gibt mittlerweile gute Syntax-Checker/Editoren für Dockerfiles.
- Man sollte darauf achten, «sinnvolle» Layers zu bilden.
 - So wenige wie möglich, so viele wie nötig.
 - Möglichst ähnliche Änderungshäufigkeit (pro Layer).
 - Möglichst kompakte Images (möglichst klein).

Komplexeres Beispiel: Dockerfile von maven

- Siehe <https://github.com/carlossg/docker-maven/blob/ac292f26884bf2be9fe69f6e397da3b124c1e35c/eclipse-temurin-17/Dockerfile>
- Interessante Details:
 - **apt update** und **install** in einem Kommando, inkl. wieder löschen der Update-Listen. Alles in einem Layer, Platz gespart.
 - Installation von Maven in einem einzigen Kommando, inkl. Download mit sha-Check, dann aufräumen und Symlink setzen. Auch hier: Nur ein Layer erstellt, Platz gespart.
- Beim Build eines Containers werden die Dateiinhalte und Kommandos «gehashed». Ein Layer wird nur neu erstellt, wenn der Hash ändert → schnellerer Build des Images, weniger Bandbreite.
 - Problem: **apt install** – Befehl (Beispiel) unverändert, aber...?

Docker im Java Umfeld

Dockerisierung von Java-Anwendungen

- Java-Applikationen brauchen im Regelfall eine JRE (kostet Platz).
- Java-Applikationen können sehr unterschiedlich deployed werden:
 - Single-JAR – alles in einem einzigen JAR verpackt.
 - App-JAR und Dependency-JAR's: `classpath` notwendig, typisch über ein Startscript (`shell`) gelöst.
 - Alles in einem, oder in getrennten Layern? (Vor- und Nachteile)
 - Modularisierte Applikation (JPMS, **J**ava **P**lattform **M**odul **S**ystem).
- Gewähltes Szenario ist individuell abzuklären, es gibt keine Empfehlung für eine Standard-Variante, sondern nur Meinungen. 😊
- Positiv: Es gibt mehrere Maven-Plugins, welche verschiedene Techniken unterstützen und automatisieren, und sich sogar kombinieren lassen.

Maven Plugins für Docker

▪ **Fabric8 Maven Docker Plugin** (Standard)

- Setzt ein Dockerfile voraus. Alle Möglichkeiten offen.
- Konfiguration teilweise über das Maven POM.
 - Minimal, z.B. für lokalen Start über Maven.
- Projekt und Dokumentation:

<https://github.com/fabric8io/docker-maven-plugin>

▪ **Google Jib Docker Plugin** (Alternative)

- Sehr hohe Automatisierung.
- Konfiguration ausschliesslich über das Maven POM.
- Image-Build und Deploy auch ohne (!) Docker-Runtime möglich.
- Projekt und Dokumentation:

<https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin>

Einfaches Beispiel: Java EchoServer in Docker

- Projekt steht auf ILIAS als ZIP-Datei zur Verfügung.
- **Wichtig:** Vor Versuchen im POM den Benutzer (**docker.user**) anpassen und Plugin-Konfigurationen (Fabric8 und Jib) ansehen.
- **EchoServer** (auf Port **5555/tcp**) - kann gestartet werden:
 - Direkt in der IDE.
 - Binär als Single-JAR packetiert (mvn package) aus **/target**
 - Verpackt in einer Container (als Single-JAR) mit **Fabric8**-Plugin.
 - Verpackt als Layered-Java App mit **Jib**-Plugin.
- **EchoClient** – ist die passende Client-Applikation:
 - Direkt in der IDE ausführen. Default auf **localhost:5555**
 - Eingabe «**quit**» beendet nur den Client, Server läuft weiter.
 - Eingabe «**exit**» sendet Kill-Message, beendet Client & Server.

Bau und Start/Stop des EchoServer Beispiels (Fabric8)

- Das Projekt kann mit Apache Maven über `mvn package` normal gebaut werden. Es wird über das shade-Plugin ein «Singel-JAR» erzeugt, dass sämtliche Dependencies integriert.
- JAR kann mit `java -jar vsk-echoserver.jar` gestartet werden, der Server bindet sich an sämtliche Interfaces auf Port 5555.
 - Empfehlung: Gleich mit Client testen!
- Bauen des Docker Images mit Maven: `mvn docker:build`
- Starten des Images mit Maven: `mvn docker:start`
 - Anzeigen des Logs über Docker Desktop, oder in Shell über `mvn docker:logs` oder mit `docker logs <hash> -f`
- Stoppen des Images mit Maven: `mvn docker:stop`
 - Löscht den Container automatisch auch gleich!

Netzwerk mit Docker

- Für die Netzwerk-Kommunikation mit einem Container, müssen die dafür genutzten Ports auf Ports des Host-Systems gemappt werden.
 - Analog zu den Dateisystemen (mit `-v "host:container"`)
 - Im EchoServer-Beispiel ist das für die Maven-Plugins bereits in der Konfiguration im POM hinterlegt.
- Dafür gibt es ein weiteres Argument: `-p "hostport:cont.port"`
- Wenn wir das EchoServer-Beispiel mit Docker starten wollen:

```
docker run -d -p "5555:5555" rgisler/echoserver
```

 - Fehlerquelle: Vergisst man das Portmapping startet der Container trotzdem fehlerfrei, aber Kommunikation klappt nicht
 - Praktisch: Mapping auf andere Ports möglich (z.B. `4444:5555`)
- Tipp: Das gute alte `telnet` ist hier praktisch. 😊

Bau und des EchoServer Beispiels mit Jib

- Das Images kann wahlweise mit Fabric8 (`docker:help`) oder mit Jib (`jib:help` gibt es leider **nicht**) gebaut werden.
- Jib baut mit dem Kommando `mvn jib:build` das Image nicht nur ohne (!) die lokale Docker-Instanz (muss nicht laufen!), sondern pusht es auch direkt in die Registry hoch.
 - Dabei wird die Java-Applikation bzw. das POM analysiert und die Dependencies (und Konfig) von der eigentlichen Applikation (classes und Konfig) in getrennten Layers im Image abgelegt.
 - Das Reduziert den Pull-Aufwand bei Applikationen mit grossen Dependencies markant (weil diese typisch viel seltener ändern).
- Jib kann aber auch rein lokal bauen (dann aber **mit** Docker):
`mvn jib:dockerBuild`

Ein Ausblick: Testcontainer mit Docker!

Docker für automatisierte Integrationstests

- Docker Container sind dafür ausgelegt, dass man sie sehr schnell hoch- und runterfahren kann. Das wäre doch perfekt für (Integrations-)Tests!

- **YEAH! Das gibt es:**

<https://www.testcontainers.org/>



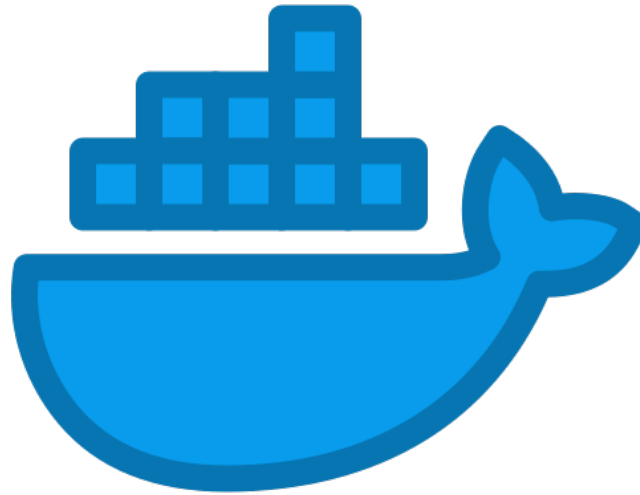
- Eine kleine Demo hab ich noch...😊
- Stichworte: Automatischen hoch-/runterfahren von generischen oder vorbereiteten Containern. Zugriff auf Logs. Dynamische, kollisionsfreie Vergabe von Ports. Automatisches Aufräumen.

Noch ein paar Tool-Tipps zum Schluss

- **cTop** - <https://github.com/bcicen/ctop>
 - Shell-Tool für die permanente Anzeige aller Container.
 - Man kann auch Container inspizieren.
- **dive** - <https://github.com/wagoodman/dive>
 - Shell-Tool für die Analyse von Images.
 - Für die Fehlersuche praktisch.
- Tipps:
 - Lassen Sie **ctop** einfach in einer Shell laufen, dann sehen Sie immer kompakt was auf ihrem System grad so los ist.
 - Untersuchen Sie mit **dive** die von den Fabric8- und Jib-Plugins unterschiedlich aufgebauten Images!
- Es gibt noch viel mehr! Weitere Tipps werden im Forum gerne aufgenommen!

Zusammenfassung

- Docker für leichtgewichtige Virtualisierung auf verschiedenen Plattformen.
- Für einfaches Deployment und Betrieb von Applikationen (typisch headless, Server-Applikationen).
- Aber auch für die Entwicklung(-sumgebung) sehr interessant.
- Sie brauchen definitiv auch Linux/Unix-Kenntnisse!
- Sehr hoher Automatisierungsgrad, Basis für CD (Continuous Deployment).
- Umfangreiches Tooling, grosse Popularität.



Zeit für eigene Versuche!

Fragen?