Verteilte Systeme - Virtualisierung und Container

Eine Einführung in moderne Deployment-Technologien

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw.de

Version: 1.0

Folien: [HTML] https://delors.github.io/ds-containers/folien.de.rst.html
[PDF] https://delors.github.io/ds-containers/folien.de.rst.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Einführung in Virtualisierung und Container

Virtualisierung - allg. Definition

Virtualisierung bezeichnet in der Informatik die ***Nachbildung eines Hard- oder Software-Objekts durch ein ähnliches Objekt vom selben Typ mit Hilfe einer Abstraktionsschicht****.*

Dadurch lassen sich virtuelle (d. h. nicht-physische) Geräte oder Dienste wie emulierte Hardware, Betriebssysteme, Datenspeicher oder Netzwerkressourcen erzeugen.

Dies erlaubt es etwa, Computer-Ressourcen (insbesondere im Server-Bereich) transparent zusammenzufassen oder aufzuteilen, oder ein Betriebssystem innerhalb eines anderen auszuführen. Dadurch können u. a. mehrere Betriebssysteme auf einem physischen Server oder „Host“ ausgeführt werden.

—***Stand 5. Oktober 2025 - Wikipedia***

Motivation - Warum Virtualisierung?

🔎 Beobachtung

Moderne Softwareentwicklung steht vor verschiedenen Herausforderungen:

- **Heterogene Umgebungen**: Software muss auf verschiedenen Betriebssystemen und Hardware-Konfigurationen laufen.
- **Skalierbarkeit**: (Server-)Anwendungen müssen flexibel skalieren können.
- **Isolation**: Verschiedene (Server-)Anwendungen sollen sich nicht gegenseitig beeinträchtigen.
- **Portabilität**: Code soll einfach zwischen verschiedenen Umgebungen übertragbar sein.
- **Ressourceneffizienz**: Hardware soll optimal genutzt werden.

Virtualisierung und Containerisierung bieten Lösungen für diese Herausforderungen.

Was ist Virtualisierung?

Definition

Virtualisierung[1] ermöglicht es mehrere virtuelle Maschinen (VMs) oder Container auf einer physischen Hardware auszuführen.

Ziel der Virtualisierung

- Abstraktion der Hardware
- Isolation zwischen Anwendungen
- Optimale Ressourcennutzung
- Flexibilität bei der Bereitstellung

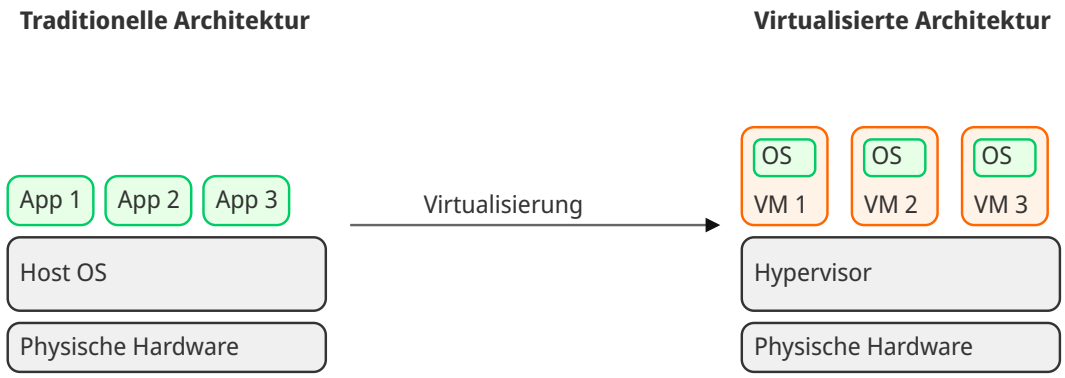
Arten der Virtualisierung

- **Bare-Metal Virtualisierung**: Virtualisierung direkt auf der Hardware
- **Hosted Virtualisierung**: Virtualisierung auf einem Host-Betriebssystem
- **Containervirtualisierung**: Virtualisierung von Systemressourcen eines Betriebssystems, teilt sich jedoch den Kernel mit dem Host-Betriebssytem.
- **Anwendungsvirtualisierung**: Ausführung einer Anwendung in einer exclusiven, teilweise oder vollständig isolierten virtuellen Umgebung.

---

[1] Im Folgenden konzentrieren wir uns auf Virtualisierung von serverseitigen Anwendungen und von Softwareentwicklung.

Architekturunterschiede



Bare-Metal Virtualisierung

Bei der **Bare-Metal Virtualisierung** (auch Type-1 Hypervisor) läuft die Virtualisierungsschicht direkt auf der Hardware, ohne ein Host-Betriebssystem.

Vorteile

✓**Bessere Performance**: Kein Overhead durch Host-OS

✓**Höhere Sicherheit**: Weniger Angriffsfläche

✓**Direkter Hardware-Zugriff**: Optimale Ressourcennutzung

✓**Enterprise-fähig**: Für produktive Umgebungen geeignet

Nachteile

❗ **Komplexität**: Schwerer zu installieren und zu verwalten

❗ **Hardware-Abhängigkeit**: Hardware-Unterstützung erforderlich

❗ **Kosten**: ggf. Lizenzkosten für Hypervisor-Software

Beispiele für Bare-Metal Hypervisoren:

- Xen
- VMware vSphere/ESXi
- Microsoft Hyper-V (Server)
- Citrix Hypervisor
- ...

Hosted Virtualisierung

Bei der **Hosted Virtualisierung** (auch Type-2 Hypervisor) läuft die Virtualisierungsschicht als Anwendung auf einem Host-Betriebssystem.

Vorteile

✓**Einfache Installation**: Wie normale Software

✓**Flexibilität**: Verschiedene Host-Betriebssysteme möglich (ggf. über Hardwarearchitekturgrenzen hinweg)

✓**Entwicklungsumgebung**: Ideal für Tests und Entwicklung

✓**Benutzerfreundlich**: Grafische Oberflächen verfügbar

Nachteile

❗**Performance-Overhead**: Host-OS verbraucht Ressourcen

❗**Sicherheitsrisiken**: Host-OS als zusätzliche Angriffsfläche

❗**Begrenzte Skalierbarkeit**: Weniger VMs pro Host möglich

Beispiele für Hosted Hypervisoren:

- Qemu (Open Source - kann Virtualisierung und Emulation)[2]
- VMware Workstation
- Oracle VirtualBox
- Parallels Desktop
- ...

---

[2] Qemu kann auch zusammen mit KVM betrieben werden.

2. Container-Technologien

Was sind Container?

Definition

Container sind leichtgewichtige, portable Einheiten, die Anwendungen zusammen mit allen notwendigen Abhängigkeiten (Code, Runtime, Bibliotheken) verpacken. Sie teilen sich den Kernel des Host-Betriebssystems, bieten aber isolierte Prozess- und Nutzerbereiche, wodurch Anwendungen unabhängig voneinander laufen können.

Beispiele für die Isolierung (Linux)

- **Namespaces**: Isolation von Nutzern, Prozessen, Netzwerk, Dateisystem
- **Control Groups (cgroups)**: Ressourcenbeschränkung (CPU, Speicher, ...)
- **Union File Systems**: Effiziente Speicherung von Layern

Beispiele für Container-Isolierung unter Linux

Namespaces: Isolieren unterschiedliche Systemressourcen, sodass Prozesse in einem Container „ihre eigene Welt“ sehen: Jeder Container hat seine eigenen Benutzer- und Gruppen und sieht nur seine eigenen Prozesse. Prozesse aus anderen Containern oder dem Host sind unsichtbar. Container haben eigene Netzwerkschnittstellen, IP-Adressen und Ports und können z. B. auf Port 80 lauschen, ohne Host oder andere Container zu stören. Weiterhin hat jeder Container eine eigene Sicht auf das Dateisystem. Ein Container kann sein Root-Dateisystem haben, ohne den Host oder andere Container zu verändern.

Control Groups (cgroups): Regeln und beschränken Ressourcennutzung, um fairen Zugriff zu gewährleisten: Insbesondere CPU-Zeit, Speicherbenutzung und Netzwerk und I/O Limits.

Union File Systems (UnionFS / OverlayFS): Ermöglichen effiziente, schichtweise Speicherung, da zum Beispiel das Basis-Image + Container-spezifische Änderungen als separate Layer gespeichert werden.

Container vs. Virtual Machines

Ressourcennutzung

VM: Vollständiges OS pro Instanz

Container: Geteiltes Host-OS

Vergleich: Container vs. VMs

Virtual Machines

Vorteile:

- ✓Vollständige Isolation
- ✓Verschiedene Betriebssysteme möglich
- ✓Hohe Sicherheit

Nachteile:

- ! Hoher Ressourcenverbrauch
- ! Langsame Startzeiten
- ! Komplexe Verwaltung
- ! Weniger portabel

Container

Vorteile:

- ✓Schneller Start
- ✓Geringer Ressourcenverbrauch
- ✓Hohe Portabilität
- ✓Einfache Skalierung

Nachteile:

- ! Weniger Isolation
- ! Gleiches Betriebssystem erforderlich
- ! Sicherheitsrisiken bei Root-Zugriff
- ! Abhängigkeit vom Host-OS

3. Containerisierung mit Docker

Docker - Container-Infrastruktur

Docker ist eine Open-Source-Plattform zur Entwicklung, Bereitstellung und Verwaltung von containerisierten Anwendungen.

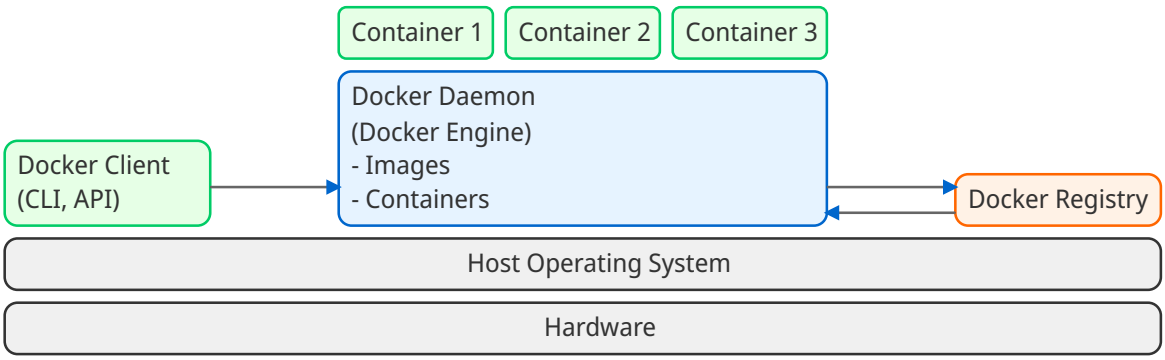
Docker-Komponenten

- **Docker Engine**: Runtime für Container
- **Docker Images**: Vorlagen für Container
- **Docker Containers**: Laufende Instanzen
- **Docker Registry**: Repository für Images
- **Docker Compose**: Orchestrierung mehrerer Container

Docker-Architektur

- **Docker Daemon**: Hintergrundprozess der Images lädt sowie Container verwaltet und steuert
- **Docker Client**: Kommandozeilen-Tool zum Interagieren mit dem Daemon.
- **Docker Registry**: Speicher für Images
- **Docker Hub**: Öffentliche Registry

Docker-Architektur



Docker-Images und Container

Ein **Docker-Image** ist eine unveränderliche Vorlage, die alle notwendigen Komponenten für eine Anwendung enthält.

Image-Eigenschaften

- **Layered Architecture**: Images bestehen aus mehreren Layern
- **Immutable**: Images können nicht verändert werden
- **Portable**: Funktionieren auf verschiedenen Systemen
- **Versioned**: Verschiedene Versionen eines Images möglich

Container-Lifecycle

- **Create**: Container aus Image erstellen
- **Start**: Container starten
- **Run**: Container ausführen
- **Stop**: Container stoppen
- **Remove**: Container löschen

Docker-Befehle

Image-Verwaltung:

- `docker pull <image>` - Image herunterladen
- `docker images` - Alle Images anzeigen
- `docker build -t <name> .` - Image erstellen
- `docker rmi <image>` - Image löschen

Container-Verwaltung:

- `docker run <image>` - Container starten
- `docker ps` - Laufende Container anzeigen
- `docker stop <container>` - Container stoppen
- `docker rm <container>` - Container löschen

Informationen:

- `docker logs <container>` - Container-Logs anzeigen
- `docker exec -it <container> /bin/bash` - In Container einloggen

4. Container Orchestrierung

Was ist Container Orchestrierung?

Container Orchestrierung ist die Automatisierung der Bereitstellung, Verwaltung, Skalierung und Vernetzung von Container-Anwendungen.

Gegenstand der Orchestrierung:

- **Deployment**: Automatische Bereitstellung von Containern
- **Scaling**: Dynamische Skalierung basierend auf Last
- **Load Balancing**: Verteilung der Anfragen
- **Service Discovery**: Auffinden von Services
- **Health Monitoring**: Überwachung der Container-Gesundheit
- **Rolling Updates**: Updates ohne Downtime

Container Orchestrierungs-Tools

Kubernetes

- **Open Source** von Google
- **De-facto Standard** für Container Orchestrierung
- **Umfangreiche Features**
- **Große Community**
- **Komplexe Einrichtung**

Docker Swarm

- **Native Docker-Lösung**
- **Einfache Einrichtung**
- **Weniger Features** als Kubernetes
- **Gut für kleinere Projekte**
- **Einfache Verwaltung**

Apache Mesos

- **Distributed Systems Kernel**
- **Unterstützt verschiedene Frameworks**
- **Hochskalierbar**
- **Komplexe Architektur**
- **Weniger verbreitet**

Exemplarische Verwendung von Docker

Docker-basiertes Deployment einer Web-Anwendung

Entwicklung und Deployment einer einfachen Web-Anwendung mit Docker.

1. **Docker-Installation prüfen**
2. **Einfache Web-Anwendung erstellen**
3. **Dockerfile erstellen**
4. **Docker-Image bauen**
5. **Container starten und testen**
6. [**Docker Compose für Multi-Container-Setup**]

Schritt 1: Docker-Installation prüfen

```
# Docker-Version prüfen
docker --version

# Docker-Status prüfen
docker info

# Ersten Container testen
docker run hello-world
```

Schritt 2: Web-Anwendung erstellen (`index.html`)

```
<!DOCTYPE html>
<html>
<head><title>Docker Demo App</title></head>
<body>
    <h2>Willkommen zur Docker-Demo!</h2>
    <p>Diese Anwendung läuft in einem Docker-Container.</p>
    <p>Aktuelle-Zeit: <span id="time"></span></p>
    <script>
        document.getElementById('time').textContent =
            new Date().toLocaleString();
    </script>
</body>
</html>
```

Schritt 3: `Dockerfile` erstellen

```
# Basis-Image verwenden (Alpine-Linux mit einem vorinstallierten nginx)
FROM nginx:alpine

# Arbeitsverzeichnis setzen innerhalb des Containers
WORKDIR /usr/share/nginx/html

# HTML-Datei kopieren (kopiert lokale Datei in den Container relativ zum WORKDIR)
COPY index.html .

```

```
# Port 80 freigeben
EXPOSE 80

# Nginx im Vordergrund starten (da sich sond der Docker-Container gleich beendet)
CMD ["nginx", "-g", "daemon off;"]
```

Schritt 4: Docker-Image bauen

```
# Image bauen mit dem Tag "webapp-demo"
docker build -t webapp-demo .

# Images anzeigen
docker images

# Image-Details anzeigen
docker inspect webapp-demo
```

Schritt 5: Container starten und testen

```
# Container mit dem Namen "webapp-container" im Hintergrund (-d) starten
docker run -d -p 8080:80 --name webapp-container webapp-demo

# Container-Status prüfen
docker ps

# In Browser testen: http://localhost:8080

# Container-Logs anzeigen
docker logs webapp-container

# Container stoppen
docker stop webapp-container

# Container (nicht Container-Image) löschen
docker rm webapp-container
```

Schritt 6: Docker Compose für Multi-Container (`docker-compose.yml`)

```
version: '3.8'
services:
web:
    build: .
    ports: ["8080:80"]
    depends_on: [db]
    environment: [DB_HOST=db]

db:
    image: postgres:13
    environment: [POSTGRES_DB=demo, POSTGRES_USER=demo, POSTGRES_PASSWORD=demo123]
    volumes: [postgres_data:/var/lib/postgresql/data] # persistent

volumes: {postgres_data:}
```

Schritt 7: Anwendung (bestehend aus mehreren Containern) starten:

```
# Multi-Container-Setup starten
docker-compose up -d

# Status prüfen
docker-compose ps

# Logs anzeigen
docker-compose logs

# Setup stoppen
docker-compose down
```

5. Zusammenfassung

Konzepte

- **Virtualisierung** ermöglicht effiziente Nutzung von Hardware-Ressourcen
- **Container** bieten leichtgewichtige Alternative zu VMs
- **Docker** ist der Standard für Container-Plattformen
- **Orchestrierung** automatisiert das Management von Container-Anwendungen
- **Multi-Container-Setups** ermöglichen komplexe Anwendungsarchitekturen

Best Practices für Docker

Image-Erstellung

- **Non-Root User** für Sicherheit
- **Layer-Caching** optimieren
- **Security Updates** regelmäßig durchführen

Container-Management

- **Health Checks** implementieren
- **Resource Limits** setzen
- **Logging** konfigurieren
- **Backup-Strategien** für Volumes

Sicherheit

- **Minimale Images** verwenden
- **Secrets** nicht im Image speichern
- **Network Policies** definieren
- **Regelmäßige Aktualisierungen** durchführen

Ausblick

Die Container-Technologie entwickelt sich weiter:

- **Serverless Container** (AWS Fargate, Azure Container Instances)
- **WebAssembly (WASM)** als Alternative zu Container
- **Edge Computing** mit Container-Technologie
- **GitOps** für automatisiertes Deployment
- **Service Mesh** für Microservice-Kommunikation

Schlussfolgerung

Container und Virtualisierung sind fundamentale Technologien für moderne Softwareentwicklung und -deployment. Die praktische Anwendung mit Docker bietet eine solide Grundlage für weiterführende Themen wie Kubernetes und Cloud-Native-Entwicklung.

Übung: Erste Schritte

5.1. Webserver im Docker Container

Loggen Sie sich auf dem Server ein und versuchen Sie alle Schritte nachvollzuziehen, die notwendig sind, um einen Docker Image mit Nginx zu bauen und danach zu starten. Orientieren Sie sich an dem Beispiel aus den Vorlesungsfolien, aber nutzen Sie eine eigene `index.html`.

Achtung!

Spezifzieren Sie beim Start des Containers ein Portmapping (z.B. `8100:80`) passend zu den Ihnen zugeteilten Ports!

Stellen Sie sicher, dass Ihr Server läuft in dem Sie die Webseite aufrufen.

Verfolgen Sie das Log, um die Zugriffe auf Ihre Webseite zu sehen.

Übung: Node.js im Container

5.2. Node.js Server im Container laufen lassen

1. Kopieren Sie die Ressourcen (player.html, admin.html, game.js, package.json) auf Ihren Server in ein neu angelegtes Verzeichnis
2. Ihr Docker-Image soll node.js und alle Ressourcen enthalten.

   D. h. Ihr `Dockerfile` muss eine Referenz auf ein entsprechendes Image mit node.js enhalten und die Ressource in das Dockerfile kopieren.

3. Passen Sie das `Dockerfile` so an, dass beim Bauen des Docker-Images die benötigten Bibliotheken mit installiert werden. Sie müssen dafür den Befehl `npm install --production` während des Bauens mit Hilfe von `RUN` im `Dockerfile` ausführen.
4. Passen Sie den Kommandozeilenbefehl (`CMD`) im `Dockerfile` so an, dass der Node Server passend gestartet wird.

   Achtung!

   Die Anwendung läuft Standardmäßig auf Port 8800.