# Containerisierung und Orchestrierung von Microservices

Dennis Flegler —*Matrikelnummer: 5119110*

# Vorwort

Als Werkstudent im Bereich Cloud-Native Java Development bringe ich bereits fundierte Erfahrungen im Umgang mit Docker und containerisierten Anwendungen mit. Die Vorlesung von Lars Hick habe ich nicht nur als willkommene Vertiefung wahrgenommen, sondern auch als äußerst inspirierend und lehrreich empfunden. Besonders seine klaren Erklärungen und praxisnahen Beispiele haben mir den Einstieg in Themen wie Cluster-Architekturen, Netzwerkrichtlinien und Telemetrie-Stacks deutlich erleichtert. Während der praktischen Phase bin ich beim Aufsetzen eines lokalen Kubernetes-Clusters unter Windows auf zahlreiche Herausforderungen gestoßen, von WSL-Inkompatibilitäten über DNS-Probleme bis hin zu Port-Forwarding-Konflikten, die mich an meine Grenzen brachten. Diese Erfahrungen haben mir eindrücklich vor Augen geführt, wie stabil und zuverlässig eine Linux-Umgebung für den Betrieb von Kubernetes-Clustern ist. Für zukünftige Projekte werde ich daher konsequent auf eine native Linux-Plattform setzen, um eine reibungslose Cluster-Inbetriebnahme sicherzustellen. Ich bin sehr dankbar für diese Vorlesung und freue mich darauf, mein Wissen im Bereich Cloud-Native Architekturen weiter zu vertiefen und in der Praxis einzusetzen.

# Inhaltsverzeichnis

# 1 Einführung in Containerisierung

## 1.1 Einführung

Containerisierung bezeichnet die Bereitstellung von Anwendungen in leichtgewichtigen, isolierten Laufzeitumgebungen (Containern), die den Anwendungscode sowie alle notwendigen Abhängigkeiten enthalten. Durch die gemeinsame Nutzung des Host-Kernels und die Kapselung aller Bibliotheken und Konfigurationen ermöglichen Container eine plattformunabhängige Ausführung, konsistente Entwicklungs- und Produktionsumgebungen sowie schnelle horizontale Skalierung bei geringem Overhead.

## 1.2 Historische Entwicklung

Die Wurzeln der Containerisierung reichen zurück in die 1990er Jahre: FreeBSD Jails und Linux Chroot legten erste Grundlagen für Prozess-Isolation. In den 2000er Jahren wurden Linux Namespaces und Cgroups als Kernel-Mechanismen ergänzt. Mit Docker (2013) wurde Containerisierung massentauglich, und die Open Container Initiative (OCI) entstand als Standardisierungsplattform.

## 1.3 Vergleich von Bare-Metal, VMs und Containern

- **Bare-Metal**: Anwendungen laufen direkt auf dem Host-Betriebssystem, maximale Effizienz, jedoch eingeschränkte Isolation.
- **Virtuelle Maschinen**: Vollständige OS-Emulation per Hypervisor, starke Isolation, hoher Ressourcen- und Verwaltungsaufwand.
- **Container**: Teilen sich den Host-Kernel, isolierte Benutzerland-Prozesse, schnelle Startzeiten und geringer Overhead.

## 1.4 Standards und Spezifikationen

Die Open Container Initiative (OCI) definiert drei Kern-Spezifikationen:

1. Runtime Specification
2. Image Specification
3. Distribution Specification

Diese Spezifikationen sorgen für Interoperabilität zwischen verschiedenen Runtime-Implementierungen (Docker, containerd, Podman) **oci2018**.

## 1.5 Cloud Native Computing Foundation

Die Cloud Native Computing Foundation (CNCF) hostet zentrale Projekte des Cloud-Native Ökosystems:

- Kubernetes (grundlegende Einführung)
- Helm (Einführung)
- Prometheus & Grafana (Monitoring)
- OCI (Standardisierung)

Sie fördert Best Practices und Interoperabilität in Cloud-Native Architekturen **cncf2020**.

## 1.6 REST-Architekturen und Microservices

In container-basierten Microservice-Architekturen sind REST-APIs weit verbreitet. Wichtige Prinzipien nach Fielding:

- **Uniform Interface**: Einheitliche, dokumentierte Endpoints.
- **Zustandslosigkeit**: Jede Anfrage enthält alle notwendigen Daten.
- **Cachbarkeit**: Kennzeichnung von Antworten als cachebar oder nicht.
- **Layered System**: Klare Trennung von Komponenten.

Das Richardson Maturity Model bewertet den REST-Reifegrad in vier Leveln (0–3).

## 1.7 Reflexion

### 1.7.1 Gelerntes

Ich bin bereits relativ vertraut mit den Inhalten des Skripts, sodass die Vorlesung eine gute Wiederholung war. Als Werkstudent in der Backend-Entwicklung nutze ich Docker täglich, um einzelne Microservice-Container zu bauen und zu testen. Besonders bestätigt und vertieft haben sich für mich:

- Die Funktionsweise von Linux Namespaces und Cgroups für Prozess- und Ressourcen-Isolation.
- Die Bedeutung der OCI-Spezifikationen (Runtime, Image, Distribution) für standardisierte, portable Images.
- Die Rolle der Open Container Initiative und der CNCF als Gremien für Interoperabilität und Best Practices.
- Die grundsätzlichen Prinzipien von REST in Microservice-Architekturen (Uniform Interface, Zustandslosigkeit).

### 1.7.2 Offene Fragen

Aus dem Skript bleiben für mich folgende Punkte unklar und müssen nachgearbeitet werden:

- Detaillierter Umgang mit Lifecycle Hooks in der OCI Runtime Specification.
- Aufbau und Aufbauflüsse von mehrschichtigen (Layered) Container-Images nach der Image Specification.
- Absicherungsmechanismen (z. B. Signaturen) bei der Registry-Kommunikation gemäß Distribution Specification.

### 1.7.3 Vorbereitung auf das Praxisprojekt

Für den begleitenden Praxisteil plane ich, basierend auf den Skript-Inhalten:

1. Ein einfaches Dockerfile zu entwickeln, das alle Schichten (Base, App, Config) klar trennt.
2. Das erstellte Image lokal zu testen (Start, Stop, Logs) und in eine öffentliche Registry zu pushen.
3. Verschiedene Image-Tags zu verwalten und die Unterschiede im Registry-Workflow zu dokumentieren.

# 2 Docker Grundlagen

## 2.1 Command Line Interface

Das Docker CLI ist das zentrale, textbasierte Werkzeug zur Interaktion mit Docker. Mit ihm lassen sich folgende Ressourcen verwalten **dockercli**:

- Images (Erstellen, Listen, Löschen, Taggen)
- Container (Starten, Stoppen, Entfernen)
- Netzwerke (Anlegen, Löschen, Inspektion)
- Volumes (Anlegen, Löschen, Inspektion)

Es ermöglicht die vollständige Automatisierung in CI/CD-Pipelines, da keine grafische UI benötigt wird.

## 2.2 Wichtige CLI-Befehle

- `docker build` – Erstellen eines Images aus einer `Dockerfile`.
- `docker pull` – Herunterladen eines Images aus einer Registry.
- `docker push` – Hochladen eines Images in eine Registry.
- `docker images` – Auflisten aller lokalen Images.
- `docker run` – Starten eines Containers aus einem Image.
- `docker ps` – Auflisten laufender Container.
- `docker exec` – Ausführen eines Befehls in einem laufenden Container.
- `docker stop`, `docker rm` – Stoppen und Entfernen von Containern.
- `docker network` – Verwaltung von Docker-Netzwerken.

## 2.3 Docker Desktop

Docker Desktop ergänzt das CLI um eine grafische Oberfläche. Es ist Bestandteil der Installation (außer auf Linux) und bietet:

- Übersicht laufender Container, Images, Volumes und Netzwerke.
- Schnellzugriff auf Stop/Start, Logs und Metriken.
- Convenience-Features wie Learning Center und Dev Environments.

## 2.4 Image, Container und Volume

**Image** Eine unveränderliche Blaupause im Sekundärspeicher, die alle Dateien und Metadaten einer Anwendung enthält.

**Container** Eine laufende Instanz eines Images, die primäre Ressourcen (RAM, CPU) nutzt und Ports zur Interaktion anbietet.

**Volume** Persistenter Datenspeicher auf dem Host, den Container mounten, um Daten zwischen Laufzyklen zu erhalten.

## 2.5 Docker Networking

Docker unterstützt drei Netzwerkmodi **dockernetwork**:

- **Bridge (default)**: Jeder Container erhält ein eigenes virtuelles Netzwerk; Port-Forwarding verbindet Host und Container.

- **Host**: Container teilt Host-Netzwerk-Stack, keine Port-Weiterleitung nötig, aber Ports können kollidieren.
- **None**: Kein Netzwerkzugang; nützlich für isolierte Tasks ohne externe Kommunikation.

## 2.6 Dockerfiles

Dockerfiles sind deklarative Skripte zur Erzeugung von Images. Wichtige Direktiven **dockerfile**:

- `FROM` – Basis-Image.
- `RUN` – Ausführung von Befehlen und Erzeugen neuer Layer.
- `COPY/ADD` – Einfügen von Dateien.
- `WORKDIR` – Arbeitsverzeichnis festlegen.
- `EXPOSE` – Dokumentation freigegebener Ports.
- `CMD/ENTRYPOINT` – Standardausführungsbefehl.

Die Reihenfolge beeinflusst Caching und Build-Geschwindigkeit.

## 2.7 Multistage Builds

Mehrere `FROM`-Stufen reduzieren die finale Image-Größe:

1. **Builder-Stage**: Kompilation und Abhängigkeitsinstallation in einem größeren Basis-Image.
2. **Final-Stage**: Übernahme nur der benötigten Artefakte in ein schlankes Runtime-Image (z. B. `alpine`).

Dies verbessert Deploy-Geschwindigkeit und Sicherheit.

## 2.8 Reflexion

### 2.8.1 Gelerntes

Ich bin bereits vertraut mit den Docker-Grundlagen, sodass dieser Block eine kompakte Wiederholung darstellte. Besonders gefestigt wurden:

- Umgang mit dem Docker CLI und den wichtigsten Befehlen.
- Verständnis der verschiedenen Netzwerkmodi.
- Aufbau und Optimierung von Dockerfiles.
- Einsatz von Multistage Builds für schlanke Images.

### 2.8.2 Offene Fragen

Folgende Punkte möchte ich noch vertiefen:

- Zusammenspiel von Docker CLI-Befehlen in komplexen Automatisierungsskripten.
- Feintuning von Caching-Strategien in umfangreichen Dockerfiles.
- Automatisiertes Security-Scanning von Images vor dem Deployment.

### 2.8.3 Vorbereitung auf das Praxisprojekt

Für die *garden-app* plane ich:

1. Entwicklung eines Dockerfiles für die *garden-app* mit Multistage Build.
2. Lokales Testen des Images (Start, Stop, Logs) in unterschiedlichen Umgebungen.
3. Versionierung des Images und Push in eine private Registry.

# 3 Docker Grundlagen

## 3.1 Command Line Interface

Das Docker CLI ist das zentrale, textbasierte Werkzeug zur Interaktion mit Docker. Mit ihm lassen sich folgende Ressourcen verwalten **dockercli**:

- Images (Erstellen, Listen, Löschen, Taggen)
- Container (Starten, Stoppen, Entfernen)
- Netzwerke (Anlegen, Löschen, Inspektion)
- Volumes (Anlegen, Löschen, Inspektion)

Es ermöglicht die vollständige Automatisierung in CI/CD-Pipelines, da keine grafische UI benötigt wird.

## 3.2 Wichtige CLI-Befehle

- `docker build` – Erstellen eines Images aus einer `Dockerfile`.
- `docker pull` – Herunterladen eines Images aus einer Registry.
- `docker push` – Hochladen eines Images in eine Registry.
- `docker images` – Auflisten aller lokalen Images.
- `docker run` – Starten eines Containers aus einem Image.
- `docker ps` – Auflisten laufender Container.
- `docker exec` – Ausführen eines Befehls in einem laufenden Container.
- `docker stop`, `docker rm` – Stoppen und Entfernen von Containern.
- `docker network` – Verwaltung von Docker-Netzwerken.

## 3.3 Docker Desktop

Docker Desktop ergänzt das CLI um eine grafische Oberfläche. Es ist Bestandteil der Installation (außer auf Linux) und bietet:

- Übersicht laufender Container, Images, Volumes und Netzwerke.
- Schnellzugriff auf Stop/Start, Logs und Metriken.
- Convenience-Features wie Learning Center und Dev Environments.

## 3.4 Image, Container und Volume

**Image** Eine unveränderliche Blaupause im Sekundärspeicher, die alle Dateien und Metadaten einer Anwendung enthält.

**Container** Eine laufende Instanz eines Images, die primäre Ressourcen (RAM, CPU) nutzt und Ports zur Interaktion anbietet.

**Volume** Persistenter Datenspeicher auf dem Host, den Container mounten, um Daten zwischen Laufzyklen zu erhalten.

## 3.5 Docker Networking

Docker unterstützt drei Netzwerkmodi **dockernetwork**:

- **Bridge (default)**: Jeder Container erhält ein eigenes virtuelles Netzwerk; Port-Forwarding verbindet Host und Container.
- **Host**: Container teilt Host-Netzwerk-Stack, keine Port-Weiterleitung nötig, aber Ports können kollidieren.
- **None**: Kein Netzwerkzugang; nützlich für isolierte Tasks ohne externe Kommunikation.

## 3.6 Dockerfiles

Dockerfiles sind deklarative Skripte zur Erzeugung von Images. Wichtige Direktiven **dockerfile**:

- `FROM` – Basis-Image.
- `RUN` – Ausführung von Befehlen und Erzeugen neuer Layer.
- `COPY/ADD` – Einfügen von Dateien.
- `WORKDIR` – Arbeitsverzeichnis festlegen.
- `EXPOSE` – Dokumentation freigegebener Ports.
- `CMD/ENTRYPOINT` – Standardausführungsbefehl.

Die Reihenfolge beeinflusst Caching und Build-Geschwindigkeit.

## 3.7 Multistage Builds

Mehrere `FROM`-Stufen reduzieren die finale Image-Größe:

1. **Builder-Stage**: Kompilation und Abhängigkeitsinstallation in einem größeren Basis-Image.
2. **Final-Stage**: Übernahme nur der benötigten Artefakte in ein schlankes Runtime-Image (z. B. `alpine`).

Dies verbessert Deploy-Geschwindigkeit und Sicherheit.

## 3.8 Reflexion

### 3.8.1 Gelerntes

Ich bin bereits vertraut mit den Docker-Grundlagen, sodass dieser Block eine kompakte Wiederholung darstellte. Besonders gefestigt wurden:

- Umgang mit dem Docker CLI und den wichtigsten Befehlen.
- Verständnis der verschiedenen Netzwerkmodi.
- Aufbau und Optimierung von Dockerfiles.
- Einsatz von Multistage Builds für schlanke Images.

### 3.8.2 Offene Fragen

Folgende Punkte möchte ich noch vertiefen:

- Zusammenspiel von Docker CLI-Befehlen in komplexen Automatisierungsskripten.
- Feintuning von Caching-Strategien in umfangreichen Dockerfiles.
- Automatisiertes Security-Scanning von Images vor dem Deployment.

### 3.8.3 Vorbereitung auf das Praxisprojekt

Für die *garden-app* plane ich:

1. Entwicklung eines Dockerfiles für die *garden-app* mit Multistage Build.
2. Lokales Testen des Images (Start, Stop, Logs) in unterschiedlichen Umgebungen.
3. Versionierung des Images und Push in eine private Registry.

# 4 Docker Compose

## 4.1 Einführung

Docker Compose ist ein Tool zur Definition und Ausführung von Multi-Container-Anwendungen. Eine einzige YAML-Datei (`docker-compose.yml`) beschreibt Services, Netzwerke und Volumes, sodass sich komplette Systeme mit einem Befehl starten und stoppen lassen **compose-gettingstarted**.

## 4.2 Nutzungsszenario

Docker Compose eignet sich besonders für:

- Lokale Entwicklung: Einheitlicher Start aller Services (API, Datenbank, Worker).
- Pipeline-Testing: Integrationstests per CI/CD, da das gesamte System per CLI verfügbar ist.
- Single-Host-Betrieb: Orchestrierung mehrerer Container auf einem Host.

## 4.3 Wichtige Befehle

- `docker compose up [-d]` – Erzeugt und startet alle definierten Services.
- `docker compose down` – Stoppt und entfernt Container, Netzwerke und Volumes.
- `docker compose ps` – Listet laufende Container des Stacks.
- `docker compose logs [-f]` – Zeigt Logs aller oder einzelner Services.
- `docker compose build` – Baut Images gemäß `build`-Konfiguration.
- `docker compose pull` – Lädt alle benötigten Images aus der Registry.

## 4.4 Aufbau der `docker-compose.yml`

Die Datei gliedert sich in:

- `version`: Versionsangabe des Compose-Formats (optional).
- `services`: Definition der Container (Image, Build, Ports, Environment, ...).
- `networks`: Benutzerdefinierte Netzwerke zur Service-Kommunikation.
- `volumes`: Persistente Datenspeicher für Services.

## 4.5 Service-Konfiguration

Unter `services:` lassen sich konfigurieren:

- `build`: Kontext und Dockerfile für automatischen Image-Build.
- `image`: Name und Tag des Images.
- `ports`: Host-zu-Container-Port-Mappings.
- `environment`: Umgebungsvariablen oder `env_file`.
- `depends_on`: Steuerung der Startreihenfolge der Services.

## 4.6 Reflexion

### 4.6.1 Gelerntes

Docker Compose hat mir gezeigt, wie ich meine *garden-app* mit allen Komponenten (API, Datenbank, Scheduler) in einer Datei konsistent starten kann. Besonders hilfreich war:

- Einfache Definition und Start aller Container mit `docker compose up`.
- Automatisches Bauen eigener Images per `build`-Angabe.
- Deklarative Verwaltung von Netzwerken und Volumes.

### 4.6.2 Offene Fragen

Folgende Themen möchte ich noch vertiefen:

- Einsatz von `healthcheck` in Compose-Dateien für robustere Startsequenzen.
- Unterschiede und Migration von Compose V1 (`docker-compose`) zu V2 (`docker compose`).
- Best Practices für große Compose-Dateien in CI/CD-Umgebungen.

### 4.6.3 Vorbereitung auf das Praxisprojekt

Für die *garden-app* plane ich:

1. Erstellung einer vollständigen `docker-compose.yml`, die API, Datenbank und Scheduler orchestriert.
2. Pflege von Override-Dateien (`docker-compose.override.yml`) für Entwicklungs- und Testumgebungen.
3. Integration von `docker compose` in die GitLab-CI, um beim Pipeline-Testing automatisiert aufzubauen und zu testen.

# 5 5 Microservices – Konzepte & Architektur

## 5.1 5.1 Definition und Prinzipien

- **Microservices** sind kleine, eigenständige Services, die jeweils eine klar definierte Aufgabe besonders gut erfüllen und kollaborieren, indem sie über das Netzwerk kommunizieren :contentReference[oaicite:0]index=0.
- Kernprinzipien (nach McIlroy et al. [B]):
  - *Single Responsibility*: Jeder Service soll genau eine Sache tun und diese exzellent umsetzen.
  - *Chaining of Outputs*: Der Output eines Services dient als Input für andere, unbekannte Services.
  - *Early Feedback*: Schnell lauffähige Services liefern rasches Feedback.
  - *Standardisierte Tools*: Wiederverwendbare, standardisierte Services statt proprietärer Lösungen.

## 5.2 5.2 Abgrenzung

**Monolith vs. Microservice** Ein **Monolith** ist eine eng gekoppelte Anwendung, die als ein einziges Artefakt skaliert wird. **Microservices** sind lose gekoppelt, skalierbar und erlauben unterschiedliche Technologien pro Service :contentReference[oaicite:1]index=1.

**SOA vs. Microservice SOA** (Service Orientierte Architektur) fokussiert Wiederverwendbarkeit auf Unternehmensebene; **Microservices** operieren auf granulierter, anwendungsbezogener Ebene mit Fokus auf Unabhängigkeit und Skalierbarkeit :contentReference[oaicite:2]index=2.

**Microservice vs. FaaS Microservices** laufen dauerhaft als Container, erlauben (leichtgewichtigen) Zustand; **FaaS** ist zustandslos, abstrahiert Laufzeit und Deployment vollständig vom Entwickler :contentReference[oaicite:3]index=3.

## 5.3 5.3 Domain-Driven Design

Domain-Driven Design (DDD) bietet Konzepte zur sauberen Aufteilung:

- **Ubiquitous Language**: Gemeinsame Fachsprache aller Beteiligten.
- **Bounded Context**: Abgegrenzte Fachkontexte mit eigener Sprache und Modellen.
- **Domain Model**: Visuelle Darstellung der Konzepte und Beziehungen innerhalb eines Kontextes :contentReference[oaicite:4]index=4.

## 5.4 5.4 Architekturprinzipien

- **Lose Kopplung**: Services kommunizieren über standardisierte Schnittstellen, Änderungen bleiben lokal.
- **Hohe Kohäsion**: Jeder Service fokussiert sich auf eine klar definierte Funktion.
- **Autonomie**: Unabhängige Entwicklung, Deployment und Skalierung.
- **Verantwortlichkeit**: Jeder Service verwaltet seine eigene Geschäftslogik und Datenhaltung :contentReference[oaicite:5]index=5.

## 5.5 5.5 Reflexion

### 5.5.1 5.5.1 Gelerntes

Die Konzepte zu Microservices und DDD haben mein Verständnis vertieft, wie man ein System in lose gekoppelte, hoch kohäsive Dienste zerlegt. Für die *garden-app* werde ich:

- Use Cases klar definieren und jedem Microservice eine einzige Verantwortung zuweisen.
- Bounded Contexts entlang der Domänenlogik der *garden-app* modellieren.
- Eigene Datenhaltung pro Service implementieren, um lose Kopplung zu gewährleisten.

### 5.5.2 5.5.2 Offene Fragen

- Wie granular sollte ein Microservice in der *garden-app* sein, um Entwicklungsaufwand und Wartbarkeit auszutarieren?
- Welche Patterns für Service-Discovery und Konfigurationsmanagement sind empfehlenswert?

### 5.5.3 5.5.3 Vorbereitung auf das Praxisprojekt

1. Identifikation von drei Kern-Domänen der *garden-app* (z. B. Plant-Management, Scheduling, Reporting) als Bounded Contexts.
2. Erstellen erster Prototyp-Services mit eigener API und Datenhaltung für jeden Kontext.

# 6 6 Entwicklung von Microservices

## 6.1 6.1 Implementierung von Microservices

In dieser Einheit geht es um den grundlegenden Aufbau einer Microservice-Architektur. Microservices sollen so konzipiert sein, dass sie weitgehend unabhängig voneinander betrieben werden können. Dazu vermeiden sie synchrone Daueraufrufe zu anderen Services, indem sie benötigte Daten lokal oder in einem Cache vorhalten.

- Jeder Service verwaltet seine eigene Datenbank (z. B. PostgreSQL für *garden*, Redis-Cache für häufige Lesezugriffe).
- Gemeinsame Verträge (APIs) werden über OpenAPI/Swagger dokumentiert.
- Versionierung der APIs ermöglicht rückwärtskompatible Änderungen.

## 6.2 6.2 Lose Kopplung

Microservices kommunizieren über klar definierte HTTP-/gRPC-APIs oder asynchrone Message Broker. Eine lose Kopplung stellt sicher, dass ein Ausfall oder Update eines Services die anderen nicht beeinträchtigt.

- **HTTP/gRPC:** Synchrone Aufrufe nur für seltene, transaktionale Anfragen.
- **Message Broker:** Asynchrone Events (z. B. mit Kafka) transportieren Domain-Events wie `GardenCreated` oder `TreePlanted`.
- **Circuit Breaker & Retries:** Resilience4j sorgt für Ausfallsicherheit und Drosselung bei Netzwerkproblemen.

## 6.3 6.3 Resilienz und Fehlertoleranz

Microservices müssen Ausfälle einzelner Komponenten tolerieren. Wichtige Mechanismen:

- **Timeouts:** Abbruch von Remote-Calls nach konfigurierbarer Dauer (z. B. 5 s).
- **Bulkheads:** Isolierung von Ressourcenpools, um Kaskadeneffekte zu verhindern.
- **Circuit Breaker:** Automatisches Schalten auf Fallback-Pfade bei wiederholten Fehlern.
- **Fallbacks:** Vorhalten von Default-Antworten oder Zwischenergebnissen, wenn ein Service nicht erreichbar ist.

## 6.4 6.4 CI/CD-Pipelines

Jeder Microservice benötigt eine eigene Pipeline zur Automatisierung von Build, Test und Deployment. In deinem Projekt könnte das so aussehen:

- **Build:** Maven-Build und Docker-Image-Erstellung (‘mvn clean package  docker build‘).
- **Test:** Unit- und Integrationstests innerhalb einer ephemeral Kubernetes-Umgebung via Kind.
- **Deploy:** Automatisches „helm upgrade“ in das Entwicklungs- oder Test-Namespace.
- **GitOps:** Argo CD oder Flux synchronisieren Repos mit dem Cluster.

## 6.5 6.5 Organisatorische Anpassungen

Der Betrieb von Microservices erfordert eine angepasste Teamstruktur:

- **Cross-funktionale Teams:** Jede Gruppe betreut Frontend, Backend und Infrastruktur eines Services.
- **DevOps-Kultur:** Gemeinsame Ownership von Code, CI/CD und Monitoring.
- **Knowledge-Sharing:** Regelmäßige Architektur-Reviews und Pair-Programming.

## 6.6 6.6 Komplexität im Betrieb

Ein verteiltes System bringt neue Herausforderungen mit sich:

- **Monitoring & Logging:** Zentralisierte Aggregation mit Prometheus und Loki.
- **Tracing:** End-to-End-Transaktionsverfolgung via OpenTelemetry und Tempo.
- **Service Discovery:** Automatisches Auffinden von Services über Kubernetes-Services und DNS.
- **Load Balancing:** Externe Zugriffe via Ingress-Controller (NGINX) und internen Traffic über ClusterIP-Services.

## 6.7 6.7 Event Driven Architecture (Optional)

Als Ergänzung zu synchronen APIs kann eine ereignisgesteuerte Architektur eingesetzt werden:

- Domain-Events werden in Topics geschrieben (z. B. ‘garden.events‘).
- Services abonnieren nur relevante Topics und reagieren ereignisgesteuert.
- Vorteile: Entkoppelte Skalierung, höhere Fehlertoleranz und flexible Erweiterbarkeit.

## 6.8 6.8 Reflexion

### 6.8.1 6.8.1 Gelerntes

- Microservices erfordern klare Verantwortungsgrenzen und eigene Datenhaltung.
- Lose Kopplung über HTTP und Messaging erhöht die Stabilität.
- Resilience-Patterns wie Circuit Breaker und Bulkheads sind essenziell.

### 6.8.2 6.8.2 Offene Fragen

- Wie verwaltet man schema-evolution in Event-Topics für langfristige Kompatibilität?
- Welche Best Practices gibt es für canary Releases in Kubernetes?
- Wie konfiguriere ich Alerting-Regeln in Prometheus für komplexe Service-Abhängigkeiten?

### 6.8.3 6.8.3 Vorbereitung auf das Praxisprojekt

1. Implementierung eines Redis-Sidecars für Caching in der *garden-app*.
2. Einrichtung einer OpenTelemetry-Collector-Instanz, um Traces und Metriken zentral zu sammeln.
3. Automatisierung von Helm-Rollouts mit Canary-Strategie über Argo Rollouts.

# 7 7 Kubernetes Einführung

## 7.1 7.1 Historische Entwicklung

Kubernetes geht auf Googles interne Cluster-Orchestrierungsprojekte Borg (2003–2004) und Omega (2006–2008) zurück. 2014 wurde das Projekt als Open Source unter dem Namen „Steuermann/Pilot“ vorgestellt und 2015 an die Cloud Native Computing Foundation (CNCF) übergeben, wo es heute De-Facto-Standard für Container-Orchestrierung ist **hightower2017**.

## 7.2 7.2 Definition und Nutzen

Kubernetes ist eine portable, erweiterbare Open-Source-Plattform zur Verwaltung containerisierter Arbeitslasten und Services mit deklarativer Konfiguration und Automatisierung. Kernvorteile:

- **Skalierbarkeit**: Automatisches Hoch- und Runterskalieren von Pods.
- **Portabilität**: Einheitliches Deployment über verschiedene Umgebungen.
- **Automatisierung**: Self-Healing, Rolling Updates und deklarative Konfiguration.

## 7.3 7.3 Architekturübersicht

**Control Plane** Enthält den `kube-apiserver` (Hauptschnittstelle), `etcd` (Key-Value-Store), `kube-scheduler` (Pod-Verteilung), `kube-controller-manager` (Status-Controller) und den `cloud-controller-manager` **burns2019**.

**Worker Nodes** Auf jedem Node laufen `kubelet` (Pod-Lifecycle), `kube-proxy` (Netzwerk-Proxy) und eine Container-Runtime (Docker, containerd, CRI-O) **burns2019**.

## 7.4 7.4 Kubernetes Ressourcen

- **Pod**: Kleinste Ausführungseinheit, Umschlag für einen oder mehrere Container.
- **ReplicaSet**: Sorgt für eine definierte Anzahl identischer Pod-Instanzen.
- **Deployment**: Orchestriert ReplicaSets, ermöglicht Rolling Updates und Rollbacks.
- **DaemonSet**: Stellt sicher, dass auf jedem Node eine Pod-Instanz läuft.
- **ConfigMap & Secret**: Externe Konfiguration bzw. vertrauliche Daten.
- **PersistentVolume & PersistentVolumeClaim**: Abstraktion für persistente Speichervolumes.
- **Ingress, Namespace, NetworkPolicy, HPA, CRD**: Zusätzliche Ressourcen für Routing, Isolation, Skalierung und Erweiterung.

## 7.5 7.5 Features

- **Rolling Updates**: Neue Versionen ohne Downtime bereitstellen.
- **Self-Healing**: Neustart fehlerhafter Pods durch Readiness- und Liveness-Probes.
- **Horizontal Pod Autoscaler**: Automatische Skalierung anhand von Metriken.
- **Load Balancing & Service Discovery**: Dienste verteilen Traffic auf Pods.
- **Rollbacks**: Schnelles Zurücksetzen auf eine stabile Version.

## 7.6 7.6 Netzwerkmodell

Jeder Pod erhält eine eigene IP und kann ohne NAT mit anderen Pods und Nodes kommunizieren. Services abstrahieren den Zugriff über virtuelle IPs und DNS-Namen, um Lastverteilung zu ermöglichen **kubernetesNetworking**.

## 7.7 7.7 Ökosystem

- **Helm**: Paketmanager für Kubernetes-Charts.
- **K9s**: Terminal-UI für die Cluster-Inspektion.
- **Monitoring Stack**: Telemetrie mit Prometheus, Loki und Grafana.

## 7.8 7.8 Reflexion

### 7.8.1 7.8.1 Gelerntes

Die Einführung in Kubernetes hat mir gezeigt, wie ich die *garden-app* später in einem Cluster betreiben kann. Besonders klar wurde:

- Die Rolle der Control Plane und der

# 8 8 Deployment und Skalierung

## 8.1 8.1 Kubectl

Das CLI-Tool `kubectl` ermöglicht die Verwaltung von Kubernetes-Clustern per YAML-Definitionen:

- `kubectl get <resource>` – Listet Ressourcen (z. B. Pods, Deployments).
- `kubectl describe <resource> <name>` – Zeigt Details einer Ressource.
- `kubectl apply -f <file.yml>` – Wendet eine Konfiguration an (Create/Update).
- `kubectl delete -f <file.yml>` – Entfernt alle in der Datei definierten Ressourcen.

## 8.2 8.2 Lebenszyklus eines Deployments

1. `kubectl apply -f deployment.yml` – Erstellt oder aktualisiert das Deployment.
2. `kubectl get pods` – Überprüft, dass die Pods laufen.
3. `kubectl logs <pod-name>` – Einsicht in die Container-Logs.
4. `kubectl delete -f deployment.yml` – Entfernt Deployment, ReplicaSet und Pods.

## 8.3 8.3 Pod

- Kleinste Ausführungseinheit, enthält einen oder mehrere Container.
- Definition über `apiVersion: v1`, `kind: Pod`, `metadata` und `spec.containers`.

## 8.4 8.4 ReplicaSet

- Sorgt für eine konstante Anzahl von Pod-Replikaten.
- `apiVersion: apps/v1`, `kind: ReplicaSet`, `spec.replicas`, `spec.selector` und `spec.template`.

## 8.5 8.5 Deployment

- Baut auf ReplicaSets auf, ermöglicht deklarative Updates (Rolling Updates, Rollbacks).
- Fast identisch zu ReplicaSet-Definition, `kind: Deployment`.

## 8.6 8.6 Quality of Service (Resource Quota)

- `resources.requests` reserviert CPU/Mem für Pods.
- `resources.limits` setzt Obergrenzen.
- QoS-Klassen: **Guaranteed** (requests = limits), **Burstable**, **BestEffort**.

## 8.7 8.7 DaemonSet

- Gewährleistet, dass von einem Pod auf jedem Node eine Instanz läuft.
- Definition analog zu ReplicaSet, jedoch ohne `spec.replicas`.

## 8.8 8.8 Service

- Abstraktion für Zugriff auf Pods via Label-Selector.
- Typen: `ClusterIP`, `NodePort`, `LoadBalancer`.
- Definition: `kind: Service`, `spec.selector`, `spec.ports`.

## 8.9 8.9 ConfigMap & Secret

- ConfigMap: Schlüssel–Wert-Paare für Konfiguration (`apiVersion: v1`, `kind: ConfigMap`).
- Secret: sensibler Datenspeicher (Base64-codiert, `kind: Secret`).
- Einbindung als Environment-Variablen oder Volumes im Pod-Template.

## 8.10 8.10 PersistentVolume & PersistentVolumeClaim

- PV: Cluster-Ressource mit Storage-Kapazität und Zugriffsmodi.
- PVC: Anforderung an PV (`spec.resources.requests.storage`).
- 1:1-Zuordnung zwischen PVC und PV via StorageClass und Größe.

## 8.11 8.11 Namespace

- Logische Trennung von Ressourcen im Cluster.
- Definition über `apiVersion: v1`, `kind: Namespace`.

## 8.12 8.12 Ingress

- HTTP/HTTPS-Routing zu Services anhand von Host- und Pfad-Regeln.
- Erfordert Ingress-Controller (z. B. NGINX, Traefik).
- Definition: `kind: Ingress`, `spec.rules`, `spec.backend`.

## 8.13 8.13 NetworkPolicy

- Steuerung von Ingress/Egress-Traffic auf Pod-Ebene.
- Whitelist- oder Blacklist-Modell.
- Definition: `kind: NetworkPolicy`, `spec.podSelector`, `spec.policyTypes`.

## 8.14 8.14 Horizontal Pod Autoscaler

- Automatisches Skalieren von Deployments nach Metriken (CPU, Memory).
- Definition: `kind: HorizontalPodAutoscaler`, `spec.scaleTargetRef`, `spec.minReplicas`, `spec.maxReplicas`, `spec.metrics`.

## 8.15 8.15 Fremdanwendungen mit Helm installieren

- Helm-Charts als paketierte YAML-Sammlungen (Values-Datei zur Parametrisierung).
- Befehle: `helm repo add <name> <url>`, `helm repo update`, `helm install <release> <chart> [-f values.yml]`.

# 9 9 Telemetrie in Kubernetes

## 9.1 9.1 Was ist Telemetrie?

Telemetrie bezeichnet die Sammlung und Auswertung relevanter Mess- und Rohdaten, die von Hintergrunddiensten bereitgestellt werden, um die Nutzung, Performance und Zuverlässigkeit einer Anwendung zu verstehen und zu optimieren **otel2021**.

## 9.2 9.2 Logs

Logs sind detaillierte Aufzeichnungen von Ereignissen. Wichtige Log-Level:

- **Trace**: Sehr ausführlich, selten im Betrieb.
- **Debug**: Diagnose-Informationen während Entwicklung und Test.
- **Info**: Allgemeine Betriebsinformationen.
- **Warning**: Unerwartete, aber nicht kritische Ereignisse.
- **Error**: Fehler, die Funktionen beeinträchtigen.
- **Fatal**: Kritische Fehler, die zum Abbruch führen.

Logs können strukturiert (z. B. JSON) oder unstrukturiert sein **loki2020**.

## 9.3 9.3 Metriken

Metriken sind numerische Werte zur Überwachung. Häufige Metriken:

- CPU-Auslastung (%).
- Speicherauslastung.
- Anfrage-Rate (Requests/s).
- Fehlerrate.
- Latenzzeiten.

Sie bieten einen kompakten Überblick und unterstützen die Anomalieerkennung **prometheus2015**.

## 9.4 9.4 Traces

Traces verfolgen den Pfad einzelner Anfragen über mehrere Services:

- **Spans**: Einzelschritte innerhalb einer Transaktion.
- **Trace-ID**: Eindeutige Kennung zur Nachverfolgung.
- Unterstützen Performance-Analysen in verteilten Systemen.

## 9.5 9.5 Kubernetes Metrics Server

Der Metrics Server liefert technische Metriken (CPU, RAM) für den Horizontal Pod Autoscaler (HPA):

- Geringer Ressourcenverbrauch, skaliert bis zu 5000 Knoten.
- Nicht geeignet für anwendungsspezifische Langzeitmetriken.

## 9.6 9.6 Prometheus

Prometheus sammelt Metriken per Scraping:

- Regelmäßiges Abfragen definierter Endpunkte.
- Speicherung in einer Time-Series-Datenbank.
- Abfragen via PromQL und Alerting über den Alertmanager.

## 9.7 9.7 Loki

Loki ist ein log-agnostischer Aggregator:

- Automatisches Sammeln von Pod-Logs.
- Indexierung nach Labels (Pod, Namespace, Level).
- Speicherung in einer Time-Series-Datenbank und Bereitstellung per API.

## 9.8 9.8 Tempo

Tempo erfasst und speichert verteilte Traces:

- Spans werden aktiv aus dem Code an Tempo gesendet.
- Speicherung im Tempo-Backend, Visualisierung über Grafana.

## 9.9 9.9 OpenTelemetry

OpenTelemetry bietet einen einheitlichen Standard für Logs, Metriken und Traces:

- SDKs für mehrere Sprachen.
- Collector leitet Daten an Prometheus, Loki und Tempo weiter.

## 9.10 9.10 Grafana

Grafana ist das zentrale Dashboard- und Alerting-Tool:

- Visualisierung von Metriken, Logs und Traces.
- Erstellung individueller Dashboards und Panels.
- Integration mit Alertmanager-Backends.

## 9.11 9.11 Installation

Der `kube-prometheus-stack`-Helm-Chart installiert Prometheus, Grafana, Loki und Tempo mit vorkonfigurierten Dashboards.

## 9.12 9.12 Portfolio-Auftrag

Implementiere für die *garden-app* einen Telemetrie-Aspekt:

- Wähle zwischen Logging, Metriken oder Tracing.
- Installiere den `kube-prometheus-stack`-Chart.
- Erstelle ein Grafana-Dashboard mit mindestens einem Panel zur Visualisierung deiner Telemetriedaten.

## 9.13 9.13 Reflexion

### 9.13.1 9.13.1 Gelerntes

Die Telemetrie-Einheit hat mein Verständnis für Monitoring-Tools geschärft und die Integration von Logs, Metriken und Traces im Kubernetes-Kontext verdeutlicht.

### 9.13.2 9.13.2 Offene Fragen

- Wie konfiguriere ich optimale Scrape-Intervalle für Prometheus?
- Wie lässt sich der OpenTelemetry Collector hochverfügbar betreiben?
- Welche Best Practices existieren für Security und Zugriffskontrolle auf Telemetriedaten?

### 9.13.3 9.13.3 Vorbereitung auf das Praxisprojekt

1. Einrichtung des `kube-prometheus-stack`-Helm-Charts im Test-Cluster.
2. Instrumentierung der *garden-app* mit dem OpenTelemetry SDK.
3. Erstellung eines Grafana-Dashboards mit Log-, Metrik- und Trace-Panels.

# 10 10 RBAC – Roles and Bindings

## 10.1 10.1 Einführung

RBAC (Role-Based Access Control) ist ein Autorisierungsmechanismus in Kubernetes, der festlegt, welche Benutzer oder Dienste welche Aktionen im Cluster ausführen dürfen. Berechtigungen werden nicht direkt an Individuen vergeben, sondern über Rollen und Bindings zugewiesen.

## 10.2 10.2 Aktivierung

RBAC ist standardmäßig in Kubernetes integriert. Der API-Server muss mit `--authorization-mod` gestartet werden, damit die API-Gruppe `rbac.authorization.k8s.io` für Autorisierungsentscheidungen genutzt wird.

## 10.3 10.3 Rollen

**Role** Definiert Berechtigungen innerhalb eines Namespaces (z. B. Lesezugriff auf Pods im Namespace `default`).

**ClusterRole** Gilt clusterweit und kann Zugriff auf Namespaced- wie auf nicht-Namespaced-Ressourcen gewähren (z. B. `nodes` oder `/healthz`).

## 10.4 10.4 Bindings

**RoleBinding** Verknüpft eine `Role` oder `ClusterRole` mit Usern, Gruppen oder ServiceAccounts innerhalb eines Namespaces.

**ClusterRoleBinding** Bindet eine `ClusterRole` an User, Gruppen oder ServiceAccounts über alle Namespaces hinweg.

## 10.5 10.5 Regeln („rules“)

Jede Rolle enthält eine Liste von `rules`, die angeben:

- `apiGroups`: Betroffene API-Gruppen (z. B. für Core).
- `resources`: Ressourcen (z. B. `pods`, `secrets`).
- `resourceNames`: Optional einzelne Objektnamen.
- `verbs`: Erlaubte Aktionen (z. B. `get`, `create`, `delete`).

Deny-Regeln existieren nicht — nur `Allow`.

## 10.6 10.6 Best Practices

- Principle of Least Privilege: Nur erforderliche Rechte vergeben.
- Vermeidung von Wildcards (`*`), um ungewollte Rechteausdehnung zu verhindern.
- Nutzung von `ServiceAccounts` pro Anwendung statt `cluster-admin`.
- Aggregierte Rollen (`aggregate-to-...`) und Standardrollen (`view`, `edit`, `admin`) nutzen und bei Bedarf erweitern.

## 10.7 10.7 Praktische Beispiele

```
# Role im Namespace default
kubectl create role pod-reader \
  --verb=get,list,watch \
  --resource=pods \
  --namespace=default

# RoleBinding für Benutzer jane
kubectl create rolebinding jane-read-pods \
  --role=pod-reader \
  --user=jane \
  --namespace=default

# ClusterRoleBinding für alle ServiceAccounts
kubectl create clusterrolebinding sa-view-all \
  --clusterrole=view \
  --group=system:serviceaccounts
```

## 10.8 10.8 Reflexion

### 10.8.1 10.8.1 Gelerntes

- RBAC ermöglicht granulare, wiederverwendbare Zugriffsregeln durch Rollen und Bindings.
- ClusterRoles und RoleBindings erlauben differenzierte Steuerung auf Namespace- und Cluster-Ebene.
- Die Prinzipien „Least Privilege“ und Vermeidung von Wildcards sind essenziell für Sicherheit.

### 10.8.2 10.8.2 Offene Fragen

- Wie lässt sich RBAC mit externen Identity Providern (z. B. OIDC) integrieren?
- Welche Tools unterstützen die Visualisierung und Verwaltung komplexer RBAC-Regeln?

### 10.8.3 10.8.3 Vorbereitung auf das Praxisprojekt

Für die *garden-app* plane ich:

1. Anlegen separater `ServiceAccounts` für API, Scheduler und Datenzugriff.
2. Definition von Rollen mit minimalen Rechten für jeden ServiceAccount.
3. Automatisches Erstellen und Aktualisieren von Bindings per GitOps-Prozess.