Microservices

Microservices sind ein Architekturparadigma, das sich in den letzten zehn Jahren stark verbreitet hat. Der Begriff steht für eine Herangehensweise, bei der Software nicht mehr als eine große, monolithische Anwendung entwickelt und betrieben wird, sondern als Sammlung kleiner, voneinander unabhängiger Dienste. Jeder dieser Dienste erfüllt eine klar abgegrenzte Aufgabe innerhalb des Gesamtsystems und kommuniziert mit anderen Diensten über wohldefinierte Schnittstellen.

Ursprünge und Motivation

Die Idee der Microservices entwickelte sich im Kontext wachsender monolithischer Systeme, die mit zunehmendem Umfang schwer wart- und testbar wurden. Die Beobachtung: Je größer der Code und je mehr Teams beteiligt sind, desto größer wird die Reibung beim gemeinsamen Arbeiten an einer gemeinsamen Codebasis. Änderungen in einem Bereich ziehen häufig Änderungen in anderen Bereichen nach sich, und das Deployment eines Features kann durch die Abhängigkeit von nicht fertiggestellten Teilbereichen blockiert sein.

Microservices sind eine Antwort auf diese Skalierungsprobleme. Sie setzen auf:

- **Lose Kopplung**: Jeder Dienst ist weitgehend unabhängig von den anderen.
- **Hohe Kohäsion**: Jeder Dienst konzentriert sich auf eine spezifische Funktion oder Domäne.
- **Unabhängige Deployments**: Jeder Dienst kann einzeln aktualisiert werden.
- **Technologievielfalt**: Dienste können mit verschiedenen Sprachen und Frameworks implementiert werden.

Technisches Grundprinzip

Ein Microservice-System besteht aus mehreren eigenständigen Prozessen, die über Netzwerkprotokolle miteinander kommunizieren – in der Regel über HTTP (REST oder GraphQL), gRPC oder asynchrone Messaging-Systeme wie Kafka oder RabbitMQ.

Jeder Dienst bringt idealerweise seine gesamte technische Infrastruktur mit:

- eigenes Repository

- eigene Build- und Deployment-Pipeline
- eigene Datenbank oder zumindest isolierten Zugriff auf persistente Daten

Diese vollständige Kapselung wird in der Praxis jedoch nicht immer konsequent umgesetzt. Besonders im Bereich der Datenpersistenz kommt es oft zu Überschneidungen, etwa wenn mehrere Dienste auf dieselbe Datenbank oder Tabelle zugreifen müssen. Das steht im Spannungsfeld zum Prinzip der vollständigen Isolation.

Paradigmen und Interpretationen

Es gibt verschiedene Interpretationen und Ableitungen des Microservices-Gedankens:

- **Domain-Driven Design (DDD)**: Dienste orientieren sich an fachlichen Domänen („Bounded Contexts“) und spiegeln die Struktur der Geschäftslogik wider.
- **Self-Contained Systems (SCS)**: Jeder Dienst enthält Backend, Businesslogik und sogar das zugehörige UI.
- **Modular Monolith**: Die Anwendung bleibt ein Prozess, ist aber intern stark modularisiert und potenziell in Microservices aufteilbar.

Diese Vielfalt führt dazu, dass unter dem Begriff „Microservices“ oft unterschiedliche Dinge verstanden werden. Eine klare und konsistente Definition fehlt bis heute.

Vorteile von Microservices

Vorteil	Beschreibung
Skalierbarkeit	Einzelne Services lassen sich unabhängig skalieren (z. B. CPU-hungrige Module).
Flexibilität	Technologieentscheidungen können pro Dienst individuell getroffen werden.
Deployment-Freiheit	Teams können unabhängig voneinander releasen.
Fehlertoleranz	Ein Fehler in einem Dienst betrifft nicht notwendigerweise das Gesamtsystem.
Teamautonomie	Kleinere Teams können Verantwortung für „ihren“ Dienst übernehmen.

Herausforderungen und Kritik

Nachteil / Herausforderung	Beschreibung
Systemkomplexität	Die Gesamtarchitektur wird komplexer, insbesondere hinsichtlich Kommunikation und Datenfluss.
Testing-Aufwand	Integrationstests und End-to-End-Tests werden aufwendiger.
Verteilte Transaktionen	ACID-Eigenschaften sind über mehrere Dienste schwer zu garantieren.
Fehlende Übersicht	Es kann schwierig sein, einen systemweiten Überblick zu behalten.
Tooling & Infrastruktur	Microservices benötigen reifes CI/CD, Observability, Logging, Monitoring.

Entwicklung und Trendwende

Microservices galten über Jahre hinweg als das Ideal moderner Softwarearchitektur. Viele Unternehmen haben ihre Systeme unter großem Aufwand von Monolithen auf Microservices umgestellt. Mittlerweile mehren sich jedoch die Stimmen, die auf die Nachteile und Überforderungen durch zu viele verteilte Komponenten hinweisen.

In der Praxis zeigt sich: Nicht jedes Team ist darauf vorbereitet, eine derart feingranulare Architektur sinnvoll zu betreiben. Auch große Unternehmen wie Amazon oder Uber haben Teile ihrer Architektur wieder konsolidiert oder stark modularisierte Monolithen eingeführt.

Derzeit entstehen vermehrt Architekturen, die eine Balance zwischen Modularität und Einfachheit suchen:

- **Modulare Monolithen** mit klar abgegrenzten Domänen und interner API-Struktur
- **„Micro-Frontends“** als Antwort auf verteilte Zuständigkeiten im UI-Bereich
- **Backend-for-Frontend (BFF)**-Muster zur Trennung von domänenspezifischer Darstellung

Wann sind Microservices sinnvoll?

Microservices lohnen sich besonders, wenn folgende Bedingungen erfüllt sind:

- Das System ist fachlich sehr komplex und wächst weiter.

- Es gibt mehrere Entwicklungsteams, die unabhängig voneinander arbeiten sollen.
- Die Anwendung muss stark skalieren oder hohe Verfügbarkeit garantieren.
- Es besteht ein Bedarf an Technologievielfalt oder Dienstgrenzen entlang von Domänen.

Für kleinere Projekte oder Teams kann ein gut strukturierter Monolith hingegen **deutlich effektiver** und wartbarer sein.

Infrastruktur und Orchestrierung

Microservices entfalten ihr volles Potenzial erst mit der passenden Infrastruktur. Zwei Schlüsseltechnologien, die den Betrieb verteilter Systeme ermöglichen, sind:

- **Docker**: Eine Container-Technologie, mit der einzelne Microservices isoliert, reproduzierbar und unabhängig voneinander paketiert werden können. Jeder Dienst läuft in seinem eigenen Container mit definierter Laufzeitumgebung.
- **Kubernetes**: Eine Open-Source-Plattform zur Orchestrierung von Containern (wie Docker). Kubernetes verwaltet die Verteilung, Skalierung, Wiederherstellung und Kommunikation von Microservices über einen Cluster aus Maschinen hinweg. Es ist damit das Rückgrat vieler Cloud-nativer Architekturen.

Ein „Cluster“ ist in diesem Zusammenhang eine Gruppe vernetzter physischer oder virtueller Server, auf denen Kubernetes die Microservices verteilt und steuert.

Der Aufbau eines produktionsreifen Microservice-Systems setzt daher Kenntnisse in Containerisierung und Orchestrierung voraus – ein Grund, warum der Infrastrukturaufwand im Vergleich zu monolithischen Architekturen deutlich höher ist.

Glossar

Begriff	Bedeutung
API-Gateway	Zentrale Anlaufstelle für externe Anfragen an ein Microservice-System.
Cluster	Gruppe aus mehreren Servern, die gemeinsam eine verteilte Umgebung bilden.
Container	Leichtgewichtige Umgebung zur isolierten Ausführung von Software-Komponenten.
DDD	Domain-Driven Design – domänenzentrierter Ansatz zur Softwaremodellierung.

Begriff	Bedeutung
Docker	Plattform zur Containerisierung und zum Deployment verteilter Anwendungen.
Kubernetes	System zur automatisierten Verwaltung, Skalierung und Orchestrierung von Containern.
Monolith	Architektur, bei der die gesamte Anwendung als eine Einheit betrieben wird.
Self-contained System	Architekturansatz, bei dem jeder Dienst auch UI, Logik und Persistenz umfasst.
Service Discovery	Verfahren, mit dem Microservices einander automatisch auffinden können.

Revision #3
Created 13 May 2025 08:58:47 by Marius Klein
Updated 14 May 2025 00:24:03 by Marius Klein