MASSGESCHNEIDERTE DIGITALISIERUNG DURCH MICROSERVICES

Modularisierung und Modernisierung für skalierbare und wartbare Softwaresysteme

Einsatzkriterien

Wann und warum bieten sich Microservice-Architekturen an?

Skalierbarkeit und Elastizität › Ausgewählte Microservices können unabhängig repliziert und migriert werden.

Wartbarkeit › Einzelne Microservices können weitgehend unabhängig ersetzt und weiterentwickelt werden.

Agilität und Time-to-Market › Einzelne Microservices können weitgehend unabhängig und somit häufig ausgeliefert und installiert werden.

Robustheit › Bei Ausfall einzelner Microservices sollten andere Microservices weiterhin Dienste anbieten können.

Erfolgsfaktoren

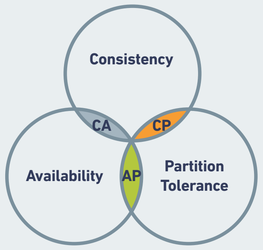
- Voraussetzung für den Erfolg ist ein guter, stabiler Modulschnitt für Microservices, die dann als verteilt installierbare Komponenten implementiert werden.
- Domänenschnitt und Bounded Context müssen einen vertikalen Schnitt für Self-Contained Systems liefern (Domain-Driven Design)
- Unabhängiges Deployment einzelner Microservices ist essentiell um DevOps zu ermöglichen.
- Die Teamstruktur muss der Microservice-Struktur entsprechen (Conway's Law).
- Gemeinsamer Code zwischen Microservices nur als Open Source Software.

Microservice-Architekturen

Architekturprinzipien

Microservices, werden als verteilt installierbare Komponenten implementiert.

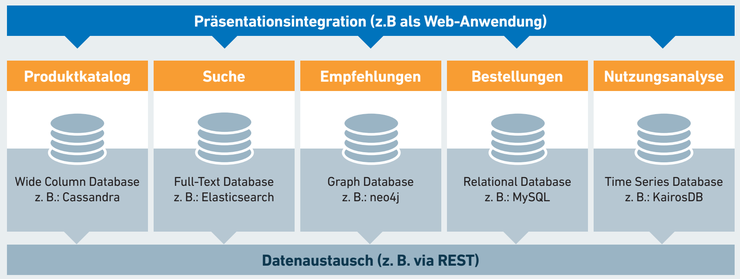
- In verteilten Systemen müssen die Schnittstellen schmal und möglichst stabil sein (Consumer-Driven Contracts) → guter Modulschnitt ist essentiell
- Als vorteilhafter Effekt ergibt sich eine gute Modul-Kapselung (Information Hiding)
  - Erleichtert das Verstehen einzelner Module
  - Falls der Modulschnitt nicht gut gelingt, wird jedoch das Verstehen des Gesamtsystems erschwert.
- Damit die Microservices unabhängig entwickelt und installiert werden können, erhalten diese jeweils eine eigene Datenhaltung und eine eigene Nutzungsschnittelle (GUI und/oder API) → Self-Contained Systems mit polyglotter Persistenz und Programmierung
- Abhängigkeiten zwischen Microservices müssen klein sein
  - Keine gemeinsame Datenbank
  - Kein geteilter Programmcode
    - Nur geteilte Schnittstellen
    - Nur Wiederverwendung als Open Source Software
- Nur asynchrone Kommunikation
  → Eventual Consistency, CAP-Theorem



Polyglot Persistence

Microservices nutzen Polyglot Persistence:

- Jeder Microservice verwaltet seine eigenen Daten um unabhängig deploybar und skalierbar zu sein.
- Eine Konsequenz ist „eventual consistency" der Daten



Beispiel für eine vertikale Aufteilung in Microservices

Container- und Cluster-Technologien für Microservices

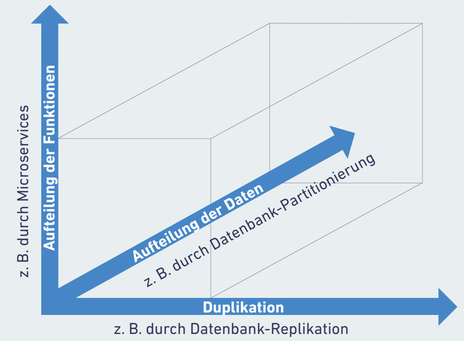
- Container ermöglichen das leichtgewichtige Deployment von Microservices → z.B. Docker oder RedHat Ansible Tower
- Cluster ermöglichen die Verteilung, Load Balancing und Skalierung u.a. in der Cloud → z.B. Kubernetes oder Apache Mesos

Modernisierung

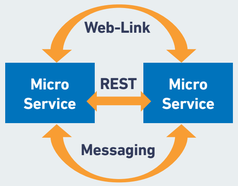
- Migration vom Monolithen zur verteilten Microservice-Architektur:
  - Existierende Anwendung „aufbrechen" in mehrere Microservices oder
  - direkte Neuentwicklung mit mehreren Microservices.
- In beiden Fällen ist eine vertikale Dekomposition nach Funktionen, keine horizontale Schichtenarchitektur, sinnvoll.
- Die einzelnen Microservices können dann unabhängig weiterentwickelt und ggf. modernisiert werden.

Skalierbarkeit

Dimensionen der Skalierung



Integration auf zwei Ebenen



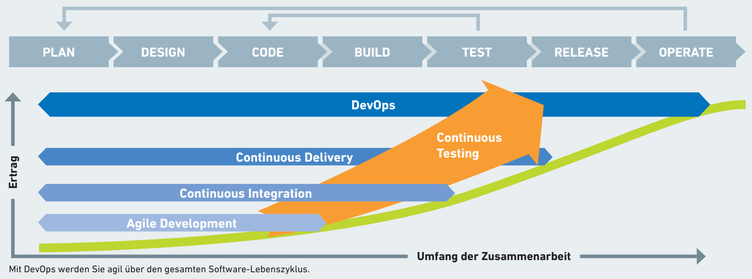
1. **Integration mehrerer Microservices in eine gemeinsame Nutzungsschnittstelle (UI):**
   - Jeder Microservice bringt „seine" UI mit.
   - Die konkrete Integration kann beispielsweise als Single-Page-Website, mittels Server- oder Client-seitigem Templating, asynchron mit Ajax, sowie mittels getrennten UIs erfolgen.
2. **Datenintegration ist transaktionslos**
   - Bevorzugt asynchrone Kommunikation via REST und/oder Messaging-Middleware, wie Kafka oder RabbitMQ.
   - „Eventual Consistency" ist eine Konsequenz, so dass verteilt gespeicherte Daten (zeitweise) inkonsistent sein können.

Qualitätssicherung

Microservices werden unabhängig deployed

- Dazu werden unabhängige Deployment Pipelines benötigt
  - Hier dienen Quality Gates im Continuous Integration zur Qualitätssicherung
  - Diese Deployment Pipelines sind auch Software-Komponenten (idealerweise auch Microservices), die qualitätsgesichert werden müssen.
- Testing auch zur Laufzeit, z.B. durch Monitoring und Chaos-Monkeys

Continuous Testing



Mit DevOps werden Sie agil über den gesamten Software-Lebenszyklus.

Vorteile

Fachlicher Modulschnitt:
- Durch stabile und schmale Modul-Schnittstellen können Teams weitgehend unabhängig arbeiten und „ihre" Microservices weiterentwickeln.

Unabhängiges Deployment:
- Ermöglicht DevOps für schnelles Time-To-Market, unabhängige Skalierung, Eingrenzung von Fehler-Propagation.

Technologische Vielfalt:
- Für jeden Service kann die jeweils optimale Technologie genutzt werden.

Nachteile

Fachlicher Modulschnitt:
- Es gelingt nicht immer, stabile und schmale Modul-Schnittstellen zu finden.

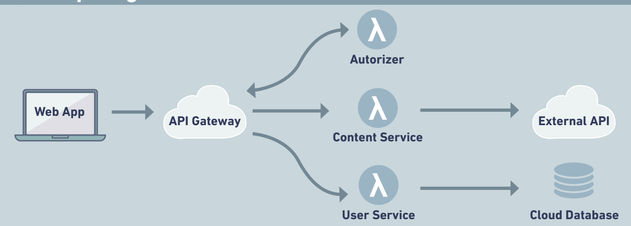
Unabhängiges Deployment:
- Verteilte Systeme sind inhärent schwierig zu entwickeln, da Netzwerk-kommunikation fehleranfällig ist und verteilte Transaktionsverarbeitung ineffizient ist.

Technologische Vielfalt:
- Entwicklung und Betrieb müssen diverse Technologien beherrschen.

Trends: Serverless Computing

Im Serverless Computing werden zustandslose Funktionen bereitgestellt und durch Ereignisse aufgerufen.

- Nur durchgeführte Funktionsaufrufe werden abgerechnet, statt bereitgestellter Server.
- Im Gegensatz zu Microservices erfolgt die Skalierung automatisch durch die Cloud-Plattform.
- Bei Microservices werden Cluster-Technologien wie Kubernetes oder Mesos durch die Entwickler eingesetzt, im Serverless Computing erfolgt dies durch den Cloud-Provider.
- Beispiel-Plattformen: Amazon Lambda, Google Cloud Functions, Microsoft Azure Functions, IBM OpenWhisk



Inhaltliche Entwicklung: Prof. Dr. W. Hasselbring, Software Engineering Group Dept. Computer Science, Kiel University