embarc Software Consulting GmbH
SIGS DATACOM FACHINFORMATIONEN FÜR IT-PROFESSIONALS

architektur SPICKER

Übersichten für die konzeptionelle Seite der Softwareentwicklung

MEHR WISSEN IN KOMPAKTER FORM:
Weitere Architektur-Spicker gibt es als kostenfreies PDF unter www.architektur-spicker.de

NR. 3

Microservices

Microservices versprechen schnelle Reaktionsfähigkeit in Entwicklung und Betrieb. Dieser Spicker bietet einen fundierten Einblick in den Architekturstil.

IN DIESER AUSGABE

Worum geht's? (Herausforderungen)

- Die Entscheidung Richtung Microservices ist gefallen oder steht kurz bevor. Welche organisatorischen und technischen Aspekte sollten Ihnen klar sein?
- Eine monolithische Applikation soll in Microservices überführt werden. Wie gehen Sie vor?
- Internet-Schwergewichte wie Netflix leben den Stil vor und stellen Ihre Lösungen bereit. Sollten Sie sich bedienen?

Was sind Microservices?

Microservices sind unabhängige Prozesse, die auch wenn separat deployt für den Anwender wie eine Applikation erscheinen.

Ein erfolgreicher Einsatz erfordert Services, die über einheitliche Kommunikationsmechanismen lose gekoppelt sind. Für Verteilung und Monitoring sind übergeordnete Integrationsmechanismen zwischen den Microservices erforderlich.

Dafür können Sie auf Veränderungen in der Umgebung rasch reagieren, z. B. durch:

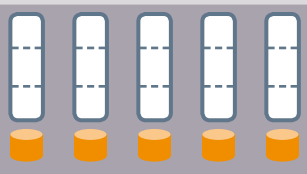
- Funktionale Erweiterungen
- Austausch qualitativ ungenügender Teile
- Skalierung einzelner Bereiche/Services
- Einsatz unterschiedlichster Technologien

Für unterschiedliche Definitionen des Microservices-Begriffes siehe „Weitere Informationen“ auf Seite 4.

Gegenüberstellung: Schichten vs. Microservices vs. SOA

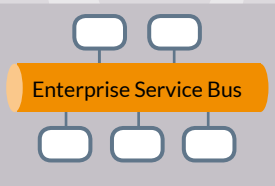
„Klassische“ Schichtenarchitektur
(auf Ebene eines Systems)

+ Hohe Konsistenz in querschnittlichen (oft technischen) Aspekten
+ Gute Portierbarkeit, Schichten leicht auszutauschen
+ Wiederverwendbarkeit von fachlichen Komponenten



Microservices:
Zerlegung in Vertikalen

+ Fördert Domänenorientierung
+ Funktionale Blöcke leicht auszutauschen und zu ergänzen
+ Technische Schulden besser kontrollierbar
+ Separate Skalierung, unabhängiges Deployment



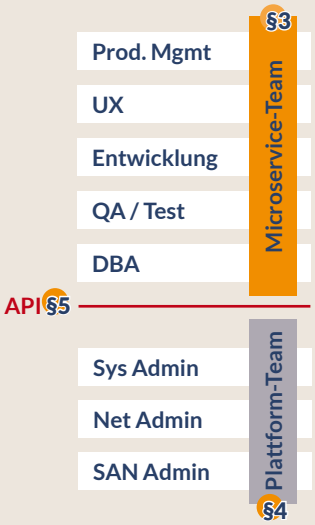
Service-orientierte Architektur
(auf Ebene einer Systemlandschaft)

+ Hohe Wiederverwendung durch unternehmensweites Teilen von Services
+ Abbildung komplexer Geschäftsprozesse durch Service-Orchestrierung
+ Geringe Kosten für neue Services durch gemeinsame Kommunikationsinfrastruktur (ESB)

Faustformel: Microservices = SOA – ESB

Microservice Prinzipien

Die folgenden Prinzipien stellen Sichtweisen und Regeln dar, die viele Microservice-Implementierungen für sich annehmen. Ganz im Sinne von Conway's Law lässt sich die technische Komponente des Architekturstils dabei nicht isoliert betrachten. Im Folgenden deshalb einige organisatorische Prinzipien (Teamaufstellung, Verantwortung), methodische Prinzipien (Vorgehen, Praktiken) und technische Prinzipien (Umsetzungsregeln).



Organisatorische Prinzipien

§1 Hohe Vertrauenskultur: Freiheit und Verantwortung für Entwickler um schnelle lokale Reaktion und Identifikation mit der Lösung zu fördern.

§2 Realitätsnahe Entwicklung: Die Lücke zwischen Entwicklungs- und Produktionsumgebung ist so klein wie möglich. Das betrifft Zeit, Personal, Werkzeuge und Technologien.

§3 Keine Disziplinensilos: Geschäft, Entwicklung und Betrieb sind in cross-funktionalen Teams rund um Domänen organisiert und für ihren Service und dessen Qualität verantwortlich.

§4 Lernende Plattform-Teams: Separate Teams verantworten grundlegende Plattformen und Technologien. Das fördert Einheitlichkeit und schnelles Lernen.

§5 Plattform ist Selbstbedienung: Deployment-Werkzeuge, Infrastruktur-Technologien, Datenbanken, Container etc. werden über APIs bereitgestellt und sind ohne Involvierung von Plattform-Teams nutzbar.

§6 Plattform ist ein Angebot: Externe Lösungen dürfen die Qualitätseigenschaften der Plattform nicht verschlechtern.

Methodische Prinzipien

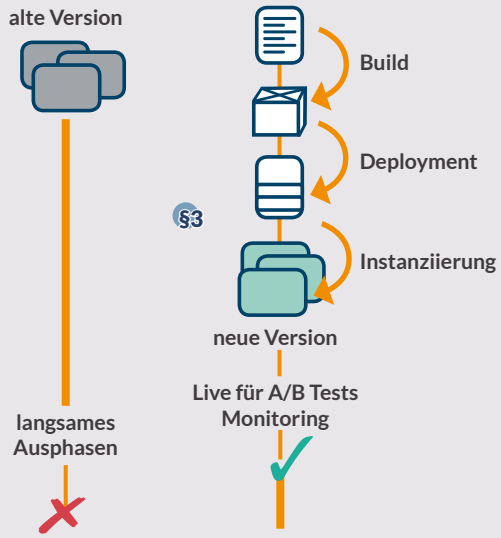
§1 Wenig technische Standardisierung: Technische Standardisierung erfolgt nur für die Kommunikationsverbindungen und von Container-Ebene abwärts. Innerhalb von Services herrscht Technologiefreiheit.

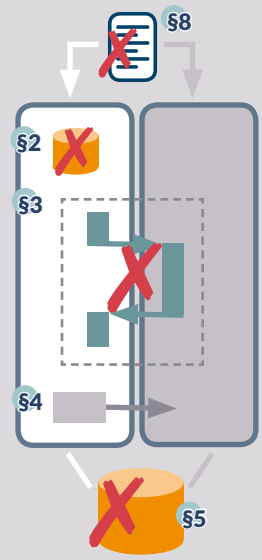
§2 Schnelles Feedback: Deployments finden oft und mit möglichst kleinen Software-Einheiten statt. Folge sind weniger Fehler pro Deployment und eine einfachere Problem-Lokalisierung.

§3 Evolutionäre Produktionsumgebung: Neu ausgelieferte Versionen haben keine Auswirkungen auf die Produktion bis Last auf sie geschaltet wird. Existierende Services sind von Updates nicht betroffen. A/B Tests, Feature Flags, Version Routing!

§4 Wenig übergreifende Tests: Funktionale Tests, die Service-übergreifend sind, sollten so gut wie möglich vermieden werden, da sie Abhängigkeiten manifestieren, schwerfällig und komplex sind.

§5 Keine Meetings: Es gibt keine geplante Zeit für Übergaben oder ausgedehnte Abstimmungen zwischen Teams. Keine personellen Flaschenhälse, dafür Monitoring, Fehlererkennung und schnelle Rollbacks.





Technische Prinzipien

§1 Überschaubare Größe: Ein Microservice beinhaltet eine einzelne Funktionalität, die in ihrem eigenen Build läuft, für einen Entwickler gut durchschaubar ist und in vertretbarem Rahmen ersetzbar ist.

§2 Zustandsloses Geschäft: Services, die Geschäftslogik abbilden, sind als zustandsloser Prozess abgebildet (shared-nothing). Keine Annahmen bezüglich Caching oder Ressourcen-Verfügbarkeit!

§3 Nicht-Blockierende Aufrufe: Alle Aufrufe zwischen Services sind multi-threaded und nicht-blockierend (Verwendung von Eventmechanismen, Reactive Extensions, Netty mit non-blocking I/O etc.).

§4 Client-seitige Treiber: Statt definierter Protokolle und Abhängigkeiten von fremden Schnittstellen werden bevorzugt Client Libraries verwendet. Diese beinhalten keine Geschäftsobjekte des aufgerufenen Services.

§5 Keine Datenkopplung: Es gibt keine zentrale Datenbank, kein kanonisches Datenmodell, oder zentralisierte Message Queues. Denormalisierte Daten werden auf Applikationsebene zusammengeführt.

§6 Optimierte Kommunikation: Zwischen Services wird auf Latenzzeiten und Effizienz optimiert. Protocol Buffer oder binäre Protokolle (z.B. SBE) sind besser lesbaren, flexibleren Formaten im Zweifel vorzuziehen.

§7 Container als Ops-Grenze: Fachliche Teams arbeiten bis auf Container-Ebene (z.B. Docker). Darunter greift die automatisierte Instanziierung und der standardisierte Betrieb des Containers.

§8 Ein Repo, ein Service: Pro Microservice gibt es genau eine Codebasis, bzw. ein Code-Repository. Es gibt kein Code-sharing, mehrere Repositories pro Service sind ein Hinweis auf zu große Services.

Vom Monolithen zu Microservices

Es gibt zwei Strategien zum Entwickeln einer Microservice-Applikation:
- Ausgehend von einer existierenden Applikation
- Start auf der „grünen Wiese“ mit einer neuen Anwendung

Da Sie in vielen Fällen eine bestehende Anwendung nicht ignorieren können , betrachten wir hier den ersten Fall.

Ausgangspunkt ist eine monolithische Applikation
(Für UI-Integration siehe letzte Seite)

Häufig greift die Applikation auf eine monolithische DB zu (siehe extra Kasten unten)

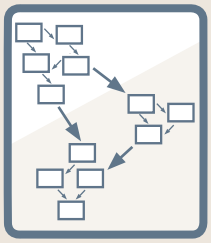
Abhängigkeiten identifizieren
Wichtige Faktoren für Ihre Analyse:
- gemeinsam genutzte Daten
- Aufrufabhängigkeiten
- Prozessgrenzen
- gemeinsam genutzte Fremdsysteme

Die fachliche Aufteilung gibt den Schnitt vor und ist damit extrem wichtig.

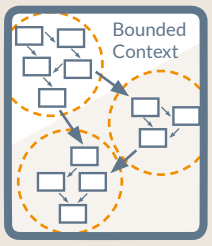
Domain Driven Design gibt Orientierung :
- Ein *Bounded Context* umfasst die Fachlichkeit eines Microservices.
- Investieren Sie besonders in Ihre *Core Domain*
- Nutzen Sie *Shared Kernel* als Startpunkt für klar definierte, gemeinsam genutzte Funktionalitäten (aber nicht als gemeinsame Ressource)
- Betrachten Sie fachliche Prozesse, da diese die Transaktionen leiten

Warnsignale für einen zu kleinen Schnitt:
1. Viele Aufrufe zwischen den Services (führen zu Performance Overhead)
2. Probleme beim Transaktionsschnitt (führen zu inkonsistenten Daten oder komplexen Fehlerkorrekturmechanismen)

Machen Sie sich neben der Aufteilung frühzeitig Gedanken, wie übergreifende Funktionalitäten (z.B. Reporting) umgesetzt werden:
- *Als zusätzlicher Microservice (Pull-Ansatz)?*
- *Als Teil jedes Microservices (Push-Ansatz: Koordination nötig)?*

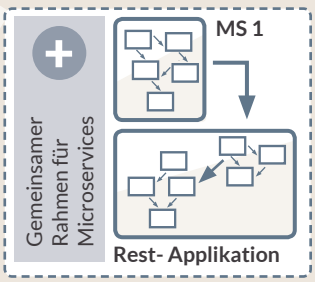


Microservice-Kandidaten herausbilden



Identifikation von Bounded Contexts als erste Microservices:
Wo haben Sie ...
...neue Anforderungen?
...besondere Qualitätsanforderungen?
...den Wunsch nach einer anderen Technologie?
...schon ein Team für einen Bereich gefunden?
...schon eine getrennte Datenbank?

Den ersten Microservice abtrennen (MS 1)



Wiederholung der Schritte für weitere Microservice-Kandidaten

Auftrennung der monolithischen Datenbank ermöglicht unabhängiges Deployment und separate Prozesse:

1. Initialzustand	2. Logische Trennung	3. Physische Trennung
		
evtl schon vorhanden: Schema oder User für fachliche Bereiche	Abhängigkeiten der Microservices maßgeblich	Synchronisierung gemeinsamer Daten via: • Replikation von Daten (Push oder Pull) • Aufruf anderer Microservices

Die fachliche Aufteilung führt von klassischen **ACID**-Transaktionen zu **BASE**-Transaktionen. Das erfordert ein fachliches Umdenken, intensivere Systemüberwachung und neue Fehlerkorrekturmechanismen.

Je mehr Funktionalität Sie als Microservices migrieren, umso wichtiger werden weitere Aspekte

Eine automatisierte Delivery Pipeline erleichtert den Umgang mit der komplexeren Microservice-Architektur.

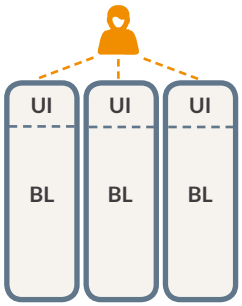
1. Die MS-Teams definieren die Service Contracts für Aufrufe zwischen den Microservices.
2. *Versioning* von Schnittstellen/APIs:
   - *Breaking Changes* vermeiden
   - *(Versions-) Koexistenz* ermöglichen
   - *Semantic Versioning* nutzen
3. *Consumer Driven Contract Testing* verringert den Bedarf von übergreifenden Tests.
   Übergreifende Tests sollten auf wenige sog. *Journeys* fokussieren.
4. Resilience-Features helfen bei Nicht-Verfügbarkeit anderer Microservices.
   Monitoring deckt Fehler auf.

Strukturierungsoptionen und Technologien für das UI

Anwender nutzen Ihre Microservice-Applikation über ein User Interface (UI).
Wie ist dieses strukturiert? Welche Technologien können Sie verwenden?

Jeweils eigenes UI

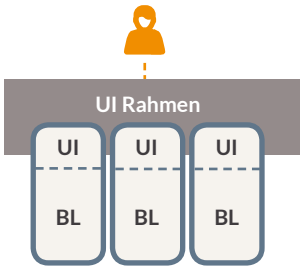
Jeder Microservice bringt sein komplett eigenes UI mit. Die Integration zwischen den UIs erfolgt innerhalb des Browsers über Links.



z.B. klassische HTML-basierte Webapplikationen à la amazon.de, Web-MVC-Framework (Request/Response) garniert mit JavaScript (etwa Spring Web MVC oder PHP Micro Framework)

Plugin-Ansatz

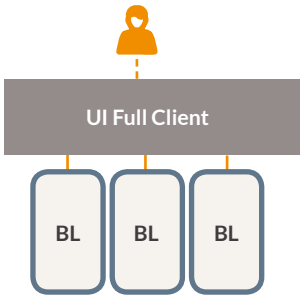
Microservices integrieren sich über eigene UI-Anteile in ein übergeordnetes UI. Im einfachsten Fall enthält dieses nur die Hauptnavigation.



z.B. Desktop-Applikation à la Spotify, Portalserver, anderweitig server- oder client-seitig eingebettete HTML-Fragmente (etwa mit Server Side Includes oder als Single Page Application z. B. mit AngularJS).

Full Client

Ein gemeinsames UI nutzt alle Microservices über deren Schnittstellen. Diese sind selbst UI-los und enthalten nur Business-Logik (BL).



z.B. Mobile-App à la YouTube für Smartphone und Tablet, nativ entwickelt für Zielsysteme wie iOS, Android etc. oder hybrid erstellt, etwa mit PhoneGap

Die Strukturierungsoptionen haben Stärken und Schwächen

*Wir diskutieren die Optionen anhand **typischer Szenarien** und schätzen ab wie leicht sich diese tendenziell umsetzen lassen.*	Wie leicht umzusetzen? Mit ... Jeweils eigenes UI	Plugin-Ansatz	Full Client
• Ein Team entwickelt und veröffentlicht einen neuen Microservice. Dieser lässt sich ohne Änderungen am bestehenden UI nutzen.	Leicht	Mittel	Schwierig
• Ein Team experimentiert für einen Service mit einer neuen Frontendbibliothek. Es bringt den Service damit reibungslos in Produktion.	Leicht	Schwierig	Schwierig
• Ein Benutzer verwendet Funktionalität aus unterschiedlichen Microservices. Die Applikation erscheint ihm „aus einem Guß“.	Mittel	Leicht	Leicht
• Das Marketing des Unternehmens veröffentlicht ein neues CI. Die Teams übernehmen es zügig in die komplette Applikation.	Schwierig	Mittel	Leicht
• Benutzer wollen auf die Applikation über einen neuen Client zugreifen (z.B. Smart-TV). Die Applikation mit all ihren Services ist fix angebunden und nutzt den Client optimal aus.	Schwierig	Mittel	Leicht

Weitere Informationen

Bücher (Auswahl)

- Sam Newman: „Microservices: Konzeption und Design“, mitp-Verlag 2015
- Eberhard Wolff: „Microservices: Grundlagen flexibler Softwarearchitekturen“, dpunkt.verlag 2015
- Uwe Friedrichsen, Stefan Toth, Eberhard Wolff: „Resilience: Wie Netflix sein System schützt“, entwickler.press 2015

Blog-Beiträge

- James Lewis, Martin Fowler: „Microservices - a definition of this new architectural term“ http://martinfowler.com/articles/microservices.html
- Tobias Flohre: „Kaffeeküchengespräche – Microservices und Domain Driven Design“ https://blog.codecentric.de/2015/06/kaffeekuchengesprache-microservices-und-domain-driven-design/
- embarc-Blog: „Netflix-Architektur“-Serie von Stefan Toth und mehr http://www.embarc.de/tag/microservices/

Wir freuen uns auf Ihr Feedback:
spicker@embarc.de

embarc Software Consulting GmbH

http://www.embarc.de
info@embarc.de

SIGS DATACOM
FACHINFORMATIONEN FÜR IT-PROFESSIONALS

http://www.sigs-datacom.de
info@sigs-datacom.de