Fachhochschule
Dortmund
University of Applied Sciences and Arts

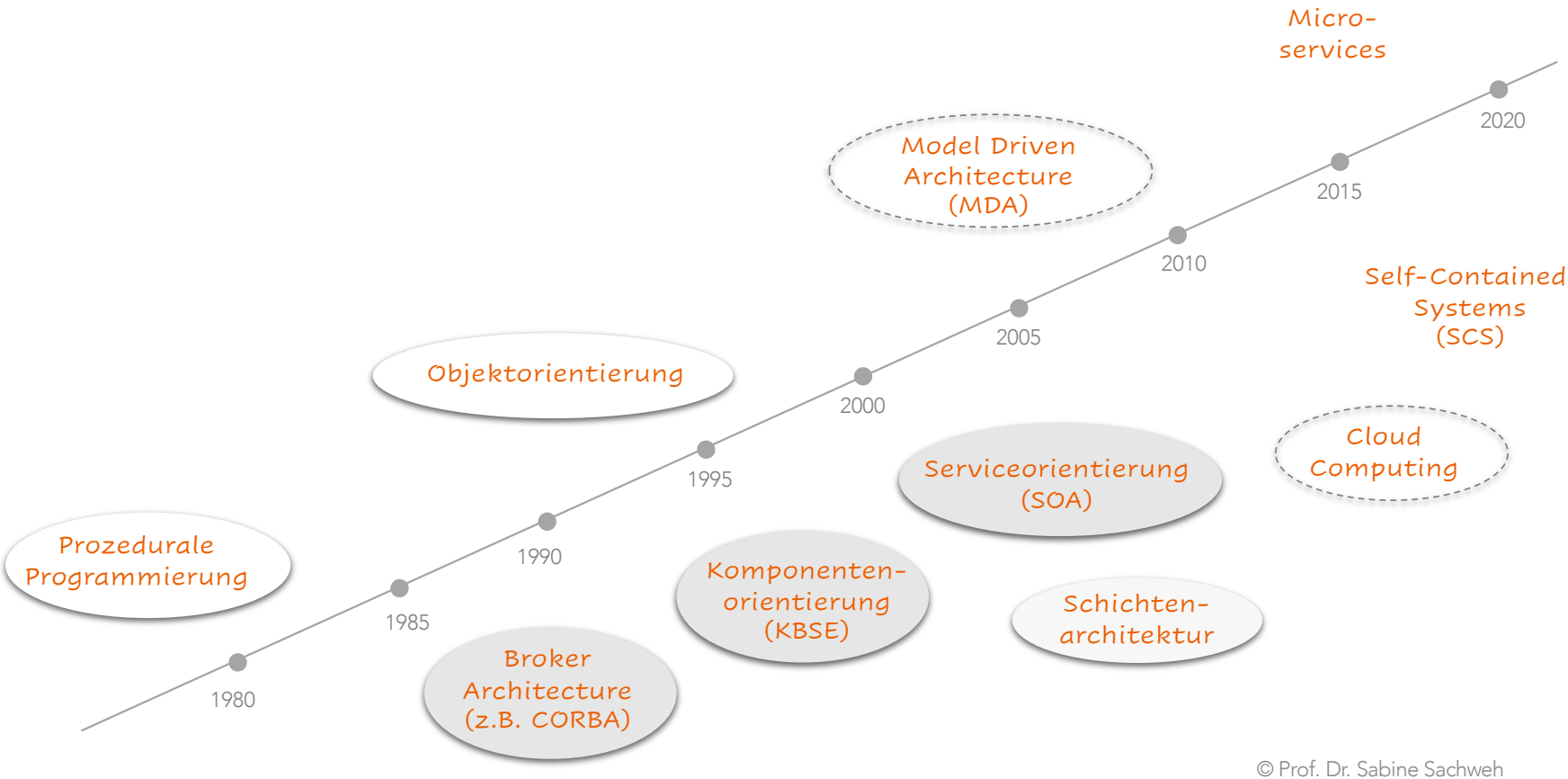
we
focus
on
students

Softwaretechnik 2

Architekturstile III

Entwicklungs- und Architekturtrends im Zeitverlauf

Unvollständige Auswahl großer Softwaretechnik-Trends



Fachhochschule
Dortmund
University of Applied Sciences and Arts

Architekturstile III

Microservice Architecture (MSA)

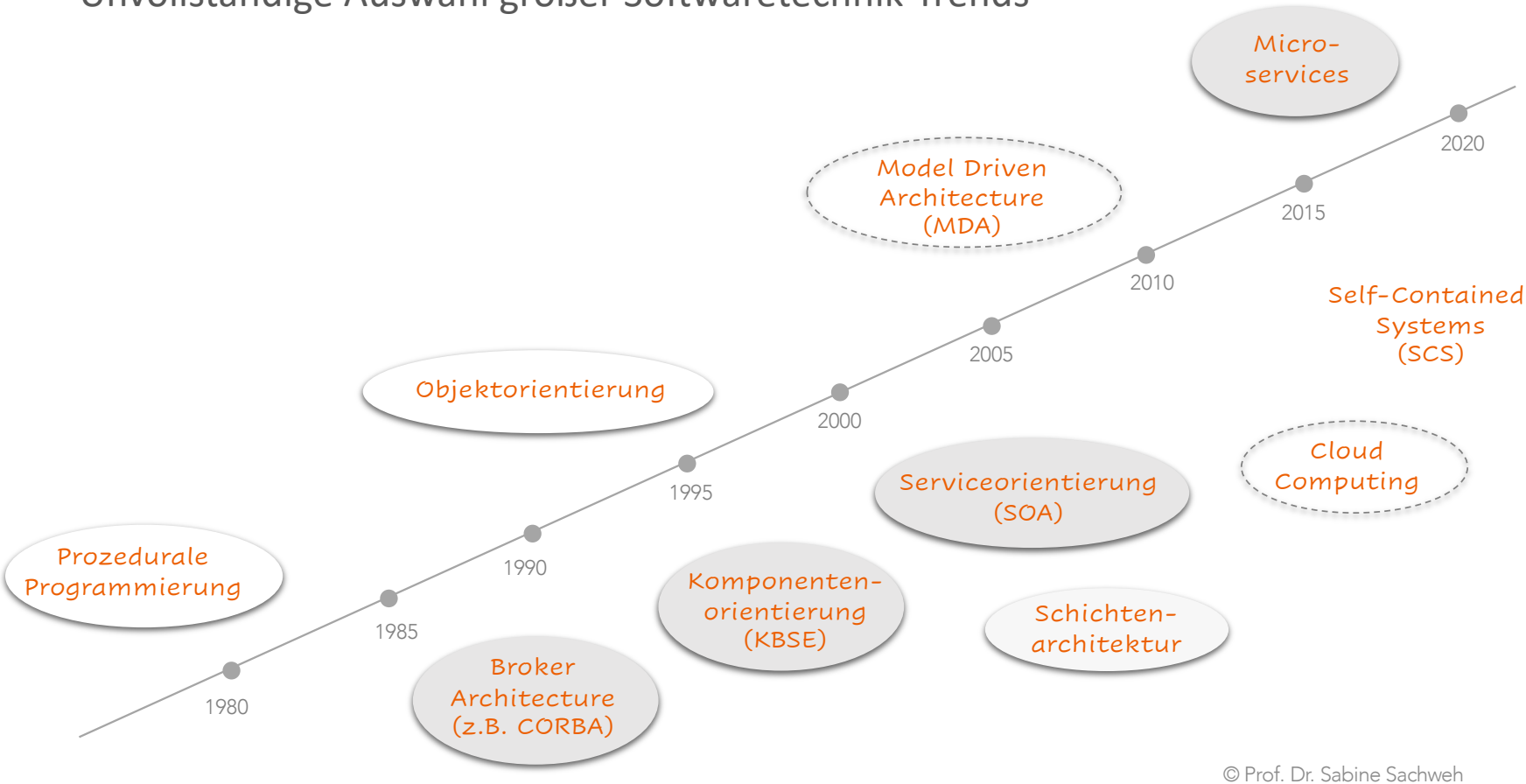
Self-contained Systems (SCS)

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Microservice Architecture (MSA)

Entwicklungs- und Architekturtrends im Zeitverlauf

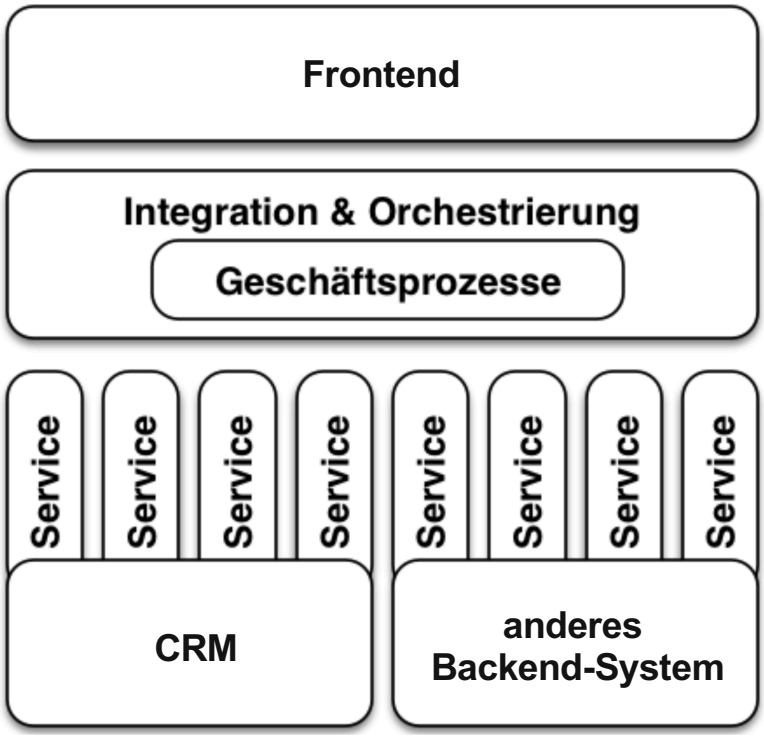
Unvollständige Auswahl großer Softwaretechnik-Trends



SOA

Rückblick

- In einer SOA-Landschaft **jede Anwendung in Services** aufgeteilt, auch solche, die schon vor Einführung der SOA existierten.
- Beispielsweise könnte ein **Customer Relationship Management** (CRM) Dienste zum **Anlegen von Kunden**, zum **Abfragen von Informationen** über Kunden oder zum **Anlegen neuer Interaktionen** mit einem Kunden anbieten.
- Das CRM ist ein Backend-System und damit eine **Deployment-Einheit**, da alle Services nur gemeinsam in Produktion gebracht werden können.



[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

SOA

Rückblick

- Die **Kommunikation der Services** muss eine **einheitliche Technologie** nutzen, über die alle Services erreichbar sind.
  - asynchron: beispielsweise ESBs (Enterprise Service Bus)
  - synchron: beispielsweise SOAP, das u.a. die Kommunikation über HTTP erlaubt

- Die **Integration und Orchestrierung** wird üblicherweise in **einer eigenen Schicht** umgesetzt. Sie implementiert Geschäftsprozesse mit Hilfe der Services.
  ⇒ Wenn ein neuer Service in den Prozess integriert werden soll, ist dazu nur eine Änderung des Prozesses notwendig. Die Services bleiben unverändert.

- Verschiedenste Frontends können die Orchestrierung und die einzelnen Services nutzen, so dass schnell dedizierte Frontends angeboten werden können.

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

SOA

Vorteile

Vorteile (u.a.) 

- Services können in unterschiedlichsten **Geschäftsprozessen** und/oder unterschiedlichsten **Frontends wiederverwendet** werden.

- **Änderungen** an den **Geschäftsprozessen** sind möglich, ohne dass dazu die Services geändert werden müssen. Es muss lediglich die Orchestrierung angepasst werden.
  → **Flexibilität + Ressourcenschonung**

Herausforderungen (u.a.) 

- Allerdings ist eine SOA mit einem **erheblichen Aufwand** und **erheblichen Investitionen** verbunden:
  - Alle IT-Systeme müssen **in Services aufgeteilt werden**, die im Netzwerk ansprechbar sind.
  - Alle **Geschäftsprozesse in der Orchestrierung-Schicht** umgesetzt werden.
  - **Benutzungsschnittstellen müssen neu gestaltet** werden – üblicherweise als Portal.
- Die intendierte Flexibilität kann nur erzielt werden, wenn notwendige **Änderungen** in den Services, der **Orchestrierung oder im Portal isoliert** werden können.

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

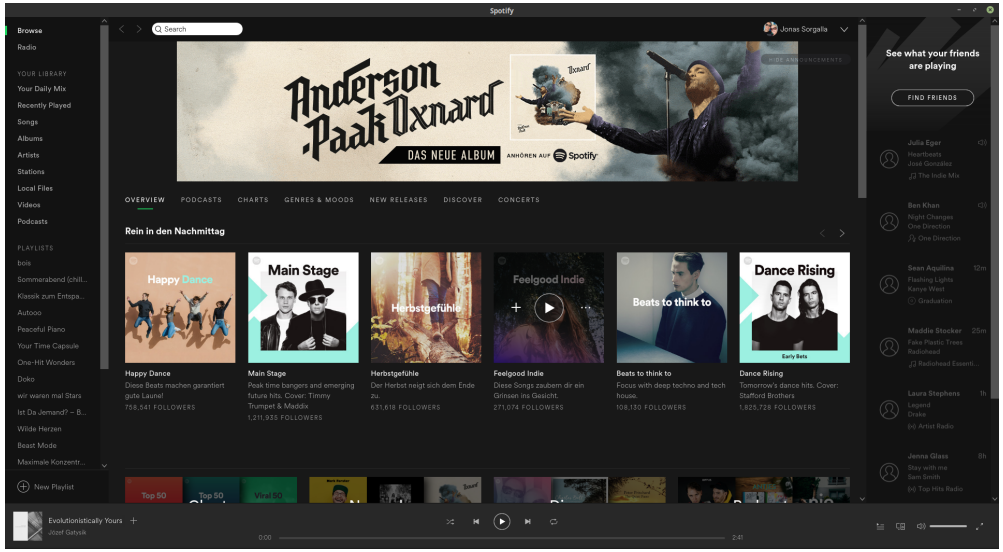
SOA

Lessons learned

- Auf der Ebene einer unternehmensweiten IT bietet sich eigentlich immer zumindest eine **Aufteilung in Services** an.
- Dazu ist eine **einheitliche Kommunikationstechnologie** notwendig.
- Es ist aber fraglich, ob eine **vollständige SOA** mit einer **Aufteilung in eine getrennte Orchestrierung und Portal** sinnvoll ist, denn das ist sehr aufwändig und bietet kaum einen Mehrwert.
- Anforderungen **moderner internetbasierter Dienstleistungen** wie beispielsweise Streaming wird dieser Architekturstil vor allem in Bereichen wie **Skalierung**, **Robustheit**, oder **Entwicklungsgeschwindigkeit** nicht gerecht

Beispiel Spotify

Streaming-Dienstleister heben Skalierung auf ein anderes Niveau



- 75+ Millionen aktive Nutzer jeden Monat
- 58 Länder
- 20.000 Neue Songs täglich
- 2+ Milliarden Playlisten
- „Saisonales Nutzerverhalten“

Traditionelle Architekturstile stoßen hier an ihre Grenzen!

Microservices

SOA is dead long live services – Services reloaded

Kernidee

Monolithen werden in unabhängige, separate Prozesse zerschnitten!

⇒ **„Microservices“**

(So benannt seit ca. 2011/2012)

"Microservices are **small, autonomous** services that **work together**."

[S. Newman, Building Microservices, M. Loukides und B. MacDonald, Hrsg. O'Reilly Media, 2015.]

Microservice: Ursprung

- der Begriff "microservice" wurde im Mai 2011 auf einem Architekten-Workshop in der Nähe von Venedig diskutiert
- um zu beschreiben, was aus Sicht der Teilnehmer einen üblichen Architekturstil beschrieb, den viele von ihnen kürzlich untersucht hatten
- in May 2012 entschied sich die gleiche Gruppe für den Begriff „Microservices“ als angemessene Bezeichnung
- **James Lewis** von ThoughtWorks präsentierte einige Ideen dazu auf der 33rd Degree, einer Java Konferenz, in Krakow in Polen „Microservices – Java, the Unix Way“
  [http://2012.33degree.org/pdf/JamesLewisMicroServices.pdf]
- **Fred George** (ThoughtWorks) und **Adrian Cockcroft** damals Cloud-Architekt bei Netflix (mit seinem Ansatz:"fine grained SOA“) und **Joe Walnes, Dan North, Evan Botcher** und **Graham Tackley** propagierten parallel diesen neuen Architekturstil

Begriffsdefinition

James Lewis

Martin Folwer

"In short, the microservice architectural style is an approach to developing a single application as a **suite of small services**, each running **in its own process** and communicating with **lightweight** mechanisms, often an HTTP resource API. These services are built around **business capabilities** and **independently deployable** by fully automated deployment machinery."[2)]

- James Lewis & Martin Fowler

[J. Lewis und M. Fowler, "Microservices", https://martinfowler.com/articles/microservices.html, 2014.]

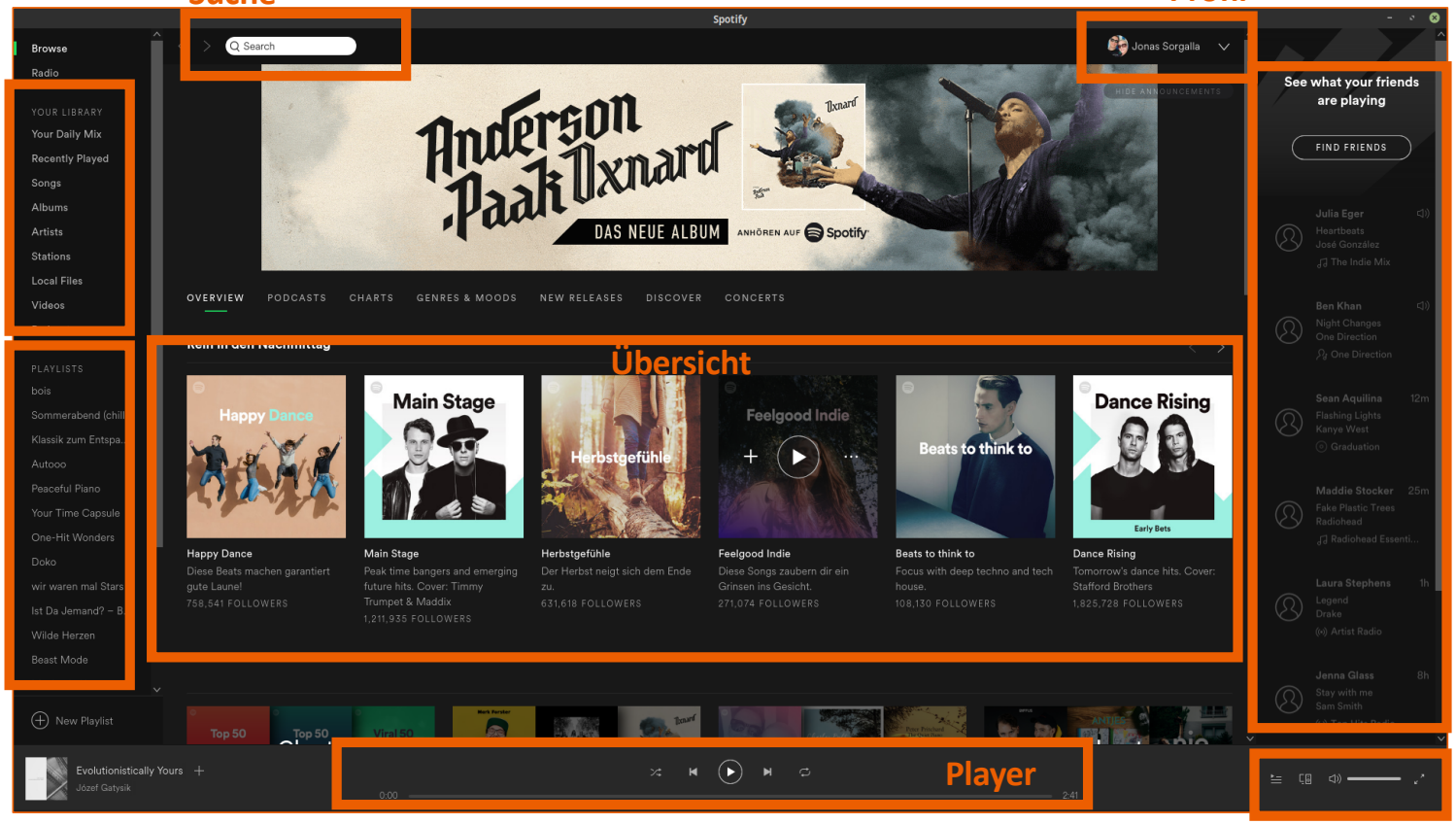
Begriffsdefinition

- dementsprechend sind Microservices definiert ...

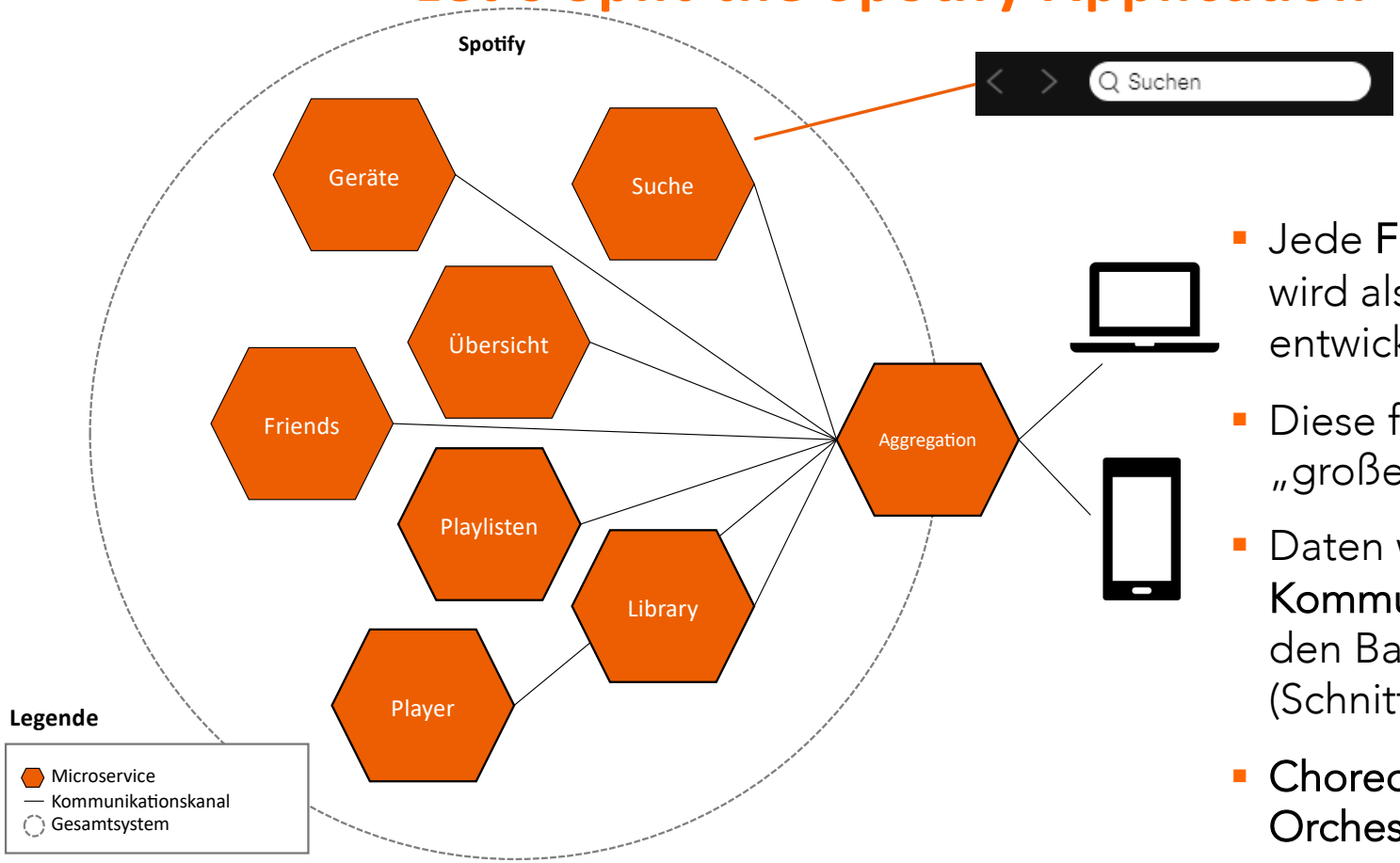
  … als Ansatz eine einzelne Anwendung als **Menge von kleinen Services** zu entwickeln, von denen jeder
  - in **seinem eigenen Prozess** läuft und
  - **leichtgewichtig** kommuniziert, häufig über eine HTTP Ressource API
  - diese Services werden ausgerichtet auf **Geschäftsfunktionen** entwickelt
  - sie können **unabhängig depoyed** werden durch ein voll automatisiertes Deployment

Der Gedanke dahinter entspricht weitgehend dem der Unix-Philosophie:
(„**Do One Thing and Do It Well**“)

Let‘s Split the Spotify Application



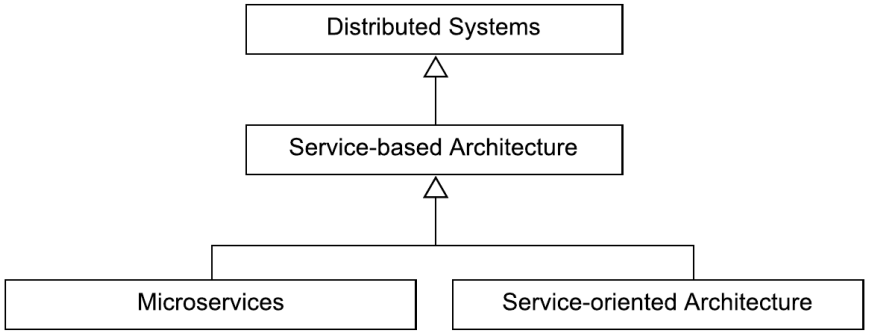
Let‘s Split the Spotify Application



- Jede **Funktion** bzw. **jeder Service** wird als **eigenständige Software** entwickelt → Deployment
- Diese fungieren als **Bausteine** des „großen Ganzen“ bzw. des Systems
- Daten werden über **fest vereinbarte Kommunikationskanäle** zwischen den Bausteinen ausgetauscht (Schnittstellen)
- **Choreographie anstatt Orchestrierung**

Microservices

Weitere Abgrenzung zur SOA



„Perhaps, Microservice are SOA done right."

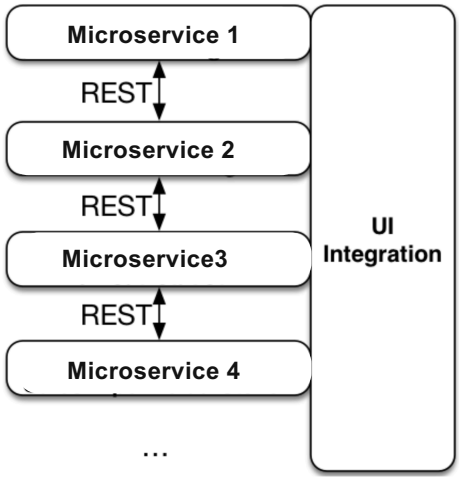
- Microservices und SOA haben das **Ziel die Entwicklung von Software zu flexibilisieren**, allerdings auf sehr verschiedene Art.
- Microservices setzen auf einer **anderen Ebene** als SOA an.
  - Microservices dienen zur **Strukturierung einer Anwendung**, während
  - SOA eine Strategie zur **Strukturierung einer gesamten IT eines Unternehmens** ist.

⇒ **SOA und Microservices unterscheiden** sich von der Ausrichtung **fundamental**, auch wenn sie ähnliche Kommunikationsmechanismen nutzen können.

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Microservices

Weitere Abgrenzung zur SOA



- Microservices können auf ein Projekt beschränkt sein
  ⇒ **keine unternehmensweite Entscheidung**.
- Microservices sind **nur eine Möglichkeit** der Modularisierung. Andere wie beispielsweise Bibliotheken oder andere Mechanismen werden nicht ausgeschlossen.
- Microservices **unabhängig voneinander in Produktion** gebracht werden.
- Jeder Microservice kann in einem **eigenen Prozess**, einer **eigenen virtuelle Maschine** (VM) oder einem **Docker-Container** laufen.
  ⇒ Starke Trennung (Entkopplung) der Microservices.
- Die Microservices müssen **zu einer Anwendung kombiniert** werden, wie beispielsweise
  - über REST-Schnittstellen oder
  - als Teil einer Web-Schnittstelle

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Microservices

Ziele

Erhoffte Vorteile durch den Einsatz von Microservices[1)]

- Wiederverwendbarkeit
- Ersetzbarkeit
- Technologische Heterogenität
- Robustheit
- Skalierbarkeit
- Einfache Bereitstellung
- Parallelisierung des Entwicklungsprozesses

1) [S. Newman, Building Microservices, M. Loukides und B. MacDonald, Hrsg. O'Reilly Media, 2015.]

Microservices

Eigenschaften

Microservice-Architekturen legen typischerweise einen Fokus auf **folgende**

Prinzipien[1,2,3]

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

1) [S. Newman, Building Microservices, M. Loukides und B. MacDonald, Hrsg. O'Reilly Media, 2015.]
2) [J. Lewis und M. Fowler, "Microservices", https://martinfowler.com/articles/microservices.html, 2014.]
3) [I. Nadareishvili, R. Mitra, M. Mclarty und M. Amundsen, Microservice Architecture, B. MacDonald und H. Bauer, Hrsg. O'Reilly Media, 2016.]

Single Responsibility

Prinzipien

- [x] Single Responsibility
- [ ] Lose Kopplung, hohe Kohäsion
- [ ] Self-Containment
- [ ] Unabhängigkeit und Autonomie
- [ ] Single Team Ownership

Ein Microservice ist in der Regel für die Realisierung genau einer Geschäftsfunktion verantwortlich.

→ Entsprechend der Unix-Philosophie:
"Do One Thing and Do It Well"

Lose Kopplung, hohe Kohäsion

Prinzipien

- ☐ Single Responsibility
- ☒ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

Microservicearchitekturen streben nach einer möglichst losen Kopplung bezüglich:

- **Austauschbarkeit**
  Austausch bzw. Update eines Services sollte sich nicht auf abhängige Services auswirken (bei unveränderten Schnittstellen)
- **Kommunikation**
  Die Kommunikation zwischen Services sollte auf ein Minimum beschränkt werden. Die Kommunikation verläuft durch synchronen oder asynchronen Austausch einfacher Nachrichten.
  - Paradigma: Smart Endpoints, dumb pipes.
  - Typische Protokolle: HTTP (RESTful), AMQP, KAFKA, gRPC, …
  - Typische Nachrichtenformate: JSON, XML, Binary.

Worst Case bei zu enger Kopplung: Verteilter Monolith

Lose Kopplung, **hohe Kohäsion**

Prinzipien

- ☐ Single Responsibility
- ☒ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

Ein Microservice enthält typischerweise alle zur Umsetzung seiner Geschäftsfunktion relevanten Domänenkonzepte.

- Verringert den Grad der Kopplung.
- Verringert die Wahrscheinlichkeit, dass Änderungen an einem Service Änderungen an anderen Services nach sich ziehen.
- Redundanzen werden in Kauf genommen.

Self-Containment

Prinzipien

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☒ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

Neben den relevanten Domänenkonzepten und ihrer Implementierung umfasst jeder Microservice auch alle weiteren zu seinem Betrieb notwendigen Ressourcen, wie bspw.:

- Schnittstellenbeschreibungen
- Datenbanken
- Application- und Web-Container wie Tomcat , Jetty
- Konfigurationen für **OS-Container wie Docker**, Continuous-Integration-Pipelines (z.B. Jenkins-Pipelines) und Build-Management-Tools wie
- Maven oder Gradle
- Ggf. auch Teile der grafischen Oberfläche

Ziel: Verringerung von Abhängigkeiten zwischen den Microservices sowie den zuständigen Teams

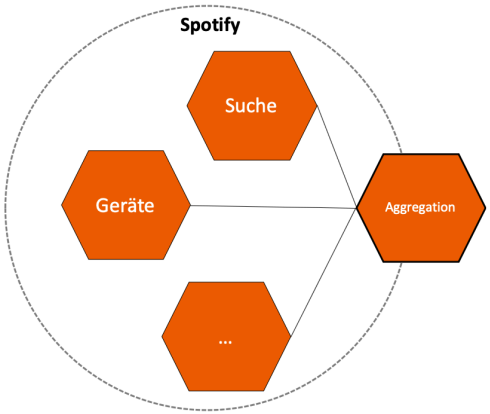
Self-Containment

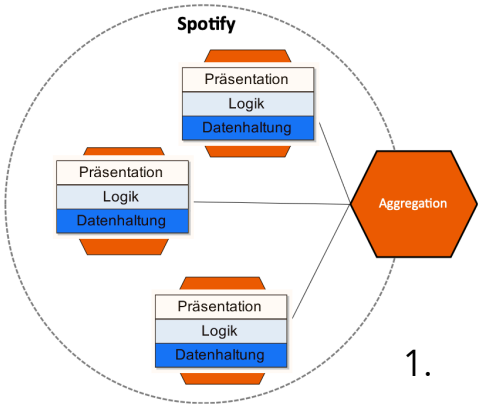
Prinzipien

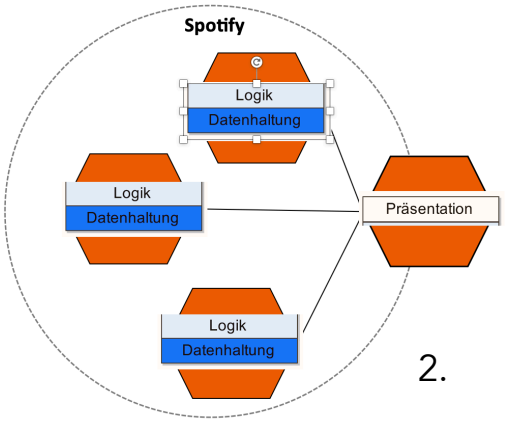
- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☒ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

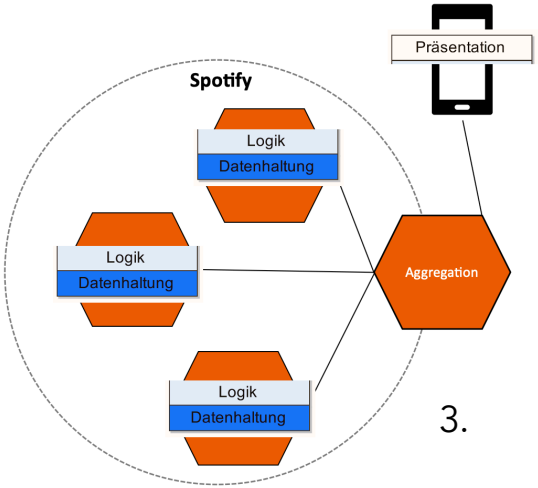
Varianten für die Auslieferung von grafischen Oberflächen

1. Jeder Service hat seine **eigene Präsentationsschicht**
2. Die Präsentation erfolgt **zentral**
3. Die Präsentation erfolgt **komplett clientseitig**









Unabhängigkeit und Autonomie

Prinzipien

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☒ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

Der Software-Lebenszyklus eines Microservices ist unabhängig von denen anderer Microservices

Dies beinhaltet folgende Phasen:

- Entwicklung
- Test
- Bereitstellung (Deployment)
- Betrieb

> "The **golden rule**:
> Can you make a change to a service and deploy it by itself without changing anything else?"

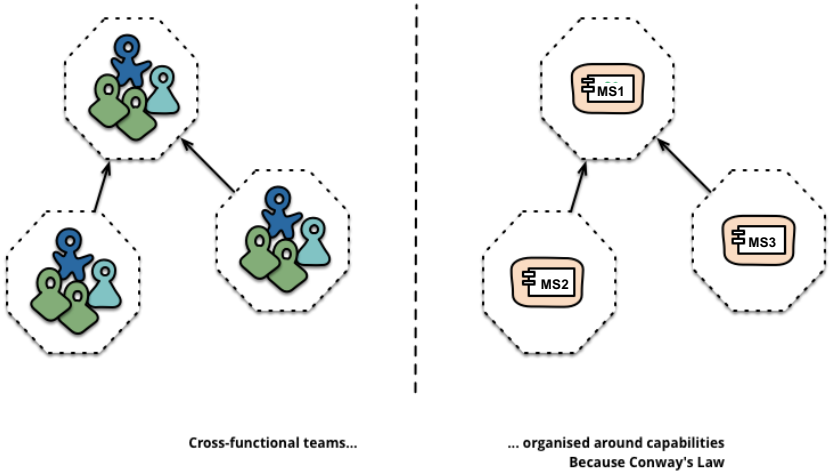
Ziel: Jeder Microservice soll autonom entwickelt, deployed und betrieben werden können.

[S. Newman, Building Microservices, M. Loukides und B. MacDonald, Hrsg. O'Reilly Media, 2015.]

Single Team Ownership

Prinzipien

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☒ Single Team Ownership



[Bild:https://martinfowler.com/articles/microservices.html]

Für jeden Microservice sollte **genau ein Team** verantwortlich sein:[1]

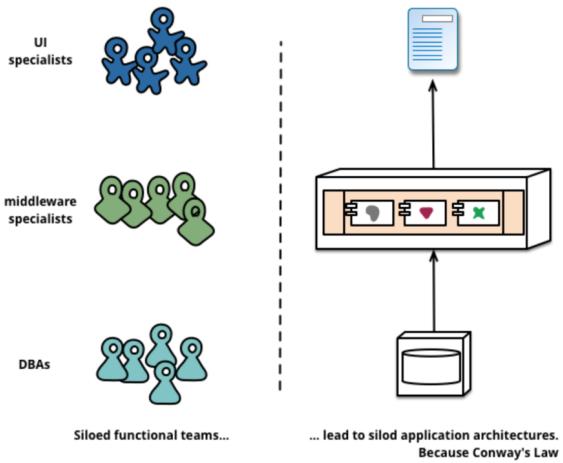
- Das Team ist nicht nur für Entwurf und Entwicklung des Microservices verantwortlich, sondern auch für alle anderen Aspekte (Konfiguration, Deployment, Betrieb)
- Cross-Functional Teams, z.B. DevOps[2]

1) [S. Newman, Building Microservices, M. Loukides und B. MacDonald, Hrsg. O'Reilly Media, 2015.]
2) [I. Nadareishvili, R. Mitra, M. Mclarty und M. Amundsen, Microservice Architecture, B. MacDonald und H. Bauer, Hrsg. O'Reilly Media, 2016.]

Single Team Ownership

Prinzipien

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☒ Single Team Ownership



- Microservice-Architekturen erfordern in der Regel **mehrere Teams** für die Entwicklung
- Teams können **anhand der technischen Skills organisiert** werden:
  Zum Beispiel können alle Frontend-Entwickler zu einem Team zusammengefasst werden
  ⇒ **Urlaubsvertretung** oder **fachlicher Austausch** sind so sehr einfach.
- Die **Aufteilung der Teams beeinflusst aber die Architektur**.
  ⇒ Gesetz von Melvin Edward Conway

Belegt durch empirische Untersuchungen!

Conway's Law:
Eine Organisation kann nur eine Architektur hervorbringen kann, die ihren Kommunikationsbeziehungen entspricht.

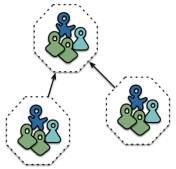
[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016 https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Single Team Ownership

Prinzipien

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☒ Single Team Ownership

- Probleme der **Teambildung nach technischen Skills**:
  - Die Arbeit der Teams muss eng koordiniert werden.
  - Eine Verzögerung bei einem Team beeinflusst die anderen Teams.
  - Verantwortlichkeiten können ggf. zwischen den Teams hin- und her verschoben werden



- **Cross-funktionale Teams** sind bei Microservices quasi vorgezeichnet:
  - **Änderung** soll möglichst **nur einen Microservice** betreffen.
  - Team bekommt die **Verantwortung** für **eine bestimmte Fachlichkeit**, die in einem oder mehreren Microservices implementiert ist. (→ 2 Pizza-Teams)
  - Das Team muss daher **technisch breit aufgestellt** sein. Es muss schließlich Backend, Frontend und Datenbank für die Fachlichkeit verantworten.
    → passt zu agilen Teams

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Single Team Ownership

Prinzipien

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☒ Single Team Ownership

- Vorteile cross-funktionaler Teams:
  - **Unabhängigkeit der Teams** ⇒ wenig Absprachen
  - Jeder Microservice kann mit einem **eigenen Technologiestack** implementiert werden.
  - Jedes Team kann die Technologie nutzen, die **für das jeweilige Problem angemessen** ist.

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Microservices

Eigenschaften

Microservice-Architekturen legen typischerweise einen Fokus auf **folgende**

Prinzipien[1,2,3]

- ☐ Single Responsibility
- ☐ Lose Kopplung, hohe Kohäsion
- ☐ Self-Containment
- ☐ Unabhängigkeit und Autonomie
- ☐ Single Team Ownership

1) [S. Newman, Building Microservices, M. Loukides und B. MacDonald, Hrsg. O'Reilly Media, 2015.]
2) [J. Lewis und M. Fowler, "Microservices", https://martinfowler.com/articles/microservices.html, 2014.]
3) [I. Nadareishvili, R. Mitra, M. Mclarty und M. Amundsen, Microservice Architecture, B. MacDonald und H. Bauer, Hrsg. O'Reilly Media, 2016.]

Integration - Kommunikation

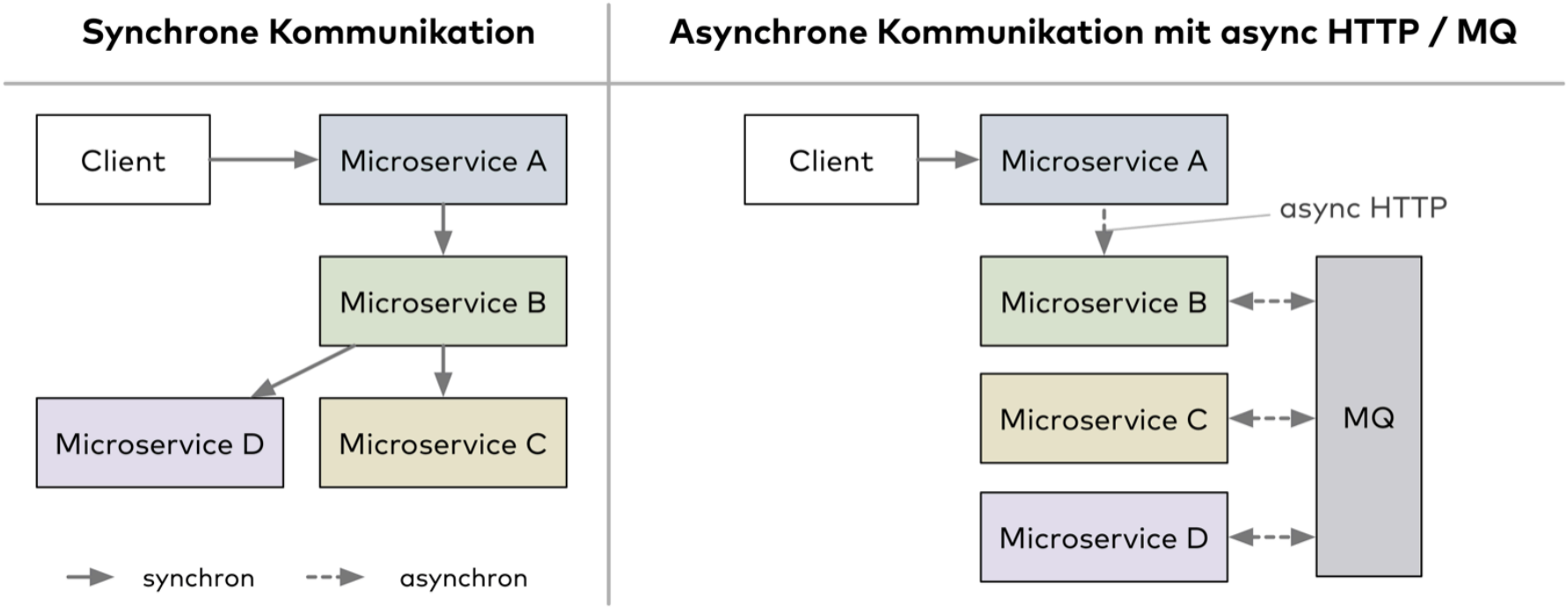
Herausforderung Kommunikation

- Die **Probleme** von Microservices sind zum großen Teil mit den **gegenseitigen Netzwerkaufrufen** verbunden.
- Die **einfachste Möglichkeit**, eine monolithische Anwendung zu Microservices zu migrieren, ist die Anwendung aufzuspalten und Methodenaufrufe zwischen Modulen (bzw. Microservices) mit **synchronen Netzwerkaufrufen** zu ersetzen.
- Eine **asynchrone Umsetzung** der Kommunikation bietet allerdings Vorteile wie eine **geringere Latenz** und eine **losere Kopplung** der Microservices.
  ⇒ Die Schwierigkeit ist, sich von der **Frage-Antwort-Interaktion** loszulösen.

[H. Prinz: „Brauchen asynchrone Microservices und SCS ein Service Mesh?“, 2020
https://www.innoq.com/de/articles/2020/02/service-mesh-asynchrone-microservices-scs/]

Integration - Kommunikation

Herausforderung Kommunikation



[H. Prinz: „Brauchen asynchrone Microservices und SCS ein Service Mesh?“, 2020
https://www.innoq.com/de/articles/2020/02/service-mesh-asynchrone-microservices-scs/]

Bestimmung der Servicegröße

- **Risiken** bei **zu kleinen oder zu großen Services**
  - Netzwerkauslastung,
  - Komplexität bei Transaktionen,
  - Infrastrukturaufwand

- **Wie groß ist ein Microservice** bzw. wie groß ist eine „Geschäftsfunktion“?
  - Zeilen an Code → ?
  - Komplexitätsmetriken → ?
  - Anzahl an Testfällen → ?

- Es existiert **kein quantitatives Maß** für die „richtige“ Größe!

Bestimmung der Servicegröße

Mögliche (unscharfe) Kriterien zur Bestimmung der Größe[1]

- **Geschäftsfunktion**
  Der Microservice sollte alle zur Realisierung seiner Geschäftsfunktion relevanten Domänenkonzepte umfassen.
- **Teamgröße**
  Es sollte nicht mehr als ein Team notwendig sein, um einen Microservice zu entwickeln. Als Teamgröße werden häufig 5 – 7 Personen angegeben („Two-Pizza-Teams“).
- **Verstehbarkeit**
  Ein Microservice sollte zur Gänze von einem einzelnen Entwickler verstanden werden können.
- **Ersetzbarkeit**
  Ein Microservice sollte leicht ersetzbar sein („Two-Week-Implementation“).

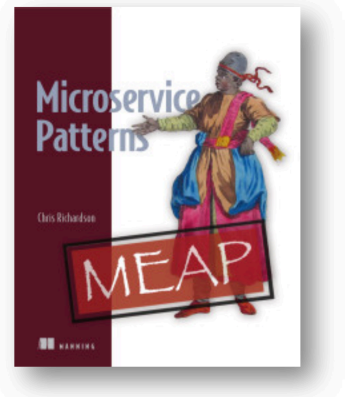
1) [Eberhard Wolff: Das Microservices-Praxisbuch, Hrsg. dpunkt.verlag, 2018.]

Entwurfsmuster in Microservicearchitekturen

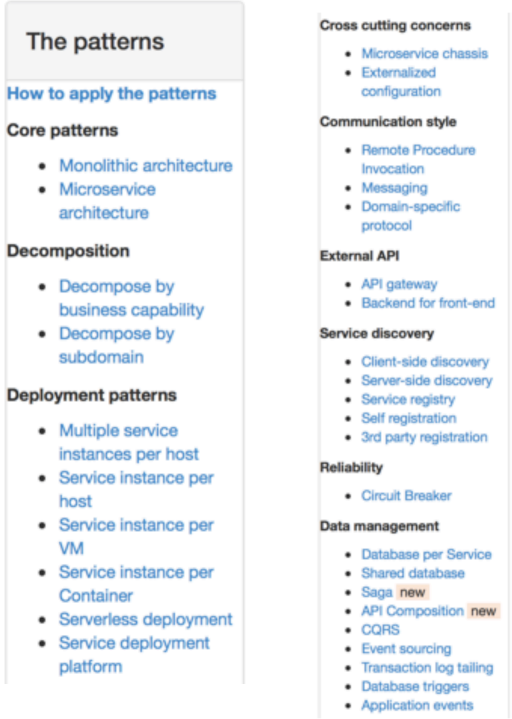
- In der Regel werden bestimmte **Geschäftsfunktions-unabhängige Services** benötigt, sogenannte **infrastrukturelle Services**.

⇒ wiederverwendbare, bewährte Lösungen für die Erstellung von Microservicearchitekturen

(→ Entwurfsmuster) [https://microservices.io/patterns/]



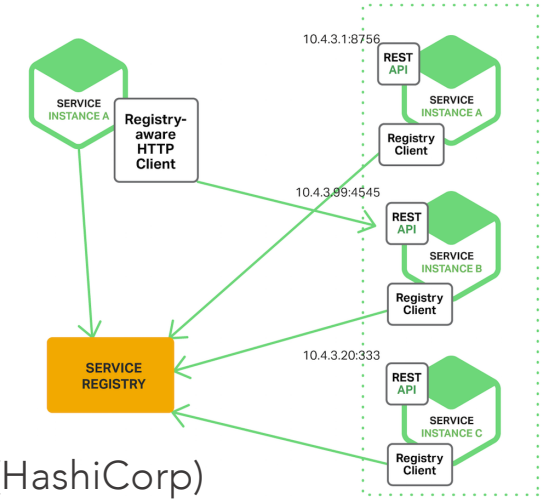
Chris Richardson:
Microservice Patterns.
Hrsg. Manning, 2018.



Beispiele:
Service Discovery, API Gateway, Load Balancer, Cuircuit Breaker, Authentication,Config Storage …

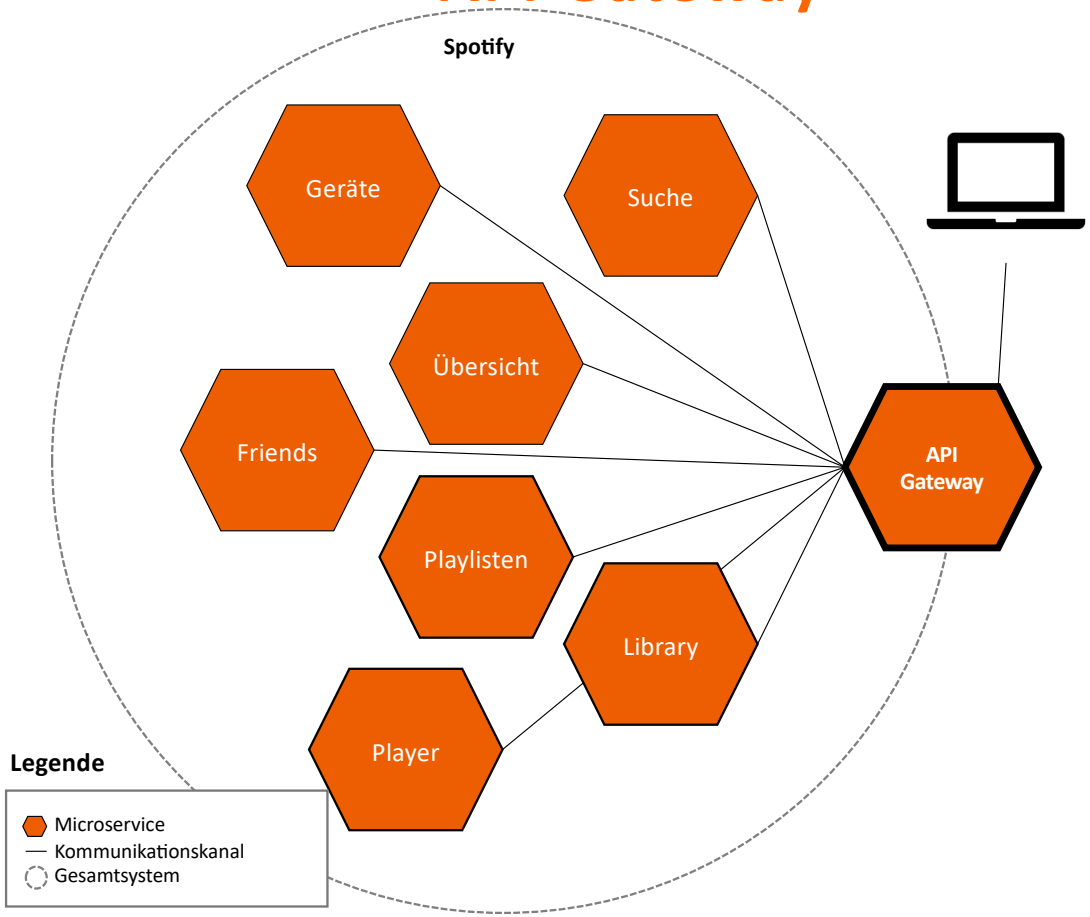
Service Discovery

- Woher wissen die einzelnen Services eigentlich, wie sie die anderen Services erreichen können?
- **Service Discovery** ist vergleichbar mit einer SOA-Registry.
  - Services **melden sich an** einer zentralen Stelle unter einem Alias an.
  - Andere **Services können diesen Alias** bei der Service Discovery **anfragen** und erhalten die tatsächliche Adresse.
- **Client-side Service Discovery**
  - Netzwerkadresse einer Serviceinstanz **wird automatisiert bei einer Service Registry** registriert.
  - Sollte die Serviceinstanz terminieren, wird der "Tod" des Service mittels **Heartbeat-Mechanismus** detektiert.
  - Services können mithilfe einer bereitgestellten API die Adresse anderer Serviceinstanzen bei der Registry anfragen.
  - Typische Vertreter: Eureka (Spring Cloud bzw. Netflix OSS) oder Consul (HashiCorp)



Bildquelle : NGINX,
https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

API Gateway

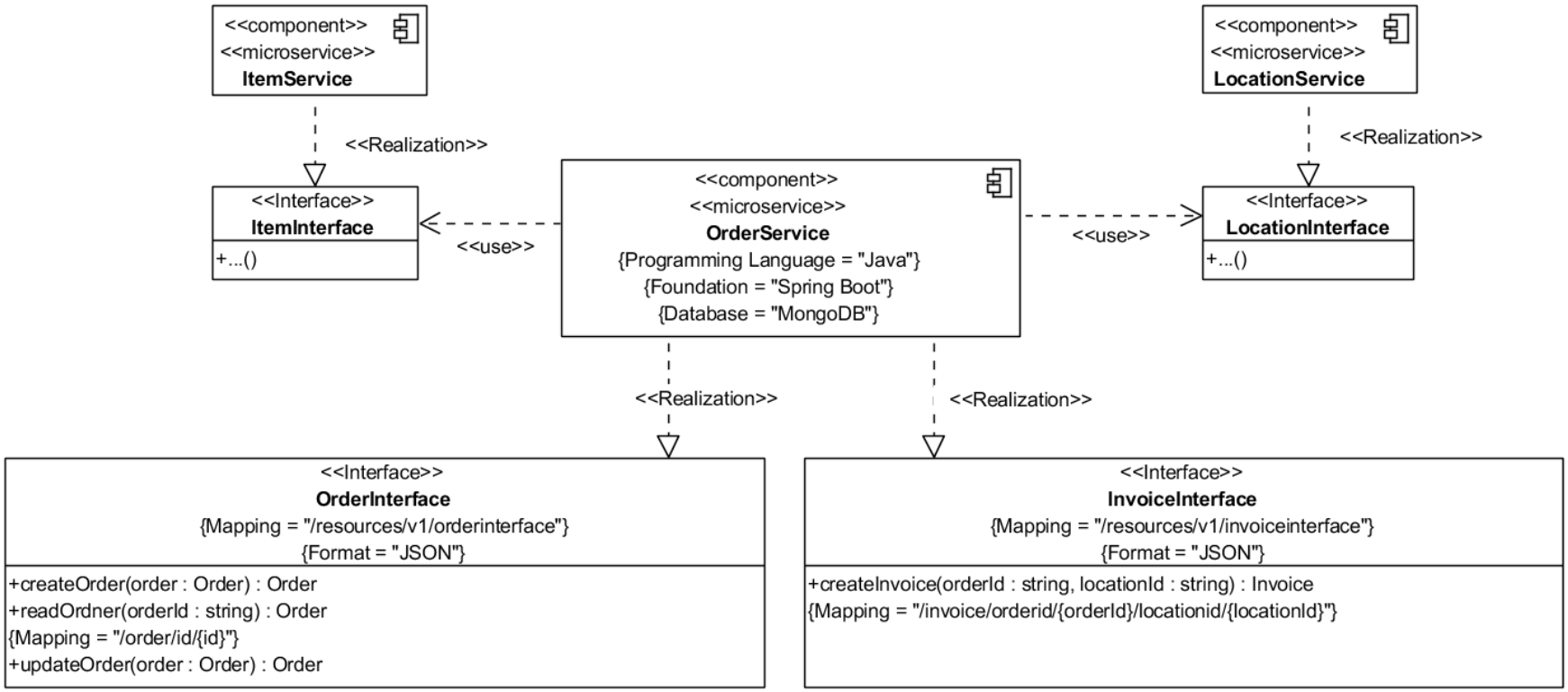


Das **API Gateway** fungiert für Clienten als zentraler Zugangspunkt zum System

- Für Clienten sind die hintergelagerten Services versteckt, d.h. ein Client merkt u.U. gar nicht, dass es sich um eine Microservicearchitektur handelt.

- API Gateways werden teilweise um weitere Funktionalitäten ergänzt
  - Load Balancing
  - Security
  - …

Codebeispiel

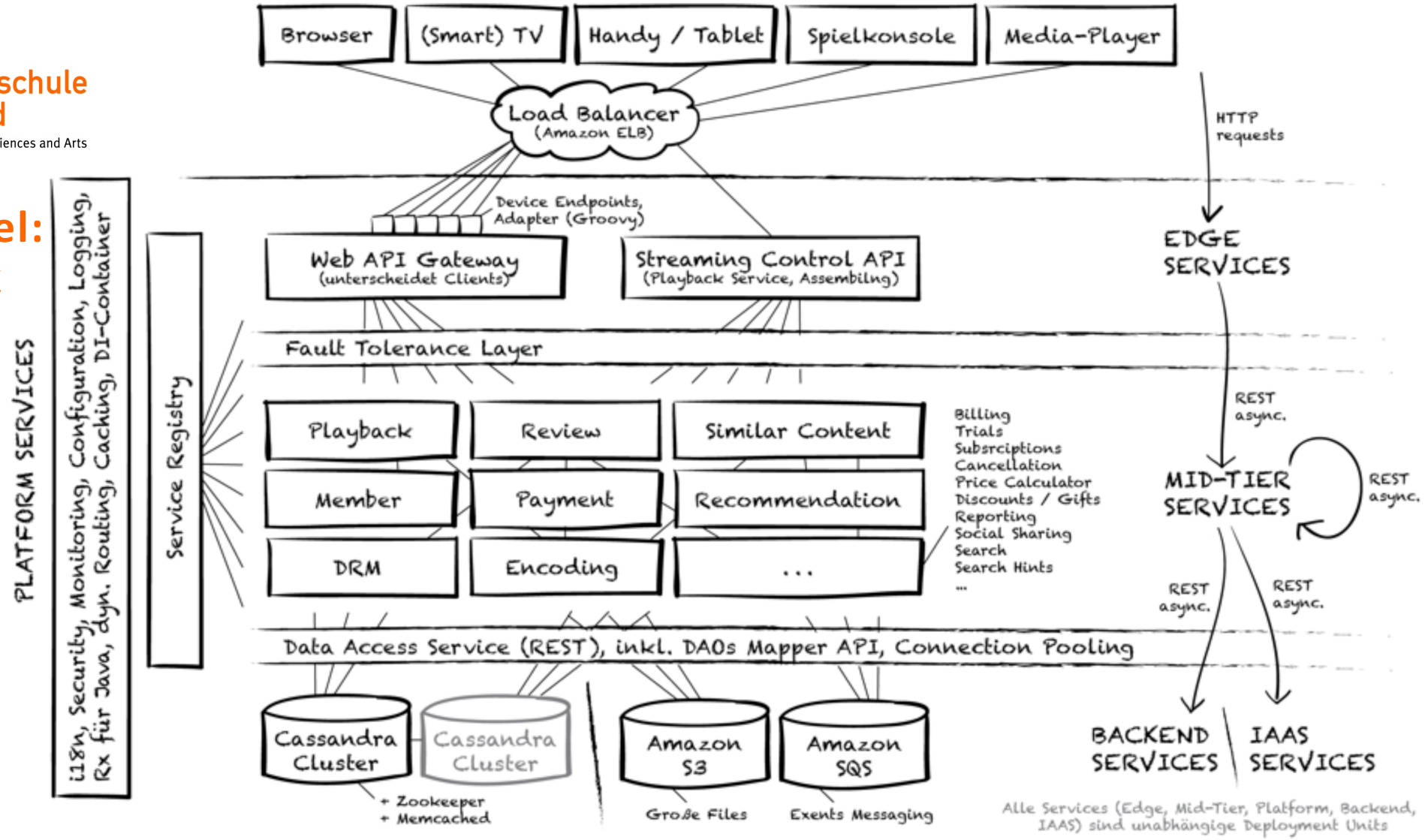


Verfügbar auf GitHub: https://github.com/SeelabFhdo/SWT2-OrderSystem

Anwendungsbeispiele

Fachhochschule Dortmund
University of Applied Sciences and Arts

Beispiel: Netflix



[Stefan Zörner: Drei zentrale Entwurfsfragen bei vertikalen Anwendungsarchitekturen, Java Forum Nord. Hannover, 2018.]

Microservices

Fazit - Silver Bullet ?!

- Heterogenität
  Vorteil: Flexibilität in Technologie-Wahl
  Nachteil: Mehr Know-how notwendig, höhere Betriebskosten, mehr Komplexität
- Resilienz (Widerstandsfähigkeit)
  Vorteil: Sicherheit gegen Infrastruktur-Ausfall
  Nachteil: muss explizit dafür programmiert werden
  („Circuit Breaker" Pattern, Auto-Reconnect, etc.)
- Skalierbarkeit
  Vorteil: höhere Entwickler-Effizienz, höhere Runtime-Performance
  Nachteil: erhöhte Komplexität, Service Discovery, schwierigeres Monitoring
- EasyDeployment
  Vorteil: jeder einzelne Microservice leichter installierbar/upgradebar
  Nachteil: Gesamtanwendung hat viele Abhängigkeiten

[R.S. Engelschall: „Microservices - Architekturansatz mit großen Herausforderungen und gewissen Nebenwirkungen", 2016]

Microservices

Fazit - Silver Bullet ?!

- Organisatorische Ausrichtung
  Vorteil: stärkerer Fokus auf fachliche Einheiten (→ Conway's Law)
  Nachteil: eventuell mehrere technische Durchstiche notwendig
- Komponierbarkeit
  Vorteil: Funktionalitäten flexibel zusammenbaubar
  Nachteil: erhöhte Komplexität durch Orchestrierung/Choeographie, mehr Abhängigkeiten entstehen
- Wiederverwendbarkeit
  Vorteil: Funktionalitäten in mehreren Anwendungen, nur einmal pflegen
  Nachteil: Alle Anwendungen gleichzeitig betroffen
- Austauschbarkeit
  Vorteil: Funktionalitäten getrennt austauschbar (Update/Upgrade)
  Nachteil: Alle Anwendungen gleichzeitig betroffen

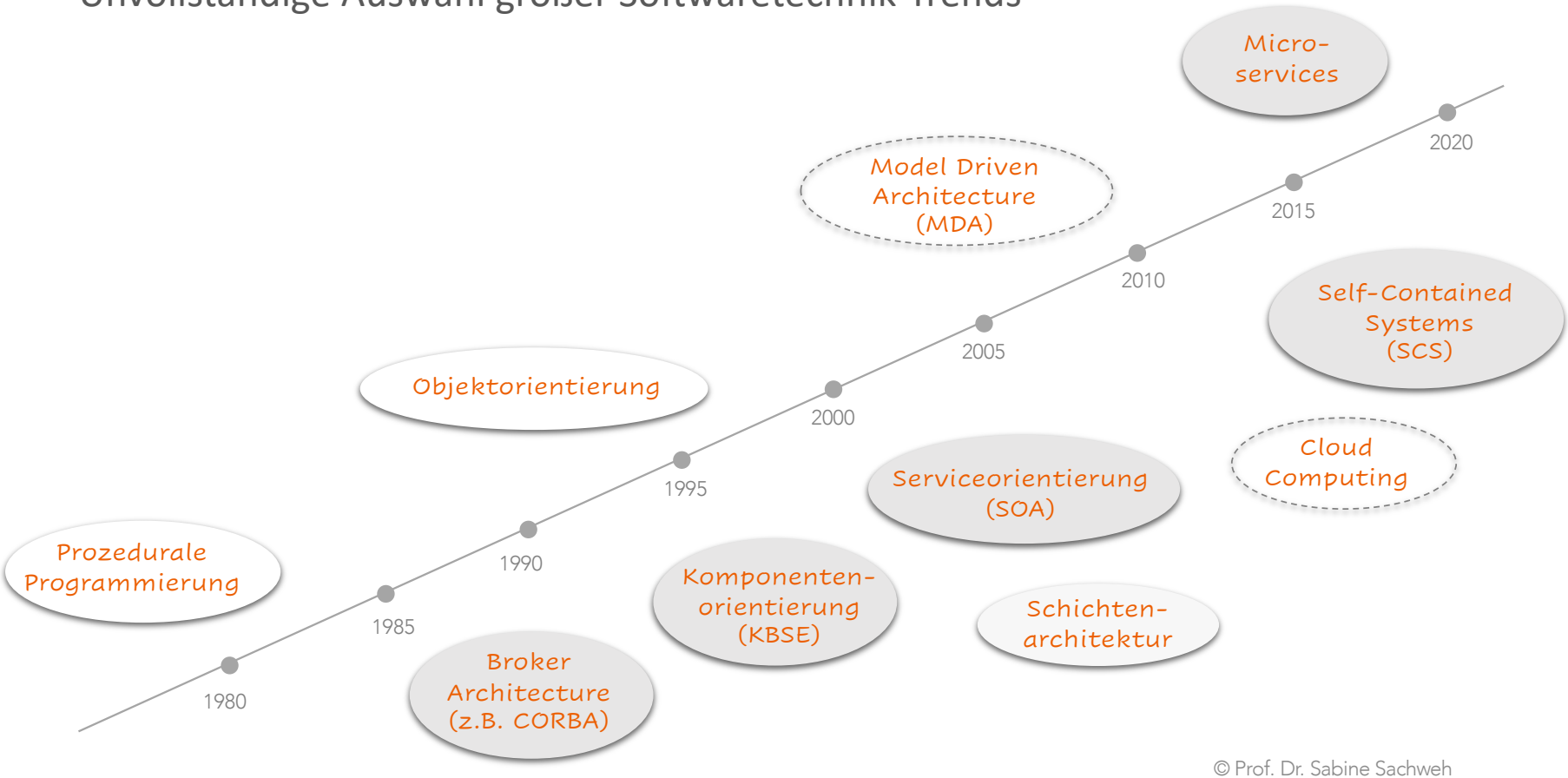
[R.S. Engelschall: „Microservices - Architekturansatz mit großen Herausforderungen und gewissen Nebenwirkungen", 2016]

Fachhochschule
Dortmund
University of Applied Sciences and Arts

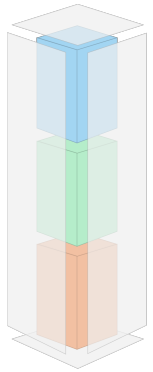
Self Contained Systems (SCS)

Entwicklungs- und Architekturtrends im Zeitverlauf

Unvollständige Auswahl großer Softwaretechnik-Trends



Self-contained System (SCS)

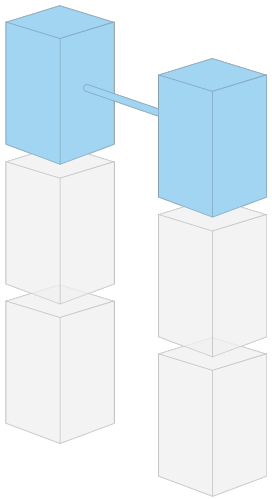


- **Microservices** enthalten **viele Freiheitsgrade**, wie beispielsweise die Größe der Services, die Art der Kommunikation und Integration oder das Schneiden der Services
- **Self-contained Systems** (SCSs) sind Microservices mit einer Reihe konkreter **Festlegungen**:
  - Jedes SCS ist eine **eigenständige Web-Anwendung** inkl. Daten, Logik und Code zur Darstellung der Web-Schnittstelle.
  - Jedes SCS soll seine **eigene UI** haben und sich auch **keinen Geschäftscode** mit anderen SCS **teilen**.
  - Ein SCS **kann** auch eine **Service-API** haben, um die Logik für andere SCS oder mobile Clients anzubieten.
  - Jedes SCS wird von **einem Team** verantwortet.
  - Die **Kommunikation** mit Fremdsystemen und anderen SCSs ist nach Möglichkeit **asynchron**.
    → Entkopplung

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Self-contained System (SCS)

Integration



- SCSs können zu Anwendungen zusammengesetzt werden.

- Neben der asynchronen Kommunikation wird primär eine Integration auf Ebene der Weboberflächen empfohlen. Beispielsweise über
  - **Weblinks**, die ein Nutzer allerdings explizit anklicken muss; oder über
  - **JavaScript-Code**, der eine Seite nach bestimmten Links scannt und die Links durch den Inhalt der referenzierten Seite ersetzt; oder durch
  - **Features von Web-Servern** wie Server-Side Includes, d.h. der Web Server ersetzt selbst Teile der Web-Seite durch die referenzierten Inhalte.

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Self-contained System (SCS)

Mikro- und Makro-Architektur

- SCS können größer als Microservices sein (→ komplette Web-Anwendung)
- ein SCS kann intern **aus mehreren Microservices zusammengesetzt sein**
  ⇒ mehrere kleinere Deployment-Einheiten
  ⇒ Komplexität steigt, weil diese Instanzen alle verwaltet bzw. betrieben werden müssen
  ⇒ Aufteilung ist also ein Trade-Off
- Bei der Architektur eines SCS unterscheidet man die Mikro- und Makro-Architektur:
  - **Mikro-Architektur** ⇔ Entscheidungen, die jedes Team für sein SCS selber treffen kann.
  - **Makro-Architektur** ⇔ Entscheidungen, die global für alle Teams und SCSs festgelegt werden.
- Nur **wenige Entscheidungen** müssen zwingend in der **Makro-Architektur** getroffen werden wie beispielsweise das Protokoll für die Kommunikation der SCSs untereinander oder die Technologien für die UI-Integration.

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Self-contained System (SCS)

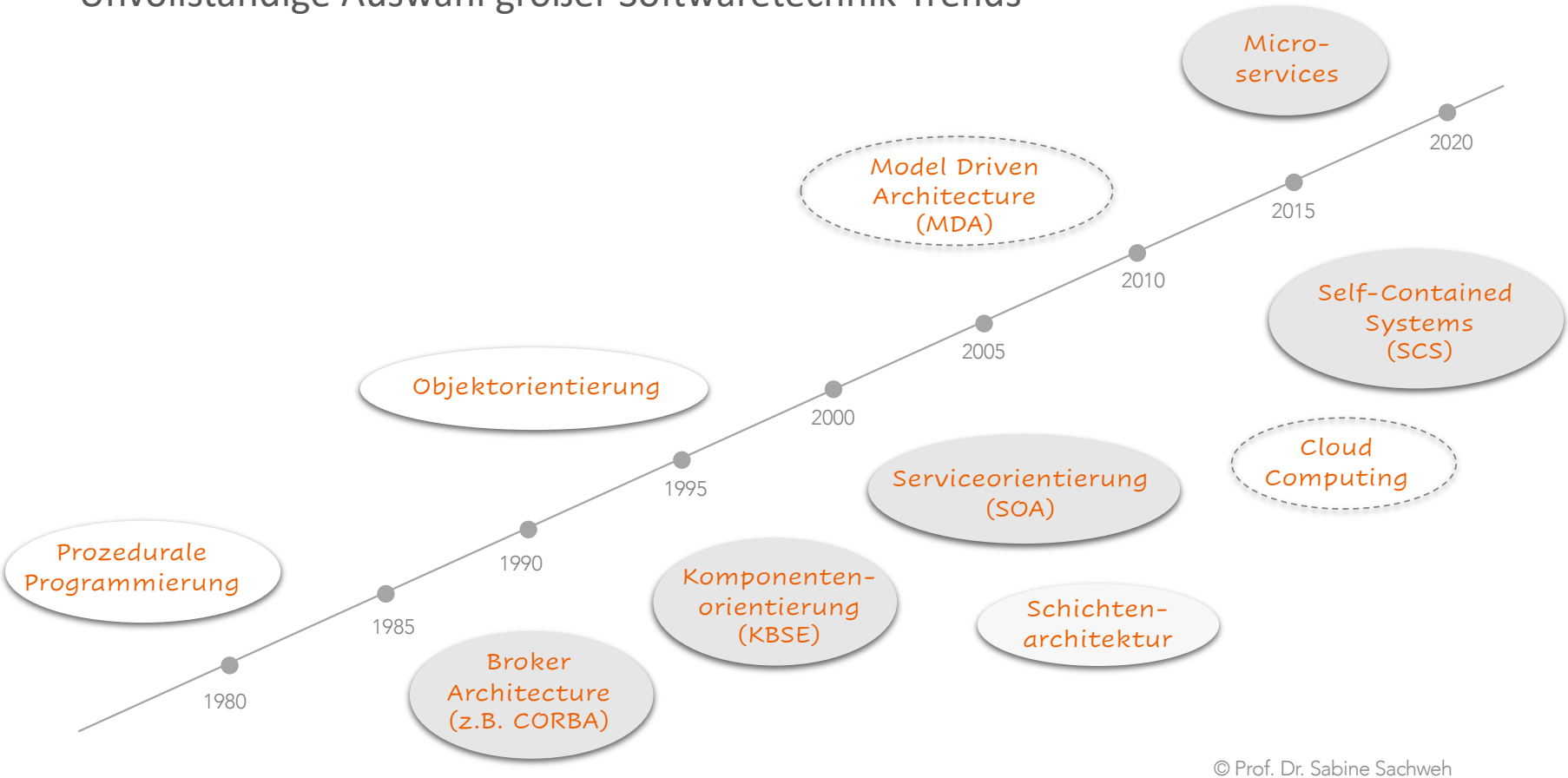
Abgrenzung gegenüber Microservices

- Beide Ansätze:
  - Aufteilung eines Systems in kleinere (Service-)Einheiten auf.
  - Ein Team verantwortet Komponente (Microservice oder SCS)
- SCSs sind im Vergleich zu Microservices eher grobgranular
  → komplette Webanwendungen
- SCS fordern noch stärker die lose Kopplung der Komponenten

[Eberhardt Wolff: „Services: SOA, Microservices und Self-contained Systems", 2016
https://www.innoq.com/de/articles/2016/11/services-soa-microservices-scs/]

Entwicklungs- und Architekturtrends im Zeitverlauf

Unvollständige Auswahl großer Softwaretechnik-Trends



Abschlussbemerkung

Lohnt es sich einen Sportwagen zu kaufen, um damit morgens zum 300m entfernten Bäcker zu fahren?

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Architekturstile III

Microservice Architecture (MSA)

Self-contained Systems (SCS)

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Architekturstile III