# Projektarbeit

Thema:

## Gemeinsamkeiten und Unterschiede von SOA und Microesrvices

**Stefan Kruk**
geboren am 14.08.1992
Matr.-Nr.: 7084972

An der Fachhochschule Dortmund im Fachbereich Informatik erstellte
Projektarbeit
im Studiengang Softwaretechnik (Dual)

**Betreuer:** Prof. Dr. Johannes Ecke-Schüth

**Fachbereich Informatik**

Dortmund, 13. Dezember 2016

# Überblick / Abstract

## Kurzfassung

In dieser Arbeit werden die Paradigmen SOA und Microservice miteinander verglichen. Dafür werden die Einsatzmöglichkeiten, sowie die Architekturen und Schnittstellen der beiden Paradigmen erläutert und miteinander verglichen. Außerdem werden die Vor- und Nachteile des jeweiligen Paradigmas, bezüglich Prozessisolation, Skalierung, Deployment, Wartbarkeit (Korrigierbarkeit, Erweiterbarkeit, Anpassbarkeit, Verbesserung), Entwicklung/Testbarkeit und der Bindung an Technologiestacks herausgearbeitet und miteinander verglichen.

## Abstract

In this Projekt, the Paradigm SOA and Microservice will be compared with each other. For this the Application possibilites, Architecture and Interfaces of both Paradigm will be explained and compared with each other. Furthermore the advantages and disadvantages in terms of Processisolation, Skaling, Deployment, Maintainability (Correction, Expandability, Adaptability, Refinement), Development/Testing and bounding to Technologiestacks of both Paradigm will be compared with each other.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Glossar

**Microservice**

Microservices sind ein Architekturmuster aus dem Bereich der verteilten Anwendungen, bei dem komplexe Anwendungen in eigenständige Prozessen zerlegt werden. Diese kommunizieren mit Hilfe von sprachunabhängigen Schnittstellen untereinander. Die Prozesse, welche ebenfalls Microservices genannt werden, sind kleine unabhängige Dienste, welche möglichst nur eine Aufgabe erledigen sollen, diese aber dafür besonders gut. Dadurch ist ein modularer Aufbau von Anwendungssoftware möglich.

**SOA**

SOA steht für Service-orientierte Architektur und ist ein Architekturmuster für Service-orientierte Systeme. SOA ist in dem Bereich der verteilten Systeme einzuordnen und wird eingesetzt, um Dienste von IT-Systemen zu strukturieren und zu nutzen.

# Kapitel 1

# Einleitung

Die Idee der Aufteilung eines komplexen Softwaresystemes ist nicht neu. Monolithische Systeme bestehen meistens aus mehreren Modulen, welche zusammen zu einer ausführbaren Datei gepackt werden. Es kann jedoch auch möglich, dass Module in eigene Bibliotheken verpackt werden, welche zur Laufzeit geladen werden. Da die Bibliotheken für die korrekte Ausführung der Software notwendig sind, müssen diese mit ausgeliefert werden. Müssen einzelne Programmteile verändert (korrigiert, erweitert, angepasst oder verbessert) werden, muss jede Software, welche diesen Programmteil verwendet neu gebaut und ausgeliefert werden. Dies kann einen erheblichen Aufwand bedeuten, wenn dieser Programmteil in einer großen Anzahl von Software verwendet wird. Daher wurde die Idee entwickelt, Fachlichkeiten in einzelne Dienste (Services) auszulagern und zentral zu verwalten. Das Konzept der Verteilten Anwendungen wurde erstellt.

Dieses Konzept wurde nach und nach weiterentwickelt. Zwei Paradigmen, welche diesem Ansatz folgen sind SOA (Service-orientierte Architektur) und Microservices. Beide Paradigmen setzten auf einzelne Services, welche ihre Funktionalitäten über APIs anbieten.

## 1.1 Begriffsabgrenzungen

### SOA und „Service-orientierte Architektur“

Sowohl SOA als auch Microservice zählen zu dem Paradigmen der „Service-orientierten Architekturen“.
Die Abkürzung SOA wird hier sowohl für das Paradigma „Service-orientierte Architektur“ verwendet, wie auch für die spezielle Herangehensweise.
Um die beiden Thematiken in dieser Arbeit abzugrenzen wird der Begriff „SOA“ für die Herangehensweise verwendet und die ausgeschriebene Variante zur Kennzeichnung des allgemeinen Paradigmas.

## 1.2 Zielsetzung und Aufgabenstellung

Ziel dieser Projektarbeit ist es, die Unterschiede und Gemeinsamkeiten von SOA und Microservices zu beleuchten. Es werden sowohl die Einsatzmöglichkeiten, als auch die nötigen Architekturen und Schnittstellen, des jeweiligen Paradigmas, ermittelt. Außerdem werden die Vor- und Nachteile des jeweiligen Paradigmas, bezüglich der Prozessisolierung, Skalierung, Deployment, Wartbarkeit (Korrigierbarkeit, Erweiterbarkeit, Anpassbarkeit, Verbesserung), Entwicklung/Testbarkeit und der Bindung an Technologiestacks herausgearbeitet.

## 1.3 Vorgehensweise

Zu beginn wird eine Problemanalyse von monolithischen Systemen durchgeführt. Danach werden die allgemeinen Grundlagen für die Verwendung von Verteilten Systemen und die damit verbundenen Probleme erläutert. In diesem Zusammenhang werden die Unterschiede zu monolithischen Systemen herausgearbeitet und der Begriff „Service“ genauer Definiert. Auf dieser Basis werden die Paradigmen SOA und Microservices vorgestellt und genauer beleuchtet.

Darauf aufbauend werden die Einsatzmöglichkeiten der jeweiligen Paradigmen analysiert und die dafür nötigen Architekturen und Schnittstellen herausgearbeitet. Insbesondere soll

ermittelt werden, welche Plattformen und Voraussetzungen für den Einsatz der jeweiligen Paradigmen notwendig sind. Außerdem werden die Vor- und Nachteile der Paradigmen, hinsichtlich der Prozessisolierung, Skalierung, Deployment, Wartbarkeit (insbesondere der Korrigierbarkeit, Erweiterbarkeit, Anpassbarkeit, Verbesserung), Entwicklung/Testbarkeit und der Bindung an Technologiestacks herausgearbeitet und mit einander verglichen.

Abschließend wird ein Fazit aus den gewonnen Erkenntnissen gezogen und ein Ausblick auf weiterführende Arbeiten gegeben.

# Kapitel 2

# Problemanalyse von Monolithischen Systemen/Anwendungen

In der klassischen Softwareentwicklung sind monolithische Anwendungen, oft die Grundlage für komplexe Systeme. Dabei besitzt die Anwendung alle nötigen Ressourcen und Eigenschaften, um eine bestimmte Aufgabe zu erfüllen. Dies fängt an bei der Datenverwaltung, wie zum Beispiel der Persistierung und dem laden von Daten. Dies kann beispielsweise durch eine Datenbankschnittstelle umgesetzt werden. Dabei wird jedoch meistens nur eine einzelnes Datenbanksystem unterstützt. Neben der Datenverwaltung gehören noch weitere Aufgaben zu einer Anwendung, wie zum Beispiel das durchführen von verschiedenen Berechnungen auf Grundlage der vorhandenen Daten. Damit ein Benutzer, eine Anwendung benutzen kann, muss zudem eine Benutzerinterface vorhanden sein. Dies kann zum Beispiel durch eine Internetseite oder durch eine native Desktop Darstellung geschehen.

## 2.1 Herausforderungen bei der Verwendung von monolithischen Systemen

Zu Beginn der Entwicklung, stellt das Erstellen einer Anwendung, auf Basis eines Pflichtenheftes keine große Herausforderung dar. Es können die Anforderungen an das neue System analysiert werden und darauf aufbauend eine Programmiersprache und die interne Architektur gewählt werden.

Werden hingegeben, bei einer bestehenden Anwendung, neue Anforderungen gestellt, ist man zunächst einmal an den zuvor gewählten Technologie Stack gebunden. Das Korrigieren von Fehlern ist relativ einfach, im Vergleich zur Anpassbarkeit und Erweiterbarkeit des Systems. Je nach Größe der Anwendung, kann dadurch die Umsetzung der neuen Anforderungen problematisch werden, sofern diese nicht mit den bestehenden internen Architekturen vereinbar sind. In diesem Fall kann die Anforderung dazu führen, dass die Anwendung, umgeschrieben werden muss und im schlimmsten Falle neu geschrieben werden muss.

Ein weiteres Problem besteht in der parallelen Weiterentwicklung.

> »Es gibt Teams, die an verschiedenen neuen Features arbeiten. Aber die parallele Arbeit ist kompliziert: Die Struktur der Software ist dafür zu schlecht. Die einzelnen Module sind zu schlecht separiert und haben zu viele Abhängigkeiten untereinander. « [7, S. 16]

Ohne eine Versionsverwaltung ist das parallele Arbeiten nicht möglich, da ansonsten die Gefahr zu groß ist, Programmteile von anderen Teams, ungewollt zu überschreiben. Ein weiteres Problem besteht bei der Durchführung von Integrationstests.

> »Wenn der Deployment-Monolith durch einen Integrationstest läuft, dürfen in dem Test nur die Änderungen eines Teams enthalten sein. Es gab Versuche, mehrere Änderungen auf einmal zu testen. Dann war bei einem Fehler nicht klar, woher das Problem kam, und es gab lange und komplexe Fehleranalysen.« [7, S. 16]

Dadurch entsteht ein Flaschenhals, da andere Teams warten müssen, bis die Integrationstests erfolgreich durchlaufen wurden.

### 2.1.1 Skalierung

Skalierung ist ein wichtiges Thema von Webanwendungen. Greifen viele Benutzer auf ein und dieselbe Anwendung zu, so kann es dazu führen, dass dies das System überlastet und alle Benutzer länger auf die Antwort des Systems warten müssen. Um dies zu verhindern, müssen Anwendungen skaliert werden.

> »Skalierbarkeit bedeutet, dass ein System mehr Last bearbeiten kann, wenn es mehr Ressourcen bekommt.« [7, S. 150]

Es gibt nach [7, S. 150] zwei verschiedene Arten der Skalierbarkeit:

- *Horizontale Skalierbarkeit* bedeutet, dass mehr Ressourcen zur Verfügung stehen, die jeweils einen Teil der Last bearbeiten, die Anzahl der Ressourcen steigt also.
- *Vertikale Skalierbarkeit* bedeutet, dass leistungsfähigere Ressourcen genutzt werden, um mehr Last handzuhaben. Eine einzelne Ressource wird also mehr Last abarbeiten. Die Anzahl der Ressourcen bleibt konstant.

Beide Arten der Skalierbarkeit haben ihre Vor- und Nachteile. Welche Art benutzt wird, muss im individuell entschieden werden, jedoch ist die Vertikale Skalierung bei Monolithen, meistens deutlich einfacher, als die Horizontale Skalierung.

# Kapitel 3

# Allgemeine Grundlagen zur Verwendung von Service-orientierten Systemen

Service-orientierte Systeme, sollen Dienste innerhalb eines IT-Systems strukturieren. Es ist einzuordnen in dem Bereich der verteilten Systeme.

Anstatt eine einzige große Anwendung einzusetzen, setzt man auf viele kleine, verteilte, autarke Anwendungen, welche als Dienste bezeichnet werden, ein. Diese bieten nach außen entsprechende Schnittstellen an, um den jeweiligen Dienst nutzen zu können. Dabei ist dieses Architekturmuster eine weitere Form der Modularisierung von Softwaresystemen, da einzelne Komponenten durch eigenständige Anwendungen abgebildet werden.

## 3.1 Was ist ein Service?

Ein Service ist in einem „Service-orientiertem System“, eine Anwendung mit bestimmten Aufgaben. Dabei wird versucht autarke Services zu erstellen, welche unabhängig von anderen Diensten verwaltet und verwendet werden können. Die Größe von Diensten ist dabei jedoch nicht festgelegt und kann je nach Paradigma bzw. Definition und Unternehmen variieren.

## 3.2 Verteilte Systeme

Damit Service-orientierte Architekturen verstanden werden können, müssen zunächst verteilte Systeme verstanden werden. *Andrew S. Tanenbaum* definiert ein verteiltes System wie folgt:

> »Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzer wie ein einzelnes kohärentes System erscheinen.«[1, S. 19]

Im Falle von Service-orientierten Architekturen wird das System auf mehrere eigenständige Computer bzw. Anwendungen aufgeteilt. Ein Vorteil von verteilten Systemen ist, dass sie zum einen sehr dynamisch und schnell anpassbar sind, zum anderen jedoch auch die Komplexität von Software in einzelne Teile zerbricht, wodurch eine entfernte Präsentation möglich ist.

Eines der wichtigsten und größten Probleme besteht dabei in der Kommunikation. Zum einen muss diese gewährleistet werden, zum anderen jedoch auch in angemessener Zeit erfolgen. Dabei muss ebenfalls darauf geachtet werden, dass Nachrichten erfolgreich zugestellt werden, selbst wenn einzelne Dienste temporär nicht erreichbar sind.

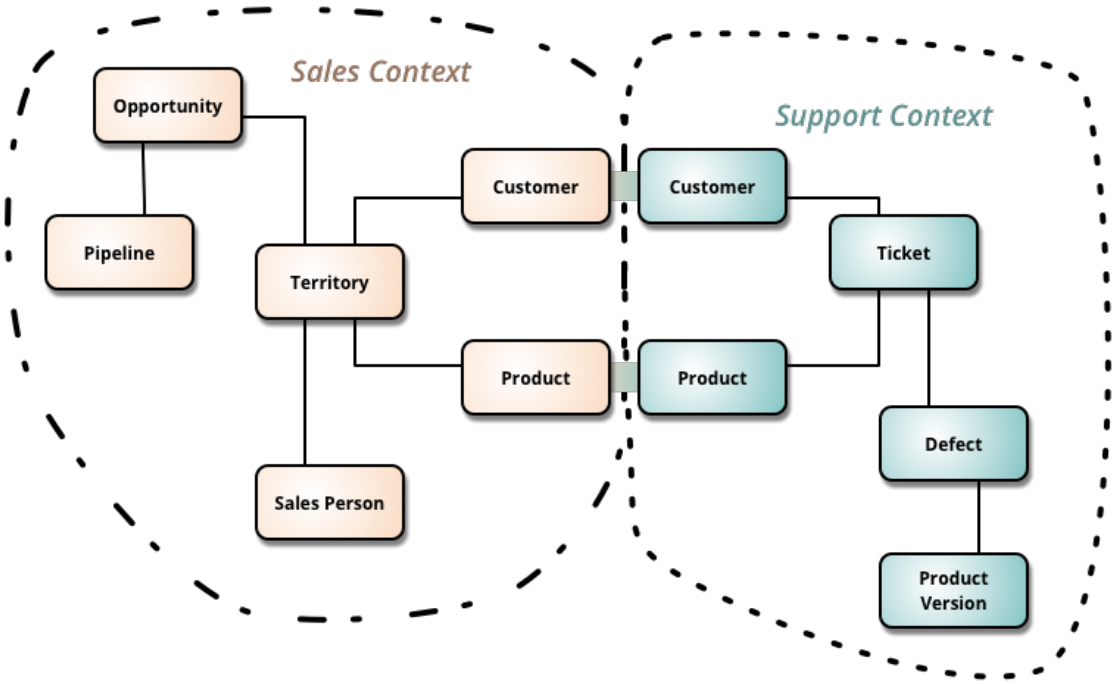
Ein weiteres Problem besteht darin, zu erkennen wann ein Dienst ausgefallen ist. Meistens erkennt man dies nur dadurch, dass ein Teilsystem nicht funktioniert. Das ausgefallene System zu identifizieren, stellt, wenn keine entsprechenden Vorkehrungen getroffen wurden, eine nicht zu unterschätzende Problematik da. Zudem kann eine lange Zeit vergehen, bis das Unternehmen feststellt, dass ein System ausgefallen ist.

### 3.2.1 Domain-Driven Design und Bounded Context

Domain-Driven Design (DDD) beschreibt dabei die Herangehensweise zur Modellierung von komplexer Software. Dabei ist die Modellierung maßgeblich an die umzusetzende Fachlichkeit gebunden und wird durch diese beeinflusst. Das Ziel jeglicher Software ist es, eine bestimmte Anwendungsdomäne zu unterstützen. Damit dies erfolgreich geschieht, muss Software harmonisch und in höchster Form interoperabel zur Anwendungsdomäne sein. Domain-Driven Design soll genau dies gewährleisten.

Arbeitet man mit Service-Orientierten Architekturen, versucht man fachliche Komponenten, welche zu einem bestimmten Kontext gehören, möglichst nahe beieinander zu halten. Man spricht hierbei von *Bounded Context*.

> »Bounded Context ist ein zentrales Muster in Domain-Driven Design. [..] DDD arbeitet mit großen Modellen, indem es diese in kleine verschiedene zusammengehörige Kontexte unterteilt und auf ihre Wechselwirkung unterteilt.« [3]




Quelle: http://martinfowler.com/bliki/BoundedContext.html

**Abbildung 3.1:** Bounded Context

In dieser Grafik wird noch einmal der Begriff Bounded Context genauer verdeutlicht. Es existieren zwei eigenständige Prozesse. Auf der linken Seite der Sales Kontext und auf der rechten Seite der Support Kontext. Jeder Kontext besitzt verschiedene Services, welche benötigt werden um den Prozess durchführen zu können. Lediglich zwischen den *Customer* und *Product* Services besteht eine Verbindung der beiden Prozesse.

### 3.2.2 Das Gesetzt von Conway

Spricht man von „Service-orientierten Architekturen“, sollte das „Gesetzt von Conway“ nicht fehlen, da es Prinzipien beschreibt, nach denen eine Unternehmens-Architektur entworfen wird.

Melvin Conway ist ein amerikanischer Informatiker und formulierte seine Beobachtungen bezüglich der Kommunikationsstrukturen und Organisationen innerhalb eines Unternehmens. Seine Beobachtung, auch „Gesetzt von Conway“ genannt lautet wie folgt:

> *Organisationen, die Systeme designen, können nur solche Designs entwerfen, welche die Kommunikationsstruktur dieser Organisationen abbilden.*

Conway möchte damit ausdrücken, dass die internen Kommunikationswege wichtig bei der Planung der Architektur sind. Jedes Team innerhalb einer Organisation trägt zu der Entwicklung der Architektur bei. Damit Teams untereinander kommunizieren können, sind Schnittstellen erforderlich. Dabei müssen Kommunikationswege nicht immer offiziell sein. Oft gibt es informelle Kommunikationsstrukturen, die ebenfalls in diesem Kontext betrachtet werden können.

Service-orientierte Systeme arbeiten nach dem gleichen Prinzip. Dienste in diesen Systemen sind eigenständig und müssen, damit daraus eine funktionierende Anwendung bzw. System wird, unter einander problemlos kommunizieren können.

## 3.3 Kommunikation: Orchestration vs Choreographie

### Orchestration

Bei der Orchestration handelt es sich um eine Komposition von Services. Ein Geschäftsprozess wird zwar mit Hilfe von mehreren Services abgebildet, jedoch ist nur ein Service dafür zuständig den Geschäftsprozess durchzuführen.

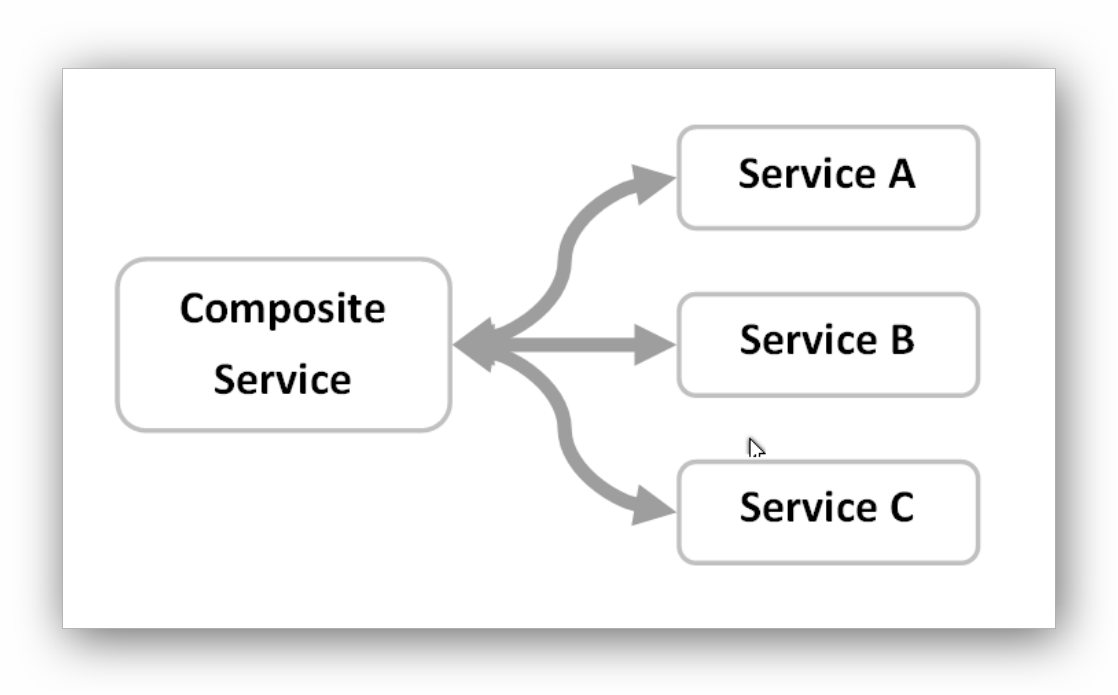



**Abbildung 3.2:** Orchestration

Wie die Abbildung 3.2 zeigt besteht bei der Orchestration **keine** Verbindung zwischen:

- A & B
- A & C
- B & C

Nur der „Composite Service“ nutzt die anderen Services, um den Geschäftsprozess abzubilden.

## Choreographie

Anders als bei der Orchestration können Services bei der Choreographie beliebig untereinander kommunizieren. Das ist sinnvoll, wenn verschiedene Dienste, sich untereinander über Änderungen oder andere Aktionen informieren müssen.

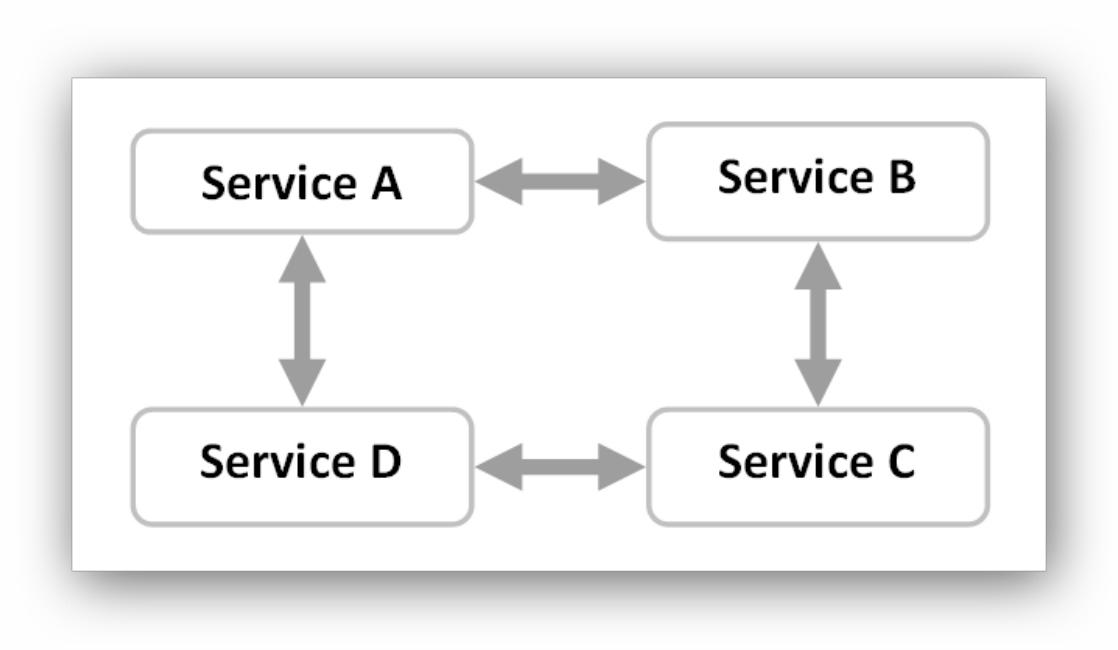



**Abbildung 3.3:** Choreographie

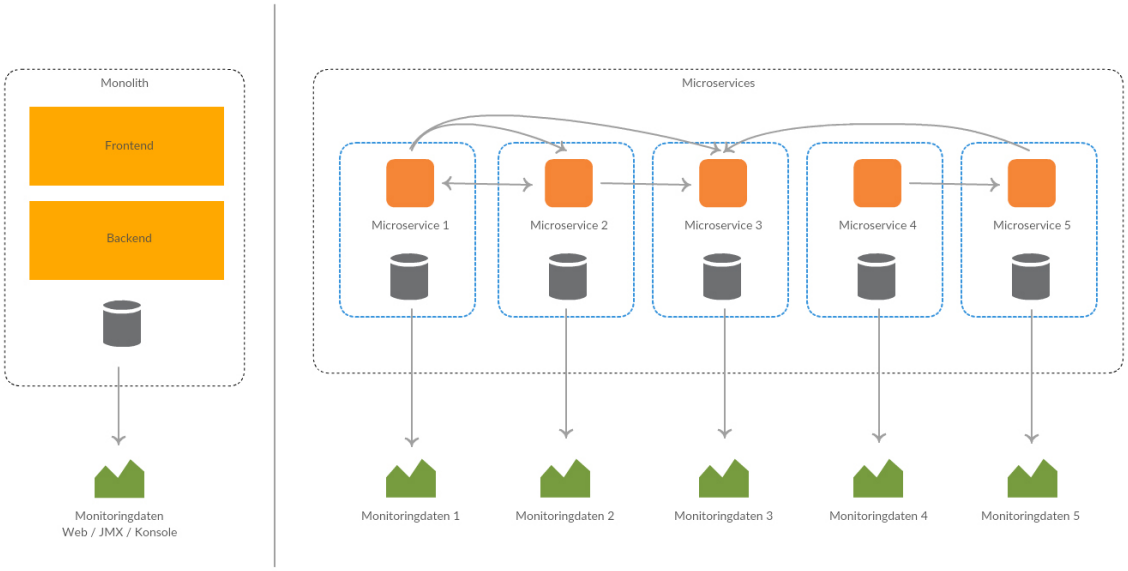
So ist, wie in Abbildung 3.3 zu erkennen, eine beliebige Kommunikation zwischen den einzelnen Diensten möglich.

## 3.4 Abgrenzung von monolithischen Systemen

Darauf aufbauend ist ersichtlich, dass im Vergleich zu monolithischen Systemen, bei dem Einsatz von Service-orientierten Systemen einiges beachtet werden muss. So ist die Kommunikation ein wichtiger Faktor. Man muss sich, unter anderem, nicht nur zwischen einer der beiden zuvor genannten Kommunikationsformen, Choreographie oder Orchestrierung, entscheiden, es muss zudem sichergestellt werden, dass Nachrichten zugestellt werden.

Während bei einem monolithischen System alle Abhängigkeiten innerhalb der Anwendung zu finden sind, besteht die Anwendung in einem Service-orientieren System aus mehreren Diensten, welche untereinander kommunizieren. Zudem ist es dadurch die Verwendung von Services möglich, dass jeder Dienst eine eigene Datenbank besitzt, wodurch verschiedene Datenbanksysteme (SQL und NoSQL) eingesetzt werden können. In der nachfolgenden Abbildung ist einem Monolithen, ein Microservice System entgegenge-

setzt. Zusätzlich zu den oben genannten unterschieden, sieht man in der Abbildung außer-




Quelle: https://jaxenter.de/wp-content/uploads/2016/10/fichtner_microservices_1.jpg

**Abbildung 3.4:** Monolithisches vs Service-orientiertes System

dem ein Monitoring-System. Während bei einem Monolithen, nur ein Monitoring-System benötigt wird, muss bei einem Service-orientiertem System, jeder Service ein Monitoring-System besitzen. Bei großen Systemen, müssen zudem die Monitoring-Informationen gebündelt werden und zentral einsehbar sein. Daraus ist ersichtlich, dass das Monitoring bei einer monolithischen Anwendung deutlich geringer ist, als in einem Service-orientieren System.

# Kapitel 4

# SOA

Der Begriff SOA ist nicht eindeutig definiert. Je nachdem welche Person man in einem Unternehmen befragt, erhält man unter Umständen eine komplett andere Definition. »Die meisten Definitionen stimmen jedoch zum größten Teil überein und stehen nicht im Konflikt mit einander.«[5, vgl. Seite 6]
Für einen Kaufmann ist SOA etwas Anderes als für einen Analysten. Um die unterschiedlichen Sichten auf SOA besser zu verstehen, werden zunächst einmal einige Definitionen nach [5] genannt:

1. »To the chief information officer (CIO), SOA is a journey that promises to reduce the lifetime cost of the application portfolio [...].«[5, vgl. Seite 6]

2. »To the business executive, SOA is a set of services that can be exposed to their customers, partners, and other parts of the organization. Business capabilities, function, and business logic can be combined and recombined to serve the needs of the business now and tomorrow. Applications serve the business because they are composed of services that can be quickly modified or redeployed in new business contexts, allowing the business to quickly respond to changing customer needs, business opportunities, and market conditions.«[5, vgl. Seite 6]

3. »To the business analyst, SOA is a way of unlocking value, because business processes are no longer locked in application silos. Applications no longer operate as inhibitors to changing business needs.«[5, vgl. Seite 6]

4. »To the chief architect or enterprise architect, SOA is a means to create dynamic,

> highly configurable and collaborative applications built for change. SOA reduces IT complexity and rigidity. SOA becomes the solution to stop the gradual entropy that makes applications brittle and difficult to change. SOA reduces lead times and costs because reduced complexity makes modifying and testing applications easier when they are structured using services.«[5, vgl. Seite 6]

Jeder der genannten Rollen hat eine eigene Definition von dem was SOA ist. Jede der Definitionen ist jedoch nur ein Teil dessen für was SOA verwendet werden kann, denn SOA ist ein Paradigma und kein festes Modell. Es dient zur Orientierung, gibt jedoch keine feste Richtung vor. Dabei ist das Ziel von SOA nicht die Entwicklung zu vereinfachen oder voran zu bringen, sondern die unternehmensweiten Geschäftsprozesse zu standardisieren und vereinheitlichen.

## 4.1 Business und IT

IT darf nie zum Selbstzweck existieren. Sie wird immer zur Unterstützung der Geschäftsprozesse eingesetzt. Oft existieren bereits verschiedene kommerzielle Systeme und Eigenentwicklungen. Mit SOA soll dafür gesorgt werden, dass diese Systeme möglichst effizient miteinander arbeiten können.

> »The complexity of the technology infrastructure at many companies in the financial services sector makes it very hard to leverage IT services in a coordinated way across the enterprise. Many large companies have either merged or acquired other very large companies resulting in the integration of new business units with very different work cultures and widely different information infrastructurse. The need to be able to trust and understand the information about the business across its many disaggregated parts has been a prime motivator for change in the IT infrastructure at these companies.«[4, S. 17]

Mit SOA wird die IT-Infrastruktur eines Unternehmens in zwei Teile geteilt. Auf der einen Seite existiert die Geschäftsschicht mit der Geschäftslogik und auf der anderen Seite die IT-Schicht, welche die Computing-Ressourcen verwaltet. Durch diesen Aufbau ist es nicht nötig, dass ein Business Manager die IT-Schicht verstehen muss.

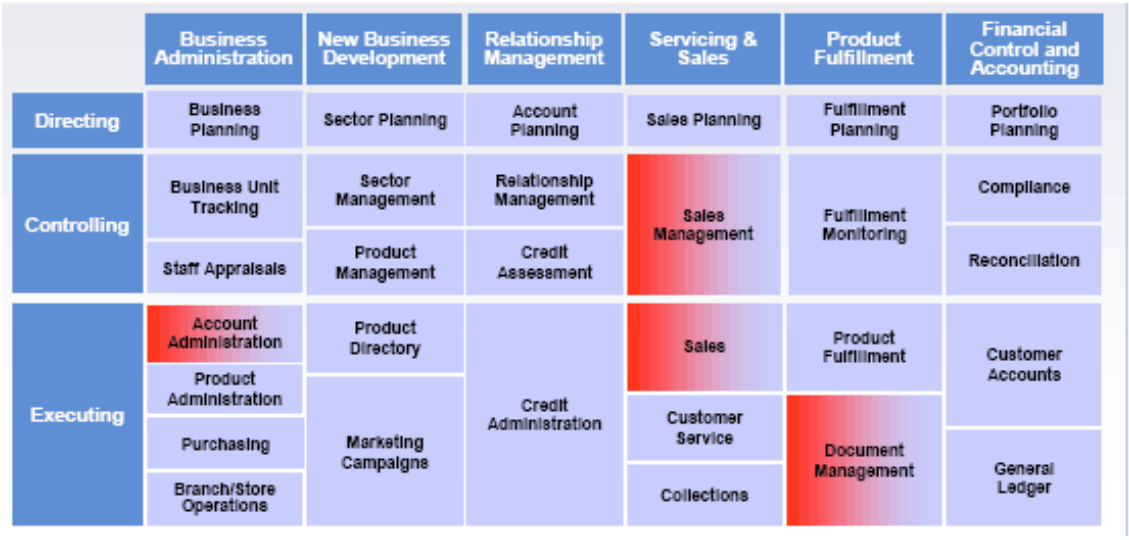
In der Geschäftsschicht sind nur Dienste, mit denen Kunden, Lieferanten und Business Partner interagieren. Diese Personen benötigen, genauso wie ein Business Manager, kein Wissen darüber, was in der IT-Schicht existiert oder wie diese aufgebaut ist. Andersherum sind in der IT-Schicht nur Dienste und Applikationen vorhanden, wofür die IT-Abteilung zuständig ist.

Damit diese Schichtentrennung funktioniert, wird darauf geachtet, dass in der Geschäftsschicht möglichst wenig Komplexität nach außen sichtbar ist.

## 4.2 Unternehmenskomponenten

Unternehmenskomponenten sind ein zentraler Bestandteil von SOA. Die Informationen aus den Komponenten, müssen durch APIs abgerufen werden können. Dienste in einem SOA-System sind nicht, wie anzunehmen, die Unternehmensanwendungen selber, sondern Adapter, welche für die Steuerung der jeweiligen Anwendungen verantwortlich sind. Dabei dient der Adapter nicht nur als API Exposure Gateway, sondern sorgt auch dafür, dass keine Fehlerhaften Anfragen an das System geschickt werden. Spricht man bei SOA von Diensten, ist meistens der Adapter, zusammen mit den Unternehmenskomponenten gemeint. Wird der Begriff „Dienst“ im Zusammenhang mit SOA verwendet, ist damit der Adapter und die Anwendung bzw. Unternehmenskomponente gemeint.

Damit die Geschäftsprozesse mit Hilfe von SOA abgebildet werden können, müssen daher zunächst einmal die dazu gehörenden Komponenten identifiziert werden. beginnend mit den Wichtigsten.




Quelle: http://www.jot.fm/issues/issue_2008_05/column5/

**Abbildung 4.1:** Unternehmens Komponenten

Daraus ergeben sich anschließend die Komponenten, welche untereinander kommunizieren müssen. Die in der Abbildung rot dargestellten Komponenten sind schließlich das Resultat aus der Analyse. Hat man die Komponenten über Kommunikationswege verbunden, werden die nächsten Geschäftsprozesse identifiziert. Diese Prozedur wird solange wiederholt, bis alle Komponenten miteinander verbunden sind.

## 4.3 Enterprise-Service Bus - ESB

Der Begriff „Enterprise-Service Bus“ (ESB) wird oft im Zusammenhang mit SOA verwendet. Der ESB ist jedoch kein Teil von SOA, sondern nur ein Mittel zum Zweck, um die Kommunikation zu vereinheitlichen und ein Exposure Gateway bereitzustellen. Der Enterprise-Service Bus dient dabei als Nachrichtenkomponente, welcher die Nachrichten entgegennimmt und an die jeweiligen Empfänger weiterleitet. Es dient ebenfalls als Transformator von Nachrichten, um die Interoperabilität sicherzustellen. Dabei liegen auf der einen Seite des ESB die Services und auf der anderen Seite die Anwendungen.

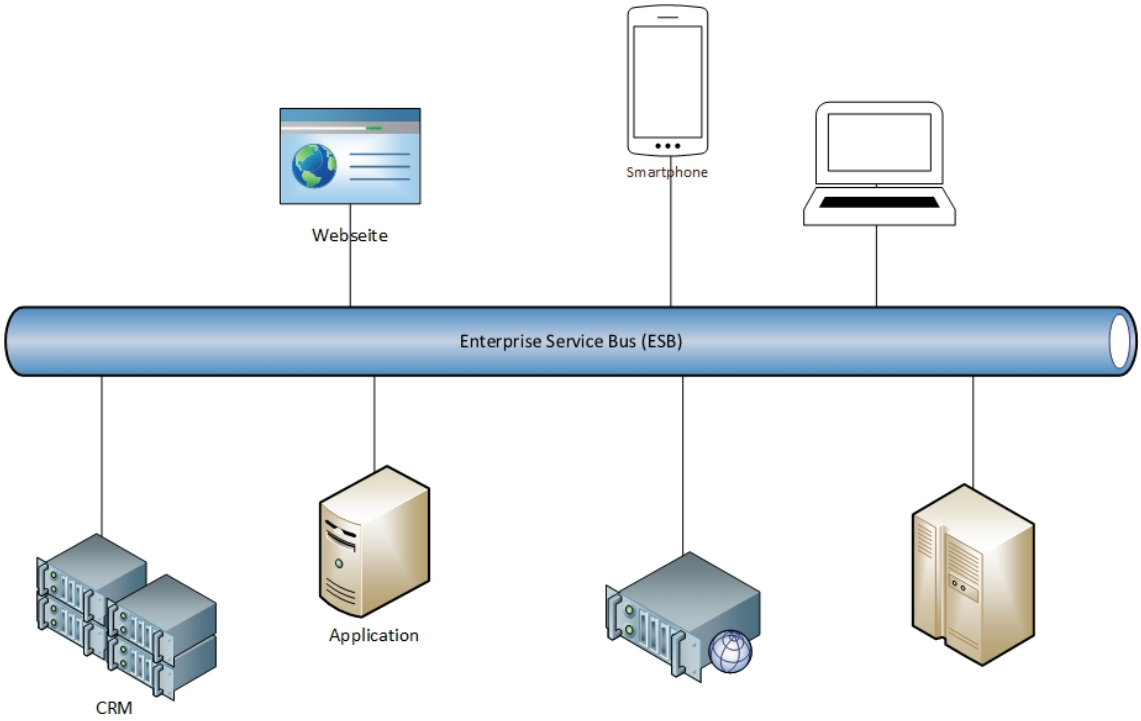



**Abbildung 4.2:** Enterprise Service Bus

Dienste bieten eine große Menge an Informationen über deren APIs an. Oft werden jedoch nicht alle Informationen benötigt. Gerade auf mobilen Geräten können viele Daten zu langen Ladezeiten führen. Der ESB filtert und bereitet die Daten der Dienste auf und sendet nur die benötigten Daten. Dadurch werden Anwendungen und Anwender nicht mit unwichtigen Daten überfordert.

# Kapitel 5

# Microservices

Ein Microservices-System besteht aus vielen verteilten Anwendungen (Dienste), welche man selber ebenfalls Microservices nennt. Dabei wird versucht, komplexe Anwendungen, in einzelne Dienste aufzuteilen.

> »Modularisierung ist nichts Neues. Schon lange werden große Systeme in kleine Module unterteilt, um Software einfacher zu erstellen, zu verstehen und weiterzuentwickeln. Das Neue: Microservices nutzen als Module einzelne Programme, die als eigene Prozesse laufen. Der Ansatz basiert auf der UNIX-Philosophie. Sie lässt sich auf drei Aspekte reduzieren:«[7, S. 2]

Weiter nach [7, S. 2]:

- Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen.
- Programme sollen zusammenarbeiten können.
- Nutze eine universelle Schnittstelle. In UNIX sind das beispielsweise Textströme.

Diese Art der Aufteilung wurde schon lange von großen Unternehmen wie Amazon oder Google genutzt, jedoch fehlte lange Zeit ein Begriff für dieses Paradigma.

> »2006 hielt Werner Vogels (CTO, Amazon) einen Vortrag auf der JAOO-Konferenz, wo er die Amazon Cloud und Amazons partnermodell vorstellte [1]. Dabei erwähnte er das CAP-Theorem - heute Basis für NoSQL. Und

> dann sprach er von kleinen Teams, die Services mit eigenener Datenbank entwickeln und auch betreiben. Diese Organisationen nennen wir heute DevOps und die Architektur Microservices.« [7, S. 1]

Der Begriff Microservices ist nicht eindeutig definiert. Als erste Näherung dienen, nach Eberhard Wolff [7, S. 2], folgende Kriterien:

- Microservices ist ein Modularisierungskonzept. Sie dienen dazu, ein großes Software-System aufzuteilen - und beeinflussen die Organisation und die Software-Entwicklungsprozesse.
- Microservices können unabhängig von Änderungen an anderen Microservices in Produktion gebracht werden.
- Microservices können in unterschiedlichen Technologien implementiert sein. Es gibt keine Einschränkung auf eine bestimmte Programmiersprache oder Plattform.
- Microservices haben einen eigenen Datenhaushalt: eine eigene Datenbank - oder ein vollständig getrenntes Schema in einer gemeinsamen Datenbank.
- Microservices können eigene Unterstützungsdienste mitbringen, beispielsweise eine Suchmaschine oder eine spezielle Datenbank. Natürlich gibt es eine gemeinsame Basis für alle Microservices - beispielsweise die Ausführung virtueller Maschinen.
- Microservices sind eigenständige Prozesse - oder virtuelle Maschinen, um auch die Unterstützungsdienste mitzubringen.
- Dementsprechend müssen Microservices über das Netzwerk kommunizieren. Dazu nutzen Microservices Protokolle, die lose Kopplung unterstützen. Das kann beispielsweise RE ST sein - oder Messaging-Lösungen.

Grundsätzlich kann man Microservices in drei Kategorien einteilen:

**Producer** Ein Service der etwas produziert oder auf eine Anfrage reagiert. Das reicht von Daten aus einer Datenbank zu extrahieren bis hin zu komplexen Berechnungen.

**Consumer** Ein Service oder eine Anwendung, welche einen oder mehrere produzierende Services verwendet und entweder weiterverarbeitet oder ausgibt. Im Falle der Weiterverarbeitung ist ein Consumer ebenfalls ein Producer.

**Self-Contained System (SCS)** »„Microservice mit UI“ oder “Self-Contained System“ wie es Stefan Tilkov nennt, sind in sich abgeschlossene Systeme. [..] Sie enthalten eine UI und sollten möglichst nicht mit anderen SCS kommunizieren.«[7, vgl S. 55].

## 5.1 Aufbau von Microservices

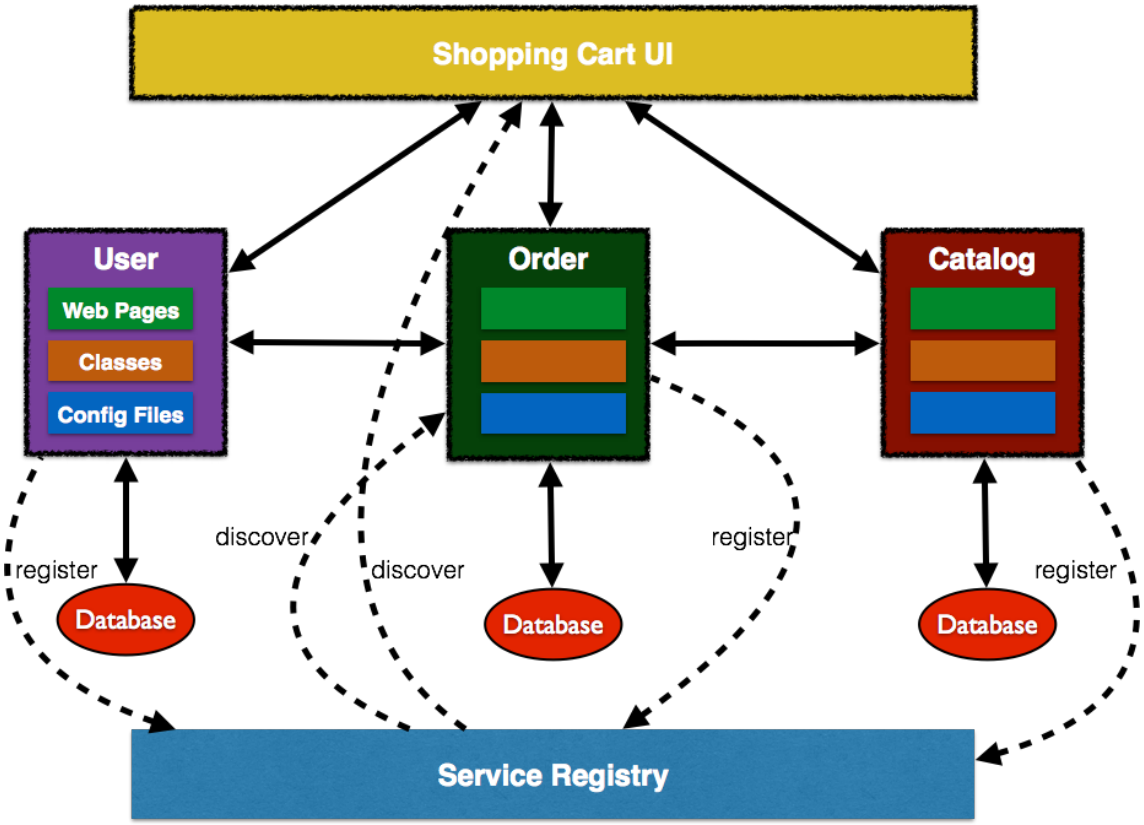
Wie bereits erwähnt, sind Microservices eigenständige Dienste. Sie müssen so gebaut sein, dass sie möglichst ohne Abhängigkeiten funktionieren. Um das zu realisieren dürfen Microservices nur lose gekoppelt sein. Damit ist gemeint, dass der Dienst seine Aufgabe selbständig erledigen kann ohne weiter Dienste in Anspruch zu nehmen. Bei „Self-Contained Systems“ ist ebenfalls eine UI vorhanden. Damit ist das System in sich abgeschlossen und stellt alle nötigen Eigenschaften bereit, welche für die Darstellung von Informationen benötigt werden. Müssen jedoch andere Dienste verwendet werden, sollte darauf geachtet werden, dass diese austauschbar bleiben und nicht fest mit miteinander verbunden sind. Ist ein Dienst fest mit einem anderen verbunden, sollte die Notwendigkeit des Dienstes infrage gestellt werden.

In direkter Konkurrenz stehen konsumierende Dienste, welche produzierende Dienste verwenden müssen, um die eigenen Aufgaben fertig zu stellen. Konsumierende Dienste werden häufig als Middleware verwendet, um mehrere Daten aufzubereiten. Ein Konsumierender Dienst könnte zum Beispiel die Aufgabe haben eine Webseite darzustellen oder Informationen zu aggregieren.

Damit ein Dienst selbständig bleibt, besitzt dieser alle dafür notwendigen Ressourcen. Hierzu zählt, unter anderem, eine eigene Datenbank. Dadurch kann das Schema der Datenbank frei verändert werden, ohne, dass weitere Teile des Systems beeinflusst werden. Es ist jedoch darauf zu achten, dass die Schnittstelle des Dienstes, worüber dieser von anderen Diensten ansprechbar ist, nicht grundlegend verändert werden oder die Dienste, welche diese Schnittstelle verwenden, ebenfalls geändert werden. Letzteres ist beispielsweise der Fall, wenn sich Fachlichkeiten des Systemes ändern, da diese in einigen Fällen, über mehrere Microservices verteilt sind. Jedoch besteht hierbei weiterhin die Gefahr,

dass einzelne Dienste, die Schnittstelle nutzen, um einzelne Informationen abzurufen, jedoch nichts mit den Fachlichkeiten zu tun hat. In diesem Fall ist zu beachten, dass die Schnittstelle weiterhin das liefert, was zu erwarten ist.

Geht man von *Self-Contained Systems* aus, besitzen Microservices zusätzlich noch eigene UI-Elemente. Dadurch kann der Microservice, seine Informationen selber darstellen und damit bestimmen, wie diese dargestellt werden.




Quelle: http://blog.arungupta.me/monolithic-microservices-refactoring-javaee-applications/

**Abbildung 5.1:** Microservice Architektur

## 5.2 Größe von Microservices

»Der Name „Microservices“ verrät schon, dass es um die Servicegröße geht - offensichtlich sollen die Services klein sein."[7, S. 31] Es gibt verschiedene Möglichkeiten die Größe von Programmen zu ermitteln. Eine Variante

> ist zum Beispiel das Zählen von Lines of Code (LOC), jedoch hat diese Methode auch Nachteile. Denn die Anzahl der Codezeilen hängen stark von der verwendeten Programmiersprache ab. Einige Programmiersprachen benötigen mehr Zeilen Code, um eine bestimmte Tätigkeit abzubilden, als andere.
> Die Größe von Services sollte jedoch nicht von zentraler Bedeutung sein, denn eine untere Grenze gibt es für Services nicht. "Wohl aber eine obere Grenze: Wenn der Microservice so groß ist, dass er von einem Team nicht mehr weiterentwickelt werden kann, ist sie zu groß. Ein Team sollte dabei eine Größe haben, wie sie für agile Prozesse besonders gut funktioniert. Das sind typischerweise drei bis neun Personen.«[7, S. 34]

Bei der Größe eines Services ist darauf zu achten, dass ein Service nicht zu viele oder zu wenige Funktionen besitzt. Wie bereits beschrieben, sind Microservices modulare, lose gekoppelte Services. Wird ein Service zu klein angesetzt, können daraus Abhängigkeiten zu anderen Services entstehen und damit das Gesetzt der losen Kopplung verletzten. Besitzt hingegen ein Service zu viele Funktionen, wird es meistens nicht mehr als Microservice angesehen, da es nicht eine, sondern mehrere Aufgaben übernimmt und diese wahrscheinlich nicht mehr gut erledigen kann.

## 5.3 Orchestration vs Choreographie

Möchte man ein Microservice System aufbauen, stellt sich die Frage, wie einzelne Services strukturiert werden und wie diese untereinander kommunizieren sollen. Ein bestimmter Vorgang startet in der Regel bei einem Service. Nun muss entschieden werden, ob weitere Services hinzugezogen, beziehungsweise informiert werden müssen. Je nach Anwendungsfall muss man sich zwischen Service Orchestration und Choreographie entscheiden. Dabei ist es fast unmöglich ein ganzes Microservice-System aus nur einem der beiden Varianten zu bauen.

### 5.3.1 Herausforderung

Der Ausfall eines Services kann im schlechtesten Fall dazu führen, dass alle anderen Microservices nicht mehr funktionieren. Um das zu verhindern muss klar definiert werden,

was Microservices in dieser Situation tun sollen. Zusätzlich muss, sofern Datenbankoperationen eine wichtige Rolle spielen, das Problem der einheitlichen Transaktion gelöst werden. Wenn zum Beispiel eine Operation Daten über verschiedene Microservices in Datenbanken schreibt, muss bei nicht Erreichbarkeit oder dem Auftreten eines Fehlers in einem Microservices ein einheitliches Rollback durchgeführt werden, um keine inkonsistente Dateien im System zu erzeugen.

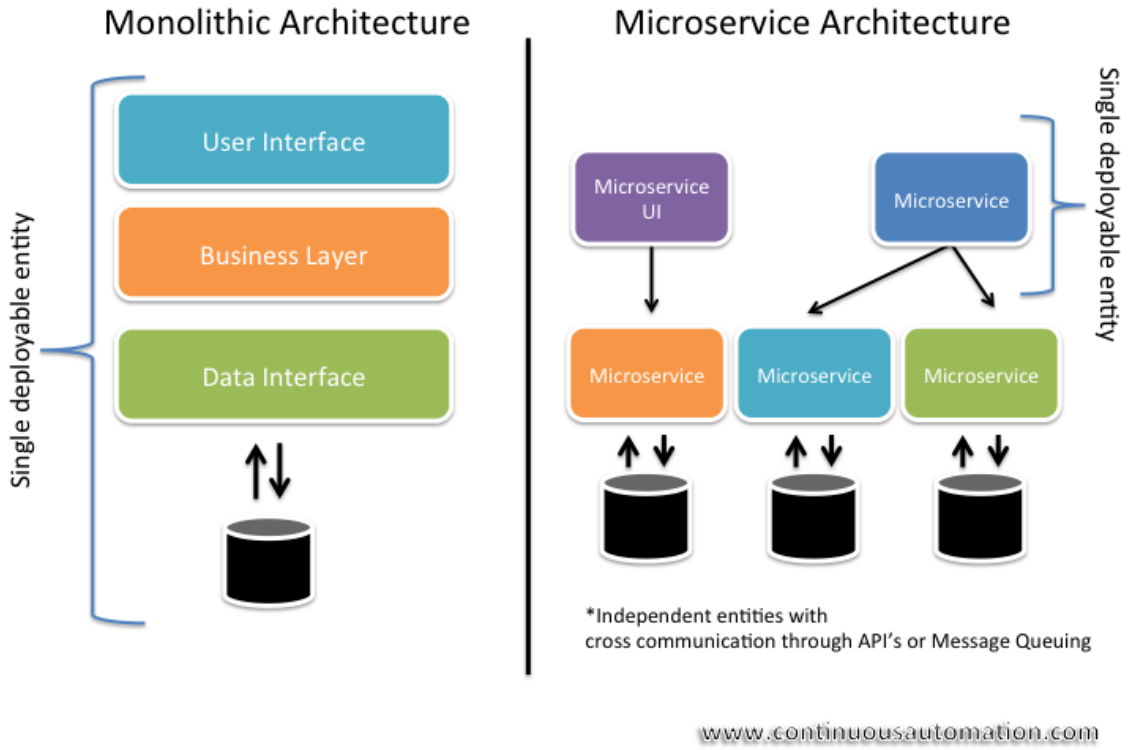
> »Transaktion haben die ACID-Eigenschaften. [..] Innerhalb eines Microservice können Änderungen in einer Transaktion stattfinden. Ebenso kann die Konsistenz der Daten in einem Microservice sehr einfach garantiert werden. Über einen Microservice hinaus wird das schwierig. Dann ist eine übergreifende Koordination notwendig. Bei einem Zurückrollen einer Transaktion müssten alle Änderungen in allen Microservices rückgängig gemacht werden.« [7, S. 35]

Daneben muss sichergestellt werden, dass Nachrichten erfolgreich verarbeitet werden und auch bei mehrfachen senden einer Nachricht keine Fehler auftreten. Dafür müssen Microservices Idempotent sein. Das Heißt, dass eine Wiederholte Aktion (mit gleichen Daten) nicht zu unterschiedlichen Resultaten führt. Wird keine Idempotenz sichergestellt, kann dies zu unerwarteten und unerwünschten Ereignissen führen.

Eine weitere Herausforderung besteht in dem Grundkonzept von Microservices. Da nicht definiert ist, welche Programmiersprache für Microservices verwendet wird, kann ein Service zum Beispiel in Java, ein anderes in Scala oder Python geschrieben werden. Es muss daher dafür gesorgt werden, dass die einzelnen Services untereinander interoperabel sind. Um das zu gewährleisten, müssen die Schnittstellen möglichst einheitlich und auf dem gleichen Protokoll aufbauend programmiert werden. Hier bieten REST-Schnittstellen eine gute Lösung. Diese beruhen auf dem HTTP-Protokoll. Zusätzlich bietet das HTTP-Protokoll die Möglichkeit, ein einheitliches Medium, wie zum Beispiel XML oder JSON, als Informationsträger zu nutzen. Dabei kann theoretisch jeder Microservice mit jedem anderen Microservice kommunizieren, sofern die Schnittstellen einheitlich definiert sind.

Damit die Schnittstellen genutzt werden können, müssen die Kommunikationskanäle da-

für existieren. Nach dem Gesetz von Conway, bedeutet dies, dass die einzelnen Abteilungen bzw. Teams die nötigen Kanäle schaffen müssen.




Quelle:
https://dzone.com/articles/scalable-cloud-computing-with-microservices

**Abbildung 5.2:** Monolithisch vs Microservice Architektur

Im Unterschied zu monolithischen Anwendungen, bei der alles in einer Anwendung ist, sind Microservices Module eines gesamten Systems und werden von den jeweiligen Teams entwickelt und verwaltet. Wird eine neue Funktion benötigt, welche nicht in der eigenen Fachlichkeit liegt, muss das entsprechende Team damit beauftragt werden, diese zu implementieren, bzw. in einem bestehenden Microservice anzupassen.

## 5.4 PUSH- VS PULL-Architektur

Grundlegend können Microservices mit Hilfe zwei verschiedener Kommunikations-Architekturen kommunizieren, PUSH- und PULL-Architektur. Dabei ist jedoch nicht ausgeschlossen, dass sobald eine Architektur gewählt worden ist, die andere nicht mehr genutzt werden kann. Genauso wie bei der Entscheidung über die Kommunikationsstruktur (siehe 3.3 Orchestration), kann es von Vorteil sein, beide Architekturen zu nutzen.

**PULL-Architektur**

Eine PULL-Architektur basiert auf einen einfachen Request-Replay-Schema. Dementsprechend ist das Web PULL-basiert. Der Browser macht eine Anfrage an einen Server, dieser wiederum verarbeitet die Anfrage und liefert eine Antwort (Replay) zurück. Dies hat den Vorteil, dass nicht lange auf eine Antwort gewartet werden muss und sich die teilhabenden Kommunikationspartner gegenseitig kennen. Ein Nachteil jedoch ist es, dass dadurch weitgehend eine synchrone Kommunikation stattfindet und eine Antwort häufig nicht gleichzeitig an mehrere Empfänger senden kann.

**PUSH-Architektur**

Eine PUSH-Architektur wird eingesetzt, wenn man verschiedene Kommunikationspartner über bestimmte Ereignisse informieren möchte oder eine Rückantwort ist nicht erforderlich. Zum Beispiel muss eine Registrierung in unserem fiktiven Unternehmen möglich sein. Dabei sendet der Microservice der für die Registrierung zuständig ist eine einfache Event-Nachricht, wie Benutzer XY hat sich Registriert. Im Hintergrund erhalten anderer Microservices diese Nachricht und führen zusätzliche Aktionen durch, wie das Erstellen des Warenkorbs oder das Anlegen des Profils.

Bei der PUSH-Architektur stehen meist nicht die Kommunikationspartner, sondern die Informationen im Vordergrund. Dafür wird meistens ein eigenständiger Service (Broadcaster) eingesetzt, der die Verteilung dieser Informationen übernimmt. Dabei kann ein Service als Informationsprovider dienen, zum Beispiel ein Nachrichten-Feed (von einer Nachrichtenseite). Alle anderen Services abonnieren den Broadcaster und erhalten dadurch alle Nachrichten, die der Informationsprovider sendet.

Es gibt jedoch auch den Fall, dass die Kommunikation sternförmig um den Broadcaster angeordnet ist. Dadurch ist jeder Service der diesen abonniert, sowohl Provider, als auch Consumer. Anders als bei PULL-basierten Systemen kann hier nicht durchgehend sichergestellt werden, dass alle Nachrichten von allen Konsumenten gleichzeitig gelesen und ggf. verarbeitet werden. Jedoch können so Informationen innerhalb eines Microservice-Systems relativ zuverlässig verteilt werden.

Der Vorteil von PUSH-Architekturen ist, dass eine asynchrone Informationsverbreitung aufgebaut werden kann. Zudem können Serviceausfälle, solange es nicht der Broadcaster oder wichtige Microservices sind, überbrückt werden, indem der Broadcaster die Nachrichten für eine bestimmte Zeit vorhält. Dadurch kann ein ausgefallener Microservice, die Nachricht zu einem späteren Zeitpunkt entgegennehmen und verarbeiten.

# Kapitel 6

# Gemeinsamkeiten und Unterschiede von SOA und Microservices

## 6.1 Einsatzgebiet

Sowohl SOA, als auch Microservices sind Paradigmen aus der Service-orientierten Architektur, verfolgen jedoch unterschiedliche Ansätze. Während das Microservice-Paradigma die Softwareentwicklungsabteilung unterstützen soll, soll mit Hilfe von SOA die Kommunikation zwischen den Anwendungen und den Abteilungen standardisiert und optimiert werden.

Mit Hilfe des Microservice Paradigmas wird die Struktur einer Anwendung in mehrere, kleinere Dienste aufgeteilt. Dadurch ist es möglich neue Technologien einzusetzen und gegebenenfalls zu ersetzten, sollten diese nicht den gewünschten Erfolg bringen.

Anders sieht es mit SOA aus. Hierbei werden Adapter für Anwendungen, wie Kommerzielle Systeme oder Eigenentwicklungen, entwickelt, welche zusammen mit der Anwendung als Dienst dienen. Der Adapter stellt eine Schnittstelle für die Funktionalität der Anwendung bereit und übernimmt das steuern der Anwendung und das abfangen von Fehleingaben.

## 6.2 Architekturen und Schnittstellen

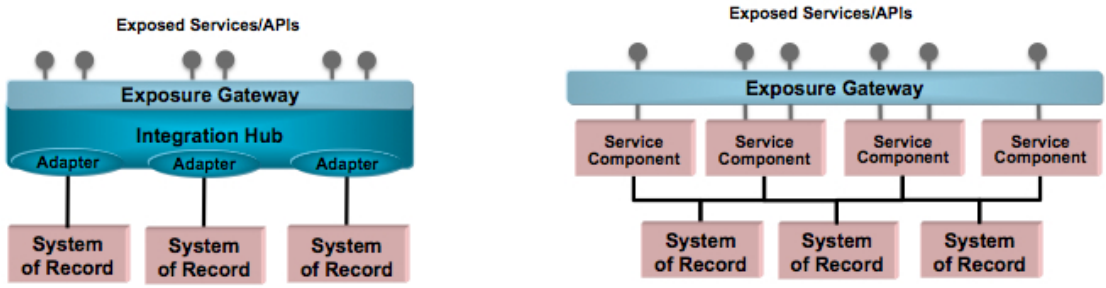
Beide Paradigmen sind Service-orientierte Systeme, welche auf ein verteiltes System setzten. Dabei existiert eine beliebige Anzahl verschiedener Dienste, welche verschiedene Funktionalitäten bereitstellen. Damit der Ausfall eines Dienstes, keine weiteren Dienste direkt beeinflusst, werden diese oft in eigenen Umgebungen betrieben, wie Virtuelle Maschinen. Dies ist sowohl bei SOA, als auch bei Microservices der Fall, jedoch kann durch den Ausfall eines Dienstes die Kommunikation von anderen Diensten gestört werden.

Während bei SOA eine zentrale Kommunikationseinheit (der ESB) existiert, sind Microservices in der Kommunikation frei. Dies bedeutet, dass der ESB in einem SOA-System entsprechende Werte oder Fehlermeldungen zurückgeben kann, während Microservices selber auf fehlgeschlagene Anfragen reagieren muss. Um dies zu verhindern, werden oft mehrere Instanzen eines kritischen Dienstes betrieben.

Damit Dienste untereinander kommunizieren können, sind Schnittstellen (APIs) notwendig, über welche die Informationen abgefragt bzw. bereitgestellt werden können. Da jedoch in einem SOA-System ganze Applikationen verwendet werden, ist es notwendig zunächst einmal zu entscheiden, wie dessen Funktionalitäten angeboten werden sollen.

Zum einen können Adapter verwendet werden, die jegliche Informationen der Anwendung bereitstellen. Dadurch ist es möglich durch Standardisierte Verfahren wie SOAP/HTTP oder REST/HTTP den Dienst wiederzuverwenden. Jedoch muss dafür neben den Adaptern ein Integrations Hub (ESB) existieren, welche die Informationen verarbeiten und aufbereiten kann.

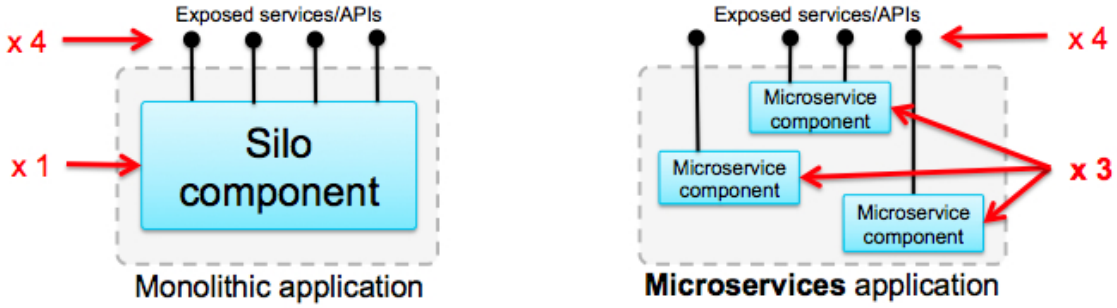
Zum anderen bedeutet diese Aufteilung, dass oft eine große Menge an Informationen abgefragt werden. Dies macht gegenüber dem Business kein Sinn, da zum Teil unnötig viele Informationen für die nächste Generation von Anwendungen bereitgestellt werden. In diesem Fall werden „Service Komponenten“ verwendet, welche als Adapter dienen und die benötigten Informationen aus den Applikationen abfragen und bereitstellen.
Bei einem Microservice System stellt ein Microservice genau eine Fachlichkeit da und bietet diese über Schnittstellen an. Anders als bei SOA sind die Dienste deutlich kleiner,




Quelle: https://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html

**Abbildung 6.1:** Technische und Funktionale Sicht von SOA

wodurch weniger Informationen bereitgestellt werden, je Microservice.




Quelle: https://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html

**Abbildung 6.2:** Monolithische vs Microservice Applikation

Dies bedeutet jedoch auch, dass weitere Microservices notwendig sind, welche die Informationen von verschiedenen Microservices abfragen und zusammenfassen. Dadurch ergibt sich ein Geflecht aus Microservices, welche untereinander kommunizieren und zum Teil aufeinander angewiesen sind. Solange sich die Schnittstelle eines Dienstes nicht ändert, besteht kein Problem bei der Interoperabilität für das bestehende System. Erst wenn sich die Schnittstelle ändert und eine ältere Schnittstelle nicht mehr angeboten wird, schadet dies der Interoperabilität.

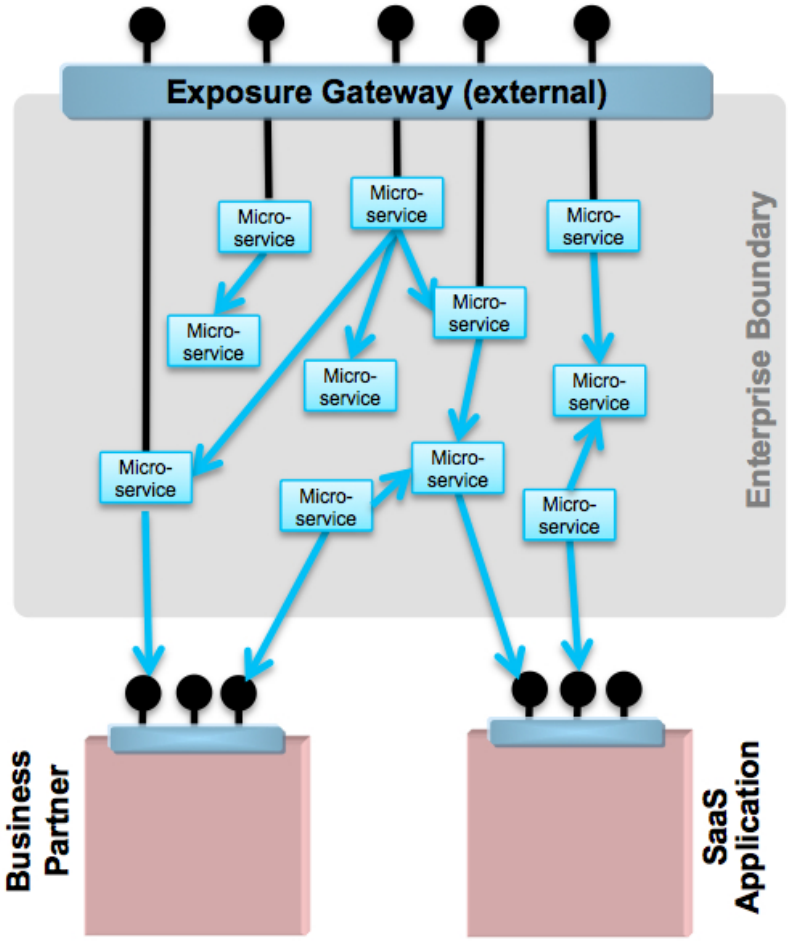
Dadurch, dass die Kommunikation über Schnittstellen erfolgt, kann, sofern diese Standardisiert sind und Beispielsweise als SOAP/HTTP oder REST/HTTP umgesetzt wurden, der Dienst Plattform unabhängig deployed werden. Da jedoch monolithische Anwendungen oft sehr umfangreich sind, werden diese meistens für ein bestimmtes Betriebssystem gebaut. Microservices hingegen sind kleine autarke Dienste, Dadurch ist es möglich diese Plattform unabhängig zu deployen. Hier drauf wird in weiter unten, unter 6.3.5 Bindung an Technologie Stacks, genauer eingegangen.

## 6.3 Vor- und Nachteile

### 6.3.1 Prozessisolierung

Ein wichtiger Grundsatz von Service-orientierten Systemen ist, dass Dienste autark bleiben. Um dies zu erreichen, darf nur eine lose Kopplung zwischen einzelne Dienste stattfinden. Dies wird durch Zustandslose Schnittstellen und Eigenständigkeit erreicht. Damit ist gemeint, dass jeder Dienst seine Aufgaben, zum größten Teil alleine durchführen kann ohne auf andere Dienste angewiesen zu sein.

Eine Vollständige Autarkheit ist in einem Microservice System nicht möglich, da ein vollständiges Microservice System die Abbildung einer Applikation ist. Dabei wurden einzelne Fachlichkeiten in Microservices verpackt. Dadurch können einige Microservices nicht lose gekoppelt sein, sondern benötigen weitere Microservices.




Quelle: https://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html

**Abbildung 6.3:** Microservice Architektur

Bei SOA hingegen besteht ein Dienst, wie bereits in 4.2 Unternehmenskomponenten erwähnt, aus einer vollständigen Anwendung und einem Adapter, welcher die Steuerung der Anwendung über APIs bereitstellt. Dadurch, dass eine Anwendung alle nötigen Fachlichkeiten und Ressourcen, für die Ausführung der Anwendung besitzt, ist der Dienst vollständig Autark. Lediglich die Frontend-Anwendungen sind auf die APIs und damit auf die Backend-Anwendungen angewiesen. Fällt zum Beispiel eine Anwendung aus, kann es dazu führen, dass das gesamte System nicht mehr funktioniert, da die Dienste häufig in eine Wertschöpfungskette eingebunden sind.

### 6.3.2 Deployment

Das Deployment ist in einem Service-orientiertem System von entscheidender Bedeutung und darf nicht unterschätzt werden. Oft werden Continuous Deployment Pipelines aufgebaut, um diesen Prozess zu automatisieren. Innerhalb der Pipeline wird sowohl die Umgebung, in welcher der Dienst später laufen soll, aufgebaut, sowie der Dienst in die zuvor, vorbereitete Umgebung deployed. Unter Umständen ist ein erneutes Deployen notwendig, wenn sich die Anforderungen an einen Dienst geändert haben oder Fehler behoben wurden. Dabei stellt das Deployment eine besondere Herausforderung dar. Hauptproblem ist die nötigen Abhängigkeiten, für die Ausführung der Dienste, bereitzustellen und die nötigen Konfigurationen vorzunehmen.

Dadurch, dass Microservices kleine autarke Dienste sind, ist es oft nicht nötig, viele Konfigurationsdateien anzupassen bzw. bereitzustellen. Jedoch sind oft eine Vielzahl von Microservices in einem System vorhanden und müssen deployed, sowie gemanagt werden. Durch Werkzeuge wie Docker und Vagrant wird das Deployment zusätzlich vereinfacht.

Dienste in einem SOA-System hingegen sind häufig ganze Anwendungen. Diese erfordern oft eine aufwändige Konfiguration, welche nicht immer mit Hilfe von Konfigurationsdateien vorgenommen werden können. Einige Anwendungen dürfen nur durch den Hersteller oder durch ihn Zertifizierte Unternehmen Installiert und/oder Konfiguriert werden. Dadurch ist ein automatisches Deployment nicht möglich und auch ein manuelles Deployment wird erschwert.

Im Rahmen einer Wartung müssen Dienste, oft erneut ausgeliefert werden, um Fehler zu beheben oder ihn anzupassen. In diesem Falle existieren in der Regel die Umgebungen, für den Einsatz des Dienstes. Jedoch müssen diese gegebenenfalls angepasst werden. Die ist in einem Microservice-System durch Werkzeuge wie Docker ebenfalls standardisiert. In einem SOA-System hingegen muss dies oft manuell durchgeführt werden.

### 6.3.3 Skalierung

Nicht immer können stark genutzte Anwendungen, durch Aufwertung oder Erweiterung von Hardware, belastbarer gemacht werden. Kritische oder Zentrale Anwendungen eines Unternehmens sind nicht nur stark genutzt, sondern dürfen nicht ausfallen. Durch Hardware, kann nicht durchgängig sichergestellt werden, dass eine Anwendung erreichbar ist. Um diese Problematiken entgegen zu wirken, muss Skaliert werden. Dabei werden mehrere Instanzen eines Systems parallel betrieben.

In einem Microservice-System, ist durch das vereinfachte Deployment möglich, in kürzester Zeit eine weitere Instanz eines Dienstes zu starten und zu betreiben. Dies kann durch einen Standardisierten Deployment Prozess automatisiert werden. Dadurch kann in kürzester Zeit auf verschiedene Lasten von Diensten reagiert werden, ohne dass ein Mensch in das System eingreifen muss. Es ist bei einem Microservice-System möglich, wenig genutzte Dienste, nur bei Bedarf zu starten und nach einer gewissen Zeit der Inaktivität, zu stoppen. Kritische oder stark genutzte Dienste hingegen, besitzen meistens mehrere Instanzen, auf denen die Lasten verteilt werden. Dadurch wird ebenfalls eine Ausfallsicherheit gewährleistet.

SOA-Systeme besitzen deutlich größere Dienste als Microservice-Systeme. Dadurch, dass, wie bereits unter 6.3.2 Deployment erwähnt, der Deployment Prozess nicht Standardisiert ist, ist ein einfaches, automatisches Skalieren nicht möglich. Das Skalieren eines Dienstes kann dementsprechend, nur manuell geschehen und ist oft mit viel Aufwand verbunden. Muss ein Dienst dennoch mehrere Instanzen besitzen, so werden diese meistens dauerhaft, parallel betrieben.

Viele monolithische Anwendungen sind zudem nicht Zustandslos, wodurch ein Skalieren zusätzlich erschwert wird. Microservices hingegen müssen so erstellt werden, dass sie Zustandslos sind, da ein Dienst nicht sicherstellen kann, dass die Anfragen vom gleichen Dienst oder Benutzer stammen.

### 6.3.4 Wartbarkeit

Im Laufe der Zeit wachsen die Anforderungen an einen Dienst, wodurch dieser angepasst, verbessert oder erweitert werden muss. Zur Wartung gehört jedoch auch das korrigieren von Fehlern, welches einer der einfachsten Aufgaben in einem Service-orientierten System ist, da sich dadurch meistens, nicht die Schnittstellen, Umgebung oder die Konfiguration des Dienstes ändert. Anders sieht dies jedoch beim Anpassen, Verbessern oder Erweitern aus.

Das Anpassen einer Anwendung kann sich von dem Verbessern oder Erweitern einer Anwendung unterscheiden. Zum Beispiel reicht es aus die Mehrwertsteuer in einer Anwendung zu ändern, welches eine Anpassung darstellt, jedoch keine Verbesserung oder Erweiterung. Das Anpassen ist durch die kaum veränderten Strukturen relativ leicht. Anders sieht dies jedoch bei Verbesserungen oder Erweiterungen aus. Beide Änderungen sorgen für eine Veränderung der internen Struktur des Quelltextes. Zusätzlich ist man an den vorgegebenen Technologie Stacks gebunden. Oft sind Verbesserungen oder Erweiterungen nicht ohne das Verändern der Quelltextstruktur möglich.

Während in einem SOA-System der Dienst aus einer Anwendung und einem Adapter besteht und damit ein fast monolithisches System abbildet. Besteht ein Microservice-System aus viele unterschiedlichen Diensten. In beiden Systemen ist es möglich, in kürzester Zeit Fehler zu korrigieren, sofern die nötigen Voraussetzungen gegeben sind. Das Anpassen, Verbessern oder Erweitern eines Dienstes, gestaltet sich jedoch einfacher in einem Microservice-System, als in einem SOA-System. Aufgrund der Größe von Microservices, kann ein Microservice von Grund auf neu erstellt werden, sollte dies nötig sein. Muss das System erweitert werden, können neue Microservices erstellt, welche die neuen Anforderungen bzw. Fachlichkeiten abbilden. Durch die Autarkheit von Microservices, können diese deployed werden, ohne die Interoperabilität des bestehenden Systems zu gefährden oder zu beeinflussen. Anschließend müssen gegebenenfalls einzelne Dienste angepasst werden, um die neuen Funktionalitäten, in das bestehende System, einzubinden.

### 6.3.5 Bindung an Technologie Stacks

In beiden Paradigmen, sollten Dienste eine standardisierte Schnittstelle besitzen, damit diese, durch andere Dienste verwendet werden können. Zusätzlich ist dadurch eine Wiederverwendung des Dienstes möglich. Das verwenden einer standardisierten Schnittstelle schränkt jedoch die Auswahl der einzusetzenden Technologien ein. Zudem ist das verwenden einer standardisierten Schnittstelle nicht immer möglich.

In einem Microservice-System ist die Wiederverwendbarkeit ein wichtiges Thema. Dadurch wird man gezwungen, die Schnittstellen zu standardisieren. Oft werden hierfür REST-HTTP Schnittstellen, welche Sprachen unabhängig sind, verwendet. REST-HTTP basiert dabei auf dem HTTP-Protokoll. Dadurch kann die Schnittstelle in fast jeder Programmiersprache verwendet werden, wodurch ein Dienst ebenfalls in fast jeder Programmiersprache geschrieben werden kann. Durch die Größe eines Microservices ist es ebenfalls möglich, neue Technologien zu Testen und in der Produktion einzusetzen. Dabei können verschiedenen Sprachen, sowie Frameworks verwendet werden, um die Aufgabe zu erledigen. Stellt sich heraus, dass eine Technologie, Sprache oder Framework sich nicht für die Erledigung einer Aufgabe eignet, kann der Dienst, aufgrund seiner Größe, in kürzester Zeit ausgetauscht werden. Jedoch birgt das einsetzten neuer Technologien auch Gefahren. So ist die Versuchung groß, experimentelle oder ungetestete Technologien in Produktion einzusetzen, ohne zuvor zu testen, welche Auswirkungen die Technologie auf das System hat.

Ein Dienst in einem SOA-System beinhaltet eine monolithische Anwendung. Die Schnittstellen werden durch Adapter bereitgestellt. Diese müssen mit der Anwendung interagieren können, wodurch der Adapter oft in derselben Sprache geschrieben werden muss, wie die zu interagierenden Anwendung. Aufgrund der Größe von Monolithen ist es nicht möglich, diese in kürzester Zeit auszuwechseln. Zudem muss man sich bei der Entwicklung einer Anwendung, auf einen Technologie Stack festlegen. Dadurch ist man bei der Weiterentwicklung oder Wartung der Anwendung an diesen gebunden. Eine neue Technologie einzuführen, würde bedeuten die Struktur der Anwendung zu verändern.

### 6.3.6 Entwicklung und Testbarkeit

Damit ein Dienst in die jeweiligen Systeme eingebunden werden kann, muss dieser zunächst entwickelt und getestet werden. Dabei ist zu beachten, dass in einem SOA-System, ein Dienst eine monolithische Anwendung mit Adapter ist, während ein Microservice-System eine vollständige Anwendung beschreibt und ein Dienst nur ein Teil der Anwendung ist.

Durch diese Unterschiede ist nicht nur die Größe des Entwicklerteams unterschiedlich, sondern auch der Entwicklungsprozess. In einem Microservice-System werden häufig DevOp-Teams in der Größe von 4-6 Personen eingesetzt. Da ein Dienst in einem Microservice-System, nur von einem Team entwickelt werden sollte, stellt dies eine Begrenzung der Größe eines Dienstes dar. Dadurch ist in der Regel ein Microservice deutlich kleiner, als eine monolithische Anwendung, jedoch nicht immer einfacher zu entwickeln.

Während bei einer monolithischen Anwendung, alle nötigen Abhängigkeiten in einem Projekt vorhanden sind, muss bei Microservices dafür gesorgt werden, dass die Abhängigkeiten korrekt aufgelöst werden, da diese in dem Bereich anderer Microservices liegen. Damit die Abhängigkeiten korrekt aufgelöst werden können, muss ein Entwickler wissen, welche Abhängigkeiten die korrekten sind. In einem Microservice-System gibt es eine sehr große Menge an Diensten, welche verschiedene Fachlichkeiten bereitstellen, daher ist es nicht immer einfach den richtigen Service auszuwählen.

Nach der Entwicklung eines Dienstes, muss dieser getestet werden, bevor der Dienst in die Produktion deployed werden kann. In einer monolithischen Anwendung, kann mit Hilfe von automatischen JUnit Tests ein Großteil der Funktionalität getestet werden. Es kann unter anderem getestet werden, ob jegliche Abhängigkeiten innerhalb der Anwendung einwandfrei funktionieren. Benötigt ein Microservice keine anderen Dienste, so kann mit Hilfe von JUnit Tests ebenfalls die Funktionalität vollständig geprüft werden. Anders sieht das jedoch bei Microservices aus, die auf weitere Dienste angewiesen sind. Hierbei müssen entweder die anderen Dienste simuliert werden oder der Microservice muss in eine entsprechende Testumgebung deployed und dort getestet werden. Das Betreiben einer solchen Testumgebung ist nicht unproblematisch, da eine exakte Kopie des Produktivsys-

tems notwendig ist, um relevante Testergebnisse zu erlangen.

Zudem sollte in jedem der beiden Systeme getestet werden, wie sich einzelne Services, beziehungsweise das gesamte System, bei zufälligen Ausfällen von verschiedenen Systemen, verhalten. Dies stellt eine besondere Herausforderung dar, da dies meistens nur in der Produktionsumgebung möglich ist. Netflix Inc., welches auf Microservices für ihren Dienst „Netflix“ setzt, hat für diesen Zweck ein Werkzeug Namens „chaosmonkey“ entwickelt, welches willkürlich Systeme innerhalb seines Dienstes „Netflix“ abschaltet.[1]

[1] [2]

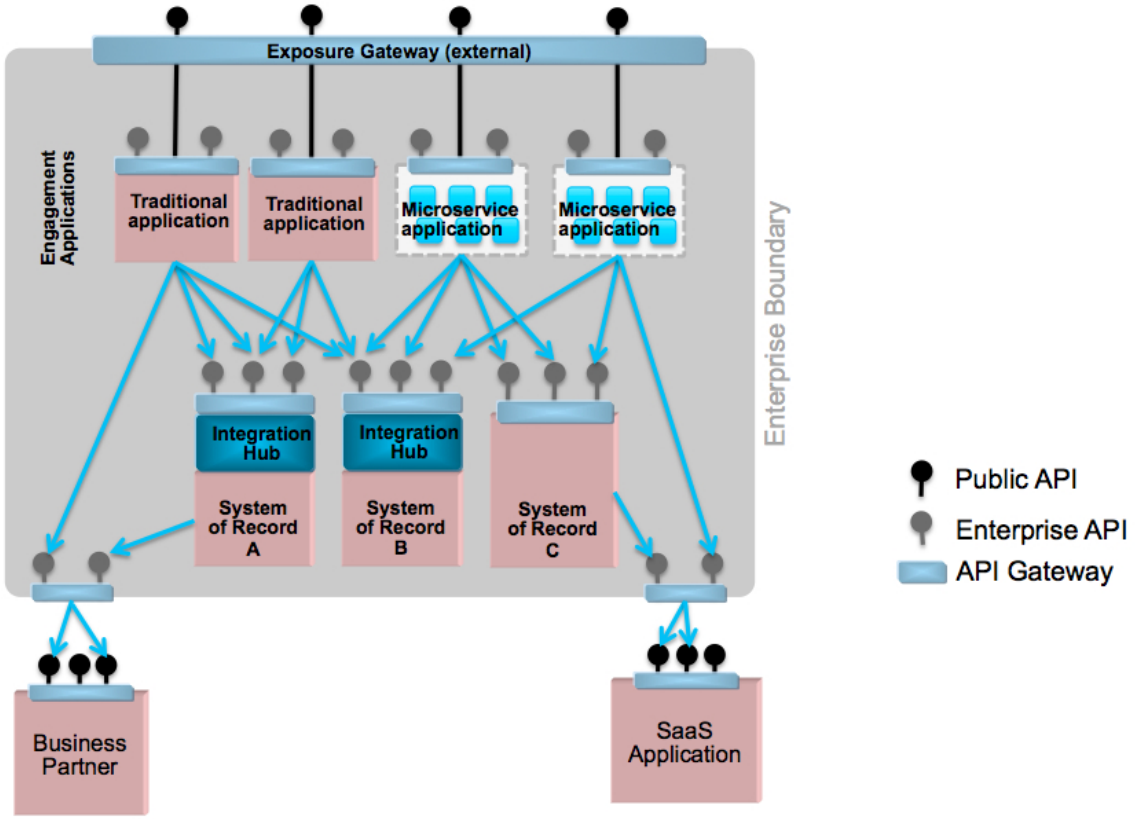
# Kapitel 7

# Fazit

Ziel dieser Arbeit war es, die Gemeinsamkeiten und Unterschiede zwischen SOA und Microservices herauszuarbeiten. Dazu wurden die Vor- und Nachteile der beiden Paradigmen hinsichtlich Prozessisolierung, Skalierung, Deployment, Wartbarkeit (Korrigierbarkeit, Erweiterbarkeit, Anpassbarkeit, Verbesserung), Entwicklung/Testbarkeit und die Bindung an Technologiestacks herausgearbeitet und anschließend verglichen werden.

Nach Abschluss des Projektes, kann man sagen, dass beide Paradigmen, unterschiedliche Ansätze verfolgen. Während bei dem Microservice Paradigma, eine Anwendung in kleinere Dienste, sogenannte Microservices, aufgeteilt wird, um die Entwicklung zu vereinfachen. Das SOA Paradigma hingegen verwendet vollständige Anwendungen, in Kombination eines dafür entwickelten Adapters, als Dienst. SOA soll die Kommunikation zwischen den einzelnen Anwendungen standardisieren und aufbereiten. Dabei soll darauf geachtet werden, dass die Benutzer der Dienste, nur die Informationen bereitgestellt bekommen, welche sie wirklich benötigen. In einem Unternehmen existiert nur ein einziges SOA-System, es können jedoch mehrere Microservice-Systeme vorhanden sein.

SOA und Microservices schließen sich nicht gegenseitig aus. Microservice-Systeme können Teil eines SOA-Systems sein und ihre Aufgaben in diesen einbinden. Dabei wird ein Microservice-System wie eine vollständige Anwendung behandelt. Der Unterschied ist jedoch, dass meistens deutlich mehr Schnittstellen zur Verfügung gestellt werden können, aufgrund der internen Struktur.
Möchte man eine Service-orientierte Architektur einsetzten, muss zunächst einmal die




Quelle: https://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html

**Abbildung 7.1:** Microservice, SOA und APIs Kombiniert

Frage gestellt werden, welchen Zweck damit erfüllt werden soll. Möchte man den Softwareentwicklungsprozess eines Unternehmens unterstützen, so sollte das Microservice Paradigma, vor dem SOA Paradigma gewählt werden. Möchte man jedoch die Businessprozesse in einem Unternehmen verbessern, sollte das SOA Paradigma eingesetzt werden.

Es sollte jedoch nicht nur die Fragen nach dem Zweck gestellt werden, sondern auch nach den Einsatzmöglichkeiten. Wird ein System benötigt, welches einfach zu skalieren ist oder ein System, bei dem der Benutzer nur die Informationen zur Verfügung gestellt bekommt, welche wirklich benötigt werden. Während in einem Microservice-System deutlich einfacher Skaliert werden kann, als in einem SOA-System, werden durch die Schnittstellen sehr viele Informationen bereitgestellt. Bei einem SOA-System hingegen wird dafür gesorgt, dass die Informationen gefiltert und aufbereitet werden, sodass der Nutzer nur die Informationen bekommt, welche er benötigt, jedoch kann nicht schnell skaliert werden.

Es muss dementsprechend gut überlegt werden, welches Paradigma eingesetzt werden soll. Zudem sollte man sich den Aufwand bewusstwerden, welcher benötigt wird, um das eingesetzte Paradigma einzusetzen.

## 7.1 Ausblick

In der Bachelorarbeit soll eine Anwendung, für die Zusammenarbeit, beispielhaft mit dem Microservice Paradigma erstellt werden. Anhand diesem beispielhaften System soll untersucht werden, welche Voraussetzungen gegeben sein müssen, um ein Microservice-System zu betreiben und welche Werkzeuge eingesetzt werden müssen, um ein Microservice-System aufzubauen, bzw. zu verwenden und zu verwalten. Konkret soll Untersucht werden, wie genau ein Microservice deployed und skaliert werden kann. Zudem soll untersucht werden wie Microservices in solch einem System gefunden und verwendet werden können. außerdem soll untersucht werden, wie die Benutzerverwaltung und die Authentifizierung in solch einem System funktioniert. Konkret soll untersucht werden, wie Benutzerberechtigungen verwaltet und abgefragt werden können, welche unterschiedlichen Ansätze dafür existieren und welche Vor- und Nachteile die jeweiligen Ansätze besitzen. Zusätzlich soll untersucht werden, wie das Monitoring in solch einem System funktioniert.

# Literaturverzeichnis


[1] ANDREW S. TANENBAUM, Maarten van S.: *Verteilte Systeme - Prinzipien und Paradigmen*. Bd. 2., aktualisierte Auflage. 2007. – ISBN 978-3-8273-7293-2

[2] BENNETT, Cory ; TSEITLIN, Ariel: *Chaos Monkey Released Into The Wild*. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html. 2012. – last visited: 04.12.2016

[3] FOWLER, Martin: *BoundedContext*. http://martinfowler.com/bliki/BoundedContext.html. – Stand 09.05.2016

[4] HURWITZ, Judith ; BLOO, Robin ; KAUFMAN, Marcia ; HALPER, Dr. F.: *Service Oriented Architecture For Dummies*

[5] KERRIE HOLLEY, Dr. Ali A.: *100 SOA Questions Asked and Answered*. http://www.professores.uff.br/screspo/100-SOA-Questions.pdf

[6] WOLFF, Eberhard: *Continuouos Delivery - Der Pragmatische Einstieg*. Heidelberg : dpunkt.verlag, 2015. – ISBN 978-3-86490-208-6

[7] WOLFF, Eberhard: *Microservices - Grundlagen flexibler Softwareentwicklung*. Heidelberg : dpunkt.verlag, 2016. – ISBN 987-3-86490-313-7

**Eidesstattliche Erklärung**
Ich versichere an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 13. Dezember 2016 Stefan Kruk

**Erklärung**
Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Dortmund, den 13. Dezember 2016 Stefan Kruk