Projektarbeit

Thema:

Gemeinsamkeiten und Unterschiede von SOA und Microservices

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

An der Fachhochschule Dortmund im Fachbereich Informatik erstellte
Projektarbeit
im Studiengang Softwaretechnik (Dual) - Modul Entwicklung verteilter
Anwendungen

Betreuer: Prof. Dr. Johannes Ecke-Schüth

Fachbereich Informatik

Dortmund, 4. November 2016

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 4. November 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 4. November 2016 Stefan Kruk

Überblick

Kurzfassung

In dieser Arbeit werden die Vor- und Nachteile von Service-orientierte Architektur und Microservices erörtert und anschließend miteinander verglichen. Dafür wird eine Ausgangssituation beschrieben und die Problematiken analysiert, aus welcher sich die zu untersuchende Fragestellung bildet. Nachdem beide Paradigmen getrennt erläutert wurden, werden diese miteinander verglichen.
Nach dem Vergleich wird auf die zu untersuchenden Fragestellungen eingegangen. Es soll insbesondere untersucht werden, in wie weit die Paradigmen für die Lösung dieser Fragestellungen geeignet sind. Abschließend wird ein Fazit aus den gewonnenen Ergebnissen gezogen und ein Ausblick auf folgende Arbeiten beschrieben.

Abstract

In this Projekt will be the advantages and disadvantages of SOA and Microservices described. After that both paradigm models will be compared. For this there will be described a initial situation and the problems will be analyzed from which it will be formulated questions. After both paradigm were seperately explained, they will be compared.
After comparison the to analyzed questions will be analyzed. It should in particular analyzed if one or both paradigm are solutions for the questions. Finally there will be drawn a conclusion from the results and a outlook for following Projekts.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Grundlagen

Das in dieser Arbeit behandelnde Thema ist für jede Person mit einer allgemeinen Informatikausbildung ohne Weiteres zu verstehen. Es wird bei dieser Personengruppe, die Kenntnisse über grundsätzliche Architekturen innerhalb der IT vorausgesetzt. Zudem kann vorausgesetzt werden, dass jede Person dieser Gruppe, der englischen Sprache mächtig ist. Weiterhin sollen einige Begrifflichkeiten geklärt werden, welche für das Verstehen dieser Arbeit notwendig sind:

1.1.1 Time-to-Market (TTM)

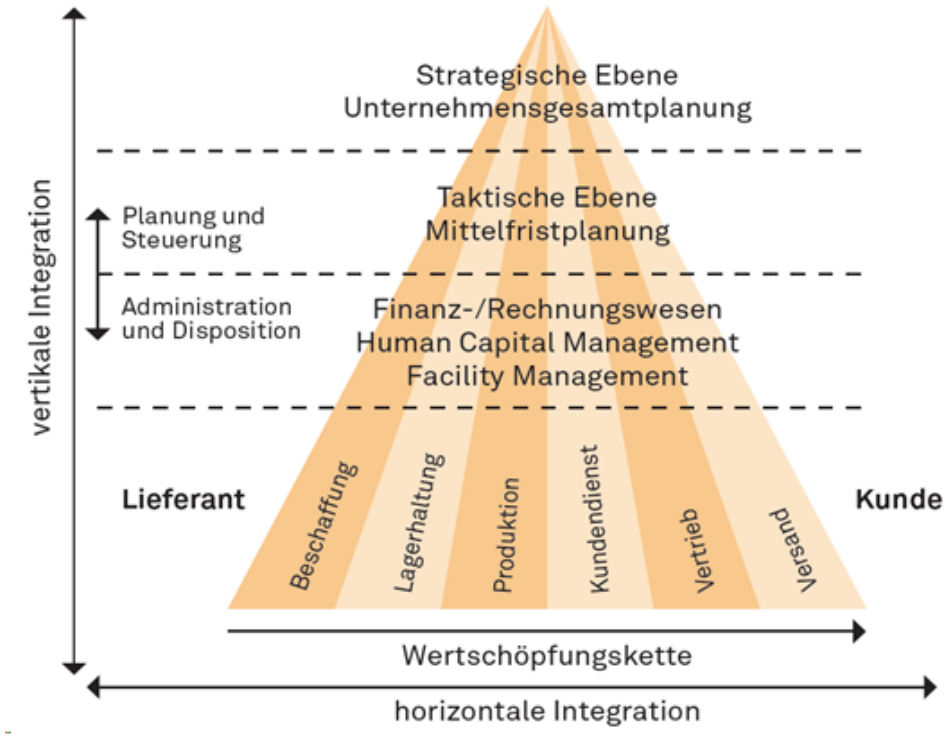
Unter dem Begriff Time-to-Market wird die Zeit von der Produktentwicklung bis zur Auslieferung auf dem Markt verstanden.[1] In dieser Zeit müssen Kosten für die Erstellung/Entwicklung aufgebracht werden, es werden in dieser Zeit jedoch keine Umsätze erzeugt. Daher strebt jedes Unternehmen eine möglichst geringe Time-to-Market Zeit an. Insbesondere wenn es um Wettbewerb geht, muss diese Zeit kurz gehalten werden, um neue, innovative Produkte schnell auf dem Markt zu bringen.

1.1.2 Wertschöpfungskette

»Die Wertschöpfungskette stellt die zusammenhängenden Unternehmensaktivitäten des betrieblichen Gütererstellungsprozesses grafisch

[1] Vergleich mit [4]

dar.«[1]



Quelle: http://www.referenzportal.ch/fuehrung/vom-erp-zum-integrierten-informationssystem/

Abbildung 1.1: Darstellung einer typischen Integrations-Pyramide

Das Bild zeigt eine typische Integrations-Pyramide eines Unternehmens. In dieser Pyramide ist zum einen ersichtlich, dass für die Wertschöpfungskette die horizontale Integration wichtig ist, zum anderen jedoch auch, dass mehrere Abteilungen eines Unternehmens in einer Wertschöpfungskette durchlaufen werden müssen.

Die Dauer zum durchlaufen der Wertschöpfungskette nennt man auch Time-to-Market. Wird die Kette in möglichst geringer Zeit durchlaufen, wird hierdurch die Time-to-Market Zeitspanne verringert.

1.2 Begriffsabgrenzung

SOA und „Service-orientierte Architektur“

Sowohl SOA als auch Microservices zählen zu dem Paradigma 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.

1.3 Motivation

Die Anforderungen an Software werden zunehmend komplexer. Diese muss nicht nur funktionieren, sondern muss zum Teil in kürzester Zeit erweitert oder geändert werden, was eine besondere Herausforderung dar stellt. Dabei spielen sowohl funktionale, als auch nicht funktionale Anforderungen eine wichtige Rolle. Je komplexer Software wird, desto schwieriger ist sie zu warten und zu pflegen.

> »Die Zeitspanne des Time-to-Market [(TTM)] kann [dabei] ein sehr bedeutsamer Faktor für den Erfolg des Unternehmens darstellen und ist daher nicht zu vernachlässigen. Kurze Entwicklungszeiträume bei der Time-to-Market garantieren dem Unternehmen nämlich einen Vorteil gegenüber der Konkurrenz. Hier geht es darum, dem Kunden so schnell wie möglich ein neues und innovatives Produkt anbieten zu können.
>
> Um den Erfolg durch eine kurze Time-to-Market zu unterstützen und zu fördern, investieren Unternehmen in ihre Entwicklungsabteilungen. Diese können sich so nur auf ihre jeweiligen Bereiche konzentrieren. Durch eine leistungsfähige Entwicklungsabteilung kann so, in Kombination mit einem gut organisierten Projektplan und eindeutigen Zuständigkeiten, die Zeitspanne Time-to-Market so klein wie möglich gehalten werden. Gerade in Branchen, in denen Innovation eine übergeordnete Rolle spielt und der Produkt-Lebens-Zyklus generell eher kurz

> ausfällt, spielt eine kurze Time-to-Market eine außerordentlich wichtige Rolle.«[2]

Zusätzlich entstehen Kosten für die Planung und Entwicklung des Produktes. Bis zur Veröffentlichung des Produktes hat das Unternehmen Geld und Zeit investiert. Je länger die Zeitspanne des TTM ist, umso erfolgreicher muss das Produkt sein, bzw. der Preis hoch genug sein. Daher muss die Zeitspanne des TTM möglichst gering gehalten werden, damit möglichst zeitnah die Schwelle, ab der die Erlöse und die Produktionskosten gleich hoch sind (Break-even-Point) erreicht wird. Die Time-to-Markt Zeitspanne beschreibt dabei die Zeit, in der die Wertschöpfungskette durchlaufen wird. Indem diese in möglichst geringer Zeit durchlaufen wird, wird die TTM-Zeitspanne gering gehalten. Jedes Unternehmen strebt daher eine kurze TTM Zeitspanne an.

Zudem gibt es in einem Unternehmen nicht nur ein Softwareprodukt, sondern eine Vielzahl verschiedener Produkte für unterschiedliche Anwendungsfälle. Dabei kann es passieren, dass einige Anwendungen gleiche Funktionalitäten beinhalten. Müssen solche Funktionalitäten geändert werden, müssen dafür alle Anwendungen, welche diese Funktionen bereitstellen, geändert werden. Einfacher ist es daher, nur eine Anwendung zu haben, welche diese Funktionalität anbietet. Dadurch bedarf es nicht mehr der Anpassung von vielen Anwendungen, sondern nur noch von dieser einen.

1.4 Ausgangssituation

Bei der Ausgangssituation handelt es sich um ein fiktives Szenario, welches an ein reales Problem angelehnt ist.

Das Unternehmen *Auktionen GmbH* ist ein modernes und aktives Internet-Unternehmen, welches seit 1995 existiert. Wie aus dem Namen zu entnehmen, betreibt das Unternehmen eine Auktionsplattform im Internet. Diese wurde zunächst als monolithische Anwendung implementiert.

Mit zunehmenden Nutzerzahlen und damit täglichen Benutzungen, stellte das Unternehmen fest, dass es zunehmend schwieriger wurde, auf das Verhalten der Nutzer in angemessener Zeit zu reagieren. Da die zentrale Anwendung ein monolithisches System war, war es zudem schwierig diese zu deployen.

Muss eine neue Version herausgebracht werden, so bedeutet dies entweder die gesamte Infrastruktur für Wartungszwecke offline zu nehmen und die Anwendung neu zu deployen oder Server im Parallelbetrieb laufen zu lassen und jeden neuen Besucher auf die neue Version zu leiten. Die letzte Methode bietet jedoch einige Schwierigkeiten, denn es könnten Änderungen eingebaut worden sein, welche im Konflikt mit der alten Version stehen. Dann muss in jedem Fall die erste Variante gewählt werden und die gesamte Infrastruktur offline genommen werden.

Beide Varianten der Produktivname von Änderungen sind jedoch zeitaufwendig und komplex. Nicht nur das Deployen könnte Probleme bereiten, sondern auch die darauf folgende Ausführung des Programms. Wird der Code in einer monolithischen Applikationen geändert, kann dies ebenfalls Auswirkungen auf bestehende Teile des Codes haben, welche vorher, ohne Probleme, funktionierten. Werden Tests vernachlässigt oder wird nicht ausreichend getestet, kann es leicht passieren, dass sich Fehler einschleichen, wodurch eine Version in Betrieb genommen wird, welche Fehler enthält. Darauf folgend müssten diese behoben werden und die Anwendung erneut deployed werden.

Im Falle einer monolithischen Architektur bedeutet das viele Änderungen für neue Features. Oft passiert es daher, dass Codestücke zurückbleiben, welche nicht mehr benötigt werden. Irgendwann ist die Applikation daher so groß, dass sie nicht mehr wartbar ist und neue Features nur noch schwer zu implementieren sind. Entstehen Fehler in solch einer Anwendung ist es umso schwerer diese zu finden und zu beheben.

Aus diesem Grund hat die *Auktionen GmbH* sich entschlossen, die monolithische Strukturen nicht weiter zu entwickeln, sondern auf eine Service-orientierte Archi-

tektur umzustellen.

1.5 Vorgehen und Kapitel

Zunächst werden die Probleme bei der Verwendung von monolithischen Architekturen, unter dem Aspekt der Softwareentwicklung und Wartung analysiert. Darauf aufbauend soll die grundlegende Problematik herausgearbeitet werden. Anschließend werden die Grundlagen zur Verwendung von Service orientierten Architekturen und deren Vorteile, aufbauend auf die zuvor erläuterte Problematik, erklärt.
Darauf folgend wird zunächst das Architekturparadigma „SOA“ und anschließend das „Microservice“ Paradigma betrachtet. Beide Paradigmen werden einzeln und unabhängig voneinander erläutert, um die spezifischen Eigenschaften beider Herangehensweisen herauszuarbeiten.
Abschließend werden die gewonnen Kenntnisse zusammengetragen und ausgewertet. Dabei sollen beide Architekturparadigmen verglichen und bewertet werden. Zuletzt wird das Thema noch einmal zusammengefasst und ein Ausblick auf kommende Projekte bzw. die Bachelorarbeit gegeben.

Kapitel 2

Problemanalyse

Betrachtet man die Ausgangssituation aus Kapitel 1.4 Ausgangssituation wurde es zunehmend schwieriger die monolithische Anwendung zu warten. Schließlich wurde die Entscheidung getroffen, dass eine klassische monolithische Anwendung, dem Unternehmensziel nicht mehr behilflich ist. Es wurde daher beschlossen, die bisherige Struktur in eine Service-orientierte zu ändern.

2.1 Herausforderungen

Oft passiert es, dass ein Unternehmen große Vorstellungen von dem Unternehmensziel hat. Dies führt häufig dazu, dass Software entwickelt wird, welche weit über die aktuellen Anforderungen hinaus geht. Man möchte damit verhindern, Software an einem späteren Zeitpunkt neu zu entwickeln oder austauschen zu müssen. Jedoch kann dies zu großen Problemen führen sobald das Unternehmen wächst und den am Anfang genannten Vorstellungen näher kommt. In dieser Phase entstehen meistens Probleme, welche vorher nicht berücksichtigt worden sind, weil niemand sie kannte. Dadurch muss die vorhandene Software, welche ursprünglich für dieses Szenario ausgelegt war, geändert werden.

Eine weitere Herausforderung besteht in der Umstellung auf eine Service-orientierte Architektur. Wurde, wie im Fall von der *Auktionen GmbH*, zunächst auf ein monolithisches System gesetzt, ist die Umstellung auf ein Service-orientierte Architektur nicht trivial. Die bestehende Anwendung muss weiterentwickelt werden, während

auf die neue Architektur umgestellt wird.

2.2 Die zu untersuchende Fragestellung

Das Problem besteht darin, eine Anwendung flexibel und einfach erweiterbar zu gestalten, damit ein Unternehmen schnell auf veränderte Bedürfnisse der Nutzer reagieren kann. Die Time-to-Market Zeitspanne muss daher möglichst gering gehalten werden. Damit solch eine Software ebenfalls gut wartbar ist, sollten Redundanzen möglichst vermieden werden. Das heißt, eine Funktionalität ist nur einmal im gesamten Unternehmen vorhanden. So können zum Beispiel alle Codestücke einer bestimmten Berechnung in ein Modul gepackt werden, wodurch diese nur an einer zentralen Stellen geändert werden muss.

Um dieses Vorgehen zu vereinfachen hat man sich dazu entschieden, derartige Codestücke in eigene Module/Applikationen (Dienste) zu verpacken und diese über eine Schnittstelle anzubieten. Das sorgt dafür, dass jede Software, welche dieses Modul benötigt, sie nicht mehr eigenständig implementieren muss, sondern den dafür vorgesehenen Dienst aufrufen kann, um die nötigen Informationen zu erhalten. Hierbei spricht man von einer Service-orientierten Architektur. Sowohl SOA als auch Microservice kommen in diesem Szenario in Betracht .Es stellen sich dementsprechend folgende Fragen:

1. Welche Unterschiede existieren zwischen den Paradigmen?
2. Welche Vor- und Nachteile hat das jeweilige Paradigma?
3. Gibt es Grenzen oder Beschränkungen bei der Benutzung der genannten Paradigmen?
4. In wie weit helfen die Paradigmen die angesprochenen Problematiken zu lösen?

Kapitel 3

Allgemeine Grundlagen zur Verwendung von Service-orientierten Systemen

Software zu entwickeln ist nicht immer einfach. Je komplexer eine Software ist, umso mehr Probleme können auftreten. Es bedarf einer genauen Planung und Verständnis der Unternehmensinfrastruktur, um eine Software mit allen Anforderungen zufriedenstellend zu implementieren.
Vor allem wenn es um die Weiterentwicklung und Wartung von Software geht, können Schwierigkeiten auftreten. Wurde die Architektur nicht gut gewählt oder schlecht umgesetzt, kann es das weitere Vorgehen stark beeinträchtigen oder verhindern. Es wurden daher Software-Architekturen und -Paradigmen entwickelt, welche flexibler und einfacher Änderbar sind.

3.1 Die Service-orientierte Architektur

Diese Architekturen zielen, wie der Name schon sagt, auf eigenständige Dienste (Services) ab, welche durch verschiedene Kanäle miteinander kommunizieren können und dadurch die gewünschten Geschäftsprozesse abbilden.

> »Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen«[8, S. 2]

Anstatt eine einzige große Anwendung ein zu setzten, setzt man auf viele kleine, verteilte, autarke Anwendungen. Diese Services bieten nach außen entsprechende Schnittstellen an, um den jeweiligen Dienst nutzen zu können. Eine Möglichkeit der Bereitstellung ist REST-HTTP oder SOAP. Es gibt zwar noch weitere Möglichkeiten, jedoch sind diese beiden besonders geeignet, da sie Programmiersprachen unabhängig sind und als Übertragungsmedium XML oder JSON nutzen. Durch die verteilten Anwendungen kann das System auch dann noch funktionieren, wenn einzelne Dienste nicht verfügbar sind, jedoch bringt es ebenfalls die typischen Probleme von verteilten Anwendungen mit sich.

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.«[3, 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. Jedoch entstehen dadurch Probleme, welche in monolithischen Systemen/Anwendungen nicht vorhanden sind.

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 wenn 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 merkt, dass ein System ausgefallen ist.

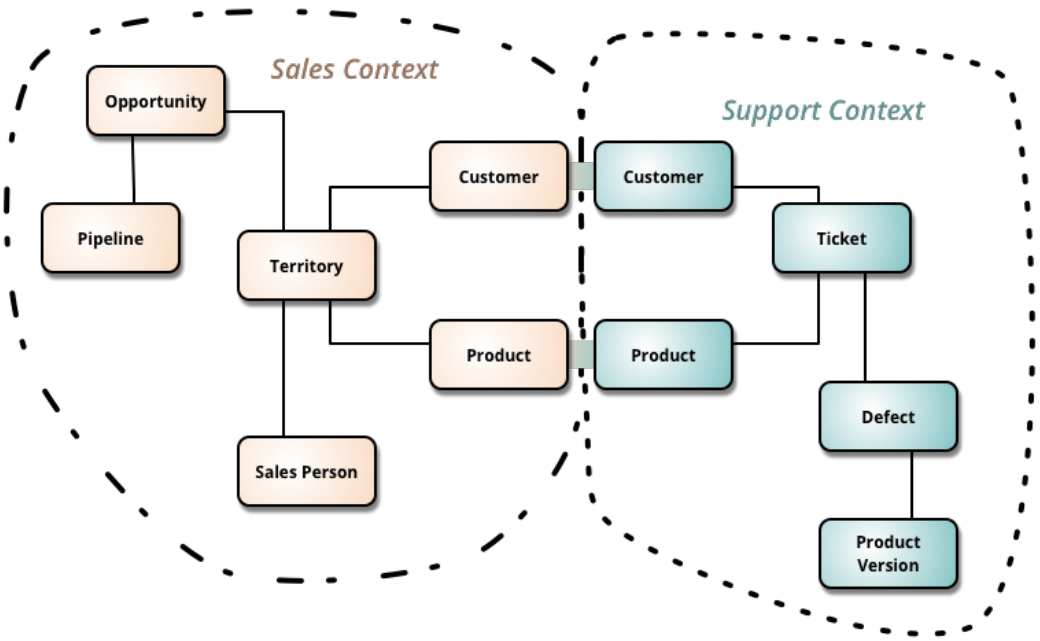
3.2.1 Domain-Driven Design und Bounded Context

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.«[5]

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.



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

Abbildung 3.1: Bounded Context

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 ist. Jedes Team innerhalb einer Organisation trägt zu der Entwicklung der Architektur bei. Wird eine Schnittstelle zwischen zwei Teams benötigt, so müssen diese Teams auch kommunizieren können. 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.

3.2.3 Synchrone vs Asynchrone Kommunikation

Kommunikation ist ein zentraler Bestandteil von verteilten Systemen. Ohne Kommunikation kann kein verteiltes System arbeiten. Der Ausfall eines Dienstes oder eines Kommunikationsweges hat die Folge, dass das System nicht mehr ordnungsgemäß arbeiten kann. Daher müssen Vorkehrungen getroffen werden, damit auch in diesem Falle das System funktioniert. Es muss grundlegend zwischen zwei Kommunikationsarten unterschieden werden.

Eine Kommunikationsart ist die synchrone, dabei werden Nachrichten zwischen zwei Anwendungen in Echtzeit versendet und empfangen. Ein Nachrichtenspeicher existiert hierbei nicht. Hierdurch gehen Nachrichten verloren, wenn sie nicht empfangen werden können. Die einzige Möglichkeit Fehler dieser Art abzufangen, ist es die aktuelle Aktion abzubrechen.

Eine weitere Kommunikationsart ist die asynchrone. Anders als bei der synchronen Variante, wird bei der asynchronen Kommunikation ein Nachrichtenspeicher verwendet, welcher die Nachrichten entgegen nimmt und an den eigentlichen Empfänger weiterleitet. Ist der Empfänger nicht erreichbar, werden die Nachrichten solange gespeichert, bis der Empfänger sie abruft.

3.3 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.

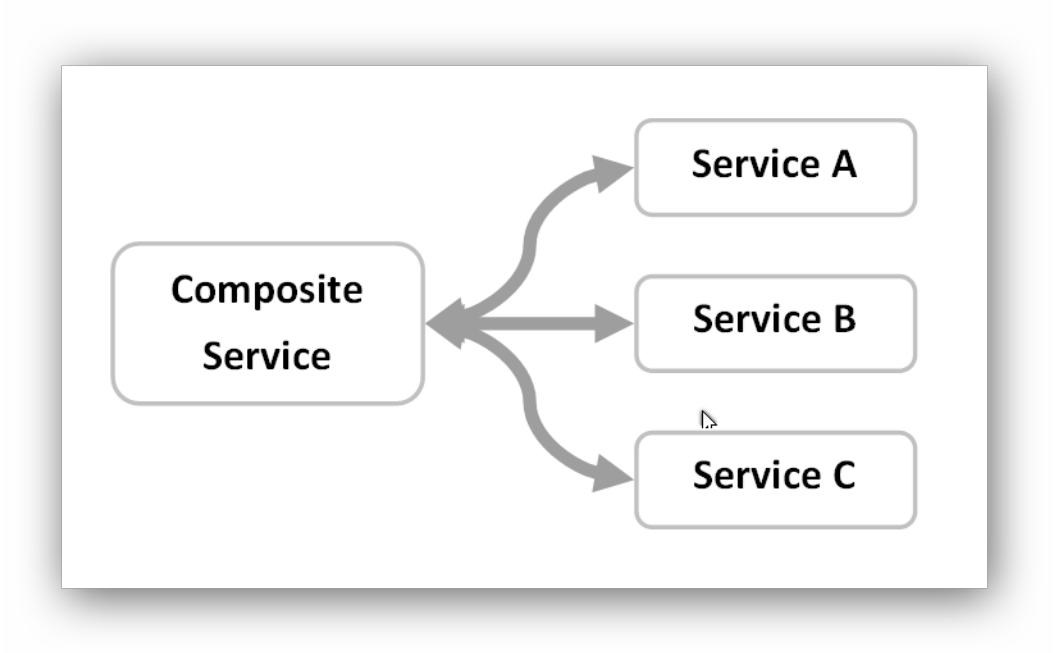


Abbildung 3.2: Orchestration

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.

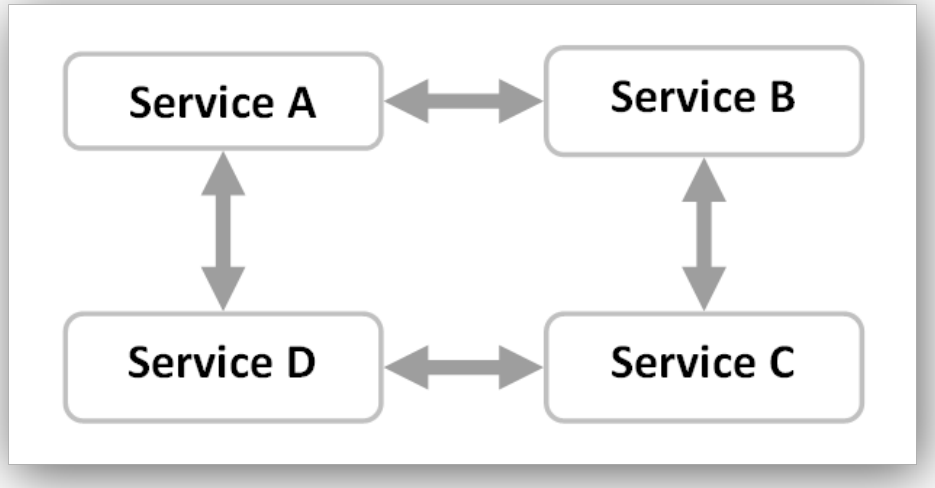


Abbildung 3.3: Choreographie

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

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.«[7, vgl. Seite 6]
Für einen Kaufmann ist SOA etwas anderes als für einen Analysten. Um die unterschiedlichen Sychten auf SOA besser zu verstehen, werden zunächst einmal einige Definitionen nach [7] genannt:

1. »To the chief information officer (CIO), SOA is a journey that promises to reduce the lifetime cost of the application portfolio [...].«[7, 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.«[7, 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.«[7, 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.«[7, vgl. Seite]

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 eine Herangehensweise und kein festes Modell.

4.1 Grundlagen

Das Ziel von SOA ist nicht die Entwicklung zu vereinfachen oder voran zu bringen. SOA soll die unternehmensweiten Geschäftsprozesse standardisieren und vereinheitlichen. Aus diesem Grund sollte SOA nicht als Modell, sondern als Herangehensweise verstanden werden.

Bei der Vereinheitlichung und Standardisierung spielt die Wertschöpfungskette eine wichtige Rolle, da diese die Geschäftsprozesse miteinander verbindet.

Bei der Vereinigung verschiedener Geschäftsprozesse spielt die Kommunikation ebenfalls eine entscheidende Rolle. Wie in Kapitel 3.2.2 Das Gesetzt von Conway erwähnt, können nur solche Systeme entworfen werden, welche die Kommunikationsstrukturen der Organisation abbilden. Daher müssen, wenn man SOA verwenden möchte, die nötigen Kommunikationswege geschaffen werden, um eine ordnungsgemäße und standardisierte Kommunikation zu gewährleisten.

SOA soll dabei helfen Geschäftsprozesse und Geschäftskomponenten in ein bestehendes Unternehmen einzubinden. Ein Beispiel: Die *Auktionen GmbH* übernimmt das Unternehmen *Handel GmbH*. Beide Unternehmen besitzen zwar ähnliche Prozesse und Komponenten, können jedoch nicht ohne weiteres in die bestehende

Struktur übernommen werden. Mit Hilfe von SOA soll dies vereinfacht werden.

4.1.1 Business und IT

Ein zentraler Bestandteil von SOA ist die IT. IT darf nie zum Selbstzweck existieren. Sie wird immer zur Unterstützung der Geschäftsprozesse eingesetzt. Je mehr Geschäftsprozesse existieren, umso mehr Software existiert in einem Unternehmen.

Oft existieren bereits verschiedene Anwendungen wie ERP- oder COBOL-Systeme. Mit SOA soll dafür gesorgt werden, das diese Systeme möglichst effizient miteinander arbeiten können.

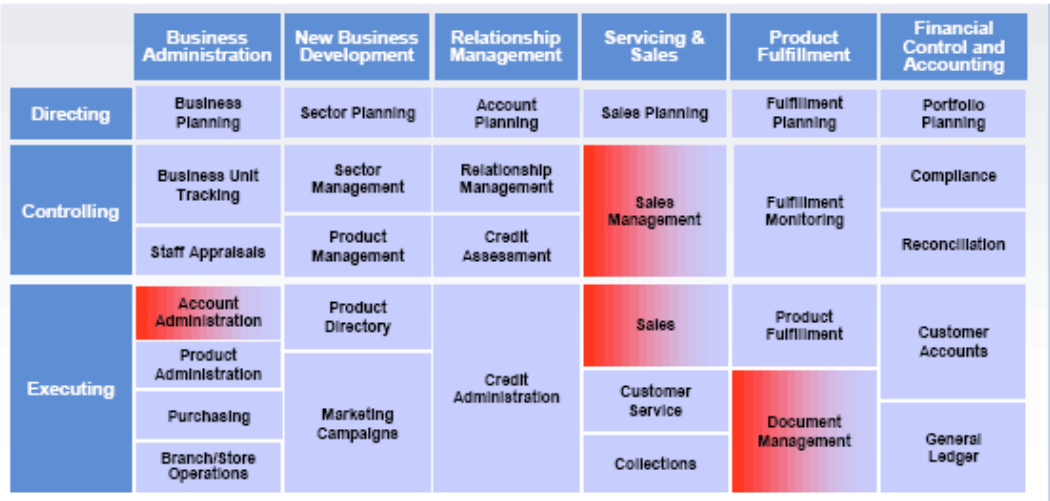
> »The complexity of the technology infrastructure at many companies in the financial services sector makes it very hard to leverage IT services in a coordinated way across the enterprise. Many large companies have either merged or acquired other very large companies resulting in the integration of new 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.«[6, 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, keine 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.

4.1.2 Unternehmenskomponenten

Dazu müssen zunächst einmal alle Komponenten eines Unternehmens identifiziert werden. Die nachstehende Abbildung (4.1) zeigt ein Beispiel dieser Identifizierung:



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

Abbildung 4.1: Unternehmens Komponenten

Aus diesen Komponenten müssen nun die Geschäftsprozesse identifiziert werden, beginnend mit den Wichtigsten. 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.

4.2 Architektur

Die Architektur in einem SOA-System unterliegt dem Paradigma einer „Service-orientierten Architektur“. Mit diesem Paradigma ist der Begriff *Dienst* verbunden. Dieser ist im Falle von SOA jedoch zunächst nicht richtig, denn Anwendungen wie ERP- oder COBOL-Systeme sind eigenständige Anwendungen und keine Dienste.

Damit eine ERP-Anwendungen ihre Funktionalitäten bereitstellen kann, muss ein Adapter erstellt werden. Dieser Adapter stellt die Funktionalität in Form von Schnittstellen bereit und kann als Dienst bereitgestellt werden. Der Adapter übernimmt dabei Aufgaben wie Fehler abzufangen und die Daten in angemessener Weise aufzuarbeiten. Diese können über die jeweiligen Schnittstellen abgerufen werden und die eigentliche Anwendung steuern.

Schnittstellen können dabei zum Beispiel als HTTP-Rest oder SOAP Implementierung zur Verfügung gestellt werden. Es ist jedoch auch möglich weitere Schnittstellentechnologien zu verwenden, wie RMI. Genauso kann es möglich sein „Message Oriented Middleware“ zu verwenden, um Informationen bereit zu stellen.

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. Der Enterprise-Service Bus

dient dabei als Nachrichtenkomponente, welcher die Nachrichten entgegen nimmt und an die jeweiligen Empfänger weiterleitet. Es dient ebenfalls als Transformator von Nachrichten, um die Interoperabilität sicherzustellen.

Grundsätzlich wird ein ESB eingesetzt, um an einer zentralen Stelle die Kommunikation zu steuern und die Geschäftsprozesse abzubilden. Neben den genannten Funktionen, wird der ESB häufig als *Service Discovery Dienst* eingesetzt. Dies ist notwendig, damit alle vorhandenen Anwendungen im System registriert werden können. Am einfachsten lässt sich das an einem Beispiel erklären.
Man Stelle sich eine Bank-Anwendung vor, an der man sich anmeldet. Es werden daraufhin folgende Informationen angezeigt:

1. Name
2. Kontostand
3. EC- und Kreditkarten
4. Liste der Aktienfonds

Jede Information stammt aus einem anderen Teil des Systems und werden von verschiedenen Anwendungen über Schnittstellen bereitgestellt. Die Schnittstellen können sich dabei von Anwendung zu Anwendung unterscheiden. So kann zum Beispiel eine Anwendung die Informationen über HTTP bereitstellen, während eine andere sie über SOAP anbietet. So können die Informationen zum Beispiel aus einem CRM-System (Kundenbeziehungsmanagement-System) stammen oder durch PHP oder Ruby erzeugt werden.

Damit die Schnittstellen jedoch verwendet werden können, muss der ESB wissen, wo diese im System zu finden sind. Dafür wird ein *Service Discovery Dienst* benötigt. Zwar kann diese Aufgabe auch von einer externen Anwendung bearbeitet werden, jedoch muss in diesem Fall der ESB wissen, wo diese Anwendung zu finden ist und wie diese anzusprechen ist.

Anders als man vermuten würde, lässt man die Oberfläche, bzw. das System, nicht

direkt mit den einzelnen Komponenten reden. Die gesamte Kommunikation läuft dabei über den *Enterprise Service Bus* ab. Nachstehende Abbildung soll dies genauer erläutern:

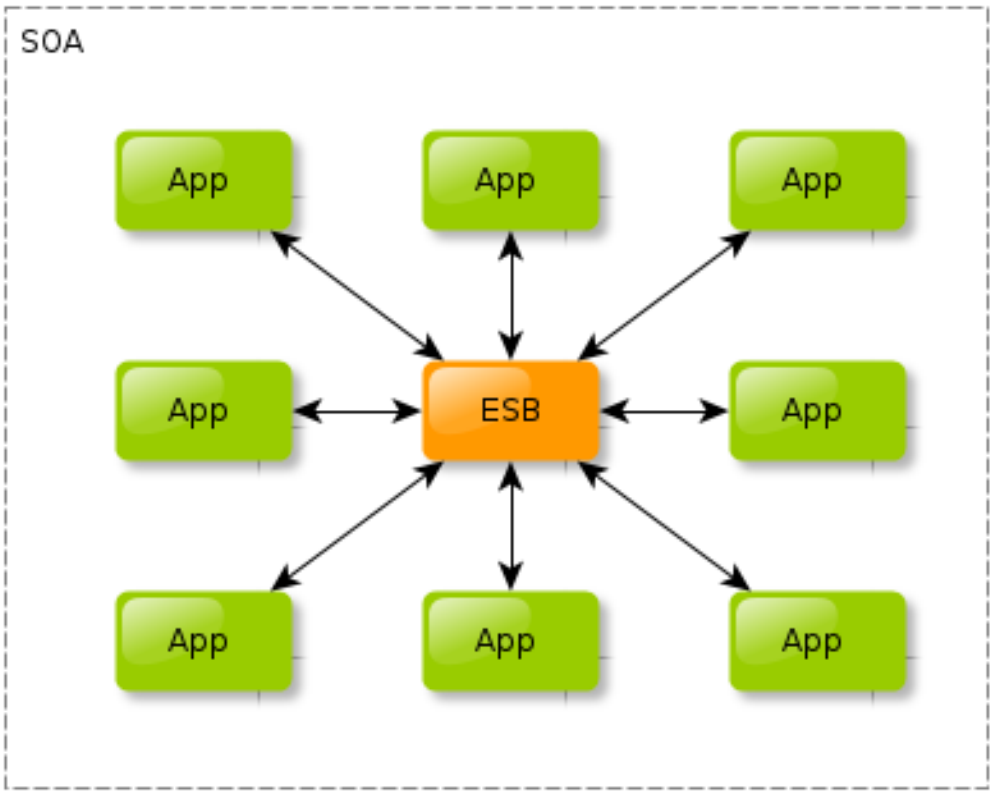


Abbildung 4.2: Enterprise Service Bus
Quelle: https://zato.io/docs/intro/esb-soa-de.html

Benötigt eine Anwendung (in der Abbildung App genannt) Informationen, konsultiert diese zunächst den ESB. Der ESB kennt die anderen Anwendungen oder benutzt einen Dienst, welcher die Informationen bereitstellen kann, und stellt daraufhin die angeforderten Informationen zur Verfügung. Dabei ist es egal wo die Dienste liegen, solange sie über das Netzwerk ansprechbar sind.

Kapitel 5

Microservices

Microservice Systeme gehören, wie SOA auch, zu den Service-orientierten Systemen. Dabei werden die einzelnen Dienste Microservice genannt. Dies hängt mit der Größe eines Dienstes zusammen.

5.1 Überblick

Ein Microservices-System besteht aus vielen verteilten Anwendungen (Dienste), welche man selber ebenfalls Microservices nennt. Dabei wird versucht, eine 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:«[9, 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.

Der Begriff Microservices ist nicht eindeutig definiert. Als erste Näherung dienen, nach Eberhard Wolff [9, 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.«[9, vgl S. 55].

5.2 Aufbau von Microservices

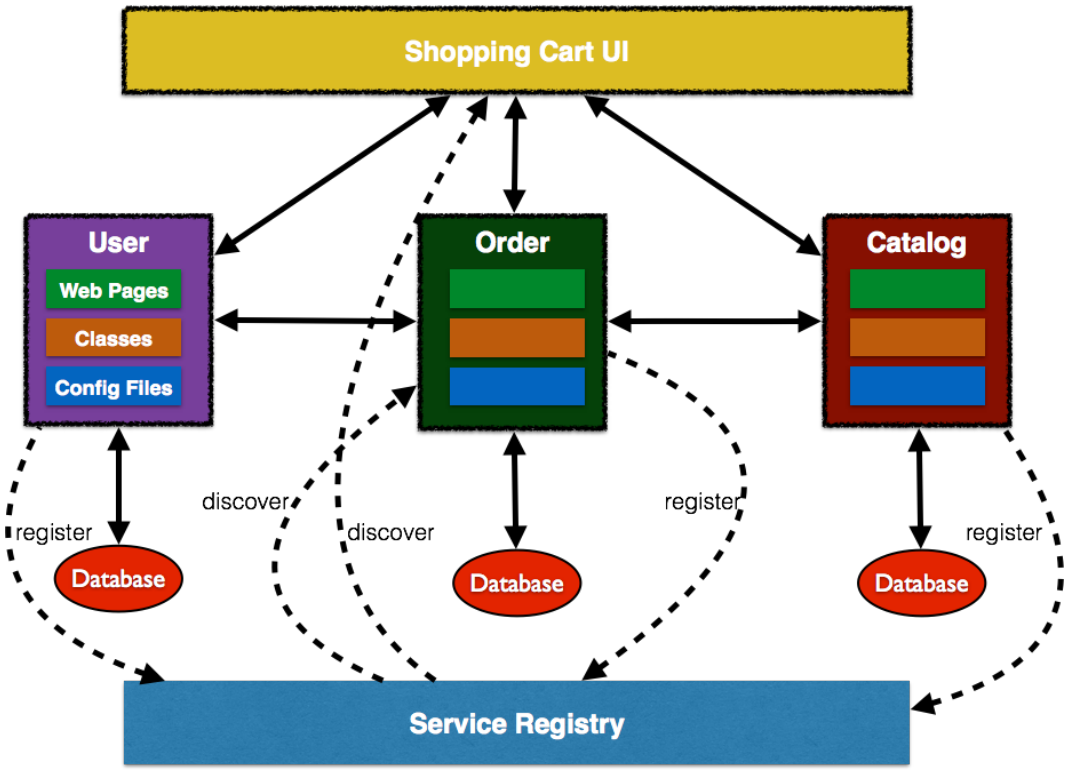
Wie bereits erwähnt, sind Microservices eigenständige Dienste. Sie müssen so gebaut sein, dass sie möglichst ohne Abhängigkeiten funktionieren. Um das zu realisieren dürfen Microservices nur lose gekoppelt sein. Damit ist gemeint, dass der Dienst seine Aufgabe selbständig erledigen kann ohne weiter Dienste in Anspruch zu nehmen. Bei „Self-Contained Systems“ ist ebenfalls ist ebenfalls eine UI vorhanden. Damit ist das System für sich selbständig und stellt ebenfalls alle nötigen Eigenschaften bereit für die Darstellung von Informationen. 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 jedoch 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

wird.

Geht man von *Self-Contained Systems* aus, besitzen Microservices zusätzlich noch eigene UI-Elemente. Dadurch ist nur noch ein aggregierender Dienst notwendig, welcher alle Informationen zusammenfasst.



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

Abbildung 5.1: Microservice Architektur

5.3 Größe von Microservices

»Der Name „Microservices“ verrät schon, dass es um die Servicegröße geht - offensichtlich sollen die Services klein sein."[9, 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 be-

> stimmte 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.«[9, S. 34]

Bei der Größe eines Services ist darauf zu achten, das 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.

5.4 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 man entscheiden 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.

5.4.1 Herausforderung

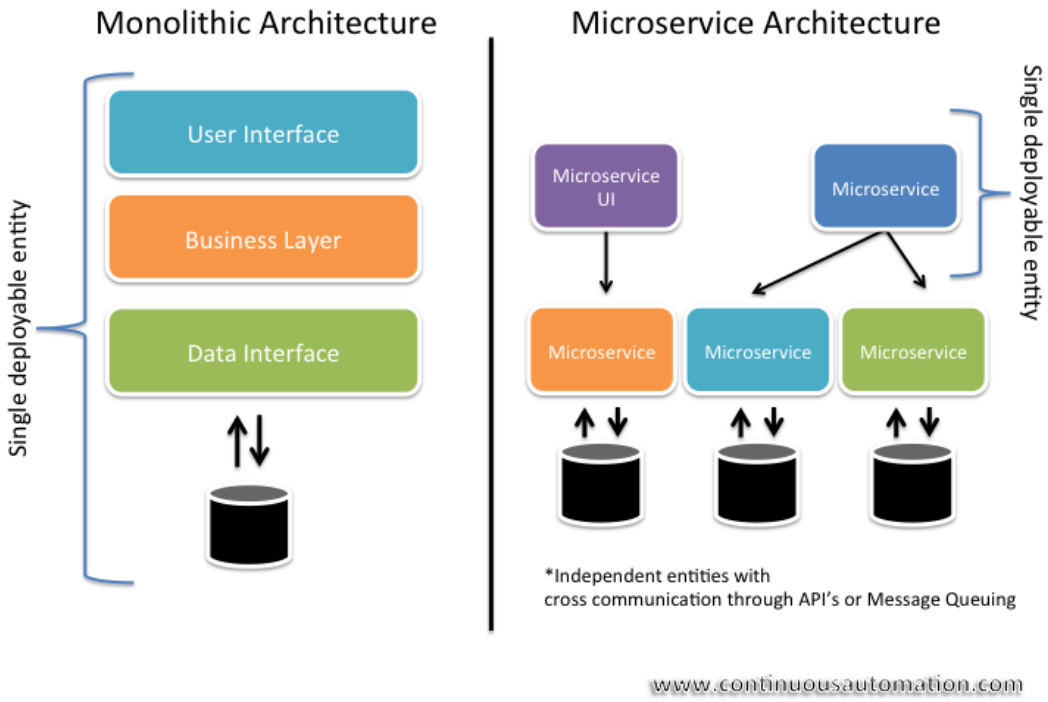
Der Ausfall eines Services kann im schlechtesten Fall dazu führen, dass alle anderen Microservices nicht mehr funktionieren. Um das zu verhindern muss klar definiert werden, was Microservices in dieser Situation tun sollen. Zusätzlich muss, sofern Datenbankoperationen eine wichtige Rolle spielen, das 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 einheitlicher Rollback durchgeführt werden, um keine inkonsistente Dateien im System zu erzeugen.

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 dafür 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 einer Fachlichkeit zugeordnet 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.

5.5 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 sind. 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 Boradcaster oder wichtige Microservices sind, überbrückt werden, indem der Broadcaster die Nachrichten für eine bestimmte Zeit vorhält und so der Microservice, welcher nicht erreichbar war, die Nachrichten trotzdem noch erhält.

Kapitel 6

Vergleich

Nachdem in den Kapitel 4 und 5 herausgearbeitet wurde, was unter **Microservice** und **SOA** zu verstehen ist, werden nun beide Paradigmen mit einander verglichen und kritisch betrachtet.

Bevor die Paradigmen mit einander verglichen werden können, ist zu erwähnen, dass ein direkter Vergleich zwischen Funktionen und den Vor- und Nachteilen der jeweiligen Modelle nicht möglich ist. Warum das so ist, wird weiter unten in Grundlagen genauer erläutert. Es ist außerdem zu erwähnen, dass weder der Begriff Microservice, noch SOA eindeutig definiert ist. Literaturen wie [7] und [9] bieten einen guten Ansatz, um diese beiden Begriffe grob einzuordnen.

6.1 Grundlagen

Der Grundgedanke beider Paradigmen ist sehr unterschiedlich. Zwar stammen beide Paradigmen aus dem Bereich der „Service orientierten Architekturen“ und auch *SOA* ist eine Abkürzung dieses Begriffes, jedoch verfolgen beide Paradigmen verschiedene Ansätze.

SOA hat seine Grundlagen nicht in der IT, sondern in der Geschäftswelt. Dies wird dadurch bestätigt, dass die Zielgruppe der Befragten in Kapitel 4 SOA aus dem kaufmännischen und analytischen Bereich stammen. So wird zum Beispiel der «business executive» und der «business analyst» gefragt, wie SOA zu verstehen ist.

SOA ist demnach *business-driven*. Zudem sollte SOA nicht als Modell, sondern als Herangehensweise verstanden werden.

Microservices hingegen stammen aus der Notwendigkeit große Software möglichst effizient zu erstellen und die Wartung zu vereinfachen. Wie schon im Überblick in Kapitel 5.1 erwähnt, ist Modularisierung nichts neues und wird aus den gerade genannten Notwendigkeiten eingesetzt. Die Microservice-Architektur ist nichts anderes als die Modularisierung einer Software, bei der diese in kleine Softwaremodule, sogenannte Services, geteilt werden.

Während bei SOA die Geschäftsprozesse im Vordergrund stehen, will man mit Microservices die Entwicklung von Software unterstützen. Das Microservice-Modell wurde entwickelt, um umfangreiche Software möglichst einfach und schnell mit vielen Personen, entwickeln zu können. Bei SOA geht es um den möglichst effizienten Einsatz von Software und nicht deren Entwicklung. Oftmals existiert schon Unternehmenssoftware. Durch SOA soll diese möglichst effizient untereinander kommunizieren.

Aus diesen Gründen ist ein direkter Vergleich der beiden Paradigmen nicht möglich.

Grundsätzlich wurden schon viel früher Architektur-Modelle entwickelt, um die in den jeweiligen Kapiteln 4 SOA und 5 Microservices genannten Probleme und Anforderungen umzusetzen. Ein Begriff für diese Modelle existierte jedoch noch nicht. Die jeweiligen Begriffe wurden erst später verwendet, um die jeweiligen Modelle spezifizieren und erklären zu können.

6.2 Architektur

Wie bereits weiter oben erläutert, wird mit SOA versucht verschiedene Geschäftsprozesse miteinander zu verknüpfen und eine leichtere Kommunikation unter diesen zu ermöglichen. Dadurch sind die meisten Dienste nicht Bestandteil einer großen

Software, sondern nur Bestandteil eines Unternehmensbereiches. Oft wird daher versucht alle EDV-Komponenten miteinander zu verknüpfen ohne eine Abhängigkeit zu erschaffen. Die Anwendungen selber sind dabei keine Dienste. Damit diese ihre Funktionalität für andere Anwendungen bereitstellen kann, muss ein Adapter entwickelt werden. Dieser Adapter wird als Dienst bereitgestellt.

Das Microservice-Modell arbeitet zwar auch mit Diensten, jedoch sind diese meist nicht so groß wie im SOA-Modell. Während bei dem SOA-Modell auch ganze Anwendungen mit Hilfe eines Adapters interoperable gemacht werden, wird mit dem Microservice-Modelle versucht eine ganze Anwendung in eigenständige Dienste aufzuteilen. Dabei soll ein Dienst nur eine Aufgabe erledigen, diese aber jedoch besonders gut. Zudem gibt es in der Regel keine zentrale Anwendung, welche als Schnittstelle zwischen Datenbanken und Frontend dient.

6.3 Kommunikation

Nachdem die Architekturunterschiede erläutert wurden, werden nun die Unterschiede in der Kommunikation erläutert.

Die Modelle unterscheiden sich hinsichtlich der Anordnung von Diensten und damit der Kommunikationsfluss innerhalb der Modelle. Grundsätzlich kann man zwischen zwei Kommunikationsflüssen unterscheiden:

1. Orchestration
2. Choreographie

Nicht beide Kommunikationsflüsse sind in beiden Service-orientierten Architekturen möglich. In SOA ist der ESB die zentrale Einheit über welches die gesamte Kommunikation läuft. Dadurch ist nur eine Orchestration der Dienste möglich und die direkte Kommunikation zwischen den Diensten ist nicht möglich. Bei Microservices hingegen können beide Kommunikationsflüsse realisiert werden, wodurch die Kommunikation viel dynamischer gestaltet werden kann, da jeder Dienst mit jedem anderen Dienst kommunizieren kann. Oftmals sprechen Frontend Anwen-

dungen wie Websiten direkt mit mehreren Services um bestimmte Informationen zu erhalten.

6.4 Beteiligte Personen

Neben den technischen Unterschiede, gibt es auch Unterschiede hinsichtlich der beteiligten Personengruppen. Dabei geht es nicht um die Nutzer einer Software, sondern um Personen die aktiv im Prozess der jeweiligen Modelle beteiligt sind.

Bei dem SOA-Modell wurden weiter oben einige Personengruppen identifiziert. Unter anderem gehören darunter Business Executiver und Business Analysten. Weitere Personengruppen sind zum Beispiel Personen aus dem kaufmännischen Bereich. Natürlich spielen auch Personen aus der IT-Abteilung eine Rolle, wie zum Beispiel Administratoren und Architekten. Da SOA zum Teil auf bestehende Anwendungen aufbaut wird meistens nur ein Adapter benötigt, welcher die Anwendung SOA-fähig macht. Dadurch kann meistens der Entwicklungsanteil gering gehalten werden. Dies reduziert tendenziell den Anteil von Personen aus der IT im Vergleich zu anderen Abteilungen.

Anders als beim SOA-Modell, steht beim Microservice-Modell die Entwicklung im Vordergrund. Wie bereits beschrieben, wird mit Microservices eine Anwendungen in viele eigenständige Dienste aufgeteilt. Hier spielen Entwickler eine zentrale Rolle, jedoch auch Personen aus dem kaufmännischen Bereich, da IT nie als selbst Zweck existieren darf, sondern immer die vorhandenen Businessprozesse unterstützen soll.

Die Unterschiede in den beteiligten Personen zeigen neben den unterschiedlichen Personengruppen auch in welchem Umfeld sich beide Architekturmodelle bewegen bzw. aus welchem sie entstanden sind.

Kapitel 7

Ergebniss

Nachdem die Technologien Microservice und SOA nun ausführlich beschrieben und verglichen wurden, muss überprüft werden, ob die Fragen aus Kapitel 2.2 Die zu untersuchende Fragestellung beantwortet werden können. Dabei wurde die Frage auf die Unterschiede der beiden Paradigmen bereits in Kapitel 6 Vergleich beantwortet. Es müssen jedoch noch folgende Fragestellungen beantwortet werden:

1. Welche Vor- und Nachteile hat das jeweilige Paradigma?
2. Gibt es Grenzen oder Beschränkungen bei der Benutzung der genannten Paradigmen?
3. In wie weit helfen die Paradigmen die angesprochenen Problematiken zu lösen?

7.1 Welche Vor- und Nachteile hat das jeweilige Modell?

Beide Paradigmen sind, wie bereits geschrieben „Service-orientierte Modelle“. Damit ist gemeint, dass eine Anwendung in mehrere Dienste aufgeteilt ist. In klassischen monolithischen Anwendungen kann es passieren, dass ein und dieselbe Funktion in verschiedenen Unternehmensanwendungen existiert.

Der Vorteil bei Microservices und SOA ist, dass eine Funktion in der gesamten

Infrastruktur nur einmal existiert. Dadurch ist es möglich bereits vorhandene Funktionen auf einfache Weise zu ändern ohne eine Vielzahl von Anwendungen, die diese Funktion benötigen, zu ändern und neu zu deployen. Ein weiterer Vorteil liegt in der Skalierung. Dadurch, dass Funktionen in einzelnen Diensten verpackt werden, ist es möglich, viel genutzte Funktionen und damit viel genutzte Dienste zu identifizieren und zu skalieren. In monolithischen Systemen muss immer die gesamte Anwendung skaliert werden, selbst wenn nur einzelne Teile beansprucht werden. Dadurch, dass für eine Funktion ein Dienst existiert, müssen diese nicht in der gleichen Programmiersprache geschrieben worden sein. Es können zum Beispiel viele unterschiedliche Sprachen verwendet werden. Die Dienste benötigen dabei standardisierte Schnittstellen, wie zum Beispiel REST-HTTP oder SOAP. Durch standardisierte Schnittstellen können jegliche Dienste diese nutzen ohne dass eine zusätzliche Brücke dafür gebaut werden muss.

Der Nachteil von Service-orientierten Systemen liegt in den Kommunikationswegen. Fällt zum Beispiel ein Dienst in diesen Systemen aus oder wird die Kommunikation zu diesen unterbrochen, kann es zu Teilausfällen im gesamten System kommen. Ist ein kritischer Dienst betroffen und es existieren keine weiteren Instanzen dieses Dienstes, kommt es im schlimmsten Fall zu einem Ausfall des gesamten Systems. Die Erkennung von Ausfällen ist ein weiteres Problem in solchen Systemen.

Bei SOA existieren zudem oft Anwendungen wie ERP- oder COBOL-Systeme. Auch diese müssen dem SOA-System hinzugefügt werden. Damit dies funktioniert, müssen oft Wrapper geschrieben werden, welche „um diese Systeme gelegt“ werden und die Kommunikation, sowie die Interpretation der Nachrichten steuern. Zusätzlich werden diese Wrapper dazu eingesetzt fehlerhafte Nachrichten bzw. Anweisungen heraus zu filtern.

7.2 Gibt es Grenzen oder Beschränkungen bei der Benutzung der genannten Paradigmen?

Einschränkungen gibt es in den Paradigmen hauptsächlich durch die Kommunikation zwischen den einzelnen Diensten. Wie bereits in 6.3 Kommunikation erläutert können Microservice-Dienste ohne Einschränkungen miteinander kommunizieren und Daten/Nachrichten austauschen. Bei SOA hingegen existiert ein Enterprise Service Bus (ESB). Alle Nachrichten werden durch den ESB an die jeweiligen Dienste verteilt. Eine direkte Kommunikation unter den Diensten ist nicht gewollt.

Im Falle von SOA spielt nicht nur die IT eine Rolle. Da SOA ursprünglich aus dem Businessbereich stammt müssen die Geschäftsprozesse so umgestellt werden, dass diese in das neue Modell passen. Zudem muss sich die IT dem Business an nähren und ihre Systeme auf das neue Modell umstellen. Dies sollte gut überlegt werden, da hiermit häufig ein großer und schwieriger Prozess verbunden ist. Oft ist es nicht einfach alle Geschäftsprozesse und die IT-Infrastruktur auf dieses Modell umzustellen.

7.3 In wie weit helfen die Paradigmen die angesprochenen Problematiken zu lösen?

In Kapitel 1.3 Motivation wurde die Problematik der Time-to-Market Zeitspanne genannt. Paradigmen wie Microservice und SOA können hier durchaus diese Problematik lösen. Die Time-to-Market Zeitspanne ist die Zeit, in der die Wertschöpfungskette durchlaufen wird. Je schneller dies geschieht, je geringer ist die Zeitspanne.

Mit Hilfe von SOA sollen alle Unternehmenskomponenten möglichst optimal miteinander kommunizieren. Damit dies geschieht, müssen die einzelnen Unternehmensprozesse standardisiert werden. Dadurch kann ebenfalls die Wertschöpfungskette standardisiert durchlaufen und im laufe dessen, die Prozesse optimiert wer-

den. Dadurch kann die Wertschöpfungskette in möglichst optimaler Zeit durchlaufen werden.

Bei einem Microservice-System hingegen, wird eine Anwendung in verschiedene Module (Services) aufgeteilt. Dadurch können diese schnell gewartet werden, ohne das gesamte System offline zu nehmen oder neu deployen zu müssen. Außerdem können neue Funktionen schnell hinzugefügt werden, ohne die Integrität der anderen Dienste zu gefährden. Hierbei beschreibt die Time-to-Market Zeitspanne, die Zeit bis eine neue Funktion implementiert und hinzugefügt wurde. Bei unserem fiktiven Unternehmen, der *Auktionen GmbH* könnte dies zum Beispiel die Möglichkeit sein, regional Beschränkte Angebote zu erstellen.

Damit die Time-to-Market (TTM) Zeitspanne möglichst gering gehalten wird und ein Benutzer nicht durch hinzufügen vieler neuer Funktionen zur gleichen Zeit überfordert ist, wird oft ein Continuous Delivery/Deployment Ansatz verwendet. Bei diesem Ansatz wird ein Feature Deployed und zur Anwendung hinzugefügt, sobald dieses fertig ist. Einzelne Funktionen sind oft schnell erstellt, jedoch werden diese in monolithischen Systemen oft nur in regelmäßigen Abständen, wie zum Beispiel einmal im Monat deployed. Dadurch beträgt die TTM Zeitspanne immer ein Monat und sollte ein Feature nicht rechtzeitig fertig werden, verlängert sich diese um einen weiteren Monat. Da in Service-orientierten Systemen Features deployed werden können, sobald sie fertig sind, beträgt die maximale TTM Zeitspanne genau solange, wie die Entwicklungsdauer eines Features. Es sollte jedoch auf die Versionierung der Schnittstellen geachtet werden, denn wenn diese geändert werden, kann es dazu führen, dass andere Dienste nicht mehr darauf zugreifen können. Oft werden daher ältere Schnittstellen weiterhin gepflegt und zu einem späteren Zeitpunkt, sobald kein Dienst mehr auf diese zugreift, entfernt.

7.4 Fazit

Beide Architekturen unterscheiden sich hinsichtlich ihrer Nutzung und Handhabung. So ist das Microservice Modell ein Entwicklungs-Modell. Es unterstützt die

IT-Abteilung bei der Entwicklung von Software, indem eine Anwendung in verschiedene Dienste aufgeteilt wird. Dies ähnelt den herkömmlichen monolithischen Anwendungen insofern, dass in diesen einzelne Funktionen in Pakete oder Bibliotheken verpackt sind. In der Microservice-Architektur sind diese in Dienste (Microservices) verpackt. Dagegen ist SOA ein Business-Modell. Es kommt aus dem Businessbereich und soll damit auch diesem Helfen. Hierbei sind Dienste keine „Microservices“ sondern zum Teil große ERP- oder COBALT-Systeme.

Das Einsatzgebiet der jeweiligen Paradigmen ist sehr verschieden, wodurch ein direkter Vergleich nicht möglich ist. Man kann jedoch kann aus den gesammelten Erkenntnissen schließen, dass beide Paradigmen Schnittpunkte haben. Diese Schnittpunkte sind zum Beispiel der Aufbau auf einzelnen Diensten. Dies zeigt jedoch auch, dass, obwohl Schnittpunkte existieren, diese sehr unterschiedlich sind.

Microservices sind eine gute Möglichkeit, um Software zu entwickeln, welche sich oft schnell weiterentwickeln und verändern muss, sowie Software die große Nutzerlasten auf einzelne zentrale Teile aushalten muss. In diesen Anwendungsfällen ist es oft problematisch monolithische Systeme einzusetzen, da sie mit steigender Größe weniger dynamisch werden. SOA hingegen ist ein System welches vor allem im Businessbereich anklang findet und nicht als IT Architekturmodell gesehen werden sollte sondern als Herangehensweise. Dies liegt an der Tatsache, dass die vorhandenen Geschäftsprozesse einen großen Teil dieses Systems ausmachen.

Bei beiden Paradigmen sollte jedoch darauf geachtet werden, dass die Organisation und der Aufbau des Basissystems (Service Discovery, Ausfallsicherheit, etc.) nicht dem eigentlichen Nutzen entgegen wirkt oder deutlich übersteigt. Keiner der beiden Paradigmen sollte eingesetzt werden, wenn kein Bedarf vorhanden ist. Zudem muss beachtet werden, dass mit dem Einsatz dieser Paradigmen neue Probleme entstehen, welche gelöst werden müssen.

Weder Microservices, noch SOA ist eine Allround-Lösung für jegliche Probleme. Es können zwar gewisse Problemfelder, wie sie schon erläutert wurden, gelöst wer-

den, jedoch nicht für alle, welche in der IT oder im Business vorhanden sind. Es sollte daher vor Einführung eines der Paradigmen eine Machbarkeitsstudie durchgeführt.

Grundsätzlich schließen sich beide Systeme nicht aus. Mit SOA sollen möglichst alle Anwendungen eines Unternehmens mit einander verbunden werden, während ein Microservice-System eine Anwendung in viele Teilanwendungen, auch Dienste genannt, aufteilt. Dementsprechend kann ein Microservice-System ein Teil von einem SOA-System sein. Gerade wenn die Frage aufkommt, wie das neu entwickelte System in die bereits bestehende Systemlandschaft eingebunden wird, kann SOA ein guter Ansatz sein um das System einzubinden.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Ziel dieser Projektarbeit war es, die Unterschiede zwischen dem Microservice- und dem SOA-Paradigma herauszuarbeiten und zu bewerten. Dafür wurde zunächst eine Ausgangssituation dargelegt und darauf aufbauend eine Problemstellung formuliert. Diese wurde anhand eines hypothetischem Beispiels in Anlehnung an ein reales Problem verdeutlicht. Anschließend wurden die zentralen Fragestellungen formuliert.

Damit diese Fragestellungen beantwortet werden können, wurden zunächst die allgemeinen Grundlagen zur Verwendung Service-orientierter Systeme erläutert. Darunter zählte unter anderem die Erläuterung der Architektur, sowie Domain-Driven Design und Bounded Context.

Im Anschluss, wurde auf das SOA-Paradigma eingegangen. Dazu zählt unter anderem der Zusammenhang zwischen der Businessabteilung und der IT-Abteilung. Daran Anknüpfend, wurde das Microservice-Paradigma erläutert.

Abschließend wurden beide Paradigmen, soweit es möglich war, miteinander verglichen und anschließend die zentralen Fragestellungen aus Kapitel 2.2 Die zu untersuchende Fragestellung versucht zu beantworten, woraus sich das Fazit ableiten lies.

8.2 Ausblick

Da in dieser Arbeit die Unterschiede zwischen dem Microservice-Paradigma und dem SOA-Paradigma ausgearbeitet wurden, kann darauf aufbauend zum Beispiel weiter auf eines der Paradigmen eingegangen werden. Es kann etwa für das Microservice-Paradigma eine Analyse erstellt werden, wie viel Aufwand es kostet, eine bestehende monolithische Anwendung, auf ein Microservice-System umzustellen.

Für das SOA-Paradigma kann analysiert werden, wie viel Aufwand das Unternehmen aufwenden muss, um die Geschäftsprozesse in ein SOA-System zu überführen. Zusätzlich kann analysiert werden, welche nötigen Schritte für diese Umstellung nötig sind.

Literaturverzeichnis

[1] *Definition: Wertschöpfungskette*. http://wirtschaftslexikon.gabler.de/Definition/wertschoepfungskette.ht – visited: 06.08.2016

[2] *Time-to-Market (TTM)*. – URL http://www.gruenderszene.de/lexikon/begriffe/time-to-market-ttm. – Stand 17.10.2016

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

[4] BUSINESSDICTIONARY.COM: *time to market*. – URL http://www.businessdictionary.com/definition/time-to-market.html

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

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

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

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

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

Anhang A

Diagramme und Tabelle