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, 25. September 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 25. September 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 25. September 2016 Stefan Kruk

Inhaltsverzeichnis

Abbildungsverzeichnis

Überblick

Kurzfassung

In dieser Arbeit sollen die Vor- und Nachteile von Service-orientierte Architektur und Microservices erörtert und anschließend miteinander verglichen werden. Dabei soll explizit auf die Unterschiede und Gemeinsamkeiten der beiden Architekturmodelle eingegangen werden. Für eine bessere Verständlichkeit wird zusätzlich ein bestimmter Prozesskontext mit beiden Modellen implementiert.

Abstract

In this Projekt i will describe the Pro and Cons of service-oriented architecture and Microservices. After that i will compare both architectural models and describe the differences and similarities between these two. For a better understanding i will implement an application with both architectural models.

Kapitel 1

Einleitung

1.1 Motivation

Die Anforderungen an Software werden zunehmend komplexer. Sie muss nicht nur funktionieren, sondern auch ggf. in kürzester Zeit erweitert oder geändert werden. Diese Anforderungen stellen besondere Herausforderungen dar. Je größer Software wird, desto schwieriger ist sie zu warten und zu pflegen.

> »Die Zeitspanne des Time-to-Market 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.«[3]

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 groß genug sein, damit möglichst zeitnah der Break-even-Point, also die Schwelle, ab der die Produktionskosten wieder eingenommen worden sind und ein Gewinn entsteht, erreicht wird.
Zudem gibt es in einem Unternehmen nicht nur ein Software Produkt, sondern eine viel Zahl verschiedener Produkte für unterschiedliche Anwendungsfälle. Dabei kann es passieren, dass einige Anwendungen gleiche Funktionalitäten beinhalten. Muss solch eine Funktionalität geändert werden, müssen dafür alle Anwendungen, welche diese Funktionen bereitstellen, geändert werden. Einfacher ist es daher, nur eine Anwendung zu haben, welches diese Funktionalität anbietet. Dadurch bedarf es nicht mehr die Anpassung von vielen Anwendungen, sondern nur noch von dieser einen.

1.2 Ausgangssituation

Bei der Ausgangssituation gehen wir von einem fiktiven Szenario aus, welche ich an das aus [8, S. 15] anlehne und im wesentlichen übernehme.

> »Die neugegründete *OnlineCommerceShop GmbH* möchte einen E-Commerce-Shop, als Hauptgeschäft betreiben. Es ist eine Web-Anwendung, die sehr viele unterschiedliche Funktionalitäten anbietet. Unter anderem zählen dazu die Benutzerregistrierung und - verwaltung, sowie die Produktsuche, Überblick über die Bestellungen und der Bestellprozess.« (vgl. [8, S. 15])

Das Unternehmen ist, auch wenn es ein reines IT-Unternehmen ist, Gewinn orientiert. A.1 zeigt den internen Aufbau eines typischen Unternehmens.
Die Geschäftsführung der GmbH hat bereits als Software-Entwickler in anderen Unternehmen Erfahrung mit dem Umgang und den Aufbau von E-Commerce-Shops gesammelt. Zu den Erfahrungen zählen unter anderem das programmieren, testen, deployen und weiterentwickeln der Anwendung. Während dieser Prozesse sind die

Programmierer zur Erkenntnis gelangt, das bei steigender Größe der Anwendung, die Aufwände für Wartung und Weiterentwicklung stark ansteigen. Dies war auch der Grund warum das Unternehmen irgendwann Insolvenz anmelden musste. Die Kosten für Wartung und einer zeitnahen Reaktion auf die Veränderungen der Nutzerbedürfnisse konnten nicht mehr getragen werden und Anbieter wie Amazon, welche deutlich schneller, als die OnlineCommerceShop GmbH, auf diese reagieren konnten, habe sie schließlich vom Markt gedrängt.
Aus diesem Grund möchte die Geschäftsführung der neugegründeten OnlineCommerceShop GmbH eine Software-Architektur wählen, welches schneller anpassbar und einfacher zu warten ist. Für sie kommen daher das Microservice-Modell und Service-orientierte Architektur-Modell infrage.

1.3 Vorgehen und Kapitel

Zunächst werden die Probleme bei der Verwendung von Monolithischen Architekturmodellen, 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. In den Grundlagen werden außerdem Werkzeuge erklärt, mit deren Hilfe solche Architekturen realisiert werden können und die Wartung vereinfachen.
Darauf folgend wird das Architekturmodell “Microservice“ unter der Verwendung von den erläuterten Werkzeugen erklärt und der Begriff Continuous Delivery eingeführt. Anschließend wird, unter Verwendung des angeeigneten Wissens aus dem Kapitel “Microservice“, auf das Architekturmodell “Service-orientierte Architektur (SOA)“ eingegangen.
Abschließend werden die gewonnen Kenntnisse zusammengetragen und Ausgewertet. Dabei sollen beide Architekturmodelle verglichen und bewertet werden. Zuletzt wird das Thema noch einmal zusammengefasst und ein Ausblick auf kommende Projekte bzw. die Bachelorarbeit gegeben.

1.4 Definitionen

Da sowohl SOA als auch Microservices zu den „Service-orientierten Architektur“ gehören, jedoch die Abkürzung SOA ebenfalls ausgeschrieben wird als Service-orientierte Architektur, wird zum besseren Verständnis die Formulierung „Service-orientierte Architektur“ nur benutzt, um ein Architekturmodell aus Services zu beschreiben. Weiterhin wird der Begriff SOA verwendet, wenn dieses spezifische Architekturmodell gemeint ist.

Kapitel 2

Problemanalyse

Wie bereits in Kapitel 1.1 Motivation erwähnt, wird Software nicht nur entwickelt, sondern muss auch gewartet und gepflegt werden. Dies ist bei großer Software nicht immer direkt möglich und bedarf größerer Aufwendungen. Diese Aufwendungen sind meistens sehr Zeitintensiv wodurch eine lange Time-to-Market Zeitspanne entsteht. Ist die TTM-Zeitspanne zu groß, sind meistens die Entwicklungskosten zu hoch, um das Projekt Wirtschaftlich durchzuführen. Zudem kann es passieren, dass in dieser Zeit neue Technologien von Konkurrenz Unternehmen veröffentlicht werden, wodurch das eigene Produkt nicht mehr Erfolgreich vermarktet werden kann, sobald dieses Fertiggestellt worden ist.

2.1 Herausforderungen

Oft passiert es, das ein Unternehmen große Vorstellungen von dem Unternehmens-Ziel hat. Dies führt häufig dazu, dass Software entwickelt wird, welches weit über den aktuellen Anforderungen hinaus gehen. 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 eigentlich für dieses Szenario ausgelegt war, geändert werden.
Wie es bereits in 1.1 Motivation erläutert wurde, ist die Zeitspanne es Time-to-

Market für ein Unternehmen von äußerster Bedeutung. Damit diese Zeitspanne möglichst gering ist, bestehen besondere Anforderungen an Software. Vor allem, wenn es um die Einführung neuer Funktionen geht entstehen meistens lange TTM-Zeiten.

2.2 Hypothetisches Beispiel in Anlehnung an ein reales Problem

Ein Modernes und aktives Internet-Unternehmen ist eBay. eBay wurde 1995 von Pierre Omidyar unter dem Namen *ActionWeb* gegründet und wurde 1997 in eBayumbenannt. eBay wurde als Monolithische Perl Anwendung implementiert (siehe [2]). Mit der Steigerung der Reichweite und der täglichen Benutzung, hatte man sich dann aber dazu entschlossen auf C++ als Code-Basis umzusteigen und die Seite mit CGI zu implementieren. Mittlerweile war eBayein großes und sich rasant entwickelndes Unternehmen. Das bedeutet aber auch, dass eBayständig auf das Verhalten der Nutzer reagieren und sich anpassen muss. Man versuchte also eine Monolithische Applikation mit der Fähigkeit auszustatten, auf Änderungen schnell zu reagieren, implementieren und deployen. Aber ein Monolith zu deployen, bedeutet, entweder die gesamte Infrastruktur für Wartungszwecke offline zu nehmen und die Applikation neu zu deployen, oder Server im Parallel betrieb laufen zu lassen und jeden neuen Traffic auf die neue Version zu routen. 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 Änderungen sind jedoch zeitaufwendig und schwierig, denn nicht nur das deployen könnte Probleme bereiten, sondern auch die darauf folgende Ausführung des Programms. Ändert man Code in Monolithischen Applikationen kann das auch Auswirkungen auf bestehende Teile des Codes haben, welche vorher, ohne Probleme, funktioniert haben. Werden Tests vernachlässigt oder wird nicht ausreichend getestet, kann es leicht passieren, dass sich ungewollt Fehler einschleichen, wodurch dann eine Version in betrieb genommen wird, welche Fehler enthält. Darauf folgend müssten diese wieder behoben werden und die Anwendung

erneut deployed werden.
Im Falle einer Monolithischen Architektur bedeutet das viele Änderungen und neue Features. Oft passiert es daher, dass Code Stücke 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 um so schwerer diese zu finden und zu beheben. Schließlich hat sich eBayentschlossen ihre Änwendungïn Java neu zu implementieren. Dieses mal jedoch mit dem Hintergrund einer leicht erweiterbaren und wartbaren Architektur.

2.3 Die zu untersuchende Fragestellung

Das Problem besteht darin, ëine Anwendung"flexibel und einfach erweiterbar zu gestalten, damit ein Unternehmen schnell auf Änderungen und die veränderten Bedürfnisse der Nutzer reagieren kann. 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, diese Codestücke in eigene 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 aufrufe kann, um die nötigen Informationen zu erhalten. Hierbei spricht man von einer Service-orientierten Architektur. Ein Architekturmodell welches dieser Architektur zur Grunde liegt ist die Microservice-Architektur. Ein anderes Architekturmodell ist die SOA-Architektur.

Beide Architekturmodelle basieren auf einen Service-orientierten Ansatz. Es stellen sich dementsprechend folgende Fragen:

1. Wo liegen die Unterschiede zwischen diesen beiden Modellen?

2. Welche Vor- und Nachteile hat das jeweilige Modell?

3. Gibt es Grenzen oder Beschränkungen bei der Benutzung eines Modelles?

4. In Wie weit helfen die Architekturmodelle die angesprochenen Problematiken aus Kapitel 1.1 Motivation zu lösen?

Kapitel 3

Allgemeine Grundlagen zur Verwendung von Service-orientierten Systemen

Software zu entwickeln ist nicht immer einfach. Umso größer diese ist, umso mehr Probleme können auftreten. Es bedarf einer genauen Planung und Verständnis von Infrastruktur um eine Software mit allen Anforderungen, zufriedenstellend zu implementieren.
Vor allem wenn es um die Weiterentwicklung und Wartung von Software geht, können große Probleme auftreten. Wurde die Architektur nicht gut gewählt oder schlecht umgesetzt, kann es das weitere Vorgehen stark beeinträchtigen, bis hin zum unmöglich machen. Es wurden daher Software-Architekturen entwickelt, welche flexibel und einfacher zu ändern sind. Außerdem kann in diesen Architekturen neue Funktionen deutlich schneller hinzugefügt werden.

3.1 Die Service-orientierte Architektur

Service-orientierte Architekturen können, wenn sie richtig angewendet werden, sehr flexibel und schnell änderbar sein. Diese Architekturen zielen, wie der Name schon sagt, auf eigenständige Dienste ab, welche durch verschiedene Kommunikationskanäle miteinander kommunizieren können und dadurch die gewünschten Geschäftsprozesse abbilden.

»Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen«[7, S. 2]

Anstatt eine einzige große Anwendung ein zu setzten, setzt man auf viele kleine, verteile, autarke Anwendungen, welche jeweils Schnittstellen nach außen hin anbieten damit der Service genutzt werden kann. Diese Schnittstellen können unter anderem durch REST-HTTP angeboten werden. Durch die verteilten Anwendungen funktioniert das System auch dann noch, wenn einzelne Dienste nicht verfügbar sind, jedoch bringt es ebenfalls die typischen Probleme von Verteilten Anwendungen mit sich, welche in **?? ??** noch genauer erläutert werden.

3.1.1 Das Gesetzt von Conway

Wie in [8, S. 39 ff.] beschrieben, stammt das Gesetzt von dem amerikanischen Informatiker Melvin Conway und besagt:

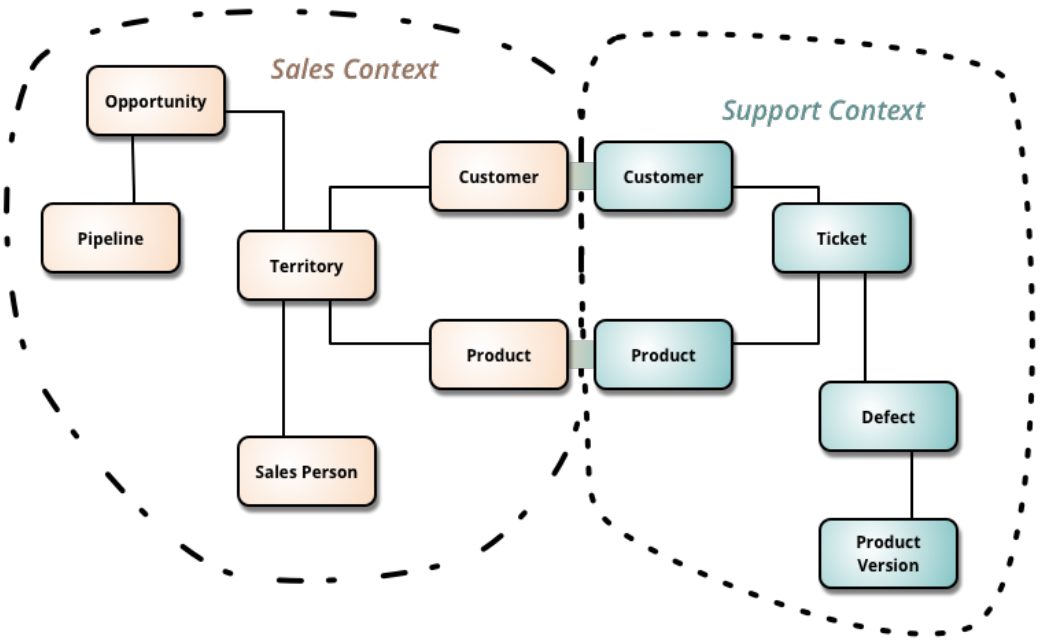
Organisationen, die Systeme designen, können nur solche Designs entwerfen, welche die Kommunikationsstruktur dieser Organisationen abbilden.

"Conway möchte damit ausdrücken, dass die internen Kommunikationswege wichtig bei der Planung der Architektur ist. Jedes Team innerhalb einer Organisation trägt bei 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."[8, vg. S. 39]

3.1.2 Domain-Driven Design und Bounded Context

Arbeitet man mit Service-Orientierten Architekturen, versucht man Services, 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.«[4]



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.

Kapitel 4

SOA

Der Begriff SOA ist nicht eindeutig definiert. Je nachdem welche Person in einem Unternehmen man fragt, erhält man unter Umständen eine komplett andere Definition. »Die Meisten Definitionen stimmen jedoch zum größten Teil überein und stehen nicht im Konflikt mit einander.«[6, vgl. Seite 6]
Für einen Kaufmann ist SOA etwas anderes als für einen Analysten, damit SOA jedoch verstanden werden kann, werden zunächst einmal einige Definitionen nach [6] genannt:

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

Jeder der genannten Rollen hat eine eigene klare Definition von dem was Service-orientierte Architektur ist. Jeder der Definitionen ist jedoch nur ein Teil dessen für was SOA verwendet werden kann.

4.1 Grundlagen

Bevor jedoch damit angefangen werden kann SOA in ein Unternehmen einzuführen, muss zunächst einmal geklärt werden, welches Ziel hinter SOA en stehen. Zunächst sei erwähnt, dass SOA zu den SService-orientierten Architekturenßählt und dem Prinzip der verteilten Anwendung unterliegt. Das Ziel von SOA ist nicht die Entwicklung zu vereinfachen oder voran zu bringen. SOA soll die Unternehmensweiten Geschäftsprozesse optimieren.

Oft existieren bereits verschiedene Anwendungen wie ERP- oder COBAL-Systeme die in den jeweiligen Unternehmensprozessen eingesetzt werden. 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 busi-

ness across its many disaggregated parts has been a prime motivator for change in the IT infrastructure at these companies.«[5, S. 17]

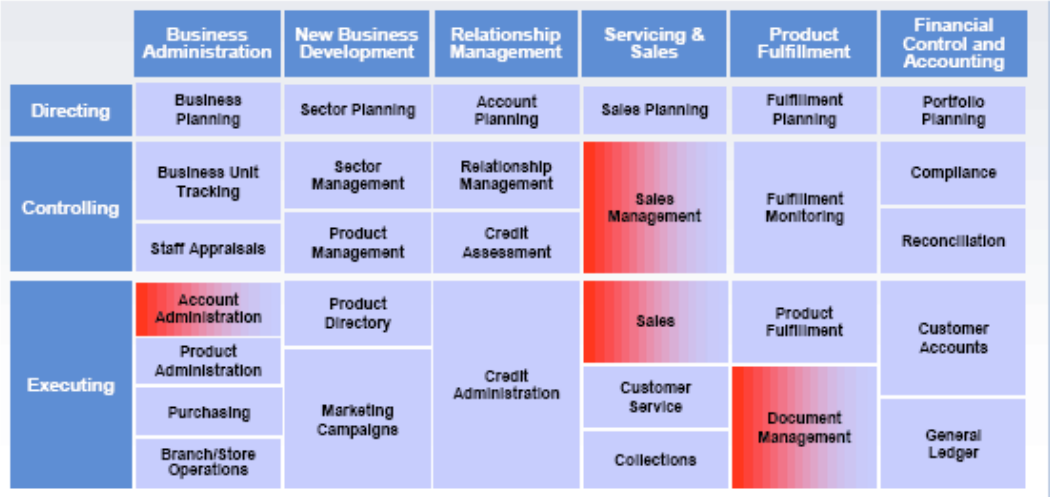
4.1.1 Business und IT

Mit SOA wird die 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, das 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 sichtbar ist.

4.1.2 Unternehmens Komponenten

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. Daraus ergeben sich anschließend die Komponenten, welche unter einander kommunizieren müssen. Die in der Abbildung Rot dargestellten Komponenten sind schließlich das Resultat aus der Analyse.

4.1.3 Wertschöpfungskette

»Die Wertschöpfungskette stellt die zusammenhängenden Unternehmensaktivitäten des betrieblichen Gütererstellungsprozesses grafisch dar.«[1]
SOA soll dabei helfen die Wertschöpfungskette sowohl darzustellen, als auch sie zu verwalten und damit zu optimieren. In Abbildung 4.1 sind die zentralen Komponente eines Geschäftsprozesses rot markiert. Mit Hilfe von SOA werden aus diesen Komponenten eine Wertschöpfungskette gebildet.

4.2 Architektur

Die Architektur in einem SOA-System unterliegt der einer SService-orientierten Architektur". Es existieren einzelne Dienste die verschiedene Funktionalitäten bereitstellen. Dienste können dabei ERP-Anwendungen, COBAL-Systeme oder eigene erstellte Applikationen sein. Dabei soll wie in jeder SService-orientierten Architektur"die einzelnen Dienste wiederverwendbar und nur lose gekoppelt sein. Außer-

dem sollen keine zwei Dienste existieren, welche die selbe Funktionalität bereitstellen.

Damit zum Beispiel ERP-Anwendungen ihre Funktionalitäten über Schnittstellen bereitstellen können, ist es oft notwendig Adapter für diese Systeme zu erstellen. Ist einmal ein Adapter erstellt worden, kann dieser von da an immer wieder neu benutzt werden. Die Abbildung der Geschäftsprozesse erfolgt darauf in einem sogenannten Enterprise-Service-Bus (Kurz ESB), welcher die einzelnen Services Orchestriert (siehe Abbildung 4.3) und die Kommunikation steuert.

4.2.1 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.

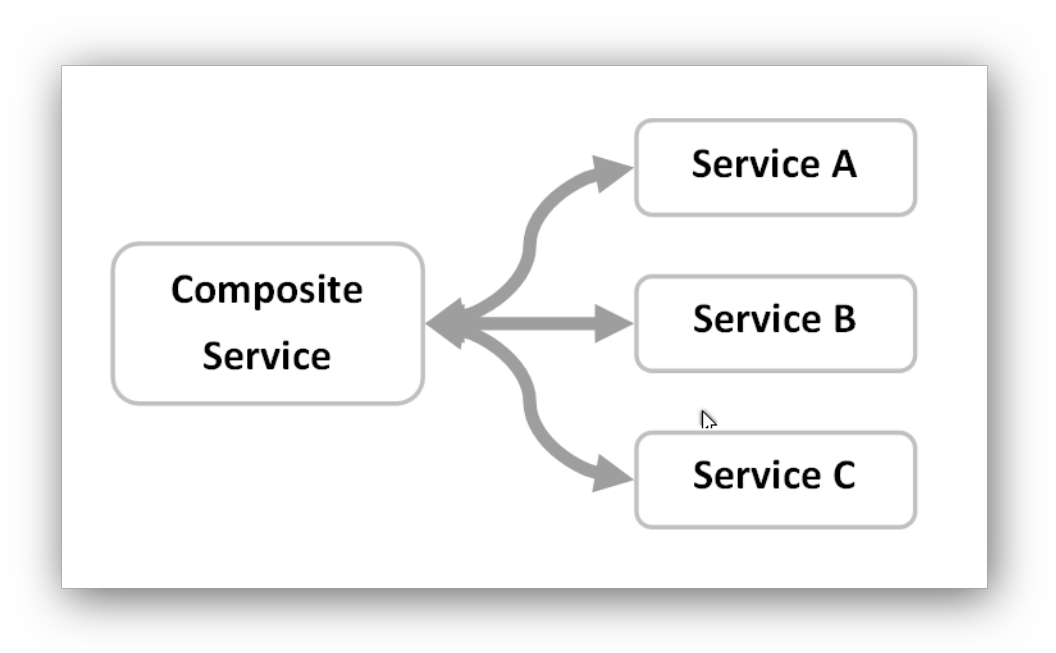


Abbildung 4.2: Orchestration

Wie die Abbildung zeigt besteht **keine** Verbindung zwischen:

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

Nur der „Composite Service“ nutzt die anderen Services, um den Geschäftsprozess abzubilden. Diese Art der Kommunikation nennt man Orchestration.

4.2.2 Choreographie

Anders als bei der Orchestration können Services bei der Choreographie beliebig untereinander kommunizieren. Das ist vor allem dann Sinnvoll, wenn verschiedene Service andere Service über Änderungen oder andere Aktionen informieren müssen.
Wie bereits oben erwähnt ist der Enterprise Service Bus die zentrale Einheit bei der Service-orientierten Architektur (SOA) über die alle Kommunikation läuft und welcher die Geschäftsprozesse abbildet. Daher ist eine Choreographie unter den Services in diesem Architekturmodell nicht möglich.

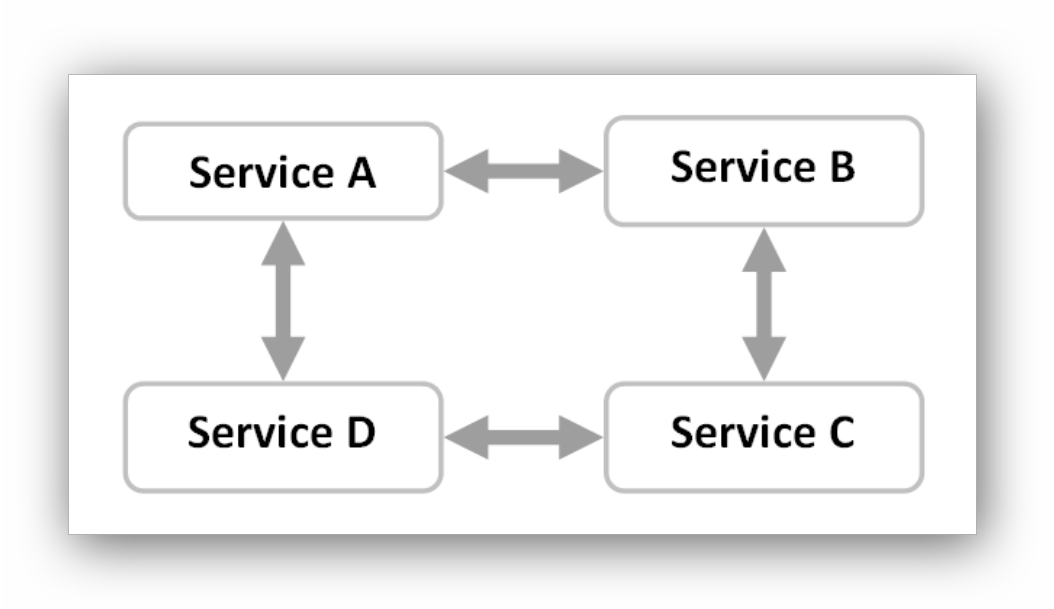


Abbildung 4.3: Choreographie

4.3 Enterprise Service Bus - ESB

Der *Enterprise Service Bus* ist die zentrale Einheit in einem SOA-System. Am einfachsten lässt sich das an einem Beispiel erklären.
Man Stelle sich eine Banking-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 bereitstellt. So können die Informationen zum Beispiel aus einem CRM-System (Kundenbeziehungsmanagement-System) stammen oder durch PHP oder Ruby erzeugt werden.

Anders als man jedoch 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:

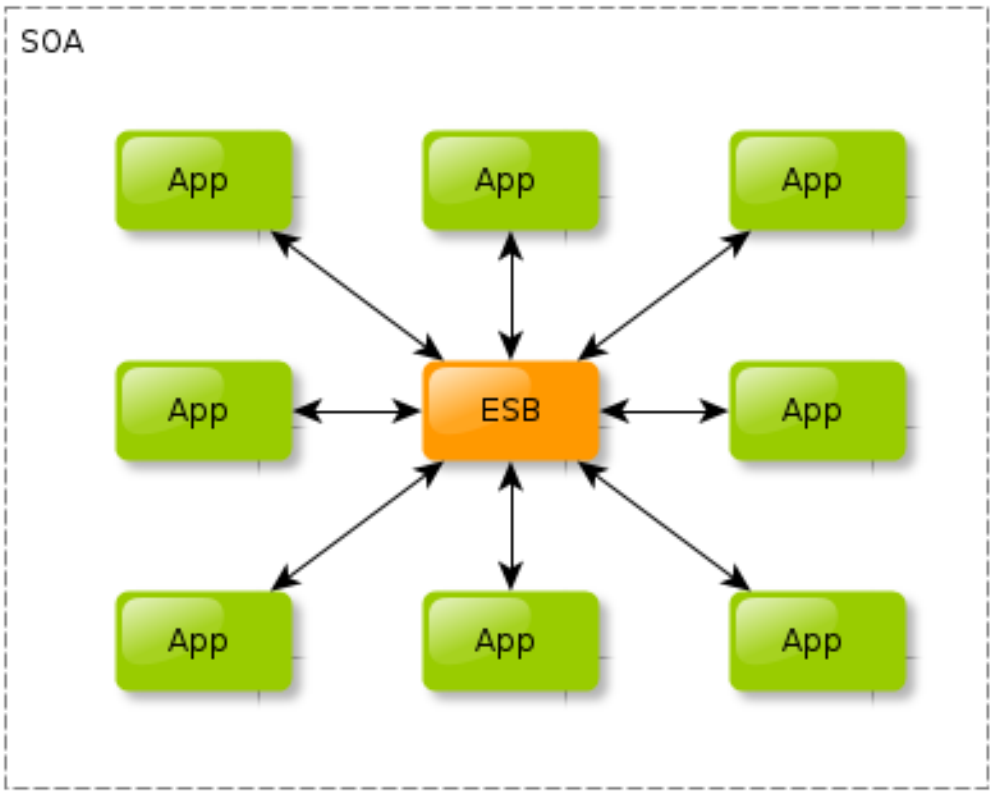


Abbildung 4.4: Enterprise Service Bus
Quelle: https://zato.io/docs/intro/esb-soa-de.html

Benötigt eine Anwendung (in der Abbildung App genannt) Informationen, konsultiert diese zunächst den ESB. Der ESB kennt die anderen Anwendungen und stellt daraufhin die angeforderten Informationen bereit.

Kapitel 5

Microservices

5.1 Überblick

> »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:«[8, 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 Textströme.

Diese Art der Aufteilung wurde schon lange von großen Unternehmen wie Amazon oder Google genutzt und wurde zu nächst Service-orientierte Architektur(SOA) genannt. Jedoch unterscheiden sich Microservices und SOA voneinander. Daher wird SOA im Nächsten Kapitel genauer erläutert und im Kapitel **??** die Ergebnisse zusammen gefasst und beide Technologien mit einander verglichen.

Der Begriff Microservices ist nicht eindeutig definiert. Als erste Näherung dienen, nach Eberhard Wolff [8, S. 2], folgende Kriterien:

- Microservices sind 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 REST sein - oder Messaging-Lösungen.

Obwohl der Begriff nicht eindeutig definiert ist, liegt jedoch bei jedem Service-orientierten Systemen, ein Verteiltes System zu Grunde und auch die damit verbundenen Probleme und Herausforderungen.

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 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 sein.

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."[8, vgl S. 55]. SCS sind jedoch nur im Microservice und nicht im SOA Umfeld zu finden. Warum wird in **?? ??** weiter erläutert.

5.2 Größe von Microservices

> »Der Name »Microservices« verrät schon, dass es um die Servicegröße geht - offensichtlich sollen die Services klein sein."[8, 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.«[8, S. 34]

Bei der Größe eines Services ist jedoch 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.

5.3 Orchestration vs Choreographie

Möchte man ein Microservice System aufbauen, stellt sich die Frage, wie einzelne Services Strukturiert werden und wie diese unter einander 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.

5.3.1 Herausforderung

Wie bereits in 5.1 Überblick erwähnt, liegt ein Verteiltes System zu Grunde und damit auch die Grundlegenden Probleme der Kommunikation (siehe [8, S. 25]). 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 Datenbank Operationen 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 Fehlers eines Microservices eine einheitlicher Rollback durchgeführt werden, um keine inkonsistente Dateien im System zu haben. 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 ode 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 können auf dem HTTP-Protokoll

aufgebaut werden. 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.

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 4.2.1 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, das nicht lange auf eine Antwort gewartet werden muss und die teilhabenden Kommunikationspartner gegenseitig kennen, jedoch bringt es auch den Nachteil, 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. Hier 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 unbedingt sichergestellt werden, dass alle Nachrichten von allen Konsumern gleichzeitig gelesen und ggf. verarbeitet werden. Jedoch können so Informationen innerhalb eines Microservice-Systems relativ zuverlässig verteilt werden. Der Vortiel 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.

Oft ist es nicht notwendig eine Antwort zu erhalten. Zum Beispiel muss eine Registrierung in unserem fiktiven Unternehmen, der OnlineCommerceShop GmbH möglich sein. Dabei sendet der Microservice der für die Registrierung zuständig ist eine einfache Event-Nachricht, wie Benutzer XY hat sich Registriert. Im Hintergrund kann dann zum Beispiel ein anderer Microservice diese Nachricht erhalten und zusätzliche Aktionen durchführen, wie erstellen des Warenkorbs.

Kapitel 6

Vergleich

Nachdem erläutert wurde, was unter **Microservice** und **Service-orientierte Architektur (SOA)** zu verstehen ist, werden nur beide Architektur Modelle mit einander Verglichen und kritisch betrachtet. Anschließend wird aus dem Vergleich ein Fazit gezogen und dieses erläutert.

Bevor die Architekturmodelle mit einander verglichen werden können, muss erwähnt werden das ein direkter Vergleich zwischen Funktionen und den Vor- und Nachteilen der jeweiligen Modelle nicht möglich ist. Außerdem ist weder der Begriff Microservice noch Service-orientierte Architektur eindeutig definiert. Literaturen wie [6] und [8] bieten einen guten Ansatz, um diese beiden Begriffe grob einzuordnen.

6.0.1 Grundlagen

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

SOA hat seine Grundlagen nicht in der IT, sondern in Geschäftswelt. Dies wird dadurch bestätigt, das 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.

Microservices hingegen stammen aus der Notwendigkeit große Software möglichst effizient zu erstellen und die Wartung zu vereinfachen. Wie auch schon in Kapitel 5.1 Überblick 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 Software Pakete, sogenannte Services, geteilt werden.

Während bei SOA das Geschäft und die Geschäftsprozesse im Vordergrund steht, will man mit Microservices die Entwicklung von Software unterstützen. Das Microservice-Modell wurde entwickelt, damit große Software möglichst einfach und schnell mit vielen Personen entwickelt werden kann. 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.

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.

6.0.2 Architektur

Ein großer Unterschied der beiden Modelle liegt in der Architektur. Da beide Modelle aus dem Gebiet der verteilten Anwendungen stammen, spielt die Kommunikation hinsichtlich der Architektur eine wichtige Rolle. Die Kommunikation der beiden Modelle soll daher zu einem späteren Zeitpunkt im Kapitel 6.0.3 Kommunikation verglichen werden und wird in diesem Kapitel außer acht gelassen.

Wie bereits erwähnt liegen beide Modelle der Architektur der verteilten Anwendungen zur Grunde, jedoch unterscheiden sie sich hinsichtlich der Strukturierung. Grundsätzlich benutzen beide Modelle eigenständige Dienste, welche sich jedoch

hinsichtlich ihrer Größe unterscheiden. Dies kommt dadurch zur Stande, dass beide Modelle einen unterschiedlichen Ansatz besitzen.

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. Dabei sind die einzelnen Dienste oft eigenständige Anwendungen wie ERP oder COBALD Systeme. Um dies zu ermöglichen wird bei SOA ein sogenannter „Enterprise-Service Bus (ESB)“ eingesetzt, welcher Frontend Dienste wie eine Website mit Datenbanken und Server-Anwendungen verbindet. Der ESB bildet dabei die Zentrale Einheit einer SOA-Architektur und dient außerdem als Dienst-Register, der jegliche Dienste in einem Unternehmen kennt und weiß wie sie anzusprechen sind.

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, wie ERP-Systeme als Services verwendet 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.

6.0.3 Kommunikation

Nachdem die Architektur Unterschiede erläutert wurden, werden nun die Unterschiede in der Kommunikation erläutert. Grundsätzlich liegt jedoch bei beiden Architektur-Modellen, wie auch schon in 6.0.2 Architektur erwähnt, das Modell der verteilten Anwendungen zur Grunde und auch die damit verbundenen Probleme.

Die Modelle unterschieden sich jedoch 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 Anwendungen wie Websiten direkt mit mehreren Services um bestimmte Informationen zu erhalten.

6.0.4 Beteiligte Personen

Neben den Technischen Unterschiede, gibt es auch Unterschiede hinsichtlich der Personengruppen die beteiligt sind. 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 schon einige Personengruppen identifiziert, als die Unterschiede der Grundlagen erläutert wurden. 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, jedoch sind hier meistens nur Administratoren und Architekten mit gemeint, da SOA keinen direkten Bezug zur Entwicklung besitzt und ohne eine Entwicklungsabteilung verwendet werden kann.

Anders als beim SOA-Modell, steht beim Microservice-Modell die Entwicklung im Vordergrund. Dadurch ändern sich die Personengruppen stark. Während beim SOA-Modell hauptsächlich Personen aus dem Business Bereich beteiligt waren, ist

beim Microservice-Modell kaum einer aus diesem Personenkreis beteiligt. Stattdessen sind Personen aus der Entwicklung und der IT beteiligt.

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 natürlich überprüft werden ob die Fragen aus Kapitel 2.3 Die zu untersuchende Fragestellung beantwortet werden können. Dabei wurde die Frage auf die Unterschiede der beiden Modelle bereits in Kapitel 6 Vergleich beantwortet. Es müssen jedoch noch folgende Fragestellungen beantwortet werden:

1. Welche Vor- und Nachteile hat das jeweilige Modell?
2. Gibt es Grenzen oder Beschränkungen bei der Benutzung eines Modelles?
3. In Wie weit helfen die Architekturmodelle die angesprochenen Problematiken aus Kapitel 1.1 Motivation zu lösen?

7.1 Welche Vor- und Nachteile hat das jeweilige Modell?

7.2 Gibt es Grenzen oder Beschränkungen bei der Benutzung eines Modelles?

7.3 In Wie weit helfen die Architekturmodelle die angesprochenen Problematiken zu lösen?

7.4 Fazit

Kapitel 8

Zusammenfassung und Ausblick

Literaturverzeichnis

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

[2] *Ebay - Wikipedia*. – Stand 25.02.2016

[3] *Time-to-Market (TTM)*. http://www.gruenderszene.de/lexikon/begriffe/time-to-market-ttm. – Stand 30.04.2016

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

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

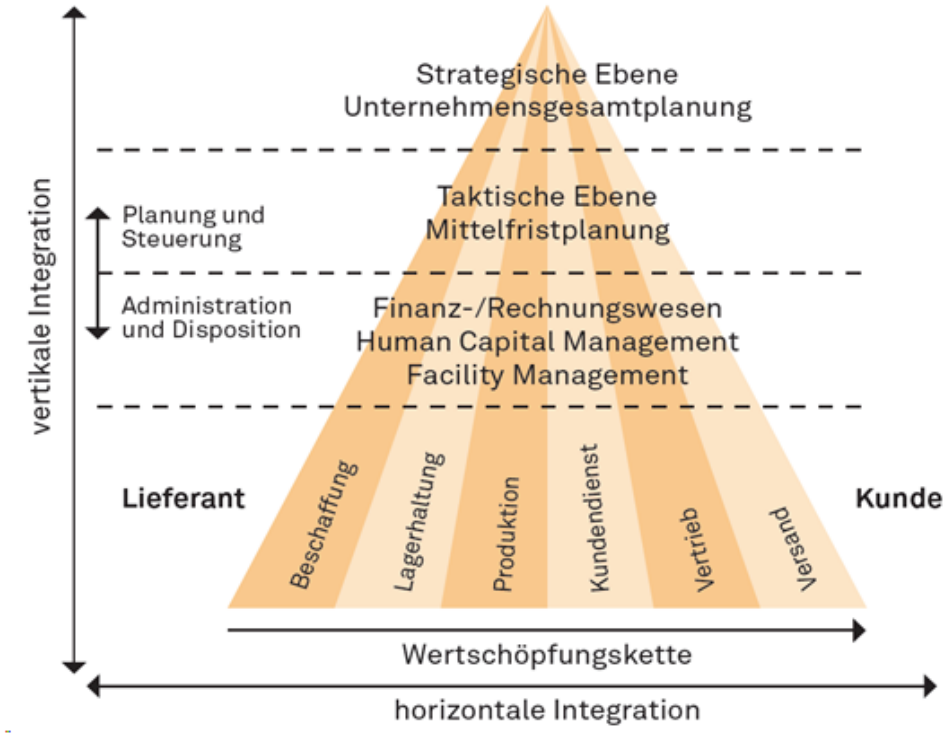
[6] KERRIE HOLLEY, Dr. Ali A.: *100 SOA Questions Asked and Answered*. http://www.professores.uff.br/screspo/100-SOA-Questions.pdf

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

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

Anhang A

Diagramme und Tabelle



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

Abbildung A.1: Darstellung einer typischen Integrations-Pyramide