Projektarbeit

Thema:

Gemeinsamkeiten und Unterschiede von Service-orientierte Architektur (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

Fachhochschule Dortmund

University of Applied Sciences and Arts

Fachbereich Informatik

Dortmund, 30. März 2016

TODOS

1. Dokument auf das Wort SOll überprüfen und ggf. durch wird/werden austauschen
2. Dokument auf das zu ofte vorkommen vom Wort "man"überprüfen
3. Reicht die Zitierform in 1.2 Ausgangssituation ?
4. richtige Überschriften Wahl (vorallem in 2 Problemanalyse)

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 30. März 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 30. März 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

In fast jedem Unternehmen wird Software eingesetzt und oft wird neue Software eingeführt oder, was deutlich seltener der Fall ist, alte Software durch neue ausgetauscht. In jedem Fall muss sich die neue Software in die bestehenden Prozesse und Architekturen integrieren lassen, damit sie genutzt werden kann.

Kleinere Unternehmen haben es daher deutlich einfacher, denn sie besitzen meistens nur einen Zentralen Server mit wenig Software. Große Unternehmen hingegen haben es da deutlich schwerer. Deren Infrastruktur basiert nicht auf einen Server, sondern auf ganze Rechenzentren weltweit, in denen die Server stehen, die sie nutzen. Damit Unternehmenssoftware miteinander, anstatt gegen- oder parallel zueinander arbeiten können, sollten Gedanken darüber gemacht werden, wie Software in dem Unternehmen aufgebaut sein soll. Große Systeme sind meist heterogen und haben gewollt oder ungewollt Redundanzen. Das kann anfangen mit der Berechnung eines bestimmten, unternehmensweiten, Zinssatzes, bis hin zu ganzen Prozessen welche doppelt in Software abgebildet werden. Dies verkompliziert die Wartung und Optimierung bestimmter Prozesse enorm.

Nehmen wir das Beispiel der Zinsberechnung. Soll diese Berechnung angepasst oder verändert werden, muss dies in jeder Software geschehen, welche diese Zinsen berechnet. Wählt man jedoch eine geeignete Softwarearchitektur, kann dieses vorgehen deutlich vereinfacht werden. Eine Lösung könnte sein, Microservices zu nutzen. Eine andere könnten Service-orientierte Architekturen sein. Bei beiden Mo-

dellen basiert das vorgehen darauf, dass kleine Services vorhanden sind, welche bestimmte aufgaben übernehmen und die entsprechenden Programme auf diese Services zurückgreifen, anstatt sie selber zu implementieren.

1.2 Ausgangssituation

Bei der Ausgangssituation gehen wir von einem fiktiven Szenario aus, welche ich an das aus [3, 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. [3, 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

Zunächst werden die Grundlagen von Microservices und Service-orientierte Architekturerläutert. Dabei werden grob die Unterschiede zwischen dem jeweiligen Architektur-Modell und eines Monolithischen-Modells geschildert werden. Darauf aufbauend wird der Kontext, welcher im Kapitel 1.2 Ausgangssituation beschrieben wurde, mit den beiden Modellen implementiert und anschließend miteinander verglichen. Zum Vergleich gehören die Unterschiede zwischen den beiden Architekturen, sowie deren jeweiligen Vor- und Nachteile. Danach werden die Vor- und Nachteile im Vergleich mit Monolithen dargestellt.

Abschließend werden die Ergebnisse präsentiert, bewertet und ein Fazit daraus gezogen. Zuletzt wird ein Ausblick über die weiteren Möglichkeiten der Architekturen erläutert.

Kapitel 2

Problemanalyse

Jedes Internet-Unternehmen fängt klein an und wächst mit der Zeit. In dieser Zeit muss das Unternehmen viele Probleme bewältigen. Es muss die Produkte, welches das Unternehmen verkauft, weiterentwickeln und auf die Bedürfnisse der Nutzer reagieren. In unserer heutigen Zeit, in der das Internet das bevorzugte Informationsmedium ist, können sich Nutzer sehr schnell von einem Produkt zu einem anderen wechseln. Um das zu verhindern, muss ein Produkt immer aktuell sein.

2.1 Herkömmliche Produkte

Herkömmliche Software-Produkte wurden auf CDs/DVDs verkauft und änderten sich daher nie. Um ein aktuelles Produkt zu erhalten, musste die Software in einer aktuellen Version erworben werden und auch diese änderte sich dann nicht. Ggf. wurden Patches zu einem billigeren Preis angeboten, wodurch eine Software geändert und mit neuen Features ausgestattet werden konnte.

Die Informationen welche den Nutzern zur Verfügung standen war außerdem sehr gering, wodurch ein Produktwechsel selten war. Durch die kosten, welche aufgewandt werden mussten, um ein Produkt zu kaufen, war es sehr selten, dass Nutzer sich schnell entschieden ein anderes Produkt zu nutzen.

2.2 Derzeitige Produkte

Wie bereits erwähnt ist das Internet das heute bevorzugte Medium, um Informationen über Produkte und andere Dinge zu erhalten. Nutzer können innerhalb von Minuten Produkte vergleichen und durch Demos diese testen, meistens sogar mit vollem Funktionsumfang. Durch SaaS (Software-as-a-Service) ist auch das wechseln von Produkten nach dem "bezahlen"möglich, da man beim SaaS-Model nur das Zahlt, was man auch wirklich nutzt.

Aus genau diesen Gründen, muss sich ein Internet-Unternehmen sehr schnell anpassen und auf die Bedürfnisse der Nutzer reagieren. Schafft ein Unternehmen dieses nicht, kann es schnell dazuführen, dass es Insolvent wird.

2.3 Aktuelles Beispiel: eBay

Ein Modernes und aktives Internet-Unternehmen ist eBayund darf man der Seite [1] glauben, so startet die Seite 1995 als Monolithische Perl Applikation. Selbst wenn diese Informationen nicht auf eBayzutreffen sollten, so dient es uns doch als gutes Beispiel einer realen Problemstellung.

Laut Wikipedia (https://en.wikipedia.org/wiki/EBay) wurde eBay1995 von Pierre Omidyar unter dem Namen *ActionWeb* gegründet und wurde 1997 in eBayumbenannt. eBaywurde wie oben schon angesprochen als Monolithische Perl Anwendung implementiert. 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.

2.4 Das Problem

Das Problem besteht also 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. Hier steigen wir in Modularität ein, denn damit eine Software möglichst einfach Wartbar ist, sollte jeder Code mit einem bestimmten Kontext in ein eigenes Modul gepackt werden. So können zum Beispiel alle Codestücke einer bestimmten Berechnung in ein Modul gepackt werden, wodurch diese dann nur an einer zentralen Stellen geändert werden muss. Wenn die Module jedoch auch von verschiedenen Applikationen genutzt werden, müssen diese in Libraries ausgelagert werden. Auch hier hat man wieder nur

eine zentrale Stelle, an der Änderungen vorgenommen werden müssen, um die Berechnung anzupassen oder zu ändern. Jedoch besteht hier das Problem, dass wenn Änderungen durchgeführt werden, diese noch lange nicht in jeder laufenden Applikation zu finden sind. Dazu müssen diese nämlich erst einmal neu gebaut und deployed werden, damit diese Produktiv werden.

Um dieses Vorgehen zu vereinfachen hat man sich dazu entschieden, diese Libraries in eigene Applikationen (Services) 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 Service aufrufe kann, um die nötigen Informationen zu erhalten. Diese Services werden auch als Microservices bezeichnet, da sie nur einem bestimmten Zweck dienen, dafür ihre Aufgabe aber besonders gut erledigen. Hierbei kann man unter anderem von einer Service-orientierte Architektur (SOA) sprechen. Es sollte jedoch darauf geachtet werden, dass weder zu viel, noch zu wenig in ein Service gepackt wird. Welche Größe genau richtig ist, wird im Kapitel 3 Grundlagen weiter erläutert.

2.5 Herausforderung

Bevor ein Programm entwickelt werden kann, müssen verschiedene Schritte durchlaufen werden. Es muss zunächst ein bedarf für die Software bestehen. Sei es Unternehmens-Software, welche eingesetzt wird um eine bestimmte Aufgabe zu erledigen oder aber um eine neue Technologie zu testen, ohne das diese Produktiv eingesetzt wird. Im jeden Fall benötigt man ein Kontext für das zu entwickelnde Programm und damit auch die Anforderungen an dieses. Nach dem erfassen dieser, muss die interne Architektur der Software geplant und schließlich umgesetzt werden.

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. Das selbe gilt, wenn man eine Architektur verwenden möchte, mit der man nicht vertraut ist. Wer zum Beispiel wenig mit Microservices und Service-orientierte Architektur (SOA) gearbeitet hat, aber es trotzdem verwenden möchte, weil es die Probleme lösen kann, welche später auftreten könnten, muss damit rechnen, dass

Kapitel 3

Grundlagen

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 Architektur

Microservice-Architekturen und Service-orientierte Architektur(SOA) Architekturen können, wenn sie richtig angewendet werden, sehr flexibel und schnell änderbar sein. Beide Architekturen zielen, wie der Name schon sagt, auf eigenständige Services ab, welche durch verschiedene Kommunikationskanäle miteinander kommunizieren und dadurch die gewünschten Geschäftsprozesse abbilden. Ëin Programm soll nur eine Aufgabe erledigen, und das soll es gut machen"[2, 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 Services nicht verfügbar sind, jedoch bringt es ebenfalls die typischen Probleme von Verteilten Anwendungen mit sich, welche in **?? ??** noch genauer erläutert werden.

Services kann man 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."[3, vgl S. 55]. SCS sind jedoch nur im Microservice und nicht im SOA Umfeld zu finden. Warum wird in **?? ??** weiter erläutert.

3.1.1 Das Gesetzt von Conway

Wie in [3, 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."[3, vg. S. 39]

3.2 Werkzeuge

Anders als bei Monolithischen-Projekten, bestehen Service-Orientierte-Projekte aus vielen kleinen Services, die alle einzeln verwaltet werden müssen. Das sie laufen reicht nicht aus, sie müssen auch geupdated und gewartet werden.

Im folgenden sollen daher einige Werkzeuge vorgestellt werden, welche das bauen, prüfen, ausliefern und verwalten der einzelnen Services vereinfacht und automatisiert. Einige dieser Werkzeuge werden für die Umsetzung, der in 1.2 Ausgangssituation vorgestellten Anforderung, eingesetzt und in dem dafür vorgesehenen Kapitel weiter erläutert.

3.2.1 Jenkins

Jenkins ist ein in Java geschriebenes, webbasiertes Software-System, für Continuous Integration (CI) Continuous Delivery (CD). Es kann unter anderem durch Plugins um weitere Funktionen erweitert werden. Mit Hilfe von Jenkins sollen das Bauen, Testen und Ausliefern automatisiert werden.[1]

3.2.2 Puppet

Puppet ist ein Systemkonfigurationswerkzeug. Es soll durch Konfigurationsdateien eine Standardisierte Installation von Software ermöglichen. Über ein Master-Slave System können diese dann via Netzwerk auf mehreren Computern/Servern verteilt werden.

3.2.3 Docker

Docker soll die Auslieferung von Anwendungen vereinfachen, in dem die Anwendung in ein Container isoliert wird. Die so entstehende Datei wird Image genannt. Zum Ausliefern der Anwendung reicht es, wenn das Image an den jeweiligen Bestimmungsort kopiert und dort ausgeführt wird. Das kann zum Beispiel durch Puppet geschehen. Dies ermöglicht außerdem, dass eine Anwendung auf verschiedenen

[1] [2, vgl. S. 98 ff.]

Servern exakt gleich ausgeführt wird. Docker setzt dabei auf Linux Container. Dadurch ist nur eine Instanz des Kernels im Speicher, jedoch sind die Prozesse, Benutzer, Dateisystem und Netzwerk von einander getrennt. Das ermöglicht außerdem, dass mehrere Docker Images zusammen arbeiten.[2]

3.2.4 Vagrant

Mit Hilfe von Vagrant kann man einfach und schnell Virtuelle Maschinen standardisiert aufsetzten. Dabei nutzt das Werkzeug fertige Betriebssystem Images, wie sie zum Beispiel mit VirtualBox erstellt werden können. Die Virtuelle Maschine wird durch das sogenannte Vagrantfile konfiguriert. Innerhalb dieser Datei kann zum Beispiel mit Puppet zusätzliche Software installiert werden. Dadurch kann schnell ein neues Produktiv- oder Testsystem aufgebaut werden. [3]

3.2.5 Swagger

Die bisherigen Werkzeuge dienten dazu, Services auszuliefern, aber sie müssen, wie bereits weiter oben erwähnt, Schnittstellen anbieten, damit sie genutzt werden können. Diese müssen jedoch auch dokumentiert sein, damit ein Entwickler weiß, welche Schnittstelle er wie anzusprechen hat. Durch Swagger ist eine einfache und zentrale Dokumentation von REST-HTTP Schnittstellen möglich.

3.2.6 Service Discovery

Mit Swagger haben wir die Schnittstellen dokumentiert, jedoch kann es bei vielen Services problematisch werden, den richtigen zu finden. Dafür sind sogenannte Service Discovery Werkzeuge wie Spring Eureka von Netflix oder Apache Zookeeper zuständig. Sie können außerdem dafür sorgen, dass ein Programm automatisch einen anderen Server bekommt, wenn ein genutzter Service ausfallen sollte. [4]

[2] [2, vgl. S. 53 ff.]
[3] [2, vgl. S. 49 ff.]
[4] [3, vgl. S. 326 ff.]

3.2.7 ELK (ElasticSearch, LogStash, Kibana)

Durch Service-Orientierte Systeme gibt es viele verteilte Anwendungen. Um sie zu warten und Fehler beheben zu können, müssen die Protokoll Dateien ausgewertet werden. Da jedoch die Services im Netzwerk verteilt sind, sind auch die Protokolle verteilt. Durch den ELK-Stack ist ein einheitliches Logging möglich. Mit Hilfe von *Logstash* können die Log-Dateien im von Servern im Netzwerk eingesammelt und mit *Elasticsearch* zentral gespeichert werden. *Kibana* ist eine Weboberfläche, mit der die gespeicherten Protokoll Daten in ein für Menschen leserliches Format angezeigt und ausgewertet werden kann. [5]

[5] [3, vgl. S. 244 ff.]

Kapitel 4

Microservices

4.0.8 Herausforderung

Wie in [3, S. 25] beschrieben, besteht ein wesentliches Problem in der Kommunikation zwischen den Services. 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

4.1 Continous-Delivery

"Wer bisher nur Deployment-Monolithen betrieben hat, ist bei Microservices damit konfrontiert, dass es sehr viel mehr deploybare Artefakte gibt, weil jeder Microservice unabhängig in Produktion gebracht wird"[3, S. 241] Ünabhängiges Deployment ist ein zentrales Ziel von Microservices. Außerdem muss das Deployment automatisiert sein, weil ein manuelles Deployment oder auch nur manuelle Nacharbeitung aufgrund der großen Anzahl Microservices nicht umgesetzt werden können"[3, S. 256]

Kapitel 5

Service-orientierte Architektur (SOA)

5.1 Vergleich

Kapitel 6

Ergebnisse

Kapitel 7

Zusammenfassung und Ausblick

Literaturverzeichnis

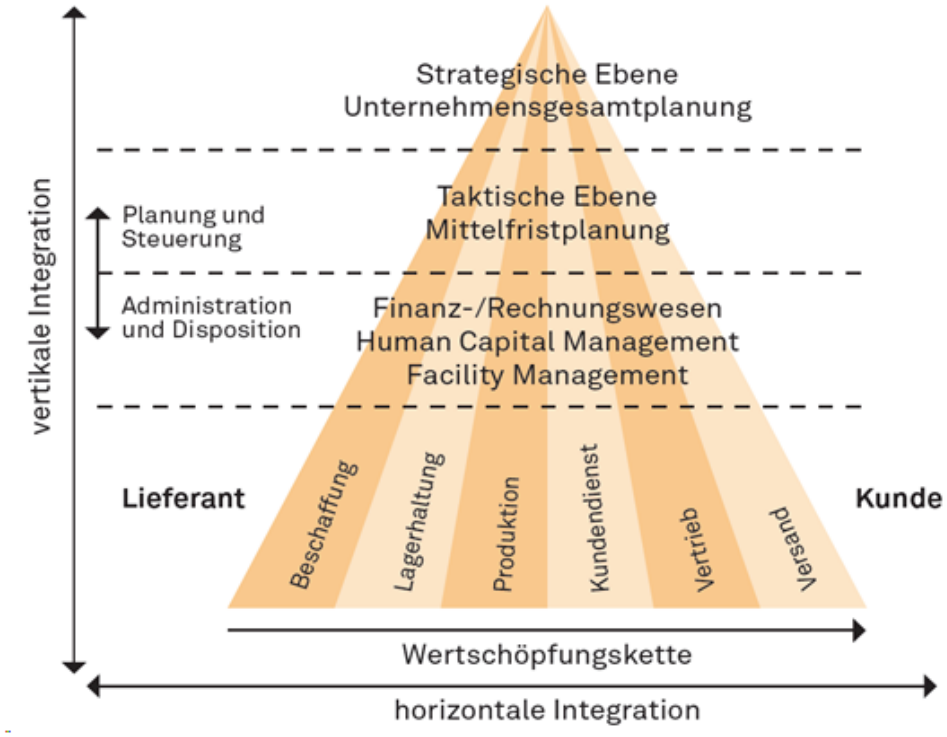
[1] HOFF, Todd: *Deep Lessons from Google and eBay on Building Ecosystems of Microservices - High Scalability* -. http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html. – 25.02.2016

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

[3] 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