Projektarbeit

Thema:

Gemeinsamkeiten und Unterschiede von Service-orientierter 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

Fachbereich Informatik

Dortmund, 21. Mai 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)
5. Anführungsstiche durch glqq und grqq ersetzten.

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 21. Mai 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 21. Mai 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

Das Internet entwickelt sich rasend schnell und damit auch Online Unternehmen, welche dieses Medium nutzten. Dadurch entsteht ein enormer Wettbewerbsdruck unter den Unternehmen, denn Nutzer können sich innerhalb kürzester Zeit entscheiden ein Produkt zu nutzen oder nicht. Um den Anforderungen gerecht zu werden und eine möglichst große Preisspanne zu haben, nutzen Unternehmen zunehmend sogenannte Cloud-Angebote, bei dem der Kunde monatlich einen Betrag für die Nutzung des Dienstes zahlt. Hat ein Unternehmen eine zulange Time-to-Market (TTM) Zeit, können sich Nutzer schnell für andere, modernere Dienste entscheiden.

"Die Zeitspanne des Time-to-Market kann 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 übergeord-

nete 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 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.

Große Unternehmen haben dieses Problem schon früh erkannt und haben sich von Monolithischen Applikationen, bei der alle Funktionen in einer großen Applikation stecken, zu dynamischeren Architekturen weiterentwickelt. Diese Unternehmen haben angefangen Funktionen in einzelne Services zu packen und diese über Schnittstellen, wie zum Beispiel REST, anzubieten.

1.2 Ausgangssituation

Bei der Ausgangssituation gehen wir von einem fiktiven Szenario aus, welche ich an das aus [6, 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. [6, 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 "Microserviceünter 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 SService-orientierte Architektur (SOA)ëingegangen.
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.

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. Dabei ist die Wahl des Architekturmodells entscheidend.

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 eBayund darf man der Seite [4] 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.

eBay wurde 1995 von Pierre Omidyar unter dem Namen *ActionWeb* gegründet und wurde 1997 in eBayumbenannt. eBay wurde wie oben schon angesprochen als Monolithische Perl Anwendung implementiert (siehe [1]). 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 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 Allgemeine Grundlagen weiter erläutert.

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 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"[5, 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."[6, 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 [6, S. 39 ff.] beschrieben, stammt das Gesetzt von dem amerikanischen Informatiker Melvin Conway und besagt:

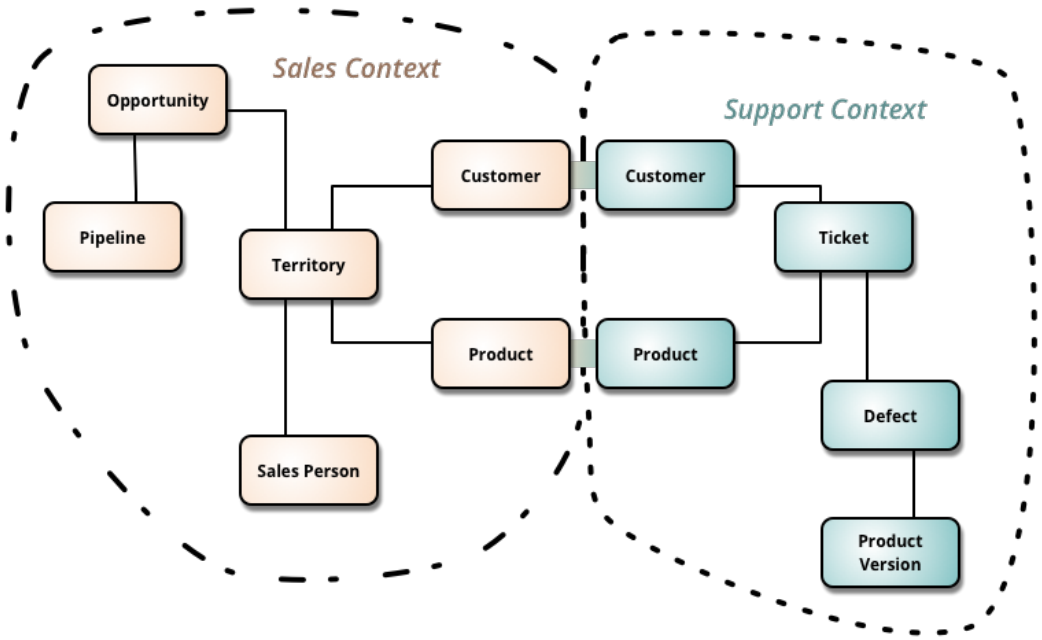
Organisationen, die Systeme designen, können nur solche Designs entwerfen, welche die Kommunikationsstruktur dieser Organisationen abbilden.

"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."[6, 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."[3]



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

Abbildung 3.1: Bounded Context

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

Lediglich zwischen den *Customer* und *Product* Services besteht eine Verbindung der beiden Prozesse.

3.2 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.

[1] [5, vgl. S. 98 ff.]

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 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] [5, vgl. S. 53 ff.]
[3] [5, vgl. S. 49 ff.]
[4] [6, 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] [6, vgl. S. 244 ff.]

Kapitel 4

Microservices

4.1 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:"[6, 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 6 die Ergebnisse zusammen gefasst und beide Technologien mit einander verglichen.

Der Begriff Microservices ist nicht eindeutig definiert. Als erste Näherung dienen, nach Eberhard Wolff [6, 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.

4.2 Größe von Microservices

"Der Name »Microservices« verrät schon, dass es um die Servicegröße geht - offensichtlich sollen die Services klein sein."[6, 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."[6, 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.

4.3 Orchestration vs Choreographie

Möchte man ein Microservice System aufbauen, stellt sich immer wieder 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.

4.3.1 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.

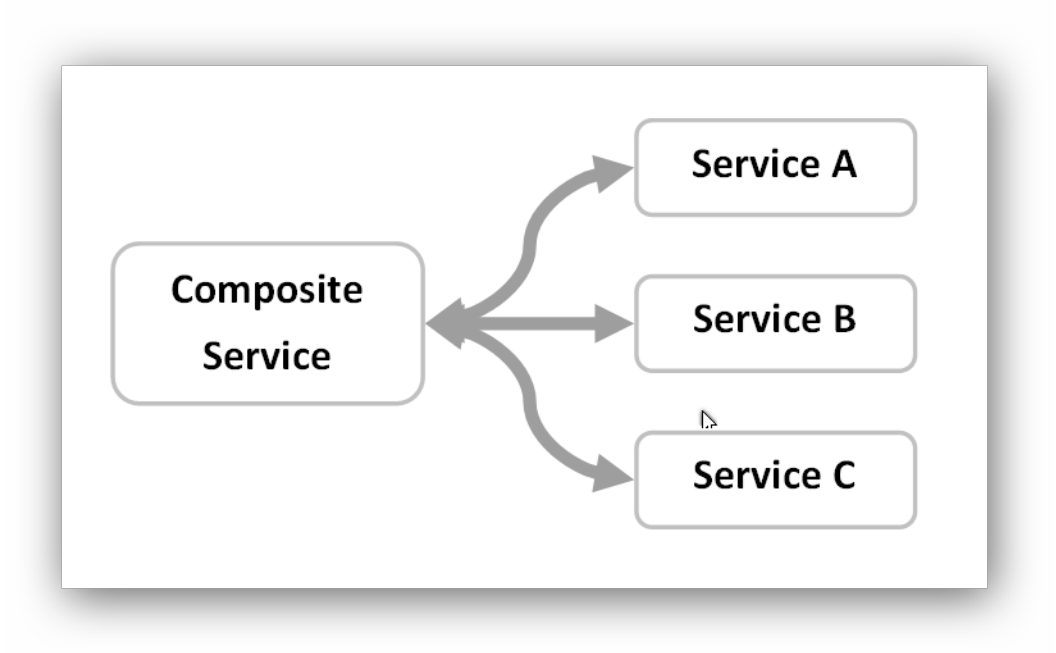


Abbildung 4.1: Orchestration

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

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

Nur der „Composite Service“ nutzt die anderen Microservice um den Geschäftsprozess abzubilden. Diese Art der Kommunikation nennt man Orchestration und wird im Kapitel 5 Service-orientierte Architektur (SOA) noch einmal behandelt.

4.3.2 Choreographie

Anders als bei der Orchestration können Microservices bei der Choreographie beliebig untereinander kommunizieren. Das ist vor allem dann Sinnvoll, wenn verschiedene Microservice andere Microservice über Änderungen oder andere Aktionen informieren müssen.

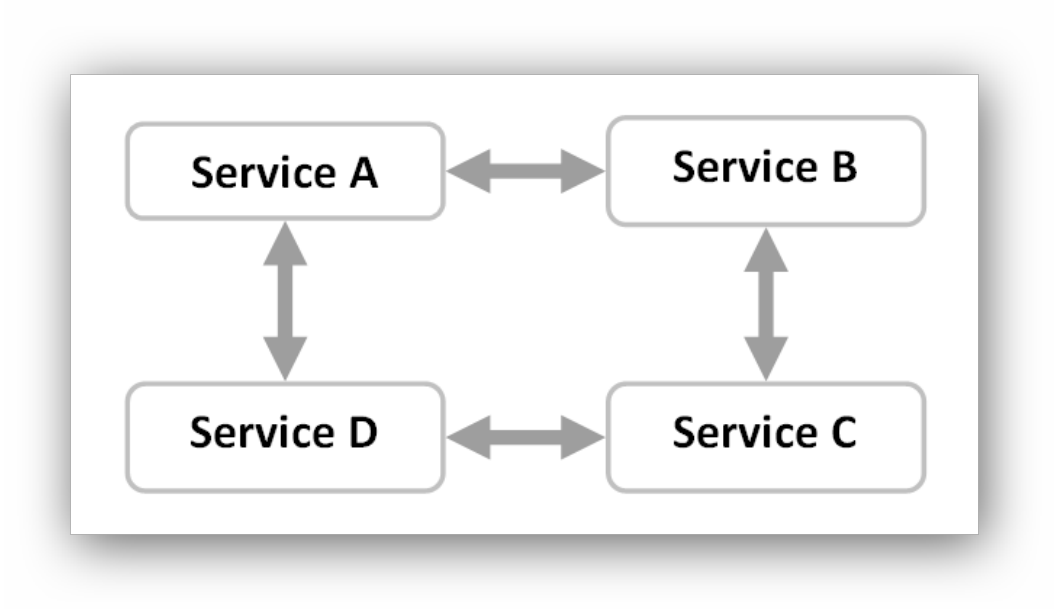


Abbildung 4.2: Choreographie

4.3.3 Herausforderung

Wie bereits in 4.1 Überblick erwähnt, liegt ein Verteiltes System zu Grunde und damit auch die Grundlegenden Probleme der Kommunikation (siehe [6, 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.

4.4 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 4.3.1 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.

4.5 Continuous-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"[6, 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"[6, S. 256]

Kapitel 5

Service-orientierte Architektur (SOA)

5.1 Architektur

5.2 EnterpriseServiceBus - ESB

5.3 Vergleich

Kapitel 6

Ergebnisse

Kapitel 7

Zusammenfassung und Ausblick

Literaturverzeichnis

[1] : *Ebay*

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

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

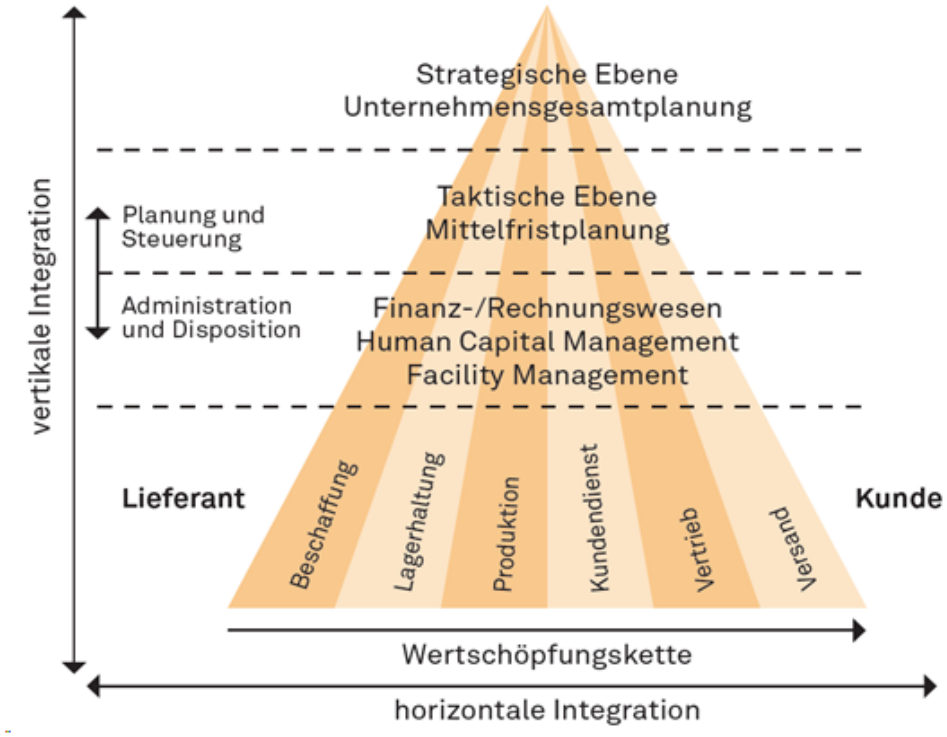
[4] HOFF, Todd: *Deep Lessons from Google and eBay on Building Ecosystems of Microservices - High Scalability* -. http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html. – Stand 25.02.2016

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

[6] 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