Projektarbeit

Thema:

Gemeinsamkeiten und Unterschiede von SOA und Microesrvices

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

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

Betreuer: Prof. Dr. Johannes Ecke-Schüth

Fachbereich Informatik

Dortmund, 1. Dezember 2016

Inhaltsverzeichnis

Überblick

Kurzfassung

Hier sollte eine halbseitige Kurzfassung der Arbeit stehen.

Abstract

Here, an abstract written in English should appear.

Kapitel 1

Einleitung

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

Dieses Konzept wurde nach und nach weiterentwickelt. Zwei Paradigmen, welche diesem Ansatz folgen sind SOA (Service-orientierte Architektur) und Microservices. Beide Paradigmen setzten auf einzelne Services, welche einzelne Fachlichkeiten beinhalten und diese über APIs nach außen anbietet.

1.1 Begriffsabgrenzungen

SOA und „Service-orientierte Architektur“

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

1.2 Zielsetzung und Aufgabenstellung

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

1.3 Vorgehensweise

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

Darauf aufbauend werden die Einsatzmöglichkeiten der jeweiligen Paradigmen analysiert

und die dafür nötigen Architekturen und Schnittstellen herausgearbeitet. Insbesondere soll ermittelt werden, welche Plattformen und Voraussetzungen für den Einsatz der jeweiligen Paradigmen notwendig sind. Zudem wird analysiert, welche Problematiken bei dem jeweiligen Paradigma

Danach werden die Vor- und Nachteile der Paradigmen, hinsichtlich der Prozessisolierung, Skalierung, Deployment, Wartbarkeit (insbesondere der Korrigierbarkeit, Erweiterbarkeit, Anpassbarkeit, Verbesserung), Entwicklung/Testbarkeit und der Bindung an Technologiestacks herausgearbeitet. Anschließend werden diese mit einander verglichen und die Gemeinsamkeiten und Unterschiede zwischen den Paradigmen herausgearbeitet.

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

Kapitel 2

Problemanalyse von Monolithischen Systemen/Anwendungen

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

2.1 Herausforderungen bei der Verwendung von monolithischen Systemen

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

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

2.1.1 Skalierung

Skalierung ist ein wichtiges Thema von Webanwendungen. Greifen viele Benutzer auf ein und die selbe Anwendung zu, so kann es passieren, dass dies das System überlastet und alle Benutzer länger auf die Antwort des Systems warten muss. Dies kann man nicht immer durch leistungsstarke Hardware ausgleichen. Oft muss daher mehrere Instanzen des Systems existieren und ein Load Balancer die Benutzer auf die Instanzen verteilen. Nicht immer ist es jedoch möglich in kurzer Zeit eine neue Instanz zu starten, sodass zu jederzeit eine feste Anzahl von Instanzen vorhanden sein muss. Dies ist nicht nur schwer zu warten, sondern auch Kostenintensiv, da die Hardware dauerhaft in gebrauch ist. PaaS Systeme, bei denen man nach genutzter Rechenleistung zahlt, können daher sehr Kostenintensiv werden, müssten zu jederzeit, zum teil ungenutzte, Instanzen eines Systemes vorhanden sein.

Kapitel 3

Allgemeine Grundlagen zur Verwendung von Service-orientierten Systemen

Ein Service-orientiertes System, ist ein Architekturmuster, bei dem einzelne Dienste (Services) ein IT-System bilden. Es liegt dem Bereich der verteilten Systemen zur Grunde und dient oft zur Strukturierung von komplexen Systemen.

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

3.1 Was ist ein Service?

Ein Service ist in einem „Service-orientiertem System“ eine Fachlichkeit eines gesamt Systems. Es wird versucht autarke Services zu erstellen, welche unabhängig von anderen Diensten verwaltet und verwendet werden können.. Die Größe von Diensten ist dabei jedoch nicht festgelegt und kann je nach Paradigma bzw. Definition und Unternehmen

variieren.

3.2 Verteilte Systeme

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

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

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

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

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

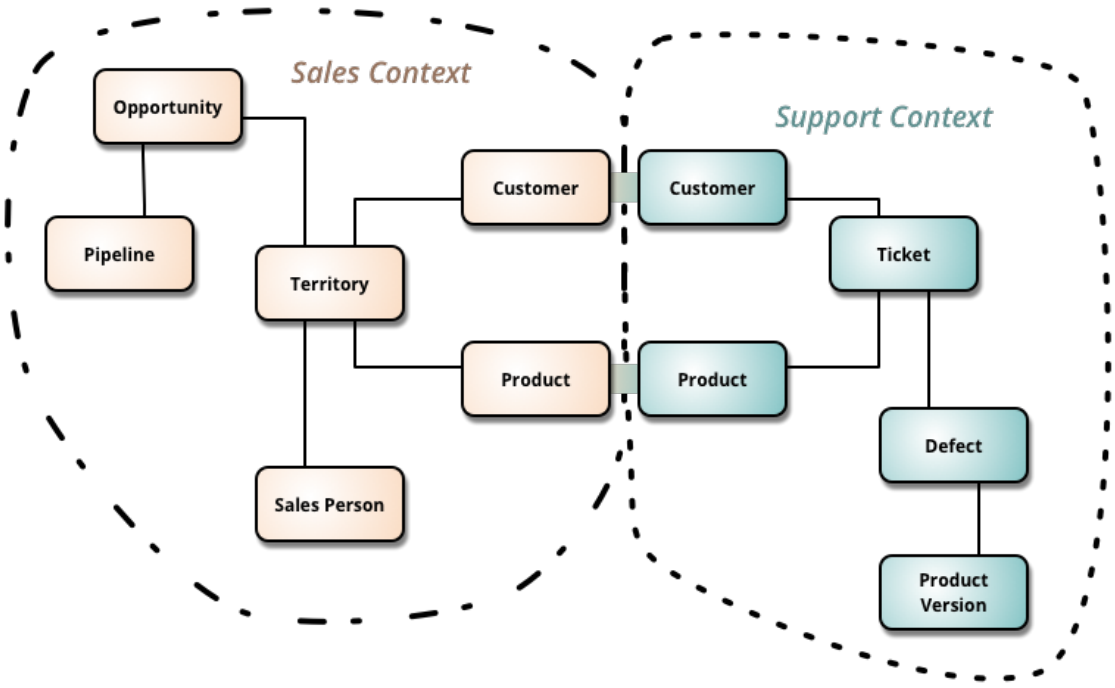
3.2.1 Domain-Driven Design und Bounded Context

Domain-Driven Design (DDD) beschreibt dabei die Herangehensweise zur Modellierung von komplexer Software. Dabei ist die Modellierung maßgeblich an die umzusetzende Fachlichkeit gebunden und wird durch diese beeinflusst. Das Ziel jeglicher Software ist es,

eine bestimmte Anwendungsdomäne zu unterstützen. Damit dies erfolgreich geschieht, muss Software harmonisch und in höchster Form interoperabel zur Anwendungsdomäne sein. Domain-Driven Design soll genau dies gewährleisten.

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

> »Bounded Context ist ein zentrales Muster in Domain-Driven Design.[..] DDD arbeitet mit großen Modellen, indem es diese in kleine verschiedene zusammengehörige Kontexte unterteilt und auf ihre Wechselwirkung unterteilt.«[2]



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

Abbildung 3.1: Bounded Context

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

3.2.2 Das Gesetzt von Conway

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

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

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

Conway möchte damit ausdrücken, dass die internen Kommunikationswege wichtig bei der Planung der Architektur ist. Jedes Team innerhalb einer Organisation trägt zu der Entwicklung der Architektur bei. Wird eine Schnittstelle zwischen zwei Teams benötigt, so müssen diese Teams auch kommunizieren können. Dabei müssen Kommunikationswege nicht immer offiziell sein. Oft gibt es informelle Kommunikationsstrukturen, die ebenfalls in diesem Kontext betrachtet werden können.

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

3.3 Kommunikation: Orchestration vs Choreographie

Orchestration

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

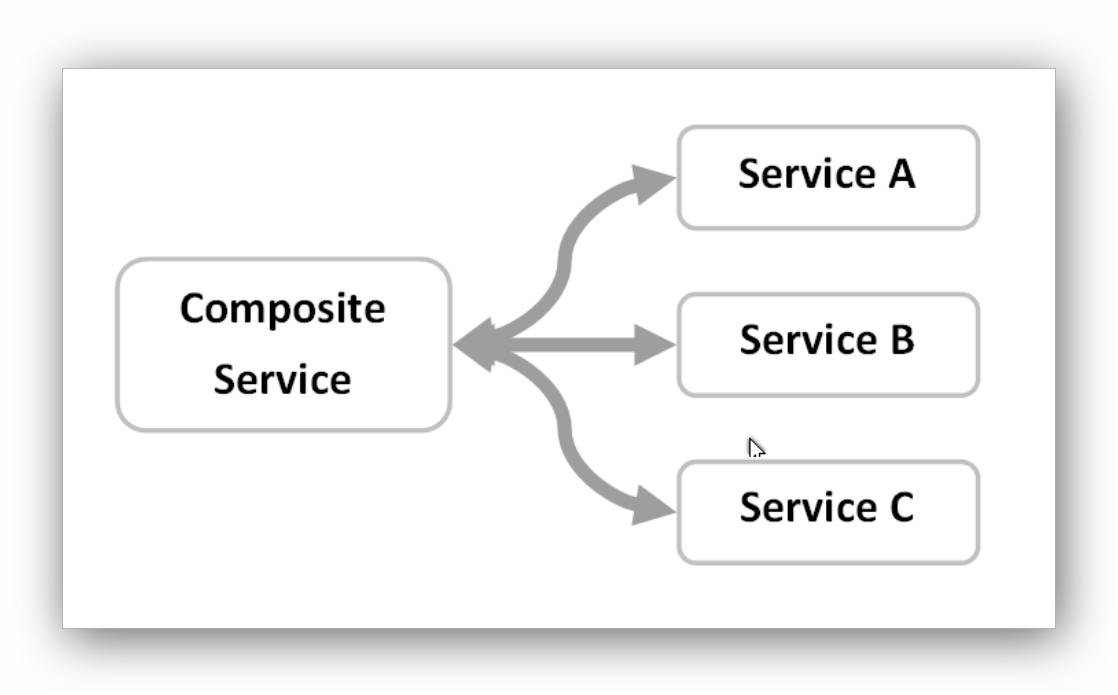


Abbildung 3.2: Orchestration

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

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

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

Choreographie

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

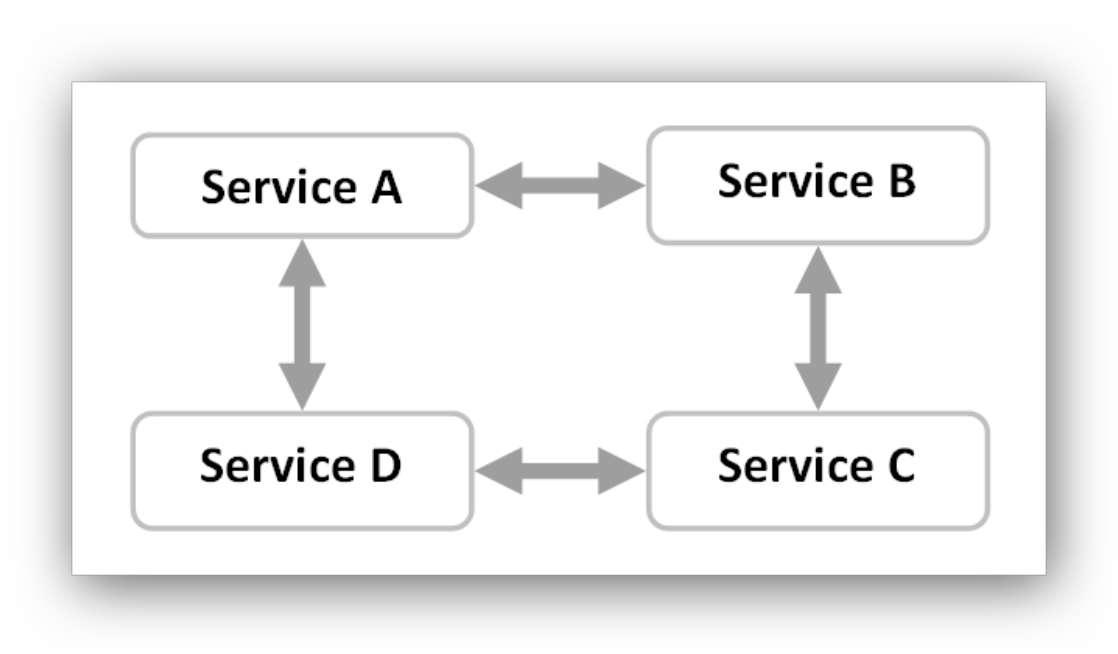


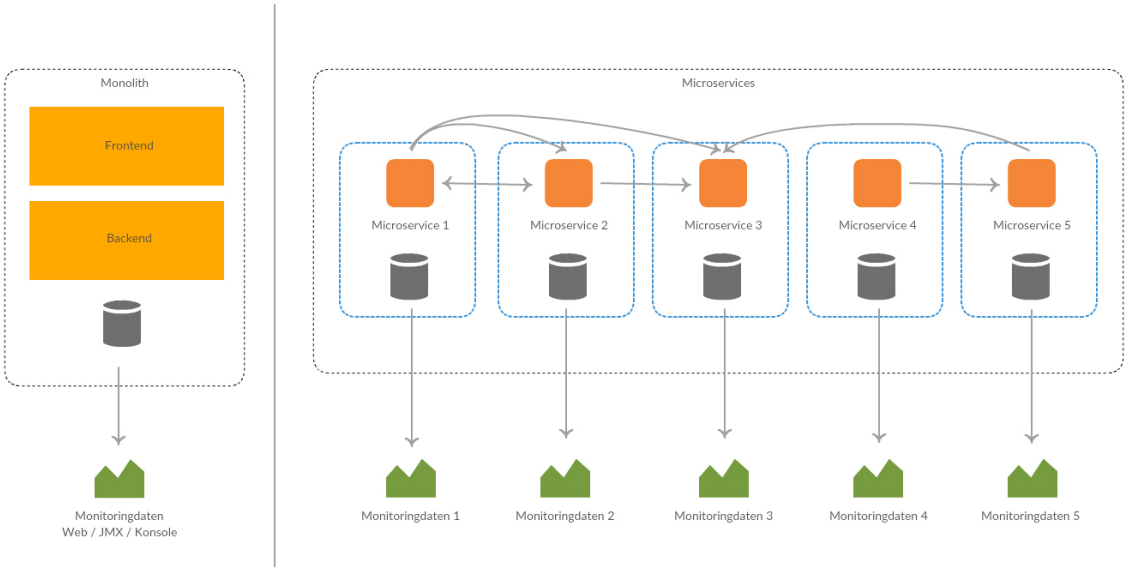
Abbildung 3.3: Choreographie

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

3.4 Abgrenzung von monolithischen Systemen

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

Während bei einem monolithischen System alle Abhängigkeiten innerhalb der Anwendung zu finden sind, besteht die Anwendung in einem Service-orientieren System aus mehreren Diensten, welche untereinander kommunizieren. Jeder Dienst bildet dabei einen gewissen Context ab (siehe Domain-Driven Design und Bounded Context). Zudem ist es dadurch möglich, dass jeder Dienst eine eigene Datenbank hat, wodurch verschiedene Datenbanksysteme (SQL und NoSQL) eingesetzt werden können. In der nachfolgenden Abbildung ist einem Monolithen, ein Microservice System entgegen gesetzt. Zusätzlich zu



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

Abbildung 3.4: Monolithisches vs Service-orientiertes System

den oben genannten unterschieden, sieht man in der Abbildung außerdem ein Monitoring-System. Während bei einem Monolithen, nur ein Monitoring-System benötigt wird, muss bei einem Service-orientiertem System, jeder Service ein Monitoring-System besitzen. Bei großen Systemen, müssen zudem die Monitoring-Informationen gebündelt und zentral einsehbar sein.

Literaturverzeichnis

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

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

Anhang A

Anhang

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 1. Dezember 2016 Stefan Kruk

Erklärung

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

Dortmund, den 1. Dezember 2016 Stefan Kruk