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, 5. März 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 5. 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 5. März 2016 Stefan Kruk

Inhaltsverzeichnis

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

Folgende Institute sind gegeben:

- Einwohnermeldeamt
- Standesamt
- KFZ-Zulassungsstelle
- Polizei

Diese Institute müssen Möglichkeiten haben, um bestimmte Daten voneinander abzufragen. Zum Beispiel muss der Fahrzeughalter und sein derzeitiger Wohnort bei einem vergehen mit dem PKW ermittelt werden können. Dazu muss die Polizei zunächst einmal mit Hilfe des Kennzeichens den Fahrzeughalter über die KFZ-Zulassungsstelle ermitteln. Anschließend muss dieser über das Einwohnermeldeamt ausfindig gemacht werden.

Ein weiteres Beispiel: Zwei Personen Heiraten Standesamtlich. Somit ändert sich der Nachnamen von mindestens einer Person. Das Standesamt muss also das Einwohnermeldeamt darüber informieren, dass sich der Nachname dieser Person/en ändert.

1.3 Vorgehen

Zunächst werden die Grundlagen von Microservices und Service-orientierte Architektur, sowie monolithischen Architekturen erläutert. Darauf aufbauend wird der Kontext, welcher im Kapitel 1.2 Ausgangssituation beschrieben wurde, mit dem Microservice-Modell und dem Service-orientierte Architektur Modell implementiert und beide Architekturen miteinander verglichen.

Abschließend wird noch einmal das Ziel dieser Arbeit erläutert. Danach 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 der Infrastruktur um eine Software mit allen Anforderungen, zufriedenstellend zu implementieren. Das heißt aber nicht, dass kleine Programme keine Probleme bereiten können.

3.1 Architektur

3.2 Swagger

3.3 Spring Cloud Service Discovery

3.4 Continous-Delivery

3.5 Weitere Werkzeuge

3.5.1 Docker

3.5.2 Vagrant

3.5.3 puppet

3.5.4 ELK (ElasticSearch, LogStash, Kibana)

Kapitel 4

Fallstudie

4.1 Microservices

4.1.1 Herausforderung

Wie in [2, 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

4.2 Service-orientierte Architektur (SOA)

4.3 Vergleich

Kapitel 5

Ergebnisse

Kapitel 6

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

Anhang A

Diagramme und Tabelle