Universität Hamburg
Fachbereich Informatik

Microservices und die Zukunf von DevOps

am Arbeitsbereich Softwarearchitektur (SWA)

Bennet Brunsen, Mathias Schmialek

28. November 2015

Blubb

Inhaltsverzeichnis

1 Einleitung

Lorem Ipsum...

1.1 Developer

Viele Betriebe wie zum Beispiel Versicherungen sind heutzutage beinahe vollständig auf Software angewiesen, die ihre Arbeit korrekt abbilden und vereinfachen. Diese Software wird von Entwicklern (Developern) mit Hilfe von Abstraktion und Programmiersprachen hergestellt. Damit die Software auch den Ansprüchen der Anwender entspricht, nimmt der Entwickler die Anforderungen vom Anwender beziehungsweise dem Auftraggeber entgegen und in Programmcode umgesetzt. Zusätzlich ist es die Aufgabe des Entwicklers den geschriebenen Programmcode sowohl manuell als auch programmatisch zu testen und damit ein korrektes Verhalten der Anwendung sicherzustellen.
Das Ergebnis der Arbeit eines Entwicklers ist ein Stück Software, das dem Kunden als Installationsdatei oder Ähnlichem übergeben wird. Der Entwickler selbst ist nicht dafür zuständig, dass die Anwendung in den laufenden Betrieb des Kunden integriert wird.

1.2 Operation

Unter Operations werden im herkömmlichen Sinne Mitarbeiter mit IT-Kenntnissen verstanden, deren Ziel es ist, den täglichen IT-Betrieb am laufen zu halten. Zu ihren Aufgaben gehören unter anderem die Bereitstellung von Hardware für Mitarbeiter des Betriebes sowie die Verwaltung von Servern. Sie sind ebenfalls dafür zuständig, Software unter anderem auf die Geräte der Mitarbeiter und auf die Server des Betriebes zu spielen. Dabei ist kann die Software entweder käuflich erworben oder im eigenen Unternehmen entwickelt worden sein. Dies wird in der Regel mit Skripten realisiert, die Operations selber anlegen.
Zu einer Software, die kommerziell erworben wurde gibt es in der Regel so gennante Service Level Agreements, die eine mit dem Softwarehersteller vereinbarte Leistungsqualität der einzusetzenden Software beschreiben. Ein Beispiel hierfür wäre, dass ein Hersteller von einer Cloudbasierten Anwendung, seinem Kunden verspricht, dass die Anwendung mit einer bestimmten Anzahl an Anfragen innerhalb einer definierten Zeitspanne zurecht kommt, sodass keine Ausfälle entstehen. Innerhalb eines Betriebes überwachen Operations die Service Level Agreements melden gegebenenfalls Verstöße ihren Vorgesetzten oder dem Hersteller der Software.
Um die Einhaltung der Service Level Agreements zu überprüfen, wird die Software mit Hilfe von Monitoring und Logging Werkzeugen überwacht. Dabei geben die Werkzeuge des Monitorings den Operations, wichtige Informationen wie zum Beispiel den Speicherverbrauch und den Status der laufenden Anwendungen. Mit Hilfe der Logging Werkzeuge, können die Operations auf Fehler innerhalb der Anwendung schließen. Für den Fall, dass die Umgebung der Anwendung falsch konfiguriert ist, wird dies üblicherweise in Log-Dateien geschrieben, die den Operations zur Verfügung stehen.

Wie bereits erwähnt, ist das Ziel von Operations den IT-Betrieb am laufen zu halten. Dies wird in der Regel als Business Continuity bezeichnet. Allerdings ist dies nicht nur mit einer für die Anwendung passend konfigurierten Umgebung möglich. Die Rechner und Server eines Betriebes müssen nach Außen abgesichert werden, um Ausfälle durch Schadangriffe so gut wie möglich zu unterbinden. Dies ist ebenfalls ein Aufgabenbereich von Operations.

1.3 Agile Softwareentwicklung

Neben der klassischen Softwareentwicklung, die nach starren Modellen wie zum Beispiel dem Wasserfallmodell arbeitet, hat sich in den letzten Jahren die agile Softwareentwicklung etabliert. Die agile Softwareentwicklung mit agilen Vorgehensweisen wie zum Beispiel Scrum oder Xtreme Programming, bindet den Kunden stärker in den Entwicklungsprozess ein und bietet die Möglichkeit schon während der Entwicklung auf das die Software Einfluß zu nehemen.
Len Bass hat bei der agilen Softwareentwicklung drei Phasen identifiziert, auf die im folgenden eingegangen wird. Die erste Phase ist die sogenannte Inception Phase. In dieser Phase werden unter anderem die Anforderungen an die Software aufgenommen und erste Modellierungsarbeiten betrieben. Zustäzlich wird in dieser Phase ein Release Plan erstellt, der mit dem Kunden abgestimmt wird.
Die zweite Phase ist die so genannte Construction Phase. In dieser Phase wird die Software nach agilen Methoden wie zum Beispiel Scrum entwickelt. Das Ergebnis dieser Phase ist ein Software Artifakt, das dem Kunden übegeben werden kann.
Die finale Phase ist die Transition Phase. In dieser Phase wird das zuvor hergestellte Artifakt beim Kunden auf die Rechner beziehungsweise Server gespielt und somit in den laufenden Betrieb integriert.

1.4 Monolith

Unter einem Monolithen wird in der Softwareentwicklung ein Architekturstil verstanden, bei dem die Software auf höchster Ebene nur als eine einzelne Komponente gesehen wird. Monolithen sind in der Regel sehr große Komponenten, die mehr als 50.000 Zeilen an Programmcode aufweisen. Darüber hinaus enthalten Monolithen die gesamte Geschäftslogik einer Anwendung. Monolithen entstehen implizit, wenn kein spezieller Architekturstil vorgegeben wird, da die Codebasis mit der Zeit durch weitere Entwicklungsarbeiten wächst. Häufig sind monolithisch entworfene Anwendungen im Serverbereich von Webanwendungen aufzufinden.
Anwendungen die nach diesem Stil entworfen wurden, weisen einige Nachteile auf. Zum einen erfordert jede Änderung wie zum Beispiel eine Fehlerkorrektur ein erneutes Deployment der gesamten Anwendung. Zusätzlich skalieren monolithische Anwendung schlecht. Sollte zum Beispiel eine logische Einheit des Monolithen wie die Datenpersistierung nicht unter der aktuellen Last bestehen, müsste eine zweite Instanz der gesamten Anwendung erzeugt werden, um die Last zu bewältigen. Dies ist nicht optimal, da eigentlich nur die Datenpersistierung mehr Instanzen benötigen würde.
Ein weiterer Nachteil, der durch die monolithische Architektur entsteht, findet sich im Release einer neuen Version wieder. Da bei einem erneuten Deployment die gesamte Anwendung erneut auf die Rechner beziehungsweise Server gespielt werden muss, sind diese in der Zeit des Deployments nicht verfügbar. Dies kann je nach Anwendungsfeld zu einem Ausfall an

Einnahmen führen. Zum Beispiel könnte ein Händler im Internet keine Produkte verkaufen, während seine Anwendung nicht verfügbar ist. Aus diesem Grund werden neue Versionen einer monolithischen Anwendung so selten wie nötig auf die Systeme gespielt. Dies führt wiederum dazu, dass eine neue Version nicht bei jedem neuen Feature entsteht, sondern erst mit einer Vielzahl neuer Features. Die vielen neuen Features können allerdings Inkompatibilitäten in der Anwendung erzeugen sowie zu einer erhöhten Ferleranfälligkeit führen. Daher erfoderdet eine neue Version einen sehr hohen Aufwand beim Testen der Anwendung. Darüber hinaus kann eine neue Version auch eine völlig anders konfigurierte Umgebung erfordern, was den Aufwand einer neuen Version ebenfalls erhöht. All dies erfordert einen sehr hohen Koordinationsaufwand und eine langfristige Planung, was wiederum zu einem langsamen Release der Anwendung führt.

2 DevOps

2.1 Motivation

2.2 Practices

2.3 Ziele

3 Microservices

In diesem Kapitel werden Mircoservices genauer betrachtet. Dazu wird zunächst der Zusammenhang zwischen DevOps und Mircoservices erläutert. Es wird gezeigt, dass Microservices optimal dazu geeignet sind die Anforderungen von DevOps zu erfüllen. Daraufhin werden Architektur, Eigenschaften sowie Herausforderungen von Mircoservices vorgestellt. Dies wird immer unter dem Gesichtspunkt betrachtet, dass Microservices für DevOps eingesetzt werden. Dementsprechend werden stets die Punkte hervorgehoben die für oder gegen den Einsatz in DevOps sprechen. Zum Schluss dieses Kapitels werden noch einige Gefahren und Antipattern vorgestellt.

3.1 Motivation

Entwicklerteams, die die DevOps Praktiken anwenden sind üblicherweise sehr klein und haben eine geringe Koordination zu anderen Teams. Dadurch haben die Teams eine begrenzte Übersicht über die gesamte Anwendung. Sobald allerdings neu entwickelte Komponenten eingesetzt werden sollen, ist es zwingend Notwendig Kompatibilität zu anderen Komponenten sicher zu stellen. Dies kann zum einen durch teamübergreifende Koordination oder durch Einsetzen einer bestimmten Architektur gelöst werden.
Des weiteren fordern die DevOps Praktiken das continuous Deployment. Dies sollte stets ohne große architekturelle Anpassungen einsetzbar sein, um so die benötigte Zeit für die Einführung einer neuen Komponente möglichst gering zu halten. Weitere Anforderungen an die Architektur sind: Unterstützung unterschiedlicher Versionen einer Komponente, sodass Teammitglieder ohne größere Koordination eigene Neuentwicklungen einsetzen können. Ebenso sollen Rollbacks möglich sein, um zum einen live testing zu ermöglichen und im Falle von Fehlern die eingesetzte Komponente rückgängig zu machen.

Diese Anforderungen werden allesamt von der Microservice Architektur erfühlt. Im folgenden Abschnitt wird diese Architektur genauer erläutert.

3.2 Architektur

Bei der Mircroservice Architektur besteht die Softwareanwendungen beziehungsweise der Business Services aus mehreren kleinen Komponenten. Diese Komponenten werden Mircoservices genannt, der eine angeschlossene kleine Funktion der Anwendung beinhaltet. Jede Komponente ist einzeln ausführbar, dies hat den Vorteil dass für die Bereitstellung eines Mircoservices keine weiteren Komponenten benötigt werden und so keine beziehungsweise nur geringe Abhängigkeiten bestehen. Dementsprechend verfügen Microservices, sofern sie es benötigen, über ihre eigene Datenbank, auf die nur sie zugreifen können.
Die Funktionen und Daten eines Mircoservices werden über das Interface des Mircoservices

angeboten. Die Kommunikation zwischen den Microservices verläuft über das Netzwerk. Üblicherweise wird dafür ein simples Kommunikationsprotokoll wie HTTP verwendet. Der Aufruf eines oder mehrerer Microservices von einem Mircoservice bildet den gesamten Business Service. Abbildung X zeigt ein Beispiel.
Instanzen eines Services werden in einer Registry verwaltet und sobald ein Service eine Funktion eines anderen Services benötigt fragt es bei der Registry nach diesem Service und erhält die benötigten Daten. Der genaue Ablauf der verschiedenen Akteure wird im folgenden Abschnitt genauer betrachtet.

3.3 Design Entscheidung

In diesem Abschnitt wird zunächst das Koordinationsmodell mit den drei Akteuren *Client*, *Service* und *Registry/Load Balancer* vorgestellt. Daraufhin wird auf das Ressourcenmanagement von Instanzen eingegangen, dabei wird auf verschiedene Szenarien eingegangen, wie die Anzahl der Instanzen eines Microservices reguliert wird. Zum Schluss werden die Arbeitseinteilung und Zuweisung der architektonischen Elemente (Komponenten) erläutert. Dabei werden zum einen Möglichkeiten, die die Microservice Architektur anbietet präsentiert und daraufhin welche dieser Möglichkeiten für DevOps geeignet sind.

3.3.1 Koordinationsmodell

Beim Koordinatiosmodell gibt es drei Akteure. Zum einen die *Registry/Load Balancer*, wobei die Registry für die Verwaltung der Serviceinstanzen zuständig ist und der Load Balancer für die Verteilung der Last auf die verschiedenen Instanzen eines Services. So kann beispielsweise der Load Balancer bei erhöhter Anzahl von Anfragen auf einen Service weitere Instanzen anfordern, sodass Instanzen nicht überlastet werden.
Der zweite Akteur ist die *Instanz eines Services*, das Daten beziehungsweise Dienstleistungen anbietet. Die Instanz registriert sich bei der Registry und hinterlegt seinen Namen, Adresse und Interface. Darauf steht nun die Instanz für den dritten Akteur bereit: Der *Client* kann dabei ein Benutzer oder ein anderer Service sein, der für einige Aufgaben weitere Service benötigt. Dafür fragt dieser bei der Registry nach den gewünschten Service und erhält die hinterlegten Daten der Serviceinstanz. Mit diesen Daten kann der Client nun die Instanz über das Interface ansprechen und verwenden. Die folgende Abbildung X gibt eine Übersicht der Beziehungen der drei Akteure.

Da es jederzeit Möglich ist, dass Instanzen ausfallen können oder nicht mehr aktiv sind, ist es wichtig das die Registry über diese Instanzen informiert ist. Denn sonst vermittelt die Registry Instanzen an Clienten, die der Client nicht ansprechen kann. Für dieses Problem gibt es den Healthcheck. Dabei werden Instanzen nur für eine bestimmte Zeit in der Registry verwaltet. Damit Instanzen nicht aus der Registry entfernt werden, müssen sich die Instanzen periodisch bei der Registry melden. Alternative kann die Registry proaktive Anfragen an Instanzen stellen, die sich in der Registry befinden. Sollte die Instanz nach bestimmter Zeit keine Antwort liefern wird diese aus der Registry entfernt.

3.3.2 Ressourcenmanagement

Ein wichtiger Anforderungspunkt von DevOps ist die Skalierbarkeit. Weitere Serviceinstanzen lassen sich bei Ausfällen von Instanzen oder bei höhere Belastung initialisieren. Auch ist es möglich Instanzen bei geringerer Belastung wieder entfernen um so Ressourcen zu sparen. Bei der Entscheidung welcher Akteur über die Anzahl der Instanzen bestimmt gibt es drei Szenarien: Zum einen kann der Service selber entschieden ob er von sich weitere Instanzen bereitstellt, dies geschieht wenn der Service erkennt, dass Instanzen überbelastet sind. Zum Anderen kann der Client weitere Instanzen anfordern, sofern dieser bereits im Vorfeld weiss, dass er in naher Zukunft eine erhöhte Anzahl von Anfragen an einen Service hat. Im dritten Szenario kann eine externe Verwaltung durch Monitoring über die Anzahl der Instanzen entscheiden. So kann Monitoring erkannt werden, dass zu bestimmten Uhrzeiten ein Service vermehrt verwendet wird. Beispielsweise am Abend oder Wochenende, wo viele Benutzer zuhause sind und den Service verwenden wollen.

3.3.3 Abbildung von architektonischen Elementen

Bei der Arbeitsaufteilung von Teams, können zum einen mehrere Teams einem Service arbeiten und ein Team an mehreren Services. Dies benötigt allerdings ein erhöhten Koordinationsaufwand. Idealerweise arbeiten an einem Service nur ein Team, wobei ein Team an mehreren Services entwickeln kann. Dies entspricht auch den Anforderungen von DevOps, das geringe Koordination zwischen den Teams fordert.

Neben der Entwicklung einer Komponente von nur einem Team, sollen die Komponenten auch unabhängige Einheiten darstellen. So können diese unabhängig ausgeliefert werden und Änderungen haben keine Auswirkung auf andere Komponenten. Dies verringert ebenfalls die Koordination und beschleunigt die Auslieferung einzelner Komponenten. Die Anforderungen des continuous Deployment des DevOps werden damit ebenfalls erfüllt.

3.4 Herausforderungen

Aufgrund der Eigenschaften, wie geringe Koordination zwischen Teams, hat die Architektur einige Herausforderungen. Dazu zählt zum einen die Systemstabilität und leichte Modifizierbarkeit der Module bei Erkennung von Problemen wie beispielsweise neuen Versionen von benutzter Dritt-Software. Im folgenden werden die Herausforderungen der Microservicearchitektur genauer betrachtet.

3.4.1 Systemstabilität

Clienten sprechen Service über ein Interface an. Diese Interface können sich im Laufe der Entwicklung verändern und aufgrund der geringen Koordination zwischen den Teams, können Missverständnisse bezüglich der Semantik der Interface zwischen Client- und Service Team entstehen. So können Service mit neuen Interface einen unerwarteten Input erhalten, oder Clienten von einem Service mit neuen Interface ein unerwarteten Output. Um diesen Problemen entgegenzuwirken erfordert es ein defensives Programmieren, das heißt es benötigt eine große

Anzahl unterschiedlicher Exceptions um den Fehler eingrenzen zu können. Zwar ist es möglich bei neuen Änderungen von Services Integrationstests durchzuführen, dies ist allerdings auf Grund der kurzen Releasephasen und der hohen Anzahl von Microservices sehr mühsam und Zeitaufwendig. Daher wird der *Consumer driven contract* verwendet. Für jeden Microservices werden Testcases festgelegt, die den Input und Output festlegen. Sobald die Interfacespezifikationen geändert werden, müssen die Testcases angepasst werden und von den Clienten, die den Service verwenden, akzeptiert werden.
Ein weitere Punkt für die Sicherstellung der Systemstabilität ist die Korrektheit der Umgebung. Systemumgebungen können fehlerhaft oder falsch konfiguriert sein. Damit der Service dennoch einwandfrei läuft muss bei der Initialisierung die Umgebung geprüft werden und gegebenenfalls muss sich der Service der Umgebung anpassen oder der Service passt die Umgebung an.
Ausfälle von Instanzen sind keine Seltenheit. Diese Ausfälle können mit Hilfe von Timeouts ermittelt werden und Clienten müssen alternative Mechanismen bei Timeout bereitstellen und ausführen.

3.4.2 Modifizierbarkeit

Modifizierbarkeit bedeutet, dass Services leicht und schnell auf Änderungen angepasst werden können. Zwar ist es schwer ein Service so robust zu programmieren, dass auf jedes mögliche Problem schnell reagiert werden kann. Dennoch sollte der Service schnell die wahrscheinlichsten Änderungen identifizieren. Dazu gehört die Umgebung eines Services, das auf verschiedenen Rechnern unterschiedlich konfiguriert sein kann. Auch kann sich ein Zustand eines anderen verwendeten Services verändern. Das Interface sowie die angebotene Dienstleistung kann auf einer anderen Art bearbeitet werden, die möglicherweise mehr Zeit in Anspruch nimmt.
Neue Versionen von verwendete Bibliotheken und Dritt-Software können ebenfalls veränderte Funktionalität haben und so zu Problemen führen.
Da üblicherweise auf Identifizierte Veränderungen nicht direkt reagiert werden kann, wird versucht kaskadierende Effekte so gering wie möglich zu halten, sodass andere Bereiche nicht ebenfalls von den Problemen betroffen werden. Dazu werden mit Hilfe von Modulen die Änderungen gekapselt. Module sind in jeden Service enthalten und lokalisieren sowie isolieren Änderungen. Außerdem bieten die Module ein stabiles Interface zu den Änderungen an, sodass der Service möglichst normal weiterarbeiten kann.

3.5 Antipattern

In diesem Abschnitt werden einige Antipattern vorgestellt und gezeigt wie die korrekte Vorgehensweise ist.//

Funktionsumfang eines Services wächst mit der Zeit und den Anforderungen:// Services sollten möglichst klein gehalten werden und neu eingeführte Funktionen sollten jeweils ein neuen Service bilden.// Zu wenig Automatisierung der Test und des Deployments:// Um kurze Releasephasen einzuhalten ist es wichtig in allen möglichen Bereichen zu automatisieren, da sonst zu viel Zeit verloren geht.// Horizontale Schichten werden als Services abgebildet (z.B. Datenschicht und Geschäftslogik):// Microservices sollten möglichst unabhängig und einzeln ausführbar sein. Sobald Datenschicht und Geschäftslogik getrennt wird, benötigt mindestens ein Service für den Zugriff auf die Datenbank immer ein anderen Service. Dies entspricht nicht

den gewünschten Eigenschaften von Microservices.// Jeder Service wird manuell konfiguriert:// Müssen beispielsweise eine Internetadresse in den Servicen geändert werden, würde es sehr Zeitaufwenig sein die Änderungen in jeden Service einzeln auszuführen. Daher sollten die Services mit einem Konfigurationsserver verbunden sein und Änderungen über diesen Server ausgeführt werden.// Zu jedem Service existiert nur eine Version:// Kurze Releasephasen bedeuten häufige Versionsänderungen. Existiert nur eine Version hat dies zur Folge, dass alle Instanzen der alten Version der alten Version zerstört werden müssen. Dabei könnten diese Instanzen allerdings gerade von Clienten in Benutzung sein. Daher sollten nur Instanzen durch Serviceinstanzen der neuen Version ersetzt werden, wenn sie nicht verwendet werden.//

3.6 Gefahren

Microservices beinhalten einige Gefahren. Die hohe Anzahl an Netzwerkverbindungen für die Kommunikation zwischen den Servicen, Clienten und der Registy kann zu mögliche Latenzen führen. Wodurch fälschlicherweise Timeouts ermittelt werden können. Auch verschlechtert sich die Effizienz des Business Services, da Clienten länger auf die Antwort der Services warten müssen.// Unerwartete Überlastungen von Serviceinstanzen verlangsamen die Performance. Ebenso führen Ausfälle von Instanzen zu geringerer Effizienz der gesamten Anwendung.// Bei einer hohen Anzahl von Serviceinstanzen kann es dazu kommen, dass die Registry nicht mit der Verwaltung der Instanzen hinter herkommt und bei der Vermittlung der Instanzen an die Clienten zu Verzögerungen kommt oder sogar zur falschen Vermittlung einer Instanz.

3.7 Beispiel

Ein Beispiel für den Einsatz von Mircoservices ist Netflix. Ihr ganzes Streamingsystem ist auf der Microservicearchitektur aufgebaut. Durch die leichte Skalierbarkeit und updaten von Microservices im laufenden Betrieb erreicht Netflix eine Verügbarkeitsquote von 99.7%. Im Schnitt ist Netflix nur zwei Stunden im Monat nicht verfügbar.

4 Zukunft von DevOps

4.1 Vorraussichtliche Marktakzeptanz

4.2 Organisatorische Probleme

4.3 Prozess Probleme

4.4 Technologische Probleme

4.5 Bugfixing

5 Zusammenfassung und Fazit