Serverlose und verteilte Softwarearchitektur in einer Microservice-Umgebung (V2)

Philipp Haider

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Juli 2017

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 7. Juli 2017

Philipp Haider

Inhaltsverzeichnis

Kapitel 1

Microservices

Der Begriff Microservices stiftet derzeit noch große Verwirrung. Nicht nur weil er relativ neu ist, sondern auch weil er sehr breit gefächert ist und eine klare Abgrenzung kaum möglich ist. Im nachfolgenden Kapitel wird der Begriff Microservice ausführlich definiert, beschrieben und zu anderen Konzepten abgegrenzt.

1.1 Was ist ein Microservice

Der Begriff *Microservice* ist aus den Wörtern „Micro“ – also klein – und „Service“ zusammengesetzt. Ein Service, oder gleichbedeutend auch Dienst, Bezeichnet in der Softwareentwicklung einen Mechanismus zum Bereitstellen von Ressourcen und Funktionen [Mac+06, S. 12]. Der sogenannte Service-Anbieter stellt den Service beliebigen, teilweise unbekannten, Service-Konsumenten zur Verfügung. Die Schnittstelle des Service ist in der Service-Beschreibung eindeutig festgelegt und den Service-Konsumenten bekannt.

Hinter Microservices verbirgt sich keine konkrete Technologie oder ein Konzept, das aktiv entwickelt wurde. Vielmehr ist es ein Sammelbegriff, der nachträglich für über die Jahre entstandener Praktiken, Methoden und Technologien im Umfeld von komplexen Softwaresystemen eingeführt wurde. Am häufigsten ist mit Microservices die sogenannte Microservice-Architektur gemeint. Charakteristisch für dieses Softwarearchitekturmuster ist die Zerlegung eines Softwaresystems in kleine, autonome Dienste, die über ein leichtgewichtiges Kommunikationsprotokoll miteinander kommunizieren. Die Fähigkeiten eines einzelnen Dienstes ist genau auf die Geschäftsanforderungen eines bestimmten Unternehmens oder Einsatzgebietes zugeschnitten [Fowc]. Die Verwendung dieses Muster bringt aber neben technischen Einflüssen meistens auch organisatorische Einflüsse mit sich. Die Microservice-Architektur hat Auswirkungen auf die Teamstruktur, Verantwortung, Continuous Integration, Testen und viele andere Bereiche. Daher können mit dem Begriff Microservices viele verschiedene Aspekte gemeint sein.

Um den rasanten Verbreitung der Microservice Architektur zu verstehen, ist es notwendig die klassische Architektur derartiger Systeme zu kennen. Dazu beschreibt der nächste Abschnitt den sogenannten monolithischen Ansatz, die damit verbundenen Probleme und Herausforderungen.

1.1.1 Monolithischer Ansatz

Mittlerweile haftet der monolithischen Softwarearchitektur ein negativer Ruf an. Jedoch zu Unrecht, da dieser Ansatz seit Jahrzehnten in vielen Bereichen der Softwareentwicklung sehr gut funktioniert. Im Java- und .NET-Umfeld war bzw. ist dieser Ansatz noch immer gängige Praxis. Dennoch wird er vielerorts von der Microservice-Architektur abgelöst. Doch was macht eine Anwendung überhaupt zu einer monolithischen Anwendung?

Unter einer monolithischen Anwendung versteht man in der Softwareentwicklung eine Anwendung, in der verschiedene Geschäftsbereiche oder Funktionalitäten als eine einzige Anwendung zusammengeschlossen sind [Fowc]. Intern kann die Applikation beispielsweise als Mehrschicht-Architektur organisiert sein. Üblich ist eine Drei-Schicht-Architektur mit folgenden Bestandteilen [Fow02, S. 19]:

- **Präsentationsschicht:** Hier ist jener Quelltext angesiedelt, der sich mit der Darstellung der Benutzerschnittstelle, z. B. in Form einer Web- oder Desktop-Anwendung auseinandersetzt.
- **Geschäftslogikschicht:** Diese Schicht stellt den Kern der Anwendung dar. Sie enthält alle geschäftsrelevanten Funktionen.
- **Datenzugriffsschicht**: In dieser Schicht befindet sich die Funktionalität, die zum Zugriff auf externe Datenquellen, wie z. B. einer relationalen Datenbank, notwendig ist.

In Abbildung 1.1 ist der Aufbau einer monolithischen Anwendung mit Drei-Schicht-Architektur skizziert. Der Zugriff darf jeweils nur auf die direkt darunterliegende Schicht erfolgen. Innerhalb jeder Schicht ist eine Modularisierung in verschiedene Bereiche, die auf die Geschäftsfunktionen abgestimmt sind, sinnvoll.

Nachteile und Schwierigkeiten

Mit steigender Komplexität der Software kommen nach und nach Probleme zum Vorschein, die diesen Ansatz aufwändig oder sogar unpraktikabel machen. Eine der größten Herausforderungen bei der Entwicklung von Anwendungen mit globaler Reichweite ist die Skalierbarkeit. Die einzige Möglichkeit, eine monolithische Anwendung zu skalieren, ist auf mehreren Servern jeweils eine Instanz der Anwendung zu installieren und über einen Load-Balancer zu verbinden. Abbildung 1.2 zeigt, wie ein solcher Aufbau aussehen kann. In vielen Einsatzgebieten ist dieser Ansatz aber nicht fein-

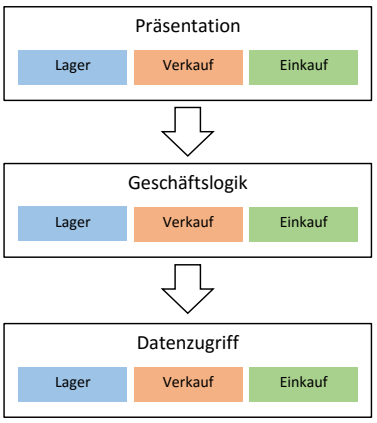


Abbildung 1.1: Drei-Schicht-Architektur

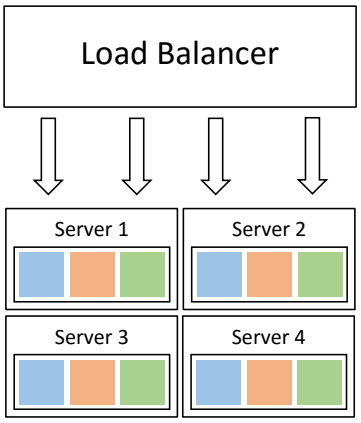


Abbildung 1.2: Skalieren einer monolithischen Anwendung

granular genug. Oft ist es sinnvoller, nur bestimmte Teile des Gesamtsystems zu skalieren.

Ein Monolith stößt aber nicht nur aus technischer Sicht auf Skalierbarkeitsprobleme, sondern auch aus organisatorischer Sicht. Alle Entwickler arbeiten zwangsläufig an der gleichen Codebasis, die dadurch enorme Größe annehmen kann. Kein einzelner Entwickler ist mehr in der Lage, die ganze Codebasis zu überblicken und zu beurteilen, welche Auswirkung eine Änderung haben kann. Das führt über kurz oder lang zur Verlangsamung der Entwicklungsgeschwindigkeit oder sogar zu völligem Stillstand.

Anforderungen an eine Software ändern sich häufig sehr rasch. Agile Softwareprozesse unterstützen dabei, die geänderten Anforderungen zeitnahe in die Software einfließen zu lassen. Für den Kunden bzw. Endbenutzer zählt nur die schnellstmögliche Umsetzung seiner neuen Anforderungen. In einer monolithischen Anwendung sind mehrere Geschäftsbereiche zusammengefasst. Somit muss die Anwendung immer als ganzes getestet und ausgerollt werden. Dadurch verlangsamt sich der gesamte Releasezyklus auf den langsamsten Teilbereich, obwohl vielleicht der geänderte Teilbereich schon längst fertiggestellt ist.

Hinter einer monolithischen Anwendung steht immer eine bestimmte Technologie oder eine Sammlung von mehreren Technologien, wie z. B. Java EE oder ASP.NET. Somit müssen alle Teilbereiche der Anwendung auf die selben Technologien zurückgreifen. Für manche Teilbereiche könnte aber möglicherweise die Verwendung von alternativen Technologien und Speichersystemen große Vorteile bringen.

In diesem Abschnitt wurden einige der Hauptkritikpunkte am monolithischen Architekturmuster angeführt. Es mag den Anschein vermittelt haben, dass dieser Ansatz nicht mehr verwendet werden soll oder veraltet

ist. Aber das ist so nicht der Fall. Viele Experten sind der Meinung, dass gerade am Anfang eines Projekts der monolithische Ansatz zu bevorzugen ist [Fowe]. Erst nachdem das Verständnis für die Domäne vorhanden ist und Anforderungen klarer sind, ist der fließende Übergang zu einer Microservice Architektur einfacher und sinnvoll.

1.2 Charakteristiken von Microservices

Dieser Abschnitt beschreibt einige charakteristische Merkmale eines Microservice. Wie bereits erwähnt, gibt es keine eindeutige Definition eines Microservice. Daher ist es sinnvoller, allgemeine anerkannte Charakteristiken der Microservice-Architektur zu betrachten.

Fowler identifizierte in [Fowc] neun Charakteristiken von Microservices. Obwohl diese Liste bereits sehr umfangreich ist, sind immer wieder sinnvolle Ergänzungen, wie in [Hor16, S. 3-9] zu finden. Die nachfolgenden Abschnitte beschreiben eine repräsentative Auswahl dieser Charakteristiken.

1.2.1 Organisation um Geschäftskompetenzen

Normalerweise sind Softwareentwicklungsteams anhand ihrer technologischen Spezialisierung z. B. in folgende Teams eingeteilt:

- Front-End-Entwickler
- Backend-End-Entwickler
- Datenbank-Administratoren
- IT-Administratoren

Durch die Microservice-Kultur hat sich aber eine andere Art der Teamstrukturierung etabliert. Für jede Geschäftskompetenz ist genau ein Team, welches intern aus einer kleinen Anzahl an den eingangs genannten Technologieexperten besteht, verantwortlich. Damit soll die Identifizierung und das Verantwortungsbewusstsein jedes einzelnen Teammitglieds für den von ihm entwickelten und betriebenen Dienst gestärkt werden.

Was aber dennoch eine große Herausforderung bleibt, ist die Definition des Aufgabenbereichs eines einzelnen Teams. Die Identifikation und Abgrenzung der Geschäftskompetenzen eines Unternehmens ist eine sehr komplexe Aufgabe. Als Hilfsmittel lässt sich hier das Single-Responsibility-Principle heranziehen [Mar06, S. 116]:

> *A class should have only one reason to change.*

Das aus der objektorientierten Programmierung bekannte Entwurfsprinzip kann auch im Kontext von Microservices angewandt werden. Es erweitert diesen Grundsatz von der Klassen-Ebene auf die Dienst-Ebene. Das heißt, jeder Dienst soll genau eine Fähigkeit realisieren. Damit ist sichergestellt,

dass geänderte Geschäftsanforderungen so wenig wie möglich andere Microservices beeinflussen.

1.2.2 Modularisierung durch Dienste

Seit Jahrzehnten strebt die Softwareentwicklungsbranche danach, komplexe Softwaresysteme einfach durch die Integration verschiedener Komponenten realisieren zu können. Durch die vielen über Jahre hinweg entwickelten Ansätze, ist leider auch der Begriff Komponente sehr überstrapaziert und hat dementsprechend viele unterschiedliche Bedeutungen. Beispielsweise kann in der objektorientierten Programmierung eine Klasse als Komponente betrachtet werden. Im Java-Umfeld werden *Enterprise Java Beans* gerne als verteilte Komponenten bezeichnet. In anderen Bereichen wiederum wird der Begriff für ein Software-Modul verwendet.

Auch wenn die konkrete Umsetzung einer Komponente sehr vielfältig sein kann, haben alle Ansätze dieselben Ziele:

- **Austauschbarkeit:** Es soll möglich sein, eine Komponente durch eine andere zu ersetzen.
- **Wiederverwendbarkeit:** Eine entwickelte Komponente soll auch in anderen Systemen wiederverwendet werden können.
- **Aktualisierbarkeit:** Durch die Austauschbarkeit ist es automatisch möglich, eine Komponente durch eine neuere Version zu ersetzen, ohne den Rest der Anwendung zu beeinflussen.

Im Kontext der monolithischen Softwarearchitektur sind am häufigsten die Begriffe Modul oder Software-Bibliothek ein Synonym für den Begriff Komponente. Eine Anwendung besteht somit aus einer Menge von Modulen, die miteinander verbunden sind. Leider erfüllt dieser Ansatz die zuvor beschriebenen gewünschten Eigenschaften der Komponenten-Orientierung nicht ganz zufriedenstellend. Abhängigkeiten zwischen den einzelnen Modulen oder bestimmte Voraussetzungen an das Laufzeitsystem der monolithischen Anwendung können das Aktualisieren eines Moduls wesentlich erschweren. Somit sind die verwendeten Komponenten nicht völlig voneinander unabhängig, sondern haben versteckte Abhängigkeiten.

In einer Microservice-Architektur wird die Modularisierung eines Softwaresystems, durch die Zerlegung in mehrere Services erreicht. Im Unterschied zu einer Komponente, stehen bei einem Service die Bedürfnisse des Service-Konsumenten im Vordergrund. Jeder Service ist ein eigener Prozess, der über ein Kommunikationsprotokoll mit anderen Services und seinen Konsumenten interagiert. Mit diesem Ansatz lassen sich die gewünschten Eigenschaften viel leichter realisieren.

1.2.3 Dezentrale Datenverwaltung

Eine monolithische Anwendung legt sehr häufig alle persistenten Daten in einer einzigen relationalen Datenbank ab. Jeder Programmteil greift einfach auf die benötigten Daten direkt zu. Dieser Ansatz ist in Abbildung 1.3 dargestellt. In einer Microservice-Architektur ist dieser Ansatz nicht mehr empfehlenswert. Persistente Daten eines bestimmten Geschäftsbereichs sollen nur von einem einzigen Dienst verwaltet werden. Direkter Zugriff auf diese Daten durch andere Dienste führt zu unerwünschten Abhängigkeiten. Andere Dienste müssen Daten über die öffentliche Schnittstelle des zuständigen Microservices abfragen oder verändern.

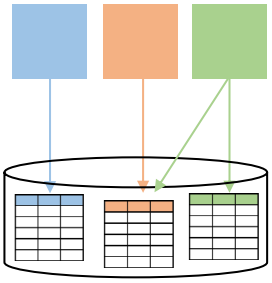


Abbildung 1.3: Zentraler Datenspeicher

Wenn jeder Dienst wie in Abbildung 1.4 seinen eigenen Datenspeicher besitzt, kann für die Erfüllung seiner Aufgabe der optimale Speichermechanismus ausgewählt werden. Beispielsweise können Daten mit komplexen Beziehungen in einer Graphdatenbank abgelegt werden, einfache Daten jedoch in einem schnellen Key-Value-Speicher.

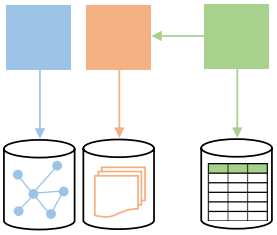


Abbildung 1.4: Polyglot Persistance

Auch eine monolithische Anwendung könnte Gebrauch von mehreren Datenbanken machen. Dieser Ansatz ist unter dem Namen *Polyglot Persistence* bekannt [Fowf]. Aus teilweise sehr vielfältigen Gründen ist er aber selten anzutreffen. Dazu zählen beispielsweise folgende:

- Die IT-Strategie einiger Unternehmen sehen nur die Verwendung ganz

bestimmter Datenbanken vor.

- Bereits getätigte Investitionen in bestehende Datenbanksysteme müssen amortisiert werden.
- Angst vor dem Betrieb einer Datenbank, für die noch nicht die notwendige Erfahrung aufgebaut wurde.

Für Systeme mit hohen Anforderungen an die Verfügbarkeit ist die Verwendung einer einzigen Datenbank nicht ausreichend. Denn eine einzige Datenbank stellt einen kritischen *Single Point of Failure* dar.

Eine zentrale Datenbank ist aber keinesfalls ein Antimuster, sondern in vielen Fällen die richtige Entscheidung. In verteilten Szenarien wird meistens die Konsistenz zugunsten der besseren Verfügbarkeit und Skalierbarkeit eingeschränkt. Aber diese Einschränkung ist nicht in jedem Szenario akzeptabel. Das *CAP-Theorem* von Brewer besagt, dass in einem verteilten System nur zwei der drei Eigenschaften Verfügbarkeit, Konsistenz und Ausfalltoleranz gleichzeitig erreicht werden können [Bre00]. Somit ist eine zentrale Datenbank, für Anwendungen in denen es besonders auf Konsistenz und Verfügbarkeit ankommt, immer noch eine gute Wahl.

1.2.4 Automatisierung

Für ein erfolgreiches System, das aus Microservices besteht, ist ein hoher Grad an Automatisierung erforderlich. Durch die große Anzahl an Komponenten im System, muss die Anzahl an manuell erforderlichen Schritten sehr gering sein. Ansonsten ist der Aufwand für das Aktualisieren, Ausrollen, Testen und Überwachen viel zu groß.

Um von den Vorteilen der Microservice-Architektur gegenüber einer monolithischen profitieren zu können, ist es unabdingbar, dass jeder Dienst einzeln ausgerollt werden kann. Denn nur damit sind die Entwicklungs- und Releasezyklen der einzelnen Dienste im Gesamtsystem voneinander unabhängig.

Wenn jeder Dienst einzeln ausgerollt werden kann, ist es auch möglich, unterschiedliche Versionen gleichzeitig zu betreiben. Somit kann eine neue Version getestet werden und erst wenn diese Testphase erfolgreich abgeschlossen ist, der gesamte Datenverkehr auf diesen Dienst umgeleitet werden. Bei Problemen kann schnell der ursprüngliche Zustand wiederhergestellt werden. Insgesamt mindert ein derartiges Vorgehen das Risiko eines großen Big-Bang-Deployments.

Damit dieser Ansatz funktioniert, müssen die Dienste einige Anforderungen erfüllen. Jeder Dienst steht mit einigen anderen in Verbindung. Wenn nun eine neue Version eines Dienstes ausgerollt wird, muss dessen Schnittstelle abwärtskompatibel sein, da seine Konsumenten noch die alte Schnittstelle verwenden. Fehlertoleranz ist eine weitere wichtige Eigenschaft, da während eine neue Version ausgerollt wird, der Dienst kurzzeitig nicht ver-

fügbar sein kann.

1.2.5 Größe

Wie der Name bereits suggeriert, soll ein Microservice klein sein. Das sollte schon durch die Anwendung des *Single-Responsibility* Prinzips aus Abschnitt 1.2.2 gegeben sein. Wie groß ein Dienst genau sein soll, lässt sich aber aus vielen Gründen schwer festlegen. Die Anzahl der Quelltextzeilen ist ein schlechtes Maß für die Größe, da sie je nach Programmiersprache stark variiert. Schon aussagekräftiger ist die benötigte Zeitdauer für die Entwicklung des Dienstes. Aber auch diese Kennzahl kann trügerisch sein. Beispielsweise kann man mit der Verwendung von externen Bibliotheken eine große Zeitersparnis erzielen. Jedoch inkludiert der Service dann auch die Komplexität dieser Bibliothek.

Eine genaue domänen-unabhängige Größenangabe ist praktisch nicht möglich. Es kann lediglich ein grober Rahmen abgesteckt werden. Ein Microservice sollte eine Entwicklungszeit von mehreren Wochen nicht übersteigen.

Neben quantitativen Kennzahlen sind oft subjektive Kriterien erforderlich. Eine davon ist, dass ein einzelner Entwickler die gesamte Funktionalität eines Dienstes noch im Kopf behalten können muss. Sobald es keinen einzelnen Menschen mehr gibt, der alle Aspekte des Dienstes auf einmal überschauen kann, ist er auf jeden Fall zu groß. Meistens aber haben erfahrene Architekten und Entwickler ein sehr gutes Gefühl, ab welcher Größe ein Dienst zu groß ist.

1.3 Vorteile

Ein neuer Softwareentwicklungsansatz setzt sich nur dann durch, wenn er auch einen erkennbaren Mehrwert bietet. Dieser Abschnitt zeigt einige der Vorteile, die durch den intelligenten Einsatz einer Microservice-Architektur entstehen können [Fowa; New15].

Es ist wichtig zu verstehen, dass die Vorteile, die durch Microservices entstehen, erst ab einer gewissen Größe bzw. Komplexität des Gesamtsystems zum Tragen kommen. In einfachen Softwaresystemen steht die Komplexität der Microservice-Architektur in keinem angemessenen Verhältnis zu deren Vorteilen [Fowb].

1.3.1 Heterogener Technologieeinsatz

Beim monolithischen Ansatz sind die Technologieentscheidungen äußerst eingeschränkt [Fowa]. Nachdem die Haupttechnologie festlegt ist, kann aus diesem Korsett nur noch geringfügig ausgebrochen werden. Aber nicht für alle Aufgaben muss der eingeschlagene Weg auch der optimale sein.

Hier kommt ein wesentlicher Vorteil von Microservices zum Tragen. Da jeder Dienst eine kleine autonome Komponente ist, sind für jeden Dienst unterschiedliche sinnvolle Technologieentscheidungen möglich. Selbst die Wahl des Speichermechanismus kann individuell erfolgen. Die eingesetzte Technologie kann beispielsweise von der Aufgabenstellung oder den Fähigkeiten der Teammitglieder abhängig gemacht werden. Damit werden die Technologieentscheidungen nicht mehr von zentraler Stelle gesteuert, sondern die Kompetenz in die einzelnen Teams verlagert.

Die Macht der freien Technologieauswahl ist aber mit Bedacht einzusetzen. Es kann für ein System auch kontraproduktiv sein, wenn zu viele Technologien eingesetzt werden, die womöglich noch nicht einmal ein stabiles Stadion erreicht haben. Daher macht die Definition von Team-übergreifenden Richtlinien zur Technologieauswahl durchaus Sinn. Der entscheidende Vorteil gegenüber dem monolithischen Ansatz ist aber die Wahlfreiheit, auch nach dem Start der Entwicklung.

1.3.2 Robustheit

Durch den hohen Modularisierungsgrad in einer Microservice-Architektur herrscht ein ganz anderes Verständnis bezüglich der Verfügbarkeit von einzelnen Komponenten, wie in einer monolithischen Architektur. In einer monolithischen Anwendung befinden sind nämlich alle Komponenten in einem einzelnen Prozess. Daher kann ein Entwickler davon ausgehen, dass alle Komponenten immer verfügbar sind. Nicht so in einer Microservice-Architektur. Hier muss jederzeit davon ausgegangen werden, dass Dienste nur eingeschränkt oder gar nicht verfügbar sind.

Wenn eine monolithische Anwendung ausfällt, ist das ganze System nicht mehr verwendbar. Im Gegensatz dazu kann das Gesamtsystem trotz des Ausfalls eines oder mehrere Microservices, zumindest eingeschränkt, weiterlaufen.

1.3.3 Einfaches Deployment

Jedes Release einer großen monolithischen Anwendung stellt ein gewisses Risiko dar. Daher passieren solche Releases in eher großen Abständen. In einem agilen Umfeld sind derartige lange Releasezyklen nicht tragbar. Außerdem kann jede kleine Fehlfunktion das ganze Deployment zum Scheitern bringen.

Mit Microservices ist es möglich, jeden Dienst einzeln auszurollen. Damit ist die Auswirkung aus Sicht des Gesamtsystems viel geringer. Wenn der Dienst nach dem Ausrollen einen Fehler aufweist, kann nur dieser eine Dienst auf den Ausgangszustand zurückgesetzt werden.

Auch die Abhängigkeiten zwischen den einzelnen Teams ist durch den Microservice-Ansatz reduziert. Ein Team kann ihren Service um Funktiona-

lität erweitern und sofort ausrollen. Andere Teams, die diese Funktionalität verwenden wollen, können erst dann neu ausrollen wenn ihr Dienst stabil ist. Dafür ist es aber notwendig, dass alle Dienste ihre Schnittstellen stets rückwärtskompatibel halten.

Die in diesem Abschnitt beschriebenen Vorteile einer Microservices Architektur sind keinesfalls allumfassend. Je nach Einsatzgebiet kommen möglicherweise auch andere Vorteile stärker zum Tragen als die hier angeführten. Um von den Vorteilen profitieren zu können, ist eine ordentliche Umsetzung dieses Architekturmusters aber unumgänglich. Denn ansonsten hat man zwar die gesamte Komplexität die Microservices mit sich bringen, jedoch nicht die gewünschten Vorteile.

1.4 SOA

Ein kontrovers diskutiertes Thema im Bezug auf Microservices ist die Beziehung zu Service-orientierter Architektur, kurz SOA. Das abstrakte Architekturmuster SOA wurde bereits 1996 in [SN96] beschrieben und hat sich seither ständig weiterentwickelt.

Auch bei SOA ist es schwer, eine genaue Definition zu geben. Grundsätzlich handelt es sich dabei um eine lose gekoppelte Softwarearchitektur, in der die Komponenten der Software in Form von autonomen Diensten, um die Geschäftsprozesse gestaltet werden [BdH]. Neben dieser einfachen informellen Definition hafteten sich über die Jahre immer neue Konzepte an den Begriff an. Mittlerweile verbinden viele Architekten konkrete Technologien mit dem eigentlich abstrakten Konzept. Sehr häufig zählen dazu Technologien, wie SOAP, WSDL, WS-* Protokolle oder auch Nachrichtenorientierte Middleware Lösungen, wie ESB [Fowd]. Aus diesem Grund stößt dieses Muster oft auf Ablehnung, da fälschlicherweise viele komplexe Technologien damit in Verbindung gebracht werden.

Newman beschreibt in [New15, S. 8], dass Microservices und SOA so wie Scrum und agile Softwareentwicklung miteinander in Beziehung stehen. Scrum ist eine konkrete Form von agiler Softwareentwicklung. So sind auch Microservices ein bestimmter Stil SOA umzusetzen. Da alle Konzepte einer Microservice Architektur auch in SOA enthalten sind, können Microservices als Teilmenge und Konkretisierung von SOA verstanden werden.

Oft wird SOA dafür kritisiert, viel zu abstrakt und breit gefächert zu sein. Es gibt viel zu wenig konkrete Hilfestellungen, beispielsweise für den Entwurf von Service-Grenzen oder die Bestimmung der Service-Größe. Das Konzept der Microservices hingegen wurde für einen bereits existierenden Stil der Anwendungsentwicklung nachträglich geprägt. Bei SOA hingegen wurde zuerst das Konzept definiert.

1.5 Zusammenfassung

In diesem Abschnitt wurden Microservices als um Geschäftskompetenzen entworfene, lose gekoppelte und autonome Dienste in einer service-orientierten Architektur, definiert. Der Begriff Microservices dient als Sammelbegriff für viele Konzepte, die sich um dieses Architekturmuster scharen. Im Gegensatz zu SOA ist die Vorstellung, was Microservices sind, viel konkreter, obwohl auch Microservices immer noch viel Interpretationsspielraum offenlassen.

Monolithischen Anwendung bündeln sämtliche Funktionalität eines Systems in einem einzigen großen ausführbaren Prozess. Bei der Microservice-Architektur stehen wichtige Funktionalitäten als eigenständiger Dienst zur Verfügung. Das ermöglicht großen Teams die Arbeit an komplexen Aufgaben, ohne sich gegenseitig zu stören. Damit ist es jedem Team selbst überlassen, die optimalen Technologieentscheidungen zu treffen, den Releasezyklus des Dienstes zu bestimmen und den Dienst in robuster Art und Weise zu implementieren.

Den Vorteilen der Microservice-Architektur steht aber eine nicht zu unterschätzende Komplexität gegenüber. Beispielsweise die Orchestrierung aller einzelnen Dienste zu einer Einheit stellt eine große Herausforderung dar. Außerdem fließt viel Arbeit in die saubere Definition der Schnittstellen zwischen den Diensten ein. Einfache Anpassungen können schnell ausarten, da sie mehrere Dienste betreffen können.

Nichtsdestotrotz hat das Microservice-Architekturmuster in vielen Einsatzgebieten seine Vorzüge. Am Beginn einer Microservice-Architektur sollte eine ausführliche Aneignung des erforderlichen Domänenwissens stehen. Oft funktioniert das nur durch den Start der Entwicklung mit einer monolithischen Architektur. Das Ziel sollte eine evolutionäre Architektur sein. Eine monolithische Architektur kann mit steigenden Verständnis für die Servicegrenzen leicht in eine Microservice-Architektur überführt werden.

Kapitel 2

Serverlose Softwarearchitektur

Jede zentral zur Verfügung gestellte Software Applikation bedeutet gleichzeitig auch nicht zu unterschätzenden Wartungsaufwand der Infrastruktur auf der sie läuft. Obwohl viele Technologien der vergangenen Jahre, wie etwa Hardware-Virtualisierung, Cloud-Computing und Software-Container die Arbeit bereits wesentlich erleichtern, ist für den Betrieb noch immer ein IT-Administrator oder sogar ein Entwickler notwendig.

Für den Betrieb einer monolithischen Software hält sich der Aufwand für die Verwaltung der Infrastruktur in Grenzen, da die Software lediglich aus einer großen Applikation besteht und dementsprechend wenig Hardware-Ressourcen benötigt. Die Verwendung des Mircoservice-Architekturmuster verändertet diese Situation völlig. Denn in diesem Szenario besteht ein einzelnes System aus einer Vielzahl von kleinen Diensten, die möglichst isoliert voneinander ausgeführt werden. Dazu ist eine sehr flexible und agile Bereitstellung der erforderlichen Ressourcen notwendig.

Cloud-Computing Anbieter bieten mit *Infrastructure-as-a-Service* eine Möglichkeit, Server in wenigen Minuten bereitzustellen. Der Verwaltungsaufwand bleibt trotzdem relativ hoch, weil Betriebssystem-Updates, Sicherheit, Netzwerk, usw. in der Verantwortung des Verwenders liegen.

Viele Standard-Applikationen benötigen kaum Kontrolle über die Umgebung in der sie ausgeführt werden. Für diesen Fall ist die Verwendung eines *Platform-as-a-Service* Dienstes meistens sinnvoller als die von *Infrastructure-as-a-Service*. Hier übernimmt also der *PaaS*-Betreiber die vollständige Verwaltung auf Hardware und Betriebsystemebene, sodass der Verwender lediglich seine Anwendung im richtigen Format bereitstellen muss. Er hat nur noch aus einer abstrakten Sichtweise die Möglichkeit, die Kapazität und Skalierbarkeitseigenschaften seiner Anwendung zu beeinflussen. Die genaue Umsetzung übernimmt der Dienst-Anbieter als Dienstleistung.

Die Grundidee die sich hinter *Serverless Computing* verbirgt ist dem An-

wendungsentwickler Möglichkeiten bereitzustellen, mit denen er Programme ausführen kann, ohne sich über Hardware oder Server Gedanken machen zu müssen. Bei *IaaS* und *PaaS* ist das nicht gegeben.

Wie viele Konzepte im Microservice-Umfeld lässt sich auch Serverlose Softwarearchitektur nur schwer abgrenzen. Im nächsten Abschnitt werden die zwei zur Zeit häufigsten Ausprägungsformen näher definiert.

2.1 Arten Serverloser Softwarearchitektur

Serverlose Softwarearchitektur ist ein sehr junges Konzept, dessen weitere Zukunft noch offen ist. Derzeit haben sich aber schon zwei unterschiedliche Sichtweisen auf dieses Themengebiet herauskristallisiert [Rob].

In der älteren Sichtweise beschreibt der Begriff *Serverlos* Applikationen die sehr stark auf vollständig verwaltete Dienste von Cloud Anbietern zurückgreifen. Darunter fallen beispielsweise Datenbanken, Authentifizierungs- oder Benachrichtigungssdienste. Dieser Ansatz ersetzt also einen Großteil der Logik auf der Serverseite durch derartige Dienste. Daher hat sich auch die Bezeichnung *Backend-as-a-Service* dafür etabliert. Ein Teil der Logik muss aber dadurch vom Client übernommen werden. Aufgrund der großen Beliebtheit von JavaScript und den damit verbundenen Applikationsbibliotheken, sind die dafür benötigten *Rich-Applications* relativ komfortabel umzusehen.

Seit etwa 2014 hat sich die Sichtweise durch die Einführung des Dienstes *AWS Lambda* von *Amazon* etwas geändert. Dieser Dienst erlaubt es, einfache Ereignis-gesteuerte Funktionen zu schreiben, die in der Cloud in einer zustandslosen Ausführungsumgebung vollständig verwaltet laufen. Die Artefakte die der Entwickler erstellen muss sind einfache Funktionen wie sie in den meisten Programmiersprachen bekannt sind. Aus diesem Grund ist dieser Ansatz unter dem Namen *Functions-as-a-Service* bekannt. Der folgenden Abschnitt beschäftigt sich intensiv mit dieser neuen Sichtweise auf Serverlose Softwarearchitektur.

2.2 Functions-as-a-Service

Im Grunde erlaubt es *Functions-as-a-Service* kleine Aufgaben in Form von Funktionen zu programmieren und skalierbar ohne weiteren Aufwand in der Cloud zu betreiben. Der Entwickler kann sich voll auf die Kernfunktionalität seiner zu entwickelnden Software konzentrieren und muss sich nicht um Infrastrukturbelange kümmern.

Wie in den allermeisten Programmiersprachen auch, sind Funktionen in diesem Kontext eine relativ kleine Code-Einheit die nach einer Aktivierung mit bestimmten Eingaben, Ausgaben produziert. Die Aktivierung erfolgt bei Programmiersprachen durch einen Funktionsaufruf. In *FaaS* hingegen

durch das auftreten bestimmter Ereignisse die der Entwickler festlegen kann. Beispiele für derartige Ereignisse sind:

- Hinzufügen eines Eintrags in einer Datenbank oder Warteschlange
- HTTP-Anfrage
- Eine neue Nachricht in einem Messaging-System
- Zeitgesteuerte Ereignisse

Eine Funktion ist nur dann sinnvoll, wenn sie auch Ausgaben oder zumindest Seiteneffekte produziert. Im Falle von *FaaS* können diese wieder sehr vielfältig sein. Häufig aber ist das Ergebnis eine Interaktion mit einem anderen Cloud-Dienst:

- Hinzufügen oder manipulieren von Daten in einer Datenbank
- HTTP-Antwort
- Versenden von Benachrichtigungen oder E-Mails

In den folgenden Abschnitten wird näher darauf eingegangen, in welchen Anwendungsgebieten der Einsatz von *FaaS* sinnvoll ist und wie er sich von *PaaS* unterscheidet.

2.2.1 Anwendungsgebiete

Für *FaaS* gibt es in den verschiedensten Bereichen sinnvoll Anwendungsgebiete. Hauptsächlich werden sie aber für kleine und abgeschlossene Funktionalitäten herangezogen. Beispielsweise für die Konvertierung und Validierung von Daten. Eine Funktion kann dabei auf ein bestimmtes Ereignis, z. B. dem Hinzufügen eines Elements in einer Warteschlange warten, und danach die gewünschte Funktionalität ausführen. Das Ergebnis der Funktion kann z. B. automatisch in eine Datenbank gespeichert oder an ein anderes System gesendet werden.

Microservice Architektur oder allgemeiner Cloud-Architekturen sind aufgrund ihrer großen Anzahl an Komponenten, wie verschiedenen Datenspeichern, Warteschlangen und Diensten, meist sehr komplex. Damit alle Einzelkomponenten in Summe ein funktionstüchtiges Gesamtsystem bilden, ist viel Logik notwendig, die die Komponenten verbindet und administriert. Im Englischen wird diese Art von Logik gerne als *Glue-Clode* bezeichnet. Die Interaktion mit Cloud-Komponenten ist also ein essentieller Bestandteil von *FaaS*. Darum ist das Programmiermodell sehr stark für diese Szenarien ausgelegt.

Der Erfolg der Microservice Architektur und *FaaS* führte bereits zur Entstehung eines möglicherweise neuen Paradigmas: *Nanoservices* [Avr]. Bei Microservices stehen einzelne Geschäftsanforderungen im Vordergrund. Mit Nanoservice werden einzelne Geschäftsanforderungen noch weiter auf Funktionsebene heruntergebrochen. Ein Beispiel für einen Microservice könnte die Abwicklung von Bestellungen sein, mit dem es möglich ist Bestellungen

anzulegen, zu ändern oder zu verfolgen. Mit Nanoservices wäre jede einzelne dieser Funktionen ein eigener Dienst. Derzeit hat dieser Ansatz aber noch keine große Verbreitung.

Amazon beschreibt in ihrem Whitepaper zu AWS Lambda wie klassische Drei-Schicht-Architektur, z.B. Web- oder Mobile-Anwendungen, mit Serverlosen Technologie umgesetzt werden können [Bai+15]. Darüber hinaus eignet sich *FaaS* aber genau so gut für Microservice Architekturen. Die Einsatzgebiete sind daher sehr breit, was *FaaS* zu einem mächtigen Werkzeug macht.

Viel Potential besteht in neuen Domänen wie *Internet of Things*, *ChatBots* oder *DevOps* [Bra]. In diesen Bereichen ist die Nachfrage nach kleinen skalierbaren Programmen, die sich einfach entwickeln lassen, sehr hoch. Durch die Einfachheit von *FaaS* eignet es sich sehr gut für den Prototypenbau.

2.2.2 Beziehung zu Platform-as-a-Service

In vielen Bereichen überschneiden sich die Möglichkeiten von *FaaS* mit anderen Konzepten wie z. B. *PaaS*. Dieser Umstand ist nicht weiter verwunderlich, da *FaaS* auf der Basis von *PaaS* aufbaut. Der signifikanteste Unterschied ist aber die Ereignis-gesteuerte Funktionsweise von *FaaS*. Funktionen werden nach dem Auftreten eines bestimmten Ereignis nur für die Dauer einer Aktivierung ausgeführt. Daher bezahlt der Verwender auch nur die Anzahl der Aufrufe und die Dauer der Ausführungszeit. Bei *PaaS* ist jedoch meistens zumindest eine Virtuelle-Maschine dauerhaft erforderlich, die auf Ereignisse warten kann. Der Verwender trägt also auch die Kosten dieser Maschine, wenn sie untätig ist.

Skalierbarkeit ist in Cloud-Computing ein essentieller Faktor. *PaaS* bietet dafür die Möglichkeit, abhängig von Metriken wie Prozessorlast, die Anzahl der Instanzen auf denen die Anwendung ausgeführt wird, dynamisch zu erhöhen oder zu verringern. Dieser Ansatz erfordert bereits sehr wenig manuelles Zutun. Aber *FaaS* geht hier noch einen Schritt weiter und erfordert praktisch keine manuellen Handlungen um die Funktion skalierbar zu machen. Es ist die Aufgabe des Cloud Anbieters die Funktion automatisch skalierbar zu machen. Weil die Funktionen zustandslos sind, kann man sie einfach beliebig oft parallel ausführen.

Die Verwendung von *PaaS* schränkt Technologieentscheidungen sehr stark ein, weil man sich auf eine konkrete Platform bindet. Bei *FaaS* hingegen ist die Auswahl an möglichen Programmiersprachen sehr breit. Laufen fügen Cloud Anbieter neue Sprachen hinzu. Damit ist die Technologieabhängigkeit durch die Verwendung von *FaaS* sehr viel geringer.

2.2.3 Markt

Es gibt bereits einige Cloud Anbieter, die *FaaS* anbieten. Darunter befinden sind alle namhaften Anbieter wie Amazon, Microsoft, IBM und Google. Die nachfolgenden Abschnitte zeigen die Funktionsweise und Prinzipien anhand von Microsoft Azure Functions, da diese Implementierung Open-Source verfügbar ist und somit tiefe Einblicke in die Umsetzung bietet.

Amazon AWS Lambda ist von allen Diensten am längesten am Markt und somit auch vom gebotenen Funktionsumfang am größten. Jedoch haben auch die anderen Anbieter das Potential und die Nachfrage von *FaaS* erkannt und versuchen seither den Technologierückstand zu schließen.

Derzeit befindet sich dieses doch recht neue Thema noch sehr stark im Wandel. Es ist sehr wahrscheinlich, dass sich einige Dinge in naher Zukunft ändern werden. Die Grundideen haben aber alle Anbieter ähnlich umgesetzt. Trotzdem unterscheiden sie sich in einzelnen Punkten:

- Obwohl alle Anbieter die Zahl der unterstützten Programmiersprachen laufend nach oben schrauben, unterscheiden sich die einzelnen Anbieter hier doch sehr.
- Auch wie das konkrete Skalierbarkeitsverhalten aussieht, muss beim jeweiligen Anbieter getestet werden.
- Meistens sind nur andere Dienste innerhalb des selben Cloud Anbieters mit *FaaS* integriert. Dadurch kann sehr leicht eine starke Abhängigkeit zum gewählten Anbieter entstehen.
- Natürlich unterscheiden sie die einzelnen Angebote auch im Preis.

Schlusssatz

2.3 Azure Functions

Quellenverzeichnis

Literatur

[Bai+15] Andrew Baird, Stefano Buliani, Vyom Nagrani und Ajay Nair. „AWS Serverless Multi-Tier Architectures“. In: Nov. 2015. URL: https://d0.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf (siehe S. 15).

[Bre00] Eric A. Brewer. „Towards Robust Distributed Systems (Abstract)“. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, S. 7–. URL: http://doi.acm.org/10.1145/343477.343502 (siehe S. 7).

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (siehe S. 2).

[Hor16] Christian Horsdal. *Microservices in .NET Core*. Manning, 2016 (siehe S. 4).

[Mac+06] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown und Metz Rebekah. *Reference Model for Service Oriented Architecture 1.0*. 2006 (siehe S. 1).

[Mar06] Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentie Hall, 2006 (siehe S. 4).

[New15] Sam Newman. *Building Microservices*. O'Reilly, 2015, S. 4–8 (siehe S. 8, 10).

[SN96] Roy W Schulte und Yefim V Natis. „Service oriented architectures, part 1“. *Gartner, SSA Research Note SPA-401-068* (1996) (siehe S. 10).

Online-Quellen

[Avr] Abel Avram. *FaaS, PaaS, and the Benefits of the Serverless Architecture*. URL: https://www.infoq.com/news/2016/06/faas-serverless-architecture (besucht am 13. 10. 2016) (siehe S. 14).

[BdH] Don Box, John deVadoss und Kris Horrocks. *SOA in the Real World*. URL: https://msdn.microsoft.com/en-us/library/bb833022.aspx (besucht am 15. 09. 2016) (siehe S. 10).

[Bra] Mary Branscombe. *Microsoft Prepares for Serverless Computing with Azure Functions Preview*. URL: http://thenewstack.io/azure-functions-serverless-computing-handling-iot-devices/ (besucht am 27. 10. 2016) (siehe S. 15).

[Fowa] Martin Fowler. *Microservice Trade-Offs*. URL: http://martinfowler.com/articles/microservice-trade-offs.html (besucht am 17. 09. 2016) (siehe S. 8).

[Fowb] Martin Fowler. *MicroservicePremium*. URL: http://martinfowler.com/bliki/MicroservicePremium.html (besucht am 17. 09. 2016) (siehe S. 8).

[Fowc] Martin Fowler. *Microservices - a definition of this new architectural term*. URL: http://martinfowler.com/articles/microservices.html (besucht am 08. 09. 2016) (siehe S. 1, 2, 4).

[Fowd] Martin Fowler. *Microservices - GOTO 2014*. URL: https://www.youtube.com/watch?v=wgdBVIX9ifA (besucht am 15. 09. 2016) (siehe S. 10).

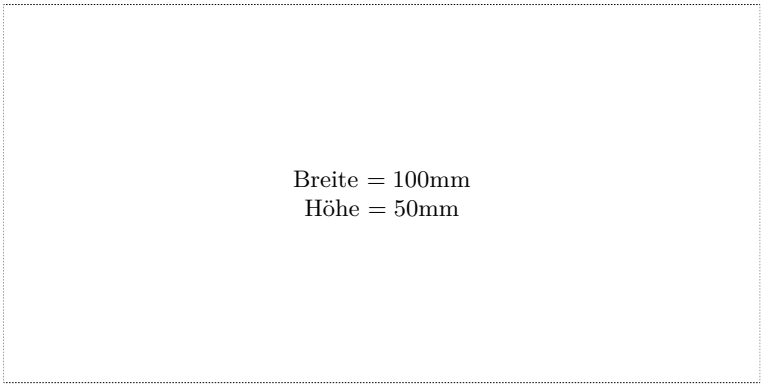
[Fowe] Martin Fowler. *Monolith First*. URL: http://martinfowler.com/bliki/MonolithFirst.html (besucht am 09. 09. 2016) (siehe S. 4).

[Fowf] Martin Fowler. *PolyglotPersistence*. URL: http://martinfowler.com/bliki/PolyglotPersistence.html (besucht am 09. 09. 2016) (siehe S. 6).

[Rob] Mike Roberts. *Serverless Architectures*. URL: http://martinfowler.com/articles/serverless.html (besucht am 05. 10. 2016) (siehe S. 13).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —