

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

Erklärung	i
1 Microservices	1
1.1 Was ist ein Microservice	1
1.1.1 Monolithischer Ansatz	2
1.2 Charakteristiken von Microservices	4
1.2.1 Organisation um Geschäftskompetenzen	4
1.2.2 Modularisierung durch Dienste	4
1.2.3 Dezentrale Datenverwaltung	5
1.2.4 Automatisierung	7
1.2.5 Größe	7
1.3 Vorteile	8
1.3.1 Heterogener Technologieeinsatz	8
1.3.2 Robustheit	9
1.3.3 Einfaches Deployment	9
1.4 SOA	10
1.5 Zusammenfassung	10
2 Serverlose Softwarearchitektur	12
2.1 Arten serverloser Softwarearchitektur	13
2.2 Functions-as-a-Service	13
2.2.1 Anwendungsgebiete	14
2.2.2 Beziehung zu Platform-as-a-Service	15
2.2.3 Markt	16
2.3 Azure Functions	16
2.3.1 Azure App Service	17
2.3.2 Site Control Manager	17
2.3.3 Web Jobs SDK	18
2.3.4 Bindungen	19
2.3.5 Web Jobs Script SDK	21
2.3.6 Azure Function App	24
2.3.7 Verrechnungsmodell	25
2.3.8 Kaltstart	26

2.3.9	Zusammenfassung	26
2.4	Evolution der Anwendungsentwicklung	27
2.4.1	Automatisierung der Softwareauslieferung	28
2.4.2	Leistungsverbesserung der Hardware	29
2.4.3	Organisatorische Veränderungen	29
2.4.4	Von Microservices zu serverlosen Anwendungen	30
Quellenverzeichnis		31
	Literatur	31
	Online-Quellen	32

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, die verschiedene Geschäftsbereiche oder Funktionalitäten vereint [Fowc]. Intern kann die Applikation z. B. 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.

Abbildung 1.1 zeigt den Aufbau einer monolithischen Anwendung mit Drei-Schicht-Architektur, in der jede Schicht nur auf die nächst tiefere zugreifen darf. Innerhalb jeder Schicht ist eine Modularisierung in verschiedene Bereiche, die auf die Geschäftsfunktionen abgestimmt sind, sinnvoll.

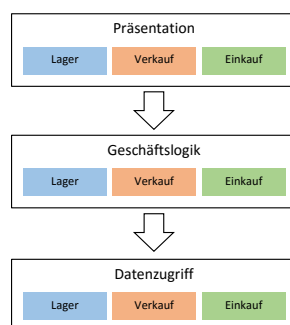


Abbildung 1.1: Drei-Schicht-Architektur

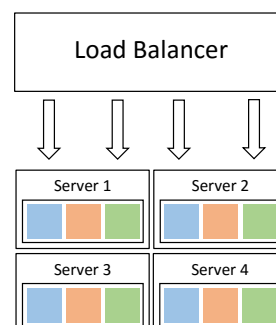


Abbildung 1.2: Skalieren einer monolithischen Anwendung

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 feingranular 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 oder mehrere bestimmte 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

Softwareentwickler sind normalerweise anhand ihrer technologischen Spezialisierung z. B. in folgende Teams eingeteilt:

- Front-End-Entwickler
- Backend-End-Entwickler
- Datenbank-Spezialisten
- Infrastruktur-Spezialisten

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.

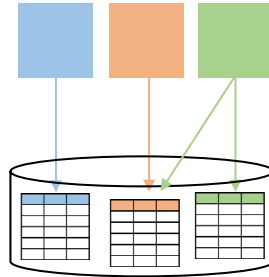


Abbildung 1.3: Zentraler Datenspeicher

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.

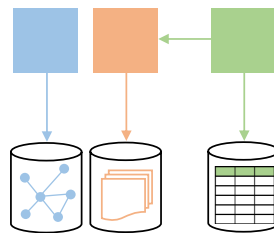


Abbildung 1.4: Polyglot Persistence

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 zu profitieren, ist es erforderlich, jeden Dienst einzeln ausrollen zu können. Denn nur so sind die Entwicklungs- und Releasezyklen der einzelnen Dienste einer Applikation 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 verfügbar 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 Zeiterparnis 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 festgelegt 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 Sta-

dion 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 sich 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 Funktionalität 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 serviceorientierter 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 haften 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 serviceorientierten Architektur, definiert. Der Begriff Microservice dient als Sammelbegriff für viele Konzepte, die sich um dieses Architekturmuster scharen. Im Gegensatz zu SOA ist die Vorstellung, was Microservices sind, etwas konkreter. Sie lassen aber immer noch viel Interpretationsspielraum offen.

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 überlas-

sen, 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 steigendem Verständnis für die Servicegrenzen leicht in eine Microservice-Architektur überführt werden.

Kapitel 2

Serverlose Softwarearchitektur

Die Bereitstellung und der Betrieb von Software-Applikationen sind zwei oft unterschätzte Kostenfaktoren. Viele technologische Entwicklungen der letzten Jahre, wie etwa Hardware-Virtualisierung, Automatisierungswerkzeuge für Infrastruktur und Cloud-Computing, haben diese Kosten bereits stark reduziert. Dennoch ist der Einsatz einer selbst verwalteten Infrastruktur aufwändig und erfordert intensive Zusammenarbeit zwischen Entwicklern, IT-Administratoren und Release-Managern.

Vor der Bereitstellung einer monolithischen Software steht zuerst die Dimensionierung und Beschaffung der benötigten Hardware-Infrastruktur durch die IT-Abteilung. Diese muss danach noch an die Bedürfnisse und Voraussetzungen der Software angepasst werden. Die Installation der eigentlichen Software ist oft ein manueller oder halb-automatisierter Prozess. Das ist langsam und fehleranfällig, aber meistens akzeptabel, denn eine monolithische Software hat lange Release-Zyklen und besteht aus einem einzigen Artefakt. Erst das Microservice-Architekturmuster hat eine isolierte, agile und effiziente Bereitstellung von Software vorangetrieben.

Mit *Infrastructure-as-a-Service (IaaS)* bieten viele Cloud-Computing-Dienstleister die Möglichkeit, Server in wenigen Minuten bereitzustellen. Der Verwaltungsaufwand bleibt trotzdem relativ hoch, weil Betriebssystem-Updates, IT-Sicherheit, Netzwerkkonfiguration, usw. in der Verantwortung des Verwenders liegen. Viele Applikationen benötigen kaum Kontrolle über die Umgebung in der sie ausgeführt werden. Für diesen Fall ist die Verwendung eines *Platform-as-a-Service (PaaS)* Dienstes meistens vorteilhafter. Hier übernimmt der *PaaS*-Betreiber die vollständige Verwaltung der Hardware- und Betriebssystemebene. Der Dienstleistungsnehmer muss dafür nur seine Anwendung im richtigen Format bereitstellen, sowie die Kapazität und Skalierbarkeitseigenschaften festlegen.

Die Grundidee hinter *Serverless Computing*, ist dem Softwareentwickler

eine Plattform für die Bereitstellung von Diensten zu bieten, ohne dass sich dieser um Server, deren Konfiguration oder Kapazitätsmanagement kümmern muss. Bei *IaaS* und *PaaS* ist das nicht oder nur zum Teil 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 betrachtet.

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 verwaltete Datenbanken, Authentifizierungs- oder Benachrichtigungsdienste. Dieser Ansatz ersetzt also einen Großteil der Server-Logik durch Dienste von Drittanbietern. Daher hat sich auch die Bezeichnung *Backend-as-a-Service* dafür etabliert. Ein Teil der Applikationslogik muss aber dadurch vom Client übernommen werden. Mit JavaScript und verschiedenen Bibliotheken für die Erstellung von Benutzeroberflächen, lassen sich die dafür benötigten *Rich-Applications* effizient realisieren.

Seit etwa 2014 hat sich die Sichtweise durch die Einführung des Dienstes *AWS Lambda* durch *Amazon* etwas geändert. Dieser Dienst erlaubt es, einfache ereignisgesteuerte Funktionen zu schreiben, die in der Cloud in einer zustandslosen Ausführungsumgebung vollständig verwaltet laufen. Diese Funktionen enthalten fast ausschließlich Geschäftslogik und werden in Skript-Dateien erstellt. Anstatt klobiger Artefakte können somit einfache Skript-Dateien verteilt werden. Weil Funktionen das zentrale Bereitstellungsformat sind, ist dieser Ansatz unter dem Namen *Functions-as-a-Service (FaaS)* bekannt. Der folgende Abschnitt beschäftigt sich intensiv mit dieser neuen Sichtweise auf serverlose Softwarearchitektur.

2.2 Functions-as-a-Service

Im Grunde erlaubt *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 Geschäftslogik seiner Applikation konzentrieren und muss sich kaum noch um Infrastrukturaufgaben kümmern.

Wie in den allermeisten Programmiersprachen, sind Funktionen in diesem Kontext eine relativ kleine Quelltexteinheit, die Eingangs- in Ausgangswerte transformiert. Die Aktivierung erfolgt bei Programmiersprachen durch

einen Funktionsaufruf. Hingegen bei *FaaS*, durch das Auftreten bestimmter vom Entwickler festgelegter Ereignisse. Beispiele für derartige Ereignisse sind folgende:

- Das Hinzufügen oder Manipulieren von Daten in einem Datenspeicher.
- Das Empfangen einer HTTP-Anfrage.
- Das Empfangen einer Nachricht von einem Nachrichtendienst.
- Das Auftreten eines zeitgesteuerten Ereignisses.

Eine Funktion ist nur dann sinnvoll, wenn sie auch Ausgaben oder zumindest Seiteneffekte produziert. In *FaaS* können diese wieder sehr vielfältig sein. Meistens ist das Ergebnis, die Interaktion mit einem anderen Cloud-Dienst wie z. B.:

- Das Hinzufügen oder Manipulieren von Daten in einem Datenspeicher.
- Das Senden einer HTTP-Antwort.
- Das Senden einer Nachricht an einen Nachrichtendienst.
- Das Versenden einer E-Mail.

Die folgenden Abschnitte beschreiben vielversprechende Anwendungsgebiete für *FaaS*. Des Weiteren werden die Konzepte *FaaS* und *PaaS* voneinander abgegrenzt.

2.2.1 Anwendungsgebiete

Für *FaaS* gibt es in den verschiedensten Bereichen sinnvolle Anwendungsgebiete. Hauptsächlich werden sie aber für kleine und abgeschlossene Funktionalitäten herangezogen. Beispielsweise eignet es sich sehr gut für die Konvertierung und Validierung von Daten. Eine Funktion kann auf ein bestimmtes Ereignis warten – z. B. dem Hinzufügen eines Elements in einer Warteschlange – 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- und Cloud-Anwendungen verwenden oft eine große Anzahl an Cloud-Komponenten, wie verschiedenen Datenspeicher, Warteschlangen und Nachrichtendienste. Damit alle Einzelkomponenten in Summe ein funktionstüchtiges Gesamtsystem bilden, ist viel Logik für die Verbindung und Administration der Komponenten notwendig. Im Englischen wird diese Art von Logik häufig als *Glue-Code* bezeichnet, weil er die Einzelteile zum einem Ganzen „zusammengeklebt“. Die Interaktion mit Cloud-Komponenten ist also ein essentielles Einsatzgebiet und ist deswegen ein großer Einflussfaktor auf das Programmiermodell von *FaaS*.

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

Nanoservices werden einzelne Geschäftsanforderungen noch weiter auf Funktionsebene heruntergebrochen. Ein Beispiel für einen Microservice könnte ein Dienst für die Erstellung, Änderung und Nachverfolgung von Bestellungen in einem Online-Shop sein. Mit Nanoservices wäre jede einzelne dieser Funktionen ein eigener Dienst.

Amazon beschreibt in [Bai+15] wie klassische Drei-Schicht-Architektur, z. B. Web- oder Mobile-Anwendungen, mit serverlosen Technologie umgesetzt werden können. 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*, *Chat-Bots* oder *DevOps* [Bra]. In diesen Bereichen ist die Nachfrage nach kleinen, skalierbaren Programmen, die sich einfach entwickeln und betreiben lassen, sehr hoch. Durch die Einfachheit von *FaaS* eignet es sich auch sehr gut für den Prototypenbau.

2.2.2 Beziehung zu Platform-as-a-Service

In vielen Bereichen überschneiden sich die Möglichkeiten von *FaaS* mit denen anderer Technologien, wie z. B. *PaaS*. Dieser Umstand ist nicht weiter verwunderlich, da *FaaS* auf der Basis von *PaaS* aufbaut. Der signifikanteste Unterschied ist die ereignisgesteuerte 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 meistens zumindest eine ständig laufende virtuelle Maschine erforderlich, die auf Ereignisse wartet. Das verursacht Kosten, auch wenn die Maschine kaum oder gar nicht genutzt wird.

Skalierbarkeit ist in CloudComputing 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 Eingreifen durch einen Entwickler oder Administrator. 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 zu skalieren. Weil die Funktionen zustandslos sind, kann man sie beliebig oft parallel ausführen.

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

2.2.3 Markt

Alle namhaften Cloud-Anbieter, wie Amazon, Microsoft, IBM und Google haben bereits *FaaS*-Produkte in ihrem Angebot. Die nachfolgenden Abschnitte zeigen die Funktionsweise und Prinzipien anhand von *Microsoft Azure Functions*, da diese Implementierung unter der MIT-Lizenz Open-Source verfügbar ist und somit tiefe Einblicke in die Umsetzung bietet.

Amazon AWS Lambda ist von allen Produkten am längsten am Markt und bietet den größten Funktionsumfang. Jedoch haben auch die anderen Anbieter das Potential und die Nachfrage von *FaaS* erkannt und versuchen seither, den Entwicklungsrückstand zu schließen.

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

- Jeder Anbieter bietet unterschiedliche Programmiersprachen an. Es werden aber laufend neue Sprachen in die Produkte integriert.
- 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 sich die einzelnen Angebote auch im Preis.

Derzeit investieren Cloud-Anbieter sehr viel in die Entwicklung ihrer *FaaS* Produkte. Das gibt einen Hinweis auf das große Potential dieser Technologie.

2.3 Azure Functions

Im März 2016 veröffentlichte die Firma *Microsoft* eine Vorschauversion ihrer eigenen serverlosen Plattform mit dem Namen *Azure Functions* [Mas]. Nur ein halbes Jahr später folgte die erste offizielle Version [Kir]. *Azure Functions* ist eine Erweiterung der ohnehin schon sehr umfangreichen *Microsoft* Cloud-Plattform. Diese Technologie eignet sich vor allem für die ereignisgesteuerte Verarbeitung und Transformation von Daten aus verschiedenen Datenquellen. Das deklarative Programmiermodell von *Azure Functions* ermöglicht eine einfache Interaktion mit Daten- und Ereignisquellen.

Microsoft griff für die Implementierung von *Azure Functions* auf folgende, schon länger in *Microsoft Azure* enthaltene, Dienste und Bibliotheken zurück [Chaa]:

- *App Service - Web App*
- *App Service Plan*

- *Site Control Manger (SCM)*
- *Web Jobs*
- *Web Jobs SDK*

Folgende Neuentwicklungen waren tatsächlich notwendig:

- *Web Jobs SDK Script*
- *App Service - Function App*
- *Dynamic Hosting Plan*

Die nachfolgenden Abschnitte geben einen Überblick über die hier aufgelisteten Bestandteile, die Entstehungsgeschichte und Anwendungsmöglichkeiten von *Azure Functions*. Großteils beziehen sich die Grundlagen von *App Services* und *Web Jobs* auf [Rai15].

2.3.1 Azure App Service

App Service ist in der *Microsoft Azure* Cloud ein Oberbegriff für Services, die in die Kategorie *Platform-as-a-Service* einzuordnen sind. Diese Services haben eine etwas konsequentere Interpretation von *PaaS* als die älteren *Cloud Services*. Bei einem *App Service* übernimmt *Microsoft* die Verwaltung der Infrastruktur, des Betriebssystems und der Laufzeitumgebungen.

Ein *App Service* ist einem sogenannten *App Service Plan* zugeordnet. Dieser kann mehrere *App Services* beinhalten, die dann auf einem oder mehreren Servern gemeinsam ausgeführt werden. Der *App Service Plan* ist also eine abstrakte Infrastrukturbeschreibung, in der die Anzahl, die Größe und das Skalierungsverhalten der virtuellen Maschinen festlegt. Daher wird er oft als *Server Farm* bezeichnet.

Der nächste Abschnitt beschreibt mit *Azure Web Apps* einen konkreten *App Service*, der ein Grundbaustein von *Azure Functions* ist.

Azure Web App

Eine *Web App* ist in *Microsoft Azure* der einfachste Weg, um eine Web-Anwendung bereitzustellen. Die virtuellen Maschinen des *App Service Plan*, in dem eine *Web App* ausgeführt wird, sind bereits mit den gängigsten Webentwicklungsumgebungen ausgestattet. Damit lassen sich Web-Anwendungen unterschiedlicher Technologien, wie z. B. Node.js, .NET oder PHP, gemeinsam betreiben. Grundsätzlich ist eine *Web App* ein abstrakter Web-Server.

2.3.2 Site Control Manager

Die Funktionalität eines *App Service* und somit auch einer *Web App* ist mit Hilfe von sogenannten *Site Extensions* erweiterbar. Diese Erweiterungen werden im selben Kontext wie die eigentliche Web Anwendung ausgeführt. Der *Site Control Manager* – oft auch *KUDU* genannt – ist eine Erweiterung, die bei jedem *App Service* automatisch vorinstalliert ist.

Anfänglich war die einzige Aufgabe des *Site Control Manager* die Auslieferung des *App Service* mittels Versionsverwaltungssystemen. Aber im Laufe der Zeit wurden weitere Funktionen ergänzt. Dazu zählt auch die Funktion *Web Jobs*, welche die Abarbeitung von Hintergrundaufgaben ermöglicht. Für kleinere Aufgaben, für die eine eigene virtuelle Maschine überdimensioniert wäre, sind *Web Jobs* oft eine wirtschaftlichere Alternative.

Web Jobs

Als *Web Job* eignen sich ausführbare Dateien oder unterstützte Skripte. Diese werden als eigener Prozess im Kontext eines *App Service* entweder zeitgesteuert oder kontinuierlich ausgeführt. Damit können Hintergrundaufgaben überschüssige Kapazität ausnützen. Es kann aber auch zu negativen Effekten kommen, denn Hintergrundaufgaben können die Performanz der eigentlichen Web-Anwendung für den Endbenutzer beeinflussen.

2.3.3 Web Jobs SDK

Die *Web Jobs SDK* ist eine .NET-Bibliothek für die Implementierung ereignisgesteuerter Hintergrundaufgaben. Diese Bibliothek bietet eine reichhaltige Integration externen Datenquellen. Ein deklaratives Programmiermodell sorgt dafür, dass Entwickler die Anbindung an eine Datenquelle deklarativ beschreiben können, ohne selbst eine Zeile Quelltext zu schreiben [Chab].

Um vorab einen Einblick zu geben, zeigt Programm 2.1 beispielhaft die Verwendung dieser Bibliothek. Der nächste Abschnitt enthält dann eine detailliertere Aufarbeitung der verwendeten Konzepte. Die folgende Aufzählung gibt eine Übersicht wichtiger Aspekte des Einführungsbeispiels:

- Die Zeilen (1-5) definieren eine Datenstruktur mit drei Attributen.
- Zeile (7) startet die *Web Jobs SDK* Laufzeitumgebung. Diese sucht per .NET Reflection nach kompatiblen Funktionen.
- Zeile (10) definiert ereignisgesteuerte Funktion.
- Zeile (11) definiert das Ereignis, das einen Funktionsaufruf auslöst. Hier wird die Funktion bei jedem neuen Eintrag in einer Warteschlange aufgerufen und dessen Inhalt automatisch an den Funktionsparameter übergeben.
- Zeile (12) definiert einen Ausgangsparameter. Wertzuweisungen an diesen Parameter werden automatisch in dem konfigurierten *Blob Storage* gespeichert. Für den Pfad dieses Eintrags werden die aus dem Parameter in Zeile (11) übergebenen Werte extrahiert.
- Zeile (13) definiert einen weiteren Ausgangsparameter, der an eine andere Warteschlange weitergeleitet wird. Dieses Ereignis könnte wiederum einen anderen Funktionsaufruf auslösen.

Programm 2.1: Web Jobs SDK Beispiel

```
1 public class SensorData {  
2     public string SensorId { get; set; }  
3     public string Timestamp { get; set; }  
4     public double Value { get; set; }  
5 }  
6 class Program {  
7     static void Main() => new JobHost().RunAndBlock();  
8 }  
9 public class Functions {  
10     public static void ProcessSensorData(  
11         [QueueTrigger("sensorqueue")] SensorData sensorData,  
12         [Blob("values/{SensorId}/{Timestamp}")] out string sensorValue,  
13         [Queue("aggregationqueue")] out SensorData aggregationQueue) {  
14         sensorValue = sensorData.Value.ToString();  
15         aggregationQueue = sensorData;  
16     }  
17 }
```

Obwohl die Funktion in Programm 2.1 mit drei verschiedenen externen Datenquellen interagiert, enthält sie dafür fast keinen imperativen Quelltext, sondern lediglich eine deklarative Beschreibung in Form von Attributen. Die eigentliche Interaktion und das Warten auf Ereignisse ist die Aufgabe der *Web Jobs SDK*.

Eine *Web Job* Anwendung ist nichts anderes als eine gewöhnliche .NET Applikation mit einer Referenz auf die *Web Jobs SDK*. Der nächste Abschnitt beschreibt, wie statische Methoden einer *Web Job* Anwendungen zu einem ereignisgesteuerten *Web Job* werden.

Für viele Einsatzgebiete reicht dieses einfache, deklarative Programmiermodell aus. Es spricht aber nichts dagegen, die Interaktionslogik mit einer Datenquelle selbst im Funktionsrumpf zu implementieren. Das ist oft sogar notwendig, wenn der Funktionsumfang der deklarativen Variante nicht ausreicht. Der Grundgedanke ist jedoch, derartigen repetitiven Standardquelltext durch deklarative Programmierung zu vermeiden.

2.3.4 Bindungen

Bindungen definieren, wann eine Funktion aufgerufen wird und wie sie über ihre Funktionsparameter mit Datenquellen interagieren kann. Die zwei wesentlichen *Web Job* Bindungen sind Trigger-Bindungen und Nicht-Trigger-Bindungen. Erstere überwachen eine Ereignisquelle und lösen den Aufruf einer Job Funktion aus, wenn ein Ereignis auftritt. Zweitere binden Parameter einer Job Funktion an eine externe Datenquelle. Das kann sowohl zum Lesen, als auch zum Schreiben von Daten sein. Darum werden Nicht-Trigger-

Bindungen in Eingangs- und Ausgangsbindungen eingeteilt. Jede Funktion muss genau eine Trigger-Bindung haben, kann aber beliebig viele Nicht-Trigger-Bindungen enthalten.

Es gibt bereits eine Vielzahl vorhandener Bindungen. Entwickler können die Bibliothek aber auch mit selbst oder von Drittanbietern entwickelten Bindungen erweitern. Dazu muss man die wichtigsten dafür notwendigen Klassen kennen. Abbildung 2.1 zeigt den Bindungsmechanismus von Nicht-Trigger-Bindungen in einem stark vereinfachten Klassendiagramm.

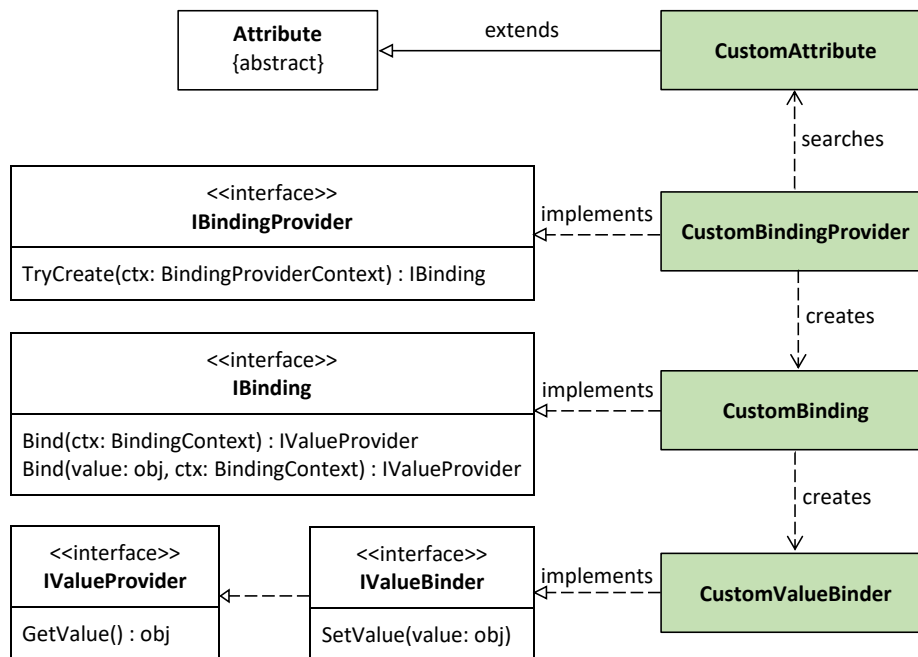


Abbildung 2.1: Web Jobs SDK Klassendiagramm (Nicht-Trigger-Bindung)

Wie [Chac] beschreibt, ist der Bindungsprozess in zwei Phasen eingeteilt. Die nächsten Abschnitte beschreiben diese Phasen und geben somit auch Aufschluss darüber, wie die in Abbildung 2.1 skizzierten Klassen in Zusammenhang stehen.

Startphase

Beim Starten der Anwendung werden Funktionen gesucht, die sich als *Web Job* eignen. Dieser Prozess umfasst folgende Schritte:

1. Das Assembly der Anwendung wird nach allen Methoden in allen öffentlichen Klassen durchsucht.
2. Für jeden Parameter einer Methode wird versucht, eine Bindung zu erzeugen.

3. Jeder registrierte `I(Trigger)BindingProvider` bekommt die Möglichkeit eine Bindung über die Fabrikmethode `TryCreate` zu erstellen. Diese Methode überprüft den Datentyp des Parameters und meistens die Existenz des Bindungs-Attributs, welches die Metadaten der Bindung beinhaltet. Sind die Voraussetzungen einer konkreten Bindung erfüllt, wird diese erzeugt und zurückgegeben.
4. Wenn alle Parameter gebunden werden konnten, wird die Funktion in der *Web Jobs* Laufzeitumgebung registriert.
5. Für jede Trigger-Bindung wird zusätzlich die Überwachung der jeweiligen Ereignisquelle gestartet.

Laufzeitphase

Nachdem alle Funktionen identifiziert wurden, beginnt die Laufzeitphase. In dieser Phase passiert die tatsächliche Bindung der Funktionsparameter. Dieser Ablauf lässt sich wie folgt zusammenfassen:

1. Das Eintreten eines Ereignisses einer Trigger-Bindung führt zur Ausführung der assoziierten Job Funktion.
2. Für jeden gebundenen Parameter wird die `BindAsync` Methode der zur Startphase festgelegten Bindung aufgerufen. In dieser passiert der eigentliche Interaktionsschritt mit einer externen Datenquelle, wie z. B. dem Lesen von Daten aus einer Datenbank.
3. Oft unterstützen Bindungen verschiedene Parametertypen, wie z. B. `String` oder `Stream`. Die Bindung muss ihren tatsächlichen Wertebereich auf den Datentyp der aufzurufenden Funktion konvertieren.
4. Nachdem alle Parameter gebunden wurden, kann der eigentliche Funktionsrumpf aufgerufen werden.

2.3.5 Web Jobs Script SDK

Die *Web Jobs Script SDK* ist das Herzstück von *Azure Functions*. Es ist eine .NET-Bibliothek, die eine interaktive Verwendung der sonst nur für .NET Applikationen geeigneten *Web Jobs SDK* auch anderen Programmiersprachen zugänglich macht. Damit konnten die bereits für *Web Jobs* entwickelten Bindungen für *Azure Functions* wiederverwendet werden.

Wie in Programm 2.1 gezeigt, agiert die Klasse `JobHost` als Laufzeitumgebung von *Web Job* Funktionen. In der *Web Jobs Script SDK* Bibliothek ist die Klasse `ScriptHost` von `JobHost` abgeleitet und um dynamische Aspekte erweitert. Anstelle die Funktionen in eine .NET Anwendung zu verpacken, werden hier die Funktionen als Skript-Dateien in einer bestimmten Ordnerstruktur abgelegt. Die Skript-Laufzeitumgebung überwacht diese Ordnerstruktur auf Änderungen. Wenn eine Funktion hinzugefügt wird, löst das

unmittelbar eine Aktualisierung und bei kompilierten Sprachen eine erneute Übersetzung der Funktion aus.

Programmiermodell

Eine *Azure Function* Applikation ist im Grunde nur eine Ordnerstruktur. Jede Funktion befindet sich in einem eigenen Ordner und enthält mindestens zwei Dateien. Meistens sind das eine Skript-Datei, mit dem eigentlichen Quelltext der Funktion und eine Konfigurationsdatei, in der die Metadaten der Parameterbindungen definiert sind. Abschnitt 2.3.3 hat die attributbasierte Definition der Bindungsmetadaten gezeigt. Für *Azure Functions* ist diese Umsetzung untauglich, weil nicht alle Programmiersprachen einen derartigen Mechanismus besitzen.

Programm 2.2: Azure Functions Beispiel C#

<pre>// function.json { "bindings": [{ "name": "sensorData", "type": "queueTrigger", "direction": "in", "queueName": "sensorqueue" }, { "type": "blob", "name": "sensorValue", "path": "values/{SensorId}/{Timestamp}", "direction": "out" }, { "type": "queue", "name": "aggregationQueue", "queueName": "aggregationqueue", "direction": "out" }] }</pre>	<pre>// run.csx public static void Run(SensorData sensorData, out string sensorValue, out SensorData aggregationQueue) { sensorValue = sensorData.Value.ToString(); aggregationQueue = sensorData; } public class SensorData { public string SensorId {get;set;} public string Timestamp {get;set;} public double Value {get;set;} }</pre>
---	--

Die in Programm 2.2 dargestellte *Azure Function* ist funktional äquivalent zu der in Programm 2.1 gezeigten *Web Job* Variante. Aus programmier-technischer Sicht ist der größte Unterschied, die Definition der Bindungsmetadaten in Form einer sprachunabhängigen Konfigurationsdatei, die auch für andere Sprachen verwendet werden kann. In diesem Beispiel ist der eigentliche Quelltext in einer C#-Skript-Datei implementiert, die aber durch jede andere unterstützte Sprache ausgetauscht werden könnte.

Der folgende Abschnitt beschreibt, wie die dynamische Übersetzung von *Azure Functions* während der Laufzeit funktioniert.

Übersetzungsvorgang

Die Hauptaufgabe der *Web Jobs Script SDK* Bibliothek ist die dynamische Übersetzung der verschiedenen unterstützten Programmiersprachen und die Integration dieser dynamisch übersetzten Funktionen in die *Web Jobs SDK* Laufzeitumgebung. Für jede Programmiersprache ist ein anderer Übersetzungsprozess notwendig. Nachfolgend wird dieser Prozess für die kompilierten Sprachen C# und F#, sowie für die interpretierte Sprache Javascript, kurz dargestellt.

Die *Roslyn Compiler API* ist eine .NET-Bibliothek, die unter anderem die Übersetzung von C#-Quelltext ermöglicht [Hej+12, S. 5]. Auch für die Sprache F# gibt es eine derartige Bibliothek. Die Skript-Laufzeitumgebung verwendet diese beiden Bibliotheken, um dynamisch C# und F# Skripte zu übersetzen und das Kompilat in ihren Prozess zu laden. Ab diesem Zeitpunkt kann die übersetzte Funktion ganz normal in der *Web Jobs* Laufzeitumgebung verwendet werden.

Wesentlich aufwändiger gestaltet sich die Interaktion mit der *Web Jobs SDK*, bei Sprachen die nicht der .NET-Familie angehören. Bei JavaScript Funktionen beispielsweise, ist eine Brücke zwischen der .NET *Common Language Runtime* und der *JavaScript*-Laufzeitumgebungen notwendig. Dieses Problem löst eine Bibliothek mit dem Namen *Edge.js*. Damit ist es möglich, .NET und *Node.js* Quellcode im selben Prozess auszuführen, indem beide Laufzeiten im selben Prozess geladen werden [Jan]. Das ist wesentlich effizienter, als beide Umgebungen getrennt auszuführen und über Interprozesskommunikation zu verbinden.

Die Programmiermodelle von .NET und *Node.js* unterscheiden sich teilweise gravierend. In *Node.js* wird Nebenläufigkeit beispielsweise durch Callback-basierte Programmierung gelöst, weil die virtuelle JavaScript-Maschine nur einen einzigen Ausführungsstrang nutzt. In .NET gibt es diese Einschränkung nicht. Hier wird Task-basierte Nebenläufigkeit bevorzugt. Programm 2.3 zeigt aber, dass beide Konzepte isomorph sind und sich darum trotzdem gut verbinden lassen.

Programm 2.3: Brücke zwischen Task-basierter Programmierung in .NET und Callback-basierter Programmierung in JavaScript

```
1 Func<object, Task<object>> >  
2  
3
```

```
1 function (arguments, callback) {  
2   callback(error, result);  
3 }
```

Die Kompilierung oder Interpretation der Job Funktion zur Laufzeit, ermöglicht ein sehr effizientes ausrollen dieser Funktionen. Es muss nicht die ganze Host-Anwendung neu übersetzt und Ausgerollt werden, wie es bei den in Abschnitt 2.3.3 beschriebenen traditionellen *Web Jobs* der Fall war.

2.3.6 Azure Function App

Eine *Function App* ist in *Microsoft Azure* ein auf die Bedürfnisse von *Azure Functions* zugeschnittener *App Service*, der mehrere zusammengehörende *Azure Functions* zusammenbindet. Alle *App Services* haben gemeinsame Basisfunktionen, wie z. B. Logging, Monitoring, Deployment-, sowie Skalierbarkeitsoptionen. Eine *Function App* hat eine zusätzliche *Site Extension*, in der die in Abschnitt 2.3.5 beschriebene Skript-Laufzeitumgebung ausgeführt wird. Auf diese Weise ist auch die Verwendung von HTTP-Triggern möglich, weil eine *Site Extension* auch die Kommunikation über HTTP ermöglicht. Damit lassen sich mit *Azure Functions* auch sehr schnell einfache Web-Services entwickeln. Alternativ wäre es auch möglich, die Skript-Laufzeitumgebung in einer .NET-Konsolenanwendung als kontinuierlichen *Web Job* eines *App Service* auszuführen. Damit wäre aber das Empfangen von HTTP-Anfragen nicht möglich.

Als zusätzliche Erweiterung bieten *Function Apps* auch eine online Entwicklungsumgebung, die in das *Microsoft Azure Portal* integriert wurde. Somit können Funktionen interaktiv erstellt, geändert und gelöscht werden. Dateiänderungen werden automatisch von der Skript-Laufzeitumgebung erkannt und die geänderten Funktionen neu übersetzt. In der Entwurfsphase ist diese Möglichkeit der Entwicklung sehr produktiv. Danach sollte aber ein automatisierter Prozess, z. B. über ein Versionsverwaltungssystem, bevorzugt werden.

Der *App Service Plan* eines *App Service* definiert, aus welchen und wie vielen virtuellen Servern die Infrastruktur bestehen soll. Die zur Verfügung gestellten Ressourcen stehen permanent zur Verfügung und verursachen somit auch laufend Kosten, wenn die Anwendungen keine Ressourcen beanspruchen. Dieses Konzept steht im Widerspruch zur Grundidee von *Function-as-a-Service*. Hier soll die Verrechnung auf Basis der Funktionsaufrufe erfolgen.

Für eine konsequente Implementierung von *FaaS* erweiterte *Microsoft* den klassischen *App Service Plans* um einen sogenannten *Consumption Plan*. Mit dieser Variante hat der Entwickler keinerlei Möglichkeit mehr, die Skalierbarkeitseigenschaften zu bestimmen. *Microsoft Azure* übernimmt völlig automatisch die Skalierung auf die optimale Größe.

Das Verrechnungsmodell bei einem *Consumption Plan* stellt nur tatsächlich konsumierte Ressourcen in Rechnung. In vielen Fällen lassen sich damit Kostenersparnisse gegenüber Varianten mit reservierter Ressourcenzeit erzielen. Man verliert aber jegliche Einflussmöglichkeit auf die Ressourcen, sodass man beispielsweise das in Abschnitt 2.3.8 beschriebene Kaltstartproblem in Kauf nehmen muss.

Derzeit ist die Verwendung eines traditionellen *App Service Plans* und eines *Consumption Plans* möglich. Konsumenten haben also die Wahlfreiheit oder können bestehende *App Service Plans* für *Azure Functions* nutzen. Die

meisten anderen Anbieter am Markt haben nur ein dem *Consumption Plan* ähnliches Modell im Angebot.

2.3.7 Verrechnungsmodell

Dieser Abschnitt bezieht sich auf den zuvor beschriebenen *Consumption Plan*, da dieser die größte Übereinstimmung mit den Ideen hinter serverloser Softwarearchitektur hat. Die Kosten von *Azure Functions* werden durch die Anzahl der Funktionsaufrufe und den Ressourcenkosten bestimmt. Für die Berechnung der Aufrufkosten wird die Anzahl der Funktionsaufrufe mit dem festgelegten Preis pro Aufruf multipliziert.

Hinter den Ressourcenkosten steckt eine aufwändigere Berechnung. Zuerst wird die Anzahl der Aufrufe mit der Aufrufdauer multipliziert, um den Ressourcenverbrauch in Sekunden zu ermitteln. Dieser Wert wird mit dem durchschnittlichen Speicherverbrauch in Gigabyte multipliziert. Danach erhält man den Ressourcenverbrauch in Gigabyte-Sekunden. Auch pro Gigabyte-Sekunde gibt es einen festgelegten Tarif, mit dem man die Ressourcenkosten berechnen kann.

Die Gesamtkosten setzen sich aus den Aufrufkosten und den Ressourcenkosten zusammen. Den Hauptteil der Kosten nehmen meistens die Ressourcenkosten ein, die abhängig vom Speicherverbrauch und der Aufrufdauer sind. In Tabelle 2.1 ist die gesamte Berechnung noch einmal verdeutlicht.

Bei einem klassischen *App Service Plan* berechnen sich die Kosten aus der Anzahl und Größe der virtuellen Maschinen. Aber wie bereits erwähnt, entspricht dieses Verrechnungsmodell nicht den Grundgedanken von *FaaS*.

Tabelle 2.1: Azure Functions Preiskalkulation

	<i>Anzahl der Aufrufe</i>	
*	<i>Aufrufdauer</i>	
=	<i>Ressourcenverbrauch in Sekunden</i>	
*	<i>Speicherverbrauch in GB</i>	
=	<i>Ressourcenverbrauch in GB-s</i>	<i>Anzahl der Aufrufe</i>
		10^6
*	<i>Preis pro GB-s</i>	*
=	<i>Ressourcenverbrauchskosten</i>	<i>Preis pro 10^6 Aufrufe</i>
		=
	<i>Aufrufkosten</i>	
	<i>Ressourcenverbrauchskosten</i>	
+	<i>Aufrufkosten</i>	
=	<i>Gesamtkosten</i>	

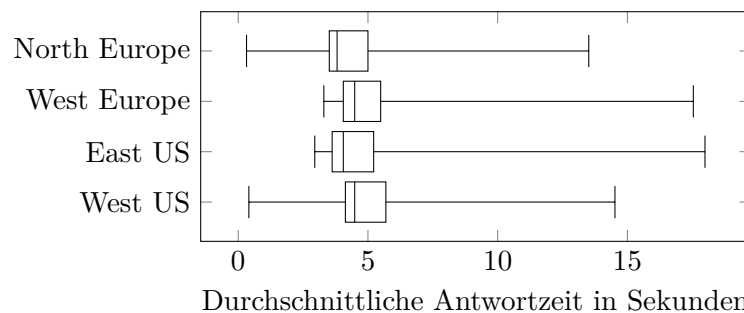
2.3.8 Kaltstart

Die dynamische Ressourcenbereitstellung des *Consumption Plans* ermöglicht zwar eine sehr feingranulare und kosteneffiziente Verrechnung von Funktionsaufrufen, jedoch hat diese Art auch Nachteile. Einer davon sind sogenannte Kaltstarts. Darunter versteht man den Effekt, dass der erste Funktionsaufruf nach längerer Inaktivität viel länger dauert, als durchschnittliche Aufrufe.

In *AWS Lambda* werden Latenzzeiten von 100 bis 800 Millisekunden bei *Node.js* und bis zu 3500 Millisekunden bei *Java* Funktionen beobachtet [Ful16, Kap. 16]. Auch wenn *AWS Lambda* und *Azure Functions* mit völlig unterschiedlichen Technologien entwickelt wurden, leiden sie dennoch beide unter diesem Problem, wie das nachfolgende Experiment zeigt.

Um das Kaltstartproblem in *Azure Functions* zu analysieren, wurden über einen Zeitraum von zwei Wochen, in mehreren *Microsoft Azure* Regionen, Funktionen über einen HTTP-Trigger aufgerufen. Die Aufrufe selbst erfolgten aus der selben Region und hatten einen Abstand zwischen fünfzehn Minuten und zwölf Stunden. Abbildung 2.2 zeigt, dass ein HTTP-Aufruf nach längerer Inaktivität durchschnittlich etwa vier Sekunden benötigt. Es ist aber auch zu erkennen, dass die Antwortzeiten sehr große Schwankungen aufweisen.

Abbildung 2.2: Azure Functions Kaltstart-Latenzzeit



Ob die großen Latenzzeiten nach einem Kaltstart ein Problem darstellen, kommt auf das jeweilige Einsatzgebiet an. Bei einem kontinuierlichen Lastaufkommen oder bei nicht zeitkritischen Szenarien, kann dieses Problem auch vernachlässigbar sein. In diesem Bereich herrscht sicherlich noch großes Optimierungspotential seitens der *FaaS*-Anbieter.

2.3.9 Zusammenfassung

In diesem Abschnitt wurde *Azure Functions* als konkrete Technologie aus dem Bereich *Function-as-a-Service* vorgestellt. Es ist eine Möglichkeit, kleine unabhängige Funktionen ereignisgesteuert auszuführen. Die Alleinstellungs-

merkmale sind die einfache Interaktion mit externen Datenquellen und das serverlose Ressourcenmodell. Serverlos bedeutet nämlich, dass der Service-Anbieter sämtliche Infrastrukturaufgaben übernimmt, selbst die Skalierung.

Azure Functions baut sehr stark auf die vorhandenen Funktionen der *Web Jobs SDK* auf und macht diese auch Programmiersprachen außerhalb der .NET-Familie zugänglich. Mit all seinen Bindungen und Triggern an verschiedenste Datenquellen, besitzt *Azure Functions* einen reichhaltigen Baukasten für die Entwicklung von Microservices in der Cloud. Weil Funktionen automatisch hoch skalierbar und verfügbar sind, ist *FaaS* auch für anspruchsvolle Anwendungen geeignet. Spielt jedoch die Latenzzeit eine große Rolle, oder ist die Lastverteilung sehr hoch und vorhersehbar, sind andere Technologien möglicherweise besser geeignet.

In dieser Arbeit wurden keineswegs alle Möglichkeiten von *Azure Functions* betrachtet. Dazu zählen beispielsweise Sicherheit, Deployment oder Überwachung. Zudem befindet sich der ganze *FaaS*-Bereich noch stark in Entwicklung und wird deshalb ständig um neue Möglichkeiten ergänzt. Die Zukunft wird zeigen, wie sich dieser sehr junge Bereich entwickeln wird.

2.4 Evolution der Anwendungsentwicklung

Dieser Abschnitt bezieht sich überwiegend auf [Coca] und [Cocb], in denen der Autor Adrian Cockcroft die Evolution von monolithischer Software zu Microservices und weiter zu ereignisgesteuerten serverlosen Anwendungen beschreibt. Außerdem werden die dafür verantwortlichen technischen und organisatorischen Entwicklungen identifiziert.

Die Aufgabe von Software-Applikationen ist es, Geschäftswerte – *engl. business value* – zu generieren. Dazu müssen sie den Benutzern die enthaltene Geschäftslogik zugänglich machen. Eine Funktion kann erst Geschäftswerte erzeugen, wenn sie dem Benutzer tatsächlich zur Verfügung steht. Daher sollte für Unternehmen die Minimierung der Zeit zwischen der Erstellung der Geschäftslogik und der tatsächlichen Verfügbarkeit für den Endbenutzer an oberster Stelle stehen. Cockcroft beschreibt diesen Zusammenhang in folgender Formel:

$$\textit{Time to Value} = \textit{Creation Cost} + \textit{Delivery Cost}$$

In der Vergangenheit war die Dauer zwischen der Erstellung und der Auslieferung einer neuen Funktion oft sehr lange. Aufgrund des hohen Aufwands und der zahlreichen Risiken eines neuen Releases, wurden Anwendungen nur in sehr großen Zeitabständen ausgeliefert. Zu dieser Zeit waren monolithische Anwendungen, in Verbindung mit einer oder wenigen zentralen relationalen Datenbanken, der effizienteste Weg, Geschäftslogik bereitzustellen. Performanzbedenken war der wesentlichste Einflussfaktor auf die

Softwarearchitektur. Die folgenden drei Abschnitte erläutern wie Hardware-Fortschritte, Auslieferungsautomatisierung und organisatorische Veränderungen diese Probleme verringerten und somit den Weg für Microservices und serverlose Anwendungen ebneten.

2.4.1 Automatisierung der Softwareauslieferung

Vor einigen Jahren war die Bereitstellung von Software noch ein manueller Prozess. Die Beschaffung, Installation, Konfiguration und Aktualisierung von physischen Servern war ein wesentlicher Zeit- und Kostenfaktor. Wegen dieser langsamen Vorgänge wurden neue Versionen einer Anwendung nur selten ausgerollt. Einzelne Versionen mussten dementsprechend viel Geschäftslogik beinhalten, damit sich die Kosten der Bereitstellung amortisierten.

Eine zusätzliche Herausforderung stellte die Kapazitätsplanung dar. Die Trägheit der Infrastrukturbereitstellung führte zu einer chronischen Überdimensionierung der Systeme. Das führte wiederum zu einer unökonomischen Ressourcenauslastung.

Die Automatisierung von Infrastrukturaufgaben regte eine Überarbeitung festgefahrener Softwareauslieferungsprozesse an. Werkzeuge, wie *Chef* und *Puppet*, erlaubten es erstmals, Skripte für die automatische Provisionierung und Konfiguration von Infrastrukturkomponenten zu erstellen. Zu diesen Komponenten zählen Server, Betriebssystem, Netzwerk, Konfiguration, aber auch die Applikationssoftware selbst. Heute bezeichnet man diese Möglichkeiten als *Infrastructure-as-Code*, weil sich Infrastruktur mittels Quelltext erstellen und manipulieren lässt. [Hüt12, S. 135].

Wenn sich Infrastruktur wie Quelltext behandeln lässt, ist es naheliegend, dass auch Softwareentwickler an diesem Prozess teilnehmen. Die Infrastrukturprovisionierung und -verwaltung verlagerte sich allmählich von den IT- in die Softwareentwicklungsabteilungen. IT-Abteilungen und Cloud-Anbieter stellten den Entwicklern nur noch Programmierschnittstellen – auch API genannt – zur Verfügung, mit denen sie selbst die Infrastruktur nach ihren Bedürfnissen erzeugen und verändern konnten. Diese Zeit- und Kostenreduktion bei der Softwareauslieferung ermöglichten häufigere Releases. Schlussendlich war das eine der Voraussetzungen für die Microservice-Architektur, da von nun an eine effiziente Bereitstellung von vielen kleinen Anwendungen möglich war. Die Auslieferung von Services veränderte sich von einem langsamen und risikoreichen, zu einem automatisierten, stabilen und kostengünstigen Prozess.

Alle Anbieter serverloser Plattformen haben auf die Erfahrungen der Vergangenheit aufgebaut und Automatisierbarkeit von Anfang an berücksichtigt. Manuell könnten Softwaresysteme mit einer großen Anzahl von Funktionen kaum gehandhabt werden.

2.4.2 Leistungsverbesserung der Hardware

Erst die Steigerung der Netzwerkübertragungs- und Festplattengeschwindigkeiten der jüngeren Vergangenheit, haben den Weg für echte serviceorientierte Architekturen geebnet. Obwohl die Ideen hinter SOA schon lange existieren, wurden sie aufgrund von Leistungsengpässen gar nicht oder nur unzureichend umgesetzt.

Nachrichtenorientierte Systeme bedeuten immer einen gewissen Mehraufwand für die Übertragung und Kodierung der Nachrichten. Die Netzwerkgeschwindigkeit hat sich in den letzten Jahren um das zehn- bis hundertfache gesteigert [DAm+]. In der Ethernet-Spezifikation *IEEE 802.3-2015* sind Übertragungsraten von bis zu 100 Gigabit pro Sekunde spezifiziert.

Ein weiterer Hemmschuh bei der Nachrichtenübertragung waren schwergewichtige, oft XML-basierte, Kodierungsprotokolle, wie z.B. SOAP. Erst die Entwicklung von leichtgewichtigeren bzw. effizienteren Protokollen, in Kombination mit den um Größenordnungen schnelleren Netzwerkverbindungen, leiteten den Siegeszug von nachrichtenorientierten Systemen ein.

Auch in der Speichertechnologie passierte durch die Ablösung von magnetischen Festplatten durch *Solid-State*-Festplatten ein weiterer essentieller Technologiefortschritt. Magnetische Festplatten sind ihren neuen Konkurrenten vor allem bei zufälligen Lesezugriffen deutlich unterlegen [RCC12]. Diese schlechten Zugriffszeiten waren der Grund für das Design großer monolithischer Datenbanken. Geschäftslogikfunktionen mussten viele Operation in einer Transaktion durchführen, um die langsamen Zugriffszeiten zu kaschieren.

Auf Basis der sehr schnellen *Solid-State*-Festplatten wurden eine Menge neuer *NoSQL* Datenbanken entwickelt. Diese wiederum haben die Dezentralisierung und die in Abschnitt 1.2.3 beschriebene polyglotte Persistenz der bis dato hauptsächlich monolithischen Applikation vorangetrieben.

Auch Funktionen in *FaaS* machen intensiven Gebrauch von Nachrichtenübertragung und verschiedenen Datenspeichern. Im Grunde transformieren sie Daten, wann immer sie über eine Ereignis benachrichtigt werden.

2.4.3 Organisatorische Veränderungen

Abschnitt 1.2.1 hat bereits beschrieben, dass die Implementierung einer Microservice-Architektur mit großer Wahrscheinlichkeit eine organisatorische Umstrukturierung mit sich bringt. Projekt- oder technologiebezogene Strukturen sollten in produktbezogene umgewandelt werden. Um die Autonomie zu steigern und den Koordinationsaufwand zu senken, ist es empfehlenswert, große Teams in kleinere Einheiten zu zerteilen. Jedes dieser kleinen Teams ist für den gesamten Lebenszyklus eines oder mehrerer Dienste verantwortlich. Diese Struktur erlaubt eine viel agilere Entwicklung und erfordert weniger definierte Prozesse.

2.4.4 Von Microservices zu serverlosen Anwendungen

Wenn man Microservices auf die Spitze treibt, erfüllt jeder Service nur noch eine einzige Aufgabe. Aufgrund der großen Anzahl von Services in einer Microservice-Architektur, erfordert dieser Ansatz eine extrem effiziente Bereitstellung von Services. Selbst automatisch erstellte virtuelle Server und Container sind dafür nicht ausreichend. Für diese Anforderung ist *Function-as-a-Service* eine gute Wahl, denn Funktionen lassen sich in Bruchteilen einer Sekunde ausrollen.

Viele Services werden nur selten oder sehr unregelmäßig genutzt. In diesen Szenarien ist es schwierig, mit Containern oder virtuellen Maschinen die Kapazität zum richtigen Zeitpunkt bereitzustellen. Die Auslastung der Ressourcen ist in solchen Fällen meistens nicht optimal. *Function-as-a-Service* ist eine gute Möglichkeit, um mit volatilen Lastaufkommen umzugehen. Anstatt Rechenkapazität dezidiert zu reservieren, werden Funktionen erst bei Bedarf ausgerollt.

Funktionen enthalten beinahe ausschließlich Geschäftslogik. Es ist kaum Standard- oder Plattform Quelltext notwendig. Damit gelingt es Entwicklern oft binnen weniger Tage, neue Funktionen zu entwickeln. Dabei verbinden sie einfach eine Funktion mit anderen Funktionen oder Services von Drittanbietern. Ein großer Vorteil von *Function-as-a-Service* ist, dass die entwickelten Funktionen automatisch hoch skalierbar und hoch verfügbar sind.

Alle zuvor beschriebenen Eigenschaften machen *FaaS* zu einer wertvollen Ergänzung der Microservice-Architektur. Es gibt aber durchaus Szenarien, in denen es nicht gut geeignet ist, beispielsweise wenn die Last sehr groß und vorhersehbar ist.

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).
- [DAm+] John D'Ambrosia, Pete Anslow, David J. Law, Wael William Diab, Adam Healey, Steven B. Carlson und Valerie Maguire. *IEEE 802.3™ Industry Connections Ethernet Bandwidth Assessment*. Techn. Ber. URL: http://www.ieee802.org/3/ad_hoc/bwa/BWA_Report.pdf (besucht am 24.02.2017) (siehe S. 29).
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (siehe S. 2).
- [Ful16] Matthew Fuller. *AWS Lambda: A Guide to Serverless Microservices*. 2016 (siehe S. 26).
- [Hej+12] Anders Hejlsberg, Peter Golde, Matt Warren und Karen Ng. „The Roslyn Project: Exposing the C# and VB compiler's code analysis“. In: Sep. 2012. URL: <https://www.microsoft.com/en-us/download/details.aspx?id=27744> (siehe S. 23).
- [Hor16] Christian Horsdal. *Microservices in .NET Core*. Manning, 2016 (siehe S. 4).
- [Hüt12] Michael Hüttermann. *DevOps for Developers*. 1st. Berkely, CA, USA: Apress, 2012 (siehe S. 28).

- [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).
- [Rai15] Rick Rainey. *Azure Web Apps for Developers*. Microsoft Press, 2015 (siehe S. 17).
- [RCC12] Nathan Regola, David A. Cieslak und Nitesh V. Chawla. „The Constraints of Magnetic Versus Flash Disk Capabilities in Big Data Analysis“. In: *Proceedings of the 2Nd Workshop on Architectures and Systems for Big Data*. ASBD '12. Portland, Oregon, USA: ACM, 2012, S. 4–9. URL: <http://doi.acm.org/10.1145/2379436.2379437> (siehe S. 29).
- [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).
- [Chaa] Mathew Charles. *Azure Functions: The Journey*. URL: <https://blogs.msdn.microsoft.com/appserviceteam/2016/04/27/azure-functions-the-journey/> (besucht am 04.03.2017) (siehe S. 16).
- [Chab] Mathew Charles. *Binding Attributes*. URL: <https://github.com/Azure/azure-webjobs-sdk-extensions/wiki/Binding-Attributes> (besucht am 06.03.2017) (siehe S. 18).
- [Chac] Mathew Charles. *Binding Attributes*. URL: <https://github.com/Azure/azure-webjobs-sdk-extensions/wiki/The-Binding-Process> (besucht am 06.03.2017) (siehe S. 20).

- [Coca] Adrian Cockcroft. *Evolution of Business Logic from Monoliths through Microservices, to Functions*. URL: <https://read.acloud.guru/evolution-of-business-logic-from-monoliths-through-microservices-to-functions-ff464b95a44d> (besucht am 24.02.2017) (siehe S. 27).
- [Cocb] Adrian Cockcroft. *microXchg 2017 - Adrian Cockcroft: Shrinking Microservices to Functions*. URL: <https://www.youtube.com/watch?v=ZgxZCXouBkY> (besucht am 23.02.2017) (siehe S. 27).
- [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. 3).
- [Fowf] Martin Fowler. *PolyglotPersistence*. URL: <http://martinfowler.com/bliki/PolyglotPersistence.html> (besucht am 09.09.2016) (siehe S. 6).
- [Jan] Tomasz Janczuk. *Run .NET and Node.js code in-process with Edge.js*. URL: https://www.infoq.com/articles/the_edge_of_net_and_node (besucht am 11.03.2017) (siehe S. 23).
- [Kir] Yochay Kiriati. *Announcing general availability of Azure Functions*. URL: <https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/> (besucht am 04.03.2017) (siehe S. 16).
- [Mas] Nir Mashkowski. *Introducing Azure Functions*. URL: <https://azure.microsoft.com/en-us/blog/introducing-azure-functions/> (besucht am 04.03.2017) (siehe S. 16).
- [Rob] Mike Roberts. *Serverless Architectures*. URL: <http://martinfowler.com/articles/serverless.html> (besucht am 05.10.2016) (siehe S. 13).