Fakultät für Informatik und Mathematik 07

Bachelorarbeit

über das Thema

Sinnvolle Einsatzmöglichkeiten und Umsetzungsstrategien für Serverless Webanwendungen

Meaningful Capabilities and Implementation Strategies for Serverless Web Applications

Autor: Thomas Großbeck
grossbec@hm.edu

Prüfer: Prof. Dr. Ulrike Hammerschall

Abgabedatum: 07.03.19

Fakultät für Informatik und Mathematik
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

Diese Erklärung ist zusammen mit der Bachelorarbeit bei dem/der PrüferIn abzugeben.

Großbeck, Thomas	München, 07.03.2019
(Familienname, Vorname)	(Ort, Datum)

12.01.1997	IF 7 / WS / 20 15
(Geburtsdatum)	(Studiengruppe / WS/SS)

Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____
(Unterschrift)

I Kurzfassung

Das Ziel der Arbeit ist es, Unterschiede in der Entwicklung von Serverless und klassischen Webanwendungen zu betrachten. Es soll ein Leitfaden entstehen, der Entwicklern und IT-Unternehmen die Entscheidung zwischen den beiden Anwendungstypen erleichtert. Dazu wird zuerst eine Einführung in die Entwicklung des Cloud Computings und insbesondere in das Themenfeld des Serverless Computing gegeben. Im nächsten Schritt werden zwei beispielhafte Anwendungen entwickelt. Zum einen eine klassische Webanwendung. Verwendet wird hier das Spring Framework im Backend und ein Javascript basiertes Frontend. Zum anderen eine Serverless Webanwendung, die über dasselbe Frontend erreicht werden kann. Hierbei werden die Besonderheiten im Entwicklungsprozess von Serverless Applikationen hervorgehoben. Abschließend werden die beiden Vorgehensweisen durch vorher festgelegter Kriterien gegenübergestellt, sodass sinnvolle Einsatzmöglichkeiten für Serverless Anwendungen abgeleitet werden können.

II Inhaltsverzeichnis

I Kurzfassung **I**

II Inhaltsverzeichnis **II**

III Abbildungsverzeichnis **III**

IV Tabellenverzeichnis **III**

V Listing-Verzeichnis **IV**

VI Abkürzungsverzeichnis **IV**

1 Einführung und Motivation **1**

2 Grundlagen der Serverless Architektur **3**
2.1 Historische Entwicklung des Cloud Computings . . . 3
2.1.1 Grundlagen des Cloud Computings . . . 6
2.1.2 Abgrenzung zu PaaS . . . 8
2.1.3 Abgrenzung zu Microservices . . . 9
2.2 Eigenschaften von Function as a Service (FaaS) . . . 11
2.3 Allgemeine Pattern für Serverless Umsetzungen . . . 12
2.3.1 Serverless Computing Manifest . . . 13
2.3.2 Schnittstellen zu anderen Architekturen . . . 15

3 Vergleich zweier prototypischer Anwendungen **17**
3.1 Vorgehensweise beim Vergleich der beiden Anwendungen . . . 17
3.2 Fachliche Beschreibung der Beispiel-Anwendung . . . 19
3.3 Implementierung der Benutzeroberfläche . . . 20
3.4 Implementierung der klassischen Webanwendung . . . 23
3.4.1 Architektonischer Aufbau der Applikation . . . 23
3.4.2 Implementierung der Anwendung . . . 26
3.4.3 Testen der Webanwendung . . . 33
3.5 Implementierung der Serverless Webanwendung . . . 35
3.5.1 Architektonischer Aufbau der Serverless Applikation . . . 35
3.5.2 Implementierung der Anwendung . . . 38
3.5.3 Testen von Serverless Anwendungen . . . 43
3.6 Unterschiede in der Entwicklung . . . 46
3.6.1 Durchführung der Evaluation . . . 46
3.6.2 Auswertung der Evaluation . . . 52

4 Vergleich der beiden Umsetzungen **55**
4.1 Vorteile der Serverless Infrastruktur . . . 55
4.2 Nachteile der Serverless Infrastruktur . . . 56
4.3 Abwägung sinnvoller Einsatzmöglichkeiten . . . 57

5 Fazit und Ausblick **59**

6 Quellenverzeichnis **60**

Anhang **I**

A Vollständige Abbildung der Bewertungskriterien **I**

III Abbildungsverzeichnis

Abb. 1 Anteil der Unternehmen, die Cloud Dienste nutzen [KS17] 1
Abb. 2 Operativer Gewinn von Amazon [Bra18] 2
Abb. 3 Zusammenhang Kenntnisstand und Kontrolllevel [Bü17] 4
Abb. 4 Hierarchie der Cloud Services [Kö17, S. 28] 5
Abb. 5 Historische Entwicklung des Cloud Computings 6
Abb. 6 Verantwortlichkeiten der Organisation nach [Rö17a] 7
Abb. 7 Aufgabenverteilung: IaaS vs. PaaS vs. Serverless [Bü17] 9
Abb. 8 FaaS Beispiel Anwendung [Tiw16] 12
Abb. 9 Under- und Overprovisioning [A+09, S. 11] 14
Abb. 10 Zusammenhang zwischen Event-driven Computing, FaaS und Serverless [FIMS17, S. 5] 16
Abb. 11 Request Response Pattern [Swa18] 16
Abb. 12 Maske Bücherausleihe 22
Abb. 13 Maske Bücherverwaltung 22
Abb. 14 (1) Dialog Buch bearbeiten, (2) Dialog Buch löschen, (3) Dialog Buch hinzufügen 23
Abb. 15 3-Tier Architektur 24
Abb. 16 Layered Architektur nach [Ric15, S. 3] 26
Abb. 17 Beziehung zwischen User und Book 28
Abb. 18 Layered Architektur in Spring 29
Abb. 19 API Gateway 36
Abb. 20 Datenbankevent ruft Function auf 38
Abb. 21 Lambda Console 43
Abb. 22 Ausschnitt der Fragen mit entsprechenden Metriken 46
Abb. 23 Evaluation des Implementierungsaufwands 47
Abb. 24 Evaluation der Frameworkunterstützung 48
Abb. 25 Evaluation des Deploymentprozesses 49
Abb. 26 Evaluation der Testbarkeit 49
Abb. 27 Evaluation zur Erweiterbarkeit 50
Abb. 28 Antwortzeit der Serverless Anwendung 51
Abb. 29 Evaluation der Performanz 51
Abb. 30 Evaluation zur Sicherheit 52
Abb. 31 Netzdiagramm für die Evaluation der klassischen Anwendung 53
Abb. 32 Netzdiagramm für die Evaluation der Serverless Anwendung 54

IV Tabellenverzeichnis

Tab. 1 Normalisierung einer Ordinalskala mit 3 Werten nach [Zar17, S. 41] . 53

Tab. 2 Normalisierung der Anzahl genutzter Frameworks nach [Zar17, S. 41] 53

V Listing-Verzeichnis

1 One-Way Binding eines Textes [Pol18] 20
2 Auflistung der Elemente eines Arrays 21
3 Einstiegsklasse für eine Spring Boot Anwendung 27
4 Repository für die Tabelle User 29
5 Beispiel BookController 30
6 Implementierung des UserDetailsService 31
7 Abfrage des authentifizierten Users 32
8 PreAuthorize an einem Endpunkt im Controller 32
9 Ausschnitt aus dem StatisticControllerTest 34
10 Integrationstest für eine Methode aus dem Bookservice 34
11 Request Handler für Lambda Function 38
12 Ressourcendefinition der Beispiel Function in template.yaml 39
13 Modul zur Bereitstellung der Datenbankverbindung 40
14 GetBookFunction 40
15 Ausschnitt des BookDaos 41
16 Auslesen des DynamoDbStreams 42
17 Ausschnitt des StatisticDao Unittests 44
18 Test des GetBookHandlers 45

VI Abkürzungsverzeichnis

AWS Amazon Web Services

IaaS Infrastructure as a Service

PaaS Platform as a Service

FaaS Function as a Service

NIST National Institute of Standards and Technology

BaaS Backend as a Service

SaaS Software as a Service

SDK Software Development Kit

ORM Object-relational Mapping

JPA Java Persistence API

DI Dependency Injection

SAM Serverless Application Model

DAO Data Access Object

IAM Identity and Access Management

1 Einführung und Motivation

Durch das enorme Wachstum des Internets werden immer mehr Dienstleistungen online angeboten [Har02, S. 14]. Viele Dienste sind so als Webanwendung direkt zu erreichen und einfach zu bedienen. Mit der Einführung des Cloud Computings sind schließlich auch Rechenleistung und Serverkapazitäten über das Internet zur Verfügung gestellt worden.

Als eines der aktuell am schnellsten wachsenden Themenfeldern im Informatiksektor entwickelt sich Cloud Computing stetig weiter. So ist beispielsweise der Anteil der deutschen Unternehmen, die Cloud Dienste nutzen, in den letzten Jahren kontinuierlich gestiegen. Mittlerweile sind es bereits zwei Drittel der Firmen (siehe Abb. 1).

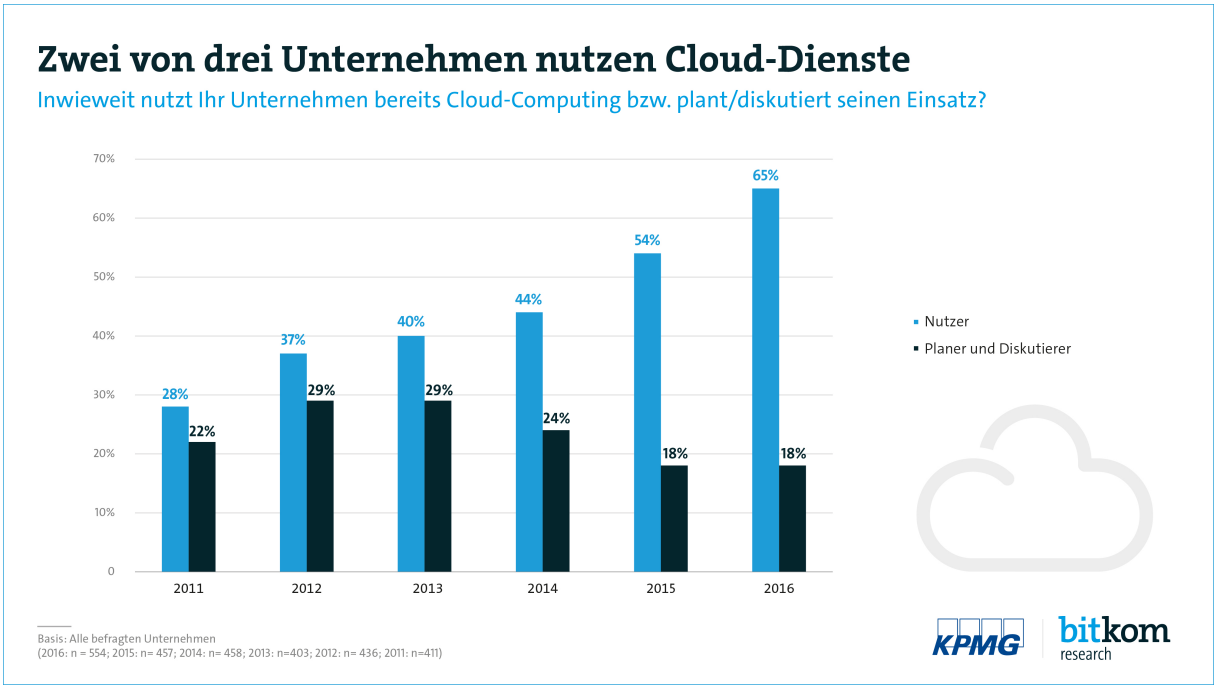

Abbildung 1: Anteil der Unternehmen, die Cloud Dienste nutzen [KS17]

Auf Seiten der Anbieter von Cloud Diensten ist ebenfalls ein großes Wachstum zu erkennen. Amazon als einer der Marktführer auf diesem Gebiet hat zum Beispiel im zweiten Quartal des Jahres 2018 55% des operativen Gewinns durch den Cloud Dienst Amazon Web Services (AWS) erzielt (siehe Abb. 2).

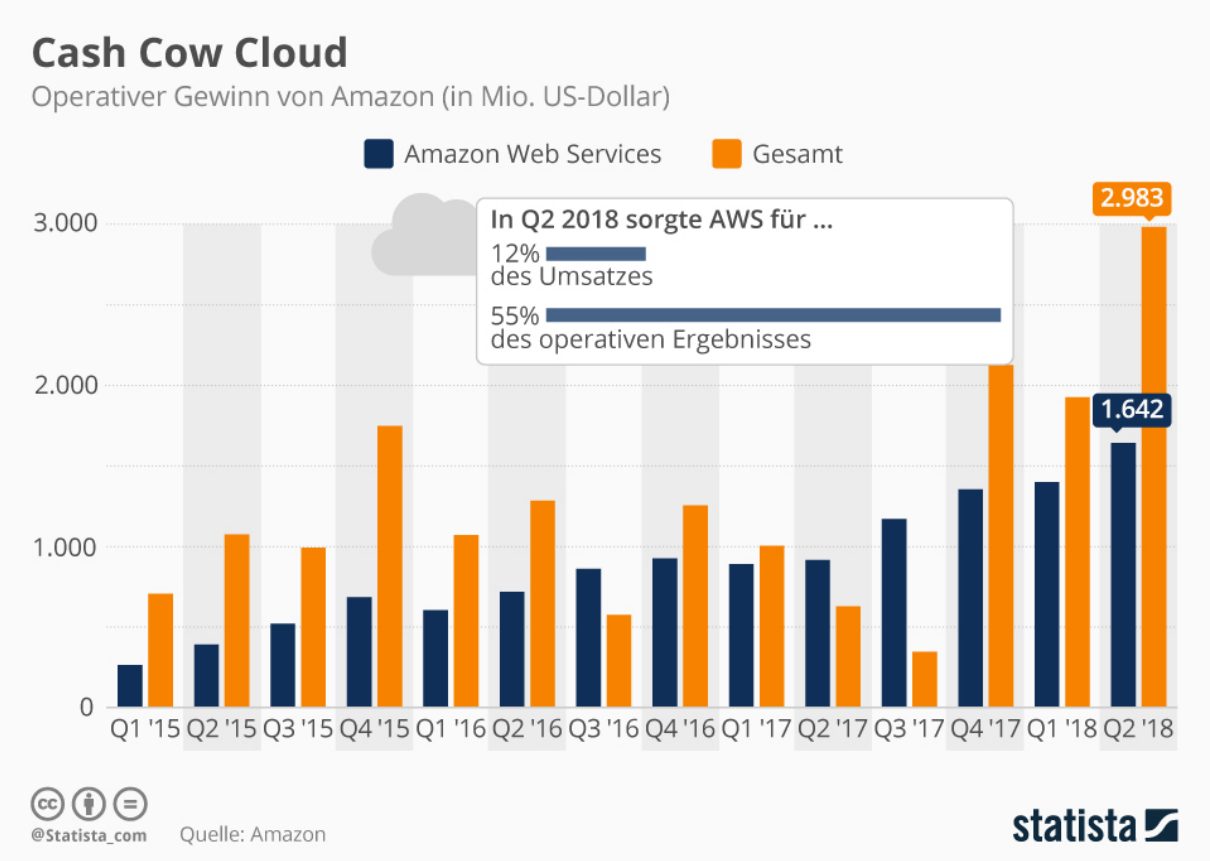

Abbildung 2: Operativer Gewinn von Amazon [Bra18]

Die neueste Stufe in der Entwicklung des Cloud Computings ist das Serverless Computing.

> *„Natürlich benötigen wir nach wie vor Server - wir kommen bloß nicht mehr mit ihnen in Berührung, weder physisch (Hardware) noch logisch (virtualisierte Serverinstanzen). [Kö17, S. 15]“*

Obwohl der Name einen serverlosen Betrieb suggeriert, müssen selbstverständlich Server bereitgestellt werden. Dies übernimmt, wie bei anderen Cloud Technologien auch üblich, der Plattform Anbieter. Allerdings muss sich nicht mehr um die Verwaltung der Server gekümmert werden [Kö17, S. 15]. Dies führt dazu, dass Serverless als sehr nützliches und mächtiges Werkzeug dienen kann. Die Möglichkeiten sind dabei vom Prototyping und kleineren Hilfsaufgaben bis hin zur Entwicklung kompletter Anwendungen. [Kö17, S. 11]

Da der Bereich Serverless erst vor wenigen Jahren entstanden ist und sich immer noch weiterentwickelt, gibt es bisher keine allzu große Verbreitung von Standards. Das heißt, es gibt wenige *Best Practice* Anleitungen und auch unterstützende Tools sind oftmals noch unausgereift. Somit ist es schwer für Unternehmen abzuwägen, ob es sinnvoll ist auf Serverless umzustellen bzw. Neuentwicklungen in diesem Umfeld zu erstellen.

Das Ziel der Arbeit ist es daher, die Unterschiede in der Entwicklung einer Serverless und einer klassischen Webanwendung anhand festgelegter Kriterien zu vergleichen, sodass hieraus sinnvolle Einsatzmöglichkeiten für Serverless Webanwendungen abgeleitet werden können, um die Vorteile des Serverless Computings ideal auszunutzen.

Um das Gebiet *Cloud Computing* besser kennenzulernen, wird zum Beginn der Arbeit die historische Entwicklung, sowie Grundlagen des Themenfelds beschrieben (Kapitel 2.1). Ebenso werden Eigenschaften der Serverless Architektur erläutert (Kapitel 2.2 und Kapitel 2.3).

Im nächsten Schritt wird die prototypische Webanwendung in zweifacher Ausführung implementiert. Einmal als klassische Variante mit Hilfe des Spring Frameworks im Backend und zum anderen als Serverless Webapplikation. Hierzu werden zuerst die Kriterien und das Vorgehen zum Vergleich der beiden Anwendungen festgelegt (Kapitel 3.1). Die Entwicklung des Frontends, das von beiden Anwendungen genutzt wird, erfolgt unter der Verwendung von Polymer (Kapitel 3.3). Nachdem die klassische Implementierung beschrieben wurde (Kapitel 3.4), wird die Serverless Umsetzung tiefer gehend betrachtet, um dem Leser einen umfangreichen Einblick in die neue Technologie zu ermöglichen (Kapitel 3.5). Abschließend werden die beiden Webanwendungen gegenüber gestellt und durch vorher erarbeitete Kriterien Unterschiede in der Entwicklung herausgearbeitet (Kapitel 3.6).

Zuletzt werden anhand der Unterschiede Vor- und Nachteile einer Serverless Infrastruktur dargelegt, um letztendlich sinnvolle Einsatzmöglichkeiten für Serverless Webanwendungen benennen zu können (Kapitel 4).

2 Grundlagen der Serverless Architektur

2.1 Historische Entwicklung des Cloud Computings

Die Evolution des Cloud Computings begann in den sechziger Jahren. Es wurde das Konzept entwickelt, Rechenleistung über das Internet anzubieten. John McCarthy beschrieb das im Jahr 1961 folgendermaßen: [Gar99, S. 1]

> *„If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as a telephone system is a public utility. [...] The computer utility could become the basis of a new and important industry.“*

McCarthy hatte also die Vision, Computerkapazitäten als öffentliche Dienstleistung, wie beispielsweise das Telefon, anzubieten. Der Nutzer soll sich dabei nicht mehr selber um

die Bereitstellung der Rechenleistung kümmern müssen, sondern die Ressourcen sind über das Internet verfügbar. Es wird je nach Nutzung verbrauchsorientiert abgerechnet.

Vor allen Dingen durch das Wachstum des Internets in den 1990er Jahren bekam die Entwicklung von Webtechnologien einen Schub. Anfangs übernahmen traditionelle Rechenzentren das Hosting der Webseiten und Anwendungen. Hiermit einhergehend war allerdings eine limitierte Elastizität der Systeme. Skalierbarkeit konnte beispielsweise nur durch Hinzufügen neuer Hardware erlangt werden. Neben der Hardware und dem Application Stack, war der Entwickler außerdem für das Betriebssystem, die Daten, den Speicher und die Vernetzung seiner Applikation verantwortlich. [Inc18, S. 6]

Durch das Voranschreiten der Cloud-Technologien konnten immer mehr Teile des Entwicklungsprozesses abstrahiert werden, sodass sich der Verantwortlichkeitsbereich und auch das Anforderungsprofil an den Entwickler verschoben hat (siehe Abb. 3).

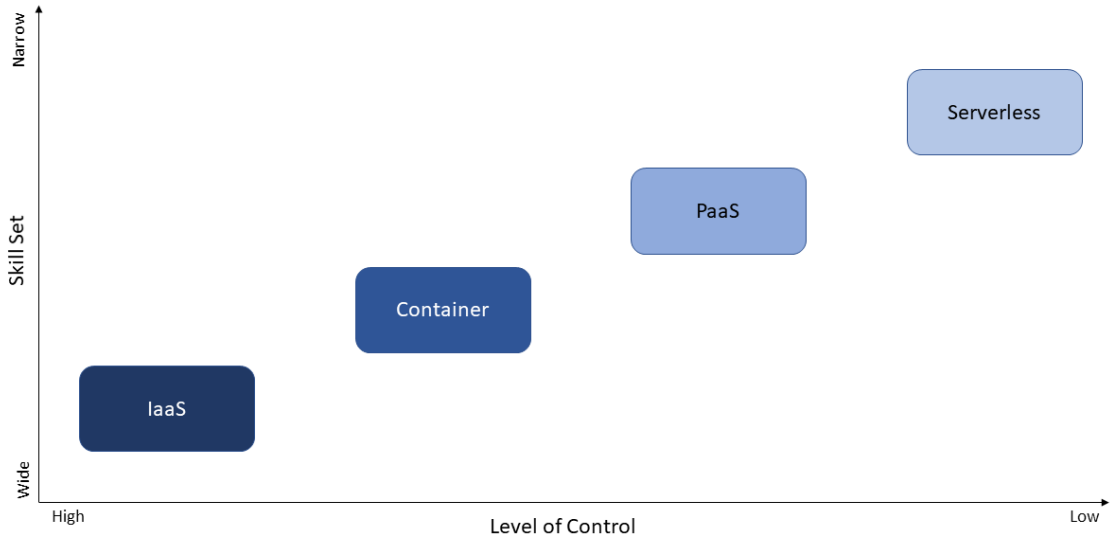

Abbildung 3: Zusammenhang Kenntnisstand und Kontrolllevel [Bü17]

Im ersten Schritt wurden hierzu häufig Infrastructure as a Service (IaaS) Plattformen verwendet. Diese wurden für eine breite Masse verfügbar, als die ersten Anbieter in den frühen 2000er Jahren damit anfingen Software und Infrastruktur für Kunden bereitzustellen. Amazon beispielsweise veröffentlichte seine eigene Infrastruktur, die darauf ausgelegt war die Anforderungen an Skalierbarkeit, Verfügbarkeit und Performance abzudecken, und machte sie so 2006 als AWS für seine Kunden verfügbar. [RPMP17]

Ein weiterer Schritt in der Abstrahierung konnte durch die Einführung von Platform as a Service (PaaS) vollzogen werden. PaaS sorgt dafür, dass der Entwickler sich nur noch um die Anwendung und die Daten kümmern muss. Damit einhergehend kann eine hohe Skalierbarkeit und Verfügbarkeit der Anwendung erreicht werden.

Auf der Virtualisierungsebene aufsetzend, kamen schließlich noch Container hinzu. Diese sorgen beispielsweise für einen geringeren Ressourcenverbrauch und schnellere Bootzeiten. Bei PaaS werden Container zur Verwaltung und Orchestrierung der Anwendung verwendet. Es wird also auf die Kapselung einzelner wiederverwendbarer Funktionalitäten als Service geachtet. Dieses Schema erinnert stark an Microservices. Die genauere Abgrenzung zu Microservices wird im weiteren Verlauf der Arbeit behandelt. [Inc18, S. 6-7]

Als bisher letzter Schritt dieser Evolution entstand das Serverless Computing. Dabei werden zustandslose Funktionen in kurzlebigen Containern ausgeführt. Dies führt dazu, dass der Entwickler letztendlich nur noch für den Anwendungscode zuständig ist. Er unterteilt die Logik anhand des Function as a Service (FaaS) Paradigmas in kleine, für sich selbstständige Funktionen. [Inc18, S. 7]

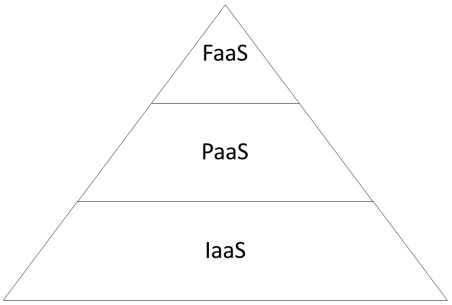

Abbildung 4: Hierarchie der Cloud Services [Kö17, S. 28]

2014 tat sich Amazon dann als Vorreiter für das Serverless Computing hervor und brachte AWS Lambda auf den Markt. Diese Plattform ermöglicht dem Nutzer Serverless Anwendungen zu betreiben. 2016 zogen Microsoft mit *Azure Function* und Google mit *Cloud Function* nach. [RPMP17]

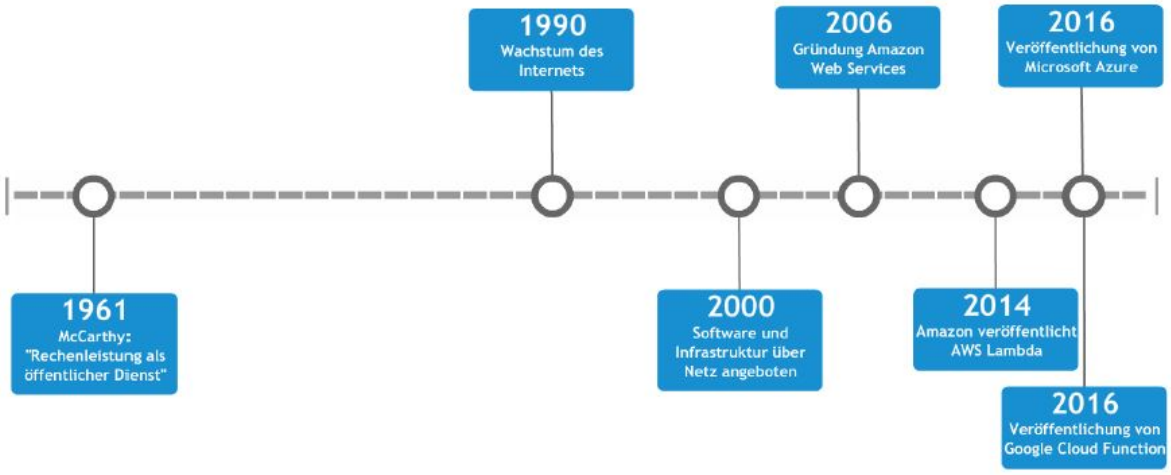

Abbildung 5: Historische Entwicklung des Cloud Computings

2.1.1 Grundlagen des Cloud Computings

„Run code, not Server [Rö17a]“

Dies kann als eine der Leitlinien des Cloud Computings angesehen werden. Die Idee ist es den Entwickler zu entlasten, sodass die Anwendungsentwicklung mehr in den Fokus gerückt wird. Das National Institute of Standards and Technology (NIST) definiert Cloud Computing folgendermaßen: [MG11]

> *„Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.“*

Der Anwender kann also über das Internet selbstständig Ressourcen anfordern, ohne dass beim Anbieter hierfür ein Mitarbeiter eingesetzt werden muss. Der Kunde hat dabei allerdings keinen Einfluss auf die Zuordnung der Kapazitäten. Freie Ressourcen werden auch nicht für einen bestimmten Kunden vorgehalten. Dadurch kann der Anbieter schnell auf einen geänderten Bedarf reagieren und für den Anwender scheint es, als ob er unbegrenzte Kapazitäten zur Verfügung hat.

Zur Verwendung dieses Angebots stehen dem Nutzer verschieden Out-of-the-Box Dienste in unterschiedlichen Abstufungen zur Verfügung (siehe Abb. 6). Dies wären zum einen das IaaS Modell, bei dem einzelne Infrastrukturkomponenten wie Speicher, Netzwerkleistungen und Hardware durch virtuelle Maschinen verwaltet werden. Skalierung kann so zum Beispiel einfach durch allokieren weiterer Ressourcen in der virtuellen Maschinen erreicht

werden. [Sti17, S. 3]

Zum anderen das PaaS Modell. Dabei wird dem Entwickler der Softwarestack bereitgestellt und ihm werden Aufgaben wie Monitoring, Skalierung, Load Balancing und Server Restarts abgenommen. Ein typisches Beispiel hierfür ist Heroku, bei dem der Nutzer seine Anwendung bereitstellen und konfigurieren kann. [Sti17, S. 3]

Ebenfalls zu den Diensten gehört Backend as a Service (BaaS). Dieses Modell bietet modulare Services, die bereits eine standardisierte Geschäftslogik mitbringen. Es muss lediglich anwendungsspezifische Logik vom Entwickler implementiert werden. Die einzelnen Services können dann zu einer komplexen Softwareanwendung zusammengefügt werden. [Rö17a]

Die größte Abstraktion bietet SaaS. Hierbei wird dem Kunden eine konkrete Software zur Verfügung gestellt, sodass dieser nur noch als Anwender agiert. Beispiele dafür sind Dropbox und GitHub. [Sti17, S. 3]

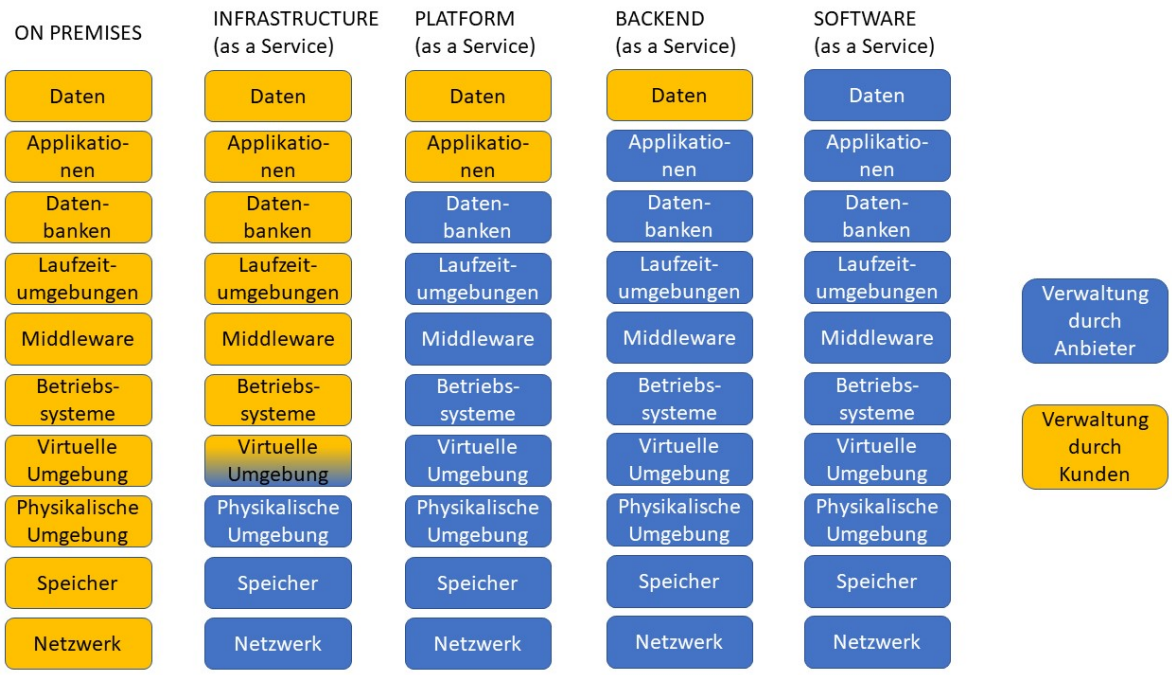

Abbildung 6: Verantwortlichkeiten der Organisation nach [Rö17a]

Oftmals nutzen PaaS Anbieter ein IaaS Angebot und zahlen dafür. Nach dem gleichen Prinzip bauen SaaS Anbieter oft auf einem PaaS Angebot auf. So betreibt Heroku zum Beispiel seine Services auf Amazon Cloud Plattformen [Her18]. Ebenso ist es möglich eine Infrastruktur durch einen Mix der verschiedenen Modelle zusammenzustellen.

Letztendlich ist alles darauf ausgelegt, dass sich im Entwicklungs- und operationalen Auf-

wand so viel wie möglich einsparen lässt. Diese Weiterentwicklung wurde zum Beispiel in der Automobilindustrie bereits vollzogen. Dabei war es das Ziel die Fertigungstiefe, das heißt die Anzahl der eigenständig erbrachten Teilleistungen, zu reduzieren [Dja02, S. 8]. Nun findet diese Entwicklung auch Einzug in den Informatiksektor.

Ebenfalls von Bedeutung ist, dass die Anwendung automatisch skaliert und sich so an eine wechselnde Beanspruchung anpassen kann. Außerdem werden hohe Initialkosten für eine entsprechende Serverlandschaft bei einem Entwicklungsprojekt für den Nutzer vermieden und auch die Betriebskosten können gesenkt werden.Voraussetzung hierfür ist das Pay-per-use-Modell. Der Kunde zahlt aufwandsbasiert. Das heißt, er zahlt nur für die verbrauchte Rechenzeit. Leerlaufzeiten werden nicht mit einberechnet. [Rö17a]

Da Cloud-Dienste dem Entwickler viele Aufgaben abnehmen und erleichtern, sodass sich die Verantwortlichkeiten für den Entwickler verschieben, ist dieser nun beispielsweise nicht mehr für den Betrieb, sowie die Bereitstellung der Serverinfrastruktur zuständig. Dies führt allerdings auch dazu, dass ein gewisses Maß an Kontrolle und Entscheidungsfreiheit verloren geht.

2.1.2 Abgrenzung zu PaaS

Prinzipiell klingen PaaS und Serverless Computing aufgrund des übereinstimmenden Abstrahierungsgrades sehr ähnlich. Der Entwickler muss sich nicht mehr direkt mit der Hardware auseinandersetzen. Dies übernimmt der Cloud-Service in Form einer Blackbox. Es muss nur der Code hochgeladen werden.

Jedoch gibt es auch einige grundlegenden Unterschiede. So muss der Entwickler bei einer PaaS Anwendung durch Interaktion mit der API oder Oberfläche des Anbieters eigenständig für Skalierbarkeit und Ausfallsicherheit sorgen. Bei der Serverless Infrastruktur übernimmt das Kapazitätsmanagement der Cloud-Service (siehe Abb. 7). Es gibt zwar auch PaaS Plattformen, die bereits Funktionen für das Konfigurationsmanagement bereitstellen, oft sind diese jedoch anbieterspezifisch, wodurch der Programmierer auf weitere externe Tools zurückgreifen muss. [Bü17]

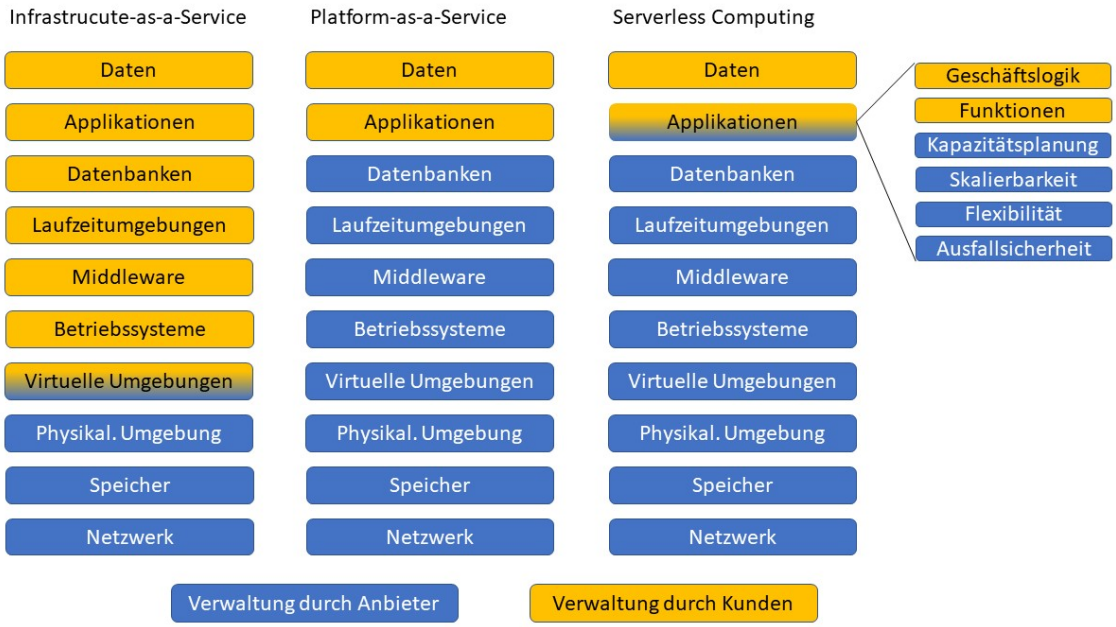

Abbildung 7: Aufgabenverteilung: IaaS vs. PaaS vs. Serverless [Bü17]

Ein weiterer Unterschied ist, dass PaaS für lange Laufzeiten konstruiert ist. Das heißt, die PaaS Anwendung läuft immer. Bei Serverless hingegen wird eine Funktion als Reaktion auf ein Event gestartet und wieder beendet. Wenn kein Request eintrifft, werden auch keine Ressourcen mehr verbraucht. [Ash17]

Aktuell wird PaaS hauptsächlich wegen der sehr guten Toolunterstützung genutzt. Hier hat Serverless Computing den Nachteil, dass es durch den geringen Zeitraum seit der Entstehung noch nicht so ausgereift ist. [Rob18]

Final stechen als Schlüsselunterschiede zwei Punkte heraus. Dies ist zum einen, wie oben bereits erwähnt, die Skalierbarkeit. Sie ist zwar auch bei PaaS Applikationen erreichbar, allerdings bei weitem nicht so hochwertig und komfortabel. Zum anderen die Kosteneffizienz, da der Nutzer nicht mehr für Leerlaufzeiten aufkommen muss. Adrian Cockcroft von AWS bringt das folgendermaßen auf den Punkt. [Rob18]

> *„If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless.“*

2.1.3 Abgrenzung zu Microservices

Bei der Entwicklung einer Anwendung kann diese in verschieden große Komponenten aufgeteilt werden. Das genaue Vorgehen wird dazu im Voraus festgelegt. Entscheidet sich das Entwicklerteam für eine große Einheit, wird von einer Monolithischen Architektur

gesprochen. Hierbei wird die komplette Applikation als ein Paket ausgeliefert. Dies hat den Nachteil, dass bei einem Problem die ganze Anwendung ausgetauscht werden muss. Auch die Einführung neuer Funktionalitäten braucht eine lange Planungsphase. [Inc18, S. 9]

Auf der anderen Seite steht die Microservice Architektur. Die Anwendung wird in kleine Services, die für sich eigenständige Funktionalitäten abbilden, aufgeteilt. Teams können nun unabhängig voneinander an einzelnen Services arbeiten. Auch der Austausch oder die Erweiterung einzelner Module erfolgt wesentlich reibungsloser. Dabei ist jedoch zu beachten, dass die Anonymität zwischen den Modulen gewahrt wird. Ansonsten kann auch bei Microservices die Einfachheit verloren gehen. Durch die Aufteilung in verschiedene Komponenten erreichen Microservice Anwendungen eine hohe Skalierbarkeit. [Bac18]

Das Konzept, die Funktionalität in kleine Einheiten aufzuteilen, findet sich auch im Serverless Computing wieder. Im Gegensatz zur Microservices ist Serverless viel feingranularer. Bei Microservices wird oft das Domain-Driven Design herangezogen, um eine komplexe Domäne in sogenannte *Bounded Contexts* zu unterteilen. Diese Kontextgrenzen werden dann genutzt, damit die fachlichen Aspekte in verschiedene individuellen Services aufgeteilt werden können. [FL14]

In diesem Zusammenhang wird auch oft von serviceorientierter Architektur gesprochen. Eine Serverless Funktion hingegen stellt nicht einen kompletten Service dar, sondern eine einzelne Funktionalität. Eine Funktion verkörpert einen Event Handler. Daher handelt es sich hierbei um eine ereignisgesteuerte Architektur. [Tur18]

Ebenso ist es bei Serverless Anwendungen nicht notwendig die unterliegende Infrastruktur zu verwalten. Es muss lediglich die Geschäftslogik als Funktion implementiert werden. Weitere Komponenten wie beispielsweise ein Controller müssen nicht selbstständig entwickelt werden. Außerdem bietet der Cloud-Provider bereits eine automatische Skalierung als Reaktion auf sich ändernde Last an. Also auch hier werden dem Entwickler Aufgaben abgenommen. [Inc18, S. 9]

> *„The focus of application development changed from being infrastructure-centric to being code-centric. [Inc18, S. 10]“*

Im Vergleich zu Microservices rückt bei der Implementierung von Serverless Anwendungen die Funktionalität der Anwendung in den Fokus und es muss keine Rücksicht mehr auf die Infrastruktur genommen werden.

2.2 Eigenschaften von Function as a Service (FaaS)

Wenn von Serverless Computing gesprochen wird, ist oftmals auch von FaaS die Rede. Der Serverless Provider stellt hierfür eine Plattform zur Verfügung. Die Infrastruktur des Anbieters kann dabei als Backend as a Service (BaaS) gesehen werden. Eine Serverless Architektur stellt also eine Kombination der beiden Konzepte dar. [Rob18]

> *„FaaS entails running back-end code without the task of developing and deploying your own server applications and server systems. [Sti17, S. 3]“*

Der Fokus kann somit vollkommen auf die Geschäftslogik gelegt werden. Jede Funktionalität wird dabei in einer eigenen Function umgesetzt [Ash17]. Die Programmiersprache, in der die Anforderungen implementiert werden, hängt vom Anbieter der Plattform ab. Die geläufigen Sprachen, wie zum Beispiel Java, Python oder Javascript, werden allerdings von allen großen Providern unterstützt [Tiw16]. Jede Function stellt eine unabhängige und wiederverwendbare Einheit dar. Durch sogenannte Events kann eine Function angesprochen und aufgerufen werden. Hinter einem Event kann sich möglicherweise ein File-Upload oder ein HTTP-Request verbergen. Die dabei verwendeten Komponenten, wie zum Beispiel ein Datenbankservice, werden Ressourcen genannt. [RPMP17]

Die Functions sind alle zustandslos. So lassen sich in kürzester Zeit viele Kopien derselben Funktionalität starten. Dadurch kann eine hohe Skalierbarkeit erreicht werden. Alle benötigten Zusammenhänge müssen extern gespeichert und verwaltet werden, da sich prinzipiell der Zustand jeder Instanz vom Stand des vorherigen Aufrufs unterscheiden kann. Auch wenn es sich um dieselbe Function handelt. [Bü17]

Der Aufruf einer Function kann entweder synchron über das Request-/Response-Modell oder asynchron über Events erfolgen. Da der Code in kurzlebigen Containern ausgeführt wird, werden asynchrone Aufrufe bevorzugt. Dadurch kann sichergestellt werden, dass die Function bei verschachtelten Aufrufen nicht zu lange läuft. Bedingt durch die automatische Skalierung eignet sich FaaS somit besonders gut für Methoden mit einem schwankendem Lastverhalten. [Rö17a]

Auch über die Verfügbarkeit muss sich der Nutzer keine Gedanken mehr machen, da der Dienstleister für die komplette Laufzeitumgebung verantwortlich ist: [Kö17, S. 28]

> *„Eine fehlerhafte Konfiguration hinsichtlich Über- oder Unterprovisionierung von (Rechen-, Speicher-, Netzwerk etc.) Kapazitäten können somit nicht passieren. [Kö17, S. 29]“*

Das heißt, dass alle Ressourcen mit bestmöglicher Effizienz genutzt werden. Die Architektur einer beispielhaften FaaS Anwendung könnte somit folgendermaßen ausschauen (siehe

Abb. 8). Hierbei nimmt das API Gateway die Anfragen des Clients entgegen und ruft die dazugehörigen Functions auf, die jeweils an einen eigenen Speicher angebunden sind. Neben der Möglichkeit HTTP-Requests über das API Gateway an die einzelnen Functions weiterzuleiten, kann auch das Hochladen einer Datei in den sogenannten *Blob Store* eine Function aufwecken. Ein Anwendungsfall in der hier aufgezeigten Beispiel-Anwendung könnte nun wie folgt ablaufen:
Ein Nutzer lädt in der Anwendung ein neues Profilbild hoch. Das API Gateway leitet den Request an die *Upload Function* weiter (siehe Abb. 8 Punkt 1). Diese speichert das Bild im *Blob Store*, wodurch der Vorgang abgeschlossen sein könnte (siehe Abb. 8 Punkt 2). Jedoch wird durch das Speichern in der Datenbank ein weiteres Event ausgelöst, das die *Activity Function* auslöst (siehe Abb. 8 Punkt 3). Diese Function könnte nun zum Beispiel genutzt werden, um das neue Profilbild zu bearbeiten, sich die Bearbeitung in der zugehörigen Datenbank zu merken und es an den Browser zurück zu schicken (siehe Abb. 8 Punkt 4). Der Vorteil dieses Vorgehens ist es, dass der Nutzer nach dem Hochladen des Bildes eine Antwort erhält und das System nicht bis zum Abschluss der Bearbeitung blockiert ist. Nebst der Möglichkeit die *Activity Function* asynchron über ein Event aufzurufen, kann sie auch über das API-Gateway erreicht werden (siehe Abb. 8 Punkt 5). So könnte ein bereits bearbeitetes Bild noch einmal angepasst werden.

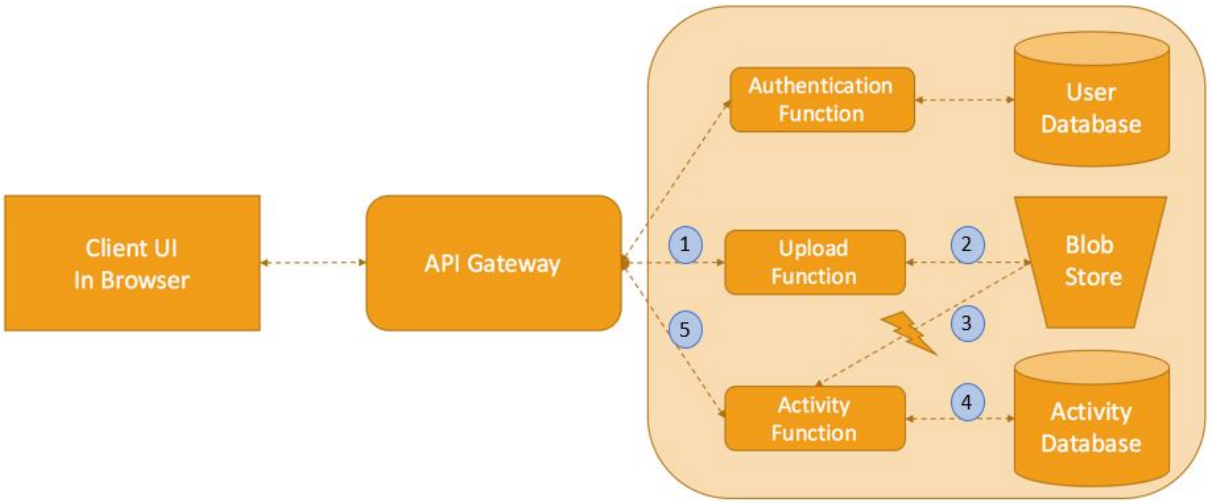

Abbildung 8: FaaS Beispiel Anwendung [Tiw16]

2.3 Allgemeine Pattern für Serverless Umsetzungen

Sogenannte Pattern dienen dazu, wiederkehrende Probleme bestmöglich und einheitlich zu lösen. Sie geben ein Muster vor, dass zur Lösung eines spezifischen Problems herangezogen werden kann. Als Richtschnur zur Bearbeitung von Herausforderungen im Serverless

Umfeld kann beispielsweise das *Serverless Computing Manifest* verwendet werden.

2.3.1 Serverless Computing Manifest

Viele Grundsätze im Softwaresektor werden durch Manifeste festgehalten. Eines der bekanntesten Manifeste ist das *Agile Manifest*, das die agile Softwareentwicklung hervorbrachte. So ist es auch kaum verwunderlich, dass es im Bereich des Serverless Computing ebenfalls ein Manifest gibt. Das *Serverless Computing Manifesto*. [Kö17, S. 19]

Die Herkunft des Manifests kann nicht eindeutig geklärt werden. Niko Köbler äußert sich hierzu in seinem Buch *Serverless Computing in der AWS Cloud* folgendermaßen: [Kö17, S. 20]

> *„Allerdings findet sich hierfür kein dedizierter und gesicherter Ursprung, das Manifest wird aber auf mehreren Webseiten und Konferenzen einheitlich zitiert. Meine Recherche ergab eine erstmalige Nennung des Manifests und Aufzählung der Inhalte im April 2016 auf dem AWS Summit in Chicago in einer Präsentation namens "Getting Started with AWS Lambda and the Serverless Cloud"von Dr. Tim Wagner, General Manager für AWS Lambda and Amazon API Gateway.“*

Das Manifest besteht aus acht Leitsätzen, die nun genauer betrachtet werden. Einige Prinzipien wurden bereits in vorherigen Kapiteln angeschnitten oder erläutert.

Functions are the unit of deployment and scaling. Functions stellen den Kern einer Serverless Anwendung dar. Eine Function ist nur für eine spezielle Aufgabe verantwortlich und auch die Skalierung erfolgt bei Serverless Applikationen funktionsbasiert. [Kö17, S. 20]

No machines, VMs, or containers visible in the programming model. Für den Nutzer der Plattform sind die Bestandteile der Serverinfrastruktur nicht sichtbar. Er kann mit der Implementierung nicht in die Virtualisierung oder Containerisierung eingreifen. Die Interaktion mit den Services des Providers erfolgt über bereitgestellte Software Development Kits (SDKs). [Kö17, S. 21]

Permanent storage lives elsewhere. Serverless Functions sind zustandslos. Das heißt, dass die selbe Function beim mehrmaligen Ausführen in verschiedenen Umgebungen laufen kann, sodass der Nutzer nicht mehr auf vorherige Daten zurückgreifen kann. Zukünftig benötigte Daten müssen daher immer über einen anderen Dienst persistiert werden. [Kö17, S. 21]

Scale per request. Users cannot over- or under-provision capacity. Die Skalierung erfolgt völlig automatisch durch den Serviceanbieter. Dieser sorgt dafür, dass die Func-

tions parallel und unabhängig voneinander ausgeführt werden können. Der Kunde kommt nicht mit diesem Aufgabenfeld in Berührung. Hierzu cachen einige Anbieter die Containerumgebung, falls sie merken, dass eine Funktion in einem kurzen Zeitraum öfters aufgerufen wird, um eine bessere Performanz zu erreichen. Hierauf kann sich der User jedoch nicht verlassen. [Kö17, S. 21]

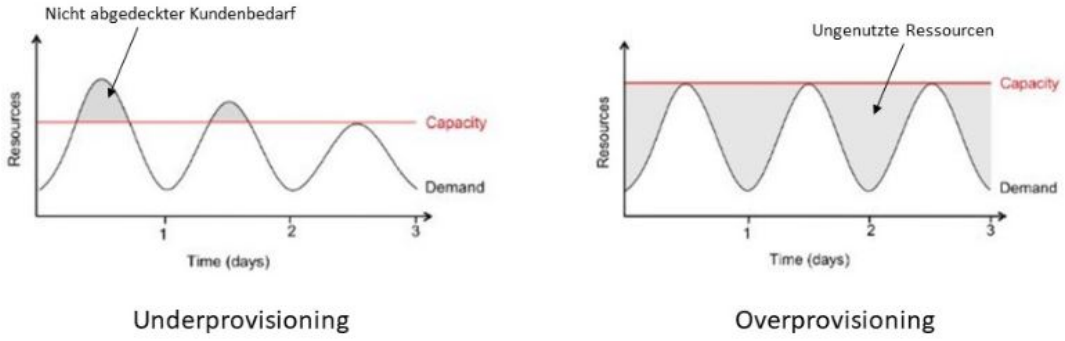

Abbildung 9: Under- und Overprovisioning [A+09, S. 11]

Im linken Diagramm ist die Auswirkung von *Underprovisioning* zu sehen. Hierbei kann es vorkommen, dass der Bedarf die gegebene Kapazität übersteigt und somit nicht mehr genug Ressourcen für alle Kunden bereitgestellt werden können. Dies führt zu unzufriedenen Nutzern und kostet schlussendlich dem Unternehmen Kunden. Bei *Overprovisioning* auf der rechten Seite hingegen ist die Kapazität gleich dem maximalen Bedarf. Hierdurch werden jedoch zu einem Großteil der Zeit mehr Ressourcen bereitgestellt als eigentlich benötigt. Es entstehen unnötige Ausgaben.

Never pay for idle(no cold servers/containers or their cost). Der Kunde zahlt nur für die tatsächlich genutzte Rechenzeit. Die Bereitstellung der Ressourcen fällt dabei nicht ins Gewicht. Um dies dem Nutzer zu ermöglichen, sollten auf Seiten der Anbieter alle Betriebsmittel optimal ausgenutzt werden. So werden diese keinem bestimmten Kunden zugeordnet, sondern stehen für viele Nutzer bereit. Je nach Bedarf können dem Anwender dynamisch benötigte Leistungen aus einem großen Pool zugeteilt werden. Sobald die Function durchgelaufen ist, werden die Ressourcen wieder freigegeben und können von jedem anderen verwendet werden. [Kö17, S. 22]

Implicitly fault-tolerant because functions can run anywhere. Da für den Nutzer nicht ersichtlich ist, wo seine Functions beim Provider ausgeführt werden, darf in den Implementierungen auch keine Abhängigkeit diesbezüglich bestehen. Dies führt zu einer impliziten Fehlertoleranz, da der Betreiber keinen Einschränkungen unterliegt,

in welchen Bereichen seiner Infrastruktur er bestimmte Functions ausführen darf. [Kö17, S. 22]

BYOC - Bring Your Own Code. Eine Function muss alle benötigten Abhängigkeiten bereits enthalten. Der Anbieter stellt lediglich eine Ablaufumgebung zur Verfügung, sodass zur Laufzeit keine weiteren Bibliotheken nachgeladen werden können. [Kö17, S. 23]

Metrics and logging are a universal right. Da für den Nutzer die Ausführung serverloser Services transparent abläuft und auch keinerlei Zustände in der Serverless Anwendung gespeichert werden, ist es für ihn nicht möglich Informationen über die Ausführung zu erhalten. Damit der User trotzdem Details seiner Anwendung zur Fehlersuche oder Analyse erhält, muss der Serviceprovider diese Möglichkeiten bereitstellen. So bietet er beispielsweise Logs zu einzelnen Funktionsaufrufen an. Des Weiteren werden Metriken, wie zum Beispiel Ausführungsdauer, CPU-Verwendung und Speicherallokation, zur Analyse der Applikation zur Verfügung gestellt. Das Loggen der Funktionsinhalte muss durch die Function selbst übernommen werden. [Kö17, S. 23]

Anhand des Manifestes ist es schon zu erkennen, dass sich Serverless Computing nicht einfach in einem Pattern beschreiben lässt. Es spielen viele Muster zusammen. So enthält das Manifest beispielsweise neben wichtigen Prinzipien auch Pattern, die bei der Umsetzung von Serverless Anwendungen angewendet werden können. Neben dem *Serverless Computing Manifest* gibt es noch weitere Richtlinien, die bei der Umsetzung von Serverless Anwendungen in Betracht gezogen werden können beziehungsweise sich in einigen Punkten des Manifestes widerspiegeln. Einige werden nun im Folgenden genauer betrachtet, um bereits bekannte Pattern besser in den Serverless Kontext einordnen zu können.

2.3.2 Schnittstellen zu anderen Architekturen

Hierzu gehört zum Beispiel das *Microservice Pattern*. Es harmoniert hervorragend mit dem Pattern *Functions are the unit of deployment and scaling*. Jede Funktionalität wird in einer eigenen Function isoliert. Dies führt dazu, dass verschiedene Komponenten einzeln und unabhängig voneinander ausgebracht bzw. bearbeitet werden können, ohne sich gegenseitig zu beeinflussen. Außerdem wird es einfach die Anwendung zu debuggen, da jede Function nur ein bestimmtes Event bearbeitet und somit die Aufrufe größtenteils vorhersehbar sind. Nachteilig hieran ist jedoch die Masse an Functions, die verwaltet werden müssen. [Hef16]

Der Aufruf der somit erstellten Functions führt zum nächsten Muster für Serverless Um-

setzungen. Die ereignisgesteuerte Architektur sorgt dafür, dass die Functions durch Events aufgerufen werden können. Dieses Architekturmuster wird natürlich nicht nur bei Serverless Anwendungen verwendet, sondern kann auch in anderen Feldern zum Einsatz kommen. Es handelt sich bei Serverless Computing also lediglich um einen kleinen Bestandteil des *Event-driven computings* (siehe Abb. 10). [Boy17]

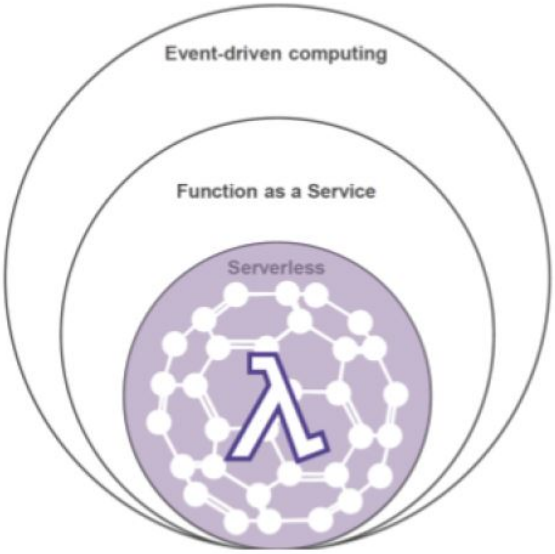

Abbildung 10: Zusammenhang zwischen Event-driven Computing, FaaS und Serverless [FIMS17, S. 5]

Neben asynchronen Events können Serverless Functions auch durch synchrone Nachrichten angesprochen werden. Hierzu kann als Einstiegspunkt einer Function ein HTTP-Endpunkt dienen. Der Aufruf folgt dann dem *Request-Response Pattern*, das als Basismethode zur Kommunikation zwischen zwei Systemen angesehen werden kann. Die anfragende Stelle startet mit einem Request die Kommunikation und wartet auf eine Antwort. Diese Anfrage ist der Aufruf einer Function. Der Provider auf der anderen Seite repräsentiert die Function und wartet auf den Request. Nach der Abarbeitung sendet der Service seine Antwort an den Anfrager zurück. [Swa18]

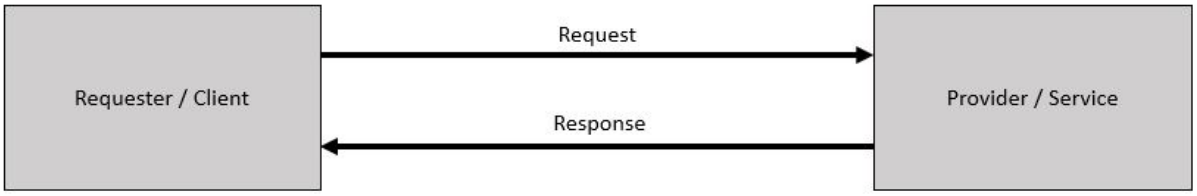

Abbildung 11: Request Response Pattern [Swa18]

3 Vergleich zweier prototypischer Anwendungen

3.1 Vorgehensweise beim Vergleich der beiden Anwendungen

Zum Vergleich der beiden Anwendungen werden einige Kriterien abgearbeitet, die dabei helfen eine Aussage über die Qualität der jeweiligen Applikation zu treffen. Diese werden nun im Folgenden genauer erläutert:

Implementierungsaufwand Es wird auf den zeitlichen Aufwand, sowie auf die Codekomplexität geachtet. Das heißt, es wird untersucht, mit wie viel Einsatz einzelne Anwendungsfälle umgesetzt werden können und wie viel Overhead bei der Umsetzung möglicherweise entsteht.

Frameworkunterstützung Dabei wird analysiert, inwieweit die Entwicklung durch Frameworks unterstützt werden kann. Dies gilt nicht nur für die Abbildung der Funktionalitäten, sondern auch für andere anfallende Aufgaben im Entwicklungsprozess, wie zum Beispiel Testen und Deployment.

Deployment Beim Deploymentprozess sollen Änderungen an der Anwendung möglichst schnell zur produktiven Applikation hinzugefügt werden können, damit sie dem Kunden zeitnah zur Verfügung stehen. An dieser Stelle sind eine angemessene Toolunterstützung, sowie die Komplexität der Prozesse ein großer Faktor. Optimal wäre in diesem Punkt eine automatische Softwareauslieferung.

Testbarkeit Hier ist zum einen ebenfalls der Implementierungsaufwand relevant und zum anderen sollte die Durchführung der Tests den Entwicklungsprozess nicht unverhältnismäßig lange aufhalten. Es ist dann auch eine effektive Einbindung der Tests in den Deploymentprozess gefragt. Im Speziellen werden im Rahmen der beiden Anwendungen Komponenten- und Integrationstests betrachtet.

Erweiterbarkeit Das Hinzufügen neuer Funktionalitäten oder Komponenten wird dabei im Besonderen überprüft. Damit einhergehend ist auch die Wiederverwendbarkeit einzelner Module. Dies bedeutet, dass beleuchtet wird, ob einzelne Teile losgelöst vom restlichen System in anderen Projekten erneut einsetzbar sind.

Performance Das Augenmerk liegt hierbei auf der Messung von Antwortzeiten einzelner Requests, sowie der Reaktion des Systems auf große Last.

Sicherheit An dieser Stelle ist zum Beispiel die Unterstützung zum Anlegen einer Nutzerverwaltung von Interesse. Außerdem wird betrachtet, inwiefern auf die Anwendung verschlüsselt zugegriffen werden kann.

Die Bewertung der beiden Anwendungen erfolgt nach der *Microservice Framework Evaluation Method (MFEM)*, die René Zarwel in seiner Bachelorarbeit zur Evaluierung von Frameworks erarbeitet hat. Diese Methode betrachtet ein Framework von drei Seiten: Nutzung, Zukunftssicherheit und Produktqualität. Angewendet auf die Beurteilung der beiden Applikationen verschiebt sich der Fokus hin zur Nutzung. Das heißt, wie gestaltet sich die Umsetzung. [Zar17, S. 22]

Der erste Schritt wurde durch Sammeln der Kriterien und Anforderungen an die Anwendungen auf Seite 17 bereits abgeschlossen.

> *„Damit der Fokus in späteren Phasen auf den wichtigen Anforderungen liegt, werden anschließend alle mit Prioritäten versehen. [Zar17, S. 28]“*

Hierzu kann eine beliebig gegliederte Rangordnung verwendet werden, wobei in der Arbeit von René Zarwel eine dreistufige Skala als angemessen angesehen wird. Da bei dem hier durchgeführten Vergleich kein Kontext, wie bei der Durchführung in einem Unternehmen besteht, wird auf eine Gewichtung der Anforderungen verzichtet.

Des Weiteren können die vorliegenden Punkte durch tiefergehende Fragen verfeinert und in mehrere Unterpunkte unterteilt werden. Die vollständige Abbildung der Kriterien mit passenden Unterpunkten, angeordnet als Baum, ist in Anhang A zu finden.

Damit die festgelegten Kriterien auf beide Applikationen angewendet werden können, werden nun für jede Kategorie Metriken aufgestellt. Diese können dann genutzt werden, um die verschiedenen Eigenschaften der Anwendungen zu messen und vergleichbar zu machen. So kann beispielsweise eine Ordinalskala dabei helfen, Erkenntnisse in verschiedenen Abstufungen auszudrücken. [Zar17, S. 29]

Im letzten Schritt folgt die Evaluationsphase und anschließend die Aufbereitung der Ergebnisse.

> *„Während der Evaluation wird das Framework auf die Anforderungen mittels der zuvor definierten Metriken untersucht. [Zar17, S. 31]“*

Die Durchführung der Evaluation wird in zwei Phasen unterteilt. Die subjektive und objektive Evaluation. Bei der Ersten erstellt der Softwareentwickler eine prototypische Anwendung und bewertet das Vorgehen anhand von subjektiven Eindrücken aus dem Entwicklungsprozess [Zar17, S. 32]. Diese Variante wird einen Großteil der Arbeit ausmachen. Die objektive Evaluation hingegen nimmt nur einen kleine Anteil der Auswertung ein und bezieht sich auf die Erhebung von neutralen Daten, wie zum Beispiel bei Messungen. [Zar17, S. 36]

Nachdem die Evaluation durchgeführt wurde, können die Ergebnisse ausgewertet werden. Dazu wird für die jeweiligen Kriterien ein Prozentwert berechnet, der aussagt, in wie weit die definierten Anforderungen erfüllt wurden.

> *„Wie stark einzelne Anforderungen in die zugehörige Kategorie einfließen, hängt von der Priorisierung dieser ab. Wurde eine Anforderung mit A bewertet, zählt das Ergebnis zu 100 Prozent. Entsprechend wird der Einfluss bei Priorität B und C auf 50 bzw. 25 Prozent gesenkt. Dies stellt sicher, dass die Nichterfüllung kleiner Anforderungen das Gesamtergebnis nicht zu stark nach unten ziehen. [Zar17, S. 40-41]“*

3.2 Fachliche Beschreibung der Beispiel-Anwendung

Als Anwendungsfall für die Beispiel-Anwendung dient ein Bibliotheksservice. Der Service kann von zwei verschiedenen Anwendergruppen genutzt werden. Das wären auf der einen Seite Mitarbeiter der Bibliothek. Diese können Bücher zum Bestand hinzufügen oder löschen und Buchinformationen aktualisieren. Zur Vereinfachung der Anwendung gibt es zu jedem Buch nur ein Exemplar.

Auf der anderen Seite gibt es den Kunden, dem eine Übersicht aller Bücher zur Verfügung steht. Von diesen Büchern kann der Kunde beliebig viele verfügbare Bücher ausleihen, wobei eine Leihe unbegrenzt ist und somit kein Ablaufdatum besitzt. Seine ausgeliehene Bücher kann er dann auch wieder zurückgeben.

Um nutzerspezifische Informationen in der Anwendung anzeigen zu können und das System vor Fremdzugriffen zu schützen, hat jeder User einen eigenen Account. Mit diesem kann er sich an der Applikation anmelden. Zum Start der Anwendungen stehen jeweils ein Nutzer mit der Rolle „Mitarbeiter“, sowie ein User mit der Rolle „Kunde“ zur Verfügung. Des Weiteren gibt es einen Administrator, der auf alle Funktionalitäten zugreifen kann. Weitere Nutzer können nicht zur Applikation hinzugefügt werden.

Damit der Servicebetreiber sein Angebot an die Nachfrage der Kunden anpassen kann, merkt sich das System bei jeder Ausleihe zusätzlich die Kategorie des ausgeliehenen Buches, sodass anhand der beliebten Bücherkategorien der Bestand sinnvoll erweitert werden kann. Die Nutzerstatistik ist ausschließlich für Mitarbeiter einsehbar.

Dieser Ablauf könnte in einem anderen Anwendungsfall beispielsweise eine Webseite sein, die den Nutzer nach der Auswahl eines Werbebanners nicht nur auf die werbetreibende Seite leitet, sondern sich gleichzeitig den Aufruf der Werbung merkt, um ihn später in Rechnung stellen zu können. [Rob18]

3.3 Implementierung der Benutzeroberfläche

Da die beiden prototypischen Anwendungen sich lediglich in der Umsetzungsart der Anwendungslogik unterscheiden, kann dieselbe Frontendimplementierung für beide Prototypen eingesetzt werden. Dies ist möglich, da beide Anwendungen die gleichen Schnittstellen zur Verfügung stellen und den exakt selben Anwendungsfall abbilden.

Die Benutzeroberfläche wird mit Polymer implementiert. Das ist eine Bibliothek zur Frontendentwicklung, die auf der *Web Components Specification* des World Wide Web Consortiums (W3C) basiert. So kann eine Seitenansicht aus mehreren verschachtelten Komponenten bestehen. Die hohe Wiederverwendbarkeit solcher Komponenten und das damit einhergehende einheitliche Erscheinungsbild sind zwei Vorteile des komponentenbasierten Konzeptes. [Sch16]

> *„Typischerweise besteht eine Polymer Komponente aus drei Teilen: Stylesheets, einem Template und natürlich JavaScript. [WJ16]“*

Alle Bestandteile einer Ansicht befinden sich somit in einer Datei. Neben der Bereitstellung der Daten, wird auf die Eingabe des Users reagiert. Auch die Kommunikation mit dem Server übernimmt jede Komponente für sich. In der Anwendung werden außer den selbst erstellten Komponenten auch weitere Module, die unter *webcomponents.org* registriert sind und von anderen Entwicklern stammen, verwendet.

Properties, die als Attribute innerhalb einer Komponente dienen, können zum Datenaustausch zwischen den verschachtelten Modulen genutzt werden (siehe Listing 1 Z. 3, 11 und 21). Mittels *Two-Way Binding* können die Attribute von beiden Seiten aus verändert werden.

Listing 1: One-Way Binding eines Textes [Pol18]

```
1  <dom-module id="user-view">
2    <template>
3      <div>[[name]]</div>
4    </template>
5
6    <script>
7      class UserView extends Polymer.Element {
8        static get is() {return 'user-view'}
9        static get properties() {
10         return {
11           name: String
12         }
13       }
14     }
```

```
15
16        customElements.define(UserView.is, UserView);
17      </script>
18    </dom-module>
19
20    <!-- Verwendung in einer anderen Komponente -->
21    <user-view name="Samuel"></user-view>
```

Häufig müssen Daten für den Nutzer übersichtlich als Liste angezeigt werden. Zur Darstellung von Arrays bietet Polymer einen *Template repeater* an. Durch die Verwendung von `<template is="dom-repeat">` kann eine Darstellung der Elemente aus einer Liste erzeugt werden (siehe Listing 2 Z. 7-12). Nach demselben Prinzip kann das Anzeigen einzelner Teile des Templates durch `dom-if` an eine Bedingung geknüpft werden (siehe Listing 2 Z. 9-11). [WJ16]

Listing 2: Auflistung der Elemente eines Arrays

```
1   <dom-module id="ba-ausleihe">
2     <template>
3       <style>
4         ...
5       </style>
6
7       <template is="dom-repeat" items="[[books]]">
8         <p>[[item.title]]</p>
9         <template is="dom-if" if="[[item.lender]]">
10          <p>ausgeliehen</p>
11        </template>
12      </template>
13    </template>
14
15    <script>
16    /**
17    * Overview of all available books.
18    * @customElement
19    * @polymer
20    */
21    class Ausleihe extends Polymer.Element {
22      static get is() { return 'ba-ausleihe'; }
23      static get properties() {
24        return {
25          /** Array with all books. */
26          books: Array
27        }
```

```
28          }
29        }
30        window.customElements.define(Ausleihe.is, Ausleihe);
31        </script>
32    </dom-module>
```

Da der Fokus der Arbeit auf der Backendendwicklung liegt, wird das Frontend recht schlicht gehalten. Die Polymeranwendung wurde initial aus dem `Polymer Starter Kit` erzeugt. Standardmäßig ist hierbei eine Headerzeile mit dem Titel der Anwendung, sowie ein linksbündiges Menü enthalten. Einzelne hinzugefügte Ansichten repräsentieren die jeweiligen Funktionalitäten. Unter dem Menüpunkt *Bücherausleihe* erhält der Nutzer beispielsweise eine Übersicht aller Bücher und kann diese über eine Checkbox zum Ausleihen auswählen (siehe Abb. 12). Falls ein Buch bereits ausgeliehen ist, wird die Checkbox zur Ausleihe gesperrt.

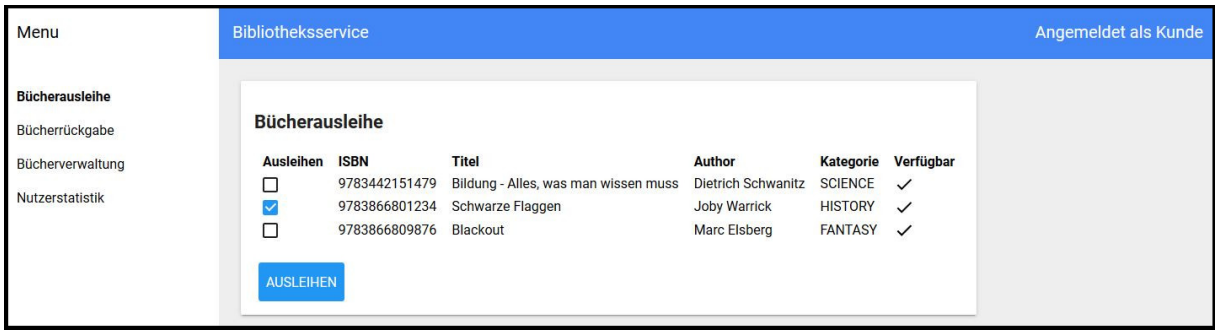

Abbildung 12: Maske Bücherausleihe

Analog dazu können auf einer weiteren Maske die ausgeliehene Bücher zurückgegeben werden.

Wie bereits erwähnt, haben Mitarbeiter die Möglichkeit den aktuellen Bücherbestand zu bearbeiten. Dies ist auf der Ansicht *Bücherverwaltung* möglich (siehe Abb. 13). Je nach Auswahl des Buttons öffnet sich ein Dialog für die jeweilige Funktionalität (siehe Abb. 14).

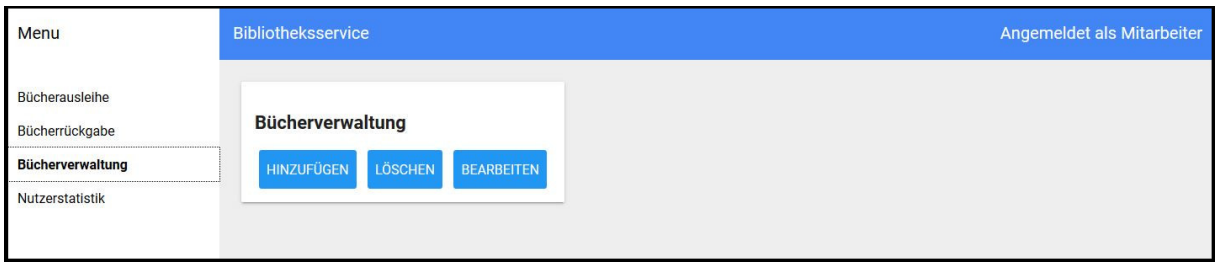

Abbildung 13: Maske Bücherverwaltung

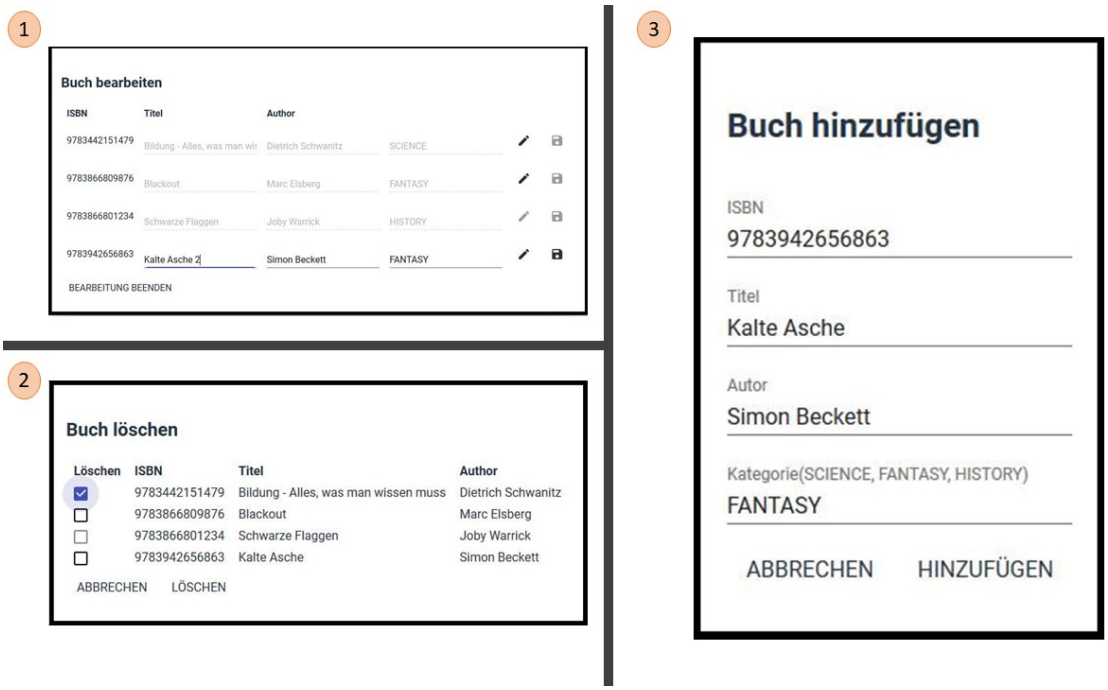

Abbildung 14: (1) Dialog Buch bearbeiten, (2) Dialog Buch löschen, (3) Dialog Buch hinzufügen

Mittlerweile gibt es auch Polymerkomponenten für den spezifischen Umgang mit AWS Modulen. Die Komponente `<aws-dynamodb>` ermöglicht beispielsweise den Zugriff auf Daten aus einer AWS DynamoDB. Ein weiteres Beispiel ist `<aws-lambda>`. Hierdurch können AWS Lambda Functions aufgerufen werden. Damit die beiden Anwendungen dasselbe Frontend nutzen können, wurde auf den Gebrauch dieser Komponenten verzichtet.

3.4 Implementierung der klassischen Webanwendung

Das Ziel ist es, eine klassischen Webanwendung zu entwickeln, die ohne die Verwendung von Cloud-spezifischen Komponenten ihre Funktionalitäten für den Nutzer über das Internet bereitstellt. Die Applikation ist konzipiert, um auf einer herkömmlichen Serverstruktur betrieben zu werden. Der Zusatz *klassisch* impliziert außerdem die Verwendung von gut erprobten und weitläufig anerkannten Frameworks zur Unterstützung in der Entwicklung.

3.4.1 Architektonischer Aufbau der Applikation

Nachdem es sich um einen recht übersichtlichen Anwendungsfall handelt, den die Anwendung widerspiegelt, werden die verschiedenen Funktionalitäten nicht in einzelne Microservices aufgeteilt. Die klassische Applikation ist ein Monolith. Dabei wird eine große Einheit als Anwendung ausgeliefert. Trotzdem kann der Code in verschiedene Komponenten unterteilt werden. [Inc18, S. 9]

Diese Unterteilung, sowie die Wahl der Architektur, kann einen großen Einfluss auf die spätere Anwendung haben. Laut Philippe Kruchten umfasst Softwarearchitektur Themen wie die Organisation des Softwaresystems und wichtige Entscheidungen über die Struktur und das Verhalten der Applikation. [Kru04, S. 288]

Im Fall einer Webanwendung bietet sich eine sogenannte *Multi-Tier Architektur* an. Hierbei wird auf eine klare Abgrenzung zwischen den einzelnen Tiers, beziehungsweise Schichten geachtet. Am weitesten verbreitet ist die 3-Tier Architektur. Die drei dabei zu trennenden Bestandteile sind Präsentation, Applikationsprozesse und Datenmanagement. Die Applikation wird in Frontend, Backend und Datenspeicher aufgeteilt.

Die Präsentationsschicht enthält die Benutzeroberfläche und stellt die Daten gegenüber dem User dar. Somit kann die Interaktion zwischen Client und Backend ermöglicht werden. Eine Ebene darunter befindet sich die Logikschicht. Diese enthält die Geschäftslogik und stellt die Funktionalität der Anwendung bereit. Außerdem dient diese Schicht als Verbindung zwischen Präsentation und Datenspeicher. Typischerweise handelt es sich bei einer Webanwendung um einen Applikationsserver, der den Code ausführt und via HTTP mit dem Client kommuniziert. Als drittes folgt die Datenhaltungsschicht. Sie übernimmt das dauerhafte Speichern, sowie Abrufen der Daten. Mittels einer API kann die Logikschicht so auf die Datenbank zugreifen (siehe Abb. 15). [Mar15]

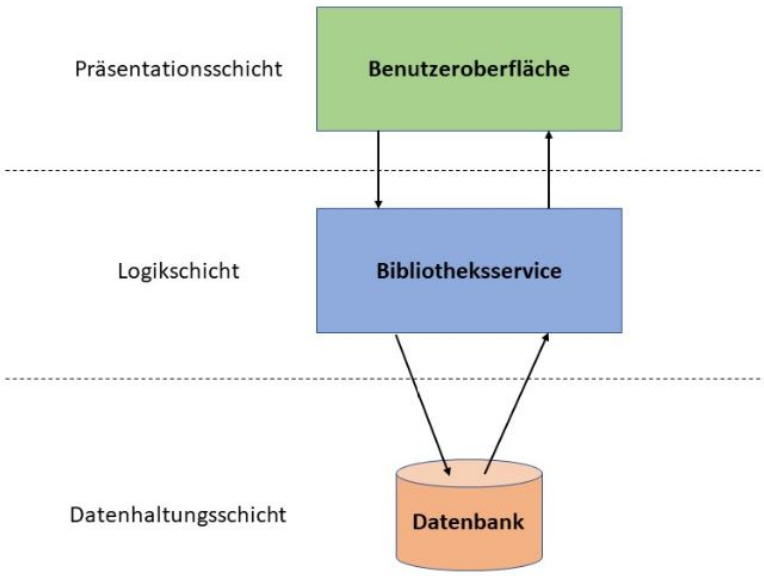

Abbildung 15: 3-Tier Architektur

Ein Vorteil dieses Architekturmusters ist beispielsweise die Möglichkeit Frontend und Ba-

ckend unabhängig voneinander ausliefern zu können, da diese in unterschiedlichen Tiers getrennt sind. Durch diese Trennung können einzelne Tiers problemlos angepasst und erweitert oder sogar komplett ersetzt werden. Auch die Skalierung gestaltet sich durch die eigenständigen Tiers wesentlich einfacher und effizienter. Des Weiteren können Logik- und Datenhaltungsschicht für unterschiedliche Präsentationen eingesetzt und somit wiederverwendet werden. [Mar15]

Wie anfangs erwähnt, kann die Implementierung der Geschäftslogik trotz monolithischer Struktur in verschiedene Komponenten unterteilt werden. Die Logikschicht wird dabei in unterschiedliche Layer eingeteilt. Es wird daher von einer *Layered Architektur* gesprochen. Ein Layer betrifft also die logische Trennung von Funktionalitäten, wohingegen ein Tier auch eine physikalische Abgrenzung mit sich bringt. Ein einzelnes Tier kann somit mehrere Ebenen beinhalten.

Die Aufteilung der Layer erfolgt ähnlich wie die Abgrenzung zwischen den einzelnen Tiers. Präsentations-, Business- und Persistenzlayer sind die Bestandteile der Applikation (siehe Abb. 16). Ersteres stellt Endpunkte für die Kommunikation mit dem Client bereit. Die Geschäfts- und Anwendungslogik befindet sich im Businesslayer und die Persistierung wird, wie der Name schon sagt, vom Persistenzlayer übernommen.

Neben der horizontalen Aufteilung in Layer ist auch eine vertikale Trennung möglich. Dabei werden die Layer nach fachlichen Aspekten in verschiedene Komponenten aufgeteilt (siehe Abb. 16).

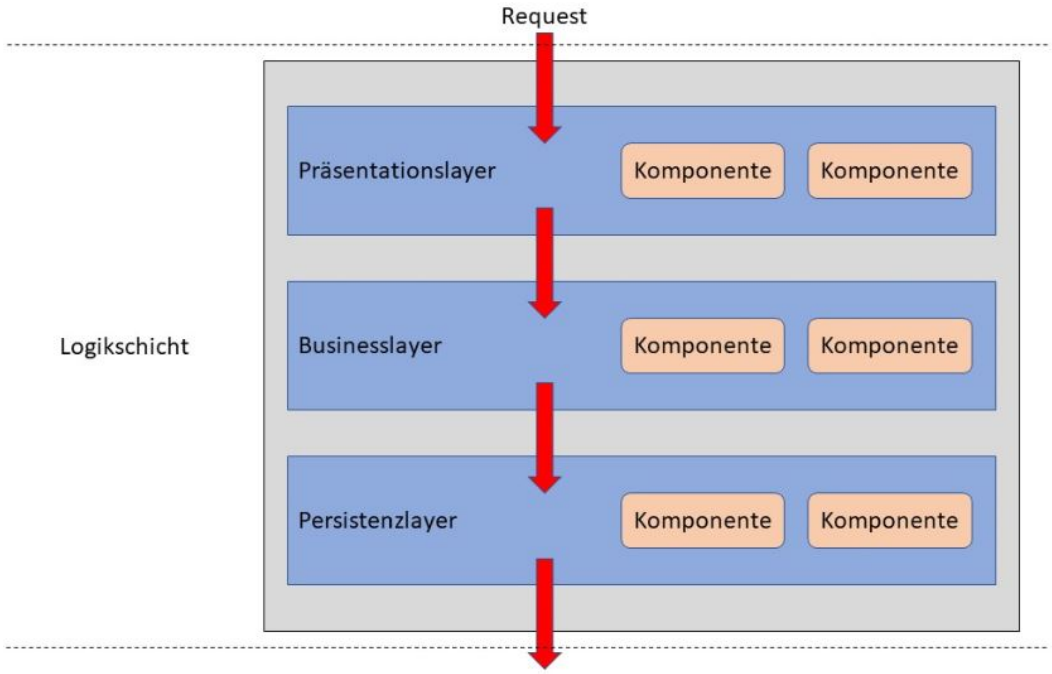

Abbildung 16: Layered Architektur nach [Ric15, S. 3]

Wie bei einigen anderen Architekturmustern ist die Schlüsseleigenschaft der *Layered Architektur* die Abstraktion und Trennung zwischen den verschiedenen Layern. So muss sich das Präsentationslayer beispielsweise nicht damit befassen, wie Kundendaten aus dem Datenspeicher geladen werden. Oder auch das Businesslayer muss nicht wissen, wo die Daten verwaltet werden. Die Komponenten einer Ebene beschäftigen sich lediglich mit der Logik innerhalb ihres Layers. Durch diese Abgrenzung zwischen den Schichten gestaltet sich die Entwicklung, das Testen und der Betrieb der Anwendung wesentlich einfacher. Auch die Einführung eines Rollen- und Zuständigkeitsmodell ist beispielsweise deutlich effizienter durchführbar. [Ric15, S. 2]

Ein weiterer wichtiger Punkt in Bezug auf die Schichtenarchitektur ist der Ablauf der Requests. Die Anfragen fließen horizontal von einem Layer zum Nächsten (siehe Abb. 16). Dabei kann es nicht vorkommen, dass eine Schicht übersprungen wird. Dies ist notwendig, um das *layers of isolation* Konzept zu erhalten. Ziel ist es dabei die Abhängigkeiten zwischen den unterschiedlichen Layers so gering wie möglich zuhalten. Änderungen in einzelnen Layern beeinflussen grundsätzlich keine weiteren Schichten. [Ric15, S. 3]

3.4.2 Implementierung der Anwendung

Um das beschriebene Architekturmodell umzusetzen, wird Spring Boot verwendet. Es dient zur Minimierung von *boilerplate code*. Erreicht wird dies durch Spring spezifische Annotationen und vereinfacht so den Umgang mit dem Spring Framework. Spring folgt

dem Prinzip *Convention over Configuration*. Dem Entwickler wird so eine Menge an konfigurativen Aufgaben abgenommen. Somit können ohne großen Aufwand standardmäßig bereitgestellte Funktionen in Anspruch genommen oder eigene Funktionalitäten hinzugefügt werden. Spring Boot bringt beispielsweise bereits einen eingebetteten Tomcat-Server mit. Dieser bietet eine vollständige Laufzeitumgebung für die Anwendung und ermöglicht ein einfaches Debugging.

Als Tool zur Abhängigkeitsverwaltung für die Beispielanwendung wird Maven verwendet. Zur Bereitstellung eines Spring Boot Programms ist dann lediglich eine `pom.xml` Datei zur Verwaltung der Abhängigkeiten, sowie eine Klasse zum Starten der Anwendung notwendig (siehe Listing 3). Durch die Abhängigkeit zu Spring Boot werden alle weiteren benötigten Bibliotheken automatisch nachgeladen. Außerdem ist es möglich neue Komponenten zum Klassenpfad hinzuzufügen, die daraufhin automatisch konfiguriert werden. [Wol13]

Listing 3: Einstiegsklasse für eine Spring Boot Anwendung

```
1  @SpringBootApplication
2  public class ClassicApplication {
3    public static void main(String[] args) {
4      SpringApplication.run(ClassicApplication.class, args);
5    }
6  }
```

Dieses Startbeispiel kann um verschiedene weitere Features aus dem Spring-Stack oder auch um eigene Funktionalitäten erweitert werden. Neben dem eingebetteten Server stellt Spring Boot von Haus aus auch eine H2 In-Memory Datenbank zur Verfügung. Diese eignet sich hervorragend, um prototypische Anwendungen zu erstellen. Bei der H2 Datenbank handelt sich um einen relationalen Datenspeicher, der mit dem Start der Applikation neu initialisiert und nach dem Beenden wieder zurückgesetzt wird. Dieses voreingestellte Datenbankmanagementsystem kann jedoch auch jederzeit durch einen eigenen Datenbankserver ersetzt werden. Hierzu müssen lediglich ein paar Einstellungen im `application.yml` Dokument, das als Konfiguration für die Spring Boot Anwendung dient, vorgenommen werden.

Damit zu Beginn der Anwendung bereits Daten, wie zum Beispiel Nutzer, vorliegen, wird das Datenbankmigrationstool *Flyway* verwendet. Hierbei können zum einen die Struktur der Datenbank validiert, sowie zum anderen die Tabellen mit Werten befüllt werden.

Zur Abbildung des Datenmodells auf die Datenbank dient das *Object-relational Mapping (ORM)*. Im einfachsten Fall werden dabei Klassen zu Tabellen, die Objektvariablen zu Spalten und die Objekte zu Zeilen in der Datenbank. Bei der Implementierung

hilft dabei die *Java Persistence API (JPA)*, die im Spring Umfeld von der Komponente `spring-boot-starter-data-jpa` unterstützt wird. So können Klassen mit der Annotation @Entity versehen werden. Ihre Objekte sind dann bereit, um in der Datenbank gespeichert zu werden. Der Schlüssel der Tabelle wird durch die Annotation @Id festgelegt. Des Weiteren kommt die Annotation @Enumerated zum Einsatz. Sie legt fest, ob eine Enum als Text oder Zahl gespeichert wird. Auch der Name @Column, sowie Einschränkungen für einzelne Spalten, wie zum Beispiel @NotNull, können über Annotationen festgelegt werden. Elementarer Bestandteil bei relationalen Datenbankmodellen sind die Beziehungen zwischen den Entitäten. Diese können ebenfalls durch Annotationen abgebildet werden. So kann zum Beispiel die Beziehung zwischen Buch und Nutzer wie folgt abgebildet werden (siehe Abb. 17):

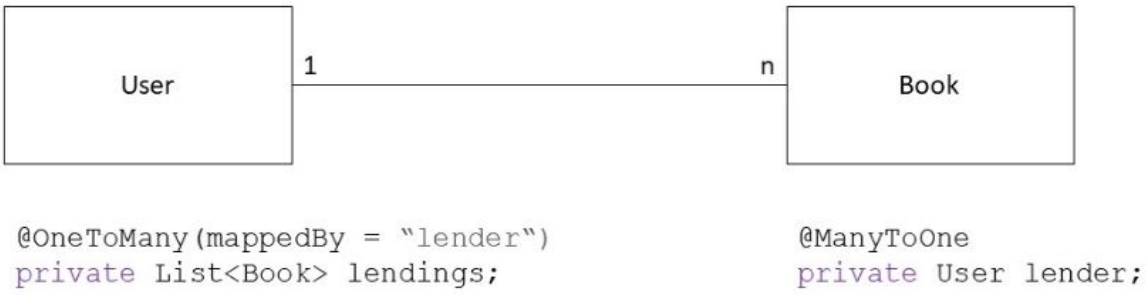

Abbildung 17: Beziehung zwischen User und Book

Diese vier Zeilen sind ausreichend, um in der Buchtabelle einen Fremdschlüssel zu erzeugen, der sich auf den Ausleiher bezieht. So kann die Beziehung zwischen einem Nutzer und seinen ausgeliehenen Büchern ohne eine weitere Zuordnungstabelle abgebildet werden.

Nachdem Datenbankmodell und Objektnetz übereinstimmen, kann die Entwicklung mit der Implementierung der Logikschicht fortgesetzt werden. Spring unterstützt hierbei die beschriebene Aufteilung in verschiedene Layer. Wie bei der Darstellung des Datenmodells kommen dabei ebenso Annotationen zum Einsatz (siehe Abb. 18).

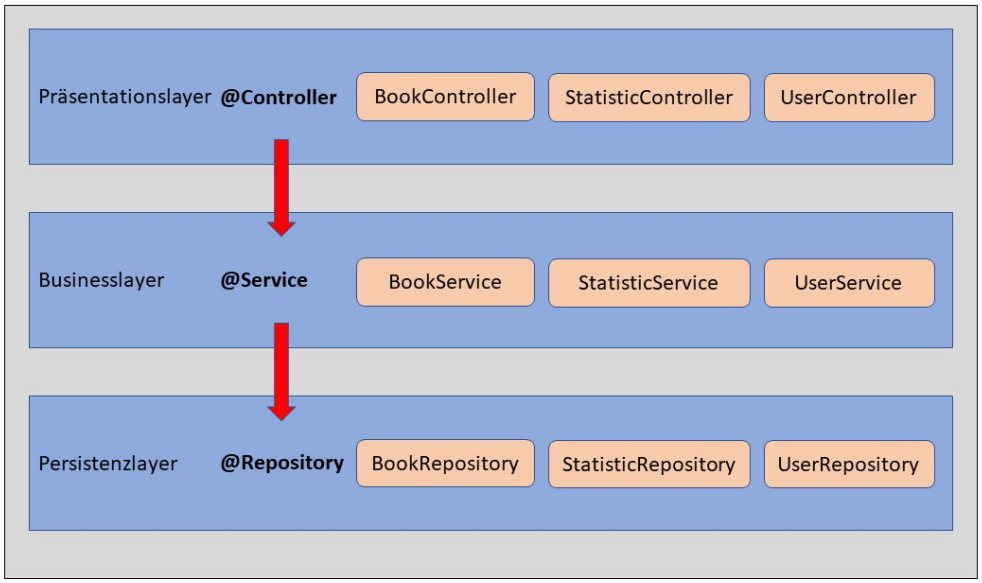

Abbildung 18: Layered Architektur in Spring

Die verschiedenen Layer werden durch sogenannte *Spring Beans* abgebildet. Diese werden aus mit Spring-Annotationen versehenen Klassen(Controller, Service, Repository) erzeugt und, wenn benötigt, instanziiert und konfiguriert. [Wol13]

Der Zugriff aus der Logikschicht heraus auf Daten aus der Datenhaltungsschicht wird durch *Repositories* ermöglicht. Hierbei handelt es sich lediglich um Interfaces, die vom `JpaRepository` erben und somit standardmäßig Methoden wie save() oder find() zum Zugriff auf die Daten im Speicher bereitstellen. Außerdem besteht die Möglichkeit spezifischere Abfragen durch sprechende Methodensignaturen hinzuzufügen (siehe Listing 4).

Listing 4: Repository für die Tabelle User

```
1  @Repository
2  public interface UserRepository extends JpaRepository<User, Integer> {
3    User findUserByUsername(String username);
4  }
```

Eine Schicht über den Repositories befinden sich die Serviceklassen. Diese enthalten die Geschäftslogik. Neben unterschiedlichsten Berechnungen und Validitätsprüfungen werden hier Daten geladen, gespeichert und modifiziert. Der Zugriff auf den Datenspeicher erfolgt über die Repositories.

Diese werden den Services mittels Dependency Injection (DI) bereitgestellt. Hierzu bietet Spring implizite Konstruktor Injektion an. Enthält die betroffene Klasse lediglich einen

Konstruktor, wird in neueren Spring Versionen die Annotation @Autowired nicht mehr benötigt. Spring erzeugt nun im Hintergrund das passende Objekt. Hierzu wird aus dem DI-Container, der alle Beans enthält, die passende Klasse initialisiert. Die Inhalte des Containers werden bereits beim Start der Anwendung erstellt und nicht im Verlauf des Programms. Typischerweise liegen alle Objekte als Singleton Instanz vor. [Kar18]

Zu einer weiteren Entkopplung führt das Implementieren gegen ein Interface. Hierfür wird für jeden Service ein Interface angelegt. Dieses kann dann im Controller mittels Konstruktor Injektion injiziert werden, sodass es jederzeit möglich ist die Umsetzung hinter dem Interface zu ersetzen (siehe Listing 5 Z. 3-7).

Die Servicemethoden werden aus den Controllern heraus aufgerufen. Controller definieren REST-Endpunkte, die für den Client als Einstiegspunkt zur Anwendung dienen und den Zugriff auf die entsprechenden Services ermöglichen (siehe Listing 5 ab Z. 9). Die Endpunkte innerhalb eines Controller unterscheiden sich anhand des Pfades, beziehungsweise des REST-Verbs. Durch die Annotation @RequestMapping an der Klasse kann ein Basispfad für alle Einstiegspunkte des Controllers festgelegt werden. So gibt es für jede REST-Methode die entsprechende Mapping-Annotation wie zum Beispiel @GetMapping, @PostMapping und @DeleteMapping. Um dem Client den Zugriff auf alle Bücher, sowie das Hinzufügen und Löschen eines Exemplars zu ermöglichen, ist folgende Implementierung des Controllers notwendig (siehe Listing 5).

Listing 5: Beispiel BookController

```
1  @RestController
2  public class BookController {
3    private BookService bookService;
4
5    public BookController(BookService bookService) {
6      this.bookService = bookService;
7    }
8
9    @GetMapping("/books")
10   public ResponseEntity<Collection<Book>> getBooks() {
11     return ResponseEntity.ok(bookService.getBooks());
12   }
13
14   @PostMapping(path = "/books", consumes = "application/json")
15   public ResponseEntity<Book> addBook(@RequestBody Book book) {
16     return ResponseEntity.ok(bookService.addBook(book));
17   }
18
19
```

```
20      @DeleteMapping(path = "/books/{isbn}")
21      public ResponseEntity<Book> deleteBook(@PathVariable String isbn) {
22        return ResponseEntity.ok(bookService.deleteBook(isbn));
23      }
24    }
```

Auch die Authentifizierung wird durch eine passende Spring Komponente erleichtert. Spring Security ermöglicht es die Anwendung durch *Basic Authentication* und andere Authentifizierungsverfahren zu schützen. Vor dem Zugriff auf die Applikation muss sich der User mit einem validen Nutzernamen und Passwort gegenüber der Anwendung authentifizieren. Alle Requests sind nun durch die Authentifizierung gesichert. Über HttpSecurity können einzelne Ressourcen oder auch Pfadgruppen individuell für einen offenen Zugriff freigegeben werden. Des Weiteren kann durch das Einbinden des UserDetailsService der Login an die eigenen Nutzer angepasst werden (siehe Listing 6). Über diesen wird bei der Anmeldung überprüft, ob ein gültiges Userobjekt in der Anwendung vorliegt. Dieses wird als UserDetails zurückgegeben. Der authentifizierte Nutzer kann nun über das Authentication Objekt in der Applikation abgefragt werden (siehe Listing 7).

Listing 6: Implementierung des UserDetailsService

```
1   @Service
2   public class UserServiceImpl implements UserService,
        UserDetailsService {
3     private UserRepository userRepository;
4
5     public UserServiceImpl(UserRepository userRepository) {
6       this.userRepository = userRepository;
7     }
8
9     @Override
10    public UserDetails loadUserByUsername(String username) {
11      User user = userRepository.findUserByUsername(username);
12      if (user == null) {
13        return null;
14      }
15      return new org.springframework.security.core.userdetails.User(
            username, "{noop}" + user.getPassword(), AuthorityUtils.
            createAuthorityList(user.getRole().toString()));
16    }
17  }
```

Listing 7: Abfrage des authentifizerten Users

```
1  /**
2   * Returns all lent books for a specific user.
3   * @param authentication authentication object containing the active
        user
4   * @return collection with all lent books
5   */
6  public Collection<Book> getLendings(Authentication authentication) {
7    User lender = userRepository.findUserByUsername(authentication.
         getName());
8    return lender.getLendings();
9  }
```

Eine rollenbasierte Autorisierung ist ebenso möglich. Dafür werden lediglich die Einstiegspunkte zur Applikation, das heißt die Endpunkte im Controller, mit @PreAuthorize und der zugehörigen Rolle annotiert (siehe Listing 8 Z. 7).

Listing 8: PreAuthorize an einem Endpunkt im Controller

```
1  /**
2   * Creates a new book in the service.
3   * @param book the new book
4   * @return response with the status and the added book
5   */
6  @PostMapping(path = "/books", consumes = "application/json")
7  @PreAuthorize("hasAuthority('ADMIN') or hasAuthority('EMPLOYEE')")
8  public ResponseEntity<Book> addBook(@RequestBody Book book) {
9    return ResponseEntity.ok(bookService.addBook(book));
10 }
```

Die letzte noch zu implementierende Funktionalität ist das in 3.2 beschriebene Anlegen einer Ausleihstatistik. Besonders elegant wäre es hierbei, die Aktualisierung der Statistik als Reaktion auf das Ausleihen auszulösen. Die Annotation @PostPersist kann genutzt werden, um auf das Speichern der Ausleihe zu reagieren. Die damit annotierte Methode wird als Callback nach dem erfolgreichen Speichern aufgerufen. Allerdings ist es in dieser Methode nicht möglich ein weiteres Mal auf die Datenbank zuzugreifen. Somit kann die neue Statistik auf diesem Weg nicht persistiert werden. Alternativ wurde nun ein weiterer REST-Endpunkt angelegt, der nach der erfolgreichen Durchführung der Ausleihe über die Oberfläche per Hand aufgerufen wird.

3.4.3 Testen der Webanwendung

Testen ist eine wichtige Aufgabe im Entwicklungsprozess, um die Qualität der Anwendung zu sichern. Neben der Überprüfung des Softwareverhaltens wird die vollständige Abdeckung der Anforderungen kontrolliert. Das Testen einer Spring Webanwendung kann in zwei Teile unterteilt werden. Zum einen werden Komponententests bzw. Unittests benötigt. Diese verifizieren individuell die Logik der einzelnen Komponenten. Zum anderen werden Integrationstests angelegt. Diese stellen das richtige Zusammenspiel der verschiedenen Komponenten untereinander sicher. [Inf18]

Im Springumfeld ist es sinnvoll, die Testklasse mit der Annotation @SpringBootTest zu versehen. So wird bei der Ausführung der Tests ein Springkontext, der mit dem Produktivsystem identisch sein kann, aber nicht muss, aufgebaut. [Gig18]

Komponenten-/Unittests

Um lediglich ein Modul zu überprüfen, müssen alle Komponenten, die mit diesem Modul interagieren, für den Test ausgeschlossen werden. Hierfür können sogenannte Mocks eingesetzt werden. Im Springkontext gibt es dafür die @MockBean Annotation. Somit können fremde Komponenten durch eine Art Platzhalter ersetzt werden. Diese nehmen ein vorhersagbares Verhalten an (siehe Listing 9 Z. 5-6 und 21-22). Dadurch kann ausgeschlossen werden, dass das betrachtete Modul durch Fremdeinflüsse beeinträchtigt wird. Für den Test eines Controllers wird also beispielsweise ein Mock für den verwendeten Service angelegt. [Gig18]

Eine weitere Schwierigkeit in den Testfällen der Controller ist das Simulieren eines HTTP-Requests. Dies ist mit Hilfe der MockMvc Klasse möglich. Im Test kann so ein Request erstellt und die Antwort überprüft werden (siehe Listing 9 Z. 23-27). [Gig18]

Ebenso eine Besonderheit ist die Annotation @WithMockUser. Hiermit wird der authentifizierte Benutzer mit der zugehörigen Rolle für den Testfall festgelegt. Somit kann auch der Zugriffsschutz mit getestet und beispielsweise eine AccessDeniedException provoziert werden (siehe Listing 9 Z. 19).

Nach demselben Prinzip wir im Test des Services ein Mock für das zu Grunde liegende Repository angelegt.

Listing 9: Ausschnitt aus dem StatisticControllerTest

```
1  @SpringBootTest
2  @RunWith(SpringRunner.class)
3  public class StatisticControllerTest {
4    private static final Statistic STATISTIC = new Statistic(1, 34,
         Category.SCIENCE);
5    @MockBean
6    private StatisticService statisticService;
7    @Autowired
8    private WebApplicationContext webAppContext;
9    private MockMvc mockMvc;
10
11   @Before
12   public void setup() {
13     MockitoAnnotations.initMocks(this);
14     mockMvc = MockMvcBuilders
15       .webAppContextSetup(webAppContext).build();
16   }
17
18   @Test
19   @WithMockUser(authorities = "EMPLOYEE")
20   public void testGetStatistic() throws Exception {
21     when(statisticService.getStatistic("SCIENCE"))
22       .thenReturn(STATISTIC);
23     RequestBuilder requestBuilder = MockMvcRequestBuilders
24       .get("/statistics/SCIENCE");
25     MvcResult result = mockMvc.perform(requestBuilder).andReturn();
26     assertEquals(200, result.getResponse().getStatus());
27     assertEquals("{\"id\":1,\"count\":34,\"category\":\"SCIENCE\"}",
           result.getResponse().getContentAsString());
28   }
29 }
```

Integrationstests

Nachdem durch die Unittests die Korrektheit der einzelnen Module festgestellt wurde, können Integrationstests dazu genutzt werden, um die Zusammenarbeit zwischen den verschiedenen Teilen zu testen. Hierzu muss lediglich auf die Mocks verzichtet werden, damit alle beteiligten Komponenten beim Aufruf ausgeführt und somit überprüft werden können (siehe Listing 10 Z. 7-12). [Gig18]

Da für den Test ein Springkontext eingesetzt wird, kann auch eine Testinstanz der Datenbank erstellt werden. Diese kann mit Testdaten befüllt werden.

Listing 10: Integrationstest für eine Methode aus dem Bookservice

```
1  @SpringBootTest
2  @RunWith(SpringRunner.class)
3  public class BookServiceIntegrationTest {
4    @Autowired
5    private BookService bookService;
6
7    @Test
8    public void testGetBooks() {
9      Collection<Book> books = bookService.getBooks();
10     assertThat(books).isNotNull().isNotEmpty();
11     assertEquals(3, books.size());
12   }
13 }
```

3.5 Implementierung der Serverless Webanwendung

Bei der zweiten Anwendung handelt es sich um die Serverless Applikation. Diese wird von einem externen Provider betrieben. Als Betreiber wurde hierfür das Serverless Angebot von Amazon AWS Lambda gewählt. So bietet Amazon als einer der Vorreiter im Cloud-Umfeld nicht nur ein großes Angebot an weiteren Cloudtools, sondern stellt dem Nutzer auch ein Freikontingent an Ressourcen zur Verfügung [Kö17, S. 12]. Des Weiteren ist AWS Lambda der populärste Vertreter auf dem Serverless Markt [Kö17, S. 18]. Es werden die Programmiersprachen JavaScript, Python, C# und Java mit entsprechenden Laufzeitumgebungen unterstützt [Kö17, S. 66]. Um eine Vergleichbarkeit der beiden Anwendungen zu erhalten, wird Java in Kombination mit Maven als Build-Management-Tool für die beispielhafte Serverless Webanwendung verwendet.

3.5.1 Architektonischer Aufbau der Serverless Applikation

Da lediglich die Anwendungslogik implementiert werden muss, unterscheidet sich die Architektur der Serverless Applikation grundlegend von dem eben erläuterten Aufbau. Die Aufteilung der Anwendungslogik in viele kleine Komponenten ähnelt dem Microservicegedanken. Die Serverless Architektur erhöht somit die Autonomie der einzelnen Funktionalitäten. Diese werden in Functions abgebildet und durch Events aus verschiedenen Quellen aufgerufen. Dabei wird auch von einer *Event-driven* Architektur gesprochen. [Inc18, S. 9-10]

Hierbei handelt es sich um asynchrone Events, die von einer Function abonniert werden können. Dieses Muster ist auf verschiedenen Anwendungstypen anwendbar und eignet sich, wie in diesem Fall zu sehen ist, besonders gut für skalierbare Applikationen [Ric15,

S. 11]. Alternativ können die Functions durch synchrone Events per HTTP-Request aufgerufen werden.

Wie oben in Kapitel 2.2 beschrieben, wird die Geschäftslogik in einzelne Functions aufgeteilt. Diese werden nach dem FaaS Konzept auf der Plattform des Betreibers ausgeführt. Hierbei können sie mit weiteren Komponenten von Drittanbietern interagieren [Kra18, S. 14]. Häufig werden diese zur Authentifizierung oder Datenpersistierung genutzt. Prinzipiell sollten nur eigene Functions implementiert werden, falls es keinen fremden Service gibt, der diese Aufgabe übernehmen kann. Um REST-Endpunkte bereitzustellen, wird ein API Gateway eingesetzt. Dieses wandelt eintreffende Anfragen in FaaS-konforme Events um und ruft somit die zugehörige Function auf (siehe Abb. 19). [Kra18, S. 16-17]

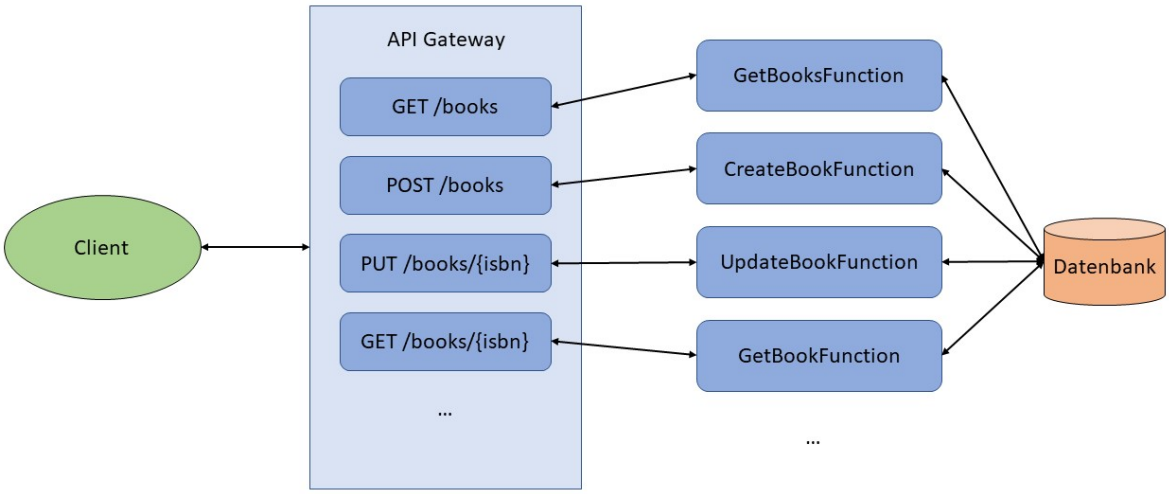

Abbildung 19: API Gateway

Als API Gateway wird das BaaS-Angebot von AWS genutzt. Dieses muss lediglich entsprechend konfiguriert werden. Je nach Konfiguration kann das AWS API Gateway direkt Aufgaben wie zum Beispiel Sicherheitsüberprüfungen übernehmen. [Rob18]

Neben der Verteilung der Logik sorgt die Trennung von Client und Cloud-Anwendung durch das Gateway ebenso dafür, dass die Anwendung nicht mehr als ein Paket dem Entwickler vorliegt. Damit einhergehend liegt auch die Ablaufsteuerung nicht mehr unter der Kontrolle einer zentralen Stelle, welche die Orchestrierung übernimmt. Der Ablaufprozess wird durch den Eventfluss organisiert. [Kra18, S. 17]

Diese Entwicklung wird auch *choreography over orchestration* genannt und schließt somit den Kreis zur Ähnlichkeit mit dem Microserviceansatz. [Rob18]

Auch die Datenhaltung wird durch eine Komponente aus dem Amazonumfeld übernommen. *DynamoDB* ist eine nicht relationale Datenbank, die gut in Kombination mit AWS Lambda Functions verwendet werden kann. Im Gegensatz zu relationalen Datenbanken werden dabei die Daten nicht in Tabellen mit Zeilen und Spalten organisiert, sondern als simple Datenobjekte oder Dokumente abgelegt. Nach dem Key-Value-Datenbankmodell werden Schlüssel als Identifikatoren für die Werteobjekte eingesetzt. Da Objekte innerhalb eines Datenspeichers nicht demselben Schema folgen müssen, sind nicht relationale Modelle flexibel einsetzbar. [Lit17]

Trotz der Möglichkeit alle Objekte in einem Datenspeicher zu verwalten, kann es sinnvoll sein den Datenbestand aufzuteilen. Dies ist beispielsweise der Fall, wenn Datensätze mit sehr unterschiedlichen Zugriffsmustern enthalten sind [Ama12]. Im Anwendungsfall des Bücherservices werden Bücher und Statistiken in unterschiedlichen DynamoDB-Tabellen gehalten, da so die Konfiguration von Sekundärindizes bei der Verwendung einer großes Tabelle eingespart werden kann. Dennoch kann der Zugriff auf den kleinen Datenbestand immer noch performant erfolgen.

DynamoDB kann auch als Auslöser für ein asynchrones Event dienen. Dabei löst eine Änderung im Datenspeicher ein Event aus, dass wiederum eine Function aufruft (siehe Abb. 20).

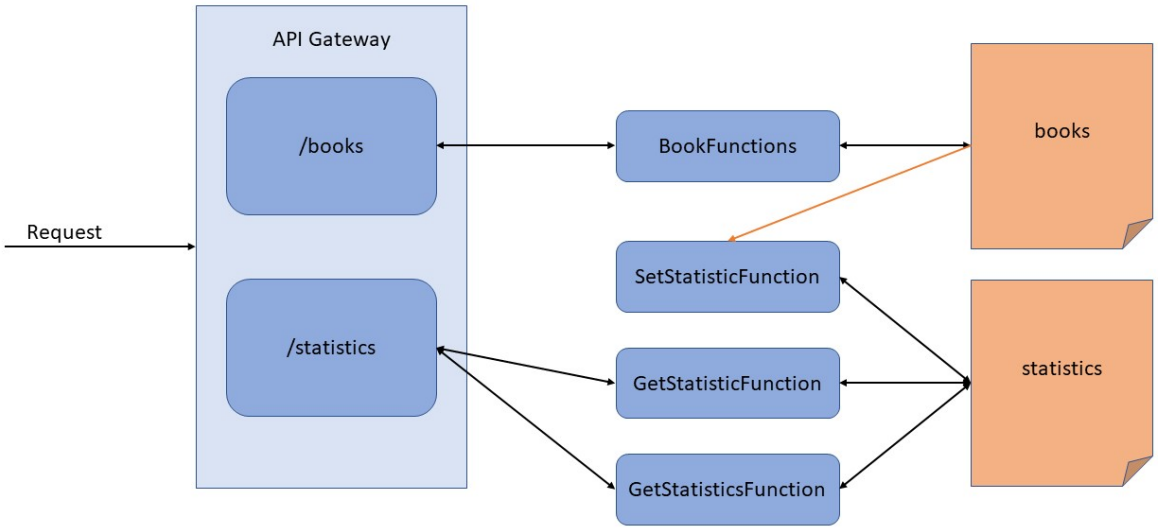

Abbildung 20: Datenbankevent ruft Function auf

3.5.2 Implementierung der Anwendung

Zur Implementierung der einzelnen Functions wird das AWS Serverless Application Model (SAM) verwendet. Es dient zur lokalen Entwicklung von Serverless Functions im Umfeld von Amazon. Mit Hilfe des Tools kann über die Kommandozeile ein Startprojekt erstellt werden. Dieses enthält als Quellcode lediglich eine Function, die durch eine Klasse abgebildet wird. Jede Function muss die Klasse RequestHandler implementieren. Dessen Methode handleRequest() dient als Einstiegspunkt (siehe Listing 11).

Listing 11: Request Handler für Lambda Function

```
1  package example;
2
3  /**
4  * Handler for requests to Lambda function.
5  */
6  public class FunctionExample implements RequestHandler<Object, Object>
       {
7
8    public Object handleRequest(final Object input, final Context
       context) {
9      //Logik der Function
10   }
11
12 }
```

Des Weiteren gibt es zur Konfiguration eine `template.yaml` Datei. Sie gibt den Ort des Handlers, sowie die Events, die zum Aufrufen der Function dienen, an (siehe Listing 12). In diesem Fall ist sie durch einen GET-Request auf dem Endpunkt `/example` ansprechbar (siehe Listing 12 Z. 12). Durch den Eventtype `Api` wird automatisch das API Gateway erstellt (siehe Listing 12 Z. 10) [Kö17, S. 185].

Listing 12: Ressourcendefinition der Beispiel Function in template.yaml

```
1  Resources:
2    FunctionExample:
3      Type: AWS::Serverless::Function
4      Properties:
5        CodeUri: target/Example-1.0.jar
6        Handler: example.FunctionExample::handleRequest
7        Runtime: java8
8        Events:
9          FunctionExample:
10           Type: Api
11           Properties:
12             Path: /example
13             Method: get
```

Durch SAM ist es möglich die Function lokal auszuführen und zu testen. Für das Deployment wird die Anwendung zuerst verpackt und in einem Amazon S3-Bucket, einer weiteren Komponente aus dem Cloudangebot von Amazon, abgelegt. Im nächsten Schritt werden dann die definierten Ressourcen als Lambda Functions ausgeliefert.

Das initiale Starterprojekt kann um weitere, eigene Functions erweitert werden. Damit nicht jedes Modul einen eigenen Kontext, wie zum Beispiel die Verbindung zur Datenbank, pflegen muss, wird ein sogenanntes *Data Access Object (DAO)* verwendet. Dies übernimmt die Kommunikation mit der Datenbank und sorgt für eine einfache Austauschbarkeit der Datenhaltungskomponente.

Bei der klassischen Anwendung wurden elementare Aufgaben, wie DI oder das Mapping von Java- zu JSON-Objekten, von Spring übernommen. Dies muss nun anderweitig erledigt werden. Für die Umwandlung von Java- zu JSON-Objekten wird der `ObjectMapper` von Jackson eingesetzt. DI wird durch das Framework *Dagger* ermöglicht.

Die Implementierung von Dagger besteht aus zwei Teilen. Zum einen den Modulen, welche die benötigten Abhängigkeiten bereitstellen, und zum anderen einer sogenannten Komponente. Diese ermöglicht die Initialisierung der im Modul definierten Objekte. [Cha17]

Nachdem beispielsweise das DAO im Modul als Abhängigkeit definiert wurde, kann es

mittels @Inject in einem Handler genutzt werden (siehe Listing 14 Z. 5-6).

Für den Zugriff auf die DynamoDB wird ebenfalls ein Modul erstellt, sodass eine Verbindung zur Datenbank hergestellt wird (siehe Listing 13). Über den DynamoDbClient kann dann im BookDao bzw. StatisticDao auf den Datenbestand zugegriffen werden.

Listing 13: Modul zur Bereitstellung der Datenbankverbindung

```
1  @Module
2  public class AppModule {
3
4    @Provides
5    DynamoDbClient dynamoDb() {
6      DynamoDbClient client = DynamoDbClient.builder()
7        .region(Region.EU_CENTRAL_1)
8        .build();
9      return client;
10   }
11 }
```

Die Umsetzung einer vollständigen Function beginnt mit dem Handler. Bei der Implementierung der RequestHandler Klasse wird der Typ des eingehenden Objekts festgelegt. Die Map als Parameter der handleRequest() Methode enthält dabei die Attribute des Requests wie Header, Pfadvariablen und den Body (siehe Listing 14 Z. 16-19). Die Klasse GatewayResponse stellt eine einheitliche Antwort mit entsprechendem Body, Header und Statuscode an das API Gateway dar (siehe Listing 14 Z. 23 und 25). Nach Verarbeitung des eingehende Parameters kann das gesuchte Buch über das BookDao erfragt werden (siehe Listing 14 Z. 20).

Listing 14: GetBookFunction

```
1  public class GetBookHandler implements RequestHandler<Map<String,
       Object>, GatewayResponse> {
2    @Inject
3    ObjectMapper objectMapper;
4
5    @Inject
6    BookDao bookDao;
7
8    private final AppComponent appComponent;
9
10   public GetBookHandler() {
11     appComponent = DaggerAppComponent.builder().build();
12     appComponent.inject(this);
13   }
```

```
14
15      @Override
16      public GatewayResponse handleRequest(Map<String, Object> input,
            Context context) {
17        String pathParameter = input.get("pathParameters").toString();
18        String isbn = pathParameter.substring(6,
19          pathParameter.length()-1);
20        Book book = bookDao.getBook(isbn);
21        try {
22          return new GatewayResponse(objectMapper.writeValueAsString(book)
                , HEADER, SC_OK);
23        } catch (JsonProcessingException e) {
24          return new GatewayResponse(e.getMessage(), HEADER,
                SC_INTERNAL_SERVER_ERROR);
25        }
26      }
27    }
```

Nachdem über den Datenbankclient das entsprechende Buch aus dem Speicher geladen wurde (siehe Listing 15 Z. 15-19), wird das GetItemResponse Objekt mit der convert() Methode in ein Buch Objekt umgewandelt und kann zurückgegeben werden (siehe Listing 15 Z. 21).

Listing 15: Ausschnitt des BookDaos

```
1   public class BookDao {
2
3     private static final String BOOK_ID = "isbn";
4     private final String tableName;
5     private final DynamoDbClient dynamoDb;
6
7     public BookDao (final DynamoDbClient dynamoDb, final String
          tableName) {
8       this.dynamoDb = dynamoDb;
9       this.tableName = tableName;
10    }
11
12    public Book getBook(final String isbn) {
13      try {
14        return Optional.ofNullable(
15          dynamoDb.getItem(GetItemRequest.builder()
16            .tableName(tableName)
17            .key(Collections.singletonMap(BOOK_ID,
18              AttributeValue.builder().s(isbn).build())))
19            .build()))
```

```
20              .map(GetItemResponse::item)
21              .map(this::convert)
22              .orElse(null);
23          } catch (ResourceNotFoundException e) {
24            throw new TableDoesNotExistException("Book table " + tableName +
                  " does not exist.");
25          }
26      }
27  }
```

Die Authentifizierung erfolgt im Amazonkosmos über das Identity and Access Management (IAM). Dabei können Nutzer, sowie Rollen erstellt werden. Der Zugriff auf einzelne Functions kann über verschiedene Rollen geregelt werden. Damit nicht für jeden Anwender ein Identity and Access Management (IAM) Benutzer angelegt werden muss, gibt es für die User die Möglichkeit sich an einem *Cognito User Pool* zu registrieren. Dieser dient als Benutzerverzeichnis und verwaltet die Informationen zu den einzelnen Nutzern. Nach einer erfolgreichen Authentifizierung stellt Cognito dem User einen Token zur Verfügung. Mit diesem kann er sich beispielsweise gegenüber dem API Gateway ausweisen. [Kö17, S. 141-143]

Damit eine Function auch durch ein Event aus der Datenbank angesprochen werden kann, muss in der Lambda Console für die entsprechende Function DynamoDB als Auslöser hinzugefügt werden. In der Konfiguration der Datenbank wird der sogenannte Stream aktiviert. Dieser erfasst alle Änderungen an der DynamoDb-Tabelle und kann in der Function ausgelesen werden. Dadurch kann auf die Änderung angemessen reagiert werden (siehe Listing 16 Z. 2-4). Als Parameter erhält der Handler das Event aus dem Datenspeicher (siehe Listing 16 Z. 1).

Listing 16: Auslesen des DynamoDbStreams

```
1   public GatewayResponse handleRequest(DynamodbEvent input, Context
        context) {
2     for (DynamodbEvent.DynamodbStreamRecord record : input.getRecords())
          {
3       //Reaktion auf Datenmodifikation
4     }
5   }
```

Auf die Implementierung der Functions folgt anschließend das Deployment. Dieses ist wie oben genannt mit Hilfe des SAM Tools möglich. Daraufhin können alle Functions über die *AWS Lambda Console* eingesehen und verwaltet werden (siehe Abb. 21). Hier können den Functions Rollen für den Zugriffsschutz zugewiesen werden. Des Weiteren muss der

zur Verfügung stehende Arbeitsspeicher für jede Function erhöht werden, da sie sonst bei der Ausführung in ein Timeout laufen.

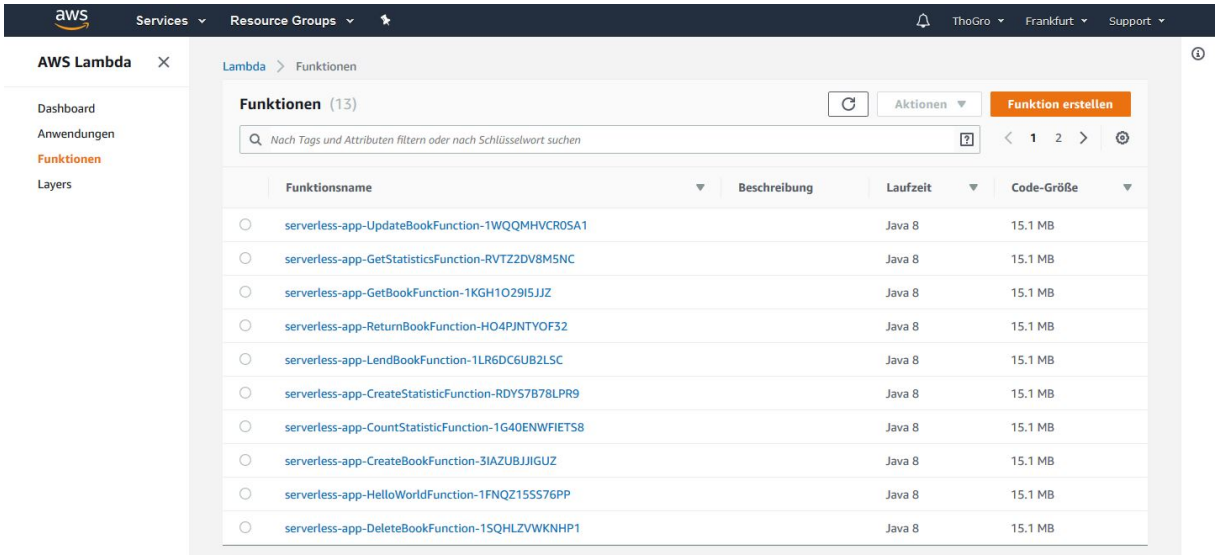

Abbildung 21: Lambda Console

Zuletzt muss auch noch das API Gateway per Hand angepasst werden. Die automatisch erstellte Konfiguration des Gateways erlaubt keine Zugriffe von anderen URLs aus. Damit das Polymerfrontend die Functions aufrufen kann, muss über die Plattform von Amazon für die einzelnen REST-Ressourcen *Cross-Origin Resource Sharing* freigeschaltet werden.

3.5.3 Testen von Serverless Anwendungen

Im Vergleich zu klassischen Applikationen ist das Testen von Anwendungen im Cloud-Umfeld eine große Herausforderung. Wie beim Überprüfen des ersten Prototypen in Kapitel 3.4.3 werden nun die Möglichkeiten zur Durchführung von Komponenten- und Integrationstests beleuchtet. Zusätzlich zu den DAOs werden auch die Handler getestet.

Komponenten-/Unittests

Die Unittests beschäftigen sich mit der Überprüfung einer isolierten Function und können lokal ausgeführt werden [Inc18, S. 15]. Zum Erzeugen von Mock-Objekten wird die Bibliothek *Mockito* verwendet. Sie muss im Gegensatz zur klassischen Anwendung, bei der sie bereits im Modul Spring-Boot-Test enthalten war, gesondert eingebunden werden. Die Nutzung erfolgt dann wieder nach dem bekannten Prinzip (siehe Listing 17).

Listing 17: Ausschnitt des StatisticDao Unittests

```
1  public class StatisticDaoTest {
2    private static final Statistic STATISTIC = new Statistic(12,
         Category.HISTORY);
3    private static final String TABLE_NAME = "statistics";
4    private StatisticDao statisticDao;
5
6    @Mock
7    private DynamoDbClient dynamoDb;
8
9    @Before
10   public void setUp() {
11     MockitoAnnotations.initMocks(this);
12     statisticDao = new StatisticDao(dynamoDb, TABLE_NAME);
13   }
14
15   @Test
16   public void testGetStatistic() {
17     GetItemResponse getItemResponse = GetItemResponse.builder().item(
           createResultMap()).build();
18     when(dynamoDb.getItem(any(GetItemRequest.class))).thenReturn(
           getItemResponse);
19     Statistic statistic = statisticDao.getStatistic(STATISTIC.
           getCategory());
20     assertEquals(STATISTIC, statistic);
21   }
22
23   private Map<String, AttributeValue> createResultMap() {
24     Map<String, AttributeValue> resultMap = new HashMap<>();
25     resultMap.put("category", AttributeValue.builder().s(STATISTIC.
           getCategory().toString()).build());
26     resultMap.put("statisticCount", AttributeValue.builder().s(Integer
           .toString(STATISTIC.getStatisticCount())).build());
27     return resultMap;
28   }
29 }
```

Integrationstests

Die Komponententests der Serverless Anwendung ähneln in der Entwicklung denen der klassischen Applikation. Herausfordernder gestalten sich die Integrationstests. Da der Entwickler über die meisten Komponenten keine Kontrolle hat und somit das Verhalten nicht beeinflussen kann, ist es schwer den kompletten Ablauf zu testen. Die abnehmende Kon-

trolle ist allgemein im Cloud-Umfeld der Fall. Bei Serverless Umsetzungen wird dies soweit getrieben, dass lediglich die Lambda Functions, sowie möglicherweise Gateway Mappings aus der Hand des Entwicklers stammen. [Inc18, S. 16]

Aus diesem Grund können Integrationstests lediglich in der Cloudumgebung durchgeführt werden. Diese kostet jedoch auch Ressourcen, da die Testfälle auf das ausgelieferte System zugreifen. [Inc18, S. 15]

Daher ist es wichtig eine Art Testkontext zu errichten, um die Produktivdaten nicht zu beeinflussen. Der Aufbau einer Testinfrastruktur befasst sich beispielsweise mit der Bereitstellung von Testdaten und den benötigten Ressourcen, dem Deployment der zu testenden Software und der Orchestrierung der Testdurchläufe. Eine Herausforderung dabei ist es die Testumgebung so aufzusetzen, dass sie für jeden Testlauf identisch ist. [CN12, S. 11]

Anstatt des Datenbankmocks im Unittest wird hier eine Verbindugn zur DynamoDb hergestellt (siehe Listing 18 Z. 9-11). Zum Start des Tests wird die Datenbank mit Daten befüllt (siehe Listing 18 Z. 13). Nach der Durchführung des Testfalls wird die Datenbank wieder auf den ursprünglichen Zustand zurückgesetzt (siehe Listing 18 Z. 16-19).

Listing 18: Test des GetBookHandlers

```
1  public class GetBookHandlerTest {
2
3    private static final Book BOOK = new Book("9783868009217",
4      "Verwesung", "Simon Beckett", Category.FANTASY, "null");
5    private BookDao bookDao;
6
7    @Before
8    public void setUp() {
9      DynamoDbClient client = DynamoDbClient.builder()
10       .region(Region.EU_CENTRAL_1)
11       .build();
12     bookDao = new BookDao(client, "books");
13     bookDao.createBook(new CreateBookRequest(BOOK.getIsbn(), BOOK.
           getTitle(), BOOK.getAuthor(), BOOK.getCategory(), BOOK.
           getLender()));
14   }
15
16   @After
17   public void cleanUp() {
18     bookDao.deleteBook(BOOK.getIsbn());
19   }
20
21
```

```
22      @Test
23      public void successfulResponse() {
24        GetBookHandler getBookHandler = new GetBookHandler();
25        Map<String, Object> input = new HashMap<>();
26        input.put("pathParameters", "{isbn=9783868009217}");
27        GatewayResponse response = getBookHandler.handleRequest(input,
              null);
28        assertEquals(200, response.getStatusCode());
29        assertEquals("{\"isbn\":\"9783868009217\",\"title\":
30        \"Verwesung\",\"author\":\"Simon Beckett\",
31        \"category\":\"FANTASY\",\"lender\":\"null\"}",
32          response.getBody());
33      }
34
35    }
```

3.6 Unterschiede in der Entwicklung

Nachdem die Implementierung der beiden prototypischen Anwendungen abgeschlossen ist, folgt der Vergleich und die Gegenüberstellung der zwei Entwicklungen. Die Evaluation wird anhand der unter 3.1 beschriebenen Kriterien durchgeführt, sodass die Unterschiede in der Umsetzung herausgestellt werden.

Für die Durchführung der Evaluation wurden zu den einzelnen Anforderungen Fragen definiert, die mittels einer Metrik beantwortet werden und somit zu einem messbaren Ergebnis führen.

Kategorie	Anf. Nr.	Anforderung	Frage	M. Nr.	Metrik
Implementierungsaufwand	1	Zeitlicher Aufwand	Kann die Anwendung schnell umgesetzt werden?	1	Messung der Entwicklungszeit
	2	Codeumfang	Wie viel Code ist zur Implementierung einer Funktionalität notwenig?	2	LoC für die Umsetzung einer Funktionalität
	3	Einarbeitungszeit	Sind die Entwicklungswerkzeuge schnell zu erlernen?	3	Erlernbarkeit: Ordinalskala (sehr gut, gut, schlecht)

Abbildung 22: Ausschnitt der Fragen mit entsprechenden Metriken

Im Folgenden wird die Evaluation an den beiden Prototypen durchgeführt. Anschließend werden die Erkenntnisse gegenübergestellt und abgeglichen.

3.6.1 Durchführung der Evaluation

Implementierungsaufwand

Grundsätzlich nimmt die Implementierung des Serverless Prototypen wesentlich mehr Zeit in Anspruch als die Erstellung der klassischen Version. Vor allem das Hinzufügen

der einzelnen Functions ist sehr zeitintensiv. Ebenso fehlt die Unterstützung durch ein Framework wie Spring. Das Schließen dieser Lücke im Ökosystem kostet deutlich Zeit.

Bis eine erste Funktionalität lauffähig umgesetzt ist, vergeht mehr Zeit wie beim Hinzufügen einer weiteren Funktion. Dies liegt an der zu implementierenden Basiskonfiguration der Anwendung. Diese Aufgabe muss somit im weiteren Verlauf der Entwicklung kein weiteres Mal ausgeführt werden. Weil bei der klassischen Anwendung deutlich mehr Aufgaben durch das Spring Framework bereits erledigt sind, kann eine Menge an Code eingespart werden. Eine neue Funktionalität kann in beiden Prototypen hinzugefügt werden. Die Erweiterung des bestehenden Controllers und Services auf Seiten von Spring erweist sich als codesparender, wie das Hinzufügen einer neuen Function bei der Serverless Applikation (siehe Abb. 23).

Besonders Spring eignet sich hervorragend für unerfahrene Entwickler. Neben einer äußert ausführlichen Dokumentation gibt es viele Beispiele zu den einzelnen Modulen des Frameworks. Amazon bietet zwar auch eine gute Dokumentation für das Bauen von serverlosen Anwendungen. Allerdings sind ansonsten wenig allgemeingültige Beispiele für die Implementierung von Standardanwendungsfälle, wie zum Beispiel Datenbankzugriffen, zu finden.

Metrik	Ergebnis	
	Klassisch	Serverless
Messung der Entwicklungszeit	2,5 AT	5 AT
LoC für die Umsetzung der ersten Funktionalität	49	93
LoC für die Umsetzung einer weiteren Funktionalität	16	39
Erlernbarkeit: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	gut

Abbildung 23: Evaluation des Implementierungsaufwands

Frameworkunterstützung

Bei der Bewertung der Frameworkunterstützung spiegelt sich die Macht des Spring Frameworks wider. So wird neben Spring Boot lediglich Lombok zur weiteren Reduzierung von Boilerplate-Code eingesetzt. Auf der Serverless Seite hingegen sind zusätzlich zum AWS SDK Frameworks und Bibliotheken wie Dagger, Jackson, Lombok und Mockito notwendig. Das heißt, es können zwar alle anfallenden Aufgaben durch die Unterstützung von fremden Frameworks erfüllt werden, jedoch nicht so kompakt wie mit Spring. Dem entsprechend schneidet die Serverless Umsetzung mit mehr genutzten Frameworks in diesem Punkt schlechter ab (siehe Abb. 24).

Spring ist seit Jahren eines der führenden Frameworks zur Entwicklung von Webanwendungen. Es ist sehr ausgereift und bietet Module zur Mithilfe in vielen verschiedenen Bereichen.

Das *Serverless Framework* ist eines der bekanntesten in seiner Sparte. Es unterstützt den Deploymentprozess, sowie eine anbieterunabhängige Entwicklung. In Bezug auf das Deployment kann es als eine Art Weiterentwicklung von AWS SAM angesehen werden. Trotz einer stetigen Fortentwicklung durch eine breite Community sind manche Unausgereiftheiten aufgrund des jungen Alters noch enthalten. [Kö17, S. 185]

Metrik	Ergebnis	
	Klassisch	Serverless
Erleichterung durch Frameworks: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	gut
Anzahl genutzter Frameworks/Bibliotheken	2	5
Ausgereiftheit der gängigen Frameworks: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	gut

Abbildung 24: Evaluation der Frameworkunterstützung

Deployment

Das Deployment einer Spring Anwendung kann mit wenig Aufwand vollzogen werden. Nachdem mit Maven aus der Applikation ein `jar`-Paket erstellt wurde, kann dieses auf einen beliebigen Server verschoben und ausgeführt werden. Hierfür ist es sinnvoll ein eigenes Skript anzulegen. Die Auslieferung der Serverless Umsetzung ist sogar noch unproblematischer. Durch das Tool SAM kann das verpackte Programm direkt in die Infrastruktur des Anbieters geladen werden.

Für beide Implementierungen ist es möglich mit der Hilfe eines Tools zur *Continuous Integration* ein automatisches Deployment einzurichten. Unter der Verwendung von Jenkins oder TravisCI ist lediglich eine Konfigurationsdatei von Nöten. Diese definiert beispielsweise die benötigte Laufzeitumgebung, sowie die Zielplattform und hat in beiden Fällen einen geringen Umfang (siehe Abb. 25).

Metrik	Ergebnis	
	Klassisch	Serverless
Toolunterstützung für Deployment: Ordinalskala (verfügbar, leicht umsetzbar, schwer umsetzbar)	leicht umsetzbar	verfügbar
Automatisches Deployment: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	sehr gut
LoC in der entsprechenden Konfigurationsdatei	8	16

Abbildung 25: Evaluation des Deploymentprozesses

Testbarkeit

Die lokale Ausführung der Testfälle ist problemlos in Spring Boot möglich. Das Hochfahren eines Springkontextes für das Testen ist integraler Bestandteil des Frameworks und kann lokal durchgeführt werden. Das Testen der Serverless Applikation gestaltet sich, wie schon beschrieben, etwas problematischer. Functions können beispielsweise nur mit weiteren Tools sinnvoll lokal getestet werden. Im Codeumfang gibt es jedoch keine Unterschiede zwischen den beiden Testumsetzungen.

Bei der Ausführungsdauer der Testfälle der klassischen Webanwendung macht sich das Hochfahren des Spring Kontextes bemerkbar. Dies kostet merklich Zeit. Die Serverless Testfälle können deutlich schneller durchlaufen werden.

Testautomatisierung ist auf beiden Seiten möglich. Die automatische Durchführung der Tests vor dem Deployment ist durch eine entsprechende Konfiguration möglich. Auch die Ausführung eines Testfalls mit verschiedenen Werten ist unabhängig vom Anwendungstyp möglich.

Metrik	Ergebnis	
	Klassisch	Serverless
Lokale Testausführung: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	gut
LoC für die Umsetzung eines Testfalls	20	19
Zeitmessung der Ausführungsdauer	59 s	28 s
Testautomatisierung: Ordinalskala (verfügbar, leicht umsetzbar, schwer umsetzbar)	leicht umsetzbar	leicht umsetzbar

Abbildung 26: Evaluation der Testbarkeit

Erweiterbarkeit

Die klassische Webanwendung mit ihren Controllern, Services und Repositories lässt sich problemlos mit weiteren Funktionalitäten erweitern. Je nach Anwendungsfall muss lediglich der Controller um einen Endpunkt erweitert und der zugehörige Service angepasst

werden. Im Serverless Umfeld lässt sich die Erweiterung noch einfacher durchführen. Es muss nur eine neue Function hinzugefügt werden. Alle bisher bestehenden Functions sind von der Änderung nicht betroffen (siehe Abb. 27).

Die Wiederverwendung von bestehenden Komponenten ist in beiden Fällen, soweit der fachliche Kontext passt, jederzeit möglich. Einzelne Functions oder auch Controller mit entsprechenden Endpunkten können unabhängig von der restlichen Anwendung in verschiedenen Applikationen eingesetzt werden.

Metrik	Ergebnis	
	Klassisch	Serverless
Beeinträchtigung des bestehenden Codes bei Erweiterungen: Ordinalskala(gar nicht, wenig, stark)	wenig	gar nicht
Wiederverwendbarkeit: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	sehr gut

Abbildung 27: Evaluation zur Erweiterbarkeit

Performance

Zum Vergleich der Performance wurden zwei Messungen an den Prototypen durchgeführt. Es wurden die Antwortzeiten, sowie der Durchsatz gemessen. Zur Berechnung der Werte wurden mehrere Messungen durchgeführt, aus denen der Durchschnitt errechnet wurde. Zum Messen wurde das Tool JMeter eingesetzt. Neben der Durchführung können hiermit die Ergebnisse in den verschiedensten Arten aufbereitet werden (siehe Abb. 28).

In beiden Fällen hat die klassische Anwendung, die während der Messung auf der Plattform Heroku lief, deutlich besser abgeschnitten (siehe Abb. 29). Dies zeichnet sich nicht nur in den Messwerten ab, sondern ist auch bei der Nutzung der beiden Applikationen spürbar. Besonders ins Gewicht fällt bei der Serverless Umsetzung der sogenannte *Cold Start*. Damit ist das Hochfahren der Functions aus dem ruhenden Zustand gemeint. Dies hat eine deutlich höhere Reaktionszeit zur Folge, sodass der erste Request den Durchschnitt nach oben treibt. Ohne Cold Starts könnte beispielsweise ein doppelt so hoher Durchsatz erreicht werden.

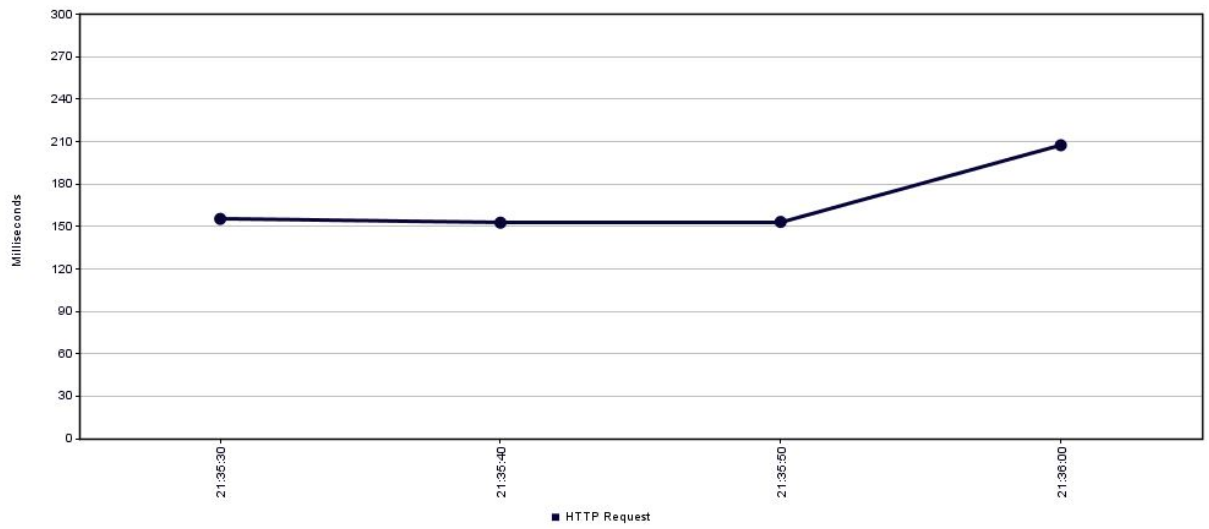

Abbildung 28: Antwortzeit der Serverless Anwendung

Dies ist insofern erstaunlich, da die AWS Umgebung sogar eine automatische Skalierung zur Verfügung stellt. Die angemessene Reaktion auf steigende Last sollte somit kein Problem darstellen. Die Spring Anwendung hingegen verfügt nicht über eine automatische Skalierung. Um hier eine Skalierbarkeit erreichen zu können, ist eine Erweiterung notwendig.

Über das Spring Boot Modul *Actuator* können Metriken zu den einzelnen Endpunkten erfasst werden. Diese werden dazu genutzt, um auf sich ändernde Bedingungen zu reagieren und je nach Fall Instanzen hinzuzufügen oder herunterzufahren. [Min18]

Anhand dieses Kriterium ist es schon absehbar, dass der Anwendungsfall nicht besonders geeignet für Serverless Anwendungen zu sein scheint. Einer der größten Vorteile, wie die automatische Skalierung, kann nicht optimal ausgenutzt werden. Allerdings konnte nicht festgestellt werden, ab wann Amazon auf Lastspitzen reagiert. Ein Fehler im Messaufbau könnte hierbei sein, dass alle Requests vom selben Host stammen. Möglicherweise ist dies für Amazon kein Grund weitere Ressourcen bereitzustellen.

Metrik	Ergebnis	
	Klassisch	Serverless
Messung der Antwortzeit	19 ms	167 ms
Messung des Durchsatzes bei 10.000 Anfragen mit 255 echt parallelen Clients	1593,8 Anfragen/sec	53,2 Anfragen/sec
Anpassung an Lastzunahme: Ordinalskala(automatisch, manuell, nicht möglich)	manuell	automatisch

Abbildung 29: Evaluation der Performanz

Sicherheit

Die Sicherung und der Schutz der Anwendung war in beiden Fällen unproblematisch. Spring bietet auf der einen Seite mit dem Modul Spring-Security, für das es viele zusätzliche Plugins gibt, die benötigte Hilfe zum Einrichten eines Rechte-Rollen Konzepts. AWS auf der anderen Seite ermöglicht mit dem IAM ein Möglichkeit die Zugriffe sehr feingranular nach Usergruppen und Rollen einzuschränken. So kann in beiden Applikationen ein Schutz vor Fremdzugriffen errichtet werden.

Auch der Schutz der Daten im Speicher ist für beide Typen möglich. Die Verschlüsselung des Datenbestandes ist ein Feature der Datenbank und wird von den meisten Modellen unterstützt. Allerdings kann es hierbei dazu kommen, dass in der Datenbank nicht mehr gesucht werden kann.

Die DynamoDb-Tabellen der Serverless Umsetzung sind bereits von Beginn an automatisch geschützt. Der hierbei verwendete Schlüssel befindet sich im Besitz der AWS DynamoDB. Ein eigener Schlüssel kann über das AWS Key Management hinzugefügt werden. Dafür fallen allerdings zusätzliche Kosten an. [Ama12]

Metrik	Ergebnis	
	Klassisch	Serverless
Zugriffsschutz: Ordinalskala (sehr gut, gut, schlecht)	sehr gut	sehr gut
Verschlüsselung der Daten: Ordinalskala (enthalten, leicht umsetzbar, schwer umsetzbar)	leicht umsetzbar	enthalten

Abbildung 30: Evaluation zur Sicherheit

3.6.2 Auswertung der Evaluation

Auf die Gegenüberstellung der beiden Anwendungen folgt nun die Auswertung der Ergebnisse. Ziel ist es die gesammelten Erkenntnisse anschaulich vergleichen und einordnen zu können. Für eine visuelle Darstellung eignet sich besonders gut ein Netzdiagramm. Die Achsen stellen dabei die sieben Kategorien dar.

> *„Die Achsen der einzelnen Qualitätskategorien stellen dabei den Gesamt-Erfüllungsgrad in Prozent dar. [Zar17, S. 40]“*

Um diesen Prozentwert zu erlangen, müssen die Ergebnisse der Evaluation in eine normalisierte Form, also Werte zwischen 0 und 1, gebracht werden. Im Fall einer dreistufigen Ordinalskala kann dies folgendermaßen aussehen.

Ergebnisse	Normalisierung
sehr gut	1
gut	0.5
schlecht	0

Tabelle 1: Normalisierung einer Ordinalskala mit 3 Werten nach [Zar17, S. 41]

Für die Messwerte kann die Normalisierung durch eine Abstufung der Ergebnisse erreicht werden. Für die Darstellung der Anzahl der genutzten Frameworks innerhalb der Anwendung kann beispielsweise folgende Normalisierung eingesetzt werden.

Ergebnisse	Normalisierung
1	1
2	0.8
3	0.6
4	0.4
5	0.2
≥ 6	0

Tabelle 2: Normalisierung der Anzahl genutzter Frameworks nach [Zar17, S. 41]

Die daraus resultierenden Diagramme bieten einen Überblick über das Abschneiden der beiden Prototypen. Je näher die Werte der einzelnen Kategorien an 100% liegen, desto reibungsloser war das Erfüllen dieser Anforderung möglich.

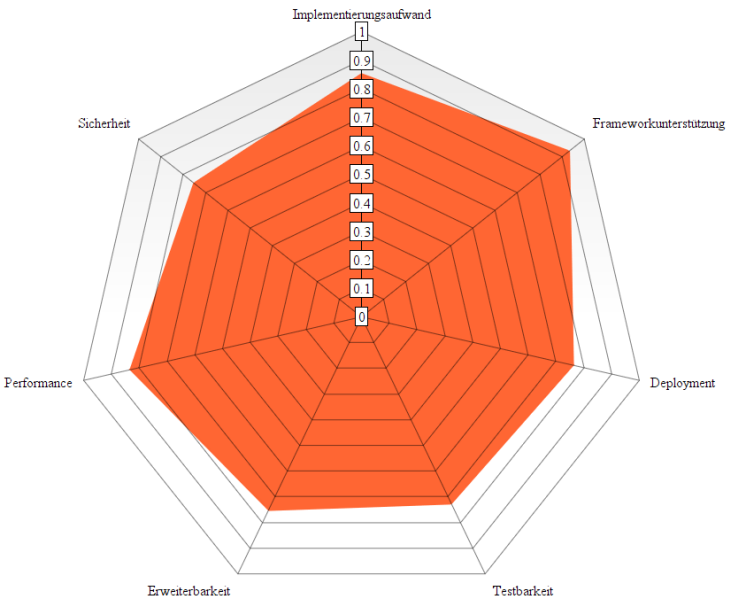

Abbildung 31: Netzdiagramm für die Evaluation der klassischen Anwendung

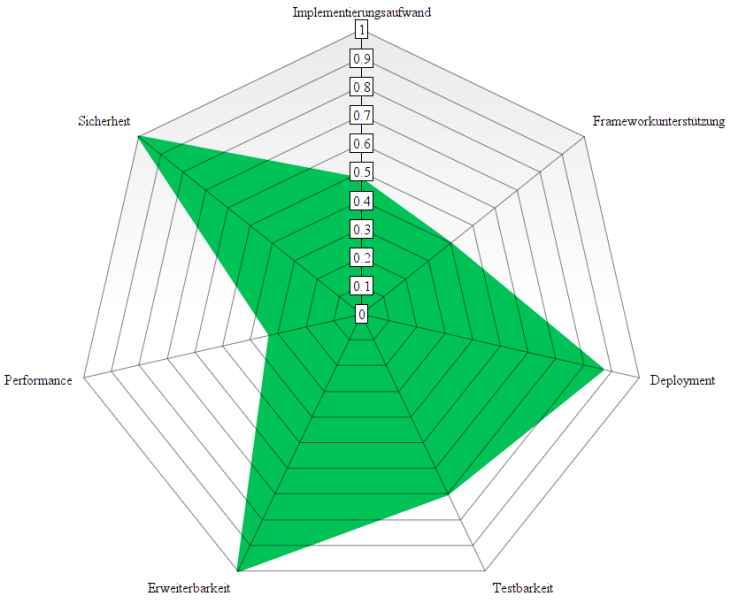

Abbildung 32: Netzdiagramm für die Evaluation der Serverless Anwendung

Im Bezug auf den Implementierungsaufwand und die Frameworkunterstützung ist das noch junge Alter der Serverless Bewegung und die damit einhergehende Unausgereiftheit zu erkennen. Hier liegt die klassische Webanwendung mit dem etablierten Spring Framework im Hintergrund deutlich vorne. Auch die Aufteilung der Anwendungsfälle in viele Functions sorgt für einen erhöhten Implementierungsaufwand.

Auf der anderen Seite spiegelt sich die Trennung der Funktionalitäten in der guten Bewertung der Erweiterbarkeit wieder. Das Deployment und die Erstellung eines angemessenen Sicherheitskontext schneiden beim Serverless Prototypen ebenfalls besser ab. Hier glänzt vor allem Amazon, beziehungsweise allgemein die Anbieter mit einem breiten Angebot an zusätzlichen Services.

Enttäuschend hingegen ist das Abschneiden der Serverless Applikation hinsichtlich der Performanzmessungen. Die einfachen Anfragen haben das genaue Gegenteil der Serverless Idee bezweckt. Anstatt flexibel auf Änderungen bezüglich der Last reagieren zu können, hat das Hochfahren immer neuer Functions für einen erhöhten Overhead gesorgt. Dies schlägt sich deutlich in den Antwortzeiten nieder.

4 Vergleich der beiden Umsetzungen

Es ist zu erkennen, dass der hier ausgewählte Anwendungsfall nicht optimal für eine Serverless Umsetzung ist. Viele Vorteile, die das Serverless Wesen mit sich bringt, konnten nicht ausgeschöpft werden. Trotz alledem gibt es Szenarien, welche die Vorzüge besser ausnutzen. Dies sind zum Beispiel die Abarbeitung rechenintensiver Aufgaben oder die Reaktion auf asynchrone Events.

Im Folgenden werden die Vor- und Nachteile des Serverless Computings beschrieben, um anhand derer und dem Ergebnis der Evaluation sinnvolle Einsatzmöglichkeiten für Serverless Anwendungen ableiten zu können.

4.1 Vorteile der Serverless Infrastruktur

Sobald die Einarbeitung in das Serverless Umfeld vollzogen ist und die ersten Versuche getätigt wurden, kann durchaus von einer einfachen Nutzung gesprochen werden. Viele Aufgaben werden vom Provider abgenommen und automatisch erledigt. Für weitere Tätigkeiten existieren oftmals bereits vorkonfigurierte Services. [Sti17, S. 7]

Zu diesen automatisch erledigten Arbeiten zählt beispielsweise die Skalierung, das Kapazitätsmanagement, sowie die Einhaltung einer gewissen Fehlertoleranz. Dies führt dazu, dass der Programmierer sich keine Gedanken über diese Problemstellungen machen muss. [Bü17]

Eine weitere Aufgabe, die der Anbieter übernimmt, ist das Beheben von Sicherheitslücken in der Infrastruktur. Es liegt somit nicht mehr im Aufgabenfeld des Unternehmens die Server aktuell zu halten, geschweige denn überhaupt zu betreiben. Dies schließt allerdings mögliche Sicherheitsmängel an der Anwendung nicht aus. So können sich zum Beispiel jederzeit Sicherheitslücken über die Functions in die Applikation einschleichen. [Pod18]

Ebenso ein Vorteil in Bezug auf eine leichte Handhabung ist die steigende Flexibilität. Nachdem lediglich der Anwendungscode implementiert werden muss, kann das Augenmerk komplett auf die Anforderungen gelegt werden. Dies führt zu einer kurzen Zeitspanne zwischen der Idee und der Veröffentlichung auf dem Markt, da die Implementierung der Functions kompakt gehalten werden kann. Serverless Computing eignet sich somit auch hervorragend für das Erstellen von ersten Prototypen. [Kö17, S. 32]

Auch eine anpassungsfähige Ressourcenverwaltung ist Teil der Flexibilität im Serverless Umfeld. Je nach Bedarf können schnell die benötigten Ressourcen von Seiten des Anbieters bereitgestellt werden. Dies führt zu einer nutzungsabhängigen Abrechnung der Ressourcen, sodass keine Leerlaufzeiten vom Kunden bezahlt werden müssen. [Bü17]

Das Starten beliebig vieler Kopien einer Function wird durch die Zustandslosigkeit von Serverless Applikationen ermöglicht. Da sich alle Instanzen gleich verhalten, spielt die Umgebung der Ausführung, sowie der aktuelle Nutzerkontext keine Rolle. Nachteilig hieran ist, dass der Zustand anderweitig verwaltet werden muss. Ist es der Fall, dass dieser über mehrere Funktionsaufrufe bestehen muss, entsteht ein Mehraufwand für den Entwickler. Dieses Problem lässt sich auch in zustandslosen klassischen Webanwendungen wiederfinden. [Kö17, S. 35]

Letztendlich wird also ein Großteil der Entwicklungs- und Betriebsaufgaben durch den Anbieter abstrahiert. Mike Roberts beschreibt die Vereinfachung des Betriebsmanagement in seinem Artikel zu Serverless Architekturen folgendermaßen:

> *„[...] a fully Serverless solution requires zero system administration. [Rob18]“*

Die hier erläuterten Infrastrukturvorteile können auch im Falle einer klassischen Webanwendung erreicht werden. Hierzu muss für die Applikation eine fremd gehostete Infrastruktur genutzt werden.

4.2 Nachteile der Serverless Infrastruktur

Zuallererst steht hier die Dezentralisierung. Hiermit ist gemeint, dass jedes Serverless Modul lediglich für eine Sache verantwortlich ist. Dies führt zu einer großen Modullandschaft bestehend aus Functions und weiteren Services, die nicht zentral verwaltet werden können und somit für eine erhöhte Komplexität sorgen. So dauerte die Implementierung des Serverless Prototypen beispielsweise länger wie die des Klassischen, obwohl lediglich die Anwendungslogik implementiert werden musste. Allerdings bietet dieses Konzept wiederum auch einen Gewinn für den Nutzer. [Kö17, S. 35]

> *„Letztendlich bietet ein Aspekt dieser Dezentralisierung aber den Vorteil, dass unsere Ressourcen auch in der Cloud flexibel zu betreiben sind und so eine hohe Ausfallsicherheit und Verfügbarkeit mit sich bringen. [Kö17, S. 35]“*

Ebenfalls negativ während der Evaluation sind die Cold Starts aufgefallen. Diese treten bei der Instanziierung einer Function auf. Muss die Plattform eine neue Instanz erzeugen, so kostet dies Zeit. Die genaue Zeitdauer ist dabei abhängig von der Programmiersprache, der Anzahl der verwendeten Bibliotheken, der Konfiguration der Laufzeitumgebung und auch von der Verwendung anderer Services. Zur Eindämmung dieses Effekts können *Keep-alives* eingesetzt werden. Diese wecken die Functions periodisch auf und verhindern somit das „Einschlafen“ der Function, woraufhin diese nicht wieder komplett hochgefahren werden muss. Im Umkehrschluss erzeugt dies allerdings auch wieder Kosten, da der Ressourcenverbrauch die nutzungsabhängige Abrechnung nach oben treibt. [Rob18]

Ein weiterer Nachteil in der Kategorie Performanz sind erhöhte Latenzen. Da kein Einfluss auf die Abarbeitung der Requests und Events genommen werden kann, ist im Voraus nicht absehbar, wie lange auf eine Antwort gewartet werden muss. Durch die Skalierung wird jedoch versucht, die Latenzzeiten so gering wie möglich zu halten. [Kö17, S.35-36]

Des Weiteren können Timeouts die Ausführung einer Function beeinträchtigen. Diese sind vom Anbieter vorgegeben, um eine zu lange Ausführung, in der die Function Ressourcen verbraucht, zu verhindern. [Ash17]

Testen ist ebenfalls schwieriger in der Serverless Infrastruktur. Da in einer lokalen Testumgebung die Cloud nicht komplett nachgestellt werden kann, wird der Entwickler dazu gezwungen in der Cloud zu testen. Dies kostet zusätzliche Ressourcen. [Rob18]

Ungewollt in allen Bereichen der Softwareentwicklung ist die Abhängigkeit von anderen Stellen. In diesem Fall besteht eine Abhängigkeit zum Anbieter des Serverless Dienstes. Dieser hält die Kontrolle über die Anwendung. Ruhezeiten der Server, Requestlimits, Preisfestlegung oder das Betreiben von zusätzlichen Services liegen in der Macht des Providers und können nicht vom Entwickler beeinflusst werden. [Rob18]

Durch den fehlenden Zugriff auf die Infrastruktur gestalten sich Aufgaben wie Debugging oder Monitoring schwierig. Die Anbieter ermöglichen lediglich die Aufzeichnung grundlegender Systemdaten. Durch die Kurzlebigkeit der Container, in denen die Functions ausgeführt werden, stellt die Sammlung umfangreicher Informationen zur Ausführung einen zusätzlichen Aufwand dar. [Rob18]

4.3 Abwägung sinnvoller Einsatzmöglichkeiten

Es ist also festzustellen, dass nicht alle Eigenschaften der Serverless Infrastruktur eindeutig in Vor- oder Nachteil kategorisiert werden können. So zum Beispiel der Aspekt der Dezentralisierung, der eine Einschränkung, jedoch auch einen Gewinn darstellen kann. Genauso verhält es sich bei der Eingrenzung von sinnvollen Einsatzmöglichkeiten für Serverless Umsetzungen. Manche Anwendungsfälle eignen sich besser für den Serverless Betrieb als andere. Prinzipiell können alle Applikationen, die nicht zustandsbehaftet sind, in der Serverless Welt umgesetzt werden.

Durch die Möglichkeit individuelle Requests in relativ kurzer Zeit abarbeiten zu können, sind Serverless Plattformen besser für Anwendungen mit Wertsetzung auf einen hohen Durchsatz wie auf kurze Latenzzeiten geeignet. [AC17]

Außerdem verhalten sich Serverless Functions besonders effizient, wenn sie für CPU-gebundene Berechnungen eingesetzt werden. Für I/O-gebundene Funktionen hingegen

ist es billiger diese auf VMs zu betreiben. Des Weiteren verhalten sich langlebige Anwendungen sehr ineffizient im Serverless Betrieb, da die Infrastruktur auf die Ausführung von kurzlebigen Containern ausgelegt ist. [BCC+17]

Mit der Einschränkung, dass die Performanztests unter korrekten Rahmenbedingen durchgeführt wurden, lässt das schlechte Abschneiden des Serverless Prototypen darauf schließen, dass der hier ausgewählte Anwendungsfall eher ungeeignet für die Serverless Entwicklung ist. Zwar gehören Webanwendungen durchaus zu den Anwendungstypen, die auch im Serverless Umfeld einfach umgesetzt werden können, jedoch kam hierbei die Leichtgewichtigkeit der Functions nicht zum greifen. [Kö17, S. 42]

Besser geeignet sind Aufgaben, dessen Fokus auf der Berechnung und Verarbeitung von Daten liegt. Prädestiniert hierfür sind *Big Data* Anwendungsfälle, bei denen die Verarbeitung einer großen Menge an Daten abgebildet werden muss. Ebenfalls passend sind Lambda Functions für die Bild- und Videoverarbeitung. Die Functions können dabei die rechenintensive Kodierung beziehungsweise Bearbeitung des Bildmaterials während dem Hochladen durchführen, sodass die Daten bei einer späteren Verwendung nicht neu berechnet werden müssen. [Kö17, S. 43-44]

AWS Lambda wird beispielsweise für Amazon-Echo, auch bekannt unter dem Namen *Alexa* eingesetzt. Die erkannten Sprachbefehle dienen als Event, um eine Lambda Function zu aktivieren. Die berechnete Antwort wird wieder an das Gerät zurückgegeben und als Sprachnachricht dem Nutzer übermittelt. Auch der textbasierte Anwendungsfall ist gut für eine Umsetzung im Serverless Bereich geeignet. Die auftretenden Nachrichten in einem Chatbot können asynchron durch die Functions abgearbeitet werden. [Kö17, S. 45-46]

Es ist also eine Richtung, in der die Serverless Entwicklung sinnvoll ist, zu erkennen. Grundsätzlich ist bei allen Serverless Implementierungen die Frage nach dem richtigen Schnitt der Functions präsent. Je nach Projekt muss eine passende Trennung gefunden, beziehungsweise die Entscheidung getroffen werden, ob eine Aufteilung in Functios Sinn ergibt. [Kö17, S. 36]

Wird die Entwicklung im Serverless Umfeld in Betracht gezogen, so sollte die Abhängigkeit von einem Anbieter nicht zu hoch bewertet werden. Trotz einer gewissen Verflechtung mit dem Provider liefert dieser eine Menge an wertvoller Unterstützung. Adrian Cockcroft versucht die Kritik am sogenannten Vendor-Lock-in und der damit einhergehenden Festlegung auf einen Anbieter zu entkräften. Er stellt hierzu folgende These auf.

> *„[...] selbst die zwei- oder gar dreimalige Reimplementierung ein und derselben Fachlichkeit auf unterschiedlichen Cloud-Plattformen sei immer noch schneller und günstiger, als das gesamte Backend proprietär selbst zu entwickeln und zu*

betreiben. [Rö17b]“

5 Fazit und Ausblick

Durch die Entwicklung der beiden prototypischen Anwendungen konnte gezeigt werden, dass der ausgewählte Anwendungsfall auch im Serverless Umfeld angemessen umgesetzt werden kann. Für den Nutzer ist es nicht zu erkennen, welche Backendimplementierung hinter der Benutzeroberfläche eingesetzt wird.

Der Evaluationsprozess brachte dann die Unterschiede der beiden Umsetzungen hervor. Aus diesen konnten sinnvolle Einsatzmöglichkeiten, wie das Abarbeiten von rechenintensive Aufgaben oder das Reagieren auf Events, abgeleitet werden.

Deutlich wurde auch, dass sich das ganze Umfeld noch im Wachstum befindet. Für viele problematische Bereiche, wie das Testen von Serverless Anwendungen, gibt es noch nicht die idealen Lösungen. Im Laufe der Zeit werden hier jedoch immer mehr unterstützende Tools entstehen. Auch die Entwickler werden ein Verständnis für Serverless Umsetzungen entwickeln. [Rob18]

Diese Weiterentwicklung wird dabei helfen die Vorteile der Serverless Infrastruktur noch besser ausnutzen zu können. Optimal wäre es hier, wenn alle zustandslosen Applikationen sinnvoll Serverless betrieben werden können. Ein Schritt in diese Richtung könnte beispielsweise die Bildung eines Ansatzes für die Migration von bestehenden nicht serverlosen Anwendungen hinzu Serverless Applikationen sein. [Rob18]

Einige weitere Aspekte konnten in der Arbeit nicht genauer betrachtet werden. Hierzu zählt die Möglichkeit zwischen verschiedenen Providern zu wechseln. Hierfür bietet das umfangreiche Serverless Framework die Gelegenheit den Code so anbieterunabhängig wie möglich zu halten. Auch der On-Premise Betrieb einer Serverless Infrastruktur ist machbar. Diese entspricht allerdings nicht mehr zu 100% dem Serverless Gedanken, da somit wieder Server verwaltet und in Stand gehalten werden müssen.

Abschließend lässt sich sagen, dass Serverless Computing hervorragend für das Erstellen von ersten Prototypen geeignet ist. Umsetzungen von Webanwendungen im großen Stile sind nach dem aktuellen Stand allerdings noch sehr gewagt, da viele offene Punkte nicht einheitlich abgedeckt sind. Wird die Implementierung hingegen im klassischen Sinne durchgeführt, kann auf einer guten Basis, sowie einer Menge Expertenwissen aufgebaut werden.

6 Quellenverzeichnis

[A+09] Armbrust, Michael u. a.: Above the Clouds: A Berkeley View of Cloud Computing. (2009). http://d1smfj0g31qzek.cloudfront.net/abovetheclouds.pdf. – Zuletzt Abgerufen am 09.01.2019

[AC17] Adzic, Gojko ; Chatley, Robert: Serverless Computing: Economic and Architectural Impact. (2017). https://doi.org/10.1145/3106237.3117767. – Zuletzt Abgerufen am 10.09.2018

[Ama12] Amazon: Was ist Amazon DynamoDB? (2012). https://docs.aws.amazon.com/de_de/amazondynamodb/latest/developerguide/. – Zuletzt Abgerufen am 25.02.2019

[Ash17] Ashwini, Amit: Everything You Need To Know About Serverless Architecture. (2017). https://medium.com/swlh/everything-you-need-to-know-about-serverless-architecture-5cdc97e48c09. – Zuletzt Abgerufen am 28.08.2018

[Bü17] Büst, René: Serverless Infrastructure erleichtert die Cloud-Nutzung. (2017). https://www.computerwoche.de/a/serverless-infrastructure-erleichtert-die-cloud-nutzung,3314756. – Zuletzt Abgerufen am 28.08.2018

[Bac18] Bachmann, Andreas: Wie Serverless Infrastructures mit Microservices zusammenspielen. (2018). https://blog.adacor.com/serverless-infrastructures-in-cloud_4606.html. – Zuletzt Abgerufen 09.11.2018

[BCC+17] Baldini, Ioana ; Castro, Paul ; Chang, Kerry ; Cheng, Perry ; Fink, Stephen ; Ishakian, Vatche ; Mitchell, Nick ; Muthusamy, Vinod ; Rabbah, Rodric ; Slominski, Aleksander ; Suter, Philippe: Serverless Computing: Current Trends and Open Problems. (2017). https://arxiv.org/abs/1706.03178. – Zuletzt Abgerufen am 10.09.2018

[Boy17] Boyd, Mark: Serverless Architectures: Five Design Patterns. (2017). https://thenewstack.io/serverless-architecture-five-design-patterns/. – Zuletzt Abgerufen am 10.01.2019

[Bra18] Brandt, Mathias: Cash Cow Cloud. (2018). https://de.statista.com/infografik/13665/amazons-operative-ergebnisse/. – Zuletzt Abgerufen am 01.12.2018

[Cha17] Chakraborty, Suhel: Dagger2 Modules, Components and SubComponents,

a Complete Story. (2017). https://medium.com/@suhelchakraborty/dagger-2-modules-components-and-subcomponents-a-complete-story-part-i-1f484de3b15. – Zuletzt Abgerufen am 21.02.2019

[CN12] Conde, Carlos ; Narin, Attila: Development and Test on Amazon Web Services. (2012). https://d1.awsstatic.com/whitepapers/aws-development-test-environments.pdf. – Zuletzt Abgerufen am 03.03.2019

[Dja02] Djabarian, Ebrahim: *Die strategische Gestaltung der Fertigungstiefe*. Deutscher Universitätsverlag, 2002. – ISBN 9783824476602

[FIMS17] Fox, Geoffrey C. ; Ishakian, Vatche ; Muthusamy, Vinod ; Slominski, Aleksander: Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. (2017). https://arxiv.org/abs/1708.08028. – Zuletzt Abgerufen am 10.09.2018

[FL14] Fowler, Martin ; Lewis, James: Microservices. (2014). https://martinfowler.com/articles/microservices.html. – Zuletzt Abgerufen 19.11.2018

[Gar99] Garfinkel, Simson L.: *Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT*. The MIT Press, 1999. – ISBN 9780262071963

[Gig18] Giglione, Marco: Unit and Integration Tests in Spring Boot. (2018). https://dzone.com/articles/unit-and-integration-tests-in-spring-boot-2. – Zuletzt Abgerufen am 13.02.2019

[Har02] Hartmann, Anja K.: *Dienstleistungen im wirtschaftlichen Wandel: Struktur, Wachstum und Beschäftigung*. 2002 http://nbn-resolving.de/urn/resolver.pl?urn=https://nbn-resolving.org/urn:nbn:de:0168-ssoar-121435

[Hef16] Hefnawy, Eslam: Serverless Code Patterns. (2016). https://serverless.com/blog/serverless-architecture-code-patterns/. – Zuletzt Abgerufen am 10.01.2019

[Her18] Heroku: Heroku Security. (2018). https://www.heroku.com/policy/security. – Zuletzt Abgerufen 08.11.2018

[Inc18] Inc., Serverless: Serverless Guide. (2018). https://github.com/serverless/guide. – Zuletzt Abgerufen am 06.09.2018

[Inf18] Inflectra: Software Testing Methodologies. (2018). https://www.

inflectra.com/Ideas/Topic/Testing-Methodologies.aspx. – Zuletzt Abgerufen am 13.02.2019

[Kö17] Köbler, Niko: *Serverless Computing in der AWS Cloud*. entwickler.press, 2017. – ISBN 9783868028072

[Kar18] Karia, Bhavya: A quick intro to Dependency Injection: what it is, and when to use it. (2018). https://medium.freecodecamp.org/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f. – Zuletzt Abgerufen am 12.02.2019

[Kra18] Kratzke, Nane: A Brief History of Cloud Application Architectures. (2018). https://doi.org/10.3390/app8081368. – Zuletzt Abgerufen am 22.11.2018

[Kru04] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. Addison-Wesley, 2004. – ISBN 0321197704

[KS17] Klingholz, Lukas ; Streim, Anders: Cloud Computing. (2017). https://www.bitkom.org/Presse/Presseinformation/Nutzung-von-Cloud-Computing-in-Unternehmen-boomt.html. – Zuletzt Abgerufen am 01.12.2018

[Lit17] Litzel, Nico: Was ist NoSQL? (2017). https://www.bigdata-insider.de/was-ist-nosql-a-615718/. – Zuletzt Abgerufen am 21.02.2019

[Mar15] Maresca, Paolo: From Monolithic Three-Tiers Architectures to SOA vs Microservices. (2015). https://thetechsolo.wordpress.com/2015/07/05/from-monolith-three-tiers-architectures-to-soa-vs-microservices/. – Zuletzt Abgerufen am 11.02.2019

[MG11] Mell, Peter ; Grance, Tim: The NIST Definition of Cloud Computing. (2011). https://csrc.nist.gov/publications/detail/sp/800-145/final. – Zuletzt Abgerufen am 03.11.2018

[Min18] Minkowski, Piotr: Spring Boot Autoscaler. (2018). https://dzone.com/articles/spring-boot-autoscaler. – Zuletzt Abgerufen am 27.02.2019

[Pod18] Podjarny, Guy: Hey cool, you went serverless. Now you just have to worry about all those stale functions. (2018). https://www.theregister.co.uk/2018/05/15/stale_serverless_functions/. – Zuletzt Abgerufen am 28.08.2018

[Pol18] Polymer, Project: Data binding. (2018). https://polymer-library.

polymer-project.org/2.0/docs/devguide/data-binding. – Zuletzt Abgerufen am 19.02.2019

[Rö17a] Röwekamp, Lars: Serverless Computing, Teil 1: Theorie und Praxis. (2017). https://www.heise.de/developer/artikel/Serverless-Computing-Teil-1-Theorie-und-Praxis-3756877.html?seite=all. – Zuletzt Abgerufen am 30.08.2018

[Rö17b] Röwekamp, Lars: Serverless Computing, Teil 2: Pro und contra. (2017). https://www.heise.de/developer/artikel/Serverless-Computing-Teil-2-Pro-und-contra-3757049.html. – Zuletzt Abgerufen am 30.08.2018

[Ric15] Richards, Mark: *Software Architecture Patterns*. O'Reilly, 2015. – ISBN 9781491924242

[Rob18] Roberts, Mike: Serverless Architectures. (2018). https://martinfowler.com/articles/serverless.html. – Zuletzt Abgerufen am 22.02.2019

[RPMP17] Rai, Gyanendra ; Pasricha, Prashant ; Malhotra, Rakesh ; Pandey, Santosh: Serverless Architecture: Evolution of a new paradigm. (2017). https://www.globallogic.com/gl_news/serverless-architecture-evolution-of-a-new-paradigm/. – Zuletzt Abgerufen am 30.08.2018

[Sch16] Schmidt, Christopher: Webcompontents mit Polymer - Teil 1: Von 0.5 zu 1.x. (2016). https://www.innoq.com/de/articles/2016/07/web-components-mit-polymer%E2%80%93teil-1/. – Zuletz Abgerufen am 19.02.2019

[Sti17] Stigler, Maddie: *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Apress, 2017. – ISBN 9781484230831

[Swa18] Swarup, Pulkit: Microservices: Asynchronous Request Response Pattern. (2018). https://medium.com/@pulkitswarup/microservices-asynchronous-request-response-pattern-6d00ab78abb6. – Zuletzt Abgerufen am 09.01.2019

[Tiw16] Tiwari, Abhishek: Stored Procedure as a Service (SPaaS). (2016). https://www.abhishek-tiwari.com/stored-procedure-as-a-service-spaas/. – Zuletzt Abgerufen am 30.11.2018

[Tur18] Turvin, Neil: Serverless vs. Microservices: What you need to know for cloud.

(2018). https://www.computerweekly.com/blog/Ahead-in-the-Clouds/Serverless-vs-Microservices-What-you-need-to-know-for-cloud. – Zuletzt Abgerufen 15.11.2018

[WJ16] Wagner, Ruben ; Jost, Simon: Webcompontents mit Polymer - Teil 2: Technische Anwendung. (2016). https://www.innoq.com/de/articles/2016/09/web-components-mit-polymer%E2%80%93teil-2/. – Zuletz Abgerufen am 19.02.2019

[Wol13] Wolff, Eberhard: Spring Boot - was ist das, was kann das? (2013). https://jaxenter.de/spring-boot-2279. – Zuletzt Abgerufen am 12.02.2019

[Zar17] Zarwel, René: *Microservices und technologische Heterogenität: Entwicklung einer sprachunabhängigen Microservice Framework Evaluationsmethode*. 2017

Anhang

A Vollständige Abbildung der Bewertungskriterien