



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: 09.03.19

I Kurzfassung

1 Das Ziel der Arbeit ist es, Unterschiede in der Entwicklung von Serverless und klassischen
2 Webanwendungen zu betrachten. Es soll ein Leitfaden entstehen, der Entwicklern und
3 IT-Unternehmen die Entscheidung zwischen klassischen und Serverless Anwendungen er-
4 leichtert. Dazu wird zuerst eine Einführung in die Entwicklung des Cloud Computings und
5 insbesondere in das Themenfeld des Serverless Computing gegeben. Im nächsten Schritt
6 werden zwei beispielhafte Anwendungen entwickelt. Zum einen eine klassische Weban-
7 wendung mit der Verwendung des Spring Frameworks im Backend und einem Javascript
8 basiertem Frontend und zum anderen eine Serverless Webanwendung. Hierbei werden
9 die Besonderheiten im Entwicklungsprozess von Serverless Applikationen hervorgehoben.
10 Abschließend werden die beiden Vorgehensweisen mittels vorher festgelegter Kriterien
11 gegenübergestellt, sodass sinnvolle Einsatzmöglichkeiten für Serverless Anwendungen ab-
12 geleitet werden können.

13	II Inhaltsverzeichnis	
14	I Kurzfassung	I
15	II Inhaltsverzeichnis	II
16	III Abbildungsverzeichnis	III
17	IV Tabellenverzeichnis	III
18	V Listing-Verzeichnis	III
19	VI Abkürzungsverzeichnis	IV
20	1 Einführung und Motivation	1
21	2 Grundlagen der Serverless Architektur	3
22	2.1 Historische Entwicklung des Cloud Computings	3
23	2.1.1 Grundlagen des Cloud Computings	6
24	2.1.2 Abgrenzung zu PaaS	8
25	2.1.3 Abgrenzung zu Microservices	9
26	2.2 Eigenschaften von Function-as-a-Service	11
27	2.3 Allgemeine Pattern für Serverless Umsetzungen	12
28	2.3.1 Serverless Computing Manifest	13
29	2.3.2 Schnittstellen zu anderen Architekturen	15
30	3 Entwicklung einer prototypischen Anwendung	17
31	3.1 Vorgehensweise beim Vergleich der beiden Anwendungen	17
32	3.2 Fachliche Beschreibung der Beispiel-Anwendung	19
33	3.3 Implementierung der Benutzeroberfläche	20
34	3.4 Implementierung der klassischen Webanwendung	24
35	3.4.1 Architektonischer Aufbau der Applikation	24
36	3.4.2 Implementierung der Anwendung	27
37	3.4.3 Testen der Webanwendung	34
38	3.5 Implementierung der Serverless Webanwendung	37
39	3.5.1 Architektonischer Aufbau der Serverless Applikation	37
40	3.5.2 Implementierung der Anwendung	39
41	3.5.3 Testen von Serverless Anwendungen	44
42	3.6 Unterschiede in der Entwicklung	47
43	3.6.1 Implementierungsvorgehen	49
44	3.6.2 Testen der Anwendung	49
45	3.6.3 Deployment der Applikation	49
46	3.6.4 Wechsel zwischen Providern	49
47	4 Vergleich der beiden Umsetzungen	49
48	4.1 Vorteile der Serverless Infrastruktur	49
49	4.2 Nachteile der Serverless Infrastruktur	49

50	4.3 Abwägung sinnvoller Einsatzmöglichkeiten	49
51	5 Fazit und Ausblick	49
52	6 Quellenverzeichnis	50
53	Anhang	I
54	A Vollständige Abbildung der Bewertungskriterien	I

III Abbildungsverzeichnis

56	Abb. 1 Anteil der Unternehmen, die Cloud Dienste nutzen [KS17]	1
57	Abb. 2 Operativer Gewinn von Amazon [Bra18]	2
58	Abb. 3 Zusammenhang Kenntnisstand und Kontroll-Level [Bü17]	4
59	Abb. 4 Hierarchie der Cloud Services [Kö17, S. 28]	5
60	Abb. 5 Historische Entwicklung des Cloud Computings	6
61	Abb. 6 Verantwortlichkeiten der Organisation nach [Rö17]	7
62	Abb. 7 Aufgabenverteilung: IaaS vs. PaaS vs. Serverless [Bü17]	9
63	Abb. 8 FaaS Beispiel Anwendung [Tiw16]	12
64	Abb. 9 Under- und Overprovisioning [A ⁺ 09, S. 11]	14
65	Abb. 10 Zusammenhang zwischen Event-driven Computing, FaaS und Serverless [FIMS17, S. 5]	16
66	Abb. 11 Request Response Pattern [Swa18]	16
67	Abb. 12 Maske: Bücherausleihe	23
68	Abb. 13 Maske: Bücherverwaltung	23
69	Abb. 14 (1) Dialog: Buch bearbeiten, (2) Dialog: Buch löschen, (3) Dialog:	
70	Buch hinzufügen	24
71	Abb. 15 3-Tier Architektur	25
72	Abb. 16 Layered Architektur nach [Ric15, S. 3]	27
73	Abb. 17 Beziehung zwischen User und Book	29
74	Abb. 18 Layered Architektur in Spring	30
75	Abb. 19 API Gateway	38
76	Abb. 20 Datenbankevent ruft Function auf	39
77	Abb. 21 Lambda Console	44
78	Abb. 22 Ausschnitt der Fragen mit entsprechenden Metriken	48

IV Tabellenverzeichnis

V Listing-Verzeichnis

81		
82	1 Binding eines Textes [Pol18]	21
83	2 Auflistung der Elemente eines Arrays	22
84	3 Einstiegsklasse für Spring Boot Anwendung	28
85	4 Repository für die Tabelle User	30

86	5	Beispiel BookController	32
87	6	Implementierung des UserDetailsService	33
88	7	Abfrage des authentifizierten Users	33
89	8	PreAuthorize an einem Endpunkt im Controller	34
90	9	Testfall im StatisticControllerTest	36
91	10	Integrationstest für eine Methode aus dem Bookservice	37
92	11	Request Handler für Lambda Function	40
93	12	Ressourcendefinition der Beispiel Function	40
94	13	Modul zur Bereitstellung der Datenbankverbindung	41
95	14	GetBookFunction	42
96	15	Modul zur Bereitstellung der Datenbankverbindung	43
97	16	Ausschnitt des StatisticDao Tests	45
98	17	Ausschnitt des StatisticDao Integrationstests	47

99 VI Abkürzungsverzeichnis

100	AWS	Amazon Web Services
101	IaaS	Infrastructure as a Service
102	PaaS	Platform as a Service
103	FaaS	Function as a Service
104	NIST	National Institute of Standards and Technology
105	BaaS	Backend as a Service
106	SaaS	Software as a Service
107	SDK	Software Development Kit
108	ORM	Object-relational mapping
109	JPA	Java Persistence API
110	DI	Dependency Injection
111	SAM	Serverless Application Model
112	DAO	Data Access Object
113	IAM	Identity and Access Management

1 Einführung und Motivation

Durch das enorme Wachstum des Internets werden immer mehr Dienstleistungen über das Netz 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 hat Cloud Computing eine rasante Entwicklung genommen. So ist beispielsweise der Anteil der deutschen Unternehmen, die Cloud Dienste nutzen, in den letzten Jahren stetig gestiegen. Mittlerweile sind es bereits zwei Drittel der Unternehmen (siehe Abb. 1).

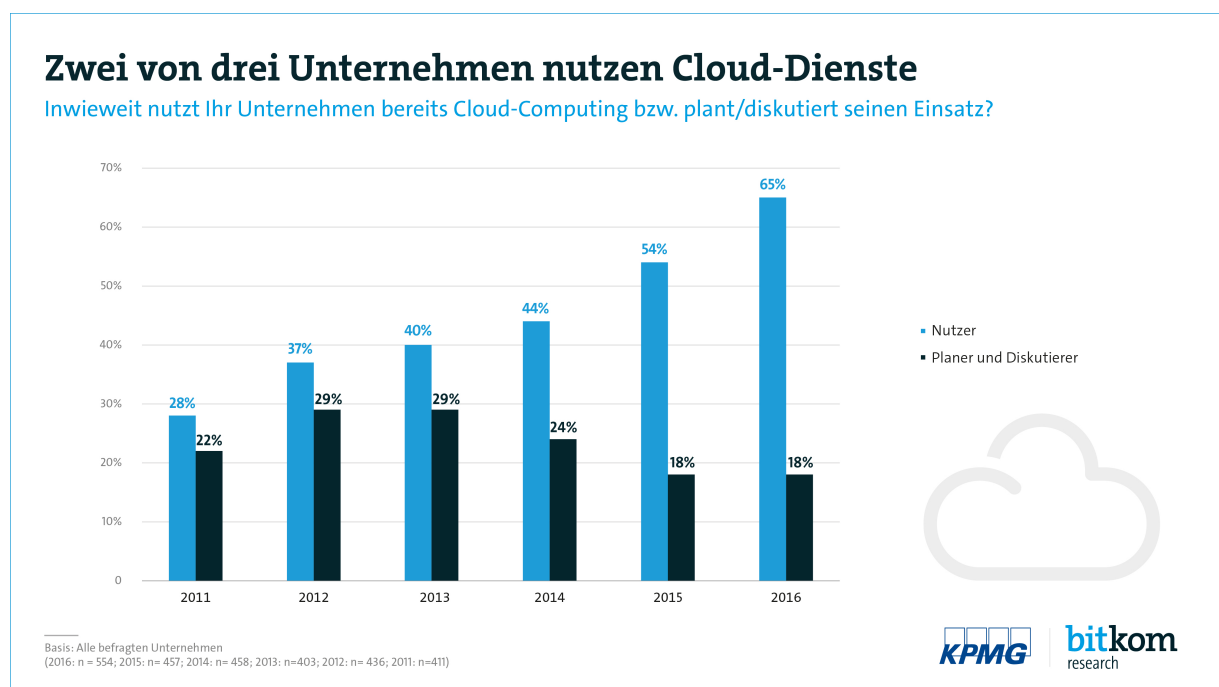


Abbildung 1: Anteil der Unternehmen, die Cloud Dienste nutzen [KS17]

Auf der Seite 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).

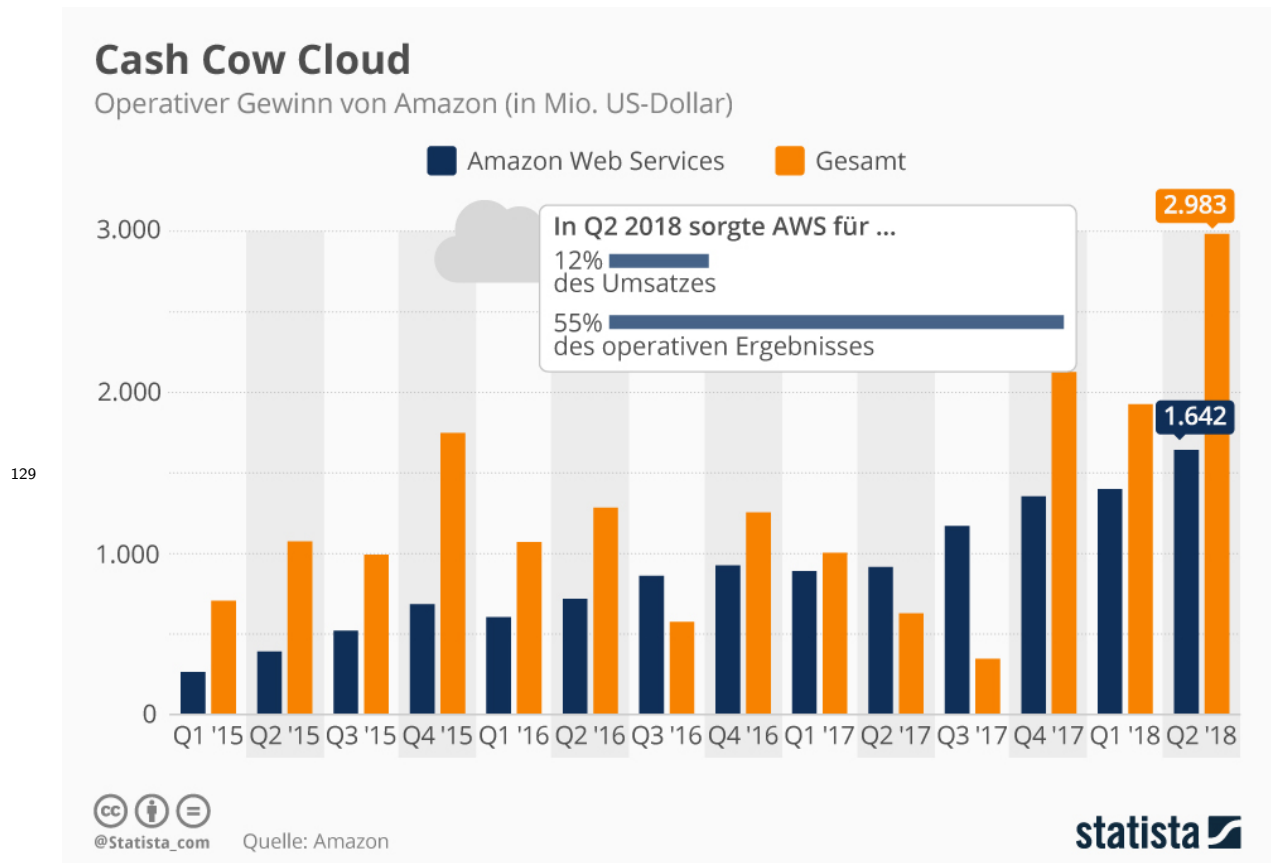


Abbildung 2: Operativer Gewinn von Amazon [Bra18]

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 Tätigkeiten können dabei vom Prototyping und kleineren Hilfsaufgaben bis hin zur Entwicklung kompletter Anwendungen gehen. [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 serverless umzusetzen.

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 sowie das Vorgehen zum Vergleich der beiden Anwendungen festgelegt (Kapitel 3.1). 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 mittels der vorher erarbeiteten Kriterien Unterschiede in der Entwicklung herausgearbeitet (Kapitel 3.6).

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

2 Grundlagen der Serverless Architektur

2.1 Historische Entwicklung des Cloud Computings

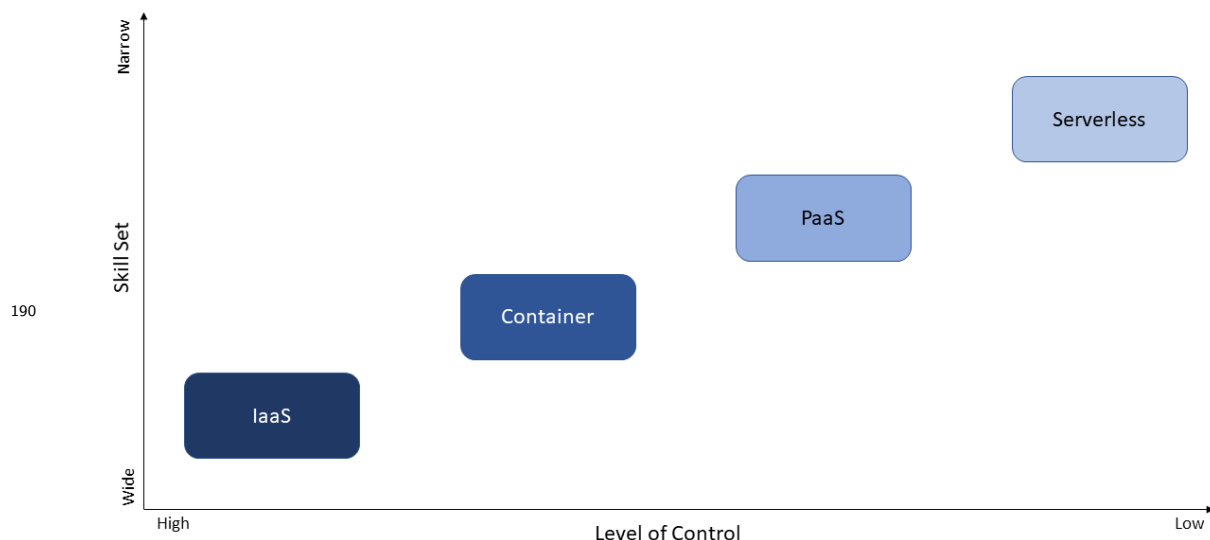
Die Evolution des Cloud Computings begann in den sechziger Jahren. Es wurde das Konzept entwickelt Rechenleistung über das Internet anzubieten. John McCarthy beschrieb das Ganze im Jahr 1961 folgendermaßen. [Gar99, S. 1]

„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.

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

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



191 Abbildung 3: Zusammenhang Kenntnisstand und Kontroll-Level [Bü17]

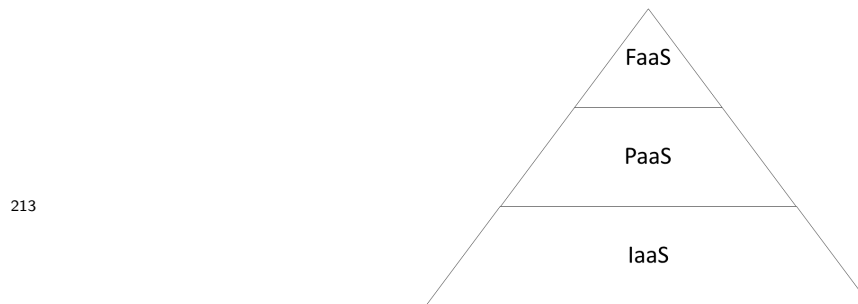
192 Im ersten Schritt werden hierzu häufig Infrastructure as a Service (IaaS) Plattformen
193 verwendet. Diese wurden für eine breite Masse verfügbar, als die ersten Anbieter in den
194 frühen 2000er Jahren damit anfangen Software und Infrastruktur für Kunden bereitzustel-
195 len. Amazon beispielsweise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

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

201 Skalierbarkeit und Verfügbarkeit der Anwendung erreicht werden.

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

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



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

215 2014 tat sich Amazon dann als Vorreiter für das Serverless Computing hervor und brachte
216 AWS Lambda auf den Markt. Diese Plattform ermöglicht dem Nutzer Serverless Anwen-
217 dungen zu betreiben. 2016 zogen Microsoft mit *Azure Function* und Google mit *Cloud*
218 *Function* nach. [RPMP17]

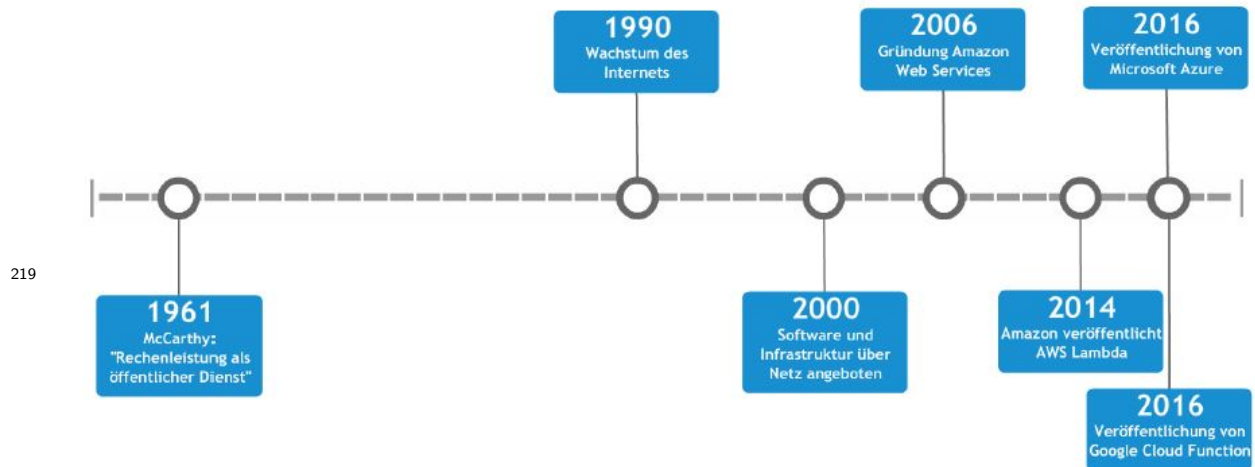


Abbildung 5: Historische Entwicklung des Cloud Computings

2.1.1 Grundlagen des Cloud Computings

„Run code, not Server [Rö17]“

Dies kann als eine der Leitlinien des Cloud Computings angesehen werden. Cloud-Angebote sollen den Entwickler 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. Ein Webservice 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, sodass lediglich anwendungsspezifische Logik vom Entwickler implementiert werden muss. Die einzelnen Services können dann zu einer komplexen Softwareanwendung zusammengefügt werden. [Rö17]

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]

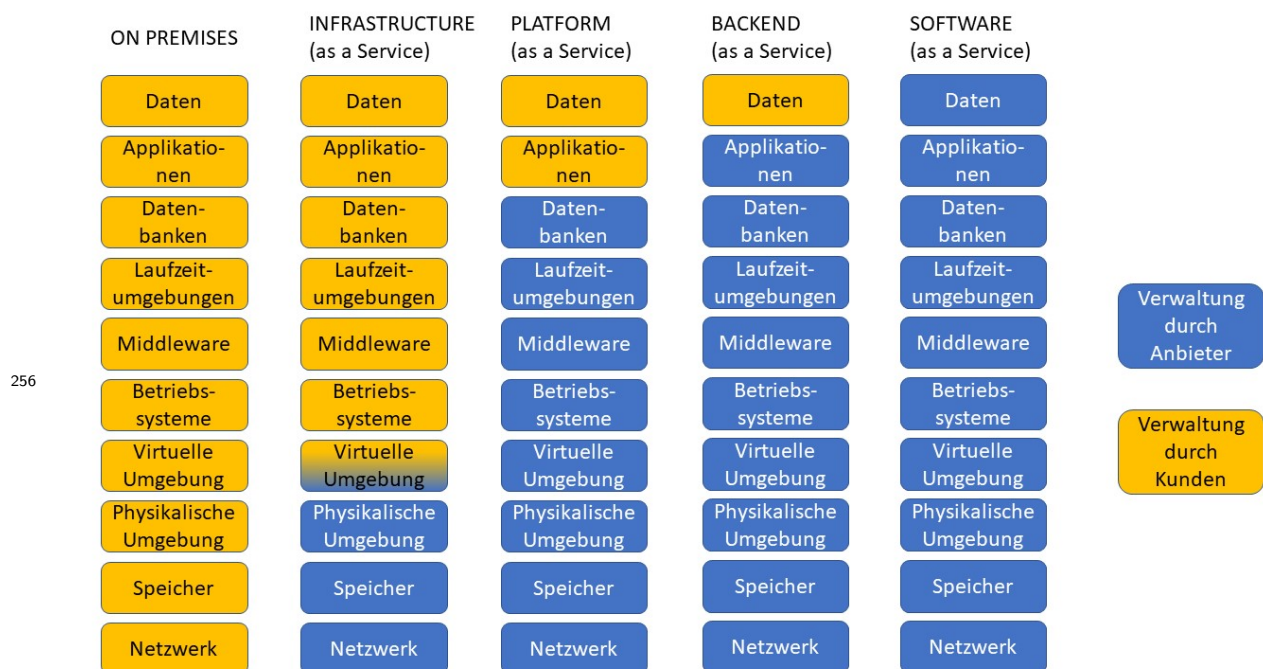


Abbildung 6: Verantwortlichkeiten der Organisation nach [Rö17]

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. Dem liegt das Pay-per-use-Modell zugrunde. Der Kunde zahlt aufwandsbasiert. Das heißt, er zahlt nur für die verbrauchte Rechenzeit. Leerlaufzeiten werden nicht mit einberechnet. [Rö17]

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, so dass lediglich der Code hochgeladen werden muss.

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 Anbieter-spezifisch, sodass der Programmierer auf weitere externe Tools zurückgreifen muss. [Bü17]

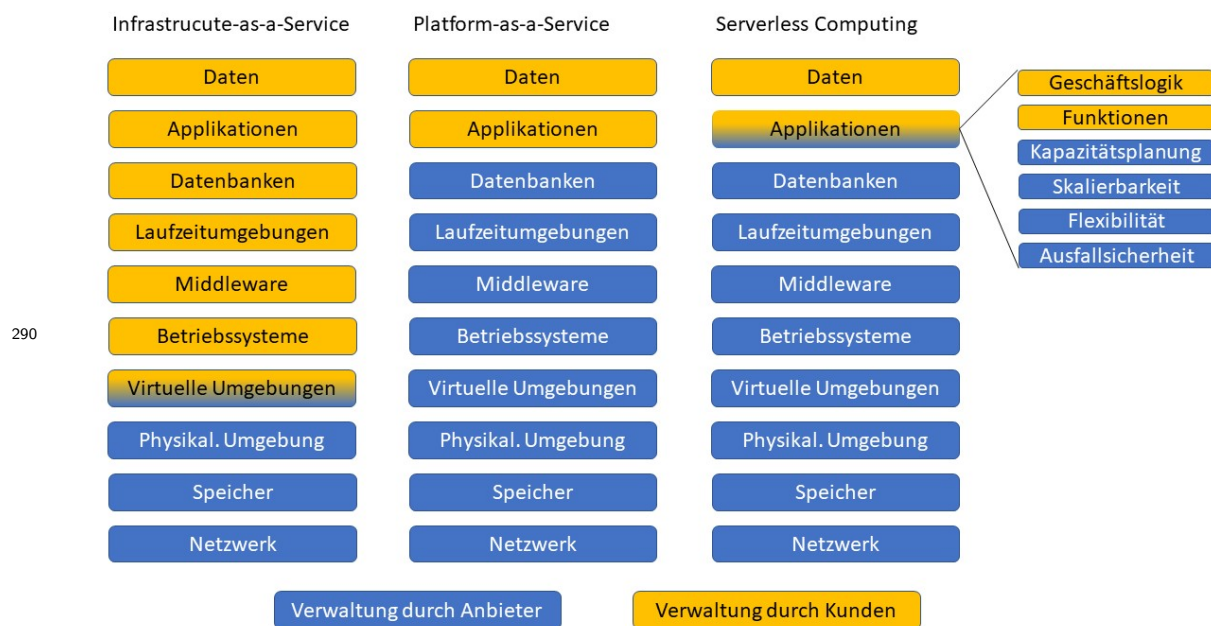


Abbildung 7: Aufgabenverteilung: IaaS vs. PaaS vs. Serverless [Bü17]

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

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

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

321 Das Konzept die Funktionalität in kleine Einheiten aufzuteilen, findet sich auch im Server-
322 less Computing wieder. Im Gegensatz zur Microservices ist Serverless viel feingranularer.
323 Bei Microservices wird oft das Domain-Driven Design herangezogen, um eine komplexe
324 Domäne in sogenannte *Bounded Contexts* zu unterteilen. Diese Kontextgrenzen werden
325 dann genutzt, damit die fachlichen Aspekte in verschiedene individuellen Services auf-
326 geteilt werden können. [FL14] In diesem Zusammenhang wird auch oft von serviceori-
327 entierter Architektur gesprochen. Dahingegen stellt eine Serverless Funktion nicht einen
328 kompletten Service dar, sondern eine einzelne Funktionalität. So eine Funktion kann bei-
329 spielsweise gleichermaßen auch einen Event Handler darstellen. Daher handelt es sich
330 hierbei um eine ereignisgesteuerte Architektur. [Tur18]

331 Ebenso ist es bei Serverless Anwendungen nicht notwendig die unterliegende Infrastruk-
332 tur zu verwalten. Das heißt, dass lediglich die Geschäftslogik als Funktion implemen-
333 tiert werden muss. Weitere Komponenten wie beispielsweise ein Controller müssen nicht
334 selbstständig entwickelt werden. Außerdem bietet der Cloud-Provider bereits eine auto-
335 matische Skalierung als Reaktion auf sich ändernde Last an. Also auch hier werden dem
336 Entwickler Aufgaben abgenommen. [Inc18, S. 9]

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

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

2.2 Eigenschaften von Function-as-a-Service

Wenn von Serverless Computing gesprochen wird, ist oftmals auch von FaaS die Rede. Der Serverless Provider stellt eine FaaS Plattform zur Verfügung. Die Infrastruktur des Anbieters kann dabei als BaaS gesehen werden. Eine Serverless Architektur stellt also eine Kombination aus FaaS und BaaS 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. Dadurch lassen sich in kürzester Zeit viele Kopien derselben Funktionalität starten, sodass eine hohe Skalierbarkeit erreicht werden kann. 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. [Rö17]

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. Diese speichert das Bild im *Blob Store*, wodurch der Vorgang abgeschlossen sein könnte. Jedoch wird durch das Speichern in der Datenbank ein weiteres Event ausgelöst, das die *Activity Function* auslöst. 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. 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. So könnte ein bereits bearbeitetes Bild noch einmal angepasst werden.

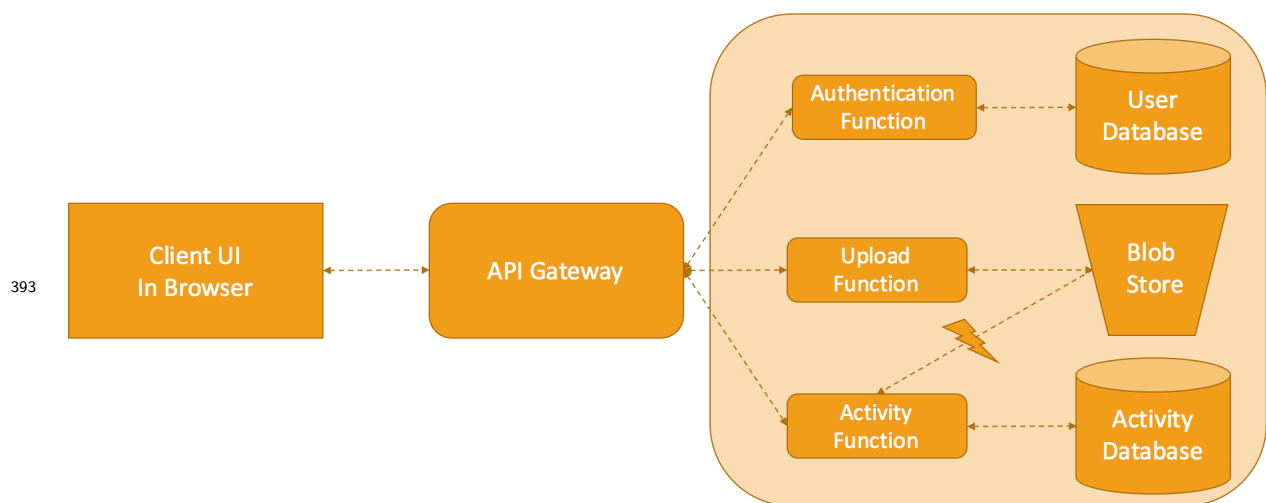


Abbildung 8: FaaS Beispiel Anwendung [Tiw16]

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 Functions parallel und unabhängig voneinander ausgeführt werden können, sodass der Kunde nicht mit diesem Aufgabenfeld in Berührung kommt. Hierzu cachen eini-

ge 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]

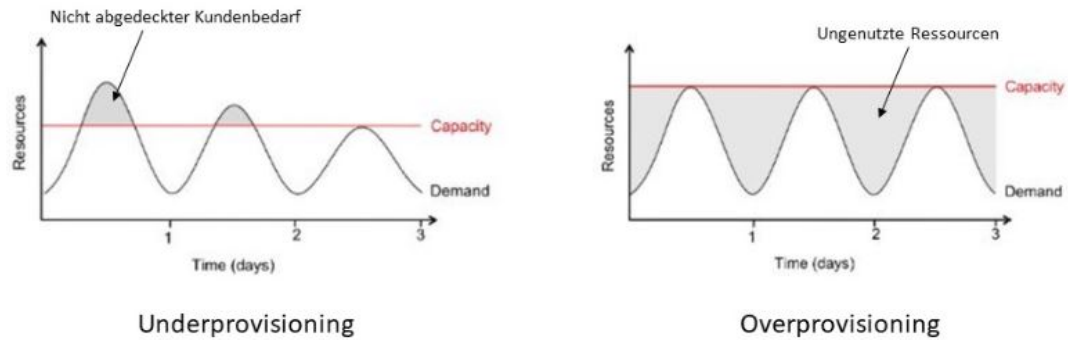


Abbildung 9: Under- und Overprovisioning [A⁺09, S. 11]

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, sodass unnötige Ausgaben entstehen.

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 Ressourcen optimal ausgenutzt werden. So werden die Ressourcen keinem bestimmten Kunden zugeordnet, sondern stehen für viele Nutzer bereit. Je nach Bedarf können dem Anwender dynamisch benötigte Ressourcen 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]

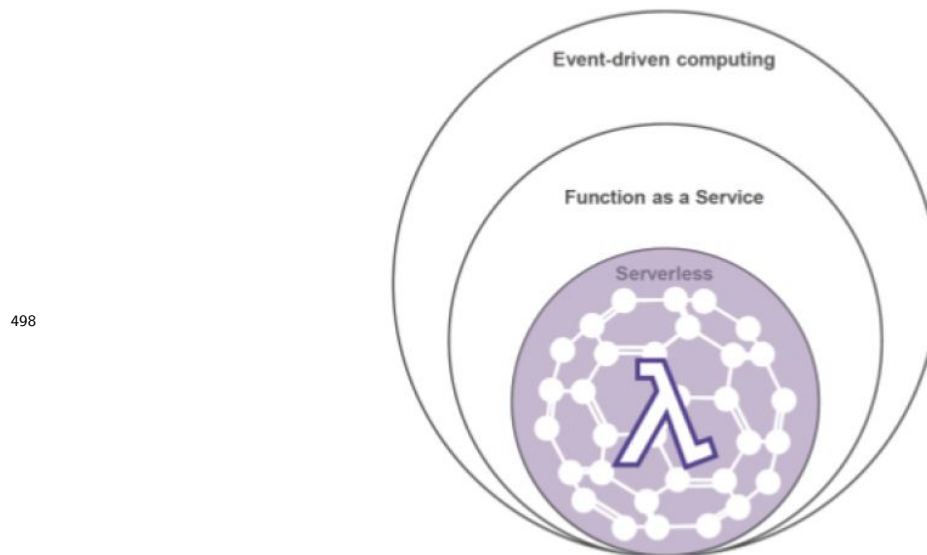
Anhand des Manifestes ist es schon zu erkennen, dass sich Serverless Computing nicht einfach in einem Pattern beschreiben lässt. Es spielen viele Muster zusammen. So enthält das Manifest beispielsweise neben wichtigen Prinzipien auch Pattern, die bei der Umsetzung von Serverless Anwendungen angewendet 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 harmonisiert 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]

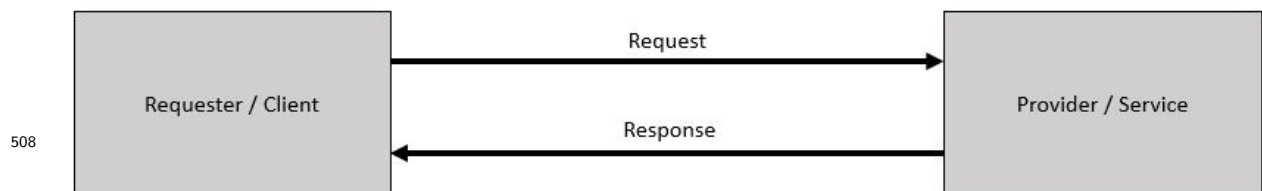
Der Aufruf der somit erstellten Functions führt zum nächsten Muster für Serverless Umsetzungen. Die ereignisgesteuerte Architektur sorgt dafür, dass die Functions durch Events

494 aufgerufen werden können. Dieses Architekturmuster wird natürlich nicht nur bei Ser-
 495 verless Anwendungen verwendet, sondern kann auch in anderen Umfeldern zum Einsatz
 496 kommen. Es handelt sich bei Serverless Computing also lediglich um einen kleinen Be-
 497 standteil des *Event-driven computings* (siehe Abb. 10). [Boy17]



498
 499 Abbildung 10: Zusammenhang zwischen Event-driven Computing, FaaS und Serverless [FIMS17, S. 5]

500 Neben asynchronen Events können Serverless Functions auch durch synchrone Nachrich-
 501 ten angesprochen werden. Hierzu kann als Einstiegspunkt einer Function ein HTTP-
 502 Endpunkt dienen. Der Aufruf folgt dann dem *Request-Response Pattern*, das als Basismethode zur Kommunikation zwischen zwei Systemen angesehen werden kann. Der Requester startet mit seinem 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 Requester zurück. [Swa18]



508
 509 Abbildung 11: Request Response Pattern [Swa18]

3 Entwicklung einer prototypischen Anwendung

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 Kriterien 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 dem Testen und dem 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 mit den 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 Komponenten. Dies bedeutet, dass beleuchtet wird, ob einzelne Teile losgelöst vom restlichen System in anderen Projekten erneut einsetzbar sind.

Betriebskosten In einer theoretischen Betrachtung werden die Betriebskosten für die jeweiligen Anwendungen gegenübergestellt. So können anhand einer Hochrechnung für die Menge der benötigten Ressourcen die Kosten berechnet werden.

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 werden auch die Möglichkeiten bezüglich verschlüsselter Zugriffe genauer betrachtet.

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 das 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 eine dreistufige Skala als angemessen angesehen wird. Da bei dem hier durchgeführten Vergleich kein Kontext, wie zum Beispiel Vorgaben eines Unternehmens, auf die sich die Analyse beziehen soll, 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 ausma-

575 chen. Die objektive Evaluation hingegen nimmt nur einen kleine Anteil der Auswertung
576 ein und bezieht sich auf die Erhebung von neutralen Daten wie zum Beispiel bei Messun-
577 gen [Zar17, S. 36].

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

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

587 3.2 Fachliche Beschreibung der Beispiel-Anwendung

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

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

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

603 Damit der Servicebetreiber sein Angebot an die Nachfrage der Kunden anpassen kann,
604 merkt sich das System bei jeder Ausleihe zusätzlich die Kategorie des ausgeliehenen Bu-
605 ches, sodass anhand der beliebten Bücherkategorien der Bestand sinnvoll erweitert werden
606 kann. Die Nutzerstatistik ist ausschließlich für Mitarbeitern einsehbar.

607 Dieser Ablauf könnte in einem anderen Anwendungsfall beispielsweise eine Webseite sein,

608 die den Nutzer nach der Auswahl eines Werbebanners nicht nur auf die werbetreibende
609 Seite leitet, sondern sich gleichzeitig den Aufruf der Werbung merkt, um ihn später in
610 Rechnung stellen zu können [Rob18].

611 3.3 Implementierung der Benutzeroberfläche

612 Da die beiden prototypischen Anwendungen sich lediglich in der Umsetzungsart der An-
613 wendungslogik unterscheiden, kann die selbe Frontendimplementierung für beide Prototy-
614 pen eingesetzt werden. Dies ist möglich, da beide Anwendungen die gleichen Schnittstellen
615 zur Verfügung stellen.

616 Die Benutzeroberfläche wird mittels Polymer implementiert. Polymer ist eine Bibliothek
617 zur Frontendentwicklung, die auf der *Web Components Specification* des World Wide Web
618 Consortiums (W3C) basiert. So kann eine Seitenansicht aus mehreren verschachtelten
619 Komponenten bestehen. Die hohe Wiederverwendbarkeit solcher Komponenten und das
620 damit einhergehende einheitliche Erscheinungsbild sind zwei Vorteile des komponenten-
621 basierten Konzepts. [Sch16]

622 „Typischerweise besteht eine Polymer Komponente aus drei Teilen: Styles-
623 heets, einem Template und natürlich JavaScript. [WJ16]“

624 Alle Bestandteile einer Ansicht befinden sich somit in einer Datei. Neben der Bereitstel-
625 lung der Daten und der Benutzereingabe wird auf die Eingabe des Users reagiert. Auch
626 die Kommunikation mit dem Server übernimmt jede Komponente für sich. In der An-
627 wendung werden außer den anforderungsspezifischen eigens erstellten Komponenten auch
628 weitere Module, die unter *webcomponents.org* registriert sind und von anderen Entwick-
629 lern stammen, verwendet.

630 Properties innerhalb einer Komponente können zum Datenaustausch zwischen den ver-
631 schachtelten Modulen genutzt werden (siehe Listing 1). Mittels *Two-Way Binding* können
632 die Attribute von beiden Seiten aus verändert werden. Zur Darstellung von Arrays bietet
633 Polymer einen *Template repeater* an. Durch die Verwendung von
634 `<template is="dom-repeat">` kann eine Darstellung der Elemente aus einer Liste erzeugt
635 werden (siehe Listing 2 Z. 13-15). Nach demselben Prinzip kann das Anzeigen einzelner
636 Teile des Templates durch `dom-if` an eine Bedingung geknüpft werden. [WJ16]

Listing 1: 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>
```

Listing 2: Auflistung der Elemente eines Arrays

```

1   <dom-module id="ba-ausleihe">
2     <template>
3       <style include="shared-styles">
4         :host {
5           display: block;
6           padding: 10px;
7         }
8         .full-width {
9           width: 100%;
10        }
11      </style>
12
13      <template is="dom-repeat" items="[[books]]">
14        <p>[[item.title]]</p>
15      </template>
16    </template>
17
638 18    <script>
19      /**
20       * Overview of all available books.
21       * @customElement
22       * @polymer
23       */
24      class Ausleihe extends Polymer.Element {
25        static get is() { return 'ba-ausleihe'; }
26        static get properties() {
27          return {
28            /** Array with all books. */
29            books: Array
30          }
31        }
32      }
33      window.customElements.define(Ausleihe.is, Ausleihe);
34    </script>
35  </dom-module>

```

639 Da der Fokus der Arbeit auf der Backendentwicklung liegt, wird das Frontend recht
640 schlicht gehalten. Die Polymeranwendung wurde initial aus dem **Polymer Starter Kit**
641 erzeugt. Standardmäßig ist hierbei eine Headerzeile mit dem Titel der Anwendung sowie
642 ein linksbündiges Menü enthalten. Zu diesem wurden einzelne Ansichten hinzugefügt,
643 die die jeweiligen Funktionalitäten repräsentieren. Unter dem Menüpunkt *Bücherausleihe*
644 beispielsweise erhält der Nutzer eine Übersicht aller Bücher und kann verfügbare Bücher

über eine Checkbox zum Ausleihen auswählen (siehe Abb. 12). Falls ein Buch bereits ausgeliehen ist, wird die Checkbox zur Ausleihe gesperrt.

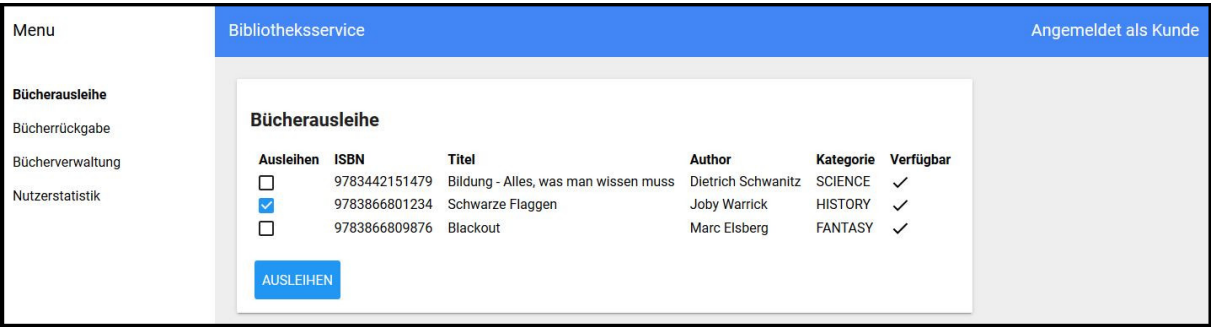


Abbildung 12: Maske: Bücherausleihe

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).

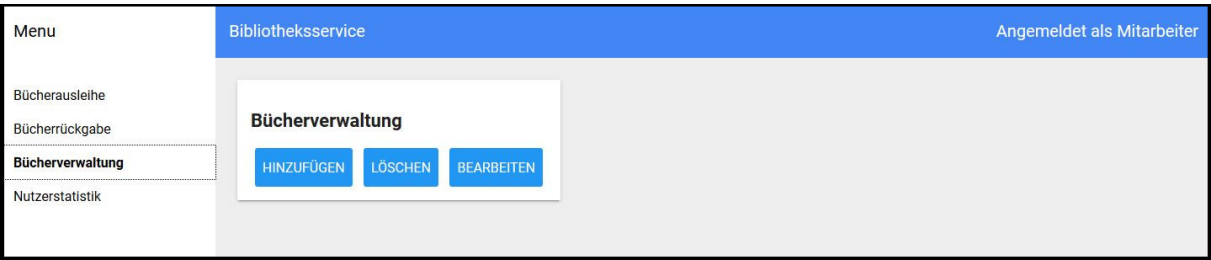


Abbildung 13: Maske: Bücherverwaltung

1 Buch bearbeiten

ISBN	Titel	Author		
9783442151479	Bildung - Alles, was man wie	Dietrich Schwanitz	SCIENCE	
9783866809876	Blackout	Marc Elsberg	FANTASY	
9783866801234	Schwarze Flaggen	Joby Warrick	HISTORY	
9783942656863	Kalte Asche 2	Simon Beckett	FANTASY	

BEARBEITUNG BEENDEN

2 Buch löschen

Löschen	ISBN	Titel	Author
<input checked="" type="checkbox"/>	9783442151479	Bildung - Alles, was man wissen muss	Dietrich Schwanitz
<input type="checkbox"/>	9783866809876	Blackout	Marc Elsberg
<input type="checkbox"/>	9783866801234	Schwarze Flaggen	Joby Warrick
<input type="checkbox"/>	9783942656863	Kalte Asche	Simon Beckett

ABBRECHEN LÖSCHEN

3 Buch hinzufügen

ISBN
9783942656863

Titel
Kalte Asche

Autor
Simon Beckett

Kategorie(SCIENCE, FANTASY, HISTORY)
FANTASY

ABBRECHEN HINZUFÜGEN

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 der Amazon-spezifischen 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]

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

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

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

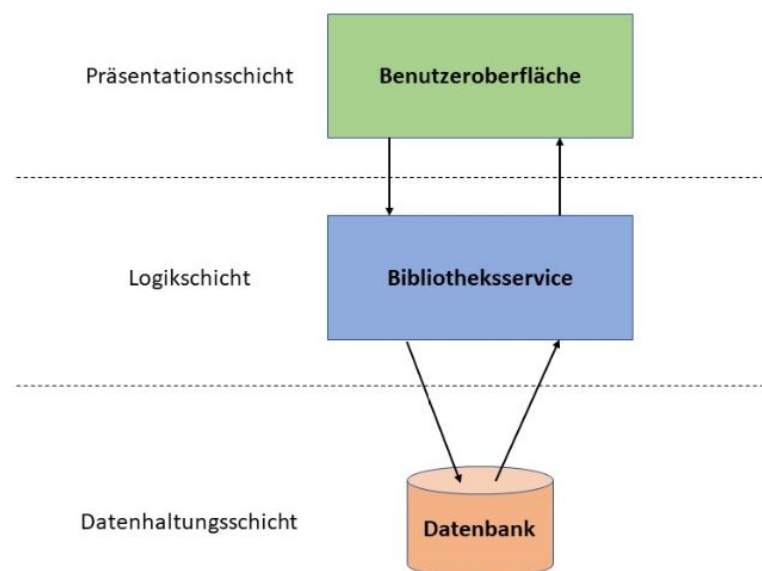


Abbildung 15: 3-Tier Architektur

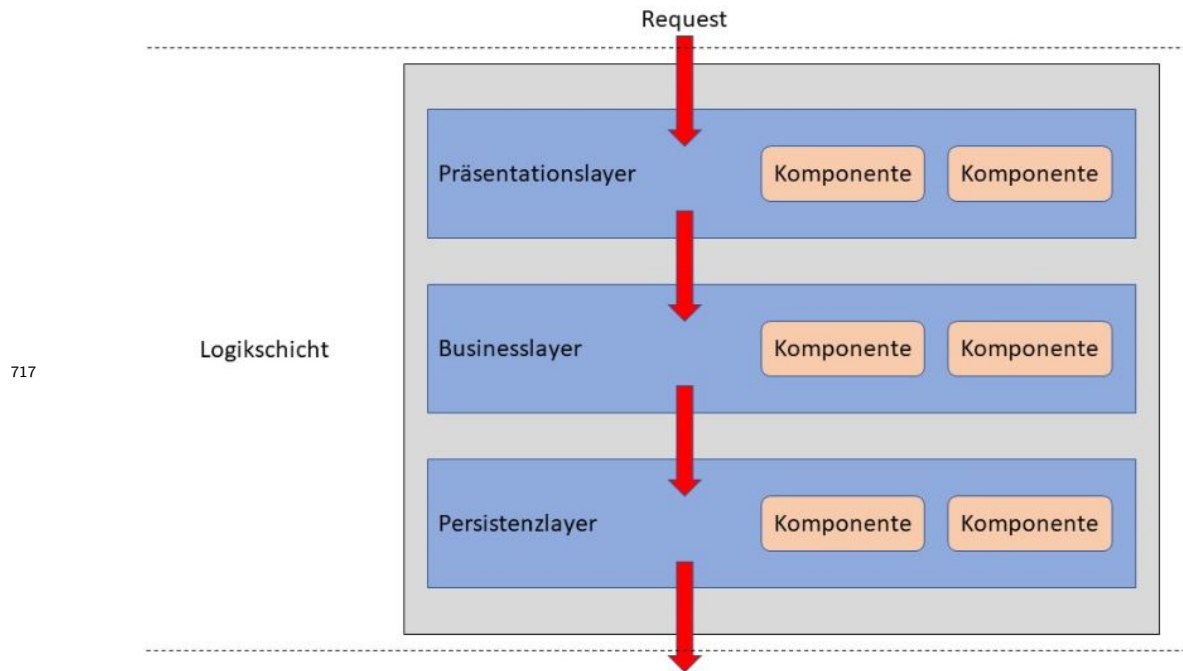
696 Ein Vorteil dieses Architekturmusters ist beispielsweise die Möglichkeit Frontend und Ba-

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 wie-
derverwendet 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.
Die Applikation besteht aus Präsentations-, Business- und Persistenzlayer (siehe Abb. 16).
Das Präsentationslayer 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).



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

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

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

734 3.4.2 Implementierung der Anwendung

735 Um das beschriebene Architekturmodell umzusetzen, wird Spring Boot verwendet. Es
 736 dient zur Minimierung von *boilerplate code* durch Spring-spezifische Annotationen und
 737 vereinfacht so den Umgang mit dem Spring Framework. Spring folgt dem Prinzip *Conven-*

738 *tion over Configuration*. Dem Entwickler wird so eine Menge an konfigurativen Aufgaben
739 abgenommen. Somit können ohne großen Aufwand standardmäßig bereitgestellte Funk-
740 tionen in Anspruch genommen oder eigene Funktionalitäten hinzugefügt werden. Spring
741 Boot bringt beispielsweise bereits einen eingebetteten Tomcat-Server mit. Dieser bietet
742 eine vollständige Laufzeitumgebung für die Anwendung und ermöglicht ein einfaches De-
743 bugging.

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

Listing 3: Einstiegsklasse für Spring Boot Anwendung

```
1  @SpringBootApplication
2  public class ClassicApplication {
750 3      public static void main(String[] args) {
4          SpringApplication.run(ClassicApplication.class, args);
5      }
6  }
```

751 Dieses Startbeispiel kann um verschiedene weitere Features aus dem Spring-Stack oder
752 auch um eigene Funktionalitäten erweitert werden. Neben dem eingebetteten Server stellt
753 Spring Boot per Default auch eine H2 In-Memory Datenbank zur Verfügung. Diese eig-
754 net sicher hervorragend, um prototypische Anwendungen zu erstellen. Bei der H2 Da-
755 tenbank handelt sich um einen relationalen Datenspeicher, der mit dem Start der Ap-
756 plikation neu initialisiert und nach dem Beenden wieder zurückgesetzt wird. Das somit
757 voreingestellte Datenbankmanagementsystem kann jedoch auch jederzeit durch einen ei-
758 genen Datenbankserver ersetzt werden. Hierzu müssen lediglich ein paar Einstellungen
759 im `application.yml` Dokument, das als Konfiguration für die Spring Boot Anwendung
760 dient, vorgenommen werden.

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

764 Zur Abbildung des Datenmodells auf die Datenbank dient das *Object-relational map-*
765 *ping (ORM)*. Im einfachsten Fall werden dabei Klassen zu Tabellen, die Objektvaria-
766 blen 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 auf den Speicher reproduziert 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).

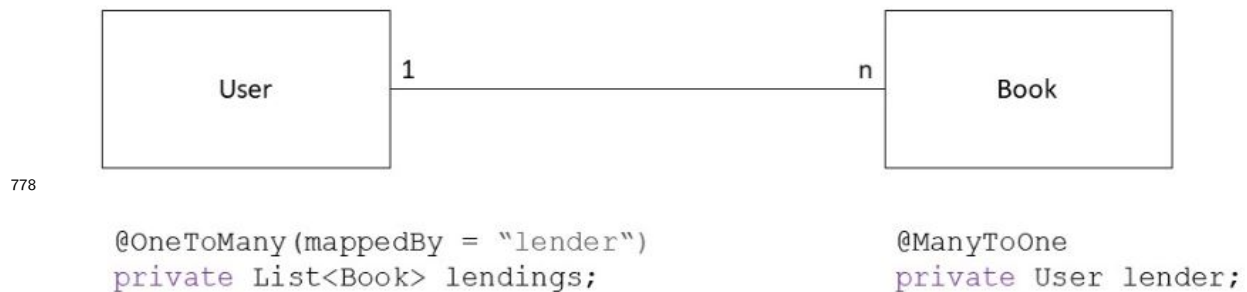


Abbildung 17: Beziehung zwischen User und Book

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 hierbei ebenso Annotationen zum Einsatz (siehe Abb. 18).

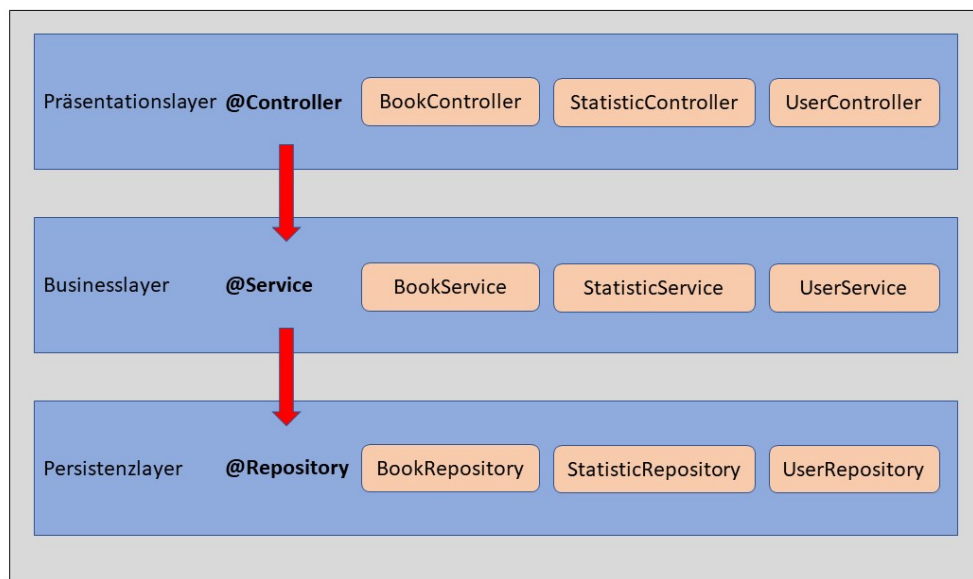


Abbildung 18: Layered Architektur in Spring

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

804 Konstruktor, wird auch nicht mehr wie bisher die Annotation `@Autowired` benötigt. Ein
805 weiterer Beitrag zur Steigerung des Komforts für den Entwickler. Spring erzeugt nun
806 im Hintergrund das passende Objekt. Hierzu wird aus dem DI-Container, der alle Be-
807 ans enthält, die passende Klasse initialisiert. Dies führt dazu, dass die Initialisierung der
808 Abhängigkeiten nicht mehr per Hand durchgeführt werden muss und die Objekte erst zur
809 Laufzeit vorliegen müssen. [Kar18]

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

814 Die Servicemethoden werden aus den Controllern heraus aufgerufen. Controller definie-
815 ren REST-Endpunkte, die für den Client als Einstiegspunkt zur Anwendung dienen und
816 den Zugriff auf die entsprechenden Services ermöglichen (siehe Listing 5 ab Z. 9). Die
817 Endpunkte innerhalb eines Controller unterscheiden sich anhand des Pfades beziehungs-
818 weise des REST-Verbs. Durch die Annotation `@RequestMapping` an der Klasse kann ein
819 Basispfad für alle Einstiegspunkte des Controllers festgelegt werden. So gibt es für je-
820 de REST-Methode die entsprechende Mapping-Annotation wie zum Beispiel `@GetMapping`
821 und `@PostMapping`. Um dem Client nun beispielsweise den Zugriff auf alle Bücher sowie das
822 Hinzufügen und Löschen eines Exemplars zu ermöglichen, ist folgende Implementierung
823 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     @DeleteMapping(path = "/books/{isbn}")
20     public ResponseEntity<Book> deleteBook(@PathVariable String isbn) {
21         return ResponseEntity.ok(bookService.deleteBook(isbn));
22     }
23 }
```

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
836 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 authentifizierten Users

```

1  /**
2   * Returns all lent books for a specific user.
3   * @param authentication authentication object containing the active
    user
837 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  }

```

838 Eine rollenbasierte Autorisierung ist ebenso durch die Security Komponente von Spring
839 erreichbar. Dafür werden lediglich die Einstiegspunkte zur Applikation, das heißt die End-
840 punkte im Controller, mit @PreAuthorize und der zugehörigen Rolle annotiert (siehe Listing
841 8 Z. 2).

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
842 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  }
```

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

852 3.4.3 Testen der Webanwendung

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

860 Im Springumfeld ist es sinnvoll, die Testklasse mit der Annotation `@SpringBootTest` zu
861 versehen. So wird bei der Ausführung der Tests ein Springkontext, der dem beim Start der
862 Anwendung gleicht, aufgebaut. Nachteil hieran ist allerdings der immer größer werdende
863 Overhead, wenn für jede Testklasse ein neuer Kontext errichtet werden muss. So kann
864 sich die Durchführung vieler Testfälle deutlich verzögern. [Gig18]

865 Komponenten-/Unittests

866 Um lediglich ein Modul zu überprüfen, müssen alle Komponenten, die mit diesem Modul

867 interagieren, für den Test ausgeschlossen werden. Hierfür können sogenannte Mocks ein-
868 gesetzt werden. Im Springkontext gibt es dafür die `@MockBean` Annotation. Somit können
869 fremde Komponenten durch eine Art Platzhalter ersetzt werden, sodass sie ein vorhersag-
870 bares Verhalten annehmen (siehe Listing 9 Z. 5-6 und 21-22). Dadurch kann ausgeschlos-
871 sen werden, dass das betrachtete Modul durch Fremdeinflüsse beeinträchtigt wird. Für
872 den Test eines Controllers wird also beispielsweise ein Mock für den verwendeten Service
873 angelegt. [Gig18]

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

877 Eine weiterer Besonderheit ist die Annotation `@WithMockUser`. Hiermit wird der authenti-
878 zierte Benutzer mit der zugehörigen Rolle für den Testfall festgelegt. Somit kann auch der
879 Zugriffsschutz mit getestet und beispielsweise eine `AccessDeniedException` provoziert werden
880 (siehe Listing 9 Z. 19).

881 Nach dem selben Prinzip wird im Test des Services ein Mock für das zu Grunde liegende
882 Repository angelegt.

Listing 9: Testfall im StatisticControllerTest

```

1  @SpringBootTest
2  @RunWith(SpringRunner.class)
3  public class StatisticControllerTest {
4      private static final Statistic STATISTIC = new Statistic(1, 34,
5          Category.SCIENCE);
6      @MockBean
7      private StatisticService statisticService;
8      @Autowired
9      private WebApplicationContext webApplicationContext;
10     private MockMvc mockMvc;
11
12     @Before
13     public void setup() {
14         MockitoAnnotations.initMocks(this);
15         mockMvc = MockMvcBuilders
883         .webApplicationContextSetup(webApplicationContext).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\"}",
28             result.getResponse().getContentAsString());
29     }

```

884 Integrationstests

885 Nachdem durch die Unittests die Korrektheit der einzelnen Module festgestellt wurde,
886 können Integrationstests dazu genutzt werden, um die Zusammenarbeit zwischen den
887 verschiedenen Teilen zu testen. Hierzu muss lediglich auf die Mocks verzichtet werden,
888 sodass alle beteiligten Komponenten beim Aufruf ausgeführt und somit überprüft werden
889 können (siehe Listing 10 Z. 7-11). [Gig18]

890 Da der entwickelte Springkontext auch zum Testen eingesetzt wird, laufen ebenso die
891 Flyway-Skripte bei der Durchführung des Tests. Somit befindet sich die Datenbank während

892 des Tests im selben Zustand wie zum Start der Anwendung.

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
893 7      @Test
8      public void testGetBooks() {
9          Collection<Book> books = bookService.getBooks();
10         assertThat(books).isNotNull().isNotEmpty();
11         assertEquals(3, books.size());
12     }
13 }
```

894 3.5 Implementierung der Serverless Webanwendung

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

905 3.5.1 Architektonischer Aufbau der Serverless Applikation

906 Da lediglich die Anwendungslogik implementiert werden muss, unterscheidet sich die Ar-
907 chitektur der Serverless Applikation grundlegend von dem eben erläuterten Aufbau. Die
908 Aufteilung der Anwendungslogik in viele kleine Komponenten ähnelt dem Microservicege-
909 danken. Die Serverless Architektur erhöht somit die Autonomie der einzelnen Funktionalitäten.
910 Diese werden in Functions abgebildet und durch Events aus verschiedenen Quellen
911 aufgerufen. Dabei wird auch von einer *Event-driven* Architektur gesprochen. [Inc18, S.
912 9-10]

913 Dabei handelt es sich um asynchrone Events, die von einer Function abonniert werden

914 können. Dieses Muster ist auf verschiedenen Anwendungstypen anwendbar und eignet
 915 sich, wie in diesem Fall zu sehen ist, besonders gut für skalierbare Applikationen [Ric15,
 916 S. 11]. Alternativ können die Functions durch synchrone Events per HTTP-Request auf-
 917 gerufen werden.

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

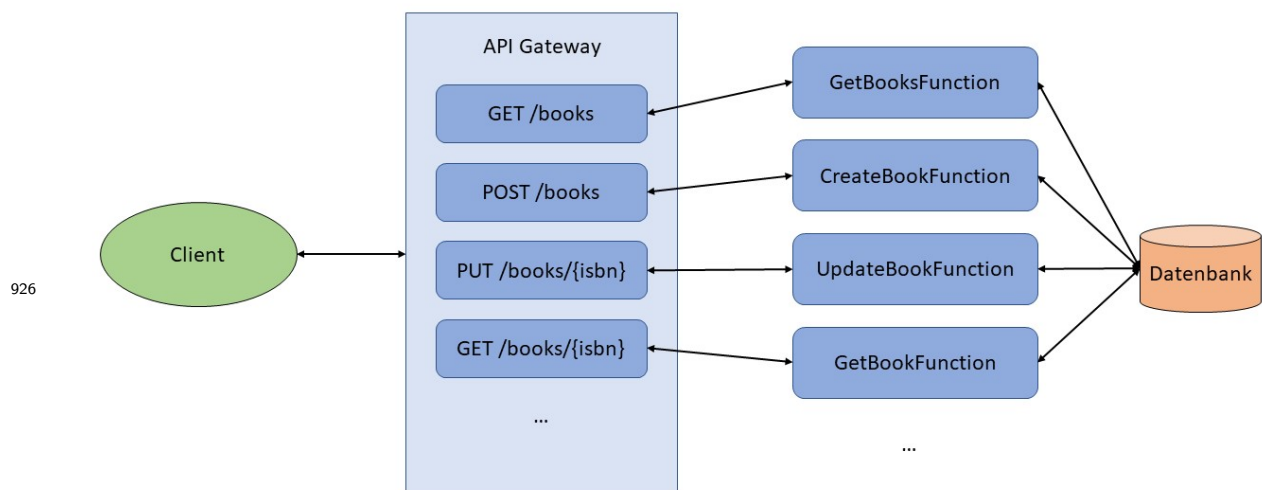


Abbildung 19: API Gateway

927
 928 Als API Gateway wird das BaaS-Angebot von AWS genutzt. Dieses muss lediglich ent-
 929 sprechend konfiguriert werden. Je nach Konfiguration kann das AWS API Gateway direkt
 930 Aufgaben wie zum Beispiel Sicherheitsüberprüfungen übernehmen. [Rob18]

931 Neben der Verteilung der Logik sorgt die Trennung von Client und Cloud-Anwendung
 932 durch das Gateway ebenso dafür, dass die Anwendung nicht mehr als Ganzes greifbar ist.
 933 Damit einhergehend liegt auch die Ablaufsteuerung nicht mehr unter der Kontrolle einer
 934 zentralen Stelle, die die Orchestrierung übernimmt. Der Ablaufprozess wird durch den

935 Eventfluss organisiert. [Kra18, S. 17]

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

938 Auch die Datenhaltung wird durch eine Komponente aus dem Amazonumfeld übernommen.
 939 *DynamoDB* ist eine nicht relationale Datenbank, die gut in Kombination mit AWS Lamb-
 940 da Functions verwendet werden kann. Im Gegensatz zu relationalen Datenbanken werden
 941 dabei die Daten nicht in Tabellen mit Zeilen und Spalten organisiert. Für die Ablage der
 942 Daten werden beispielsweise Objekte oder Dokumente eingesetzt. Da Objekte innerhalb
 943 eines Datenspeichers nicht dem selben Schema folgen müssen, sind nicht relationale Model-
 944 le flexibel einsetzbar. Weil somit auch das relationale Datenbankmodell nicht zum Einsatz
 945 kommt, wird das Key-Value-Datenbankmodell verwendet. Hierbei dient ein Schlüssel als
 946 Identifikator für eine beliebiges Werteobjekt. [Lit17]

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

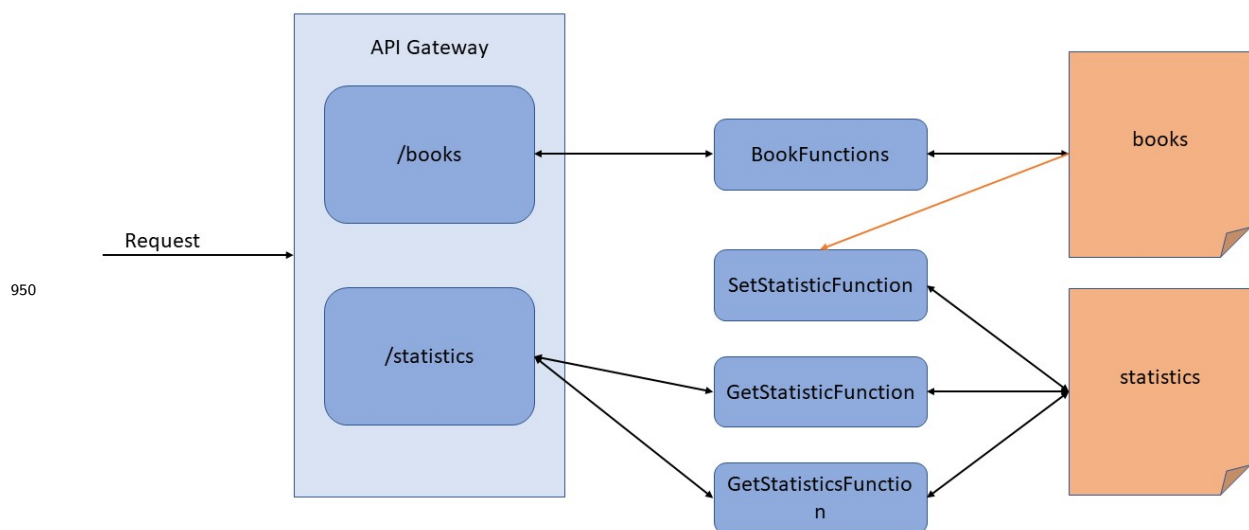


Abbildung 20: Datenbankevent ruft Function auf

952 3.5.2 Implementierung der Anwendung

953 Zur Implementierung der einzelnen Functions wird AWS Serverless Application Model
 954 (SAM) verwendet. Es dient zur lokalen Entwicklung von Serverless Functions im Umfeld

955 von Amazon. Mit Hilfe des Tools kann über die Kommandozeile ein Startprojekt erstellt
 956 werden. Dieses enthält als Quellcode lediglich eine Klasse, die die Function darstellt. Jede
 957 Function muss den RequestHandler implementieren. Dessen Methode `handleRequest()` dient
 958 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>
959  {
7
8      public Object handleRequest(final Object input, final Context
          context) {
9          //Logik der Function
10         }
11
12     }

```

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

Listing 12: Ressourcendefinition der Beispiel Function

```

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

```

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

970 Diese Musteranwendung kann um die benötigten Functions erweitert werden. Damit nicht
971 jedes Modul einen eigenen Kontext, wie zum Beispiel die Verbindung zur Datenbank, pfl-
972 gen muss, wird ein sogenanntes *Data Access Object (DAO)* verwendet. Dies übernimmt
973 die Kommunikation mit der Datenbank und sorgt somit auch für eine einfache Austausch-
974 barkeit der Datenhaltungskomponente.

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

980 Die Implementierung von Dagger besteht aus zwei Teilen. Zum einen den Modulen, die
981 die benötigten Abhängigkeiten bereitstellen, und zum anderen einer sogenannten Kom-
982 ponente, die die Initialisierung der im Modul definierten Objekte ermöglicht. [Cha17]

983 Nachdem beispielsweise das DAO im Modul als Abhängigkeit definiert wurde, kann es
984 mittels `@Inject` in einem Handler genutzt werden (siehe Listing 14 Z. 6-7).

985 Für den Zugriff auf die DynamoDB wird ebenfalls ein Modul erstellt, das die Verbindung
986 zur Datenbank bereitstellt (siehe Listing 13). Über den `DynamoDbClient` kann dann im
987 `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() {
988         DynamoDbClient client = DynamoDbClient.builder()
6             .region(Region.EU_CENTRAL_1)
7             .build();
8             return client;
9
10         }
11     }
```

989 Eine vollständige Function mit zugehöriger Methode aus dem DAO sieht wie folgt aus.

Die Map als Parameter der `handleRequest()` Methode enthält dabei die Attribute des Requests wie Header, Pfadvariablen und den Body (siehe Listing 14 Z. 17-19). 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). Nachdem über den Datenbankclient das entsprechende Buch aus dem Speicher geladen wurde, wird das `GetItemResponse` Objekt mit der `convert()` Methode in ein Buch Objekt umgewandelt (siehe Listing 15 Z. 21).

Listing 14: GetBookFunction

```

1  public class GetBookHandler implements RequestHandler<Map<String ,
    Object> , GatewayResponse> {
2
3      @Inject
4      ObjectMapper objectMapper ;
5
6      @Inject
7      BookDao bookDao ;
8
9      private final AppComponent appComponent ;
10
11     public GetBookHandler () {
12         appComponent = DaggerAppComponent . builder () . build () ;
13         appComponent . inject ( this ) ;
14     }
997
15
16     @Override
17     public GatewayResponse handleRequest (Map<String , Object> input ,
        Context context) {
18         String pathParameter = input . get ( "pathParameters" ) . toString () ;
19         String isbn = pathParameter . substring (6 ,
20             pathParameter . length () -1) ;
21         Book book = bookDao . getBook (isbn) ;
22         try {
23             return new GatewayResponse (objectMapper . writeValueAsString (book)
                , HEADER , SC_OK) ;
24         } catch (JsonProcessingException e) {
25             return new GatewayResponse (e . getMessage () , HEADER ,
                SC.INTERNAL_SERVER_ERROR) ;
26         }
27     }
28 }
```

Listing 15: Modul zur Bereitstellung der Datenbankverbindung

```

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 {
998             return Optional.ofNullable(
14                 dynamoDb.getItem(GetItemRequest.builder()
15                     .tableName(tableName)
16                     .key(Collections.singletonMap(BOOK_ID,
17                         AttributeValue.builder().s(isbn).build()))
18                     .build()))
19                 .map(GetItemResponse::item)
20                 .map(this::convert)
21                 .orElse(null);
22         } catch (ResourceNotFoundException e) {
23             throw new TableDoesNotExistException("Book table " + tableName +
24                 " does not exist.");
25         }
26     }
27 }

```

999 Die Authentifizierung erfolgt im Amazonkosmos über das Identity and Access Manage-
1000 ment (IAM). Dabei können Nutzer sowie Rollen erstellt werden. Der Zugriff auf einzelne
1001 Functions kann über verschiedene Rollen geregelt werden.

1002 ...

1003 Auf die Implementierung der Functions folgt anschließend das Deployment. Dieses ist wie
1004 oben genannt mit Hilfe des SAM Tools möglich. Daraufhin können alle Functions über
1005 die *AWS Lambda Console* eingesehen und verwaltet werden (siehe Abb. 21). Hier können
1006 den Functions Rollen für den Zugriffsschutz zugewiesen werden. Des Weiteren muss der
1007 zur Verfügung stehende Arbeitsspeicher für jede Function erhöht werden, da sie sonst bei
1008 der Ausführung in ein Timeout laufen.

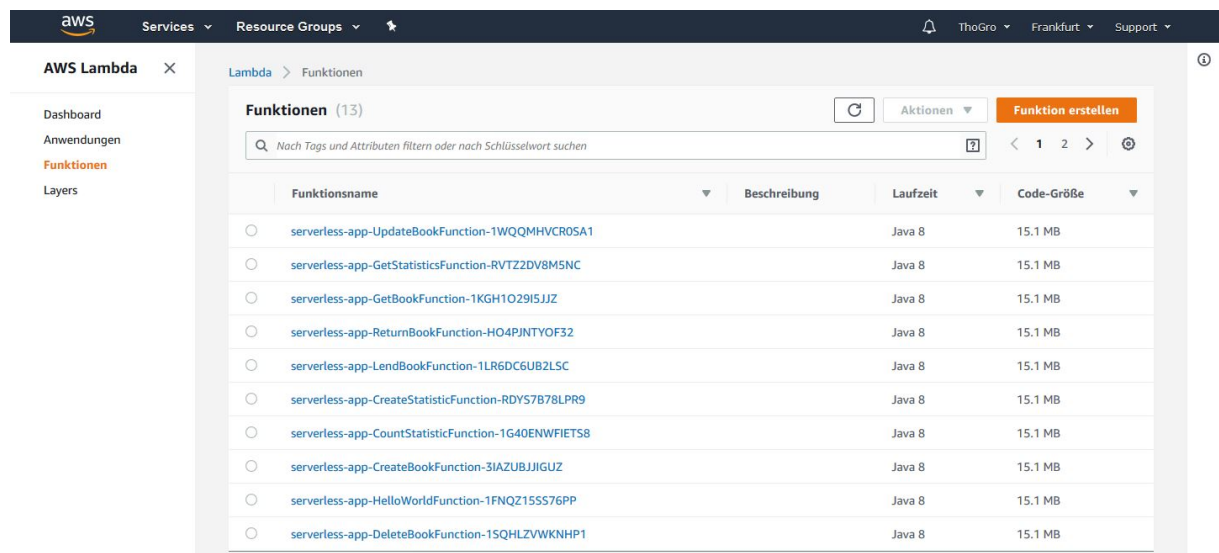


Abbildung 21: Lambda Console

Zuletzt muss auch noch das API Gateway per Hand angepasst werden. Die automatisch erstellte Konfiguration des Gateways erlaubt keine Zugriffe von fremden Quellen 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 Testen der klassischen Webanwendung in 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 16).

Listing 16: Ausschnitt des StatisticDao Tests

```

1  public class StatisticDaoTest {
2      private static final Statistic STATISTIC = new Statistic(12,
          Category.HISTORY);
3      private static final String TABLENAME = "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, TABLENAME);
13     }
14
15     @Test
1027 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 }

```

1028 Integrationstests

1029 Nachdem die Komponententests der Serverless Anwendung in der Entwicklung denen der
 1030 klassischen Applikation ähneln, verbirgt sich hinter den Integrationstests im Serverless
 1031 Umfeld die angesprochene große Herausforderung. Da der Entwickler über die meisten
 1032 Komponenten keine Kontrolle hat und somit das Verhalten nicht beeinflussen kann, ist

es schwer den kompletten Ablauf zu testen. Die abnehmende Kontrolle 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. Im einfachsten Fall kann hierfür vor dem Start des Testes der Zustand der Datenbank geladen werden, um ihn nach dem Abschluss der Testfälle wieder zurückzusetzen (siehe Listing 17 Z. 13 und 18). Anstatt des Datenbankmocks im Unittest des `StatisticDao` wird eine Verbindung zur DynamoDb aufgebaut (siehe Listing 17 Z. 9-11).

Listing 17: Ausschnitt des StatisticDao Integrationstests

```

1  public class StatisticDaoIntegrationTest {
2      private static final Statistic STATISTIC = new Statistic(12,
          Category.HISTORY);
3      private static final String TABLENAME = "statistics";
4      private StatisticDao statisticDao;
5      private Statistic statistic;
6
7      @Before
8      public void setUp() {
9          DynamoDbClient dynamoDb = DynamoDbClient.builder()
10             .region(Region.EU_CENTRAL_1)
11             .build();
12          statisticDao = new StatisticDao(dynamoDb, TABLENAME);
1045 13          statistic = statisticDao.getStatistic(STATISTIC.getCategory());
14      }
15
16      @After
17      public void cleanUp() {
18          statisticDao.createStatistic(statistic);
19      }
20
21      @Test
22      public void testGetStatistic() {
23          Statistic statistic = statisticDao.getStatistic(STATISTIC.
              getCategory());
24          assertThat(statistic).isNotNull();
25          assertEquals(STATISTIC.getCategory(), statistic.getCategory());
26      }
27  }

```

1046 3.6 Unterschiede in der Entwicklung

1047 Nachdem die Implementierung der beiden prototypischen Anwendungen abgeschlossen ist,
 1048 folgt der Vergleich und die Gegenüberstellung der zwei Entwicklungen. Die Evaluation
 1049 wird anhand der unter 3.1 beschriebenen Kriterien durchgeführt, sodass die Unterschiede
 1050 in der Entwicklung herausgestellt werden.

1051 Für die Durchführung der Evaluation wurden zu den einzelnen Anforderungen Fragen
 1052 definiert, die mittels einer Metrik beantwortet werden und somit zu einem messbaren
 1053 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 notwendig?	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.

Evaluation der klassischen Anwendung

3: Ausführliche Dokumentation des Spring Frameworks mit entsprechenden Beispielen zu den einzelnen Komponenten.

4: Spring übernimmt Aufgaben wie DI, Request Mapping, Datenbankzugriffe(Persistierung)

5: lediglich Spring und Lombok zur Reduzierung von Boilerplate Code

6: Spring ist führend als Framework für Webanwendungen und bietet Module zur Unterstützung in allen Bereichen

7: Kann einfach durch Tools wie Jenkins oder Travis CI eingerichtet werden

10: möglich, einzelne Komponenten können einfach gemockt werden

14: Controller und Service können ohne großen Aufwand erweitert werden. Repository kann einfach hinzugefügt werden

15: fachlich passende Komponenten können jederzeit wieder eingesetzt werden. Beim Controller muss beispielsweise nur der Service hinter dem Interface ausgetauscht werden.

19: Kann manuell über weitere Spring Module implementiert werden. (Spring Boot Autoscaler)

Evaluation der Serverless Anwendung

3: Amazon bietet Dokumentation. Allerdings wenig Beispiele für Standardanwendungsfälle.

4: Alle anfallenden Aufgaben können durch Frameworks und Bibliotheken abgedeckt werden, allerdings nicht so kompakt gesammelt wie bei Spring

5: AWS SDK, Lombok, Dagger, Mockito, Jackson

6: Serverless Framework bietet Unterstützung bei der Serverless Entwicklung. Auf Grund des jungen Alters noch im stetigen Wandel. Deckt nicht so viele Bereiche wie Spring ab.

7: kann ebenfalls mit Travis CI eingerichtet werden

1081 14: Hinzufügen neuer Functions läuft komplett unabhängig vom bisherigen Code ab.

1082 15: fachlich passende Functions können wiederverwendet werden

1083 19: Skalierung bei zunehmender Last wird automatisch vom Anbieter übernommen und

1084 muss nicht extra implementiert werden.

1085 **3.6.1 Implementierungsvorgehen**

1086 **3.6.2 Testen der Anwendung**

1087 **3.6.3 Deployment der Applikation**

1088 **3.6.4 Wechsel zwischen Providern**

1089 **4 Vergleich der beiden Umsetzungen**

1090 **4.1 Vorteile der Serverless Infrastruktur**

1091 **4.2 Nachteile der Serverless Infrastruktur**

1092 **4.3 Abwägung sinnvoller Einsatzmöglichkeiten**

1093 **5 Fazit und Ausblick**

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
- [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
- [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
- [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://>

- 1126 martinfowler.com/articles/microservices.html. – Zuletzt Abgerufen
1127 19.11.2018
- 1128 [Gar99] GARFINKEL, Simson L.: *Architects of the Information Society: Thirty-Five*
1129 *Years of the Laboratory for Computer Science at MIT*. The MIT Press, 1999.
1130 – ISBN 9780262071963
- 1131 [Gig18] GIGLIONE, Marco: Unit and Integration Tests in Spring Boot.
1132 (2018). [https://dzone.com/articles/unit-and-integration-tests-in-](https://dzone.com/articles/unit-and-integration-tests-in-spring-boot-2)
1133 [spring-boot-2](https://dzone.com/articles/unit-and-integration-tests-in-spring-boot-2). – Zuletzt Abgerufen am 13.02.2019
- 1134 [Har02] HARTMANN, Anja K.: *Dienstleistungen im wirtschaftlichen Wan-*
1135 *del: Struktur, Wachstum und Beschäftigung*. 2002 [http://nbn-](http://nbn-resolving.de/urn/resolver.pl?urn=https://nbn-resolving.org/urn:nbn:de:0168-ssoar-121435)
1136 [resolving.de/urn/resolver.pl?urn=https://nbn-resolving.org/urn:](http://nbn-resolving.de/urn/resolver.pl?urn=https://nbn-resolving.org/urn:nbn:de:0168-ssoar-121435)
1137 [nbn:de:0168-ssoar-121435](http://nbn-resolving.de/urn/resolver.pl?urn=https://nbn-resolving.org/urn:nbn:de:0168-ssoar-121435)
- 1138 [Hef16] HEFNAWY, Eslam: Serverless Code Patterns. (2016). [https://serverless.](https://serverless.com/blog/serverless-architecture-code-patterns/)
1139 [com/blog/serverless-architecture-code-patterns/](https://serverless.com/blog/serverless-architecture-code-patterns/). – Zuletzt Abgeru-
1140 fen am 10.01.2019
- 1141 [Her18] HEROKU: Heroku Security. (2018). [https://www.heroku.com/policy/](https://www.heroku.com/policy/security)
1142 [security](https://www.heroku.com/policy/security). – Zuletzt Abgerufen 08.11.2018
- 1143 [Inc18] INC., Serverless: Serverless Guide. (2018). [https://github.com/](https://github.com/serverless/guide)
1144 [serverless/guide](https://github.com/serverless/guide). – Zuletzt Abgerufen am 06.09.2018
- 1145 [Inf18] INFLECTRA: Software Testing Methodologies. (2018). [https://www.](https://www.inflectra.com/Ideas/Topic/Testing-Methodologies.aspx)
1146 [inflectra.com/Ideas/Topic/Testing-Methodologies.aspx](https://www.inflectra.com/Ideas/Topic/Testing-Methodologies.aspx). – Zuletzt Ab-
1147 gerufen am 13.02.2019
- 1148 [Kö17] KÖBLER, Niko: *Serverless Computing in der AWS Cloud*. entwickler.press,
1149 2017. – ISBN 9783868028072
- 1150 [Kar18] KARIA, Bhavya: A quick intro to Dependency Injection: what it is, and when
1151 to use it. (2018). – Zuletzt Abgerufen am 12.02.2019
- 1152 [Kra18] KRATZKE, Nane: A Brief History of Cloud Application Architectures. (2018).
1153 <https://doi.org/10.3390/app8081368>. – Zuletzt Abgerufen am 22.11.2018
- 1154 [Kru04] KRUCHTEN, Philippe: *The Rational Unified Process: An Introduction*.
1155 Addison-Wesley, 2004. – ISBN 0321197704
- 1156 [KS17] KLINGHOLZ, Lukas ; STREIM, Anders: Cloud Computing. (2017).
1157 <https://www.bitkom.org/Presse/Presseinformation/Nutzung-von->

- 1158 Cloud-Computing-in-Unternehmen-boomt.html. – Zuletzt Abgerufen am
1159 01.12.2018
- 1160 [Lit17] LITZEL, Nico: Was ist NoSQL? (2017). [https://www.bigdata-insider.de/
1161 was-ist-nosql-a-615718/](https://www.bigdata-insider.de/was-ist-nosql-a-615718/). – Zuletzt Abgerufen am 21.02.2019
- 1162 [Mar15] MARESCA, Paolo: From Monolithic Three-Tiers Architectures to SOA
1163 vs Microservices. (2015). [https://thetechsolo.wordpress.com/
1164 2015/07/05/from-monolith-three-tiers-architectures-to-soa-
1165 vs-microservices/](https://thetechsolo.wordpress.com/2015/07/05/from-monolith-three-tiers-architectures-to-soa-vs-microservices/). – Zuletzt Abgerufen am 11.02.2019
- 1166 [MG11] MELL, Peter ; GRANCE, Tim: The NIST Definition of Cloud Compu-
1167 ting. (2011). [https://csrc.nist.gov/publications/detail/sp/800-145/
1168 final](https://csrc.nist.gov/publications/detail/sp/800-145/final). – Zuletzt Abgerufen am 03.11.2018
- 1169 [Pol18] POLYMER, Project: Data binding. (2018). [https://polymer-library.
1170 polymer-project.org/2.0/docs/devguide/data-binding](https://polymer-library.polymer-project.org/2.0/docs/devguide/data-binding). – Zuletzt Ab-
1171 gerufen am 19.02.2019
- 1172 [Rö17] RÖWEKAMP, Lars: Serverless Computing, Teil 1: Theorie und Pra-
1173 xis. (2017). [https://www.heise.de/developer/artikel/Serverless-
1174 Computing-Teil-1-Theorie-und-Praxis-3756877.html?seite=all](https://www.heise.de/developer/artikel/Serverless-Computing-Teil-1-Theorie-und-Praxis-3756877.html?seite=all). – Zu-
1175 letzt Abgerufen am 30.08.2018
- 1176 [Ric15] RICHARDS, Mark: *Software Architecture Patterns*. O'Reilly, 2015. – ISBN
1177 9781491924242
- 1178 [Rob18] ROBERTS, Mike: Serverless Architectures. (2018). [https://martinfowler.
1179 com/articles/serverless.html](https://martinfowler.com/articles/serverless.html). – Zuletzt Abgerufen am 30.08.2018
- 1180 [RPMP17] RAI, Gyanendra ; PASRICHA, Prashant ; MALHOTRA, Rakesh ; PAN-
1181 DEY, Santosh: Serverless Architecture: Evolution of a new para-
1182 digm. (2017). [https://www.globallogic.com/gl_news/serverless-
1183 architecture-evolution-of-a-new-paradigm/](https://www.globallogic.com/gl_news/serverless-architecture-evolution-of-a-new-paradigm/). – Zuletzt Abgerufen am
1184 30.08.2018
- 1185 [Sch16] SCHMIDT, Christopher: Webcomponents mit Polymer - Teil 1: Von 0.5 zu 1.x.
1186 (2016). [https://www.innoq.com/de/articles/2016/07/web-components-
1187 mit-polymer%E2%80%933teil-1/](https://www.innoq.com/de/articles/2016/07/web-components-mit-polymer%E2%80%933teil-1/). – Zuletz Abgerufen am 19.02.2019
- 1188 [Sti17] STIGLER, Maddie: *Beginning Serverless Computing: Developing with Amazon
1189 Web Services, Microsoft Azure, and Google Cloud*. Apress, 2017. – ISBN
1190 9781484230831

- 1191 [Swa18] SWARUP, Pulkit: Microservices: Asynchronous Request Response Pat-
1192 tern. (2018). [https://medium.com/@pulkitswarup/microservices-](https://medium.com/@pulkitswarup/microservices-asynchronous-request-response-pattern-6d00ab78abb6)
1193 [asynchronous-request-response-pattern-6d00ab78abb6](https://medium.com/@pulkitswarup/microservices-asynchronous-request-response-pattern-6d00ab78abb6). – Zuletzt Ab-
1194 gerufen am 09.01.2019
- 1195 [Tiw16] TIWARI, Abhishek: Stored Procedure as a Service (SPaaS). (2016). <https://www.abhishek-tiwari.com/stored-procedure-as-a-service-spaas/>.
1196 – Zuletzt Abgerufen am 30.11.2018
- 1197
- 1198 [Tur18] TURVIN, Neil: Serverless vs. Microservices: What you need to know for cloud.
1199 (2018). [https://www.computerweekly.com/blog/Ahead-in-the-Clouds/](https://www.computerweekly.com/blog/Ahead-in-the-Clouds/Serverless-vs-Microservices-What-you-need-to-know-for-cloud)
1200 [Serverless-vs-Microservices-What-you-need-to-know-for-cloud](https://www.computerweekly.com/blog/Ahead-in-the-Clouds/Serverless-vs-Microservices-What-you-need-to-know-for-cloud). –
1201 Zuletzt Abgerufen 15.11.2018
- 1202 [WJ16] WAGNER, Ruben ; JOST, Simon: Webcomponents mit Polymer - Teil 2: Tech-
1203 nische Anwendung. (2016). [https://www.innoq.com/de/articles/2016/](https://www.innoq.com/de/articles/2016/09/web-components-mit-polymer%E2%80%933teil-2/)
1204 [09/web-components-mit-polymer%E2%80%933teil-2/](https://www.innoq.com/de/articles/2016/09/web-components-mit-polymer%E2%80%933teil-2/). – Zuletzt Abgerufen
1205 am 19.02.2019
- 1206 [Wol13] WOLFF, Eberhard: Spring Boot - was ist das, was kann das? (2013). <https://jaxenter.de/spring-boot-2279>. – Zuletzt Abgerufen am 12.02.2019
- 1207
- 1208 [Zar17] ZARWEL, René: *Microservices und technologische Heterogenität: Entwicklung*
1209 *einer sprachunabhängigen Microservice Framework Evaluationsmethode*. 2017

Anhang

A Vollständige Abbildung der Bewertungskriterien

