

# **Design und Implementierung einer Serverless Infrastructure Anwendung beim Cloud Anbieter Amazon Web Services**

**Bachelorarbeit**  
zur Erlangung des Grades *Bachelor of Science*

an der  
Hochschule Niederrhein  
Fachbereich Elektrotechnik und Informatik  
Studiengang *Informatik*

vorgelegt von  
Oktavius Wiesner  
Matrikelnummer: 1082104

Datum: 2. Oktober 2020

Prüfer: Prof. Dr. Peter Davids  
Zweitprüfer: Maik Glatki

# **Zusammenfassung**

Notwendig?

In dieser Bachelorarbeit werden folgende Themen behandelt:

- Cloud Computing und Servicemodelle
- Function as a Service
- Serverless Architektur
- Serverless Dienste beim Cloud Provider Amazon Web Services und Designentscheidungen
- Implementierung einer Webanwendung mit AWS Amplify und React
- Ausblick auf die Zukunft

## **Abstract**

Die vorliegende Bachelorarbeit beschäftigt sich im Detail mit Serverless Architektur und Function as a Service. Zur Verständlichkeit werden die verschiedenen Servicemodelle des Cloud Computings vorgestestellt und verglichen. Die Bachelorarbeit beschränkt sich auf den Cloud Provider Amazon Web Services und der entsprechenden Dienste, die für den Einsatz von Serverless Anwendungen zur Verfügung stehen. Dabei werden AWS-Dienste wie Lambda, Cognito, AppSync, DynamoDB und Amplify genauer untersucht und bewertet. Auf Basis der untersuchten Dienste wird ein Prototyp bei AWS entwickelt und implementiert. Die Programmiersprache ist bei der Implementierung NodeJS bzw. Javascript und React. Das Ziel ist eine moderne Web Applikation für die Mitarbeiter der Mediengruppe RTL, die komplett auf Serverless Architektur basiert und die in der Bachelorarbeit erwähnten Vorteile vollständig ausnutzen kann.

# Eidesstattliche Erklärung

Name: Oktavius Wiesner

Matrikelnr.: 1082104

Titel: Design und Implementierung einer Serverless Infrastructure Anwendung beim  
Cloud Anbieter Amazon Web Services  
Design and implementation of a serverless architecture using the example of  
the cloud provider Amazon Web Services

Ich versichere durch meine Unterschrift, dass die vorliegende Arbeit ausschließlich von mir verfasst wurde. Es wurden keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt.

Die Arbeit besteht aus \_\_\_\_\_ Seiten.

---

Ort, Datum

---

Unterschrift

## **Hinweis**

Bei allen Ausführungen im Folgenden, die auf Personen bezogen sind, meint die gewählte Formulierung beide Geschlechter, auch wenn aus Gründen der sprachlichen Vereinfachung und der besseren Lesbarkeit die männliche Form gewählt wurde.

# **Danksagung**

Hier kommt eine Danksagung...

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Aufgabenstellung . . . . .	1
1.2	Motivation . . . . .	1
1.3	CBC Cologne Broadcasting GmbH . . . . .	2
1.4	Gliederung . . . . .	2
<b>2</b>	<b>Cloud Computing und Serverless</b>	<b>3</b>
2.1	Allgemeines . . . . .	3
2.2	Definition und Erläuterung . . . . .	3
2.3	Cloud Computing-Modelle . . . . .	4
2.3.1	IaaS: Infrastructure as a Service . . . . .	4
2.3.2	CaaS: Container as a Service . . . . .	5
2.3.3	PaaS: Plattform as a Service . . . . .	5
2.3.4	BaaS: Backend as a Service . . . . .	6
2.3.5	FaaS: Function as a Service . . . . .	6
2.3.6	SaaS: Software as a Service . . . . .	7
2.4	Serverless . . . . .	7
2.4.1	Richtlinien für Cloud Provider . . . . .	8
2.4.2	Richtlinien für Entwickler . . . . .	8
2.4.3	Vor- und Nachteile . . . . .	9
2.5	Eignung für die Bachelorarbeit . . . . .	10
<b>3</b>	<b>AWS Serverless Dienste und Designentscheidung</b>	<b>11</b>
3.1	Amazon Web Services Allgemein . . . . .	11
3.2	API . . . . .	11
3.2.1	REST API: AWS API Gateway . . . . .	11
3.2.2	GraphQL API: AWS Appsync . . . . .	13
3.2.3	Entscheidung . . . . .	17
3.3	Datenbanksystem . . . . .	18
3.3.1	Relationale Datenbanken: AWS RDS . . . . .	18
3.3.2	Nicht relationale Datenbanken: AWS DynamoDB . . . . .	19
3.3.3	Entscheidung . . . . .	20
3.4	Authentifizierung . . . . .	21
3.4.1	AWS Cognito . . . . .	21
3.4.2	Alternative und Entscheidung . . . . .	22
3.5	Backend Logik . . . . .	23
3.5.1	AWS Lambda . . . . .	23
3.5.2	Entscheidung . . . . .	24
3.6	Frontend Framework . . . . .	25
3.7	AWS Amplify . . . . .	26
<b>4</b>	<b>Implementierung aller Komponenten</b>	<b>28</b>
4.1	Architekturdiagramm . . . . .	28
4.2	AWS Accountstruktur . . . . .	29
4.3	Amplify Voraussetzungen . . . . .	30

4.3.1	Einrichtung Amplify CLI . . . . .	30
4.3.2	Einrichtung React Projekt . . . . .	31
4.3.3	Konfiguration Amplify . . . . .	31
4.4	Implementierung API und Datenbank . . . . .	32
4.5	Implementierung Backend-Logik . . . . .	34
4.5.1	Accountübergreifender Zugriff . . . . .	35
4.5.2	Asynchrone Verarbeitung mit NodeJS . . . . .	37
4.5.3	Verifizierung Lambda-Funktion . . . . .	38
4.6	Implementierung Authentifizierung . . . . .	38
4.6.1	Anmeldung über AzureAD . . . . .	40
4.7	Implementierung Frontend und Hosting . . . . .	40
4.7.1	Programmierung React-Frontend . . . . .	40
4.7.2	Hosting . . . . .	42
<b>5</b>	<b>Fazit</b>	<b>45</b>
5.1	Diskussion . . . . .	45
5.2	Kosten . . . . .	46
5.3	Zusammenfassung . . . . .	47
5.4	Ausblick . . . . .	48
	<b>Literaturverzeichnis</b>	<b>49</b>
	<b>Anhang</b>	<b>A-0</b>
<b>A</b>	<b>Anhang Lambda</b>	<b>A-0</b>
<b>B</b>	<b>Anhang React main.js</b>	<b>A-4</b>
<b>C</b>	<b>Anhang React Greeting.js</b>	<b>A-5</b>
<b>D</b>	<b>Anhang React AccountsList.js</b>	<b>A-5</b>

# 1 Einleitung

## 1.1 Aufgabenstellung

Das Ziel dieser Bachelorarbeit ist es die Serverless Architektur zu bewerten und einen Prototyp für die Abteilung Datacenter and Clouds der Firma CBC Cologne Broadcasting GmbH (Im folgenden „CBC“) zu erstellen. Der Fokus soll hierbei gleichmäßig auf das Design der Anwendung als auch auf der Implementierung gelegt werden.

Im ersten Schritt soll überprüft werden, inwieweit sich die gewünschte Anwendung mit Serverless Diensten realisieren lässt. Anschließend sollen alle potenziellen Dienste des Cloud Providers Amazon Web Services (Im folgenden „AWS“) untersucht werden und die geeignetsten ausgewählt werden.

Bei der Implementierung sollen diese Dienste dann in der Praxis angewandt werden. Für das Frontend soll ein modernes Framework verwendet werden. Die Implementierung soll möglichst allgemein verwirklicht werden, sodass jeder Mitarbeiter der Abteilung Datacenter and Clouds in Zukunft an dieser Anwendung weiter arbeiten kann.

Im Rahmen der Bachelorarbeit sollen alle Grundfunktionen umgesetzt werden, sodass in Zukunft weitere Funktionalitäten leicht ergänzt werden können. Der Prototyp muss über eine Authentifizierung verfügen und Informationen zu bestehenden AWS Accounts (siehe 4.2 AWS Accountstruktur) anzeigen können.

Zu einem besseren Verständnis wie genau der Prototyp aussehen soll, kann es helfen das Bild im Abschnitt 8 *Fertiggestellte Website mit Daten* bereits zu Beginn zu betrachten.

## 1.2 Motivation

Die Abteilung Datacenter and Clouds innerhalb von CBC beschäftigt sich mittlerweile seit einigen Jahren mit Cloud Infrastruktur. Zu Beginn wurden größtenteils dynamisch Linux Server mit einer Datenbank und einem Loadbalancer realisiert (Infrastructure as a Service). Mit der Zeit wurden auch containerbasierte Varianten mit Docker sowie Platform as a Service Lösungen umgesetzt. Function as a Service wurde bisher nur mit dem Dienst Lambda (siehe 3.5.1 AWS Lambda) von AWS realisiert. Mittlerweile gibt es den Wunsch sich intensiv mit Function as a Service zu beschäftigen um vor allem eine schnelle Bereitstellung von Diensten mit geringen Kosten zu ermöglichen. Webanwendungen sollen schneller bereitgestellt werden, ohne dass Server konfiguriert und gewartet werden müssen. Eine genau Erläuterung der genannten „[...] as a Service“ Begriffe befindet sich im Kapitel 2.3 *Cloud Computing-Modelle*.

Da sehr viele Firmen innerhalb der Mediengruppe RTL intensiv mit AWS und anderen Cloud Providern arbeiten ist die Kostenzuweisung unübersichtlich geworden. Jeder Cloud Provider stellt Informationen zu Kosten und Abrechnungen auf unterschiedliche Weise dar und alle Daten müssen unterschiedlich aufbereitet werden. AWS z.B. exportiert die Abrechnungen monatlich in dem Cloud-Speicher S3<sup>1</sup>, wohingegen Google Cloud Plattform alle Daten in ein SQL ähnliches Data Warehouse speichert. Zudem ist es notwendig die Kosten der Mediengruppe RTL Abteilungen zuzuweisen und abzuschreiben.

---

<sup>1</sup>S3 steht für Simple Storage Service und ist ein Objektspeicherservice für eine beliebige Menge von Daten. Daten werden in S3 Buckets abgelegt und können aus dem Internet abgerufen werden. CBC verwendet S3 für viele unterschiedliche Szenarien, darunter auch das Speichern von Videodateien für den Dienst TVNow.



Deshalb besteht innerhalb der Abteilung Datacenter and Clouds der Wunsch nach einer modernen Web Applikation, welche die Informationen der jeweiligen Cloud Provider zentral sammelt und zur Verfügung stellt. Zu den Informationen gehören Daten zu den Kosten, Abrechnungen, verwendete Ressourcen und nach Bedarf weitere. Im Rahmen der Bachelorarbeit soll dafür ein Prototyp entstehen der effizient und einfach in Zukunft um weitere Anforderungen erweitert werden kann.

### **1.3 CBC Cologne Broadcasting GmbH**

Die Bachelorarbeit wird innerhalb der Räumlichkeiten des Unternehmens CBC Cologne Broadcasting GmbH in Köln Deutz realisiert. CBC ist ein Unternehmen der Mediengruppe RTL Deutschland, welches wiederum zur Bertelsmann-Tochter RTL Group in Luxemburg gehört. Neben CBC gehören unter anderem der Werbezeitenvermarkter IP Deutschland sowie die Fernsehsender RTL Television, RTL Nitro, N-TV und Vox zur Mediengruppe RTL.

Mit ca. 550 festen Mitarbeitern ist CBC für die Produktion, Programmverbreitung, Sendeabwicklung sowie die IT Infrastruktur verantwortlich. Neben der Betreuung von Projekten innerhalb der Mediengruppe RTL ist die CBC auch für Produktion und Broadcast der Berichterstattung der Fußball-Bundesliga verantwortlich. Die Abteilung Datacenter and Clouds beschäftigt sich dabei mit Infrastrukturthemen, sowohl On Premises als auch bei den Cloud Providern Amazon Web Services, Microsoft Azure sowie Google Cloud Platform. Beispiele für Projekte, die in der Cloud umgesetzt wurden sind die Streaming-Plattform TVNOW sowie die Internetpräsenz des Nachrichtensenders N-TV. Gegründet wurde die CBC 1994 in Köln, Geschäftsführer ist Thomas Harscheidt.

### **1.4 Gliederung**

Diese Bachelorarbeit gliedert sich in fünf Kapitel.

Im ersten Kapitel wird mit einer Einleitung sowie grundlegenden Informationen zum Inhalt der Bachelorarbeit und den allgemeinen Rahmenbedingungen ein Ausblick auf den weiteren Verlauf gegeben.

Anschließend werden alle grundlegenden Begriffe und Cloud Modelle erläutert und miteinander verglichen, bevor genauer auf das Thema Serverless eingegangen wird. Hier wird auch die Eignung der Architektur thematisiert.

Nach diesem Abschnitt wird im Detail auf die verschiedenen Serverless Dienste beim Cloud Provider Amazon Web Services eingegangen und begründet, warum die ausgewählten Dienste für diese Bachelorarbeit am besten geeignet sind. Zusätzlich wird die Auswahl der Programmiersprache sowie des Frameworks besprochen.

Das darauf folgende vierte Kapitel beschreibt die Implementierung der gewählten Dienste im Detail. Dazu wird der geschriebene Code vorgestellt und der Verlauf aufgezeigt.

Abschließend dient das fünfte Kapitel der Diskussion sowie einen Ausblick in die Zukunft des Systems. Hier befindet sich ebenfalls eine Zusammenfassung inklusive einem Fazit der Bachelorarbeit.

## 2 Cloud Computing und Serverless

### 2.1 Allgemeines

Bevor man sich mit dem Design und der Implementierung einer Serverless Infrastructure Anwendung beschäftigen kann, ist es notwendig sowohl Begriffe wie Cloud Computing und Serverless zu erklären als auch alle wichtigen Cloud Computing-Modelle darzustellen und miteinander zu vergleichen.

Das Modell Function as a Service ist eine logische Fortführung der bisher existierenden Servicemodelle. Somit sind auch diese Grundvoraussetzung für das Thema Serverless. Im folgenden Abschnitt werden all diese erforderlichen Voraussetzungen geschaffen und die einzelnen Cloud Computing-Modelle miteinander verglichen.

### 2.2 Definition und Erläuterung

Der Begriff Cloud Computing beschreibt im Wesentlichen die Bereitstellung von Ressourcen über das Internet. Hierbei kann es sich um jede Art von Ressourcen handeln. Dazu zählt zum Beispiel die Netzwerkinfrastruktur, Server, Speicher, Datenbanken aber auch Software. Anwender zahlen nur die tatsächlich genutzte Leistung und können den Bedarf jederzeit flexibel anpassen. Lokal muss in der Regel nur ein geeigneter Client installiert sein, etwa ein Webbrowser.

Die wichtigsten Vorteile gegenüber einer On Premises-Landschaft sind die schnelle Verfügbarkeit und Skalierung von Diensten, sowie dass keine Kosten im Voraus entstehen. Durch den Einsatz von Cloudtechnologien verlagert sich das Kostenmodell von Capex<sup>2</sup> zu Opex<sup>3</sup>. Mit einem Knopfdruck ist es möglich beliebig<sup>4</sup> viele Ressourcen und Dienste hochzufahren und zu verwenden. Je nach Servicemodell verschiebt sich der Verantwortungsbereich zwischen dem Cloud Provider und dem Anwender. [Mic20a, ]

---

<sup>2</sup>Capex(Capital Expenditure) bezeichnet Investitionen und Ausgaben die einmalig getätigt werden, um den Umsatz zu erhöhen. Dazu gehört zum Beispiel die Anschaffung von neuer Hardware.

<sup>3</sup>Opex(Operational Expenditure) bezeichnet wiederkehrende Kosten wie Verwaltungs- und Betriebskosten. Auch fallen mittlerweile viele Softwarelizenzen regelmäßig an.

<sup>4</sup>Das Wort beliebig ist nicht sinngemäß zu verstehen. Auch Cloud Provider haben nur eine endliche Anzahl von Kapazitäten und geben den Kunden diese nicht in vollem Ausmaß frei. Jede Umgebung eines Kunden hat bestimmte Limitierungen pro Service, die jedoch bei Bedarf eventuell angepasst werden können. AWS erlaubt beispielsweise pro Region und Account 2880 vCPUs für Standardinstanzen

## 2.3 Cloud Computing-Modelle

Zur Veranschaulichung der unterschiedlichen Servicemodelle dient folgende Grafik. Im Anschluss werden die einzelnen Modelle im Detail beschrieben.

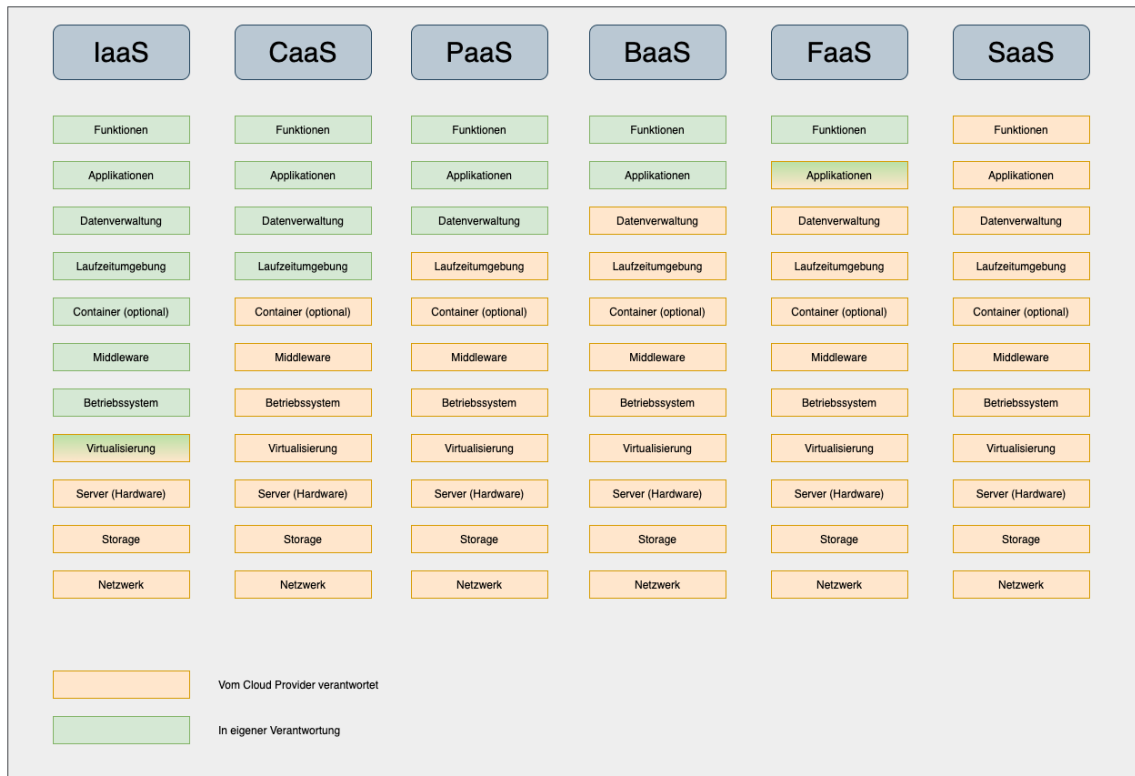


Abbildung 1: Übersicht der Servicemodelle

### 2.3.1 IaaS: Infrastructure as a Service

Unter dem Begriff Infrastructure as a Service, kurz IaaS, versteht man Infrastruktur bei der Administratoren bzw. Anwender sowohl die größte Kontrolle als auch die höchste Verantwortung tragen. Der Cloud Provider kümmert sich nur um die nötige Infrastruktur um Computing-Ressourcen über das Internet zur Verfügung stellen zu können. Dazu gehört der Erwerb und Betrieb von Servern, Switchen, Routern und Speichersystemen und der Virtualisierung von Maschinen. (siehe Abbildung *Cloud Computing-Modelle*). Dieses Modell ist am ehesten mit einer On Premises Infrastruktur vergleichbar und bietet eine hohe Vertrautheit und Ähnlichkeit zu diesen Systemen. Im direkten Vergleich muss man bei einer Cloud-Lösung, irrelevant welches Servicemodell, kein Investitionsrisiko eingehen. Ist man sich unsicher über die benötigte Rechenleistung oder schwankt diese häufig kann, man jederzeit Ressourcen terminieren und Geld sparen.

Das Betriebssystem inklusive aller damit verbunden Updates, die Regelung aller Zugriffe sowie die Installation und Wartung von Software liegen im Verantwortungsbereich des Kunden. Dieses Modell wird häufig verwendet, wenn man eine Hybridlösung mit einer On Premises Landschaft aufbauen möchte oder die eigene Applikation noch nicht für die Anwendung in der Cloud optimiert wurde, man diese aber trotzdem in die Cloud migrieren möchte. [Mic20a, ]

### 2.3.2 CaaS: Container as a Service

Container as a Service bezeichnet das Bereitstellen sämtlicher Ressourcen zur Verwaltung von Containern und der dort installierten Software. Unter Container versteht man ein „Standard-softwarepaket“ welches „den Code einer Anwendung zusammen mit den zugehörigen Konfigurationsdateien, Bibliotheken und den für die Ausführung der Anwendung erforderlichen Abhängigkeiten“ bündelt. [Mic20c, ] Dieser gebündelte Container ist leicht portierbar und die Ausführung erfolgt in einer konsistenten Umgebung.

Der bekannteste Dienst, um solche Pakete bereitzustellen, Docker unterstützt Windows und Linuxsysteme, sowohl OnPremises als auch in der Cloud.

Das zugrunde liegende Betriebssystem und die Hardware werden für den Container abstrahiert. Dies führt zu einer kürzen Entwicklungszeit, da jeder Container reproduzierbar ist, und nicht individuell an das Betriebssystem angepasst werden muss. Auch können mehrere Container dasselbe Host Betriebssystem nutzen und sind dadurch schlanker und schneller einsatzbereit als klassische virtuelle Maschinen. Außerdem erleichtert es das Management von Patches sowie die Sicherstellung der Hochverfügbarkeit. Die Skalierung der Container kann der Cloud Provider übernehmen. Auch die Images lassen sich dort hosten und automatisiert deployen. In der Regel kommt Docker Swarm oder das ursprünglich von Google entwickelte Kubernetes zum Einsatz. Mittlerweile ist Kubernetes Open Source und wird von der Cloud Native Computing Foundation weiterentwickelt.

Typischerweise werden Container as a Service zur Bereitstellung von Microservices verwendet. Das Container as a Service Angebot des Cloud Providers AWS heißt ECS (Elastic Container Service). AWS bietet Nutzern die Möglichkeit ihre Container entweder Serverless oder auf EC2-Instanzen<sup>5</sup> zu hosten. [Ama20w, ]

### 2.3.3 PaaS: Plattform as a Service

Wie man bereits der Abbildung entnehmen kann, werden bei dem Servicemodell Plattform as a Service auch die Bereiche um die Containerbereitstellung sowie die Laufzeitumgebung von dem jeweiligen Cloud Provider übernommen. Für Kunden bzw. Anwender dieses Modells entfällt somit die gesamte Verwaltung der zugrunde liegenden Infrastruktur. Im Fokus liegt das Bereitstellen einer Plattform, die das schnelle und kostengünstige Entwickeln von Anwendungen ermöglicht.

Anders als bei den vorherigen Modellen besteht hier keine Möglichkeit auf das Betriebssystem oder die Middleware zuzugreifen. Der Dienst muss mittels einer API oder einer Weboberfläche angesprochen werden. Dem Kunden werden zusätzliche Optionen zur Verfügung gestellt um leichter Testumgebungen erstellen zu können. Auch gibt es hier bereits vorinstallierte Dienste für Monitoring oder auch Alarmer. Als Beispiele lassen sich hier etwa GitHub, Google App Engine oder AWS Elastic Beanstalk nennen.

Elastic Beanstalk ist ein Service zum Bereitstellen von Webanwendungen. Da der Cloud Provider deutlich mehr Aufgaben übernimmt, kann der Kunde auch zum Beispiel nicht mehr alle Programmiersprachen verwenden, sondern muss sich auf eine von Amazon unterstützte festle-

---

<sup>5</sup>Amazon Elastic Compute Cloud-Instances(EC2) ist ein AWS Service zur Bereitstellung von virtuellen Maschinen in der Cloud. Es gibt viele verschiedene Instanztypen für jede Art von Anforderung. Zum Beispiel bietet AWS CPU, GPU oder RAM optimierte Instanzen in unterschiedlichen Größen an.

gen. Im Gegensatz dazu muss nur noch der Quellcode hochgeladen werden und die Anwendung könnte auf Wunsch bereits im Internet erreichbar sein, ohne sich mit Themen wie Skalierbarkeit, Hochverfügbarkeit beschäftigen zu müssen. Für komplexere Aufgaben oder spezielle Anforderungen ist der Dienst eventuell weniger geeignet. Direkte Änderungen am System, wie eine Anpassung des Logverhaltens von Nginx, ist nicht möglich. Dafür bietet der Dienst Nutzern einen besonders leichten Start in die Entwicklung.

GitHub ist ein Web-basierter Dienst, der öffentlich im Internet erreichbar ist, und Git für die Versionsverwaltung bereitstellt sowie weitere Funktionen zugänglich macht. Github wurde 2008 gegründet und 2018 von Microsoft aufgekauft. Auf dieser Plattform kann jede Person ihren Code(bzw. Dateien) veröffentlichen und teilen. Jeder Benutzer hat die Möglichkeit jedes andere öffentliche Repository zu klonen und selbst daran zu arbeiten oder auch dem Besitzer eines Repositories Anpassungen an den Code anzubieten. GitHub bietet viele Integrationen zu Cloud Providern an. So ist es möglich eine CI/CD<sup>6</sup> Pipeline aufzubauen um voll automatisiert Ressourcen in der Cloud hochfahren zu können. [Mor20, ]

### 2.3.4 BaaS: Backend as a Service

Dieses Servicemodell beinhaltet alle benötigten Dienste, um ein Backend für Entwickler zur Verfügung zu stellen. Entwicklern wird zum Beispiel ein Endpunkt bereitgestellt um Funktionen wie Push-Benachrichtigungen oder eine Social Media Integration verwenden zu können. Auch wird eine Datenspeicherung in relationale oder nicht relationale Datenbanken sowie das Hosting von Websites angeboten. Insgesamt gibt es einige Überschneidungen zu den anderen Servicemodellen. Backend as a Service wird genau wie Function as a Service als Serverless Dienst angeboten, hat jedoch keinen eventbasierten Ansatz oder die Möglichkeit ein Frontend anzubieten. Genau wie bei Plattform as a Service werden viele Funktionen direkt vom Betreiber angeboten.

Häufig wird dieses Modell in Kombinationen mit Function as a Service genutzt. Beide Modelle ergänzen sich und können in Kombination eine vollwertige Webanwendungen ermöglichen (siehe 2.4 *Serverless*). Der Backend as a Service Provider Backendless bietet Nutzern die Möglichkeit Datenstrukturen abzuspeichern, Geolocation zu nutzen, Benutzer zu verwalten sowie analytische Auswertungen durchzuführen. Unternehmen wie Kellogs, Vodafone oder Dell verwenden Backendless. Der Dienst eignet sich auch für Social Media Applikationen oder Spiele. Das Spiel TopAnimals nutzt die vom Dienst bereitgestellten Features wie die API, Datenbank oder die Social Media Integrationen voll aus. [Bac20b, ]

### 2.3.5 FaaS: Function as a Service

Function as a Service überlässt dem Nutzer nur die Verantwortung über die Business Logik und das Frontend. Dieses Modell ist der Kern einer Serverless Anwendung und wird deshalb auch im weiteren Verlauf der Bachelorarbeit für die Implementierung verwendet. Wichtigstes Prinzip ist, dass der zuvor hochgeladene Code bei bestimmten festgelegten Events<sup>7</sup> reagiert und ausgeführt wird. Diese eventbasierten Funktionen ermöglichen dem Entwickler den Code in

<sup>6</sup>CI/CD steht für Continuous Integration, Continuous Delivery und Continuous Deployment und beschreibt eine Brücke zwischen Integrität von Daten und Automatisierung von Prozessen. Mit Continuous Integration wird, durch Zusammenführen und Testen, stets die Integrität des Quellcodes geprüft. Durch Continuous Deployment kann ein Softwarepaket automatisch in beliebige Umgebungen deployed werden.

<sup>7</sup>Event, oder auch Ereignis, meint das Reagieren auf eine Statusänderung einer bestimmten Ressource.

Handumdrehen auszuführen und zu testen. Zudem sind sie stateless bzw. zustandslos. Bei einem zustandslosen Prozess gibt es keine Verweise oder Kenntnisse über bisherige Ereignisse. Dieser Prozess ist isoliert. Beispielfür für zustandslose Kommunikation ist ein Web- oder Druckserver. Jede Anfrage ist erst einmal unabhängig von bisherigen oder zukünftigen Anfragen. [Red20, ]

Aufgrund der Zustandslosigkeit kann eine Funktion beliebig häufig kopiert und gestartet werden. Dadurch ist es auch besonders einfach diese Funktionen zu skalieren oder Hochverfügbar bereitzustellen. Diese Aufgaben muss der Cloud Provider übernehmen.

Insgesamt bietet Function as a Service eine simplifizierte und automatisch skalierte Möglichkeit eventbasierte Funktionen zu erzeugen. Nutzer können sich auf die Entwicklung ihre Anwendung konzentrieren. Auch preislich ist der Function as a Service Ansatz häufig lohnenswert, vorausgesetzt die Anwendung ist auch für den Einsatz optimiert. Es muss im Gegensatz zu den anderen Modellen gar keine Pauschale für jegliche Infrastruktur bezahlt werden, sondern nur die tatsächliche Nutzung der Funktionen. Wird ein Code also nie ausgelöst, und somit auch nie ausgeführt, entstehen auch keine Kosten. Prominente Beispiele sind Azure Cloud Functions, Google Cloud Functions und AWS Lambda. Der Dienst AWS Lambda wird für die Implementierung der Anwendung verwendet und im Abschnitt 3.5.1 *AWS Lambda* im Detail erläutert. Im Abschnitt 4.5 *Implementierung Backend-Logik* folgt die Implementierung. [Ama20y, ]

### 2.3.6 SaaS: Software as a Service

Bei Software as a Service hat der Kunde nur noch Zugriff über eine Weboberfläche. Der Unterschied liegt jedoch darin, dass dem Nutzer ein Zugang zu einer bestimmten Software bereitgestellt wird und er keine mehr selbst entwickeln muss. Deshalb ist es auch die populärste und zugänglichste Form von Cloud Computing. Anders als bei Plattform as a Service wird hier häufig pro Benutzer und Zeitraum abgerechnet. Bei Plattform as a Service kann die Abrechnung auch pro Datenmenge und erforderliche Leistung erfolgen. Vorallem Endanwender ohne tiefgreifende IT-Kenntnisse können viele solcher Dienste nutzen und davon profitieren. [Mic20d, ]

Bekannte Vertreter sind Microsoft 365, GSuite oder auch Slack. Über Microsoft 365 können Anwender die gesamte Microsoft Office Suite sowie Email-Lösungen des Unternehmens nutzen. Anwender haben bei diesem Modell das geringste Investitionsrisiko sowie den geringsten Aufwand zur Administration. Auf der anderen Seite besteht die Gefahr eine stärkere Abhängigkeit von dem jeweiligen Betreiber zu haben sowie weniger Anpassungsmöglichkeiten als bei den bisher vorgestellten Modellen. Ein sehr großes Abhängigkeitsverhältnis wird auch als Vendor-Lock-In bezeichnet. Dort wird der Kunde so stark an das Produkt gebunden, dass es er nur schwer Produkt bzw. Anbieter wechseln kann.

## 2.4 Serverless

Nachdem die wichtigsten Servicemodelle vorgestellt wurden, ist es wichtig den Begriff Serverless zu präzisieren.

Es ist zweifelsohne nicht möglich Dienste in der Cloud zu verwenden, ohne irgendeine Art von Server zu beanspruchen. Der Cloud Provider abstrahiert diese jedoch so weit, dass sich Nutzer keine Gedanken über die Infrastruktur, das Betriebssystem oder selbst die Laufzeitumgebung mehr machen muss. Wie bereits im zuvor beschriebenen Bild verlagert sich die betriebliche Verantwortung immer mehr zum Cloud Provider und der Anwender kann sich ganz auf seine Applikation konzentrieren.

Serverless ist eine Kombination aus den Ansätzen Function as a Service ergänzend mit Backend as a Service, welche bestimmte Regeln befolgen muss. Die Eventbasierten Funktionen stellen die Logik der Anwendung dar und können im Zusammenspiel mit Backend Diensten wie Datenbanken oder Authentifizierungsdiensten eine vollwertige Applikation erzeugen.

### 2.4.1 Richtlinien für Cloud Provider

Ab wann gilt etwas als Serverless?

Das Whitepaper von Amazon *Amazon Web Services – Serverless Architectures with AWS Lambda* [Ama20s, Seitenzahl 1] stellt hierzu folgende vier Aspekte auf die zutreffen sollten:

- 1) „No server management – You don’t have to provision or maintain any servers.“  
→ Die Verwaltung der Server und Laufzeitumgebung übernimmt der Cloud Provider.
- 2) „Flexible scaling – You can scale your application automatically or by adjusting its capacity through toggling the units of consumption (for example, throughput, memory) rather than units of individual servers.“  
→ Der Cloud Provider muss eine flexible und automatisierte Skalierung ermöglichen. Dabei können eventuell bestimmte Parameter für die Kapazität gesetzt werden.
- 3) „High availability – Serverless applications have built-in availability and fault tolerance.“  
→ Der Cloud Provider stellt für seinen Dienst eine Hochverfügbarkeit und Fehlertoleranz automatisch zur Verfügung.
- 4) „No idle capacity – You don’t have to pay for idle capacity.“ [...] „There is no charge when your code isn’t running.“  
→ Kosten dürfen nur bei Nutzung entstehen. Im Leerlauf kostet der Dienst nichts.

### 2.4.2 Richtlinien für Entwickler

Auch Entwickler sollten sich an bestimmte Prinzipien und Leitlinien halten um bestmöglich von der Architektur profitieren zu können.

Zum einen sollte jeder geschriebene Code isoliert und unabhängig voneinander ausführbar sein, um so auch die Zustandslosigkeit gewährleisten zu können. Zudem muss der Entwickler darauf achten alle wichtigen Daten außerhalb des Funktionskontextes persistent zu speichern, da sie beim nächsten Ausführen verloren gehen. Die einzelnen Events dürfen dabei nicht voneinander abhängig sein und sollten eigenständig durchlaufen können. Es muss ein Push basiertes Konzept entwickelt werden um Events ordnungsgemäß auslösen zu können. Das bedeutet, dass die Funktion nicht selbst entscheiden soll in welchem Umfang sie eine Anfrage abarbeiten soll und ohne Benutzereingaben auskommen muss.

Der Code muss schnell ausführbar sein und dem Single-Responsibility-Prinzip folgen. Laut dem Single-Responsibility-Prinzip darf eine Klasse oder Funktion immer nur eine einzige Verantwortung haben. Es darf nie mehr als einen Grund geben eine Funktion auszuführen. Verwendet wird das Prinzip um übersichtlichen und leicht erweiterbaren Code zu schaffen. Je nach Cloud-Provider gibt es bei Funktionen häufig eine maximale Ausführungszeit. AWS erlaubt seinem Lambda-Dienst eine maximale Zeit von 15 Minuten. Auch der Speicherverbrauch wird bei AWS auf 3 GB RAM limitiert. Wird die Zeit oder der Speicherverbrauch überschritten, terminiert die Ausführung der Funktion sofort. [Ama20m, ]

### 2.4.3 Vor- und Nachteile

Serverless Architektur ermöglicht Entwicklern eine schnellere und einfachere Bereitstellung von Diensten in der Cloud. Viele Anforderungen werden bereits vom Cloud Provider übernommen und müssen nicht beachtet werden. Dazu gehört das Verwalten und Administrieren von virtuellen Maschinen, Hochverfügbarkeit oder Fehlertoleranz. Zudem gibt es keine Investitionskosten(Capex), sondern nur operative Kosten(Opex). Die operativen Kosten lassen sich jedoch jederzeit steuern und auf Wunsch auch gänzlich beseitigen, zum Beispiel, falls das Projekt eingestellt wird. Im Vorhinein ist es nicht notwendig Kapazitäten zu berechnen oder große finanzielle Risiken einzugehen.

Dadurch, dass bereits viele technische Voraussetzungen vom Cloud Provider übernommen werden, ist die Einstiegshürde bis zum Entwickeln geringer. Weiter gibt es weniger Abhängigkeiten zu anderen Abteilungen, da jeder einzelne die gesamte Anwendung selbst erstellen und verwalten kann. Die Realisierung von Projekten mithilfe von Serverless Architektur unterstützt dabei die DevOps-Philosophie und sorgt für schnellere Reaktionen auf Änderungswünsche. DevOps setzt sich aus den Begriffen Development und Operations zusammen und hat mehrere Ansätze und Bedeutungen. Eine mögliche Variante steht für das Zusammenschmelzen von Entwicklung, Betrieb und den operativen Aufgaben einer Anwendung. Das Ziel ist eine effektivere Zusammenarbeit aller Teilbereiche und eine erhöhte Qualität der Anwendung. Neben der Realisierung ist auch eine Anpassung an den Markt erleichtert. Folglich reduziert sich die Dauer bis zur Veröffentlichung von Testumgebungen oder sogar des gesamten Projektes, auch als Time-To-Market bekannt. Nachdem eine Version abgeschlossen wird, startet daraufhin direkt die nächste Iteration. Der Entwicklungsprozess ist kreisförmig anstatt linear. [Mic20b, ]

Dadurch, dass das Backend nur für einzelne Anforderungen genutzt wird, beinhaltet das Frontend bei Serverless Applikationen in der Regel mehr Funktionen und Logik. Zum Beispiel kann direkt mit Drittanbieter APIs kommuniziert werden ohne das Backend nutzen zu müssen. Das führt zu einem schnelleren Feedback für den Anwender und somit zu einer verbesserten Benutzererfahrung.

Da die Serverless Architektur noch nicht so lange im Einsatz ist wie die bewährten Alternativen, gibt es zum Beispiel noch nicht viele Security Tools oder Frameworks die diese Architektur voll unterstützen. Das auf Security spezialisierte Unternehmen Checkpoint hat erst im Jahr 2019, durch Zukauf der Firma Protego, ein Tool angeboten. Protego selbst wurde 2017 gegründet und spezialisiert sich auf Function as a Service Dienste wie AWS Lambda oder Azure Functions. [Mic19, ]

Neben Protego gibt es zwar noch weitere Security Tools die mit Serverless Diensten umgehen können, jedoch unterstützen sie entweder noch nicht alle Cloud Provider oder sind in den Funktionalitäten eingeschränkt. [Cha19, ]

Weiterhin ist es oft mit großem Aufwand verbunden eine bereits bestehende Applikation zu einer Serverless Architektur zu migrieren. Oft muss der gesamte Ansatz neu konzipiert und umgesetzt werden, da die meisten bestehenden Anwendungen weder eventbasiert noch zustandslos sind.

Ein zusätzlicher Nachteil ist die fehlende Kontrolle über das unterliegende System. Zwar wird einem hierdurch viel Arbeit abgenommen, jedoch kann es immer wieder einen Anwendungsfall geben, in dem mehr Kontrolle über die Hard- und Software benötigt wird. Zum Beispiel ist man an bestimmte Versionen der Programmiersprachen gebunden, die der Cloud Provider



vorgibt. Zudem ist es schwierig Serverless Funktionen von einem Cloud-Provider zum anderen zu migrieren, da jeder Cloud Provider eine unterschiedliche Implementierung besitzt und stark von weiteren Diensten abhängig ist. Zum Beispiel ist AWS Lambda mit vielen weiteren Diensten integriert und verbunden. Diese Funktionalität lässt sich nicht unverändert übertragen. Hier findet ebenfalls ein Vendor Lock-In statt.

Insgesamt können Serverless Anwendungen erhebliche Vorteile im Vergleich zu klassischen Anwendungen haben, jedoch ist es nicht immer möglich oder profitabel auf diese zu setzen. Vor allem bei größeren, bereits bestehenden Anwendung lohnt es sich nicht immer die Architektur zu wechseln, bei neuen und webbasierten Anwendungen kann die Architektur ihre Vorteile voll und ganz ausspielen.

## **2.5 Eignung für die Bachelorarbeit**

Wie bereits im Abschnitt *1.2 Motivation* erwähnt wurde beschäftigt sich die Abteilung Data-center and Clouds seit längerem mit Cloud Computing und hat auch schon Projekte mit unterschiedlichen Servicemodellen realisiert. Function as a Service wurde ebenfalls verwendet, jedoch bisher nicht im Rahmen einer Webanwendung mit einem eigenen Frontend.

Es besteht der Wunsch nach einer Webanwendung, die bestimmte Daten der unterschiedlichen Cloud Provider sammelt und in einem übersichtlichen Frontend anzeigt. Dafür soll kein dedizierter Server benötigt werden oder hohe Kosten entstehen.

Eine Serverless Architektur eignet sich für diese Anforderungen optimal. Es muss keine virtuelle Maschine, kein Betriebssystem und auch keine komplexe Konfigurationen erstellt werden. Die aufgestellten Anforderungen lassen sich ideal in isolierte Funktionen aufteilen. So könnte pro Cloud Provider eine oder mehrere Funktionen bereitgestellt werden, welche komplett autark arbeiten. Der Umfang einer Funktion wäre etwa das Sammeln einer Kostenübersicht bei AWS.

Die gesammelten Daten würden in einer Datenbank gespeichert werden und anschließend von einem Frontend dargestellt. Die Datenbank sollte im besten Falle, wie der Rest der Applikation, vom Cloud Provider verwaltet werden und erforderliche Kapazitäten selbst anpassen können. Da es keine Bestandsdaten gibt, kann die Anwendung auf die Serverless Architektur ohne Probleme optimiert werden. Für das Frontend stehen viele moderne Frameworks zur Verfügung die ohne langjährige Erfahrung genutzt werden können. Desweiteren ist es nicht notwendig die Authentifizierung komplett selbstständig zu schreiben, da bereits passende Dienste und Komponenten existieren.

Auch aus Kostensicht ist die Anwendung bestens geeignet für eine Architektur ohne Serververwaltung. Es gibt keine konstante Auslastung und bei Nichtnutzung fallen auch keine Kosten an.

Durch die Architektur ist man selbst in der Lage jeden einzelnen Schritt selbst zu entwerfen und umzusetzen, ohne Experte mit Tiefenwissen in allen Gebieten zu sein. Viele mühselige Aufgaben werden durch den Cloud Provider übernommen und man kann sich vollumfänglich auf die Anwendung selbst konzentrieren.

Mit welchen Diensten die Anwendung realisiert wird und wie genau die Komponenten konstruiert werden, beschreiben die nächsten Kapitel.

## 3 AWS Serverless Dienste und Designentscheidung

### 3.1 Amazon Web Services Allgemein

Bevor die relevanten Komponenten beschrieben werden, folgt eine kurze Übersicht zum Cloud Provider selbst. Amazon Web Services ist einer der größten Cloud Computing Anbieter der Welt. Das Unternehmen gehört zu 100 Prozent zu Amazon Inc. und bot 2006 erstmals seine Dienste an. CEO ist seit Beginn an Andrew R. Jassy. Zu den größten Kunden gehören Netflix, CocaCola, Spotify, Dropbox und viele weitere. [Ama20x, ]

Mittlerweile stehen mehr als 175 AWS-Dienste in aktuell 24 unterschiedlichen Regionen zur Auswahl. Jede Region besitzt in der Regel drei eigene Verfügbarkeitszonen (englisch: Availability Zones) die geografisch voneinander getrennt sind. Die Regionen sind auf alle Kontinente verteilt. Auch in Deutschland wird mit dem Standort Frankfurt eine eigene Region angeboten. Mit Hinblick auf Datenschutz und der Datenschutz-Grundverordnung soll Amazon die Vorgaben erfüllen können. So speichert selbst die Bundespolizei Bodycam Aufnahmen in der Amazon Cloud. [Til19, ] Es wird berichtet, „dass derzeit noch keine staatliche Infrastruktur zur Verfügung stehe, die die Anforderungen erfülle.“ [Til19, Abschnitt 1]

Amazon Web Services bietet für viele Anwendungsfälle einen oder mehrere passende Services an. Jedes im vorherigen Kapitel besprochene Servicemodell hat mehrere entsprechende AWS-Dienste und es werden laufend neue angekündigt.

Auch für den Bereich Serverless gibt es mehrere Dienste. Die nachfolgenden Abschnitte beschäftigen sich mit dem Design der Webanwendung. Um eine bestmögliche Entscheidung treffen zu können, werden die wichtigsten Dienste untersucht und auf Ihre Eignung geprüft. Falls notwendig wird zuvor die zugrundeliegende Technologie erläutert, sodass ein Verständnis für den jeweiligen Dienst geschaffen werden kann. Anhand der Ergebnisse folgt eine Entscheidung, welche Dienste im Anschluss für die Implementierung genutzt werden.

### 3.2 API

API steht für Application Programming Interface und bezeichnet eine Programmierschnittstelle für die Kommunikation von Diensten. Zur einfacheren Handhabung und höherer Flexibilität sollen Daten über einen standardisierten Weg ausgetauscht werden. Die API definiert dabei die Art und Weise der Kommunikation, also wie Daten angenommen, verarbeitet und wieder zurückgesendet werden.

Für die Webanwendung ist eine API unerlässlich, da der Datenverkehr reguliert werden muss. Amazon bietet zwei Möglichkeiten an eine solche API zu erstellen. Zur Auswahl stehen die Amazon Dienste API Gateway und AppSync. Beide Dienste basieren auf unterschiedlichen Prinzipien mit eigenen Vor- und Nachteilen. Diese Aspekte werden im folgenden Schritt gegenüber gestellt. Anschließend folgt anhand der gewonnenen Erkenntnisse eine begründete Entscheidung, welcher Dienst geeigneter ist.

#### 3.2.1 REST API: AWS API Gateway

Das Grundprinzip der REST-API (REpresentational State Transfer) Architektur ist es eine strukturierte Kombination aus Ressourcen und Methoden zu erhalten. Jede Information, die sich be-

nennen lässt kann eine Ressource sein. Identifiziert werden Ressourcen mithilfe von sogenannten URI (Unified Resource Identifier). Sie geben Inhalte in JSON oder XML zurück. Mittels den zustandslosen HTTP-Methoden GET, POST, PUT und DELETE kann mit Servern im Internet agiert werden. Laut Anforderungen soll sich REST an das Client-Server Modell halten und unabhängig voneinander funktionieren können. Der Client hat jederzeit die Möglichkeit Daten vom Server anzufordern. Dabei muss jede Anfrage des Clients alle benötigten Informationen beinhalten um die Anfrage bearbeiten zu können. Die gewünschten Daten sind immer über eine spezifische URI identifizierbar. Jede Ressource besitzt eine eigene URI. [res20, ]

Beispiel für Abfragen könnten folgendermaßen aussehen:

```
GET /AccountNames/  
GET /AccountNames/123  
GET /AccountNames/456  
POST /Accounts/1
```

Mithilfe des ersten GET Anfrage kann eine Liste von Accounts abgerufen werden, die zweite GET Anfrage liefert nur die Informationen für den Eintrag „123“ der Accountnamen. Um nur zwei Namen zu erhalten, sind dementsprechend auch zwei Abfragen notwendig, da sie separate Ressourcen sind. Eine alternative Option wäre alle Accountnamen zu erhalten und nur die benötigten heraus zu filtern. Die POST Abfrage erlaubt es einen neuen Account anzulegen. Es ist wichtig vorher zu überlegen in welchem Kontext die Daten später benötigt werden. Dank der Zustandslosigkeit wird eine leichtere Skalierung ermöglicht. Anfragen können durch einen Loadbalancer auf mehrere Server verteilt und bearbeitet werden.

Auch eine Cache-Implementierung ist mithilfe von REST leicht umsetzbar. Dank Caching werden häufig abgerufene Daten zwischengespeichert. Ziel ist es die Performance zu steigern und damit gleichzeitig die Anzahl der Abfragen an den Ursprungsserver zu reduzieren. Stellt zum Beispiel ein Browser eine Anfrage, wird zuerst geschaut, ob sich die benötigten Daten im Cache befinden. Falls ja werden sie aus dem Cache geliefert und der Server wird nicht kontaktiert. Fehlen die Daten im Cache werden sie von dem Server abgefragt und anschließend dem Client zurückgegeben und zusätzlich im Cache gespeichert, sodass jede weitere Anfrage direkt aus dem Cache ausgeliefert werden kann. Der Client stellt seine Anfrage dabei normal an den Zielserver, für ihn ist der Cache nicht direkt sichtbar. Beim HTTP-Caching entscheidet der Cache-Control Header<sup>8</sup> ob und wie lange ein Datensatz als gültig gilt. Ein gesetzter Header mit dem Wert Cache-Control: max-age=3600 teilt dem Client etwa mit, dass die Datei eine Stunde gültig ist, und in diesem Zeitraum nicht erneut abgerufen werden muss. [Mar20, ]

Wächst eine Applikation immer weiter entstehen auch immer neue API Endpunkte. Besteht die Anforderung alle Daten abzufragen, wären mehrere separate Anfragen notwendig. Es ist zudem nicht möglich die angeforderten Daten genauer zu spezifizieren. Mittels GET Abfrage erhält man immer alle Daten die die API zurückgibt, egal wie viel man tatsächlich davon benötigt. Dieses Phänomen nennt sich Over-fetching bzw. Under-fetching. Beim Over-fetching werden mehr Daten bezogen als die Anwendung eigentlich benötigt und beim Under-fetching müssen mehrere Anfragen an den Server gesendet werden um die benötigte Menge an Daten zu erhalten. [Bac20a, ]

AWS API Gateway ist ein vollständig verwalteter Service um solche API Endpunkte zu er-

<sup>8</sup>Ein HTTP-Header ist eine Zusatzinformation die zwischen Client und Server ausgetauscht werden kann.

zeugen. Dabei steht er als zentrale Schnittstelle zwischen Endgeräten und Backend-Diensten. Zu den Endgeräten zählen Webanwendungen, Mobilgeräte oder auch IoT<sup>9</sup> Geräte. Eingehende Abfragen werden mit API Gateway an einen ausgewählten Dienst weitergeleitet. Für jede unterschiedliche Abfrage lässt sich ein anderes Verhalten konfigurieren. Mithilfe des Dienstes lässt sich beispielhaft die zuvor beschriebene GET Abfrage an eine Datenbank weiterleiten, die die gewünschten Daten zurückgibt. Die POST Abfrage könnte an eine Lambda-Funktion weitergeleitet werden, welche die Daten verarbeitet und an einem anderen Ort speichert. AWS API Gateway arbeitet dabei vollkommen Serverless und skaliert automatisch mit den Anforderungen. Neben Zugriff auf den Lambda-Dienst, ist es auch möglich mit anderen Diensten wie EC2, S3 oder auch DynamoDB (siehe 3.3.2 *Nicht relationale Datenbanken: AWS DynamoDB*) zu kommunizieren.

Zusätzlich bietet der Dienst viele weitere Funktionalitäten. Dazu gehören CORS-Support<sup>10</sup>, eine Zugriffskontrolle und Einschränkung, oder auch die Verwaltung unterschiedlicher API-Versionen. Es ist auch möglich mit On Premises Servern zu kommunizieren und Anfragen weiterzuleiten. [Ama20b, ]

Die Abrechnung erfolgt anhand von API-Aufrufen und Datenübertragungen die in Richtung Internet verlaufen. Eine Pauschale gibt es bei den Aufrufen nicht. Für die ersten 333 Millionen Anfragen pro Monat kosten in der Region Frankfurt eine Millionen API-Aufrufe 3,70 USD. Wird diese Grenze überschritten sinkt der Preis in weiteren Etappen. Bei über 20 Milliarden Aufrufen kosten eine Millionen API-Aufrufe nur noch 1,72 USD. Darüber hinaus entstehen pauschale Kosten, falls man Caching nutzen möchte und eine bessere Leistung für seine API benötigt. Selbst wenn es keine Anfragen an die API gibt, würden durch das Caching dauerhaft Kosten verursacht werden, welches eine Diskrepanz zu dem Serverless Ansatz aufweist. Hier kosten 0,5 GB an Zwischenspeicher 0,02 USD pro Stunde. Je höher der Speicher, desto höher auch der Stundensatz. [Ama20c, ]

### 3.2.2 GraphQL API: AWS Appsync

2015 wurde GraphQL von Facebook veröffentlicht und 2018 in die Linux Foundation<sup>11</sup> ausgliedert. GraphQL APIs arbeiten nicht mit Ressourcen oder mehreren Endpunkten, sondern mit Typen und Feldern. Häufigste Art von Typen sind Objekttypen, welche ein Objekt mit bestimmten Feldern repräsentiert, zum Beispiel der Typ Accounts mit allen zugehörigen Informationen. Jede Anfrage sollte für gewöhnlich an den Endpunkt /graphql gerichtet sein.

Hauptbestandteil von GraphQL ist das Schema, in dem alle Typen definiert sein müssen. Um unabhängig von Programmiersprachen zu sein, wird das Schema in der eigenen GraphQL Schema Definition Language geschrieben. Das Schema legt die Regeln fest, wie der Client auf Daten zugreifen kann. Neben den bereits erwähnten Objekttypen sind zwei weitere Typenarten besonders wichtig, der Query- und Mutations-Typ. Diese zwei Typen definieren den Einstiegspunkt

---

<sup>9</sup>IoT steht für Internet of Things (deutsch: Internet der Dinge) und bezeichnet ein System von vernetzten Geräten, welche über das Internet miteinander kommunizieren können. Dazu gehören Geräte die ihre Daten selbständig sammeln und zur Verfügung stellen können, etwa auch Alltagsgegenstände wie smarte Thermostate oder digitale Sprachassistenten.

<sup>10</sup>CORS steht für Cross-Origin Resource Sharing und beschreibt einen Mechanismus der mittels zusätzlicher HTTP Header Berechtigungen für Ressourcen vergibt, falls sich diese Ressourcen auf einer anderen Domain befinden als auf der eigenen.

<sup>11</sup>Die Linux Foundation ist eine gemeinnützige Organisation mit Sitz in den USA. Das Ziel ist es Linux zu fördern und das Wachstum zu unterstützen.

jeder GraphQL Anfrage. Die HTTP-Methode GET ähnelt dabei Queries und POST Mutationen. [The20b, ]

Eine Query ist eine einfache Abfrage der Daten, eine Mutation ermöglicht eine Veränderung der Daten. GraphQL erlaubt, anders als REST, nur bestimmte Daten in einer Abfrage abzurufen. Ein Over- bzw. Under-fetching gibt es hier nicht. Im Vergleich zu REST ist GraphQL flexibler und benötigt weniger Änderungen im Backend. Der Client hat mehr Möglichkeiten mit der API zu kommunizieren, da er selbst entscheiden kann, wie er Daten abfragt. Ein Vorteil der stark definierten Typen ist eine geringere Fehlkommunikation zwischen Server und Client. Der Server kann die Anfragen automatisch auf Korrektheit prüfen und fehlerhafte Anfragen bereits zu Beginn der Transaktion ablehnen.

Um Datenobjekte in einem Schema zu erzeugen kann man folgende Struktur nutzen:

```
type Account {
  id: ID!
  accountid: String!
  name: String!
  email: String!
  num: Int!
  status: String!
}
```

Es wird der Typ Account mit sechs Feldern generiert. Den einzelnen Feldern wird zugewiesen, ob es sich um einen String, Integer oder etwas anderes handelt. Mit dem Ausrufezeichen wird ein Feld als zwingend benötigt markiert. Es darf also nicht leer sein. Die einzigartige ID wird vom Server beim Erzeugen von neuen Einträgen automatisch erstellt. Damit das erstellte Objekt Account durch eine Query abgerufen werden kann, muss noch ein entsprechender Query-Typ erzeugt werden:

```
type Query {
  getAccount(id: ID!): Account
}
```

Um diesen Typ zu verwenden muss folgende Query aufgerufen werden:

```
query MyQuery {
  getAccount(id: "MyDesiredID") {
    id
    email
    accountid
    name
    status
  }
}
```

Hierbei ist es unwichtig, ob alle oder nur ein Feld mit angegeben wird. Wird zum Beispiel die Email nicht benötigt, lässt man das Feld weg und die Information wird nicht mehr an den Client übertragen. [The20b, ] Eine HTTP-Abfrage könnte folgendermaßen aussehen [The20c, ]:

```
GET http://meineapi.com/graphql?query={me{name}}
```

Zum Hinzufügen eines neuen Eintrags wäre folgender Mutations-Typ notwendig. Zusätzlich werden mögliche Felder für die Eingabe benötigt.

```
type Mutation {  
  addAccount(userInput: UserInput!): User  
}  
  
input UserInput {  
  id: ID!  
  accountid: String!  
  name: String!  
  email: String!  
  num: Int!  
  status: String!  
}
```

Anschließend kann eine Mutation wie folgt aussehen:

```
mutation {  
  addAccount(  
    name: "AWS-XXX",  
    num: 145,  
    email: "oktavius@cbc.de",  
    accountid: "123456789",  
    status: "ACTIVE")  
  {  
    id  
  }  
}
```

Es ist außerdem möglich in einer Mutation auch direkt Daten abzufragen. Beim Erstellen eines neuen Accounts kann man gleichzeitig die vom Server angelegte ID abfragen und überprüfen. [The20b, ]

Neben den bisher genannten Funktion bietet GraphQL noch Subscriptions an. Diese ermöglichen dem Client Echtzeitbenachrichtigungen zu erhalten, sobald dem Backend neue Daten hinzugefügt worden sind. Subscriptions werden durch Mutationen ausgelöst, wobei das AWS AppSync SDK sich um die Verwaltung dieser kümmert. Realisiert wird die Echtzeitverarbeitung mithilfe von Websockets<sup>12</sup> oder dem MQTT-Protokoll<sup>13</sup> in Kombination mit WebSockets.

<sup>12</sup>Unter WebSockets versteht man ein Protokoll, dass eine bidirektionale Kommunikation zwischen Client und Server ermöglicht. Nachdem ein Client eine initiale Anfrage gesendet hat, kann die Verbindung offen bleiben und so der Server selbstständig Informationen verschicken. [Moz20b, ]

<sup>13</sup>MQTT steht für Message Queue Telemetry Transport und dient zum zuverlässigen Nachrichtenaustausch. Abgebrochene Verbindungen können wieder aufgebaut werden und verlorene Nachrichten erneut ausgeliefert. [Ele20, ]

[Ama20a, ] Dadurch dass GraphQL, anders als REST, nur einen einzelnen Endpunkt anbietet, ist eine Cache Implementierung problematischer. Bei GraphQL muss zusätzlicher Code im Client implementiert werden oder das Schema angepasst werden. Der Dienst Apollo Server<sup>14</sup> ermöglicht es eine GraphQL API bereitzustellen, die bereits Cache Header unterstützt. Dafür muss die Option `cacheControl` in das Schema eingebaut werden. Es ist möglich die Option für jedes individuelle Feld oder den gesamten Typen zu setzen. [Apo20, ] [The20a, ] [The20b, ]

Mithilfe von AWS Appsync ist es möglich APIs bereitzustellen die auf einem GraphQL Server basieren. Dabei ist AppSync in viele weitere AWS Dienste integriert. Das folgende Schaubild von Amazon zeigt die Funktionsweise des Dienstes genauer.

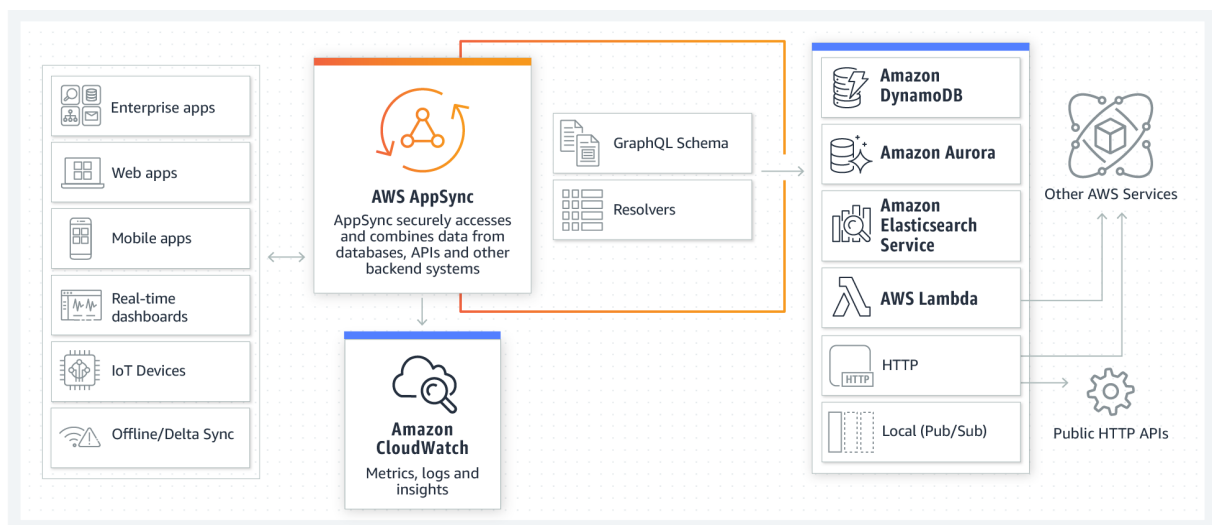


Abbildung 2: Funktionsweise von AppSync [Ama20j, ]

AppSync besteht aus den Hauptkomponenten GraphQL Schema und Resolver. Damit GraphQL weiß welche Operationen es bei Anfragen durchführen soll, werden Resolver benötigt. Resolver sind Funktionen, welche bestimmen wie jedes einzelne Feld im Schema weiterverarbeitet wird. Es ist nicht möglich Resolver direkt im Schema anzugeben, sie müssen außerhalb erstellt werden. Im Resolver kann angegeben werden, dass eine Funktion, etwa AWS Lambda, aufgerufen werden soll. Ein DynamoDB-Resolver ist in der Lage alle benötigten DynamoDB-Tabellen inklusive benötigter Berechtigungen aufzusetzen. Es ist nur notwendig im Schema die Objekttypen festzulegen. Daraufhin verknüpft der Resolver die Datenbank mit der API und erstellt alle möglichen Queries und Mutations automatisch. Infolgedessen ist es mit AppSync nicht notwendig Query-Typen oder Mutations-Typen zu erstellen. Darüber hinaus ist es möglich auch eigene Resolver zu erstellen, falls dies benötigt wird.

Zusätzlich kann AppSync eine Autorisierung mit dem Dienst AWS Cognito (siehe 3.4.1 AWS Cognito) ermöglichen. Die Autorisierung von Clients ist mit API Keys<sup>15</sup>, AWS IAM Cre-

<sup>14</sup>Apollo Server ist ein weit verbreiteter Open Source GraphQL Server, der mit jedem GraphQL Client kompatibel ist.

<sup>15</sup>Der API Key ist ein Authentifizierungsschlüssel und wird bei jeder Anfrage übertragen. AWS empfiehlt ihn bei AppSync nur für Entwicklungsumgebungen, da er leicht kompromittierbar ist und keine Aufteilung von Berechtigungen ermöglicht.

dentials<sup>16</sup>, OIDC Tokens<sup>17</sup> oder einem AWS Cognito User Pool (siehe 3.4.1 AWS Cognito) möglich. [Ste19, ]

Sämtliche Protokolldaten sowie Metriken können in dem Dienst AWS CloudWatch Logs<sup>18</sup> gespeichert und ausgewertet werden. Ähnlich wie AWS API Gateway unterstützt AppSync mehrere unterschiedliche Endgeräte.

Auch Echtzeitanwendungen können mit AppSync und GraphQL Subscriptions realisiert werden. Zusätzlich bietet Amazon ein serverseitiges Caching von Daten an, um direkten Zugriff zu reduzieren und die Geschwindigkeit zu erhöhen. Für das Caching wird der AWS Dienst ElastiCache verwendet, der auf den Speicher Redis<sup>19</sup> basiert. Um Caching nutzen zu können, ist es notwendig einen Instanztypen mit bestimmter Kapazität auszuwählen. Ähnlich wie bei der Preisgestaltung von AWS API Gateway, muss der Cache zeitbasiert bezahlt werden. Der kleinste auswählbare Instanztyp `cache.small` mit einer vCPU und 1,55 GB Arbeitsspeicher kostet zum Beispiel 0,044 USD pro Stunde. Eine Millionen API-Aufrufe kosten 4,00 USD, eine Vergünstigung bei mehr Abrufen gibt es im Vergleich zu API Gateway nicht. [Ama20j, ] [Ama20k, ]

### 3.2.3 Entscheidung

Für das, in dieser Arbeit, gewünschte Projekt wäre eine Implementierung sowohl mit einer REST API als auch mit GraphQL möglich. Beide Dienste für die API sind zudem direkt in Amplify (siehe 3.7 AWS Amplify) integriert und benötigen keine separate Konfiguration.

Mit AWS APIGateway müsste man mehrere unterschiedliche Endpunkte konfigurieren und im Frontend einbauen. Im ersten Schritt würde eine GET Abfrage auf alle Accounts inklusive aller zusätzlichen Daten ausreichen. Die benötigte Datenbank und alle damit verbundenen Autorisierungen müssten manuell eingerichtet werden.

GraphQL bietet sich an, wenn man mehrere Microservices nutzt und alle in einem Schema konsolidieren möchte. Umso größer und komplexer die Anwendung wird, desto mehr spielt GraphQL seine Vorteile aus. Trotz des Wachstums der Daten bleiben die benötigten Konfigurationsänderungen im Vergleich zu REST geringer und übersichtlicher.

Der Umfang der Webanwendung und der benötigten Daten steigt mit zunehmenden Anforderungen. Je nachdem welcher Endanwender auf die Anwendung zugreift, könnte er einen unterschiedlichen Detailgrad der Daten benötigen. Für einen groben Überblick reichen etwa die Gesamtkosten aller Cloud Provider. Will man jedoch die einzelnen Kosten analysieren und ggfs.

<sup>16</sup>AWS IAM steht für Identity Access Management und beschreibt Amazons Dienst zur Identität- und Benutzerverwaltung. Der Dienst erlaubt es Nutzer anzulegen und ihnen Berechtigungen zu zuweisen.

<sup>17</sup>OIDC steht für OpenID Connect und basiert auf dem OAuth 2.0 Protokoll zur Authentifizierung. OAuth 2.0 erlaubt es den Clients die Identität des Nutzers zu überprüfen. OpenID ermöglicht es mittels eines Tokens sich bei weiteren Diensten anzumelden die ebenfalls OpenID unterstützen (OpenID Provider). Bekanntes Beispiel ist die Option „Mit Google anmelden“ oder auch „Mit Apple anmelden“. Die Tokens zur Authentifizierung (JWT Tokens) der Identität werden verschlüsselt im JSON Format versendet und ermöglichen einen standardisierten Weg zum Anmelden.

<sup>18</sup>Amazon CloudWatch Logs ist ein Überwachungs- und Management-Service, in dem Logs von sämtlichen AWS-Diensten gespeichert und ausgewertet werden können.

<sup>19</sup>Redis steht für Remote Dictionary Server und ist ein schneller Datenspeicher der als Cache geeignet ist. Alle Daten liegen bei Redis im Arbeitsspeicher und ermöglichen einen schnellen Zugriff.



optimieren sind mehr Daten notwendig. Da bisher nur bekannt ist welche Daten benötigt werden, jedoch nicht in welchem Detailgrad, ist eine Umsetzung mit GraphQL, also AWS Appsync von Vorteil. Auf Wunsch kann die Query jederzeit flexibel angepasst werden.

Außerdem wird einem der Aufwand zur Erstellung der Datenbank und des benötigten Resolvers übernommen, sofern DynamoDB genutzt wird. Das Erstellen aller Queries, Mutationen und Subscriptions nimmt einem der Dienst ebenfalls ab, was einen schnellen Einstieg zur Folge hat. Sobald ein Schema mit Datenstrukturen definiert wurde generiert AppSync automatisch alle möglichen Operationen. Diese Operationen können dann im Anschluss direkt im Frontend verwendet werden.

Aufgrund der überzeugenderen Integration und höheren Flexibilität wird die Anwendung mit AWS AppSync realisiert, da viele Einstiegshürden von AWS übernommen werden. Es ist davon auszugehen, dass es keine gravierenden Unterschiede bei den Kosten geben wird. Zum einen sind die Kosten in einem ähnlichem Verhältnis, zum anderen werden keine Millionen von Anfragen erwartet.

### **3.3 Datenbanksystem**

Es gibt zwei unterschiedliche Technologien für die Nutzung von Datenbanken, relationale und nicht relationale Datenbanken. Sie variieren in vielen Aspekten und haben andere Anwendungsfälle. Amazon bietet für beide Technologien eigene Dienste an, die im Folgenden genauer betrachtet und verglichen werden sollen.

#### **3.3.1 Relationale Datenbanken: AWS RDS**

Relationale Datenbanken speichern Daten in Tabellenform ab, wobei einzelne Tabellen in Relation zueinander stehen. Aus Benutzersicht besteht die Datenbank nur aus Tabellen, die physische Struktur bleibt ihm verborgen. Zur Vermeidung von unzulässigen Einträgen benötigt eine Tabelle einen Primärschlüssel, der ein eindeutiges Zeilenmerkmal ist. Um Beziehungen zu anderen Tabellen zu erzeugen, wird ein Fremdschlüssel benötigt, der eine Referenz auf den Primärschlüssel der anderen Tabelle darstellt. Damit Redundanzen innerhalb einer Datenbank vermieden werden, kann die Datenbank gemäß den Normalisierungsregeln bearbeitet werden. So wird gewährleistet, dass Daten nicht mehrfach existieren und bei Veränderung dieser Daten keine Anomalien<sup>20</sup> auftreten. Die Datenstrukturen sind voneinander abhängig. Eine Auswertung oder Veränderung der Daten ist in der Regel mit der Datenbanksprache SQL möglich. [Ale17, ]

Der zugehörige Dienst von Amazon heißt RDS (Relational Database Service) und unterstützt sechs Datenbank-Engines. Dazu gehören PostgreSQL, MySQL, MariaDB, Oracle Database, Microsoft SQL-Server und Amazons eigener Dienst Aurora. RDS bietet mehrere unterschiedliche Instanztypen an. Je höher die geforderte Kapazität, desto höher auch der Preis. Der Instanztyp lässt sich jederzeit anpassen. Während des Erstellens müssen Zugangsdaten für den Zugriff angegeben werden. Man erhält keinen direkten Zugang auf die zugrundeliegende Hardware, sondern einen User für den jeweils ausgewählten SQL Service. Im Falle von MySQL würde man etwa nach Erstellen einen Endpunkt erhalten, auf den man sich per

---

<sup>20</sup>Unter Anomalien versteht man ein Fehlverhalten innerhalb von relationalen Datenbanken. Tabellen mit Spalten gleicher Bedeutung haben einen unterschiedlichen Inhalt. Eine Normalisierung verhindert Anomalien.

```
mysql -h <Endpunkt> -u <User> -p <Passwort>
```

Befehl sich anmelden kann. SSH-Zugang oder Ähnliches ist bei RDS nicht vorgesehen. Das angegebene Beispiel ist typisch für einen Plattform as a Service Dienst.

Eine Hochverfügbarkeit ist möglich, indem man den Punkt Multi-Availability Zone auswählt und so eine Standby-Instanz in einer anderen Availability Zone bereitsteht. Hierdurch steigt jedoch auch der Preis. Software-Patches können von dem Dienst automatisch eingespielt werden. Anders als bei DynamoDB gibt es keine direkte Integration von RDS Datenbanken mit Serverless Anwendungen bei AWS. AWS stellt keine direkte Verknüpfung zwischen GraphQL API und RDS zur Verfügung. Eine Implementierung mittels REST bzw. API Gateway würde eine zusätzliche Lambda-Funktion voraussetzen, welche die Abfrage gegen eine nicht öffentliche RDS Datenbank ausführt. Wäre die Datenbank im Internet erreichbar müsste das Frontend Credentials besitzen um auf die Datenbank zugreifen zu können. RDS eignet sich vor allem bei klassischen SQL-basierten Datenbanksystemen. Gibt es Einschränkungen eine bestimmte Datenbank-Engine zu nutzen, bietet sich der Dienst ebenfalls an. Der Dienstleister Airbnb nutzt RDS zum Beispiel für automatisierte Replikationen und Leistungstests. Die günstigste RDS Instanz „db.t3.micro“ mit 1 Kern, 1 GB Arbeitsspeicher und der Open Source Engine MariaDB kostet in Frankfurt mit einer ausgewählten Hochverfügbarkeit 0,04 USD pro Stunde. [Ama20g, ]

### 3.3.2 Nicht relationale Datenbanken: AWS DynamoDB

AWS DynamoDB ist ein serverloser Dienst zur Erstellung und Verwaltung nicht relationaler Datenbanken. Bei dieser Art von NoSQL(not only SQL)-Datenbanken werden Daten als Key/Value Paare aufgebaut. Daneben gibt es noch weitere Modelle, etwa Graphdatenbanken oder dokumentenorientierte Datenbanken.

Bei den Key-Value-Datenbanken können neue Paare hinzugefügt werden ohne die gesamte Struktur abändern zu müssen. Es können verschiedene Daten ohne Konvertierung gemeinsam gespeichert werden. Die Keys müssen eindeutig sein und sind mit den Primärschlüssel der relationalen Datenbank vergleichbar. Um eine Skalierung und Hochverfügbarkeit zu ermöglichen, werden die Daten auf alle vorhandenen Systeme kopiert und verteilt. Daher ist es auch problemlos möglich weitere Server zu verwenden, um eine größere Menge an Daten zu verarbeiten. Dies wird als horizontales Skalieren bezeichnet. Relationale Datenbanken bieten meist nur eine vertikale Skalierung an, d.h. um die Performance zu erhöhen, muss in der Regel ein leistungsstärkerer Server genutzt werden. Zur Erhöhung des Lesedurchsatzes besteht jedoch die Option ein, oder mehrere, Read-Replica zu nutzen, die einen parallelen Lesezugriff ermöglichen.

NoSQL Datenbanken unterstützen in der Regel keine ACID-Eigenschaften<sup>21</sup> sondern nutzen das BASE-Modell<sup>22</sup>. Da bei SQL-basierten Datenbanken lesende Abfragen so lange warten bis schreibende Vorgänge beendet sind, bleibt die Konsistenz der Daten erhalten. NoSQL Datenbanken könnten im Zweifel unterschiedliche Daten zurückgeben, falls noch nicht alles ausgetauscht wurde. [Ale17, ]

<sup>21</sup>ACID steht für Atomicity, Consistency, Isolation, Durability und bedeutet im Kern, dass alle Transaktionen konsistent sind.

<sup>22</sup>Base steht für Basically Available, Soft State, Eventually Consistent und stellt die Verfügbarkeit von Daten an eine höhere Stelle als die Konsistenz.

Anders als AWS RDS gibt es bei DynamoDB gar keine Möglichkeiten mehr sich bei der Datenbank-Engine anzumelden. Der Dienst wird als Serverless angeboten. Es müssen keine Kapazitäten im Voraus berechnet werden, da der Dienst automatisch skaliert. Anwender haben die Option direkt Tabellen anzulegen. Zudem entfällt es auch sich um Software-Patches oder der Sicherstellung der Hochverfügbarkeit zu kümmern, da Tabellen immer global bereitgestellt werden. Im Gegensatz zu den meisten NoSQL Datenbanken unterstützt DynamoDB ACID-Transaktionen. Amazon bietet dafür eine eigene API an, die in der Applikation zusätzlich implementiert werden muss. Insgesamt ist es sehr schnell möglich eine Tabelle mit dem Dienst bereitzustellen. Zudem existiert so gut wie kein Wartungsaufwand, da Amazon die Verantwortung für die wichtigsten Aspekte übernimmt. [Ama20r, ]

Das Preiskonzept für DynamoDB ist sehr granular aufgebaut und bietet zwei unterschiedliche Kapazitätsmodi an. Der Modus „Bereitgestellt“ bietet sich an, wenn es bereits möglich ist Anforderungen an die Kapazität zu bestimmen und der Datenverkehr berechenbar ist. Der Preis wird pro Stunde und benötigte Kapazität errechnet. Auf der anderen Seite ist der Kapazitätsmodus „On-demand“ für unbekannte Workloads optimiert und skaliert automatisch mit den Anforderungen. Hier wird der Preis nicht pro Zeiteinheit berechnet, sondern auf Basis der benötigten Schreib- und Leseanforderungen. Je nachdem welchen genauen API-Aufruf man tätigt kann dieser eine halbe Einheit oder zwei Einheiten erfordern. Zur Option stehen „Strongly Consistent Reads“ und „Eventually Consistent Reads“<sup>23</sup> DynamoDB nutzt standardmäßig „Eventually Consistent Reads“, wobei ein Aufruf bis zu 8 KB eine Einheit erfordert. Eine Million dieser Schreibanforderungen kosten in Frankfurt 1,525 USD und eine Million Leseanforderungen 0,305 USD. Zudem entstehen Kosten für den Datenspeicher, die Sicherung, Datenübertragung und ein paar weitere speziellere Funktionen. Beim Datenspeicher sind die ersten 25 GB pro Monat kostenlos und kosten danach pro GB-Monat 0,306 USD. [Ama20f, ]

### 3.3.3 Entscheidung

Eine NoSQL-Datenbank, und damit AWS DynamoDB, sind für diese Projektumsetzung besser geeignet. Zum einen ist der von Amazon zur Verfügung gestellte Dienst besser mit den restlichen Komponenten integriert, zum anderen ist der NoSQL Ansatz auch durch die höhere Flexibilität passender. Eine RDS Datenbank müsste separat vom restlichen Workflow aufgesetzt und konfiguriert werden. Egal ob GraphQL API oder REST API, eine Umsetzung mit RDS wäre deutlich aufwendiger als mit DynamoDB. Bei der Kombination GraphQL und RDS müsste das GraphQL Schema samt aller Queries und Mutationen manuell erstellt werden, da kein automatisierter Dienst dafür existiert. Auf der anderen Seite müsste für das API Gateway zusätzlich einen Weg zur Kommunikation mit der RDS Datenbank bereitgestellt werden. Die Möglichkeit die RDS Datenbank im Internet erreichbar zu machen ist auf Grund des erhöhten Sicherheitsrisikos keine Option.

Der DynamoDB Dienst ist im Vergleich dazu erheblich einfacher aufzusetzen. Der Workflow der Anwendung benötigt keine ACID-Garantien oder komplexe Abfragen. Beziehungen sind ebenfalls nicht vorhanden, da es sich um Daten unterschiedlicher Cloud Provider handelt die keinen direkt Bezug zueinander haben. Mit RDS ist eine Auswahl eines Instanztypes notwendig und es würden ebenfalls pauschale Kosten für die Server anfallen. Da bereits die Entscheidung

<sup>23</sup>Bei einem Strongly Consistent Aufruf gibt DynamoDB den aktuellsten Datensatz zurück. Bei Eventually Consistent besteht die Möglichkeit, dass der Wert nicht der aktuellste ist. Nachteil von Strongly Consistent ist die höhere Latenz und ein höherer Verbrauch der Leseanforderungseinheiten.

für AWS AppSync bei der API fiel, ist es zudem eine erhebliche Erleichterung eine DynamoDB Tabelle mit der API zu verknüpfen. Eine Umsetzung mit DynamoDB ermöglicht es den Konfigurationsaufwand und die Kosten minimal zu halten.

### 3.4 Authentifizierung

Da die Anwendung bei AWS gehostet wird, wird sie auch über das Internet erreichbar sein. Um den Zugriff nur für ausgewählte Mitarbeiter der Mediengruppe RTL einschränken zu können ist eine Authentifizierung essentiell. Ohne Registrierung und Anmeldung darf es nicht möglich sein Daten einzusehen. Im besten Fall ist es sogar möglich die Authentifizierung mit bereits vorhandenen Firmenidentitäten zu verknüpfen, sodass keine eigenen Benutzer registriert werden müssen.

Alle Mitarbeiter der Mediengruppe RTL werden in einer eigenen Active Directory Domäne auf On Premises Servern verwaltet. Dieses Verzeichnis wird mit dem Cloud-Dienst Azure Active Directory synchronisiert. Dadurch können sich Mitarbeiter von jedem Ort aus authentifizieren und von Single Sign-On (SSO) profitieren. Mit Single Sign-On benötigt man nur eine einmalige Authentifizierung, um auf sämtliche unterstützte Dienste mit derselben Identität zugreifen zu können. Der Anwender muss sich nicht mehr überall einzeln anmelden. Innerhalb des SSO-Systems wird die Identität zusammengeführt und übernimmt die Aufgabe die Identität des Anwenders zu bestätigen. Größter Vorteil ist eine konsolidierte Möglichkeit zur Anmeldung. Zudem muss der Anwender, falls er das Unternehmen verlässt, nur noch an einer Stelle entfernt werden und der Zugang zu allen Diensten wird automatisch entzogen. Zusätzlich wird unter bestimmten Bedingungen eine MFA-Verifizierung<sup>24</sup> erfordert.

Um den Registrierungs- und Anmeldeprozess so einfach wie möglich zu gestalten, bietet AWS den Dienst Cognito an.

#### 3.4.1 AWS Cognito

Amazon Cognito ist ein voll integrierter Dienst zur Benutzerverwaltung und Autorisierung. Cognito übernimmt dabei die Registrierung und Verwaltung neuer Benutzer sowie die Steuerung von Zugriffen. Mithilfe des AWS Amplify-Frameworks kann die Benutzeroberfläche zum Registrieren bzw. An- und Abmelden leicht in die eigene Anwendung implementiert werden. Das Framework stellt SDKs für alle gängigen mobilen Plattformen sowie JavaScript inklusive JavaScript-Frameworks wie React, Angular und Vue bereit. Ein Vergleich der Javascript-Frameworks befindet sich im Abschnitt 3.6 *Frontend Framework*. Es soll möglich sein, mit wenig Code die eigene Anwendung mit Cognito zu verknüpfen, und bei Bedarf die Oberfläche anzupassen. Der Dienst ist vollständig Serverless aufgebaut und bietet die Option „Hundert von Millionen Benutzern“ zu verwalten, „ohne dass Server-Infrastruktur aufgestellt werden muss“ [Ama20d, ]. [Ama20q, ]

Cognito besteht aus den Komponenten Benutzerpools und Identitäten-Pools. Ein Benutzerpool ist ein Verzeichnis durch den Benutzer angelegt und verwaltet werden können. Hier können Attribute konfiguriert werden, z.B. mit welcher Information sich Benutzer anmelden können. Zur Option stehen ein Username, die E-Mail Adresse oder eine Telefonnummer. Außerdem ist es

---

<sup>24</sup>MFA steht für Multi-Factor Authentication und steigert die Sicherheit für Nutzer indem neben dem Passwort ein weiterer Faktor zum Anmelden benötigt wird. Häufig wird als Zweitfaktor eine SMS mit einem Code oder ein Einmalkennwort welches mittels TOTP(Time-based One-time Password) Verfahren generiert wird.

möglich weitere Attribute zu setzen, wie etwa das Geschlecht, der Wohnort, das Geburtsdatum und noch viele weitere. Eine MFA-Verifizierung ist ebenfalls im Dienst implementiert. Über den Benutzerpool können ebenfalls App-Clients erstellt werden. Ein App-Client wird genutzt um nicht authentifizierte Operationen durchführen zu können. Dazu gehört das Registrieren, Anmelden und die Passwortwiederherstellung. Es ist möglich mehrere App-Clients zu nutzen, beispielsweise jeweils einen für einen Android, iOS und Webclient.

Der Identitäten-Pool erlaubt es Nutzern Zugriff auf andere Dienste zu erteilen. Es werden eindeutige Identitäten erstellt und diese mit anderen Diensten verbunden. Dabei unterstützt Cognito soziale Identitätsanbieter wie Google, Facebook oder Apple aber auch Unternehmens-Identitätsanbieter wie Microsoft Active Directory. Hierfür werden gängige Standards zur Verwaltung von Identitäten wie OpenID, OAuth 2.0 und SAML 2.0<sup>25</sup> genutzt. [Ama20v, ]

Ähnlich zu den meisten anderen Serverless Diensten von AWS, fallen auch bei Cognito keine pauschalen Kosten an. Nur die tatsächliche Nutzung wird in Rechnung gestellt. Amazon gewährt für die ersten 50.000 monatlich aktiven Benutzer ein kostenloses Kontingent. Dieses gilt jedoch nur für Benutzer die direkt über den Cognito User Pool oder soziale Identitätsanbieter registriert sind. Bei Anmeldungen mit einem Unternehmens-Identitätsanbieter liegt das kostenlose Kontingent bei 50 Nutzern. Darüber hinaus kostet jeder weitere Nutzer 0.0055 USD pro Monat. Ab einer Anzahl von 100.000 wird der Preis niedriger gestaffelt. Außerdem können Kosten für die SMS-Nachrichten bei der MFA Authentifizierung anfallen. [Ama20e, ]

### 3.4.2 Alternative und Entscheidung

Da Amazon Cognito alle gängigen Standards zur Authentifizierung unterstützt, wird auch kein alternativer Dienst angeboten. Alle Möglichkeiten werden bedient und mithilfe des Amplify-Frameworks ist eine Implementierung ebenfalls leicht realisierbar. Theoretisch ist es nicht einmal notwendig eine eigene Oberfläche zur Anmeldung zu schreiben.

Die einzige alternative Möglichkeit besteht darin, die gesamte Authentifizierung selbstständig zu schreiben. JavaScript-Frameworks wie React oder Vue können den Prozess etwas erleichtern, nichtsdestotrotz ist der Aufwand deutlich höher als die Verwendung von Cognito. Der Vorteil einer eigenen Implementierung ist eine höhere Flexibilität und ein größerer Lernprozess, da sich mit allen einzelnen Schritten intensiver befasst werden muss. Wird gewünscht zusätzlich noch MFA oder eine sichere Passwortwiederherstellung einzubauen, muss noch mehr Zeit einkalkuliert werden.

Angesichts der enormen Einfachheit und ausreichenden Flexibilität ist die Verwendung von Cognito die sinnvollere Wahl. Auf Wunsch kann Cognito mit einer Vielzahl von Anbietern und Diensten verknüpft werden, sowie direkt in Amplify integriert werden. Das Amplify-Framework bietet für die gängigen JavaScript-Frameworks vorgefertigte Module an, um noch weniger Konfigurationsaufwand betreiben zu müssen. Dank Cognito dürfte die Implementierung einer Authentifizierung in einer Serverless Anwendung keinen allzu großen Aufwand mehr benötigen.

---

<sup>25</sup>SAML steht für Security Assertion Markup Language und ist ein XML-Framework zur Authentifizierung und Autorisierung. Anders als bei OpenID Connect, benötigt SAML eine konfigurierte Vertrauensbeziehung. Häufig kommt SAML bei Single Sign-On Verfahren zum Einsatz.

### 3.5 Backend Logik

Um überhaupt Daten erhalten zu können, ist ein Backend-Prozess zum Verarbeiten von Anforderungen erforderlich. Für die gewünschte Implementierung ist es notwendig eine Liste aller AWS Accounts abzufragen und anschließend diese Daten in die DynamoDB Tabelle abzuspeichern. Falls notwendig soll auch die Datenstruktur angepasst werden. Dieser Prozess muss regelmäßig ausgeführt werden, da ständig neue AWS Accounts der Organisation hinzugefügt werden. Bestenfalls wird der Prozess direkt nach der Erstellung eines neuen Accounts initiiert.

Zur Bewältigung dieser Aufgabe sind die meisten AWS Dienste potenziell einsetzbar, jedoch nicht wirklich geeignet. Eine EC2 Instanz mit Linux Betriebssystem und Programmcode könnte theoretisch die Datenverarbeitung durchführen, es entspricht jedoch nicht der Serverless Architektur und benötigt signifikant mehr Aufwand in der Implementierung und Wartung. Auch eine Docker-basierte Lösung benötigt merklich mehr Arbeit, da die Laufzeitumgebung und Datenverarbeitung weiterhin im Verantwortungsbereich des Nutzers liegt. Außerdem müsste bei beiden zuvor genannten Optionen mindestens 1 Server bzw. Container dauerhaft laufen um Anfragen annehmen zu können. Beide Varianten sind keine valide Option für eine Serverlose Umsetzung.

Möchte man Code mit einem Minimum an Administrationsaufwand in der Cloud ausführen eignet sich der Dienst AWS Lambda, der im folgenden Abschnitt ausführlicher erläutert wird.

#### 3.5.1 AWS Lambda

Im November 2014 wurde der Serverlose Datenverarbeitungsservice AWS Lambda veröffentlicht, mit dem Anwender ihren Code jederzeit ausführen können. Amazon kümmert sich um die „gesamte Administration der Datenverarbeitungsressourcen, einschließlich der Server- und Betriebssystemwartung, Kapazitätsbereitstellung, automatischen Skalierung sowie der Code-Überwachung und -Protokollierung.“ [Ama20y, ] Da auch die Laufzeitumgebung von AWS verantwortet wird, ist die Auswahl der Programmiersprachen vorgegeben. Zur Auswahl stehen NodeJS, Python, Ruby, Java, Go, C# und Powershell. Mit etwas mehr Aufwand ist es zudem auch möglich eine Benutzerdefinierte Laufzeit zu implementieren, sodass weitere Sprachen möglich sind. NodeJS und Python Code können direkt in der Webkonsole geschrieben und getestet werden. Weiterhin besteht die Möglichkeit den Code mithilfe einer ZIP-Datei oder über die lokale Entwicklungsumgebungen hochzuladen. [Ama20l, ]

Wie bereits im Abschnitt 2.3.5 *FaaS: Function as a Service* erwähnt arbeitet Lambda eventbasiert und kann sowohl von anderen AWS-Diensten, als auch durch einen manuellen Aufruf ausgelöst werden. Insgesamt gibt es unterschiedliche Varianten eine Lambda-Funktion zu starten. Zum einen kann Lambda direkt Daten aus Services lesen, welche einen Datenstrom oder eine Warteschlange erzeugen. So ist es möglich jedes Mal eine Lambda-Funktion auszulösen, sobald eine DynamoDB-Tabelle aktualisiert wird. Hierbei liest Lambda die Datensätze selbst und benötigt dementsprechend auch Berechtigungen auf die DynamoDB-Tabelle. [Ama20t, ] Die zweite Variante ermöglicht es bestimmten AWS Diensten eine Lambda-Funktion aufzurufen. Dienste wie Cognito oder API Gateway können bei Ereignissen die Lambda-Funktion auslösen und warten bis sie eine Antwort erhalten. Hier findet ein synchroner Aufruf statt. Anders bei den Diensten Amazon S3 oder Amazon SES<sup>26</sup>. Da es sich um asynchrone Aufrufe handelt warten die Dienste nicht auf eine erfolgreiche Antwort. Die Dienste erfahren nur, dass Lambda das

---

<sup>26</sup>SES steht für Simple E-Mail Service und ist ein Dienst zum Versenden und Empfangen von Mails.

Ereignis in eine Warteschlange übergeben hat. Falls bei der Verarbeitung Fehler auftreten, kann die Funktion erneut ausgeführt werden. Alle Informationen zur Lambda-Funktion, auch potenzielle Fehler im Code, werden automatisch protokolliert und in Amazon CloudWatch Logs gespeichert. [Ama20u, ]

Sobald eine Lambda-Funktion ausgelöst wird, startet die Codeausführung am Handler. Der Handler ist eine spezielle Methode innerhalb des Lambda-Codes der Ereignisse verarbeitet. Je nach Programmiersprache existieren unterschiedliche Voraussetzungen für den Handler. Innerhalb des Handlers können weitere Funktionen und Methoden aufgerufen werden die zur Bearbeitung notwendig sind. Wenn der Handler mit der Ausführung des Codes fertig ist, steht er für weitere Events zur Verfügung. Werden mehrere Events ausgelöst starten auch gleichzeitig mehrere Kopien der Funktion. Nach dem Fertigstellen skaliert Lambda automatisch bis keine Ressourcen im Leerlauf übrig bleiben. Um eine fehlerfreie Ausführung von beliebig vielen Funktionen sicherzustellen, ist die Zustandslosigkeit somit essentiell. Dies bedingt ebenfalls, dass der Speicherplatz nicht persistent ist und auf 500 MB eingeschränkt ist. [Ama20s, Seitenzahl 5-7]

Während der Erstellung einer Funktion wird zusätzlich eine Lambda-Ausführungsrolle mit Berechtigungen definiert. Diese Ausführungsrolle gewährt der Funktion Zugriff auf Dienste und Ressourcen die angegeben worden sind. Die Berechtigungen können dabei jederzeit ergänzt oder entfernt werden. Dank der Ausführungsrolle ist es möglich die Zugriffskontrolle nach dem Least-Privilege-Prinzip<sup>27</sup> zu konzipieren.

Damit ein Fehler im Code eine Lambda-Funktion nicht theoretisch unendlich lang laufen lässt, wird die Funktion nach maximal 15 Minuten terminiert. Außerdem ist es nicht möglich einer Funktion beliebig viele Ressourcen zuzuteilen. Amazon erlaubt aktuell maximal 3 GB an Arbeitsspeicher und minimal 128 MB. Die Größe des Arbeitsspeichers muss selbst ausgewählt werden.

Bezahlt wird bei Lambda die Dauer der Ausführung sowie die Anzahl der Anforderungen. Die Dauer wird in 100-Millisekunden-Schritten berechnet und der Preis ist je nach Größe des Arbeitsspeichers abhängig. 512 MB Arbeitsspeicher kosten etwa 0,0000008333 USD pro 100ms. Anforderungen, zu denen jeder Aufruf oder jede Ereignisbenachrichtigung zählen, kosten 0,20 USD pro 1 Mio. Anforderungen. [Ama20m, ]

### **3.5.2 Entscheidung**

AWS Lambda eignet sich optimal zur Realisierung der Webanwendung. Direkt nach Erstellung der Lambda-Funktion und setzen der passenden Berechtigungen kann am Code geschrieben werden. Dementsprechend muss keine Zeit für Themen wie Netzwerk, Virtualisierung oder die Laufzeitumgebung vergeudet werden.

Für die Webanwendung wird eine Lambda-Funktion benötigt, die über den Dienst AWS Organizations alle zurzeit verfügbaren Accounts abrufen kann. AWS Organizations ist ein zentraler Dienst zur Steuerung und Verwaltung von AWS Accounts. Innerhalb eines Master-Accounts werden neue Accounts nach Bedarf erstellt und konfiguriert. Nur dieser Master-Account besitzt Informationen über alle verfügbaren Accounts in der Organisation. Die Lambda-Funktion, welche in einem anderen Account bereitgestellt wird, benötigt Zugriff auf den Master-Account

---

<sup>27</sup>Durch das Least-Privilege-Prinzip wird sichergestellt, dass nur die minimal erforderlichen Berechtigungen zugeteilt werden.

und muss über eine API die Liste der Accounts abrufen können. Anschließend müssen die gesammelten Daten in die DynamoDB-Tabelle gespeichert werden. Alle Komponenten dieses Projektes werden in einem separaten Account erzeugt, der durch die Organisation erstellt wurde.

Da das Frontend mit einem JavaScript-Framework erzeugt wird, bietet es sich an für Lambda NodeJS als Programmiersprache zu verwenden. So wird vermieden zwei unterschiedliche Syntaxstrukturen und Funktionsweisen lernen zu müssen. AWS bietet für NodeJS eine AWS Organizations API an, um exakt diesen Aufruf tätigen zu können. Der Zugriff auf AWS Organizations sowie die Accountstruktur der Mediengruppe RTL werden im Abschnitt (4.2 AWS Accountstruktur ) genauer erläutert. [Ama20n, ]

### 3.6 Frontend Framework

Die bisher ausgewählten Dienste Cognito und AppSync unterstützten die gängigsten JavaScript-Frameworks. Auch im Zusammenspiel mit Amplify gibt es keine Einschränkungen. Zur Auswahl für das Web-Frontend stehen React, React Native, Vue oder Angular. React Native eignet sich nicht zur Umsetzung, da es hauptsächlich für die native Entwicklung von iOS und Android Apps genutzt wird. Mit allen drei restlichen Frameworks lässt sich das Frontend mit einem ähnlichen Aufwand aufbauen. Die Entscheidung erfolgt anhand von Aspekten wie Syntax, Popularität und Performance.

Angular wurde 2010 von Google entwickelt und ist das älteste der drei JavaScript-Frameworks. Für die Syntax verwendet Angular TypeScript<sup>28</sup> in Kombination mit HTML. Angular gehört mit React zu den verbreitetsten Frameworks und bietet viele themenbezogene Beiträge auf Plattformen wie Stack Overflow<sup>29</sup> und weiteren. Aufgrund der ungleichen Syntax zu JavaScript und dem daraus resultierenden mutmaßlich schwereren Einstieg scheint Angular nicht die beste Lösung zu sein. Dementsprechend ist die Entscheidung nicht auf Angular gefallen.

Vue ist mit Veröffentlichung im Jahr 2014 das jüngste Framework und wird von keinem großen Unternehmen, sondern durch die Community weiterentwickelt. In den letzten Jahren ist Vue zunehmend populärer geworden und mittlerweile existieren auch vermehrt Anleitungen zum Erstellen von Webanwendung. Besonders in China ist Vue sehr verbreitet. [Max20, Abschnitt: Comparing Adoption & Popularity] Code wird mithilfe von JavaScript und HTML geschrieben und dabei strikt voneinander getrennt. Der HTML Code befindet sich im `<template>` Bereich und JavaScript Code ist im `<script>` Bereich erlaubt. Hinsichtlich der geringeren Verbreitung im englischsprachigen Raum und dem Aspekt, dass weniger auf AWS-Dienste spezialisierte Anleitungen auffindbar sind, wird Vue nicht für die Realisierung verwendet.

React wurde 2013 von Facebook veröffentlicht und wird zusammen mit der Community weiterentwickelt. Anders als bei Vue, wird in React kein Code getrennt. React arbeitet, wie die meisten JavaScript-Frameworks, komponentenbasiert und trennt Einheiten auf falls möglich. Zum Rendern von HTML Elementen nutzt React die JavaScript-Erweiterung JSX, welche JavaScript und HTML Syntax kombiniert. Der JSX Code wird vor dem Start der Anwendung in

---

<sup>28</sup>TypeScript ist eine von Microsoft entwickelte Erweiterung oder auch Überkategorie von JavaScript. Mit TypeScript generierter Code lässt sich in JavaScript kompilieren. Im Gegensatz zu JavaScript muss bei TypeScript jede Variable einen Typ zugewiesen bekommen.

<sup>29</sup>Stack Overflow ist eine Internetseite auf der Benutzer Fragen und Antworten rund um das Thema Programmierung und Softwareentwicklung stellen können.



normalen Javascript-Code übersetzt. Innerhalb von geschweiften Klammern wird alles als Javascript Code behandelt. Es ist also auch möglich alle Elemente direkt mit React und Javascript zu erzeugen, jedoch versucht JSX eine komfortablere und leichtere Lösung anzubieten. Anstatt Methoden wie `React.createElement()` zu nutzen reicht es aus die HTML Syntax zu kennen. Mit JSX lässt sich ein `<div>` Element wie folgt erzeugen:

```
const divelement = <div>Ich bin ein DIV-Element</div>;

ReactDOM.render(divelement, document.getElementById('root'));
```

Ohne JSX wäre folgender Code notwendig:

```
const divelement = React.createElement('div', {}, 'TEXT');

ReactDOM.render(divelement, document.getElementById('root'));
```

[W3S20, ]

Da im Studium bereits Erfahrung mit HTML und JavaScript gemacht wurde, ist der Einstieg in JSX und React der komfortabelste. Auch scheint es die meisten Forenbeiträge und Anleitungen im Internet zu React im Zusammenspiel mit AWS zu geben, sodass im Problemfall am meisten Lösungsvorschläge zu finden sind. Aus diesen Gründen wird die Implementierung in React erfolgen.

### 3.7 AWS Amplify

Amplify ist ein zentraler Dienst rund um das Thema Serverless Webanwendungen, die Veröffentlichung war im November 2017. Amazon bezeichnet Amplify selbst als „ein Set aus Tools und Services, das Entwicklern erlaubt, skalierbare vollständige mobile und Front-End-Anwendungen zu entwickeln, powered by AWS.“ [Ama20h, ]

Amplify kann als übergeordnete Einheit angesehen werden, welche alle zuvor erläuterten Dienste an einem Ort zusammenfasst und verwaltet. Dank Amplify soll es nicht mehr notwendig sein, Dienste einzeln zu erstellen und manuell miteinander zu verknüpfen. Zusätzlich bietet Amplify die Option für ein statisches Web-Hosting an. Es ist nur notwendig den Code für die Anwendung entweder über die Web-Konsole oder die CLI hochzuladen. Den gesamten Workflow für die Bereitstellung kann im Anschluss von Amplify übernommen werden. Dafür wird eine eigene `amplifyapp.com`-Subdomäne bereitgestellt. Diese lässt sich auch auf eine Wunsch-Domain anpassen. Außerdem besteht die Möglichkeit Amplify mit GitHub zu verknüpfen und somit einen automatisierten CI/CD Workflow aufzubauen. Sobald ein Code in einem bestimmten Repository aktualisiert wurde, kann Amplify die Anwendung automatisch aktualisieren und veröffentlichen. [Ama20p, ]

Amplify lässt sich sowohl über die Amazon Web Console, als auch über eine eigene Amplify CLI konfigurieren. Alle verfügbaren Dienste sind in der Amplify Bibliothek vereint und können in das Frontend eingebaut werden. Das Amplify Framework lässt sich in drei Komponenten unterteilen: Bibliotheken, UI-Komponenten und eine CLI Toolchain. Jede Komponente kann einzeln oder gemeinsam genutzt werden.

Bibliotheken sind Open Source und nutzen bereits vorhandene AWS Dienste, um eine native Webanwendung für die Cloud bereitzustellen. Hierzu zählen unter anderem Module für zuvor

erwähnte Dienste wie Cognito, AppSync, API Gateway, oder S3. Darüber hinaus existieren auch Module für Chatbots, Push Benachrichtungen, AR/VR, künstliche Intelligenz sowie weitere.

Auch die UI-Komponenten sind Open Source und sollen eine einfache Verzahnung zwischen dem User Interface und den Workflows der Dienste anbieten. So gibt es zum Beispiel fertige Interface Elemente für das Hochladen von Bildern und Dateien in S3.

Die Amplify CLI Toolchain dient dazu, ein Serverless Backend zu erstellen und verwalten. Dazu gehört die Erstellung aller zuvor genannten Dienste und Funktionen. Zudem ist es möglich ein statisches Hosting einzurichten. Dabei werden die benötigten Daten über ein S3 Bucket bereitgestellt.

Amplify erleichtert es einem Projekte als Full Stack Developer<sup>30</sup> zu realisieren. Aus demselben Grund ist die Erstellung eines Prototypen ebenfalls schneller realisierbar. Neben dem statischen Hosting bietet Amplify vor allem den großen Mehrwert alle benötigten Dienste zentral verwalten zu können. Dies führt zu einer besseren Übersicht und einem einfacheren Einstieg in die Realisierung. [Ama20i, ]

Nachdem nun alle Dienste vorgestellt wurden und das Design entschieden wurde, wird im kommenden Kapitel die Implementierung durchgeführt. Auf Basis der bisher gesammelten Erkenntnisse kann ein Architekturbild entwickelt werden, um die Zugehörigkeit aller Komponenten zu verdeutlichen. Im Anschluss folgt die Implementierung inklusive Codebeispielen.

---

<sup>30</sup>Als Full Stack Developer wird jemand bezeichnet, der sowohl Client als auch Server erstellen und betreuen kann. Er ist allein Verantwortlich für das Frontend, Backend und eventuell damit verbundene Datenbanken.

## 4 Implementierung aller Komponenten

### 4.1 Architekturdiagramm

Im dritten Kapitel wurden alle benötigten Dienste ausgewählt und ihre Notwendigkeit erläutert. Nun muss aus allen Komponenten eine zusammenhängende Architektur aufgebaut werden. Das folgende Diagramm soll die Verbindung zwischen allen Diensten aufzeigen:

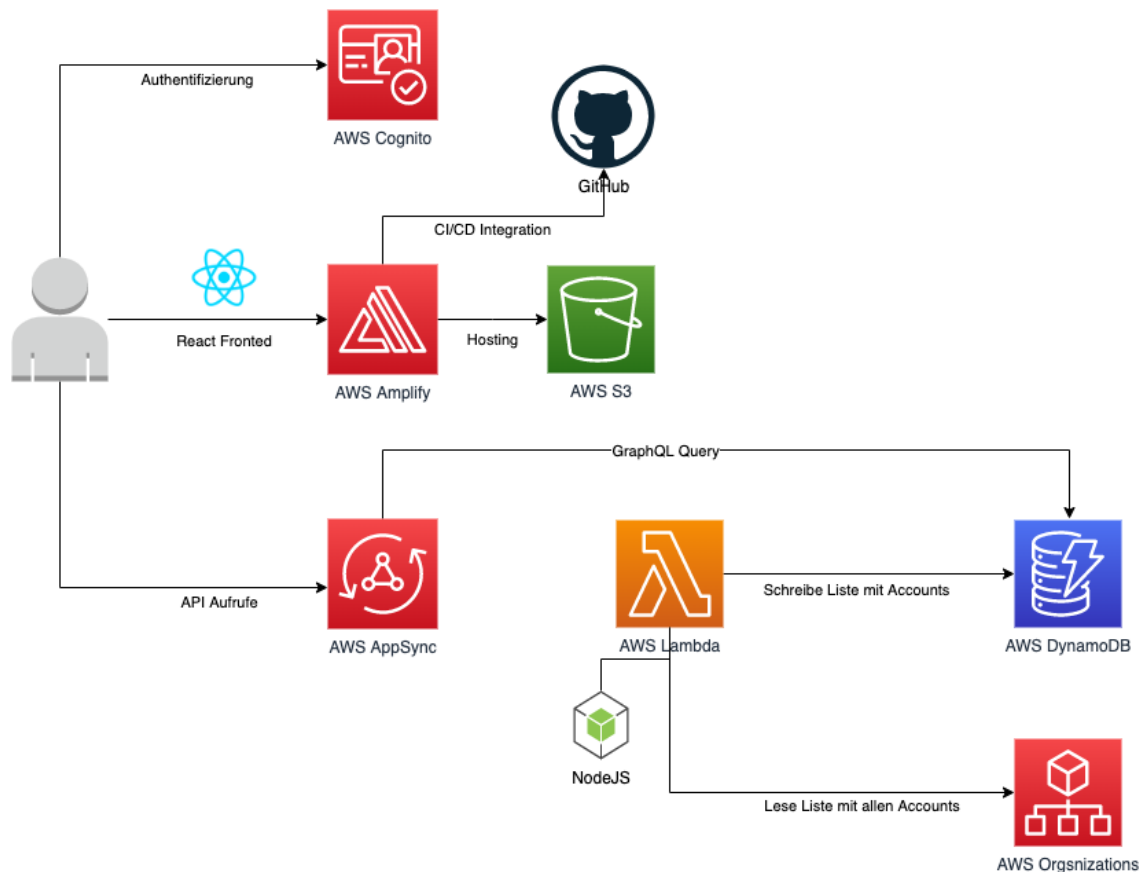


Abbildung 3: Übersicht der Architektur

Das Diagramm zeigt die Kommunikationswege zwischen allen Diensten. Sobald Anwender die Webanwendung aufrufen, kommunizieren sie mit den Diensten Cognito, Amplify Hosting sowie AppSync. Das Frontend greift mithilfe von AppSync wiederum auf die DynamoDB-Tabelle zu, um die benötigten Daten abzufragen. Die Lambda-Funktion sorgt im Hintergrund dafür, dass die Daten in der Tabelle stets aktuell sind. Dafür wird ein Zugriff auf einen anderen AWS Account benötigt. GitHub wird für eine CI/CD Pipeline genutzt, und kann über Amplify konfiguriert werden.

Die Umsetzung der Anwendung erfolgt in fünf einzelnen Schritten. Erzeugt werden alle Dienste über die Amplify CLI. Zuerst müssen alle Voraussetzungen erfüllt werden. Zu den Voraussetzungen zählt das Installieren aller benötigten Pakete und der Auswahl eines AWS Accounts. Außerdem wird ein neues React-Projekt erstellt und Amplify initialisiert. Im zweiten Schritt wird die GraphQL API über Amplify bereitgestellt. Wie bereits in 3.2.2 *GraphQL API: AWS*

*Appsync* erwähnt kümmert sich AppSync gleichzeitig um den DynamoDB-Resolver und übernimmt so die Aufgabe eine DynamoDB-Tabelle anzulegen. Während des dritten Schritts wird die Lambda-Funktion erstellt, welche die Backend-Logik abbildet. Ein Lambda-Resolver wird im Rahmen der Bachelorarbeit nicht benötigt, da die Lambda-Funktion in einem regelmäßigen Zyklus ausgeführt wird. Der vierte Schritt hat das Ziel eine Authentifizierung mit Cognito bereitzustellen. Es muss sichergestellt werden, dass die Erstellung neuer Benutzeraccounts eingeschränkt wird.

Zum Testen wird eine lokale Umgebung verwendet, sodass der Code erst nach Fertigstellung hochgeladen wird, und die Anwendung zuvor nicht aufgerufen werden kann. Das erlaubt ein schnelleres Testen sowie die Sicherheit, dass während der Entwicklung keine Daten öffentlich zugänglich sind. Im letzten Schritt kann die lokale Anwendung, sofern sie ordnungsgemäß funktioniert, bei AWS gehostet werden, sodass sie über das Internet zugänglich ist.

In den folgenden Abschnitten werden alle einzelnen Schritte im Detail erläutert sowie auf potenzielle Probleme bei der Umsetzung hingewiesen.

## 4.2 AWS Accountstruktur

Zuerst ist es wichtig die Organisationsstruktur der AWS Accounts innerhalb der Mediengruppe RTL zu verstehen. Zum einem ist es notwendig, da entschieden werden muss in welchem AWS Account die Webanwendung umgesetzt wird. Zum anderen benötigt die Lambda-Funktion ebenfalls Zugriff auf den Dienst AWS Organizations.

Wie bereits in der Einleitung erwähnt besitzt die Mediengruppe RTL über 150 AWS Accounts. Dafür gibt es mehrere Gründe. Eine potenzielle Sicherheitslücke eines Accounts hat keinen Einfluss auf einen anderen Account, da alle Ressourcen voneinander isoliert sind. Zusätzlich ist eine Kostenzuweisung deutlich einfacher, da pro Account eine eigene Kostenstelle angegeben werden kann.

Pro Funktion bzw. Projekt wird jeweils ein Account erstellt. Aus Sicherheitsgründen wird zudem immer eine separate Umgebung für Test- und Produktivzwecke erzeugt. Dadurch ist es möglich den Zugriff für bestimmte Umgebungen einzuschränken. Eine Produktivumgebung sollte so wenig Zugriff wie möglich erlauben. Beispiele für solche Accounts sind etwa `Ntv-Streaming-Dev` und `Ntv-Streaming-Prod`. Jeder der Accounts ist grundsätzlich unabhängig und weißt nur über die gemeinsame Organization eine Verbindung auf. Zusätzlich gibt es für N-TV noch weitere Accounts für andere Projektbereiche. Die Accounts `Ntv-Web-Dev` und `Ntv-Web-Prod` werden von anderen Personen betreut werden. Daneben ist es möglich, dass noch weitere Accounts oder Entwicklungsumgebungen benötigt werden, etwa eine `PreProd` Umgebung.

Wie bereits im Abschnitt 3.5.2 *Entscheidung* erwähnt, werden alle Accounts zentral über den Dienst AWS Organizations erstellt. Es wird Zugriff auf den Master-Account benötigt um Informationen zu allen Accounts zu erhalten. Details zur Umsetzung befinden sich im Abschnitt 4.5.1 *Accountübergreifender Zugriff*.

Für die Abteilung Datacenter and Clouds wurden speziell Accounts zum Testen erstellt. Der Account `Cbc-Clouds-Sandbox` eignet sich für eine erste Testumgebung daher optimal. Wird die Anwendung in Zukunft produktiv genutzt, kann sie ohne viel Aufwand in einen anderen Account migriert werden.

## 4.3 Amplify Voraussetzungen

### 4.3.1 Einrichtung Amplify CLI

Bevor mit der Amplify CLI gearbeitet werden kann, muss sichergestellt sein, dass alle benötigten Pakete installiert sind und ein Zugang zu dem AWS Account Cbc-Clouds-Sandbox existiert. Für die Verwendung von Amplify müssen die Pakete `Node.js` (Version > 10.x), `npm`<sup>31</sup> (Version > 5.x) und `git` (Version > 2.14.1) installiert werden. Mithilfe von `npm` kann im Anschluss mit folgendem Befehl `npm install -g @aws-amplify/cli` die Amplify-CLI installiert werden.

Um den Zugang auf den AWS Account Cbc-Clouds-Sandbox zu realisieren, muss ein AWS IAM-User mit Zugangsdaten angelegt werden. Diesen Prozess kann Amplify übernehmen. Die Konfiguration wird mit dem Befehl `amplify configure` gestartet. Es wird nach der gewünschten Region sowie Benutzernamen gefragt. Anschließend öffnet sich ein neues Browserfenster und der Vorgang kann fortgesetzt werden. In Browser wird das Erstellen des Benutzers mit passenden Berechtigungen bestätigt. Die im Browser angezeigten IAM Zugangs-schlüssel werden im Terminal eingegeben und der Vorgang ist abgeschlossen. [Amp20c, ]

Hinweis: Der interne Name des Projektes lautet Amplify-Kumo. Kumo ist die japanische Bezeichnung für „Cloud“.

```
# amplify configure
Follow these steps to set up access to your AWS account:

Sign in to your AWS administrator account:
https://console.aws.amazon.com/
Press Enter to continue

Specify the AWS Region
? region: eu-central-1
Specify the username of the new IAM user:
? user name: amplify-kumo
Complete the user creation using the AWS console
https://console.aws.amazon.com/iam/[...]

Press Enter to continue

Enter the access key of the newly created user:
? accessKeyId: *****
? secretAccessKey: *****

Successfully set up the new user.
```

<sup>31</sup>Npm steht für Node Package Manager und ist ein Paketmanager für NodeJS. Mithilfe von npm können weitere Pakete installiert werden.

### 4.3.2 Einrichtung React Projekt

Eine weitere Voraussetzungen für Amplify ist die Erstellung eines neuen React-Projektes. Dafür bieten die Entwickler von React den Befehl `npx create-react-app` an, welcher eine fertige Entwicklungsumgebung aufsetzt. Dabei werden Aspekte wie die Ordnerstruktur, benötigte JavaScript-Pakete oder auch die Konfiguration eines Webserver übernommen. [Fac20b, ] Nachdem das Projekt erstellt wurde, kann der Webserver unter der Adresse `http://localhost:3000` gestartet werden. Eine Testseite von React wird angezeigt.

```
npx create-react-app amplify-kumo
cd amplify-kumo
npm start
```

### 4.3.3 Konfiguration Amplify

Nachdem das React-Projekt erfolgreich erstellt wurde, kann Amplify konfiguriert werden. Anschließend ist es möglich alle gewünschten Dienste hinzuzufügen.

Zur Konfiguration muss `amplify init` ausgeführt werden. Als erstes werden allgemeine Informationen benötigt, wie der Projektname oder das gewünschte Javascript-Framework. Die darauffolgenden Pfade und Befehle sind nicht geändert worden. Des weiteren wird der, in 4.3.1 *Einrichtung Amplify CLI* erstellte, IAM-User angegeben. Nach kurzer Zeit ist das Amplify Projekt bereits in der Cloud erstellt worden und verfügbar.

Wie im Code ersichtlich, benutzt Amplify den Dienst AWS Cloudformation, sodass die gesamte Konfiguration in einem Template festgehalten wird. Mit AWS CloudFormation ist möglich Templates zur Modellierung und Bereitstellung von AWS-Ressourcen zu erstellen. Die Templates unterstützen JSON und YAML. Amplify übersetzt alle Befehle in ein Cloudformation Template und startet es. Da die gesamte Konfiguration der Cloudformation-Templates von Amplify übernommen wird, spielt es im Rahmen der Bachelorarbeit eine untergeordnete Rolle.

Außerdem wird bei der Initialisierung ein S3 Bucket sowie IAM-Rollen erstellt. Das Bucket dient als Speicher für alle Konfigurationen und Templates. Die IAM-Rollen werden für die Authentifizierung von Amplify benötigt. Auch für die Lambda-Funktion sind IAM-Rollen von großer Bedeutung. Im Abschnitt 4.5.1 *Accountübergreifender Zugriff* wird das Thema IAM-Rollen im Detail erklärt.

Es wurden alle Voraussetzungen erfüllt und die einzelnen Dienste können dem Projekt hinzugefügt werden.

```
[143302S0:amplify-kumo] master # amplify init

Enter a name for the project: amplifykumo
Enter a name for the environment: dev
Choose your default editor: Visual Studio Code
Choose the type of app that you're building: javascript
Please tell us about your project
What javascript framework are you using: react
Source Directory Path: src
Distribution Directory Path: build
Build Command: npm run-script build
Start Command: npm run-script start

Using default provider awscloudformation
Do you want to use an AWS profile? (Y/n) Y
Please choose the profile you want to use: amplify-kumo
Initializing project in the cloud...

[...]

CREATE_COMPLETE DeploymentBucket AWS::S3::Bucket
CREATE_COMPLETE UnauthRole      AWS::IAM::Role
CREATE_COMPLETE AuthRole        AWS::IAM::Role
Initializing project in the cloud...

CREATE_COMPLETE amplify-kumo AWS::CloudFormation::Stack
Successfully created initial AWS cloud resources for
deployments.
Initialized provider successfully.
Initialized your environment successfully.

Your project has been successfully initialized and
connected to the cloud!
```

## 4.4 Implementierung API und Datenbank

Der erste benötigte Dienst für die Webanwendung ist die GraphQL API. Mit `amplify add api` startet der Konfigurationsassistent. Da die Authentifizierung mit Cognito erst zu einem späteren Zeitpunkt hinzugefügt wird, erfolgt sie solange mittels API Key. Die Änderung der Authentifizierung erfolgt im Abschnitt *4.6 Implementierung Authentifizierung*. Nach Bestätigung des letzten Schrittes öffnet sich die Datei `schema.graphql`, die das Schema der API darstellt. In dieser Datei werden alle benötigten Objekttypen erstellt.

```

Please select from one of the below mentioned services: GraphQL
Provide API name: amplify-kumo-api
Choose the default authorization type for the API: API key
Enter a description for the API key: My-Dev-ApiKey
After how many days from now the API key should expire (1-365): 7
Do you want to configure advanced settings for the GraphQL API: No
Do you have an annotated GraphQL schema? No
Do you want a guided schema creation? Yes
What best describes your project:
Single object with fields (e.g., Todo with ID, name, description)
Do you want to edit the schema now? Yes

```

Im Rahmen der Bachelorarbeit wird nur ein Objekttyp mit dem Namen Account benötigt. Es werden weiterhin alle verfügbaren Felder angegeben. Werden in Zukunft neue Felder oder weitere Typen benötigt, kann die Datei `schema.graphql` jederzeit angepasst werden.

```

type Account @model {
  id: ID!
  accountid: String!
  name: String!
  email: String!
  num: Int!
  status: String!
}

```

Alle angegebenen Felder müssen im späteren Verlauf von der Lambda-Funktion abgerufen und gespeichert werden. Mit dem Befehl `amplify push` wird die API in der Cloud bereitgestellt. Im folgenden Dialog besteht die Möglichkeit alle möglichen Mutationen und Queries generieren zu lassen. Wie bereits im Abschnitt 3.2.2 *GraphQL API: AWS Appsync* erwähnt, wäre es bei GraphQL normalerweise üblich die Query-Typen etc. manuell zu erzeugen. AppSync übernimmt hier also wie versprochen einen großen Teil der Arbeit.

```

# You will be walked through the following questions for GraphQL code
  generation

Do you want to generate code for your newly created GraphQL API?  Y
Choose the code generation language target: javascript

Do you want to generate/update all possible GraphQL operations -
queries, mutations and subscriptions?  Y

```

Sobald der Prozess fertig gestellt ist, sind die Dienste AWS AppSync und DynamoDB konfiguriert. Dank DynamoDB-Resolver wurde eine entsprechende DynamoDB-Tabelle angelegt und konfiguriert. In der Webkonsole von AWS kann überprüft werden, ob der Vorgang erfolgreich war. Die Ansicht von AppSync zeigt im Menüpunkt "Data Sources", dass eine DynamoDB-Tabelle passend zum GraphQL Objekttyp erstellt wurde. Der Name der Tabelle wurde nach



dem GraphQL Objekttyp gewählt. Nun ist es möglich per API auf die Datenbank zuzugreifen und Daten abzufragen. Die jeweilige Abfrage findet im Abschnitt *4.7.1 Programmierung React-Frontend* statt. Bevor das Frontend jedoch Anfragen an die Datenbank ausführen kann, muss Lambda konfiguriert werden, sodass Daten gespeichert werden.

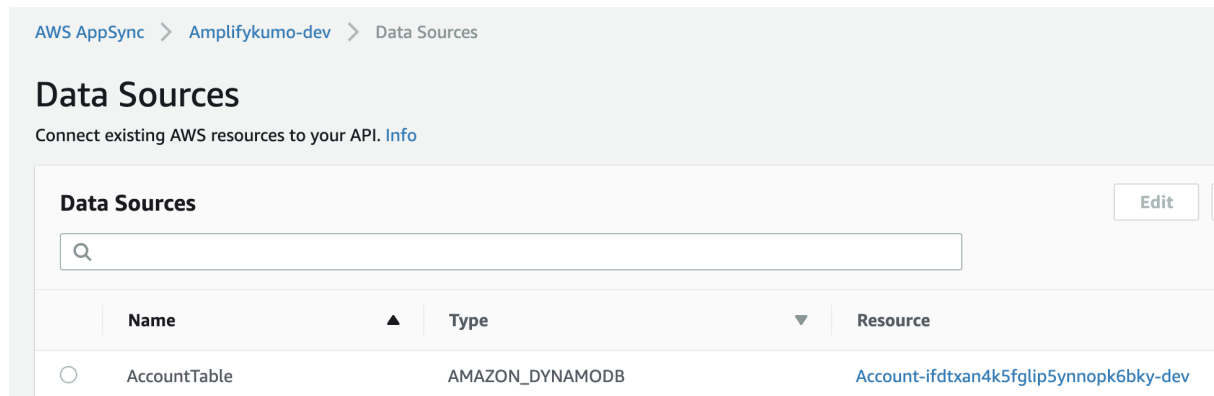


Abbildung 4: Erfolg der GraphQL API Erstellung überprüfen

## 4.5 Implementierung Backend-Logik

Die gesamte Backend-Logik wird durch Lambda realisiert. Mit dem Befehl `amplify add function` startet Amplify den Prozess. Dabei ist es im Rahmen der Bachelorarbeit ausreichend, wenn die Lambda-Funktion in regelmäßigen Abständen ausgeführt wird. Weitere Auslöser können in Zukunft ergänzt werden. In der Regel werden neue AWS Accounts in Abständen von 2-3 Wochen erstellt.

```
[143302S0:amplify-kumo] master # amplify add function

Using service: Lambda, provided by: awscloudformation
Provide a friendly name for your resource to be used as
a label for this category in the project:  getallaccounts
Provide the AWS Lambda function name:  getallaccounts
Choose the function runtime that you want to use:  NodeJS
Choose the function template that you want to use:  Hello World
Do you want to access other resources created in this
project from your Lambda function?  Yes
Select the category:  storage
Do you want to invoke this function on a recurring schedule?  Yes
At which interval should the function be invoked:  Daily
Select the start time (use arrow keys):  06:35 PM

Successfully added resource getallaccounts locally.
```

Nachdem Amplify die Lambda-Funktion erstellt hat, kann diese direkt bearbeitet werden. Der vollständige Code befindet sich im Anhang (Siehe Anhang 5 *Lambda-Code in NodeJS*). Im Kern besteht die Lambda-Funktion aus vier Funktionen.

```
getCallerIdentity()  
getCrossAccountCredentials()  
listAllAccounts()  
writeAllDynamoDBItems()
```

Die Funktionen `getCallerIdentity()` und `getCrossAccountCredentials()` dienen den Zweck temporäre Zugriff auf den Account `Cbc-Master` zu erhalten. `listAllAccounts()` liest anschließend alle Daten und `writeAllDynamoDBItems()` schreibt diese in die `DynamoDB`-Tabelle. Eine Schwierigkeit war es, eine Identität im `Cbc-Master` anzunehmen, eine weitere die korrekte Abarbeitung der Funktion sicherzustellen. In den folgenden Abschnitten werden diese beiden Hürden genauer erläutert.

#### 4.5.1 Accountübergreifender Zugriff

Die Lambda-Funktion befindet sich im `AWS Account Cbc-Clouds-Sandbox` und muss auf den Account `Cbc-Master` zugreifen. Im Account `Cbc-Master` wird eine IAM-Rolle benötigt, die der Lambda-Funktion Zugriff gewährt. Der Abschnitt 4.2 *AWS Accountstruktur* beschreibt den Aufbau aller Accounts der Mediengruppe RTL. Mit IAM-Rollen können Benutzeraccounts oder AWS-Dienste temporäre Anmeldeinformationen erhalten, die Accountübergreifend funktionieren. Dabei wird die Identität der IAM-Rolle übernommen. Dementsprechend sind alle Operationen erlaubt, die der IAM-Rolle zugewiesen worden sind. Damit die Lambda-Funktion also in dem Account `Cbc-Master` auf `AWS Organizations` zugreifen kann, benötigt die IAM-Rolle mindestens Lesezugriff auf den Dienst `AWS Organizations`. AWS bietet hierfür die Berechtigung `AWSOrganizationsReadOnlyAccess`.

Außerdem muss eine Vertrauensstellung hergestellt werden. Das folgende Bild zeigt die konfigurierte Vertrauensstellung zwischen Lambda-Funktion im `Cbc-Clouds-Sandbox` und der IAM-Rolle im `Cbc-Master Account`. Im unteren Linken Bereich bei „Trusted Identities“ ist die Identität der Lambda-Funktion aus dem Account `Cbc-Clouds-Sandbox` zu sehen. 163962199350 ist die dazugehörige AccountID. Die IAM-Rolle hat den Namen „amplify-kumo-access-role“ und besitzt einen eindeutigen Ressourcennamen (ARN). Dieser ARN muss in der Lambda-Funktion angegeben werden. Nur so weiß weiß die Funktion welche Identität sie übernehmen soll.

Roles &gt; amplify-kumo-access-role

## Summary

Role ARN	arn:aws:iam::298094174079:role/amplify-kumo-access-role
Role description	<a href="#">Edit</a>
Instance Profile ARNs	
Path	/
Creation time	2020-07-14 16:21 UTC+0200
Last activity	2020-09-09 11:23 UTC+0200 (18 days ago)
Maximum session duration	1 hour <a href="#">Edit</a>
Give this link to users who can switch roles in the console	<a href="https://signin.aws.amazon.com/switchrole?roleName=amplify-kumo-access-role&amp;account=cbc-master">https://signin.aws.amazon.com/switchrole?roleName=amplify-kumo-access-role&amp;account=cbc-master</a>

[Permissions](#)
[Trust relationships](#)
[Tags](#)
[Access Advisor](#)
[Revoke sessions](#)

You can view the trusted entities that can assume the role and the access conditions for the role. [Show policy document](#)

[Edit trust relationship](#)

**Trusted entities**

The following trusted entities can assume this role.

Trusted entities
arn:aws:iam::163962199350:role/amplifykumoLambdaRole77657a07-dev

**Conditions**

The following conditions define how and when trusted

There are no conditions associated with this role.

Abbildung 5: Vertrauensstellung zwischen beiden Accounts

Um diesen Zugriff zu programmieren, dienen die Funktionen `getCallerIdentity()` und `getCrossAccountCredentials()`. `getCallerIdentity()` ist eine Hilfsfunktion und gibt die aktuelle Identität zurück. So ist es leichter zu überprüfen, ob die Rolle “amplify-kumo-access-role” im Cbc-Master Account erfolgreich angenommen wurde. Das eigentliche Übernehmen der IAM-Rolle erledigt `getCrossAccountCredentials()`. Es wird ein neues AWS Service-Objekt für den Dienst AWS Organizations erstellt und mithilfe der Funktion `sts.assumeRole()` ein temporärer Zugriff im Master-Account erstellt.

Ein Service-Objekt wird benötigt, um auf Service-Funktionen zugreifen zu können. Die meisten AWS-Dienste bieten mindestens ein passendes Service-Objekt an. In der Lambda-Funktion muss dementsprechend ein Service-Objekt für die Dienste AWS Organizations, DynamoDB und für die Hilfsfunktion AWS STS<sup>32</sup> erstellt werden. [Ama20o, ]

Für DynamoDB reicht etwa folgender Befehl aus um eine passendes Service-Objekt zu erstellen. Die Lambda-Funktion und die DynamoDB-Tabelle befinden sich im selben Account. Auch besitzt die Lambda-Funktion bereits alle nötigen Berechtigungen, da diese durch Amplify gesetzt wurden.

```
var docClient = new AWS.DynamoDB.DocumentClient
```

Für AWS Organizations müssen bei der Erstellung des Service-Objekts zusätzlich die temporären Anmeldeinformationen mitgegeben werden, da sich der Dienst in einem anderen Account befindet. Außerdem muss die Region geändert werden. AWS Organizations ist ein Dienst, der nur in der Region `us-east-1` betrieben wird. Für diesen Vorgang wird die Funktion

<sup>32</sup>STS steht für Security Token Service und ermöglicht die Bereitstellung von temporären Anmeldeinformationen.

`getCrossAccountCredentials()` benötigt. Im ersten Schritt werden temporäre Anmeldeinformation abgerufen und anschließend dem Service-Objekt übergeben.

Listing 1: Neues ServiceObject mit einer anderen Identität

```
var accessparams = await getCrossAccountCredentials();

const cbc_master_orgs = new AWS.Organizations({
  credentials: accessparams,
  region: 'us-east-1'
});
```

Im Anschluss ist es möglich die Funktion `listAllAccounts()` mit einer anderen Identität auszuführen. Dafür muss das neu erstellte ServiceObjekt übergeben werden.

```
listAllAccounts(cbc_master_orgs)
```

#### 4.5.2 Asynchrone Verarbeitung mit NodeJS

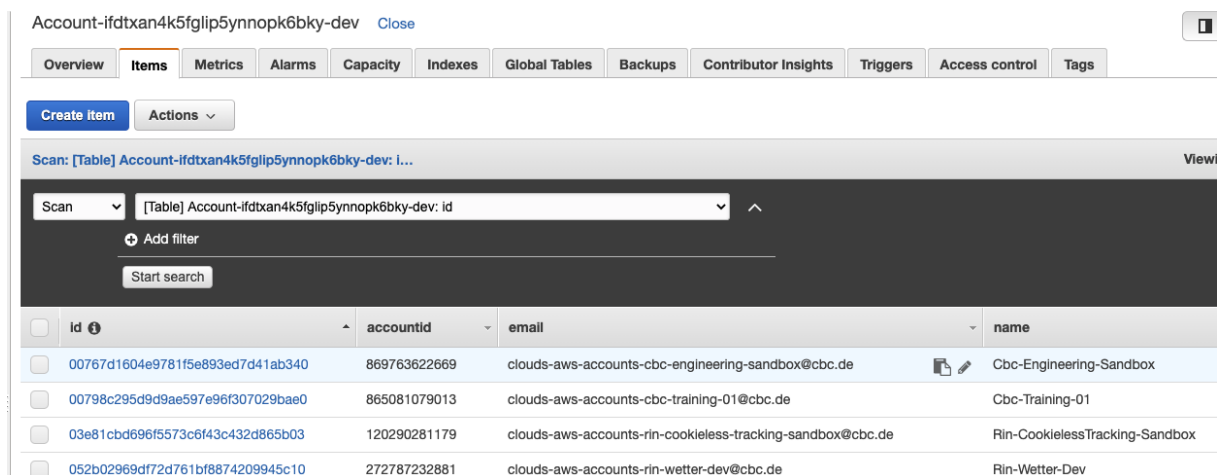
Alle vier Funktionen werden im Handler der Funktion nacheinander ausgeführt. Neben den unterschiedlichen Identitäten war die Sicherstellung der asynchronen Abarbeitung ein wichtiger Aspekt bei der Programmierung. Alle Funktionen durften nicht gleichzeitig ausgeführt werden, da sie voneinander abhängig waren. Damit die Funktion `listAllAccounts()` erfolgreich war, musste zuvor Identität des Master-Accounts angenommen werden. Zudem muss die Funktion `writeAllDynamoDBItems()` auf die Fertigstellung von `listAllAccounts()` warten, da sie sonst keine Daten zum speichern hat.

NodeJS arbeitet Single-Threaded. Das bedeutet, dass zu einem Zeitpunkt immer nur genau eine Aufgabe erledigt werden kann. Aufgrund der Abhängigkeit der Funktionen muss eine asynchrone Abarbeitung stattfinden. Zur Option stehen dazu Callbacks, Promises oder eine Implementierung mit `async/await`. Ein Callback ist eine Funktion die an eine weitere Funktion weitergegeben wird. So wird sichergestellt, dass die zweite Funktion erst aufgerufen wird, sobald die Erste beendet ist. Das Problem von Callbacks ist die große Unübersichtlichkeit bei mehreren Funktionen. Das immer tiefere verschachteln von Funktionsaufrufen wird auch als „Callback-Hölle“ bezeichnet. Promises ermöglichen die selbe Funktionalität wie Callbacks, nur ohne die Verschachtelung. „Ein Promise ist ein Objekt, das die finale Beendigung einer asynchronen Operation repräsentiert. Je nachdem, ob die Operation erfolgreich oder fehlerhaft beendet wurde, wird das Promise entsprechend gekennzeichnet.“ [Moz20a, ] Mit der Option `.then()` kann sichergestellt werden, dass die Operation beendet sein muss bevor der nächste Codeblock beginnt. Die letzte Möglichkeit ist die Verwendung von `async/await`, welche auf Promises basiert jedoch eine verständlichere Syntax ermöglicht. Eine Funktion kann mit `async` als Asynchron deklariert werden. Innerhalb einer asynchronen Funktion kann mit dem Operator `await` auf die Erfüllung des Promises gewartet werden. Es ist auch möglich den Operator `await` in einer Funktion mehrmals zu verwenden.

Die Lambda-Funktion nutzt Promises und `async/await` zu einem großen Teil aus um die Funktionalität gewährleisten zu können. Der Event Handler wird als Asynchron deklariert und innerhalb des Handlers werden alle oben genannten Funktion nacheinander aufgerufen. Die genaue Verwendung der asynchronen Abläufe befindet sich im Anhang.

### 4.5.3 Verifizierung Lambda-Funktion

Nach erfolgreicher Ausführung der Lambda-Funktion sollten sich Einträge in der DynamoDB-Tabelle befinden. Zur einfachen Überprüfung kann der Dienst DynamoDB in der Webkonsole geöffnet werden. Sind in der Tabelle Daten hinterlegt, war der Prozess erfolgreich. Die folgende Grafik zeigt den Erfolg der Lambda-Funktion. In der Grafik ist nur ein kleiner Ausschnitt der gespeicherten Accounts zu sehen. Um die Eindeutigkeit sicherzustellen, wurde als ID ein MD5-Hash aus der Kombination Accountnamen und AccountID gewählt. Dies ist notwendig, da nicht ausgeschlossen werden, dass ein anderer Wert eventuell erneut auftritt. Der Hash-Wert wird in Lambda mit dem Modul `crypto` realisiert.



The screenshot shows the AWS DynamoDB console for a table named 'Account-ifttxan4k5fglip5ynnopk6bky-dev'. The 'Items' tab is selected, displaying a list of account entries. Each entry includes a unique ID (MD5 hash), an account ID, an email address, and a name.

id	accountid	email	name
00767d1604e9781f5e893ed7d41ab340	869763622669	clouds-aws-accounts-cbc-engineering-sandbox@cbc.de	Cbc-Engineering-Sandbox
00798c295d9d9ae597e96f307029bae0	865081079013	clouds-aws-accounts-cbc-training-01@cbc.de	Cbc-Training-01
03e81cbd696f5573c6f43c432d865b03	120290281179	clouds-aws-accounts-rin-cookieless-tracking-sandbox@cbc.de	Rin-CookielessTracking-Sandbox
052b02969df72d761bf8874209945c10	272787232881	clouds-aws-accounts-rin-wetter-dev@cbc.de	Rin-Wetter-Dev

Abbildung 6: Verifizierung der Lambda-Funktion

## 4.6 Implementierung Authentifizierung

Im Abschnitt 3.4.2 *Alternative und Entscheidung* wurde bereits erwähnt, dass Amplify in Kombination mit Cognito alle benötigten Module für eine vollständige Authentifizierung beinhaltet. Um die erforderliche Komponente hinzuzufügen, muss der Befehl `amplify add auth` ausgeführt werden. Im darauffolgenden Assistenten kann die gewünschte Option zur Anmeldung gewählt werden. Nach dem Befehl `amplify push` wird ein Cognito User Pool inklusive App Clients erstellt. Über die Webkonsole besteht im Anschluss Option die Option MFA zu aktivieren.

Für die Einbindung in das React-Frontend bietet das Amplify-Framework die Komponente `withAuthenticator` an. Diese muss um die Hauptkomponente umhüllt werden. Dafür wird beim exportieren der Hauptkomponente `withAuthenticator` mitangegeben. [Amp20a, ] Die React-Komponente `AWSAccountListsMaterial` beinhaltet den Code zum Abfragen der Daten aus der DynamoDB-Tabelle und zum Rendern der Übersicht.

Mehr Details zu der Komponente befinden sich im Abschnitt 4.7.1 *Programmierung React-Frontend*.

## Listing 2: Auszug aus React-Frontend

```
import { withAuthenticator, AmplifySignIn } from "@aws-amplify/ui-react";

class AWSAccountListsMaterial extends React.Component {
  [...]
}

export default withAuthenticator(AWSAccountListsMaterial, true)
```

Startet man den Server im Anschluss und ruft die Anwendung auf, erscheint eine voll funktionsfähige Login-Maske. Diese Ansicht lässt sich auf Wunsch auch anpassen. Damit nicht jede Person einen Account erstellen kann, ist der Punkt „Allow users to sign themselves up“ in der Webkonsole deaktiviert und der Punkt „Only allow administrators to create users“ aktiviert worden. Wird über „Create Account“ versucht einen neuen Benutzer zu erstellen, erscheint Ohne Cognito und Amplify wäre die Implementierung einer sicheren und funktionierenden Authentifizierungsmethode deutlich aufwändiger. Dank der vorgefertigten Module ist es mit Amplify jedoch schnell umsetzbar.

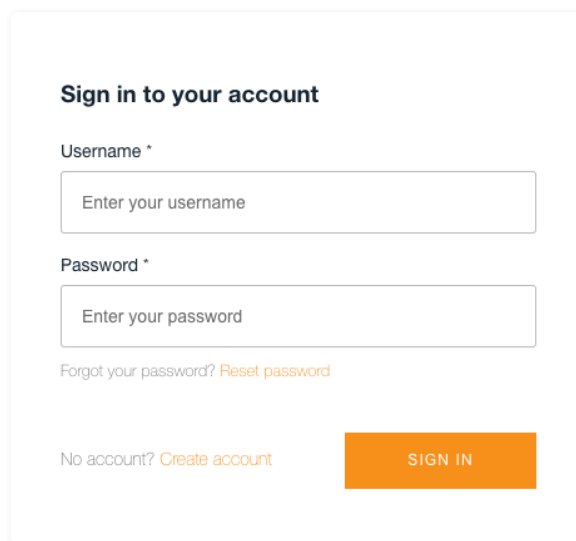


Abbildung 7: Login-Maske mit Amplify und Cognito

Da Cognito erfolgreich implementiert wurde, kann die Authentifizierung in der API geändert werden. Mit dem Befehl `amplify update api` kann jetzt der Cognito Benutzerpool ausgewählt werden. Im Anschluss kann die Änderung mit `amplify publish` veröffentlicht werden.

## Listing 3: Auszug aws-exports.js

```
Choose the default authorization type for the API
Amazon Cognito User Pool
```

### 4.6.1 Anmeldung über AzureAD

Wie im Abschnitt 3.4 *Authentifizierung* bereits erwähnt, wäre es wünschenswert die Anmeldung mit Unternehmensidentitäten zu ermöglichen. Die Mediengruppe RTL nutzt hierfür den Dienst Azure Active Directory. Mittlerweile sind einige Anbindung als SSO über AzureAD verfügbar. Die Anmeldung zu allen drei Cloud Providern erfolgt etwa über die gemeinsame Identität des Unternehmens.

Cognito bietet eine Unterstützung mit Unternehmens-Identitätsanbietern über SAML an. Dafür müssen Konfigurationen in AzureAD und AWS Cognito getätigt werden. Während des Vorgangs ist schließlich die Entscheidung gefallen, dass es keine geeignete Implementierung in Zusammenspiel mit Amplify gibt. Im Internet existieren mehrere Anleitungen zur Verbindung von AzureAD mit Cognito, jedoch ohne Amplify im Einsatz bei dem Frontend. Der Cognito Dienst wurde dabei mit einer „Azure Enterprise Application“ verknüpft. Bei „Azure Enterprise Application“ handelt es sich um einen Dienst, der zur Verwaltung von Identitäten genutzt wird. Nach der Einrichtung sind leider Fehlermeldungen aufgetreten. Das Ergebnis einer Recherche zeigte, dass Amplify für diese spezielle Anbindung nicht optimiert ist. Der SSO Authentifizierungsweg muss in das Frontend eingebaut werden. Für diesen Prozess existieren im Internet mehrere Dokumentationen für Android oder iOS Apps. Für React oder ein anderes JavaScript-Framework scheint die Integration komplexer zu sein. Auf der Website *dunlop.geek.nz* wird ein Lösungsvorschlag angeboten, der jedoch im Rahmen dieser Bachelorarbeit nicht umgesetzt wird. Laut der Anleitung ist es notwendig mehrere Stellen im Frontend stark zu bearbeiten und einen großen Teil der Authentifizierung manuell zu erstellen. Zudem ist es laut Autor notwendig die Datei `aws-exports.js` zu bearbeiten. In der Datei selbst wird davor explizit gewarnt.

Listing 4: Auszug `aws-exports.js`

```
// WARNING: DO NOT EDIT. This file is automatically generated by AWS  
Amplify. It will be overwritten.
```

Aufgrund des hohen Aufwands und der nicht zufriedenstellenden Lösung wird die Anwendung vorerst ohne eine Anbindung auskommen. Mitarbeiter, die Zugang benötigten, müssen sich über einen separaten Account anmelden. Falls in Zukunft eine bessere Option zur Verfügung steht, kann sie jederzeit hinzugefügt werden, ohne das bestehende System zu beeinträchtigen.

## 4.7 Implementierung Frontend und Hosting

### 4.7.1 Programmierung React-Frontend

Im letzten Schritt der Implementierung muss das Frontend konfiguriert werden und die Anwendung veröffentlicht werden. Bisher verfügt das Frontend über Authentifizierung, und dank der Lambda-Funktion sind Informationen in der DynamoDB-Tabelle verfügbar. Nun müssen mit React und AppSync eine Abfrage der Daten stattfinden. Im Anschluss werden diese Daten visualisiert.

Um erste Erfahrungen mit React zu sammeln, wurde eine Komponente zur Begrüßung erstellt. Die React-Komponente `<Greeting>` gibt zwei simple HTML Elemente mit einem definierten Text zurück. Als nächstes wurde die React-Komponente `<AWSAccountListsMaterial>` erzeugt, die für die restliche Logik zuständig ist.

Zuallererst wird eine Query benötigt, die alle Daten der Tabelle abfragt und in der Komponente genutzt werden kann. In der Query werden alle Felder ausgewählt, die im Schema angegeben wurden, da alle auch benötigt werden. Wie zuvor erwähnt wurde, ist es ebenfalls möglich auch nur einen Teil der Felder abzufragen und so ein Over-fetching zu vermeiden.

```
// GraphQL Query to get all data
const listAllAccounts = `query listAllAccounts {
  listAccounts (limit: 300) {
    items{
      id
      accountid
      num
      name
      email
      status
    }
  }
}`;
```

Auch hier muss eine asynchrone Abarbeitung sichergestellt werden. Die Daten müssen bereits zur Verfügung stehen bevor sie gerendert werden können. React bietet dazu die Funktion `componentDidMount()` an, die exakt für diesen Anwendungsfall existiert. [Fac20a, ] Innerhalb der Funktion `componentDidMount()` wird eine weitere Funktion `queryAccounts()` gestartet. Diese Funktion führt mithilfe von `AppSync` die Query aus und speichert sie in die State Variable. [Amp20b, ] State kann alle internen Daten einer Komponente beinhalten. Wird die State Variable mithilfe von `setState()` geändert, ruft React automatisch die Renderfunktion erneut auf und aktualisiert die Seite.

```
queryAccounts = async() => {
  try {
    const result = await API.graphql(graphqlOperation(
      listAllAccounts))
    const result_json = Object.values(result)
    const result_optimized = result_json[0].listAccounts.items
    // this will re render the view with new data
    this.setState({

      AccountList:result_optimized,})

  } catch (err) {
    console.log(err);
  }
}
```

Im letzten Schritt müssen die Information aus der State Variable im Frontend angezeigt werden. Hierfür eignet sich eine tabellarische Ansicht am meisten. Zuerst wurde mit JSX ein HTML `<table>` Element erstellt und die einzelnen Daten korrekt hinzugefügt. Da hierbei jedoch weitere Funktionalitäten manuell entwickelt werden müssten, wurde letztendlich die frei zugäng-



liche Komponente „material-table“ genutzt. Diese Komponente ermöglicht das Erstellen einer übersichtlichen Tabellen mit den wichtigsten Features, wie einer Suche oder einer Sortierfunktion. Zudem ist nahezu jeder Aspekt frei konfigurierbar, etwa die Anzahl an Elementen pro Seite. [htt20, ] Innerhalb der `<AWSAccountListsMaterial>` wird also die externe Komponente `<MaterialTable>` eingebaut. Hier werden die Spalten konfiguriert, sowie dem Feld „data“ die Daten aus der State Variable übergeben.

```
<MaterialTable
[... ]
columns=[
  { title: 'Nummer', field: 'num', defaultSort: "asc", width: '6%' },
  { title: 'Name', field: 'name', width: '30%' },
  { title: 'Account ID', field: 'accountid', type: 'numeric' },
  { title: 'Email', field: 'email', type: 'numeric', width: '50%',
    align: "center" },
  { title: 'Status', field: 'status', sorting: false, width: '6%' }
]
data={this.state.AccountList}
/>
```

Das folgende Bild zeigt das Ergebnis nach der Implementierung des Codes. Zum gegebenen Zeitpunkt existierten 162 Einträge, wovon jeweils 30 pro Seite angezeigt wurden. Die Implementierung aller Dienste wurde erfolgreich durchgeführt.

**Herzlich Willkommen bei dem Bachelorprojekt Kumo von Oktavius Wiesner**

**an der Hochschule Niederrhein**

**Tabelle mit allen AWS Accounts**

Account List <span>🔍 Search</span> <span>✕</span>				
Nummer <sup>1B</sup>	Name	Account ID	Email	Status
1	Rin-TVNow-Cdi-Preprod	892221805638	clouds-aws-accounts-rin-tvnow-cdi-preprod@cbc.de	ACTIVE
2	Rin-Tvnow-CloudFront	594083437171	clouds-aws-accounts-rin-tvnow-cloudfront@cbc.de	ACTIVE
3	Rin-SystemSquad-Prod	552604244352	clouds-aws-accounts-rin-systemsquad-prod@cbc.de	ACTIVE
4	Rin-Content-Dev	346335158574	clouds-aws-accounts-rin-content-dev@cbc.de	ACTIVE
5	Rin-TVNow-Accounts-Dev	358115957531	clouds-aws-accounts-rin-tvnow-accounts-dev@cbc.de	ACTIVE
6	Rin-Player-Sandbox	636452179547	clouds-aws-accounts-rin-player-sandbox@cbc.de	ACTIVE
7	Rin-Vms-DataVault	584761137075	clouds-aws-accounts-rin-vms-datavault@cbc.de	ACTIVE
8	Cbc-Master	298094174079	c25b5459905d9f0c339d08ac156180f4@aws.arvato.com	ACTIVE
9	Rin-TVNow-OTT-Sandbox	421842033545	clouds-aws-accounts-rin-tvnow-ott-sandbox@cbc.de	ACTIVE
10	Rin-HbbTV-Prod	948397395726	clouds-aws-accounts-rin-hbbtv-prod@cbc.de	ACTIVE

Abbildung 8: Fertiggestellte Website mit Daten

## 4.7.2 Hosting

Da alle Komponenten erfolgreich implementiert worden sind und die Webanwendung alle geforderten Aufgaben erfüllt, kann sie veröffentlicht werden. Mithilfe von

`amplify add hosting` kann die Webanwendung bei Amazon gehostet werden. In diesem Schritt kann Amplify zudem eine Integration mit GitHub übernehmen. Es wird ein bestimmtes Repository mit Amplify verknüpft. Sobald neuer Code in das Repository hochgeladen wird, startet Amplify ein automatisches Deployment mit diesen Änderungen. Werden mehrere Branches mit unterschiedlichen Umgebungen genutzt, können diese ebenfalls separat hinzugefügt werden. Nach Fertigstellung gibt Amplify eine URL zurück über die die Anwendung aufgerufen werden kann.

```
[143302S0:amplify-kumo] master # amplify add hosting
Select the plugin module to execute Hosting with Amplify Console
Managed hosting with custom domains, Continuous deployment

Choose a type Continuous deployment  Git-based deployments
Continuous deployment is configured in the Amplify Console.
Please hit enter once you connect your repository
[...]
Amplify hosting urls:

FrontEnd Env      Domain
dev               https://dev.drjgagjbys8gs.amplifyapp.com
```

Auf Wunsch kann eine eigene Domäne hinzugefügt werden, sodass die `.amplifyapp.com` Domäne nicht zwingend genutzt werden muss. In dem Account `Cbc-Clouds-Sandbox` ist bereits die Domäne `sandboxzone.aws-cbc.cloud` registriert. Somit kann direkt über den Menüpunkt „Domain Management“ eine Subdomäne für die Webanwendung konfiguriert werden. Falls die Domain über AWS registriert wurde, kümmert sich Amplify automatisch um das Erstellen des SSL Zertifikates und Setzen des CNAMEs.

The screenshot shows the 'Domain management' page in the Amplify Console. At the top, it says 'Custom domain: sandboxzone.aws-cbc.cloud'. Below this, there's a table with 'Domain' (sandboxzone.aws-cbc.cloud) and 'Status' (Domain activation). A progress bar shows three steps: 'SSL creation' (completed with a green checkmark), 'SSL configuration' (completed with a green checkmark), and 'Domain activation' (in progress with a blue circle and three dots). Below the progress bar, a message states: 'We verified domain ownership. We are propagating your custom domain to our global content delivery network which could take up to 30 minutes.' At the bottom, there's a table with 'URL' (https://amplify-kumo-dev.sandboxzone.aws-cbc.cloud) and 'Branch' (dev). A 'Troubleshooting guide' link is also visible.

Abbildung 9: Konfigurieren einer eigenen Domain

Sobald der DNS Eintrag global verfügbar ist, kann die Webanwendung unter folgender Adresse

aufgerufen werden. Für jede Entwicklungsumgebung lässt sich eine eigene Subdomain konfigurieren.

```
https://amplify-kumo-dev.sandboxzone.aws-cbc.cloud
```

## 5 Fazit

### 5.1 Diskussion

Ziel dieser Bachelorarbeit war es, die Serverless Architektur zu betrachten, und nach einer umfangreichen Betrachtung aller Dienste, einen Prototypen beim Cloud Provider AWS zu entwickeln. Ein wichtiges Merkmal war zudem, dass die Möglichkeit vorgesehen ist, dass die Webanwendung in Zukunft ohne viel Aufwand um weitere Funktionalitäten ergänzt werden kann. Die Wartung und Administration der Infrastruktur sollte dabei nach Möglichkeit so weit wie möglich vom Cloud Provider übernommen werden.

Im Rahmen der Bachelorarbeit lag der Fokus insbesondere auf die Designentscheidungen der einzelnen Dienste sowie eine grundsätzliche Funktionalität der Anwendung. Dazu gehören eine sichere Möglichkeit zur Authentifizierung sowie eine Backend-Logik, die bestimmte Daten abrufen und verarbeiten kann. Es ist sichergestellt, dass nur zugelassene Mitarbeiter die Anwendung nutzen können. Außerdem sollte das Web Frontend eine Übersicht über alle Daten darstellen können und bei Bedarf leicht angepasst werden. Neben dem Aspekt sich hauptsächlich um die Programmierung der Webanwendung zu kümmern, und nicht die zugrundeliegende Infrastruktur, sollten auch unnötige Kosten vermieden werden. Da die meisten Dienste nach benötigten Anforderungen abgerechnet werden, können die Kosten auf ein Minimum gehalten werden. Diese Ziele wurden erreicht. Die genaue Übersicht der Kosten folgt im nächsten Abschnitt.

Eine Integration mit Unternehmensidentitäten war zum Zeitpunkt leider nicht umsetzbar. Zum einem ist der Aufwand zu groß, zum anderem entspricht der potenzielle Lösungsvorschlag nicht den Wünschen.

Die gründliche Auseinandersetzung mit den einzelnen Dienste stellte sich als äußerst nützlich dar. Durch die Abstraktion von Amplify ist nicht jeder Prozess im Detail ersichtlich und es kann schwer fallen den genauen Zusammenhang zu verstehen. Dank der intensiven Beschäftigung des Designs ist es leichter die optimalen Anwendungsfälle zu erörtern, sowie auch außerhalb von Amplify mit den Diensten arbeiten zu können.

Die Entwicklung der Anwendung mit Amplify stellte sich als problemlos dar. Viele Vorgänge wurden merklich beschleunigt. Zudem war es so möglich innerhalb von kurzer Zeit eine vollständige serverlose Webanwendung mit den essentialsten Funktionen zu entwickeln, die keine Wartung oder Administration benötigt.

## 5.2 Kosten

Neben dem minimalen Aufwands zählen die Kosten ebenfalls zu den Vorteilen der Serverless Architektur. Während des gesamten dritten Kapitels wurden Informationen zur Abrechnung der einzelnen Dienste aufgestellt. Viele Dienste bieten kostenlose Kontingente an. Dies führt vor allem zu extrem geringen Kosten während der Entwicklungsphase.

Das folgende Bild bestätigt diese Annahme. In den Monaten August und September sind nur Kosten für die Dienste AppSync und DynamoDB entstanden. Lambda, Amplify und Cognito sind während des Zeitraums ohne Kosten genutzt worden. Die Kosten für September belaufen sich auf 0.12 USD für AppSync und 0.02 USD für DynamoDB. Im August sind die Kosten vergleichbar. Sobald die Anwendung produktiv genutzt wird, und die kostenlosen Kontingente aufgebraucht sind, werden die Kosten höher ansteigen.

Zur Zeit kann der Eindruck bestätigt werden, dass die Kosten mit Serverless Diensten gering gehalten werden können. Eine vergleichbare Anwendung für ca. 0.15 USD pro Monat ist nur sehr schwer bis gar nicht realisierbar. Wie sich die Kosten bei intensiver Nutzung verhalten muss zu einem späteren Zeitpunkt erneut evaluiert werden.

Service	Aug 2020	Sep 2020*	Service Total
Total cost (\$)	0.16	0.15	0.31
AppSync (\$)	0.13	0.12	0.25
DynamoDB (\$)	0.03	0.02	0.05
Lambda (\$)		0.00	0.00
Amplify (\$)		0.00	0.00
Cognito (\$)		0.00	0.00

Abbildung 10: Kostenaufstellung der letzten zwei Monate

### 5.3 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde die Serverless Architektur analysiert und ein funktionsfähiger Prototyp entwickelt. Als Motivation gilt ein steigendes Interesse an Function as a Service Diensten sowie an einer schnelleren Bereitstellung von Diensten. Die Kostenübersicht aller Cloud-Provider der Mediengruppe RTL soll vereinfacht werden und zentral aufbereitet werden. Damit geht die Anforderung einher eine einfache und administrationsfreie Realisierung zu ermöglichen, die im Besten Fall keine hohen Kosten verursacht. Um eine qualifizierte Aussage zu einer Implementierung mit Serverless Diensten geben zu können, wurden im voraus die wichtigsten Cloud Computing-Modelle erläutert und den Zusammenhang zu Serverless erklärt. Im Anschluss wurde die Serverless Architektur auf ihre Eignung geprüft und bestätigt. Die Umsetzung sollte möglichst von nur einer Person erfolgen, ohne Expertenwissen in jedem Fachgebiet zu erfordern.

Da es mehrere mögliche Ansätze zur Umsetzung gibt, ist es besonders wichtig sich mit den einzelnen Möglichkeiten auseinanderzusetzen. Aus diesem Grund wurden im Rahmen der Bachelorarbeit einzelne Dienste des Cloud Providers AWS untersucht und miteinander verglichen. Nachdem die zugrundeliegende Technologie des Dienstes und der Funktionsumfang des Dienstes geprüft wurden, konnte schließlich eine Entscheidung gefällt werden. Die einzelnen zu prüfenden Bereiche beschäftigten sich mit den Themen API, Datenbanken, Backend-Logik und Authentifizierung. In jedem Bereich wurde ein passender Dienst ausgewählt und im weiteren Verlauf auch implementiert. Als API wurde der AWS Dienst AppSync genutzt, der auf GraphQL basiert. Der Dienst AWS Cognito übernimmt die Authentifizierung und Absicherung der Anwendung. Als relationale Datenbank kommt der Dienst AWS DynamoDB zum Einsatz. Die Backend-Logik wird mit dem Function as a Service Dienst AWS-Lambda realisiert. Das Frontend wurde mit dem JavaScript-Framework React erstellt. AWS Amplify dient als zentraler Dienst, um alle zuvor genannten Komponenten an zentraler Stelle konfigurieren zu können.

Das Ziel des Prototypen war es, eine funktionale Webanwendung zu erzeugen die eine Übersicht über alle aktiven AWS Accounts liefert. Dafür musste ein Accountübergreifender Zugriff eingerichtet werden, da sich die benötigten Daten in einem anderen Account befinden als die Anwendung selbst. Die gesammelten Daten speichert die Lambda-Funktion in eine DynamoDB-Tabelle ab. Auf diese Daten greift das Frontend über die API zu, und erstellt mithilfe von React eine übersichtliche Tabelle aller Daten. Da die gesamte Webanwendung im Internet erreichbar ist, wurde mit AWS Cognito eine Authentifizierung sichergestellt. Registrierte Nutzer können die Webanwendung nutzen und auf die Daten zugreifen.

Während der Entwicklung wurde darauf geachtet eine möglichst einheitliche Programmiersprache inklusive Syntax zu verwenden. Aufgründdessen können andere Kollegen der Abteilung Datacenter and Clouds, ohne viel Einarbeitung, an dieser Webanwendung weiterentwickeln. Zudem ist dank Anbindung an den Dienst GitHub eine gemeinsame Bearbeitung mühelos möglich. Die Versionsverwaltung ermöglicht es zudem leicht Änderungen nachzuvollziehen.

## 5.4 Ausblick

Das gesammelte Wissen über die Serverless Architektur sowie über alle gewählten Dienste kann in Zukunft ebenfalls bei weiteren Projekten von Datacenter and Clouds angewandt werden können. Für künftige Projekte kann nun entschieden werden, ob sich eine Umsetzung mit Serverless Diensten anbietet oder ob die Realisierung mit einem anderen Service-Modell besser geeignet ist.

Für die Webanwendung selbst wurde das Grundgerüst geschaffen und es kann in Zukunft von jedem Mitarbeiter der Abteilung Datacenter and Clouds erweitert werden. Insgesamt gibt es mehrere Bereiche in denen die Webanwendung ausgebaut werden kann.

Für den Cloud Anbieter AWS stehen API und grundlegenden Funktionalitäten bereits zur Verfügung. Allen interessierten Mitarbeiter kann Zugriff zur Anwendung gewährt werden. Da die Kosten der einzelnen AWS Accounts für die einzelnen Abteilungen interessant sind, könnte die Lambda Funktion um diese Aufgabe erweitert werden. Jeder AWS Account zeigt eine detaillierte Übersicht der Kosten im AWS CostExplorer an. Mithilfe des CostExplorer SDKs können beispielsweise die Gesamtkosten der letzten Monate abgerufen werden. Zur Realisierung wäre es notwendig in jedem einzelnen AWS Account eine IAM-Rolle mit einer Vertrauensstellung zur Lambda-Funktion zu erstellen. Die Lambda-Funktion müsste anschließend die IAM-Rolle in jedem Account annehmen und die Abfragen gegen den CostExplorer Dienst ausführen. Zusätzlich müssten sowohl Datenbank als auch das Frontend angepasst werden. Diese Informationen könnten für die jeweiligen Kostenstellen interessant sein.

Weiterhin sollte in Zukunft erneut die Anmeldung über Unternehmensidentitäten in Betracht gezogen werden, falls es eine vorteilhaftere Methode zur Realisierung gibt.

Da bisher noch keine nützliche Auswertung von Log-Dateien oder Metriken existiert, sollte dies im weiteren Verlauf implementiert werden.

Neben AWS können auch andere Cloud Provider in die Anwendung integriert werden. Mit einer weiteren Lambda-Funktion könnte auf Daten innerhalb der Google Cloud Platform oder Azure Cloud zugegriffen werden. So wäre eine zentralisierte Stelle für alle Cloud Provider möglich.

## Literaturverzeichnis

- [Ale17] Alexander Lapp / Nico Litzel, *Big data, sql und nosql – eine kurze Übersicht*, <https://www.bigdata-insider.de/big-data-sql-und-nosql-eine-kurze-uebersicht-a-602249/>, 2017, [Abgerufen am 07.09.2020].
- [Ama20a] Amazon Web Services, *Abonnement-richtlinien für das graphql-schema*, [https://docs.aws.amazon.com/de\\_de/appsync/latest/devguide/real-time-data.html](https://docs.aws.amazon.com/de_de/appsync/latest/devguide/real-time-data.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20b] ———, *Amazon api gateway*, <https://aws.amazon.com/de/api-gateway/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20c] ———, *Amazon api gateway preise*, <https://aws.amazon.com/de/api-gateway/pricing/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20d] ———, *Amazon cognito*, <https://aws.amazon.com/de/cognito/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20e] ———, *Amazon cognito-preise*, <https://aws.amazon.com/de/cognito/pricing/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20f] ———, *Amazon dynamodb*, <https://aws.amazon.com/de/dynamodb/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20g] ———, *Amazon relational database service (rds)*, <https://aws.amazon.com/de/rds/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20h] ———, *Aws amplify*, <https://aws.amazon.com/de/amplify/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20i] ———, *Aws amplify features*, [https://aws.amazon.com/de/amplify/features/?nc1=h\\_ls](https://aws.amazon.com/de/amplify/features/?nc1=h_ls), 2020, [Abgerufen am 07.09.2020].
- [Ama20j] ———, *Aws appsync*, <https://aws.amazon.com/de/appsync/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20k] ———, *Aws appsync – preise*, <https://aws.amazon.com/de/appsync/pricing/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20l] ———, *Aws lambda – häufig gestellte fragen*, <https://aws.amazon.com/de/lambda/faqs/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20m] ———, *Aws lambda – preise*, <https://aws.amazon.com/de/lambda/pricing/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20n] ———, *Class: Aws.organizations*, <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Organizations.html#listAccounts-property>, 2020, [Abgerufen am 07.09.2020].



- [Ama20o] ———, *Erstellen und aufrufen von service-objekten*, [https://docs.aws.amazon.com/de\\_de/sdk-for-javascript/v2/developer-guide/creating-and-calling-service-objects.html](https://docs.aws.amazon.com/de_de/sdk-for-javascript/v2/developer-guide/creating-and-calling-service-objects.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20p] ———, *Häufig gestellte fragen*, <https://aws.amazon.com/de/amplify/faqs/?nc=sn&loc=5>, 2020, [Abgerufen am 07.09.2020].
- [Ama20q] ———, *Integration von amazon cognito in web- und mobile apps*, [https://docs.aws.amazon.com/de\\_de/cognito/latest/developerguide/cognito-integrate-apps.html](https://docs.aws.amazon.com/de_de/cognito/latest/developerguide/cognito-integrate-apps.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20r] ———, *Preise für on-demand-kapazität*, <https://aws.amazon.com/de/dynamodb/pricing/on-demand/>, 2020, [Abgerufen am 07.09.2020].
- [Ama20s] ———, *Serverless architectures with aws lambda*, <https://dl.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf>, 2020, [Abgerufen am 07.09.2020].
- [Ama20t] ———, *Verwenden von aws lambda mit amazon dynamodb*, [https://docs.aws.amazon.com/de\\_de/lambda/latest/dg/with-ddb.html](https://docs.aws.amazon.com/de_de/lambda/latest/dg/with-ddb.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20u] ———, *Verwenden von aws lambda mit sonstigen services*, [https://docs.aws.amazon.com/de\\_de/lambda/latest/dg/lambda-services.html](https://docs.aws.amazon.com/de_de/lambda/latest/dg/lambda-services.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20v] ———, *Was ist amazon cognito?*, [https://docs.aws.amazon.com/de\\_de/cognito/latest/developerguide/what-is-amazon-cognito.html](https://docs.aws.amazon.com/de_de/cognito/latest/developerguide/what-is-amazon-cognito.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20w] ———, *Was ist amazon elastic container service?*, [https://docs.aws.amazon.com/de\\_de/AmazonECS/latest/developerguide/Welcome.html](https://docs.aws.amazon.com/de_de/AmazonECS/latest/developerguide/Welcome.html), 2020, [Abgerufen am 07.09.2020].
- [Ama20x] ———, *Was ist aws?*, [https://aws.amazon.com/de/what-is-aws/?nc1=f\\_cc](https://aws.amazon.com/de/what-is-aws/?nc1=f_cc), 2020, [Abgerufen am 07.09.2020].
- [Ama20y] ———, *Was ist aws lambda?*, [https://docs.aws.amazon.com/de\\_de/lambda/latest/dg/welcome.html](https://docs.aws.amazon.com/de_de/lambda/latest/dg/welcome.html), 2020, [Abgerufen am 07.09.2020].
- [Amp20a] Amplify supported by Amazon Web Services, *Add authentication*, <https://docs.amplify.aws/start/getting-started/auth/q/integration/react#create-login-ui>, 2020, [Abgerufen am 07.09.2020].
- [Amp20b] ———, *Api (graphql)getting started*, <https://docs.amplify.aws/lib/graphqlapi/getting-started/q/platform/js>, 2020, [Abgerufen am 07.09.2020].
- [Amp20c] ———, *Prerequisites*, <https://docs.amplify.aws/start/getting-started/installation/q/integration/react>, 2020, [Abgerufen am 07.09.2020].

- [Apo20] Apollo Graph Inc., *Caching*, <https://www.apollographql.com/docs/apollo-server/performance/caching/>, 2020, [Abgerufen am 07.09.2020].
- [Bac20a] Back4App, *GraphQL vs rest*, <https://medium.com/@back4apps/graphql-vs-rest-62a3d6c2021d>, 2020, [Abgerufen am 07.09.2020].
- [Bac20b] Backendless Corp., *App making made simple.*, <https://backendless.com/>, 2020, [Abgerufen am 07.09.2020].
- [Cha19] Chandan Kumar, *5 best serverless security platform for your applications*, <https://geekflare.com/serverless-application-security/>, 2019, [Abgerufen am 07.09.2020].
- [Ele20] Elektronik-Kompodium.de, *Mqtt - message queue telemetry transport*, <https://www.elektronik-kompodium.de/sites/net/2204051.htm>, 2020, [Abgerufen am 07.09.2020].
- [Fac20a] Facebook Inc., *componentdidmount()*, <https://reactjs.org/docs/react-component.html#componentdidmount>, 2020, [Abgerufen am 07.09.2020].
- [Fac20b] ———, *Create a new react app*, <https://reactjs.org/docs/create-a-new-react-app.html>, 2020, [Abgerufen am 07.09.2020].
- [htt20] <https://material-table.com/>, *React data table component that is based on material-ui*, <https://reactjs.org/docs/react-component.html#componentdidmount>, 2020, [Abgerufen am 07.09.2020].
- [Mar20] Marc-Andre Giroux, *GraphQL & caching: The elephant in the room*, <https://www.apollographql.com/blog/graphql-caching-the-elephant-in-the-room-11a3df0c23ad/>, 2020, [Abgerufen am 07.09.2020].
- [Max20] Maximilian Schwarzmüller, *Angular vs react vs vue [2020 update]*, <https://academind.com/learn/angular/angular-vs-react-vs-vue-my-thoughts/>, 2020, [Abgerufen am 07.09.2020].
- [Mic19] Michael Novinson, *Check point to buy serverless security firm protego to protect cloud*, <https://www.crn.com/news/security/check-point-to-buy-serverless-security-firm-protego-to-protect-cloud>, 2019, [Abgerufen am 07.09.2020].
- [Mic20a] Microsoft Azure, *Was ist cloud computing?*, <https://azure.microsoft.com/de-de/overview/what-is-cloud-computing/#cloud-deployment-types>, 2020, [Abgerufen am 07.09.2020].
- [Mic20b] ———, *Was ist devops?*, <https://azure.microsoft.com/de-de/overview/what-is-devops/>, 2020, [Abgerufen am 07.09.2020].
- [Mic20c] ———, *Was ist ein container?*, <https://azure.microsoft.com/de-de/overview/what-is-a-container/>, 2020, [Abgerufen am 07.09.2020].

- [Mic20d] ———, *Was ist saas? saas (software-as-a-service)*, <https://azure.microsoft.com/de-de/overview/what-is-saas/>, 2020, [Abgerufen am 07.09.2020].
- [Mor20] Moritz Stückler, *Was ist eigentlich github?*, <https://t3n.de/news/eigentlich-github-472886/>, 2020, [Abgerufen am 07.09.2020].
- [Moz20a] Mozilla and individual contributors, *Promises benutzen*, [https://developer.mozilla.org/de/docs/Web/JavaScript/Guide/Using\\_promises](https://developer.mozilla.org/de/docs/Web/JavaScript/Guide/Using_promises), 2020, [Abgerufen am 07.09.2020].
- [Moz20b] ———, *Websockets*, <https://developer.mozilla.org/de/docs/WebSockets#Documentation>, 2020, [Abgerufen am 07.09.2020].
- [Red20] Red Hat, Inc., *Zustandsbehaftet oder zustandslos?*, <https://www.redhat.com/de/topics/cloud-native-apps/stateful-vs-stateless>, 2020, [Abgerufen am 07.09.2020].
- [res20] restfulapi.net, *What is rest*, <https://restfulapi.net/>, 2020, [Abgerufen am 07.09.2020].
- [Ste19] Steve Johnson, *Things to consider when you build a graphql api with aws appsync*, <https://aws.amazon.com/de/blogs/architecture/things-to-consider-when-you-build-a-graphql-api-with-aws-appsync/>, 2019, [Abgerufen am 07.09.2020].
- [The20a] The GraphQL Foundation, *Queries and mutations*, <https://graphql.org/learn/queries/>, 2020, [Abgerufen am 07.09.2020].
- [The20b] ———, *Schemas and types*, <https://graphql.org/learn/schema/#the-query-and-mutation-types>, 2020, [Abgerufen am 07.09.2020].
- [The20c] ———, *Serving over http*, <https://graphql.org/learn/serving-over-http/>, 2020, [Abgerufen am 07.09.2020].
- [Til19] Tilman Wittenhorst, *Bundespolizei speichert bodycam-aufnahmen in amazons aws-cloud*, <https://www.heise.de/newsticker/meldung/Bundespolizei-speichert-Bodycam-Aufnahmen-in-Amazons-AWS-Cloud-4324689.html>, 2019, [Abgerufen am 07.09.2020].
- [W3S20] W3Schools, *React jsx*, [https://www.w3schools.com/react/react\\_jsx.asp](https://www.w3schools.com/react/react_jsx.asp), 2020, [Abgerufen am 07.09.2020].

# Anhang

## A Anhang Lambda

---

Listing 5: Lambda-Code in NodeJS

---

```
/* Amplify Params - DO NOT EDIT
ENV
REGION
Amplify Params - DO NOT EDIT */

var AWS = require('aws-sdk')
AWS.config.update({region: 'eu-central-1'});

var docClient = new AWS.DynamoDB.DocumentClient

// Create AWS Service Objects for Lambda User
const s3 = new AWS.S3({apiVersion: '2006-03-01'});
const sts = new AWS.STS();

// Create Hash of String for Primary Key
const crypto = require('crypto')

// DynamoDB Helper Function. Currently not used.
async function getAllDynamoDBItems() {
  let params = {
    TableName: 'Comments-ifdtxan4k5fglip5ynnopk6bky-dev',
  };

  console.log("Starting Function")
  try {
    var result = await docClient.scan(params).promise()
    console.log(JSON.stringify(result))
    console.log("Success")
  } catch (error) {
    console.log("FAILED")
    console.error(error);
  }
}

// DynamoDB Function to write all collected Data
const writeAllDynamoDBItems = async
  ↪ (id,num,accountid,name,email,status) => {

  let params = {
```

```

TableName: 'Account-ifdtxan4k5fglip5ynnopk6bky-dev',
Item:{
  "id":id,
  "num":num,
  "accountid": accountid,
  "name": name,
  "email": email,
  "status": status
}
};

console.log("Writing Account " + name)
try {
  var result = await docClient.put(params).promise()
  console.log("Success")
} catch (error) {
  console.log("Failed Writing Files:")
  console.error(error);
}
}

// STS Function --> Get Current Identity
const getCallerIdentity = (sts_identity) => {
  return sts_identity.getCallerIdentity().promise();
};

// STS Function --> Assume Role from another AWS Account
const getCrossAccountCredentials = () => {
  return new Promise((resolve, reject) => {
    const timestamp = (new Date()).getTime();
    const params = {
      RoleArn:
        ↪ 'arn:aws:iam::298094174079:role/amplify-kumo-access-role',
      RoleSessionName: `New-Session-${timestamp}`
    };
    sts.assumeRole(params, (err, data) => {
      if (err) reject(err);
      else {
        resolve({
          accessKeyId: data.Credentials.AccessKeyId,
          secretAccessKey: data.Credentials.SecretAccessKey,
          sessionToken: data.Credentials.SessionToken,
        });
      }
    });
  });
};

// j helps to count all accounts
var j = 0;

```

```

var all_accounts = []

// Organizations Function --> List all Accounts
var data_all;
const listAllAccounts = (identity, params) => {
  return new Promise((resolve, reject) => {

    identity.listAccounts(params, (err, data) => {
      const params = {
        //NextToken: 'abc'
      };
      if (err) reject(err); // an error occurred
      else {
        console.log("Saved all Accounts");

        for (var i=0, max=data.Accounts.length; i < max; i++) {
          //console.log("Count: " + j + " ID:" +
            ↳ data.Accounts[i].Id + " is: " +
            ↳ data.Accounts[i].Name)
          j += 1;
          //all_accounts.push(data.Accounts[i].Name);
          let str1 = data.Accounts[i].Name;
          let str2 = data.Accounts[i].Id;
          let str3 = str1.concat(str2)

          let hash =
            ↳ crypto.createHash('md5').update(str3).digest("hex")

          all_accounts.push({id:hash, num:j,
            ↳ accountid:data.Accounts[i].Id,
            ↳ name:data.Accounts[i].Name
            ↳ ,email:data.Accounts[i].Email
            ↳ ,status:data.Accounts[i].Status});
        }

        if (data.NextToken != undefined) {
          console.log("There is more")

          params.NextToken = data.NextToken;
          // Both Resolve are needed here so that function can
            ↳ terminate successfull
          resolve( listAllAccounts(identity, params) )
        }
        else {
          console.log("ENDE ERREICHRT")
          //console.log(all_accounts)
          resolve (all_accounts);
        }
      }
    })
  })
}

```

```

        }    // successful response
    })
    });
} ;

exports.handler = async (event) => {
try {
    // Get getCrossAccountCredentials and Call Service Objects from AWS
    var accessparams = await getCrossAccountCredentials();
    const cbc_master_sts = new AWS.STS({
        credentials: accessparams
    });

    const cbc_master_orgs = new AWS.Organizations({
        credentials: accessparams,
        region: 'us-east-1'
    });

    //Check own and assumed Identity

    const msg_caller = await getCallerIdentity(sts)
    console.log("Username Current Identity: " + msg_caller.Arn)
    const msg_caller_org = await getCallerIdentity(cbc_master_sts)
    console.log("Username Assumed Identity: " + msg_caller_org.Arn)


    const all_acc = await listAllAccounts(cbc_master_orgs)
    const start = async () => {
        for (let element of all_acc) {
            await writeAllDynamoDBItems(element.id, element.num,
                ↪ element.accountid, element.name, element.email,
                ↪ element.status) ;
            //console.log(element);
        }
        console.log('Done');
    }

    await start();

}
catch(err){
    console.log("Error while Assuming Role from another Account.
        ↪ Message: " + err)
}
}

```

```
const response = {
  statusCode: 200,
  body: JSON.stringify('Hello from Lambda!'),
};
return response
};
```

---

## B Anhang React main.js

Listing 6: React index.js Main Render Funktion

---

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App'
import {AWSAccountLists} from './Oki'
import AWSAccountListsMaterial from './AccountListMaterial'
import { Greeting } from './Greeting'
import * as serviceWorker from './serviceWorker';
import Amplify from "aws-amplify";
import awsExports from "./aws-exports";

Amplify.configure(awsExports);

// Main Render Function

ReactDOM.render(
  <React.StrictMode>

    { /* Greeting Component with Header and additional Information
      ↪ */ }
    <Greeting name="Oktavius Wiesner" />

    { /* Greeting Component with Header and additional Information
      ↪ */ }
    < AWSAccountListsMaterial />

  </React.StrictMode>,

  //<App /> in React.StrictMode reinpacken.
```



```
document.getElementById('root')
);

// If you want your app to work offline and load faster, you can
  ↪ change
// unregister() to register() below. Note this comes with some
  ↪ pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

---

## C Anhang React Greeting.js

---

Listing 7: React Greeting.js Render a Greeting

---

```
import React from 'react'

export class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: "BLA",
    }
  }
  render() {
    return (
      <div>
        <h1> Herzlich Willkommen bei dem Bachelorprojekt Kumo von {
          ↪ this.props.name } </h1>
        <h1> an der Hochschule Niederrhein </h1>

        </div>

      );
    }
  }
```

---

## D Anhang React AccountsList.js

---

Listing 8: React AccountsList.js

---

```
//import React, { useEffect, useState } from 'react'
import React from 'react'
import { API, graphqlOperation } from 'aws-amplify'
import MaterialTable from 'material-table';
```

```

// GraphQL Query to get all data
const listAllAccounts = `query listAllAccounts {
  listAccounts (limit: 300) {
    items{
      id
      accountid
      num
      name
      email
      status
    }
  }
}`;

class AWSAccountListsMaterial extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      AccountList: []
    }
  }

  //componentDidMount() is invoked immediately after a component
  //↳ is mounted (inserted into the tree).
  //Initialization that requires DOM nodes should go here.
  //If you need to load data from a remote endpoint, this is a
  //↳ good place to instantiate the network request.
  componentDidMount() {
    this.queryAccounts();
  }

  // Function that fetches data and sets state
  queryAccounts = async() => {
    try {
      const result = await
        //↳ API.graphql(graphqlOperation(listAllAccounts))
        const result_json = Object.values(result)
        const result_optimized = result_json[0].listAccounts.items
        // this will re render the view with new data
        this.setState({

          AccountList:result_optimized, })

    } catch (err) {
      console.log(err);
    }
  }
}

```

```

// render all the information that is set into the state
render() {

  return (
    <div style={{ maxWidth: '90%' }} className="Accounts">

      <h2> Tabelle mit allen AWS Accounts </h2>

      <MaterialTable
        icons={{
          ResetSearch: () => <Clear />,
          Check: () => <Check />,
          Export: () => <SaveAlt />,
          Filter: () => <FilterList />,
          FirstPage: () => <FirstPage />,
          LastPage: () => <LastPage />,
          NextPage: () => <ChevronRight />,
          PreviousPage: () => <ChevronLeft />,
          Search: () => <Search />,
          ThirdStateCheck: () => <Remove />,
          ViewColumn: () => <ViewColumn />,
          DetailPanel: () => <ChevronRight />
        }}

        columns={[
          { title: 'Nummer', field: 'num', defaultSort: "asc",
            ↪ width: '6%' },
          { title: 'Name', field: 'name', width: '30%' },
          { title: 'Account ID', field: 'accountid', type:
            ↪ 'numeric' },
          { title: 'Email', field: 'email', type: 'numeric', width:
            ↪ '50%', align: "center" },
          { title: 'Status', field: 'status', sorting:false,
            ↪ width: '6%' }
        ]}
        data={this.state.AccountList}
        title="Account List"

        options= {{
          headerStyle: {
            backgroundColor: '#4b7d94',
            color: '#FFF'
          },
          rowStyle: {
            backgroundColor: '#EEE',
            fontSize: 14,
          },
          pageSize: 30,
          pageSizeOptions: [10,20,50],

```

```
        maxBodyHeight: '20%'
      }}
    />

  </div>

  );
}
}

export default withAuthenticator(AWSAccountListsMaterial, true)
```

---