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: 23. September 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

1 Einleitung

1.1 Aufgabenstellung

Das Ziel dieser Bachelorarbeit ist es Serverless 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.

1.2 Motivation

Die Abteilung Datacenter and Clouds innerhalb von CBC beschäftigt sich mittlerweile seit einigen Jahren mit Cloud Infrastruktur. Zu Beginn wurden größenteils dynamisch Linux Server mit einer Datenbank und einem Loadbalancer realisiert(Infrastructure as a Service). Mit der Zeit wurden auch Container-basierte Varianten mit Docker sowie Platform as a Service Lösungen umgesetzt. Function as a Service wurde bisher nur mit Amazon Web Services Dienst Lambda realisiert. Mittlerweile gibt es den Wunsch sich intensiv mit Function as a Service zu beschäftigen um vor allem eine schnelle Bereitstellung von Diensten zu geringen Kosten zu ermöglichen. Webanwendungen sollen schneller bereitgestellt werden, ohne dass Server konfiguriert und gewartet werden müssen.

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 seine Abrechnungen auf unterschiedliche Weise dar und alle Daten müssen unterschiedlich aufbereitet werden. AWS exportiert die Abrechnungen monatlich in dem Cloud-Speicher S3[1], wohingegen Google Cloud Plattform alle Daten in ein SQL ähnliches Data Warehouse speichert. Zudem ist es notwendig innerhalb der Mediengruppe RTL intern Abteilungen zuzuweisen und abzuschreiben.

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] S3 steht für Simple Storage Service und ist ein Objektspeicherservice von Amazon, für eine beliebe 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.

1.3 CBC Cologne Broadcasting GmbH

Die Bachelorarbeit wird innerhalb der Räumlichkeiten der Firma 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 die 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 das Thema Serverless aufgegriffen 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 ein Fazit der Bachelorarbeit.

2 Cloud Computing und Serverless

2.1 Allgemeines

Bevor man sich mit dem Design und der Implementierung von Serverless Architektur 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 Rechnerressourcen 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 keine Kosten im Voraus. Durch den Einsatz von Cloudtechnologien verlagert sich das Kostenmodel von Capex[2] zu Opex[3]. Mit einem Knopfdruck ist es möglich beliebig[4] viele Ressourcen und Dienste hochzufahren und zu verwenden. Je nach Servicemodell verschiebt sich der Verantwortlichkeitsbereich zwischen dem Cloud Provider und dem Anwender. [Mic20a,]

[2]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.

[3]Opex(Operational Expenditure) bezeichnet wiederkehrende Kosten wie Verwaltungs- und Betriebskosten. Auch fallen mittlerweile viele Softwarelizenzen regelmäßig an.

[4]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.

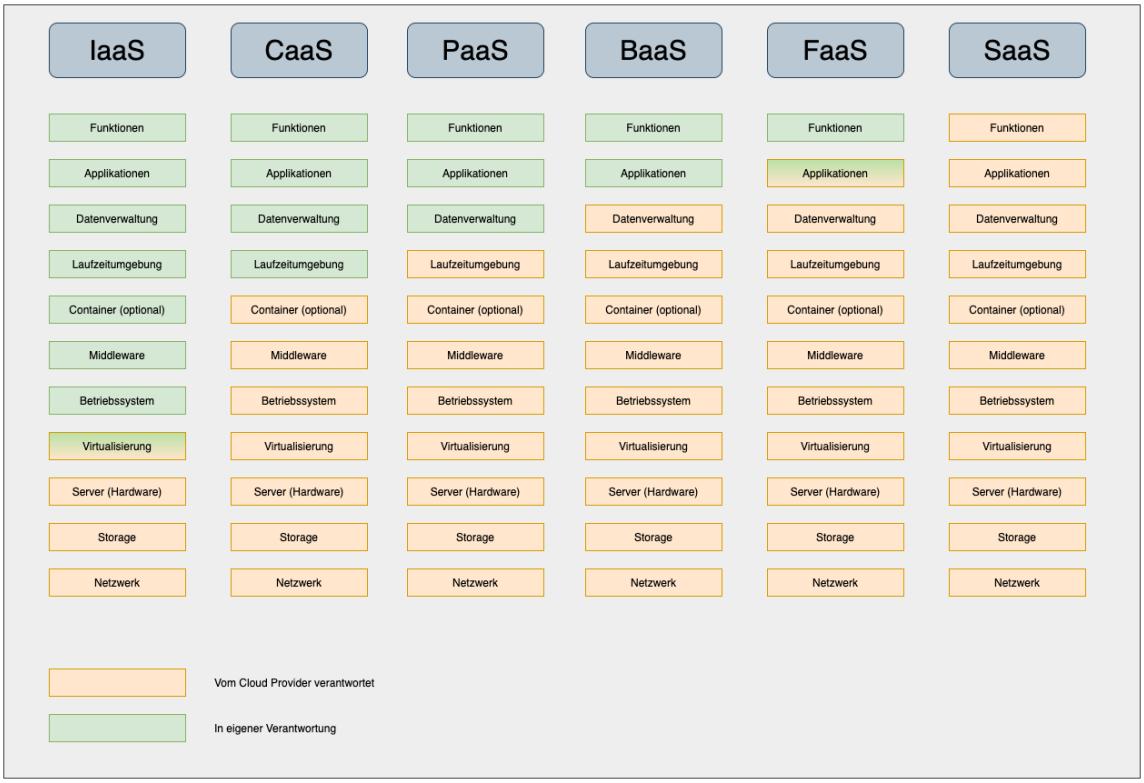

Abbildung 1: Übersicht der Servicemodelle

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 1). Dieses Modell ist am ehesten mit einer On Premises Infrastruktur vergleichbar. Der größter Vorteil jedoch sind die Initialkosten und das Investitionsrisiko. 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 „Standardsoftwarepaket “ 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 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. 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 von Google entwickelte Kubernetes zum Einsatz.

Typischerweise werden Container as a Service zur Bereitstellung von Microservices verwendet. Das Container as a Service Angebot des Cloud Providers AWS lautet ECS (Elastic Container Service). AWS bietet Nutzern die Möglichkeit ihre Container entweder Serverless oder auf EC2-Instanzen[5] zu hosten. [Ama20s,]

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 Alarme. Als Beispiele lassen sich hier 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 festlegen. Im Gegensatz dazu muss nur noch der Quellcode hochgeladen werden und die Anwendung könnte auf Wunsch bereits im Internet erreichbar sein, ohne sich um Themen wie Skalierbarkeit, Hochverfügbarkeit beschäftigen zu müssen. Für komplexere Aufgaben oder spezielle Anforde-

[5] 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.

rungen 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 wurde 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[6] 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 SQL oder NoSQL 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 einem 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 ermögliche. (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[7] reagiert und ausgeführt wird. Diese eventbasierten Funktionen ermöglichen dem Entwickler den Code on the fly 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. Beispielhaft für zustandslose Kommunikation ist ein Web- oder Druckserver. Jede Anfrage ist erst einmal unabhängig von bisherigen oder zukünftigen Anfragen. Sobald

[6] 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 Software Paket automatisch in beliebe Umgebungen deployed werden.

[7] Event, oder auch Ereignis, meint das reagieren auf eine Statusänderung einer bestimmten Ressource.

jedoch Tracking-Cookies mit einem eindeutigen Identifikator abgespeichert verwendet werden ändert sich die Kommunikation zu einer zustandsorientierten ab. [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. Es darf keine Bindung an die zugrunde liegende Infrastruktur existieren, da sonst die Zustandslosigkeit verloren ginge.

Insgesamt bietet Function as a Service eine simplifizierte und automatisch skalierte Möglichkeit eventbasierte Funktionen zu erzeugen. Nutzer können sich voll und ganz auf 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.X im Detail erläutert und im Abschnitt 4.X auch implementiert. [Ama20o,]

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 zum jeweiligen Betreiber zu haben sowie weniger Anpassungsmöglichkeiten als bei den bisher vorgestellten Modellen. Das 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. Hat ein Unternehmen alle Daten der Mitarbeiter in Microsoft 365 hinterlegt und nutzt zudem Exchange Online als Email Provider, ist es mit viel Aufwand verbunden zum Beispiel zur Konkurrenz von Google zu wechseln.

2.4 Serverless

Nach dem 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 soweit, dass sich Nutzer keine Gedanken über die Infrastruktur, das Betriebssystem oder Speicherverwaltung 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“ [Ama20o,] stellt hierzu vier Aspekte auf die zutreffen sollten. Diese lauten:

1) Der Anwender muss keine Server hochfahren oder warten. Auch ist er nicht für die Laufzeitumgebung der Software zuständig.

2) Die Applikation wird automatisch nach Bedarf flexibel skaliert. Die Skalierung selbst erfolgt nicht anhand von Servereinheiten sondern durch festlegen von Parametern wie Speicherverbrauch.

3) Es wird direkt vom Dienst eine Hochverfügbarkeit und Fehlertoleranz zur Verfügung gestellt. Jede Funktion oder Applikation die einen Serverless Dienst benutzt ist per Default auch Hochverfügbar.

4) Es dürfen keine Kosten für im Idle Zustand entstehen. Wird der Code nicht ausgeführt kostet er auch 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 man als 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 eingenständig durchlaufen können. Es muss ein Push basiertes Konzept entwickelt werden um Events ordnungsgemäß auslösen zu können.

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. Cloud-Provider erlauben Funktionen häufig nur eine maximale Ausführungszeit von 15 Minuten. Auch der Speicherverbrauch darf je nach Anbieter nicht mehr als 1,5GB bzw. 3GB RAM benötigen. Wird die Zeit oder der Speicherverbrauch überschritten, terminiert die Ausführung der Funktion sofort.

2.4.3 Vor- und Nachteile

Serverless Architekur 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 notwending 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 mehr, da jeder einzelne die gesamte Anwendung selbst erstellen und verwalten kann. Die Realisierung von Projekten mithilfe von Serverless Architketur unterstützt dabei die DevOps-Philosophie [8] und sorgt für mehr Agilität. 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. [Mic20b,]

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 des 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 mit großem Aufwand verbunden eine bereits bestehende Applikation zu einer Serverless Architekur zu migrieren. Der gesamte Ansatz muss 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 indem mehr Kontrolle über die Hard- und Software benötigt wird.

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.

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

[8] DevOps setzt sich aus den Begriffen Development und Operations zusammen. Der Begriff 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.

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 Datacenter 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 komplexen Konfigurationen erstellt werden. Die aufgestellten Anforderungen lassen sich ideal in isolierte Funktionen aufteilen. So könnte man zum Beispiel pro Cloud Provider eine oder mehrere Funktionen bereitstellen die komplett autark arbeiten. Der Umfang einer Funktion wäre etwa das Sammeln einer Kostenübersicht bei AWS.

Die gesammelten Daten können in einer relationalen Datenbank gespeichert werden und anschließend von einem Frontend dargestellt werden. Die Datenbank sollte im Besten Falle, genau 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 viele 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. [Ama20t,]

Mittlerweile gibt es mehr als 175 Services in aktuell 24 unterschiedlichen Regionen für den Kunden zur Nutzung. Jede Region besitzt in der Regel drei eigene Availability Zonen die geografisch voneinader getrennt sind. Die Regionen sind auf alle Kontinente verteilt. Auch in Deutschland wird mit dem Standort Frankfurt eine eigene Region angeboten. Vor allem in 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 mindestens einen passenden Service an. Jedes im vorherigen Kapitel besprochene Servicemodell besitzt mehrere passende AWS Services und es werden laufend neue angekündigt.

Auch für den Bereich Serverless gibt es mehrere Dienste. Um eine bestmögliche Entscheidung treffen zu können, werden im folgenden Abschnitt die wichtigsten Dienste untersucht und anhand der Ergebnisse eine Entscheidung für die Implementierung getroffen.

3.2 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.“ [Ama20g,]

Amplify lässt sich sowohl über die Amazon Web Console konfigurieren, als auch über eine eigene Amplify CLI. Das Amplify Framework ist Open Source und vereint native AWS Dienste zur Authentifizierung, Analyse, CI/CD, Hosting, Datenspeicherung für Desktop und Mobile Endgeräte. Dabei werden viele moderne Javascript Frameworks zum Aufbau des Frontends unterstützt, wie zum Beispiel React, React Native, Vue und weitere. Alle verfügbaren Dienste sind in der Amplify Bibliothek vereint und können in das Frontend eingebaut werden.

Amplify erleichtert es einem Projekte als Full Stack Developer[9] zu realisieren. Dies ermöglicht einem ein voll umfängliches Verständis über die gesamte Applikation und Änderungen können leichter durchgeführt werden. Ein Prototyp kann ebenfalls schnell erstellt werden.

[9] 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.

Das Amplify Framework lässt sich in drei Komponenten unterteilen: Bibliotheken, UI-Komponenten und einer CLI Toolchain, wobei jede Komponente einzeln oder gemeinsam genutzt werden kann.

Die Bibliotheken sind Open Source und nutzen bereits vorhandene AWS Dienste um eine native Webanwendung für die Cloud bereitzustellen. Folgende Module stehen zur Verfügung:

- Voll verwaltete Authentifizierung mit Unterstützung für Social Media Logins, wie Facebook, Google Sign-In. Multifaktorauthentifizerung, Authorisierung und Passwortwiederherstellung sind bereits inkludiert. Zugrundeliegender Dienst ist Amazon Cognito.

- DataStore zur Offline-Synchronisation zwischen mehreren Plattformen. (iOS/Android/React Native/ Webbrowser). Zugrundeliegender Dienst ist AWS AppSync.

- Echtzeitanylsen erstellen und auswerten sowie integriertes Tracking von Sessions. Zugrundeliegender Dienst Amazon Pinpoint und Amazon Kinesis.

- API Zugriffe auf Endpunkte mithilfe von GraphQL oder REST erstellen, sowie manipulieren und kombinieren von Daten unterschiedlicher Quellen. Zugrundeliegender Dienst ist Amazon AppSync und Amazon API Gateway.

- Simple Speicherverwaltung in öffentlichen oder privaten Storage Buckets. Kann auch benutzt werden um Useruploads zu verwalten. Zugrundeliegender Dienst ist Amazon S3.

- Zudem gibt es noch weitere Funktionen, wie Publish/Subcribe, Chatbots, Push Benachrichtungen, AR/VR sowie Künstliche Intelligenz. –¿ Erklärung der einzelnen Dienste mit Fußnote??

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 Interace Elemente für das Hochladen von Bildern und Datein 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. [Ama20h,]

Um alle unterschiedlichen AWS Dienste verwalten zu können nutzt Amplify einen weiteren AWS Dienst, AWS CloudFormation. Mit AWS CloudFormation ist möglich Templates zur Modellierung und Bereitstellung jedlicher AWS Ressourcen zu erstellen. Die Templates unterstützen JSON und YAML. Amplify übersetzt alle Befehle in ein Cloudformation Template und startet dieses. Dadurch ist es einfach alle, von Amplify durchgeführten Schritte, nachzuvollziehen und eventuell für andere Projekte zu übernehmen.

Im folgenden Schritt werden die einzelnen Dienste erläutert die für diese Bachelorarbeit und Implementierung notwendig sind. Zudem folgt eine Bewertung zur Eignung des jeweiligen Dienstes.

3.3 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.

Amazon bietet zwei Möglichkeiten an eine solche API zu erstellen welche im folgenden Schritt gegenüber gestellt werden. Einer der beiden Dienste wird anschließend für die Implementierung der Anwendung genutzt.

3.3.1 REST API: AWS API Gateway

Das Grundprinzip der REST (REpresentational State Transfer) API Architektur ist es eine strukturierte Kombination aus Ressourcen und Methoden zu erhalten. Ressourcen bestehen aus sogenannten URI (Unified Resource Identifier), und 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. Die gewünschten Daten sind immer über eine spezifische URI identifizierbar.

Beispiel für Abfragen könnten folgendermaßen aussehen:

```
GET  /AccountNames/
GET  /AccountNames/123
GET  /AccountEmail/12
POST  /Accounts/1
```

Mithilfe des ersten GET Requests kann eine Liste von Accounts abgerufen werden, der zweite GET Request liefert nur die Informationen für den 123 Eintrag der Accountnamen. Um nur zwei Namen zu erhalten sind dementsprechend auch zwei Requests notwendig, da sie separate Ressourcen sind. Die andere Option ist es alle Accountnamen zu erhalten und nur die benötigten heraus zu filtern. Der POST Request 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 ist eine Cache Implementierung mithilfe von REST leicht umsetzbar. Wächst eine Applikation immer weiter entstehen auch immer neue API Endpunkte. Möchte man nun alle Daten sammeln wären mehrere separate Anfragen notwendig. Es ist zudem nicht möglich die angeforderten Daten genauer zu spezifizieren. Mittels GET Request 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 erzeugen. Dabei steht er als zentrale Schnittstelle zwischen Endgeräten und Backend-Diensten. Zu den Endgeräten zählen Webanwendungen, Mobilgeräte oder auch IoT[10] Geräte. Anfragen werden vom Gateway anschließend je nach Konfiguration verarbeitet. Mithilfe des Dienstes lässt sich beispielhaft die zuvor beschriebene GET Anfrage an eine Datenbank weiterleiten, die die gewünschten Daten zurückgibt. Die POST Anfrage könnte eine Lambda-Funktion verarbeiten,

[10]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.

welche wiederum die neu erhaltenen Daten in die Datenbank 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 zu kommunizieren.

Zusätzlich bietet der Dienst viele weitere Funktionalitäten. Dazu gehören CORS-Support[11], eine Zugriffskontrolle und Einschränkung, oder auch die Verwaltung unterschiedlicher API-Versionen. Es ist auch möglich mit OnPremises Servern zu kommunizieren und Anfragen weiterzuleiten. [Ama20a,]

Die Abrechnung erfolgt anhand von API-Aufrufen und Datenübertragungen die in Richtung Internet verlaufen. Eine Pauschale gibt es nicht. Für die ersten 333 Millionen Anfragen pro Monat kosten in der Region Frankfurt eine Millionen API-Aufrufe 3,70 USD. Hat man mehr als 333 Millionen Aufrufe sinkt der Preis weiter. Bei über 20 Milliarden Aufrufen kosten 1 Millionen API-Aufrufe nur noch 1,72 USD. Darüber hinaus entstehen kosten falls man Caching nutzen möchte, und eine bessere Leistung für seine API benötigt. Hier kosten 0,5 GB an Zwischenspeicher 0,02 USD pro Stunde. Je höher der Speicher, desto höher auch der Stundensatz. [Ama20b,]

3.3.2 GraphQL API: AWS Appsync

2015 wurde GraphQL von Facebook veröffentlicht und 2018 in die Linux Foundation[12] ausgegliedert. GraphQL APIs arbeiten nicht mit Ressourcen oder HTTP Methoden. Stattdessen ist es wichtig zu Beginn ein statisches Schema mit der GraphQL Schema Definition Language zu erstellen Das Schema legt die Regeln fest wie der Client auf Daten zugreifen kann. Datentypen müssen im vorraus definiert sein. Zudem muss festgelegt werden, ob es sich um eine Mutation oder eine Query handelt.

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.

Ein Schema würde man mit GraphQL wie folgt darstellen:

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

[11] 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.

[12] Die Linux Foundation ist eine Gemeinnützige Organisation mit Sitz in der USA. Das Ziel ist es Linux zu fördern und den Wachstum zu unterstützen.

```
}
```

Den einzelnen Felder wird zugewiesen ob es sich um ein String, Int 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.

In GraphQL würde eine Query folgendermaßen aussehen:

```
query listAllAccounts {
  listAccounts {
       id
       accountid
       num
       name
       email
  }
}
```

Die Query gibt alle aufgelisteten Daten zurück. Wird zum Beispiel die Email nicht benötigt, lässt man das Feld weg und die Information wird nicht mehr an den Client übertragen. Zum Hinzufügen eines neuen Eintrags wäre folgende Mutation notwendig:

```
mutation {
  createAccount(
       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.

Neben den bisher genannten Funktion bietet GraphQL noch Subscriptions an. Diese ermöglichen dem Client Echtzeitbenachrichtigungen zu erhalten, sobald neue Daten dem Backend hinzugefügt worden sind. Dadurch dass GraphQL, anders als REST, nur einen einzelnen Endpunkt anbietet, ist eine Cache Implementierung problematischer. Bei einer REST API ist es möglich für jeden GET Request ein Cache-Control Header in der HTTP Anfrage zu implementieren, wodurch der Client das Caching übernehmen kann. Bei GraphQL muss zusätzlicher Code im Client implementiert werden oder das Schema entsprechend angepasst werden. Der Dienst Apollo Server ermöglicht es eine GraphQL API bereitzustellen, die bereits Cache Header unterstützt. Dafür muss die Option „cacheControl“ in das Schema eingebaut werden. Es ist auch möglich die Option für jedes individuelle Feld zu setzen. [Apo20,] Dadurch dass die Definition des Schemas mehr Zeit in Anspruch nimmt dauert es im Vergleich zu REST länger bis man erste Abfragen durchführen kann. [The20a,] [The20b,]

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 auch angegeben werden, dass eine Funktion aufgerufen werden soll.

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

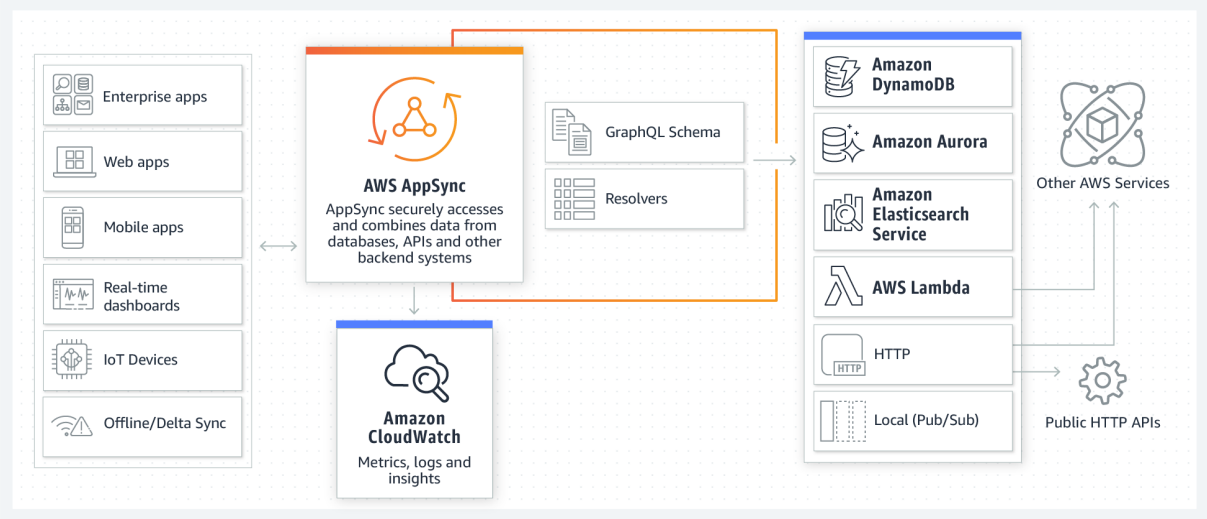

Abbildung 2: Funktionsweise von AppSync [Ama20i,]

So kann man direkt über AppSync eine NoSQL Datenbank mit AWS DynamoDB (Siehe XXX) und allen benötigten Berechtigungen aufsetzen. AWS stellt zudem auch alle benötigten Resolver für die Kommunikation zur Verfügung und kann auch auf Wunsch ein Schema erstellen. Durch die Resolver wird die Datenbank mit der API verknüpft und alle möglichen Queries und Mutationen erstellt. Darüber hinaus ist es möglich auch eigene Resolver zu erstellen, falls dies benötigt wird. Auch mit AWS Lambda und AWS Cognito(siehe XXX) ist der Dienst verbunden. Die Autorisierung von Clients ist mit API Keys[13] , IAM Credentials [14], OIDC Tokens [15] oder einem AWS Cognito User Pool (siehe XXX) möglich. [Ste19,]

Das AppSync SDK unterstützt iOS, Android und viele Javascript Varianten wie React, React Native oder Angular und kann mit dem Apollo Client genutzt werden. 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 Memory-Speicher Redis basiert. Um Caching nutzen zu können, ist es notwendig einen Instanztypen mit bestimmter Kapazität auszuwählen. 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. [Ama20i,] [Ama20j,]

[13]Der API Key wird nur für Entwicklungsumgebungen empfohlen, da er leicht kompromittierbar ist und keine Aufteilung von Berechtigungen ermöglicht.

[14]IAM steht für Identity Access Management und beschreibt Amazons Dienst zur Identität- und Benutzerverwaltung. Der Dienst erlaubt es Nutzer anzulegen und Ihnen Berechtigungen zu verteilen.

[15]OIDC steht für OpenID Connect und basiert auf dem OAuth 2.0 Protokoll zur Authentifizierung. Die Tokens zur Authentifizierung (JWT Tokens) einer Identität werden verschlüsselt im JSON Format versendet, und ermöglichen einen standardisierten Weg zum Anmelden.

3.3.3 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 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.

Da die in dieser Bachelorarbeit gewünschte Webanwendung Daten mehrerer Cloud Provider sammeln und aufbereiten soll, wird eine Implementierung zunehmend komplexer und größer. 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. optimieren sind mehr Daten notwendig. Da bisher nur bekannt ist welche Daten benötigt werden, jedoch nicht in welchem Detailgrad, ist eine Umsetzung mit AWS Appsync von Vorteil. Auf Wunsch muss nur in der jeweiligen Query das gewünschte Feld angepasst werden. Außerdem wird einem der Aufwand zur Erstellung der Datenbank und des benötigten Resolvers übernommen, sofern man sich für DynamoDB entscheidet. 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.

3.4 Datenbanksystem

Es gibt zwei unterschiedliche Technologien für die Nutzung von Datenbanken, relationelle und nicht relationelle 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.

3.4.1 SQL: 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. Um nicht zulässige Einträge zu vermeiden benötigt eine Tabelle einen Primärschlüssel, der ein eindeutiges Zeilenmerkmal ist, etwa eine fortlaufende ID. Um Beziehungen zu anderen Tabellen zu erzeugen, wird ein ein Fremdschlüssel benötigt der eine Referenz auf den Primärschlüssel der anderen Tabelle darstellt. Damit Redundanzen in-

nerhalb einer Datenbank vermeiden werden, muss die Datenbank gemäß den Normalisierungsregeln bearbeitet werden. So wird gewährleistet, dass Daten nicht mehrfach existieren und bei Veränderung dieser Daten keine Anomalien[16] auftreten. Die Datenstrukturen sind voneinander abhängig. Eine Auswertung oder Veränderung der Daten ist 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 Credentials 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 per

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

Befehl sich anmelden kann. SSH-Zugang oder ähnliches ist bei RDS nicht vorgesehen.

Eine Hochverfügbarkeit ist möglich indem man den Punkt Multi-Availability Zone auswählt, und so eine Standby-Instanz in einer anderen Availability Zone bereit steht. Hierdurch steigt jedoch auch der Preis. Software-Patches können von dem Dienst automatisch eingespielt werden. Anders als 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. [Ama20f,]

3.4.2 NoSQL: 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.

[16] Unter Anomalien versteht man ein Fehlverhalten innerhalb von relationalen Datenbanken. Tabellen mit Spalten gleicher Bedeutung haben einen unterschiedlichen Inhalt. Eine Normalisierung verhindert Anomalien.

Relationale Datenbanken bieten nur eine vertikale Skalierung an, d.h. um die Performance zu erhöhen muss in der Regel ein leistungskräftigerer Server genutzt werden.

NoSQL Datenbanken unterstützen in der Regel keine ACID-Eigenschaften[17] sondern nutzen das BASE-Modell[18]. Da bei SQL-basierten Datenbanken lesende Abfragen solange warten bis schreibende Vorgänge beendet sind, bleibt die Konsistenz der Daten erhalten. No-SQL Datenbanken könnten im Zweifel unterschiedliche Daten zurückgeben, falls noch nicht alles ausgetauscht wurde. [Ale17,]

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 vorraus berechnet werden da der Dienst automatisch skaliert. Anwender haben die Option direkt Tabellen anzulegen ohne weitere Komponenten. Somit entfällt es auch sich um Software-Patches oder 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 sogut wie kein Wartungsaufwand, da Amazon die Verantwortung für die wichtigsten Aspekte übernimmt. [Ama20n,]

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“ [19] 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. [Ama20e,]

3.4.3 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

[17] ACID steht für Atomicity, Consistency, Isolation, Durability und bedeutet im Kern, dass alle Transaktionen konsistent sind.

[18] Base steht für Basically Available, Soft State, Eventually Consistent und stellt die Verfügbarkeit von Daten an eine höhere Stelle als die Konsistenz.

[19] 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.

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 ein 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 Awendung 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 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 auf ein Minimum zu halten.

3.5 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 einsehen zu können. Im besten Fall ist sogar es 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 unterstütze Dienste mit der selben 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.

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

3.5.1 AWS Cognito

Amazon Cognito ist ein vollintegrierter Dienst zur Benutzerverwaltung und Autorisierung. Cognito übernimmt dabei die Registrierung und Verwaltung neuer Benutzer sowie die Steuerung von Zugriffen. Mithilfe des AWS Amplify-Frameworks (siehe XXXX) kann die Benutzeroberfläche zum Registrieren, 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. 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 Möglichkeit „Hunderte von Millionen Benutzern“ zu verwalten, „ohne dass Server-Infrastruktur aufgestellt werden muss“ [Ama20c,]. [Ama20m,]

Cognito besteht aus den Komponenten Benutzerpools und Identitäten-Pools. Ein Benutzerpool ist ein Verzeichnis worüber Benutzern angelegt und verwalten werden können. Hier können Attribute konfiguriert werden, z.B. mit welcher Information sich Benutzer anmelden können. Zur Option stehen ein Username, die Email Adresse oder eine Telefonnummer. Außerdem ist es möglich weitere Attribute zu setzen, wie etwa das Geschlecht, der Wohnort, das Geburtsdatum und noch viele weitere. Eine MFA[20] Verifizierung ist ebenfalls direkt 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, beispielweise 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 genutzt. [Ama20r,]

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. Aber einer Anzahl von 100.000 wird der Preis niedriger gestaffelt. Außerdem können Kosten für die SMS-Nachrichten bei der MFA Authentifizierung anfallen. [Ama20d,]

3.5.2 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 im Client eine 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 man sich mit allen einzelnen Schritten intensiver befassen muss. Möchte man zusätzlich noch MFA oder eine sichere Passwortwiederherstellung einbauen muss man noch mehr Zeit einkalkulieren.

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 sowie direkt mit Amplify genutzt werden. Das Amplify-Framework bietet für die gängigen Javascript-Frameworks vorgefertigte Module an um noch weniger Konfigurationsaufwand betreiben zu müssen. Dank Cognito dürfe die Implementierung einer Authentifizierung in einer Serverless Anwendung keinen allzu großen Aufwand mehr benötigen.

[20] MFA steht für Multi-Factor Authentication und steigert die Sicherheit für Nutzern 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.

3.6 Backend Logik

Um überhaupt Daten erhalten zu können ist ein Backend-Prozess zum Verarbeiten von Anforderungen notwendig. 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. Im besten Fall 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.

3.6.1 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.“ [Ama20u,] 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 jedoch 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.

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. Ingsesamt gibt es unterschiedliche Varianten eine Lambda-Funktion zu starten. Bei der ersten Methode 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. [Ama20p,] 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[21]. Da es sich um asynchrone Aufrufe handelt warten die Dienste nicht auf eine erfolgreiche Antwort.

[21] SES steht für Simple Email Service und ist ein Dienst zum Versenden und Empfangen von Mails.

Hier erfahren die Dienste nur, dass Lambda das Ereignis in eine Warteschlange übergeben hat. Falls bei der Verarbeitung Fehler auftreten, kann die Funktion erneut ausgeführt werden. Alle Informationen zur Lambda-Funktionen, auch potenzielle Fehler im Code, werden automatisch protokolliert und in Amazon CloudWatch Logs gespeichert.[22] [Ama20q,]

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. Sobald 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 Funktion sicherzustellen, ist die Zustandslosigkeit somit essentiell. Dies bedingt ebenfalls, dass der Speicherplatz nicht persistent ist und auf 500MB eingeschränkt ist. [Ama20o,]

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[23] 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 3GB an Arbeitsspeicher und minimal 128MB. Die Größe des Arbeitsspeicher 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 Arbeitsspeicher 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. [Ama20k,]

3.6.2 Entscheidung

AWS Lambda eignet sich optimal zur Realisierung der Webanwendung, da direkt nach Erstellung der Lambda-Funktion und passender Berechtigungen am Code geschrieben werden kann. Dementsprechend muss keine Zeit für Themen wie Netzwerk, Virtualisierung oder selbst der 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

[22] Amazon CloudWatch Logs ist ein Überwachungs- und Management-Service, indem Logs von sämtlichen AWS-Diensten gespeichert und ausgewertet werden können.

[23] Durch das Least-Privilege-Prinzip wird sichergestellt, dass nur die minimal erforderlichen Berechtigungen zugeteilt werden.

Informationen über alle verfügbaren Accounts in der Organisation. Die Lambda-Funktion, welche in einem anderen Account bereitgestellt wird, benötigt Zugriff auf den Master-Account und muss über eine API die Liste der Accounts abrufen können. Anschließend müssen die gesammelten Daten in die DynamoDB-Tabelle gespeichert werden. Die DynamoDB-Tabelle befindet sich inklusive Lambda-Funktion und allen weiteren benötigten Ressourcen für die Webanwendung in einem separaten Account. Eine genauere Beschreibung der benötigten Berechtigungen und Accountstruktur befindet sich im Abschnitt (XXXX).

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. [Ama20l,]

3.7 Frontend Framework

Amplify bietet viele Frameworks an..

3.7.1 React

Es wurde React.

4 Implementierung

Architektur Bild:

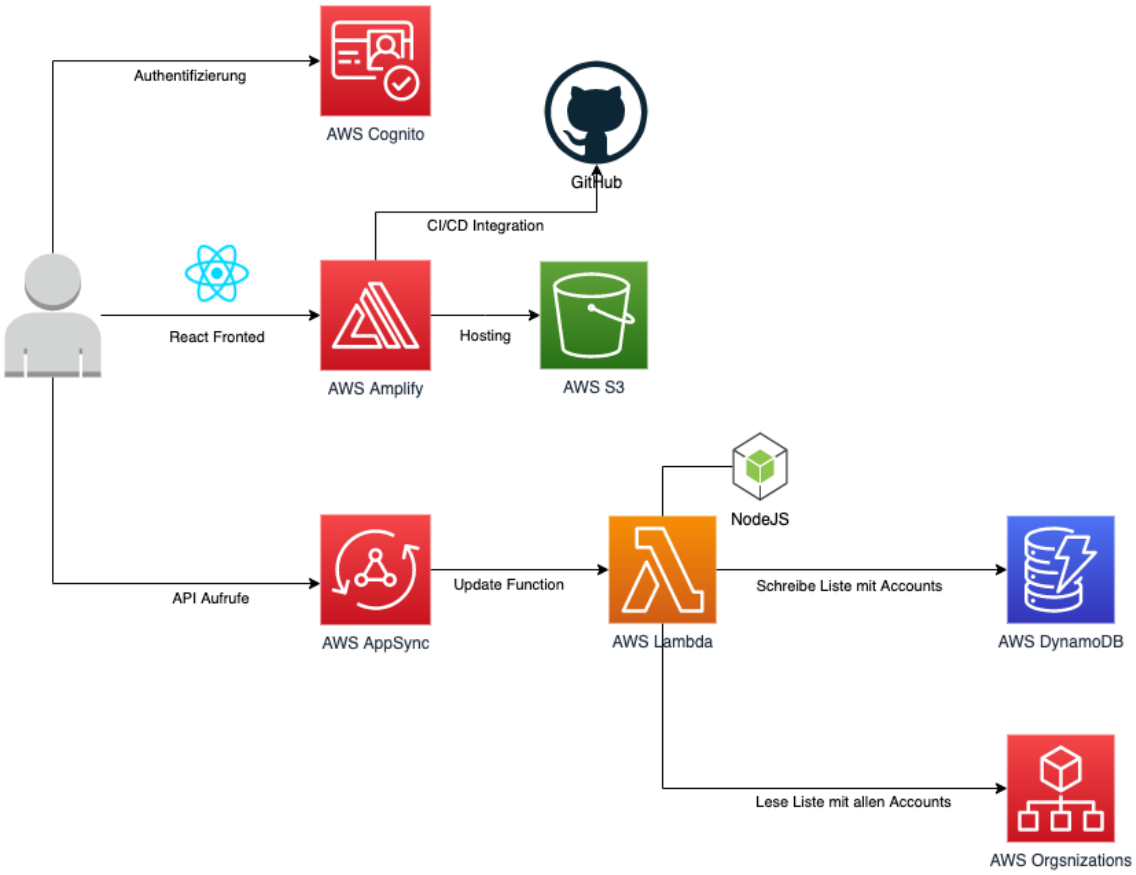

Abbildung 3: Übersicht der Architektur

Probleme mit Authentifizierung: https://medium.com/@zippicoder/setup-aws-cognito-user-pool-with-an-azure-ad-identity-provider-to-perform-single-sign-on-sso-7ff5aa36fc2a

Customize Auth: https://blog.kylegalbraith.com/2020/03/31/customizing-the-aws-amplify-authentication-ui-with-your-own-react-components/

Blöde Lösung: https://dunlop.geek.nz/aws-cognito-azure-ad-react-amplify/

5 Fazit

5.1 Zusammenfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

5.2 Ausblick

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

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, *Amazon api gateway*, https://aws.amazon.com/de/api-gateway/, 2020, [Abgerufen am 07.09.2020].

[Ama20b] _____, *Amazon api gateway preise*, https://aws.amazon.com/de/api-gateway/pricing/, 2020, [Abgerufen am 07.09.2020].

[Ama20c] _____, *Amazon cognito*, https://aws.amazon.com/de/cognito/, 2020, [Abgerufen am 07.09.2020].

[Ama20d] _____, *Amazon cognito-preise*, https://aws.amazon.com/de/cognito/pricing/, 2020, [Abgerufen am 07.09.2020].

[Ama20e] _____, *Amazon dynamodb*, https://aws.amazon.com/de/dynamodb/, 2020, [Abgerufen am 07.09.2020].

[Ama20f] _____, *Amazon relational database service (rds)*, https://aws.amazon.com/de/rds/, 2020, [Abgerufen am 07.09.2020].

[Ama20g] _____, *Aws amplify*, https://aws.amazon.com/de/amplify/, 2020, [Abgerufen am 07.09.2020].

[Ama20h] _____, *Aws amplify features*, https://aws.amazon.com/de/amplify/features/?nc1=h_ls, 2020, [Abgerufen am 07.09.2020].

[Ama20i] _____, *Aws appsync*, https://aws.amazon.com/de/appsync/, 2020, [Abgerufen am 07.09.2020].

[Ama20j] _____, *Aws appsync – preise*, https://aws.amazon.com/de/appsync/pricing/, 2020, [Abgerufen am 07.09.2020].

[Ama20k] _____, *Aws lambda – preise*, https://aws.amazon.com/de/lambda/pricing/, 2020, [Abgerufen am 07.09.2020].

[Ama20l] _____, *Class: Aws.organizations*, https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Organizations.html#listAccounts-property, 2020, [Abgerufen am 07.09.2020].

[Ama20m] _____, *Integration von amazon cognito in web- und mobile apps*, https://docs.aws.amazon.com/de_de/cognito/latest/developerguide/cognito-integrate-apps.html, 2020, [Abgerufen am 07.09.2020].

[Ama20n] _____, *Preise für on-demand-kapazität*, https://aws.amazon.com/de/dynamodb/pricing/on-demand/, 2020, [Abgerufen am 07.09.2020].

[Ama20o] _____, *Serverless architectures with aws lambda*, https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf, 2020, [Abgerufen am 07.09.2020].

[Ama20p] _____, *Verwenden von aws lambda mit amazon dynamodb*, https://docs.aws.amazon.com/de_de/lambda/latest/dg/with-ddb.html, 2020, [Abgerufen am 07.09.2020].

[Ama20q] _____, *Verwenden von aws lambda mit sonstigen services*, https://docs.aws.amazon.com/de_de/lambda/latest/dg/lambda-services.html, 2020, [Abgerufen am 07.09.2020].

[Ama20r] _____, *Was ist amazon cognito?*, https://docs.aws.amazon.com/de_de/cognito/latest/developerguide/what-is-amazon-cognito.html, 2020, [Abgerufen am 07.09.2020].

[Ama20s] _____, *Was ist amazon elastic container service?*, https://docs.aws.amazon.com/de_de/AmazonECS/latest/developerguide/Welcome.html, 2020, [Abgerufen am 07.09.2020].

[Ama20t] _____, *Was ist aws?*, https://aws.amazon.com/de/what-is-aws/?nc1=f_cc, 2020, [Abgerufen am 07.09.2020].

[Ama20u] _____, *Was ist aws lambda?*, https://docs.aws.amazon.com/de_de/lambda/latest/dg/welcome.html, 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].

[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].

[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].

[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/, 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].

Anhang

A Anhang Teil 1

Listing 1: Formularmanipulation

```
$( document ).ready(function() {


   if ( $('#TimeUnits').length )
   {
      $('#TimeUnits').autocomplete({ disabled: true });
   }


   var ITPROLABCurrentAction = $.getUrlVar('Action');

   if ( ITPROLABCurrentAction == 'CustomerTicketMessage' )
   {
      var ITPROLABQueue   = $( "#Dest option:selected" ).text();
      alert( ITPROLABQueue );

      if (ITPROLABQueue != 'ITPROLAB' ) {
         $('#ITPROLABServiceID').fadeOut();
      }
   }



$("<style>")
   .prop("type", "text/css")
   .html("\
   [title=\"10-blue\"] {\
      background-color: blue;\
      color: blue;\
      font-size: 1px;\
      opacity: 0.7;\
   }")
   .appendTo("head");
```

B Anhang Teil 2

Listing 2: Formularmanipulation

```
#!/usr/bin/perl
##
```

```
# generate PDF from Latex Template LK.tex --> lk.tex --> lk.pdf
# wird von machform formular aufgerufen.
##

use CGI qw/:standard/;
use DBI;

my $debug = 0;



my $template;
my $german = "LK/LK-D.tex";
my $english = "LK/LK-E.tex";
my $arabic = "LK/LK-A.tex";
my $lk    = "LK/lk.tex";



my $message = "<pre>\n";
my $dbm = DBI->connect (
   ↪ "dbi:mysql:$database:$server",$userid,$passwd ) || die "Could
   ↪ not connect to $database";


$stm = $dbm->prepare ( qq{ select id,element_1, element_2,
   ↪ element_5, element_6, element_7, element_8,
   ↪ element_10_1,element_12 from $table ORDER by id DESC LIMIT 1
   ↪ });
$stm->execute;
$stm->bind_columns(\$id,\$e1,\$e2,\$e5,\$e6,\$e7,\$e8,\$e10,\$e12);
if ($stm->fetch) {

   if ($e5 =~ /-/) {
      $e5 = substr ($e5,8,2) . "." . substr ($e5,5,2) . "." . substr
         ↪ ($e5,0,4);
   }
   $message .= " ID = $e1\n";
   $message .= " Name = $e2\n";
   $message .= " Datum = $e5\n";
   $message .= " Adresse= $e6\n";
   $message .= " E = $e7\n";
   $message .= " K = $e8\n";
   $message .= " data = $e10\n";
   $message .= " lang = $e12\n";

}
else {
   $message .= " nicht gefunden\n";
}
```

```
printf "$message\n" if ($debug);

undef $stm;
undef $dbm;
```

C Anhang Lambda

Listing 3: Lambda-Code

```
   /* 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 S3 service object
const s3 = new AWS.S3({apiVersion: '2006-03-01'});


const s3listfunction = new Promise((resolve, reject) => {
  let s3buckets_data = s3.listBuckets().promise();
  if(s3buckets_data ) {
    resolve("Works" + s3buckets_data);
  }
  else {
    reject("NOPE")
  }
  //return s3.listBuckets().promise()
});

const s3listfunction_1 = () => {

  return s3.listBuckets().promise()
};

var params = {
    TableName: 'Comments-ifdtxan4k5fglip5ynnopk6bky-dev',
 //  Key: {'id': '4de8a026-4599-465a-ac76-9259b4adf1fc'}
   };


   async function getAllDynamoDBItems(){
      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);
    }
}



exports.handler = async (event) => {

    //getAllDynamoDBItems()
    //console.log( "S3 Ausgabe: " + s3.listBuckets().promise() )

    //const msg = await getAllDynamoDBItems()
    //const msg1 = await s3listfunction_1()
    //console.log(msg1)

    s3listfunction_1.then((res) => console.log(res), (err) =>
        ↪ alert(err));

    // TODO implement
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    //return s3.listBuckets().promise();
    return response
};




function getCoins(callback) {
    console.log("Function starts")
  docClient.scan(params, function(err, data) {
    if (err) {
      callback(err)
      console.log("IFFFF")
      console.log(err)
    } else {
        console.log("ELLSEE")
        console.log(data.Items)
      callback(null, data.Items)
    }
```

```
  });
}
```