Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jules Fehr

Serverlose Architektur:
Function as a Service mit AWS Lambda am Beispiel einer Web-Anwendung für Fotografen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jules Fehr

Serverlose Architektur: Function as a Service mit AWS Lambda am Beispiel einer Web-Anwendung für Fotografen

Abschlussarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ulrike Steffens
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Abgegeben am 19.07.2018

Jules Fehr

Thema der Arbeit

Serverlose Architektur: Function as a Service mit AWS Lambda Beispiel einer Web-Anwendung für Fotografen

Stichworte

FaaS, Serverless, AWS Lambda, Web-Anwendung

Kurzzusammenfassung

Das Ziel dieser Bachelorarbeit ist die Konzeption und Entwicklung einer Function as a Service Web-Anwendung für Fotografen. In dieser Arbeit werden die Prinzipien von serverloser Architektur behandelt. Es wird eine Referenzarchitektur vorgestellt und es wird eine Anforderungsanalyse für die Anwendung erstellt. Basierend auf der Analyse und den Prinzipien der serverlosen Architektur wird die Umsetzung wichtiger Komponenten erläutert.

Jules Fehr

Title of the paper

Serverless Architecture: Function as a Service with AWS Lambda on the example of a web application for photographers

Keywords

FaaS, Serverless, AWS Lambda, Web-application

Abstract

The goal of this bachelor thesis is the conception and development of a Function as a Service web-application for photographers. The principles of serverless architecture will be discussed. A reference architecture will be presented and a requirement analysis will be made. The implementation of important components will be made based on the analysis and the principles of serverless architecture.

Inhaltsverzeichnis

Glossar

Accessibility	Barrierefreie Benutzeroberfläche
Amazon S3	Filehosting-Dienst
API	Programmierschnittstelle
Auth0	Authentifizierungs-Dienst
AWS	Amazon Web Services
BaaS	Backend as a Service
Big Data	Große/Komplexe Datenmengen
Backend	Gegenstück des Frontend, näher am System
Computing	Zielorientierte Tätigkeiten an Computern
Deployment	Softwareverteilung
FaaS	Function as a Service
Firebase	Entwicklungs-Plattform (Mobile & Web)
Frontend	Gegenstück des Backend, näher am User
HTTP-Request	Anfrage per HTTP durch den Client
IAM	Verwaltung des Zugriffs auf AWS-Ressourcen
Lambda-Architektur	Big-Data-Architektur
Logging	Erstellung eines Softwareprozess-Protokolls
Microservice	Architekturmuster mit modularem Aufbau
Netlify	Hosting-Dienst für statische HTML-Seiten
PaaS	Platform as a Service
Runtime	Laufzeitumgebung
JWT	JSON Web Token
SNS	Simple Notification Service
SPA	Single-Page Application
UI	User Interface

1 Einleitung

1.1 Motivation

Ein professioneller Fotograf muss heutzutage nicht nur Fotos schießen, geschossene Fotos müssen auch mit dem Kunden geteilt werden. Hinzu kommt, dass im Voraus oft mit den Kunden abgesprochen wird, wie viele Fotos in der Postproduktion bearbeitet werden, da dies viele weitere Stunden Arbeit für den Fotografen bedeutet. Zu diesem Zeitpunkt entsteht eine Problematik, weil der Kunde sich einerseits Fotos für Korrektur von Licht, Haut und Farben aussuchen muss, andererseits möchte ein Fotograf ungern die Originale weitergeben, um das Ausbleiben der Bezahlung zu vermeiden.

Hier soll die, im Zusammenhang mit dieser Arbeit realisierte, Web-Anwendung Abhilfe schaffen. Der Fotograf soll in der Anwendung lediglich die Fotos hochladen müssen, welche dann automatisch per Wasserzeichen und Qualitätsreduktion gegen Diebstahl abgesichert werden. Nachdem die Fotos für die Vorauswahl vorbereitet und im Anschluss hochgeladen wurden, soll der Kunde sie ansehen können, um seine endgültige Auswahl zu treffen.

Die Idee und die Anforderungen entstammen aus dem Gespräch mit befreundeten Fotografen und den darauffolgenden Überlegungen, was sich sinnvoll umsetzen lässt. Nichtsdestotrotz werden im Zusammenhang mit dieser Arbeit nicht alle Anforderungen umgesetzt werden können. Ziel dieser Arbeit ist das Erschaffen eines ersten Prototyps.

Die Entscheidung, eine serverlose Anwendung zu erschaffen, kam hauptsächlich aus dem Interesse an der neuen Technologie und den Möglichkeiten, die im Amazon Web Services (AWS) Ökosystem geboten werden. Für viele Anforderungen gibt es hier bereits Dienste, darüber hinaus eine gut funktionierende Anbindung von Drittanbieter-Diensten. Außerdem ist AWS Lambda der am längsten bestehende Cloud Computing Dienst und gilt als sehr ausgereift.

1.2 Überblick

In dieser Arbeit werden zu Anfang einige Grundlagen vermittelt, wie die Bedeutung des Begriffs Function as a Service. Es werden außerdem die Prinzipien von serverloser Architektur und ein Modell für eine serverlose Architektur erklärt. Dann wird ein kurzer Vergleich vom Begriff Serverless zum Begriff Microservices gezogen, um Gemeinsamkeiten und Unterschiede zu verstehen. Das Wissen wird notwendig sein, um zu erkennen, was aus diesen Grundlagen in die Konzeption eingeflossen ist.

Auf die Grundlagen folgt die Anforderungsanalyse der Anwendung. Zusammen mit den Inhalten der Grundlagen wird daraus die Basis der Konzeption entwickelt.
In der Konzeption wird die Architektur der Anwendung zusammen mit der Referenzarchitektur vorgestellt. Weiterhin werden in der Konzeption einige wichtige Komponenten erläutert. Zudem wird eine grobe Kostenanalyse durchgeführt, um auch die wirtschaftlichen Aspekte einzubeziehen, die bei einem Produktionssystem an Bedeutung gewinnen.
Zu guter Letzt erfolgt eine Evaluierung des Ergebnisses mit einem Ausblick auf mögliche Erweiterungen der Anwendung.

Der Prototyp wird während der Entwicklung den vorläufigen Namen Prevelp tragen.

2 Grundlagen

In diesem Abschnitt werden die für das Verstehen der Konzeption und Realisierung benötigten Grundlagen vermittelt.

2.1 Function as a Service

Function as a Service, entstammt ursprünglich einer Bewegung, die damit begann, dass Entwickler nicht mehr eigene Server für ihr Backend aufsetzen und überwachen wollten. Aus diesem Bedürfnis entstanden Backend-as-a-Service-Dienste (BaaS). Dieses Konzept wurde von Amazon bis zur Ankündigung von AWS Lambda in 2014 weiterentwickelt. Mittlerweile ist Amazon aber nicht mehr der einzige FaaS-Anbieter, sie teilen sich den Markt heute z.B. mit Google Cloud Functions, Microsoft Azure Functions, IBM OpenWhisk.
Während bei BaaS ein ewiger Prozess auf einem Server läuft, der auf HTTP-Requests oder API-Calls wartet, löst bei FaaS erst ein Ereignis (Event) die Ausführung einer Funktion auf den AWS Servern aus.[1]
Von dem ebenfalls verwandten Platform-as-a-Service-Konzept (PaaS) unterscheidet sich FaaS hauptsächlich durch die genauere Skalierung, die es FaaS-Diensten ermöglicht, Kosten einzusparen. Auch wenn in einer PaaS-Anwendung eine automatische Skalierung theoretisch möglich ist, ist es trotzdem kaum möglich, genau so präzise zu skalieren, da nur bei FaaS die Kosten in individuellen Requests berechnet werden.[2]
Dies bedeutet nicht, dass BaaS und PaaS keinen Mehrwert liefern, sie können lediglich das Potential der Cloud nicht in dem Ausmaß nutzen, wie es mit FaaS möglich ist.

Das Akronym FaaS wird oft genutzt, um Technologien wie AWS Lambda zu beschreiben, während der Begriff serverlos oder serverless in dieser Arbeit als Begriff verwendet wird, der mehr als nur FaaS umfasst, nämlich auch die allgemeine Bereitschaft, Drittanbieter-Services und APIs zu nutzen.

[1] [Avram 2016]
[2] [Roberts 2016]

2.2 Prinzipien serverloser Architektur

Der im Zusammenhang mit dieser Arbeit entwickelten Anwendung liegen fünf Prinzipien zu Grunde, die ein ideales serverloses System beschreiben und als Richtlinien bei der Konzeption dienen sollen. Diese Prinzipien entstammen dem Buch „Serverless Architectures on AWS“ von Peter Sbarski.[3]

I. Nutzung eines Computing-Dienstes anstelle eines Servers.

In serverlosen Anwendungen werden granulare Funktionen verwendet, die isoliert und unabhängig von anderen Funktionen in einem Computing-Dienst wie AWS Lambda ausgeführt werden. Funktionen haben in der Regel nur eine Aufgabe. Komplexere Tätigkeiten werden durch Verkettung von Funktionen in einer Pipeline realisiert.

II. Schreiben von zustandslosen Funktionen mit möglichst nur einer Aufgabe.

Gemäß dem Single Responsibility Principle soll jede Funktion nur eine einzige Aufgabe haben. Dadurch ist sie leichter zu testen und kann eventuell an anderer Stelle erneut verwendet werden. Außerdem können Fehler leichter aufgespürt werden. Funktionen sollen zudem zustandslos sein und nicht davon ausgehen, dass Prozesse oder lokale Ressourcen außerhalb der gerade laufenden Session verfügbar sind. Dies wird gefordert, damit die Plattform schnell hoch- und runterskalieren kann.

III. Entwurf von ereignisgesteuerten Pipelines.

Die flexibelsten und mächtigsten Dienste sind event-driven bzw. ereignisgesteuert. Es wird in diesem Fall also durch ein Ereignis (Event) eine Funktion ausgelöst. Dies reduziert die Kosten und Komplexität der Anwendung, da nicht nachgesehen werden muss, ob es z.B. eine Änderung gab (Stichwort: pull). Stattdessen meldet sich die Änderung als Ereignis selber (Stichwort: push).

[3] [Sbarski 2017], S. 9-12.

IV. Bauen von einem mächtigeren Frontend.

Funktionen in AWS Lambda sollten schnell ausgeführt werden, da die Kosten von der Ausführungsdauer, der Zahl der Anfragen und dem verwendeten Speicher abhängig sind. Folglich sinken die Kosten bei einer kürzeren Ausführungsdauer und weniger Anfragen auf Lambda. Aus diesem Grund sollte ein mächtigeres Frontend gebaut werden, das direkt mit den Drittanbieter-Diensten kommunizieren kann. Dies hat gleichzeitig den praktischen Nebeneffekt von weniger Latenz und einer daraus resultierenden verbesserten User Experience.

V. Nutzung von Drittanbieter-Diensten.

Die Verwendung von getesteten Drittanbieter-Diensten, die einen Mehrwert bringen, bedeutet weniger Programmierarbeit für den Ersteller der serverlosen Anwendung. Die Konfiguration eines etablierten Dienstes ist in den meisten Fällen aufgrund der Zeitersparnis die zu bevorzugende Variante dazu, den Dienst selber zu schreiben. Wichtige Faktoren sind Preis, Kapazitäten, Umfang und Support, sowie die Qualität des Drittanbieter-Dienst.

2.3 Compute as Backend

An dieser Stelle soll die Grundlage für die benutzte Referenzarchitektur, und damit die Grundlage für die Architektur des Prototyps, vorgestellt werden.
AWS Lambda und Drittanbieter-Dienste bilden bei dieser Architektur das Backend für eine Web-, Mobil- oder Desktop-Anwendung. Im Backend wird im Grunde getan, was im Frontend aus Sicherheitsgründen nicht getan werden kann. In Abbildung 1 kommuniziert der Client direkt mit den Diensten, wie z.B. Auth0 als „Authentication Service“, oder mit Amazon S3, um Dateien hochzuladen. Über das API Gateway kann mit dem Backend kommuniziert werden. Im Backend finden sich die Lambda-Funktionen, die jeweils eine dedizierte Aufgabe haben.[4] Zu beachten ist, dass ein Logging-Dienst für das Troubleshooting benötigt wird, da beim Client nur generische Error-Codes ankommen, die das eigentliche Problem im Backend meistens nur ungenügend beschreiben.

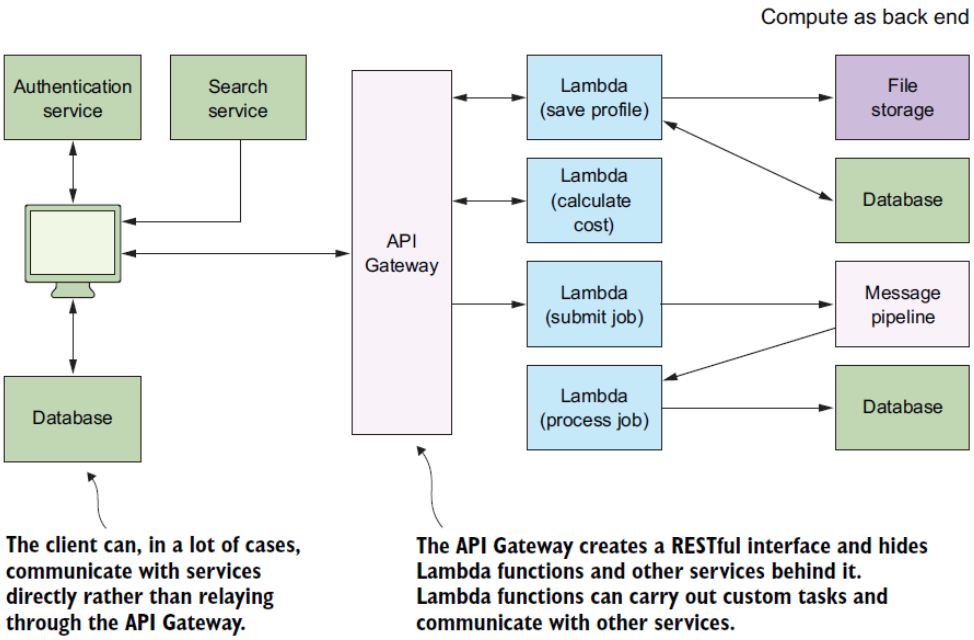

Abbildung 1: Compute as backend (Quelle: Sbarski, 2017, S. 20)

[4] [Sbarski 2017], S. 19-24.

2.4 Serverless vs. Microservice

Serverless kann als der nächste Schritt der Software-Architektur nach Microservices verstanden werden. Serverless bietet gegenüber Microservices ein ähnliches und einfaches Deployment, da beide Verfahren Container benutzen. Anders als bei Microservices konfiguriert ein serverloser Dienst wie AWS Lambda den benötigten Container automatisch. Es könnte argumentiert werden, dass dadurch die Konfiguration von serverlosen Funktionen, wie z.B. mit AWS Lambda, ein wenig einfacher ist, als eine vergleichbare Konfiguration von Containern für Microservices. Gleichzeitig gibt man einiges an Flexibilität ab, da man die Konfiguration von serverlosen Diensten nur begrenzt beeinflussen kann. Abgesehen davon bietet AWS Lambda eine bessere und schnellere Skalierbarkeit und eine höhere Kosteneffizienz. Gleichzeitig ist man durch die Auswahl eines einzelnen Anbieters von serverless Diensten wie z.B. AWS von diesem Anbieter abhängig und muss eventuell eine komplexe Anwendung migrieren, wenn der Anbieter das Angebot einstellt.[5]
Dies könnte vor allem für Systeme, die eine lange Zeit, wie z.B. 10 Jahre, in dieser Form laufen, ein Problem sein. Die Hoffnung bei einem Anbieter wie AWS ist jedoch, dass sie ihrer Größe wegen das Angebot für eine lange Zeit anbieten werden.
Zusätzlich können Firmen, die ihre Kundendaten aus Gründen des Datenschutzes nur auf eigenen Servern speichern, serverlose Architektur nicht in Anspruch nehmen. Gerade nach dem Bekanntwerden der Spionage durch die NSA, ist das Vertrauen in die, meist in den USA ansässigen, serverlosen Anbieter eventuell nicht mehr verschwunden, so dass hier auf jeden Fall die Microservices bevorzugt werden würden.
Beide Architekturstile reduzieren aber die Zeit, bis eine Anwendung veröffentlicht werden kann. Sie erlauben eine einfachere Anpassung an neue oder geänderte Anforderungen und sie versuchen die laufenden Kosten nach der Veröffentlichung zu senken.[6] [7]
Es muss also weiterhin für jedes Projekt unter Berücksichtigung der Vor- und Nachteile analysiert werden, welcher Architekturstil verwendet wird.

[5] [Benetis 2017]
[6] [Wolf a]
[7] [Wolf b]

2.5 AWS Lambda

Ein Computing-Dienst bildet das Rückgrat einer serverlosen Architektur. AWS Lambda ist ein solcher Computing-Dienst. Das bedeutet, AWS Lambda ist im Grunde dafür verantwortlich, den bereitgestellten Code auf Kommando auszuführen und gibt das Ergebnis an andere Dienste weiter.

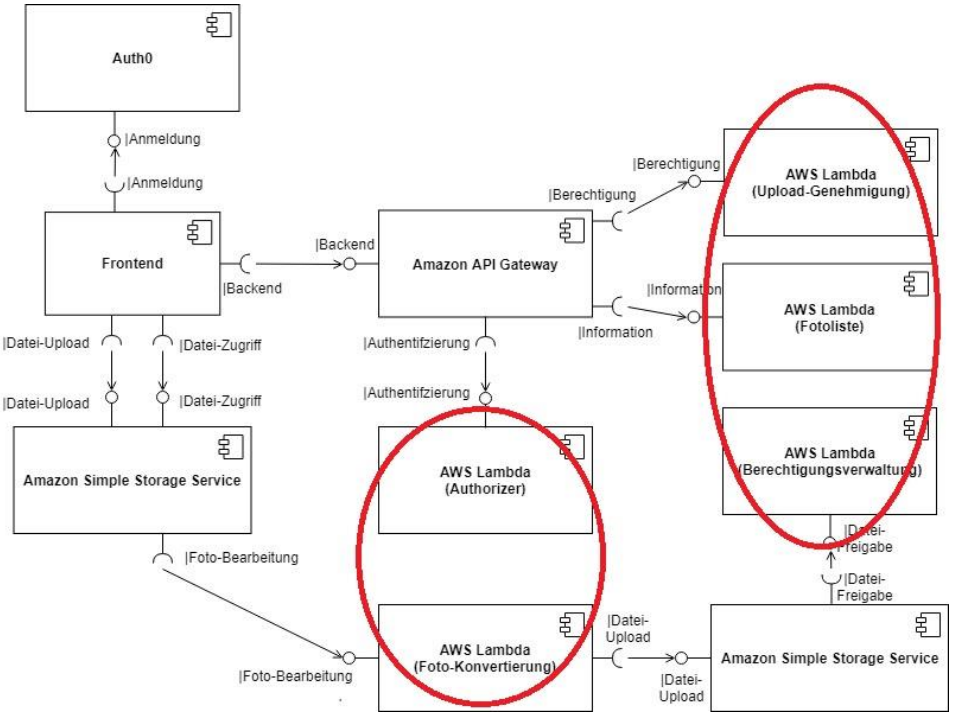

Abbildung 2: serverlose Architektur (AWS Lambda herausgehoben)

Bei AWS Lambda-Funktionen muss nur gezahlt werden, was auch genutzt wird.[8] Die Kosten berechnen sich aus den Anforderungen und deren Dauer. Eine Funktion wird als angefordert gezählt, wenn sie durch ein Ereignis oder einen direkten Aufruf gestartet wurde. Die Dauer wird vom Beginn der Ausführung bis zu deren Ende gerechnet und dann gerundet. Der endgültige Preis für die Dauer berechnet sich allerdings abhängig vom verwendeten Arbeitsspeicher. Zusätzlich werden die ersten 1 000 000 Anforderungen und 400 000 GB/s an Datenverarbeitungszeit nicht in Rechnung gestellt. Eine ausführlichere Berechnung der Kosten mit geschätzten Praxisbeispielen wird im Konzeptionssegment unter Punkt 4.6 vorgestellt.
Schlussendlich sei noch zu erwähnen, dass AWS Lambda außer dem Namen nichts mit Big Data und der damit zusammenhängenden Lambda-Architektur zu tun hat.[9]

[8] [AWS h]
[9] [Kinley 2013]

3 Anforderungen

Nach der Vorstellung der Grundlagen, folgen nun die Anforderungen. Hierbei handelt es sich um die funktionalen Anforderungen, die im Gespräch mit Fotografen entstanden sind. Die nicht-funktionalen Anforderungen wurden nach eigenem Ermessen aufgenommen und sollen bei der Auswahl eines FaaS-Anbieter helfen, der all diese Eigenschaften besser als vergleichbare Anbieter erfüllt.

Aus den Anforderungen wird ein erster Prototyp entstehen, der die Grundfunktionalität – bestehend aus Fotoupload, Bearbeitung der Fotos und Präsentation der Fotos – erfüllt. Dieser Prototyp wird im weiteren Verlauf, in einem evolutionären Prototyping, nach und nach erweitert werden.[10] Durch das evolutionäre Prototyping lässt sich das Risiko minimieren, am Bedürfnis des Kunden vorbei zu entwickeln.

[10] [KrBo 2016]

3.1 Grundfunktionalität

Funktionale Anforderungen

Die Benutzer der Anwendung sind Fotografen auf der einen Seite und die Kunden der Fotografen auf der anderen Seite. Die Fotografen sollen die Möglichkeit haben Bilder hochzuladen. Die Kunden sollen Zugriff auf die Bilder bekommen, dies aber erst nach dem Hinzufügen des Wasserzeichens und der Qualitätsreduzierung. Im ersten Prototyp wird auf die Ausprägung unterschiedlicher Rollen für Fotograf und Kunde verzichtet.

Weiterhin gehört zur Prototypen eine Funktionalität, mit der die Bilder, nach dem Upload und der Bearbeitung durch das System, vom Kunden angesehen werden können. Vorerst wird an dieser Stelle noch keine eindeutige Kundenzuordnung der Bilder benötigt, da es noch keine Kundenrolle gibt. In weiteren Entwicklungsphasen wird der Kunde aber eine Funktionalität benötigen, mit der er eine Auswahl an Bildern treffen kann, die der Fotograf in der Postproduktion weiterbearbeiten soll.
Auf User Interface und User Experience wird im ersten Schritt der Prototypisierung ebenfalls keine Rücksicht genommen, da das serverlose Backend und die daraus entstehenden Möglichkeiten im Fokus dieser Arbeit stehen.

Zusammenfassung der fachlichen Anforderungen:

I. **Upload von Bildern in der Anwendung durch den Fotografen.**

II. **Der Kunde soll die mit Wasserzeichen versehenen Bilder ansehen können.**

III. **Die Bilder sollen im Frontend der Anwendung vom Kunden betrachtet werden können.**

Nicht-funktionale Anforderungen

Die technischen Anforderungen sind in erster Linie die Zuverlässigkeit des Systems, die Skalierbarkeit, sowie Leistung und Effizienz.

Das System soll 24 Stunden am Tag und 365 Tage im Jahr verfügbar sein. Bei gleichzeitigem Fotoupload aller Fotografen (eine Anzahl der Fotografen findet sich im Rechnungsbeispiel im Punkt 4.6, das System soll aber eine beliebige Anzahl an Fotografen bedienen können), soll das ganze System sofort skalieren können, so dass die Verarbeitung der Fotos mehr Fotografen nicht länger als bei einem einzigen Fotografen dauert. Die Skalierung und Verfügbarkeit des Systems soll gleichzeitig wirtschaftlich sein. Wirtschaftlichkeit bedeutet in diesem Zusammenhang, dass die Nutzung nicht mehr als bei anderen äquivalenten Diensten kostet, und dass bei Nichtnutzung keine Kosten anfallen. Eine weitere Forderung ist Systemreife, die sich in diesem Fall durch eine detaillierte Dokumentation und eine Plattform ohne Ausfälle und größere Probleme in der Vergangenheit auszeichnet. Zusätzlich dürfen die Fotos nur von autorisierten Personen gesehen werden.

Zusammenfassung der nicht-funktionalen Anforderungen:

I. **24/7 Verfügbarkeit der Anwendung.**

II. **Skalierung ohne Zeitverzögerung und ohne Limit.**

III. **Keine Betriebskosten bei Nichtnutzung der Anwendung.**

IV. **Systemreife, gekennzeichnet durch detaillierte Dokumentation und eine stabile Plattform.**

V. **Nur autorisierte Personen dürfen die Bilder sehen.**

Besonders Anforderung II und III sind in diesem Fall wichtig, da hier die Unterschiede zu traditionellen Client-Server-Architekturen hervorstechen. Anforderung I und IV sind dabei aber auch nicht zu vernachlässigen, da ohne die Erfüllung dieser Anforderungen argumentiert werden könnte, dass zwar Vorteile aus II und III entstehen, das System aber traditionellen Systemen ansonsten nicht gewachsen ist.

3.2 Erweiterungen

Erweiterungen für den Prototyp sind die, in der Grundfunktionalität erwähnte, Trennung zwischen Fotograf und Kunden des Fotografen, sowie die Möglichkeit des Kunden, eine Auswahl an Bildern zu treffen und diese Auswahl dem Fotografen zukommen zu lassen. Ebenso sollte auf User Interface, User Experience und Accessibility Rücksicht genommen werden.

Abgesehen von diesen grundlegenden Funktionen wäre es für den Einsatz im alltäglichen Geschäft notwendig, den Prototypen um die Möglichkeit zu erweitern, dass mehrere Kunden vom Fotografen angelegt werden können. Diese Kunden dürfen dann natürlich aus Gründen des Schutzes ihrer Privatsphäre nur Zugriff auf ihre eigenen Fotos bekommen. Der Fotograf nimmt also auch eine administrative Aufgabe ein, während der Kunde ein reiner Nutzer der Anwendung ist. Mit der Anlage von Kunden kommt auch das Löschen und Ändern von Kunden hinzu, was auch ein Löschen der mit dem Kunden assoziierten Fotos mit sich bringen sollte, um die laufenden Kosten für den Fotografen zu senken. Für diese Funktionalität müssen Create, Read, Update, Delete (CRUD) Operationen implementiert werden. Dies sollte über eine nur für den Administrator zugängliche Benutzeroberfläche geschehen.

An dieser Stelle ist es ebenfalls eine Überlegung wert, ob es in einer späteren Version ein Archiv geben sollte, um eine mit dem Kunden assoziierte Sicherheitskopie der Fotos in der Cloud zu haben. Dies widerspricht zwar der vorherigen Idee die Kunden und ihre Fotos nach einer gewissen Zeit zu löschen, ist jedoch bei geringen Kosten eine Möglichkeit, ein immer zugängliches Archiv zu haben. Die Höhe der Kosten kann durch eine Kostenanalyse ermittelt werden, um eine begründete Entscheidung zwischen den beiden Optionen zu treffen.

Im Zusammenhang mit der Markierung von den Bildern, die mit einem Wasserzeichen versehen wurden, wäre eine Markierung der Bilder auf Seite des Fotografen ebenfalls sinnvoll. Das bedeutet, dass die Web-Anwendung bei Markierung von Bildern durch den Kunden eine Markierung der Originalbilder vornimmt. Dann könnte sich der Fotograf die von ihm hochgeladenen und durch den Kunden markierten Bilder runterladen.

Damit die Fotografen ihre Bilder möglichst schnell hochladen können, sollte es die Möglichkeit geben, komprimierte Ordner hochzuladen.

Auch auf die Aspekte Datenschutz und Datensicherheit muss in späteren Iterationen verstärkt Rücksicht genommen werden, da gestohlene Bilder für eine derartige Plattform den kommerziellen Untergang bedeuten würden.

4 Konzeption

Im Anschluss an die fachlichen Anforderungen im vorherigen Abschnitt, wird nun die Konzeption der Web-Anwendung betrachtet. In den Grundlagen wurde bereits erwähnt, dass sich eine Anwendung aus dem Bereich Serverless von traditionellen Web-Anwendungen unterscheidet. Diese Unterschiede finden sich allerdings eher in der Perspektive des Entwicklers, während Fotografen und Kunden von den Unterschieden nichts mitbekommen werden.
Durch die Verwendung einer serverlosen Lösung soll eine permanente Verfügbarkeit bei größtmöglicher Kosteneffizienz gewährleistet werden (nicht-funktionale Anforderungen I und III). Zudem können neue technische Anforderungen, wie ein Ansteigen der Requests (z.B. durch mehr Kunden und dadurch mehr gleichzeitig hochgeladene Fotos), dank des Betriebs in der Cloud problemlos bedient werden (nicht-funktionale Anforderung II). Die Verwendung von Tokens zur Autorisierung, soll gewährleisten, dass nur autorisierte Personen Zugriff auf die Bilder haben (nicht-funktionale Anforderung V).

4.1 Architektur

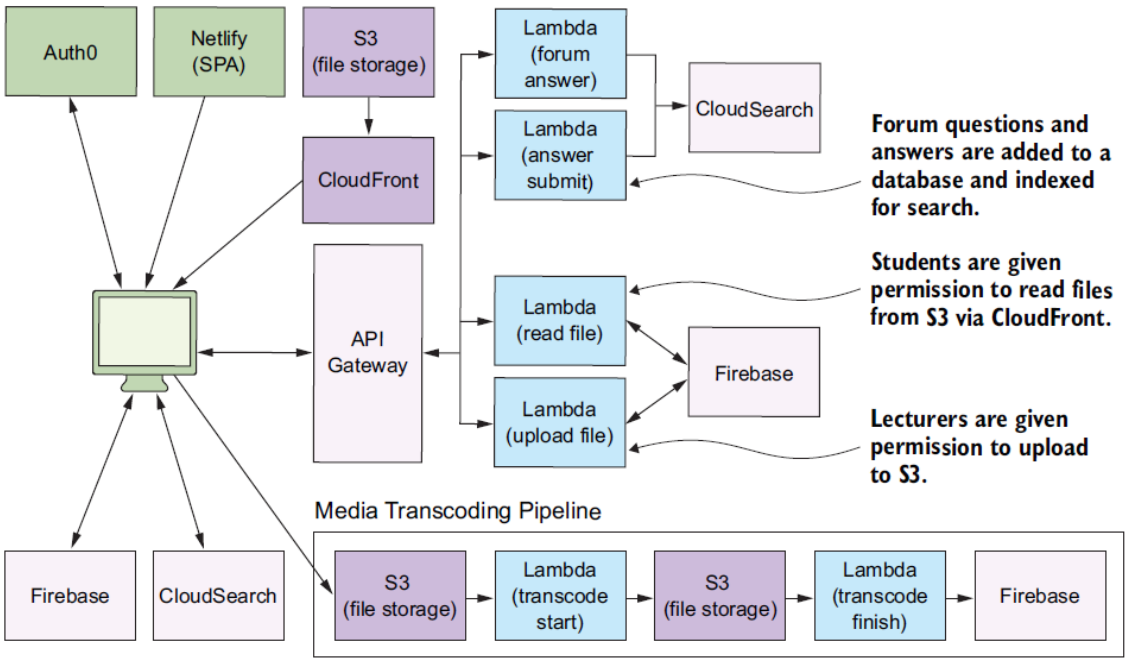

Abbildung 3: Referenzarchitektur (Quelle: Sbarski, 2017, S. 21)

Die Architektur stellt die Grundlage für eine Web-Anwendung dar. Als Referenzarchitektur dient der von A Cloud Guru (https://acloud.guru/) verwendete Aufbau. Es handelt sich hierbei um die Compute-as-Backend-Architektur aus Punkt 2.3. Die Referenzarchitektur wird im Folgenden wie in der Quelle[11] beschrieben und, um Anpassungen erweitert, für die eigene Architektur umgesetzt. Zusätzlich werden einige in der Referenzarchitektur enthaltene Komponenten im ersten Prototyp nicht verwendet werden, um den Rahmen nicht zu sprengen.

Das Frontend wird in der Referenzarchitektur von Netlify gehostet. In der Architektur für diese Arbeit wird das Frontend auf Amazon S3 gehostet werden, um das AWS Ökosystem nicht unnötig zu verlassen. Möchte man aber, dass die Seite weltweit schnell zu erreichen ist, sollte man Amazon S3 vermeiden. Die S3 Buckets sind immer spezifisch für eine Region, was Zugriffe aus anderen Teilen der Welt langsamer macht.[12] Im Prototyp kann dieser Umstand vernachlässigt werden.

Auth0 wird für die Registrierung und Authentifizierung genutzt. Durch die von Auth0 bereitgestellten ID Tokens kann das Frontend sicher und vor allem direkt mit

[11] [Sbarski 2017], S. 21-22.
[12] [Decuyper 2017]

Drittanbieter-Diensten kommunizieren.[13] Dies geht einher mit Prinzip V serverloser Architekturen.

Das von der Architekturvorlage vorhergesehene Firebase ist eine Echtzeit-NoSQL-Datenbank von Google, die es möglich macht, Updates sofort zu bekommen, ohne vorher anzufragen, ob sich etwas geändert hat.[14] Firebase wird zumindest im ersten Prototyp nicht genutzt werden, da Echtzeit-Updates im ersten Schritt eine geringere Priorität haben.

Bei A Cloud Guru werden Videos von Lehrenden hochgeladen. Diese Videos werden unsichtbar für den Nutzer hochgeladen, indem über das API Gateway eine AWS Lambda Funktion ausgelöst wird, die die nötigen Zugänge für Amazon S3 holt. Der Upload in einen S3-Bucket beginnt, sobald der Zugang erhalten wurde. Dieser Vorgang findet sich auch in dieser Arbeit wieder, mit dem Unterschied, dass Bilder anstatt Videos hochgeladen werden. Dies entspricht Prinzip III serverloser Architekturen.

Die Media Transcoding Pipeline beginnt mit der Transkodierung der hochgeladenen Videos sofort nach dem Upload. Die neuen Dateien werden in einem weiteren S3-Bucket gespeichert, der die Videos sofort dem User bereitstellt. An dieser Stelle wird erneut Prinzip III von serverlosen Architekturen verwendet. Die Media Transcoding Pipeline findet sich in abgeänderter Form als Image Converting Pipeline in der Architektur des Prototyps wieder. Hinzu kommt, dass Firebase zu diesem Zeitpunkt nicht verwendet wird.

Um ein Video sehen zu können, wird von einer weiteren Lambda-Funktion, die Erlaubnis für den User für eine begrenzte Zeit auf Read gesetzt. Die Dateien werden über CloudFront aufgerufen. Dies wird in abgeänderter Form in den Prototyp übernommen. Es wird eine SetPermissions Lambda-Funktion geben, die für einen Kunden des Fotografen das Betrachten seiner Fotos erlaubt. Im ersten Prototyp wird diese Funktion die Bilder aber für alle User auf Read setzen, anstatt nur für den Kunden des Fotografen. CloudFront wird zudem nicht für die Betrachtung der Bilder verwendet werden, da die Benutzung von CloudFront den Preis in die Höhe treiben würde. CloudFront ist wichtiger für die Präsentation von Videos, da hier höhere Datenmengen pro Sekunde übertragen werden müssen.

In der Referenzarchitektur gibt es ein Q&A-Forum, auf das ebenfalls über das API Gateway und dahinterstehende Lambda-Funktionen zugegriffen wird. Die Fragen, Antworten und Kommentare werden in der Datenbank gespeichert und zu AWS CloudSearch gesendet, um eine Suchfunktion im Forum bereitzustellen. Diese Komponente wird im Prototyp nicht genutzt werden. Es handelt sich hierbei um eine

[13] [Auth0 b]
[14] [Firebase]

nützliche Funktion für eine Website mit vielen Usern, durch die sich User austauschen können. Diese Funktion bietet aber keinen Mehrwert für Fotografen und ihre Kunden.

Aus der Referenzarchitektur und den aufgeführten Änderungen, entsteht der erste Entwurf für die serverlose Architektur in der Abbildung.

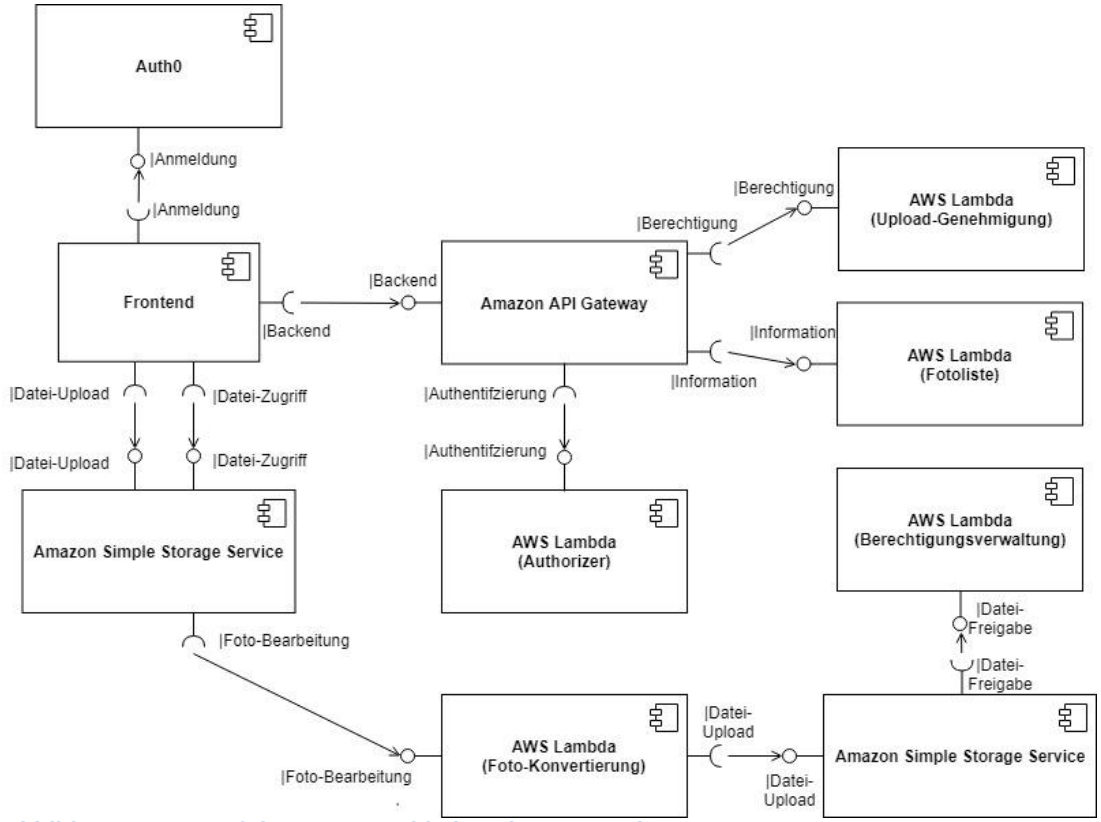

Abbildung 4: Entwurf der ServerArchitektur (1. Prototyp)

4.2 Zugriff auf Dienste

Der Zugriff auf Amazon S3 wird über das API Gateway realisiert. Auf das API Gateway selber soll mithilfe der ID Tokens von Auth0 (https://auth0.com/) zugegriffen werden.

Ein ID Token ist ein JSON Web Token (JWT), der Profilinformationen des Benutzers in der Form von Claims enthält. ID Tokens folgen dem Industriestandard IETF RFC 7519[15], beruhen auf dem OAuth 2.0 Protokoll[16] und enthalten einen Header, einen Body und eine Signatur.

Im Header findet sich, um welche Art von Token es sich handelt. Außerdem ist hier der Hash-Algorithmus hinterlegt, der für die Signierung des Tokens verwendet wurde. Im Body befinden sich die Claims wie die E-Mail-Adresse und der Name.
In der Signatur wird verifiziert, dass der Sender des JWT derjenige ist, der er vorgibt zu sein, und dass die Nachricht nicht nach dem Absenden geändert wurde.[17]

Durch die ID Tokens kann der Zugriff aufs API Gateway und damit auf Amazon S3 und AWS Lambda direkt aus dem Frontend erfolgen, da das API Gateway einen mitgeschickten ID Token durch eine Lambda-Funktion verifizieren kann. Dies entspricht Prinzip IV serverloser Architekturen.

4.3 Backend

Für die Umsetzung eines serverlosen Backends gibt es neben AWS Lambda einige weitere Möglichkeiten. Theoretisch könnte man die Entscheidung von den zur Verfügung stehenden Laufzeitumgebungen bzw. Programmiersprachen abhängig machen. Lambda bietet hier Node.js, Python, Java und C#. Google Cloud Functions bietet nur Node.js an, da sich Google mehr auf seinen Dienst Firebase zu konzentrieren scheint, Microsoft Azure Functions hatte zum Start bereits einige Programmiersprachen wie JavaScript, C#, Python und PHP. Nach der Entscheidung für eine Programmiersprache, unterscheiden sich die Anbieter im Grunde nur minimal.[18]
Amazon kann neben AWS Lambda allerdings ein mächtiges Produktportfolio bieten, an welches die anderen Dienste noch nicht heranreichen. In dieser Arbeit wurde beispielsweise Amazon S3 für die Speicherung der Fotos, Amazon CloudWatch für Monitoring und Troubleshooting und Amazon API Gateway als API benutzt. Ein weiterer Pluspunkt ist die Reife von AWS Lambda. Seit 2015 ist AWS Lambda aus

[15] [JMBPSN 2015]
[16] [OAuth]
[17] [Auth0 a]
[18] [Mytton 2017]

der Preview-Phase ausgetreten und für alle Nutzer verfügbar.[19] Das bedeutet nicht nur, dass die Plattform mittlerweile von vielen Entwicklern getestet wurde und durch Feedback viele Fehler ausbessern konnte, es heißt auch, dass es zu AWS Lambda mehr Diskussionen, Tutorials, Blogeinträge und Literatur gibt als zu den anderen Plattformen. Gerade für Anfänger in der Entwicklung einer serverlosen Anwendung ist dies ein wichtiger Punkt.

Wenn also nach dem aktuellen Entwicklungsstand der Plattform, der Größe der Community und dem Produktportfolio entschieden wird, dann sticht AWS Lambda heraus. Zusätzlich gehört AWS Lambda mit Microsoft Azure Functions zu den günstigsten Anbietern. Damit erfüllt AWS Lambda die nicht-funktionalen Anforderungen I bis IV.

4.4 Frontend

Für das Frontend könnte eines der größeren Single Page Application (SPA) Frameworks wie Angular oder React verwendet werden. Dies würde jedoch weitere Einarbeitungszeit bedeuten.

Weil das Frontend nur einen Nebenteil dieser Arbeit darstellt, wird für den Prototyp nur ein JavaScript-Grundgerüst mit Bootstrap von Initializr (http://initializr.com/) verwendet. Im weiteren Verlauf der Prototypisierung kann dann ein SPA-Framework nach Kriterien wie Entwicklungsstand, Popularität und Größe der Community ausgewählt werden. Für das Frontend wird also in einer späteren Entwicklungsphase eine Auswahl getroffen werden, wie es zuvor für das Backend geschah.

[19] [Barr 2015]

4.5 Dienste

4.5.1 Authentifizierung

Es wird ein Dienst zur Benutzerauthentifizierung benötigt. Die Anmeldung soll vorerst nur per E-Mail oder über den Google-Account möglich sein, um den Aufwand hier auf ein Minimum zu beschränken. Die verwendeten Dienste sollen durch die ID Tokens von Auth0 die Identität des Benutzers überprüfen können. Der ID Token wird für alle Anfragen zum API Gateway benötigt, wo er durch eine Lambda-Funktion verifiziert wird. Damit der Image-Upload authentifiziert wird, muss eine Lambda-Funktion, die hinter dem API Gateway steht, eine Upload-Policy erstellen. Die Einzelheiten der Upload-Policy und der Authentifizierung durch einen ID Token werden in Abschnitt 5 (Realisierung) im Detail erklärt.

4.5.2 Speicherung

Für die Bilder des Fotografen wird ein Dienst, der Speicher zur Verfügung stellt, benötigt. Auf diesem Dienst werden ebenfalls die Website, sowie die AWS Lambda Funktionen gehostet. Für die Speicherung wird Amazon S3 genutzt, um im Amazon-Ökosystem zu bleiben und die Zusammenarbeit mit AWS Lambda möglichst problemlos zu gestalten. Durch die Verwendung von Amazon S3 ist die Speicherung an eine Region gebunden. Für diese Arbeit wird die zentraleuropäische Region eu-central-1 (Frankfurt) gewählt, um möglichst nah an den Amazon-Servern zu sein.

4.5.3 API Gateway

Für den Zugriff auf Amazon S3 und für die indirekte Ausführung der AWS Lambda-Funktionen wird ein Dienst mit Schnittstelle nach außen benötigt. Um diese Anforderungen zu erfüllen, wird ein Angebot aus dem Amazon-Ökosystem namens Amazon API Gateway verwendet. Ein Dienst, der laut Aussage des Anbieters Hunderttausende gleichzeitige API-Aufrufe verarbeiten kann.[20]

Das API Gateway wird eine Art Proxy für den Zugriff auf AWS-Lambda und Amazon S3 bilden, da es im API Gateway möglich ist, die Anfragen zuvor zu autorisieren.

[20] [AWS b]

4.5.4 Identity and Access Management

AWS Identity and Access Management (IAM) ist ein Webservice, der für die Verwaltung von Berechtigungen zuständig ist.[21]
Durch IAM kann neben einem Root-Benutzer mit allen Berechtigungen, auch anderen Benutzern Berechtigungen erteilt werden. Dies geschieht, ohne dass Passwort oder Zugriffschlüssel des Root-Benutzers geteilt werden müssen.

Für die Arbeit im Backend wird es einen Benutzer geben, der vollen Zugriff auf Amazon S3 und AWS Lambda bekommt. Für den Upload zu Amazon S3 aus dem Browser wird ein besonderer Benutzer geschaffen, der lediglich die Berechtigung für den Upload zu Amazon S3 bekommt, da die Informationen dieses Benutzers für die Upload-Policy genutzt werden und diese im Frontend verwendet wird.

[21] [AWS o]

4.6 Kosten

Kosten sind für eine Web-Anwendung besonders zu Beginn der Produktivsetzung ein wichtiger Punkt. Die serverlose Anwendung in dieser Arbeit will erreichen, dass Kosten nur anfallen, wenn der Service auch genutzt wird. Unterwartete Kosten, die durch eine falsche Konfiguration von AWS Lambda-Funktionen verursacht werden könnten, sollen vermieden werden, indem der Fakturierungsalarm von Amazon Cloudwatch genutzt wird. Durch den Fakturierungsalarm warnt CloudWatch den Benutzer, wenn bei Diensten wie AWS Lambda eine zuvor festgelegte Kostengrenze überschritten wird.[22]

Weitere Kostenersparnis bei Verwendung des Amazon-Ökosystems ist durch die Dienste AWS Trusted Advisor und Cost Explorer möglich. AWS Trusted Advisor ist ein Dienst, der Verbesserungen für Performance, Fehlertoleranz, Sicherheit und Kostenoptimierung verspricht.[23] Cost Explorer ist ein kostenloses Reporting und Analytics Tool, das manuell aktiviert werden muss. Das Tool analysiert die Kosten des laufenden Monats sowie der letzten vier Monate und erstellt aus den Analysedaten eine Prognose für die kommenden drei Monate.[24]

Für den Prototyp wird nur Amazon CloudWatch verwendet werden, allerdings noch nicht für den Fakturierungsalarm, sondern für die Protokollfunktion.

4.6.1 Amazon S3

Für eine grobe Schätzung der Kosten wurde der AWS Simple Monthly Calculator (http://calculator.s3.amazonaws.com/index.html) genutzt, mit dem sich zumindest für Amazon S3 eine ungefähre Aufstellung der Kosten erstellen lässt.

Für die Rechnung wurde ein hypothetischer Fotograf angenommen, der pro Monat 10 Kunden hat. Für diese Kunden fotografiert er jeweils 500 Bilder mit einer Größe von 10 Megabyte pro Bild. Daraus folgen 5.000 POST-Requests durch den Upload der Fotos und weitere 5.000 POST-Requests, wenn die Bilder nach der Bearbeitung durch die Lambda-Funktion erneut hochgeladen werden. An Speicherplatz werden für die 5.000 Fotos aller Kunden 50 Gigabyte benötigt.

Für die GET-Requests zu S3 wurde angenommen, dass jeder Kunde seine Bilder 20 Mal aufruft, bis er weiß, welche er bearbeitet haben möchte. Daraus werden 200.000 GET-Requests. In der folgenden Abbildung findet sich ein Ausschnitt aus dem Kostenrechner für S3.

[22] [AWS j]
[23] [AWS i]
[24] [AWS d]

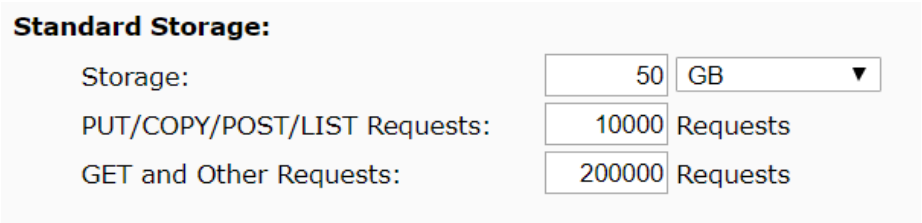

Abbildung 5: Quantitätsangaben für Amazon S3

Für Amazon S3 kommen die geschätzten Kosten damit auf $1,38 (1,12€). In den ersten 12 Monaten gibt es für Neukunden bei Amazon Web Services zusätzlich ein kostenloses Kontingent, das den Preis weiter senken würde. Da dieses kostenlose Kontingent nach 12 Monaten endet, wurde es nicht mit eingerechnet.

Abbildung 6: Kosteneinschätzung für Amazon S3

4.6.2 AWS Lambda

Die Kosten für AWS Lambda basieren auf der Anzahl der Requests, der Ausführdauer der Funktion und der Höhe des zugeteilten Arbeitsspeichers. Aufgrund der nicht in Rechnung gestellten ersten Million Anfragen würden in dem Beispiel mit nur einem Fotografen noch keine Kosten anfallen. Bis der kostenpflichtige Bereich erreicht wird, bräuchte man bei dem jetzigen Entwurf des Prototyps in etwa 50 der hypothetischen Fotografen, denn 5.000 Fotouploads mit 2 aufgerufenen Lambda Funktionen per Upload ergibt hochgerechnet nur 10.000 Requests pro hypothetischem Fotografen. Die Lambda Funktion, die den Upload autorisiert, muss nicht für jedes einzelne Foto aufgerufen werden, wenn alle Fotos eines Kunden auf einmal hochgeladen werden. Hier würde also für 500 Uploads nur 1 Request anfallen. Auch die Liste der Bilder im Bucket, die für das Betrachten der Bilder auf der Website aufgerufen wird, müsste nur einmal aufgerufen werden. Alle weiteren Anfragen gehen direkt an Amazon S3.

Um sich ein ungefähres Bild von den Kosten für Lambda zu machen, wurden mit dem Serverless Cost Calculator (http://serverlesscalc.com/) die Kosten für die beiden am häufigsten genutzten Lambda Funktionen berechnet, wenn sie jeweils eine halbe Million Requests bekommen. Die Laufzeiten der Funktionen stammen aus den von CloudWatch ausgegebenen Protokollen nach Durchlauf der Funktionen. Für eine Million Anforderungen werden von AWS Lambda $0,20 berechnet und für die Rechenzeit wird abhängig von der Höhe des verwendeten Arbeitsspeichers gerechnet. Der niedrigste Arbeitsspeicher von 128 Megabyte wird bei 100ms Laufzeit mit $0,000000208 berechnet und bei einem Gigabyte Arbeitsspeicher mit $0,000001667.

Für die Konvertierungsfunktion der Fotos ConvertImage fallen bei 500.000 Requests $0,10. Für die Rechenzeit von 200ms mit 512 Megabyte verfügbarem Arbeitsspeicher sind es $0,83, was insgesamt $0,93 (0,76 Euro) für den Monat ergibt.

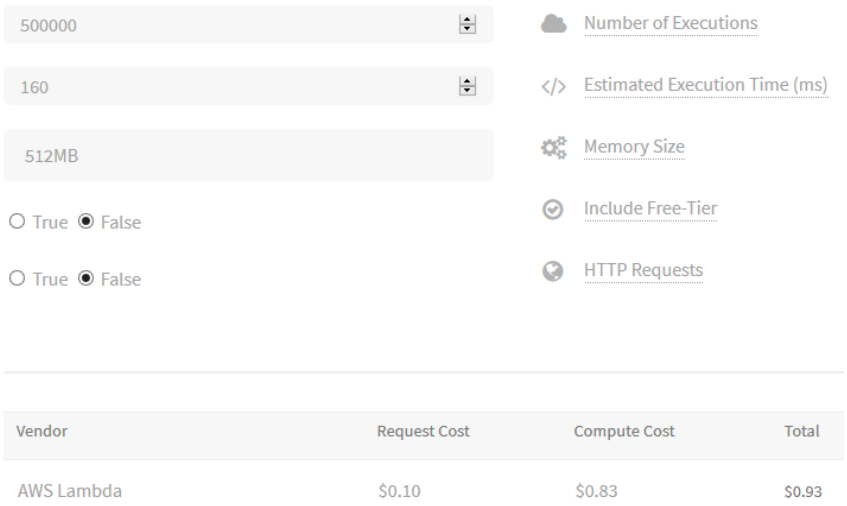

Abbildung 7: Kostenrechnung für ConvertImage Funktion

Für die Funktion, mit der die Berechtigungen gesetzt werden (SetPermissions), ergeben sich bei 500.000 Requests, 1.000ms Laufzeit und 196 Megabyte verfügbarem Arbeitsspeicher Kosten von $1,66 (1,35€).

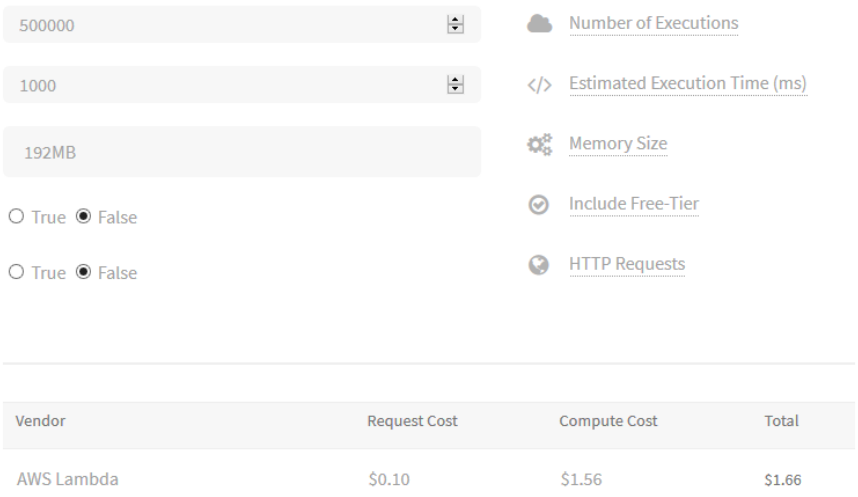

Abbildung 8: Kostenrechnung für SetPermissions Funktion

4.6.3 API Gateway

Bei der Berechnung für die Kosten des API Gateway, wird – wie bei der Amazon S3 Kostenrechnung – das kostenfreie Segment, von einer Million API Calls pro Monat[25], für die ersten 12 Monate außer Acht gelassen. Eine Million API Calls gehen mit $3,50 in Rechnung. Das niedrigste Level für Datenübertragung geht bis 10 Terabyte mit 0,09 $/Gigabyte.

Würde jedes Foto des hypothetischen Fotografen einzeln hochgeladen werden und für jeden Upload ein API Call gemacht werden, würden für 5000 API Calls lediglich $0.0175 berechnet werden.[26] Für 10 API Calls bei Upload von 500 Fotos in einem Satz sind es nur $0,000035.

Um die Gesamtgröße der Datenübertragung zu berechnen, wird sich an dem Preisbeispiel der API-Gateway-Produktvorstellungsseite orientiert.[27] Die API empfängt durch den hypothetischen Fotografen 10 bis 5000 API Calls im Monat. Jeder API Call gibt Antworten mit einer Größe von 3 KB zurück:

3 KB * 10 = 0,03 MB
3 KB * 5000 = 15 MB

Die Kosten der Datenübertragung belaufen sich bei einer Gesamtgröße von 0,03 bis 15 Megabyte auf $0,00027 bis $0,00135. Geht man etwas in die Höhe mit der Rechnung, so müssten ~111 Megabyte übertragen werden, damit überhaupt $0,01 an Kosten anfallen.

4.6.4 Zusammenfassung

Insgesamt kommen für einen einzelnen hypothetischen Fotografen nur die Kosten für Amazon S3 und Amazon CloudWatch zur Geltung. Wird die Anzahl der Fotografen auf 50 erhöht, ergeben sich geringe Kosten für AWS Lambda und das API Gateway, die Kosten für Amazon S3 steigen auf $68,25 an, bei einer Leistung von 2.500GB Speicherplatz, 500.000 POST- und 10.000.000 GET-Requests.

Abbildung 9: Kosteneinschätzung für Amazon S3 (50 Fotografen)

[25] [AWS c]
[26] [AWS l]
[27] [AWS c]

5 Realisierung

In diesem Teil der Arbeit wird die Implementierung der Komponenten für den Prototyp vorgestellt. Zu Beginn werden die AWS Lambda-Funktionen vorgestellt, um dem Leser die Arbeit mit Lambda näher zu bringen. Der Prototyp arbeitet mit fünf Lambda-Funktionen. Jede der Lambda-Funktionen erfüllt genau eine Aufgabe und hält sich damit an Prinzip II serverloser Funktionen. Die Funktion „ConvertImage“ bearbeitet ein hochgeladenes Bild und die Funktion „SetPermissions“ ändert die Berechtigungen für das Bild. Diese Funktionen sind in Java verfasst. Die Node.js Funktionen „get-upload-policy“, „custom-authorizer“ und „get-photo-list” werden benötigt, um den Client über das API Gateway auf den S3-Buckets und die Dateien in den Buckets zugreifen lassen zu können. Die Entscheidung für die späteren Funktionen Node.js zu verwenden wurde aufgrund der einfacheren Arbeit mit JSON Web Tokens in Node.js gefällt, zudem gab es Beispiele von Amazon in der Dokumentation, auf denen aufgebaut werden konnte, die für Java nicht vorhanden waren.

Im Anschluss an die AWS-Lambda-Funktionen wird die Authentifizierung mit Auth0 im Client vorgestellt. Darauf folgt die Konfiguration in Amazon S3 und im API Gateway.
Zuletzt wird die Single-Page-Application, über welche der User die Anwendung nutzt, vorgestellt.

5.1 AWS-Lambda-Funktionen

Der Upload-Prozess (siehe Abbildung 10) beginnt mit dem Click auf den Upload-Button. Für die Erlaubnis zum eigentlichen Upload in den Amazon S3 Bucket, wird eine Upload-Policy benötigt. An das API Gateway wird ein Request gesendet, der durch die Lambda-Funktion CustomAuthorizer autorisiert wird. Nach einer erfolgreichen Autorisierung wird durch das API Gateway die Lambda-Funktion get-upload-policy ausgelöst. Diese Lambda-Funktion erstellt die Upload-Policy, kodiert sie und erstellt zusammen mit den benötigten Informationen für einen POST-Request, eine Antwort. Diese Antwort wird über das API Gateway an den Browser weitergegeben. Ab dem Erhalt der Upload-Policy kann der Upload per HTTP POST beginnen.

Das hochgeladene Bild wird in einem Bucket gespeichert, der für die unbearbeiteten Bilder bestimmt ist. Nach dem Upload des Bildes wird automatisch die Lambda-Funktion ConvertImage durch ein Amazon S3 Event ausgelöst. ConvertImage fügt nun das Wasserzeichen hinzu und reduziert die Qualität des Bildes. Im Anschluss wird das Bild in einen weiteren Bucket hochgeladen, der für die bearbeiteten Bilder gedacht ist. Das Hochladen eines Fotos in diesem Bucket bietet erneut die Möglichkeit, eine Lambda-Funktion direkt durch ein Amazon S3 Event auszulösen. Eine weitere Möglichkeit, um den Aufruf der Lambda-Funktion zu implementieren, ist die Verwendung von Simple Notification Service[28] (SNS). SNS ist ein Dienst, bei dem ein Thema erstellt wird, welches z.B. von Webservern, E-Mail-Adressen, AWS Lambda-Funktionen abonniert werden kann. Die Verwendung von SNS an dieser Stelle ermöglicht neben dem Auslösen von Lambda auch ein Versenden einer E-Mail, die den Upload des Objekts meldet. Zu diesem Zeitpunkt hat diese E-Mail noch keine Relevanz, in einem Nachfolger des Prototyps kann an dieser Stelle aber leicht eine E-Mail an den Kunden gesendet werden, um ihn zu informieren, dass der Fotograf Fotos hochgeladen hat.

[28] [AWS p]

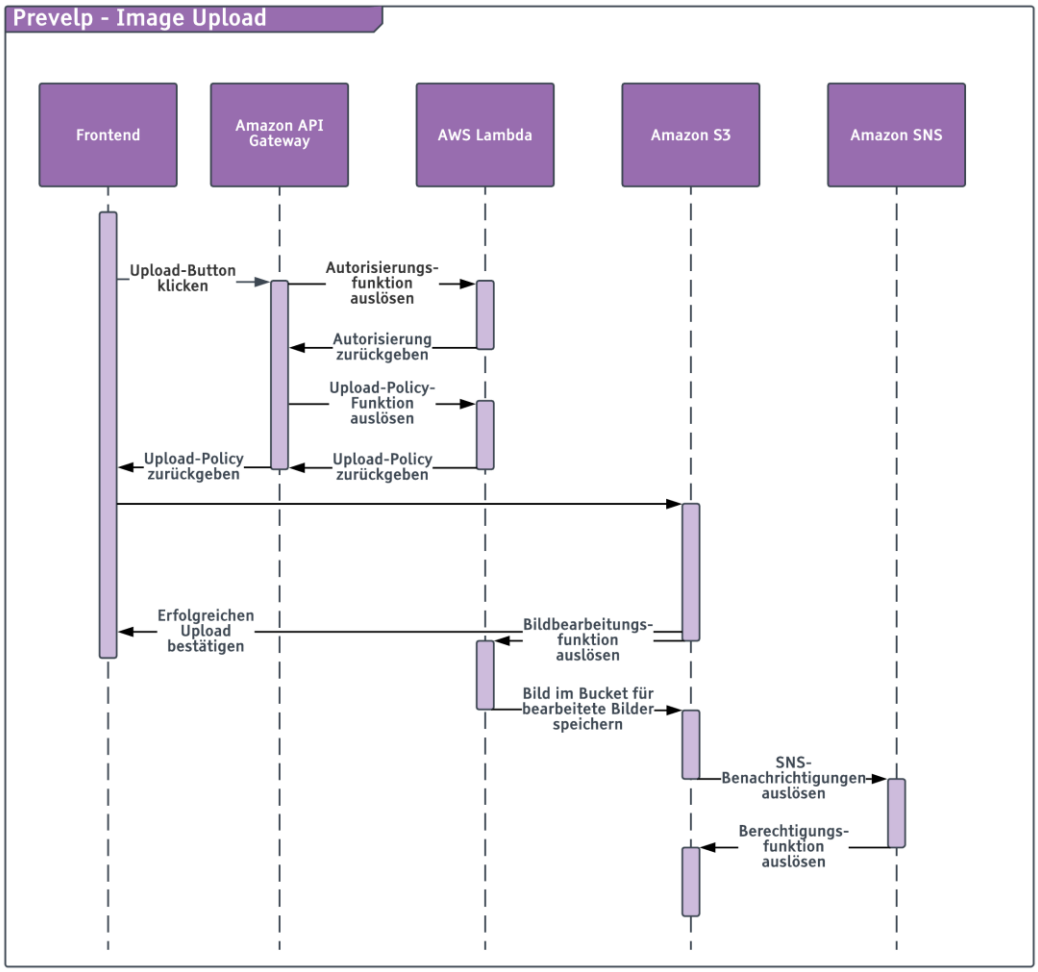

Abbildung 10: Ablauf vom Foto-Upload

In den folgenden Abschnitten werden die relevanten Teile der Lambda-Funktionen vorgestellt. Der vollständige Programmcode ist im Anhang zu finden.

5.1.1 ConvertImage

Die ConvertImage Lambda-Funktion wird automatisch ausgelöst. Das automatische Auslösen der Lambda Funktion durch ein Ereignis lässt sich in den Eigenschaften des Buckets als Event einstellen (siehe Abbildung 11).

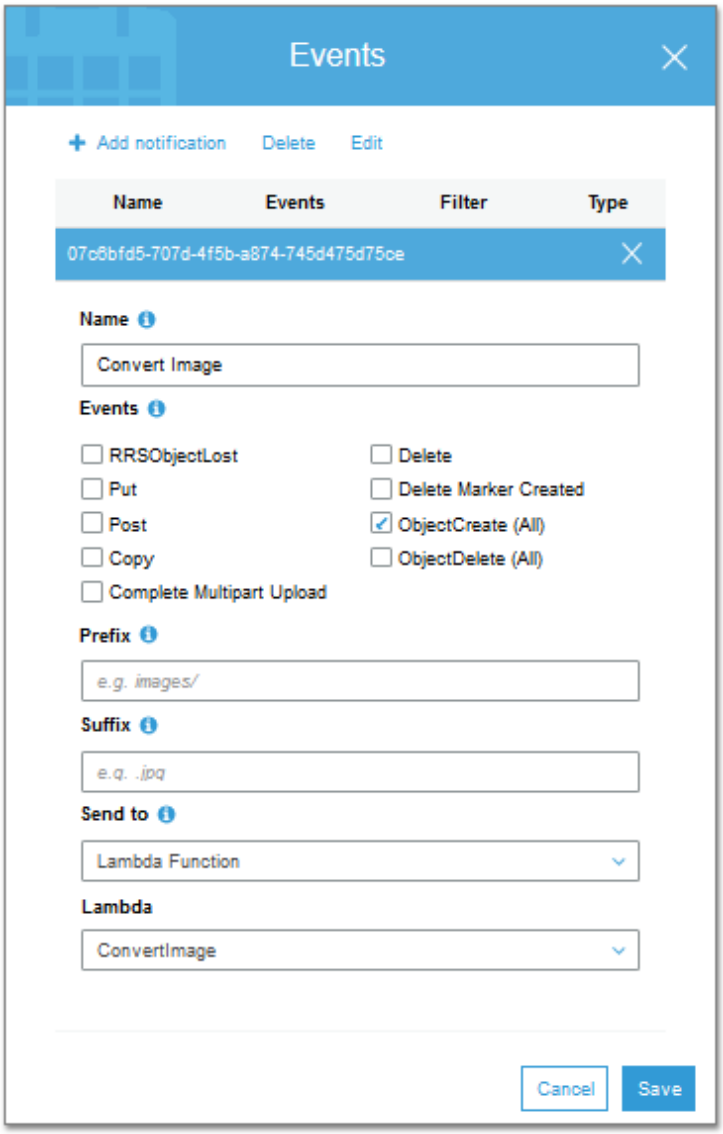

Abbildung 11: S3-Auslöser für ConvertImage

Wie in Abbildung 12 zu sehen, wird aus dem S3-Event der Name des S3-Buckets und der Key (Name des Fotos) entnommen. Mit diesen Informationen wird der Bucket für das konvertierte Foto ausfindig gemacht.

Für die Extrahierung des ursprünglichen Fotos und das anschließende Hochladen des konvertierten Fotos wird ein S3-Client benötigt, der zuvor mit einer Klasse aus dem AWS SDK für Java namens AmazonS3ClientBuilder erstellt wird.

Die Extrahierung des Fotos wird mit dem S3ObjectInputStream aus dem AWS SDK realisiert. Nach der Konvertierung – also der Qualitätsreduktion und dem Hinzufügen des Wasserzeichens durch zwei Hilfsmethoden – wird das Foto wieder hochgeladen.

```
S3ObjectInputStream objectData =
        this.s3Client.getObject(srcBucket, srcKey).getObjectContent();
BufferedImage srcImage = ImageIO.read((InputStream)objectData);
ConvertImage.reduceQuality(srcImage, imageType, srcKey);
ConvertImage.addTextWatermark("test", srcImage);
ByteArrayOutputStream os = new ByteArrayOutputStream();
ImageIO.write((RenderedImage)srcImage, imageType, os);
ByteArrayInputStream is = new ByteArrayInputStream(os.toByteArray());
ObjectMetadata meta = new ObjectMetadata();
meta.setContentLength((long)os.size());
if ("jpg".equals(imageType)) {
    meta.setContentType("image/jpeg");
}
if ("png".equals(imageType)) {
    meta.setContentType("image/png");
}
context.getLogger().log("Writing to: " + dstBucket + "/" + dstKey);
this.s3Client.putObject(dstBucket, dstKey, (InputStream)is, meta);
context.getLogger().log("Successfully converted " + srcBucket
        + "/" + srcKey + " and uploaded to " + dstBucket + "/" + dstKey);
return "Ok";
```

Abbildung 12: ConvertImage – Konvertierung und Upload des Fotos

5.1.2 SetPermissions

Der Request-Handler der SetPermissions Lambda-Funktion erhält ein SNSEvent und das Context-Objekt als Input. Aus der von SNS erhaltenen Nachricht, wird der Name des S3-Buckets und der Key des Fotos extrahiert.

Aus den erhaltenen Informationen des SNSEvent können, mit der Amazon S3 AccessControlList[29], die Berechtigungen für das konvertierte Foto im S3-Bucket angepasst werden (siehe Abbildung 13). Im 1. Prototyp wird für alle Benutzer erlaubt, dass die Datei gelesen werden darf. In späteren Versionen wird an dieser Stelle angesetzt werden, um nur dem Kunden und dem Fotografen die Erlaubnis zum Lesen der Datei zu geben.

```
AccessControlList acl = this.s3Client.getObjectAcl(bucket, key);
acl.grantPermission((Grantee)GroupGrantee.AllUsers, Permission.Read);
this.s3Client.setObjectAcl(bucket, key, acl);
return "ok";
```

Abbildung 13: SetPermissions – ACL

[29] [AWS a]

5.1.3 get-upload-policy

Der Upload von Fotos soll nur für registrierte und authentifizierte User möglich sein. Deshalb wird eine Lambda-Funktion erstellt, die den User validiert und die notwendigen Informationen für den Upload zu Amazon S3 bereitstellt. Dies geschieht über das Amazon API Gateway, das den User mit der Lambda-Funktion custom-authorizer autorisiert und dann – ebenfalls per Lambda – eine Upload-Policy für Amazon S3 erstellt. Die Erzeugung der Policy wird auf Lambda vorgenommen, damit die Zugangsdaten für den Upload zu Amazon S3 nicht im Client gespeichert werden müssen.
Die Upload-Policy wird dann mit den, für einen HTTP POST-Request zu Amazon S3, notwendigen Informationen an den Client gesendet. Nach dem Erhalten dieser Informationen beginnt der Client mit dem Upload per HTTP POST.

Amazon S3 unterstützt HTTP POST-Requests, mit denen Benutzer direkt in einen Bucket uploaden können. Der Prozess für das Senden eines POST Requests ist laut der Amazon Simple Storage Service Dokumentation[30] wie folgt:

1. Es muss eine Security-Policy erstellt werden, in der spezifiziert wird, was in dem Request erlaubt ist.
2. Basierend auf der Policy wird eine Signatur erstellt.
3. Erstellen eines HTML-Formulars, mit dem der Benutzer in den S3-Bucket hochladen kann.

Für einen authentifizierten POST-Request müssen die Policy und eine gültige Signatur mitgeschickt werden.

[30] [AWS e]

Eine Signatur wird folgendermaßen erstellt:

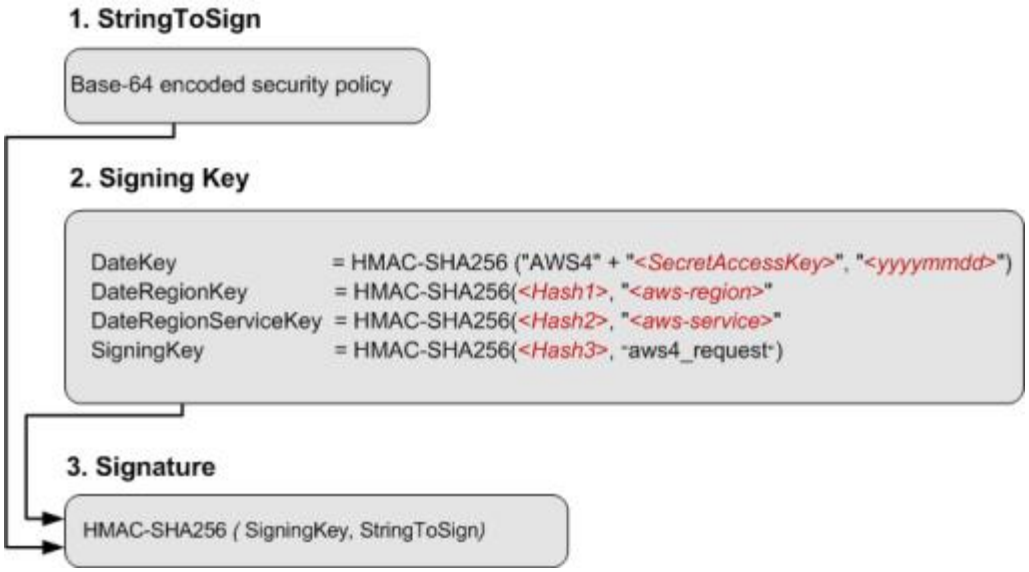

Abbildung 14: get-upload-policy – Signatur (Quelle: Amazon)

1. Eine Policy wird mit UTF-8 Zeichenkodierung erstellt.
2. Die Bytes der UTF-8 Policy werden in Base-64 umgewandelt. Daraus entsteht der StringToSign (Schritt 1 in Abbildung 14).
3. Um die Signatur zu erstellen muss ein Signing Key erstellt werden. Im ersten Schritt wird ein hashbasierter Message Authentication Code (HMAC) mithilfe der SHA256 hash-Funktion erstellt. Zuerst wird ein DateKey erstellt, indem der String „AWS4“, der SecretAccessKey eines zum Upload berechtigten Account und das Datum in der Form YYYYMMDD mit der hash-Funktion verschlüsselt werden. Danach wird der DateRegionKey aus dem DateKey und der Region des Buckets erstellt, indem wieder mit der hash-Funktion verschlüsselt wird. Im dritten Schritt wird der DateRegionKey zusammen mit dem AWS Service – für den der POST-Request erstellt werden soll – verschlüsselt. Im letzten Schritt wird der SigningKey erstellt, indem der DateRegionServiceKey zusammen mit dem String „aws4_request“ verschlüsselt wird.
4. Der SigningKey wird genutzt, um den StringToSign mit dem HMAC-SHA256 Algorithmus zu verschlüsseln und die endgültige Signatur zu erstellen.

AWS Lambda

Das Erstellen der Policy, des StringToSign, des Signing Key und der Signatur wird mithilfe des async waterfall in Node.js vorgenommen (siehe Abbildung 15). Laut der Dokumentation der async-Library werden dem waterfall eine Reihe von Funktionen in einem Array übergeben. Jede der Funktionen gibt das Ergebnis zur nächsten Funktion im Array weiter. Wenn eine der Funktionen einen Error wirft, wird die nächste Funktion nicht ausgeführt und der waterfall gibt einen Error wieder[31]. Der waterfall wird im Request-Handler benutzt und gibt ein Ergebnis – bestehend aus der Signatur, der Base-64-Policy, dem öffentlichen Access Key, der Upload-URL, dem Dateinamen, dem Datum, der Region und dem Service an den Client zurück, der damit den POST Request erzeugt. Der Access Key und die Upload URL können aus den Umgebungsvariablen „ACCESS_KEY“ und „UPLOAD_URI“ der Lambda-Funktion entnommen werden.

```
exports.handler = function(event, context, callback) {
  var filename = null;

  if(event.queryStringParameters && event.queryStringParameters.filename) {
    filename = decodeURIComponent(event.queryStringParameters.filename);
  } else {
    callback(null, createErrorResponse(500, 'Filename must be provided'));
    return;
  }

  async.waterfall([async.apply(generatePolicyDocument, filename), encode, sign],
    function(err, key, policy, encoding, signature) {
      if(err) {
        callback(null, createErrorResponse(500, err));
      } else {
        var result =
        {
          signature: signature,
          encoded_policy: encoding,
          access_key: process.env.ACCESS_KEY,
          upload_url: process.env.UPLOAD_URI + process.env.UPLOAD_BUCKET,
          key: key,
          date: date,
          region: region,
          service: service
        }
        callback(null, createSuccessResponse(result));
      }
    }
  )
}
```

Abbildung 15: get-upload-policy – Request-Handler

[31] [async]

Zu Beginn wird die Policy erzeugt (siehe Abbildung 16). Zu einer Policy muss das Feld „expiration“ gehören, sowie „key“ für den Dateinamen, „bucket“ für den Upload-Bucket, „x-amz-algorithm“ für den Signierungsalgorithmus, „x-amz-credential“ für Informationen, die für das Berechnen der Signatur verwendet wurden und „x-amz-date“ für das Datum im ISO8601-Format[32]. Durch „next(null, key, policy)“ werden der Dateiname und die Policy an die nächste Funktion „encode“ weitergegeben.

```
function generatePolicyDocument(filename, next) {
  var directory = crypto.randomBytes(20).toString('hex');
  var key = directory + '/' + filename;
  var expiration = generateExpirationDate();

  var policy = {
    'expiration': expiration,
    'conditions': [
      {key: key},
      {bucket: process.env.UPLOAD_BUCKET},
      {acl: 'private'},
      ['starts-with', '$Content-Type', ''],
      {"x-amz-algorithm": "AWS4-HMAC-SHA256"},
      {"x-amz-credential": process.env.ACCESS_KEY + '/'
        + date + '/' + region + '/' + service + '/aws4_request'},
      {"x-amz-date": date + "T000000Z"}
    ]
  };
  next(null, key, policy);
}
```

Abbildung 16: get-upload-policy – Policy

Danach muss die UTF-8-Policy in Base-64 kodiert werden (siehe Abbildung 17). Und erneut werden Dateiname, Policy und die kodierte Policy an die nächste Funktion „sign“ weitergegeben.

```
function encode(key, policy, next) {
  var encoding = base64encode(JSON.stringify(policy), 'utf-8').replace('\n', '');
  next(null, key, policy, encoding);
}
```

Abbildung 17: get-upload-policy – encode

[32] [AWS e]

In diesem Schritt wird die Signatur aus der Base-64 kodierten Policy und dem SignatureKey erzeugt, indem beide mit dem HMAC-SHA256-Algorithmus verschlüsselt werden (siehe Abbildung 18). Aus dem entstandenen Objekt muss ein String in hexadezimal Zeichenkodierung erzeugt werden, den der Client seinem POST-Request hinzufügt.

```
function sign(key, policy, encoding, next) {
  var signatureKey = getSignatureKey(process.env.SECRET_ACCESS_KEY, date, region, service)
  var s3signature = Crypto.HmacSHA256(encoding, signatureKey).toString(Crypto.enc.Hex);
  next(null, key, policy, encoding, s3signature);
}
```

Abbildung 18: get-upload-policy – sign

Für die Erzeugung des Signature Keys werden, der Reihe nach, die Schlüssel DateKey (kDate), RegionKey (kRegion), ServiceKey (kService) und SigningKey (kSigning) erstellt (siehe Abbildung 19).

```
function getSignatureKey(key, dateStamp, regionName, serviceName) {
  var kDate = Crypto.HmacSHA256(dateStamp, "AWS4" + key);
  var kRegion = Crypto.HmacSHA256(regionName, kDate);
  var kService = Crypto.HmacSHA256(serviceName, kRegion);
  var kSigning = Crypto.HmacSHA256("aws4_request", kService);
  return kSigning;
}
```

Abbildung 19: get-upload-policy – getSignatureKey

API Gateway

Im API Gateway sieht der GET-Request für eine Upload-Policy (siehe Abbildung 20) folgendermaßen aus: der Client schickt einen GET-Request an die /s3-policy-document Ressource des API Gateways. Der Request wird durch den „custom-authorizer" autorisiert. Der Integration Request mit dem Punkt „Type: LAMBDA_PROXY" bedeutet, dass der HTTP Request – in diesem Fall GET, einschließlich aller Header, Query String Parameter und dem Body – für die Lambda-Funktion durch das event-Objekt verfügbar gemacht wird. Das event-Objekt geht also an die Lambda Funktion get-upload-policy, die im vorherigen Teil vorgestellt wurde. Die Antwort wird in der Lambda-Funktion generiert und vom API Gateway zum Client durchgegeben.

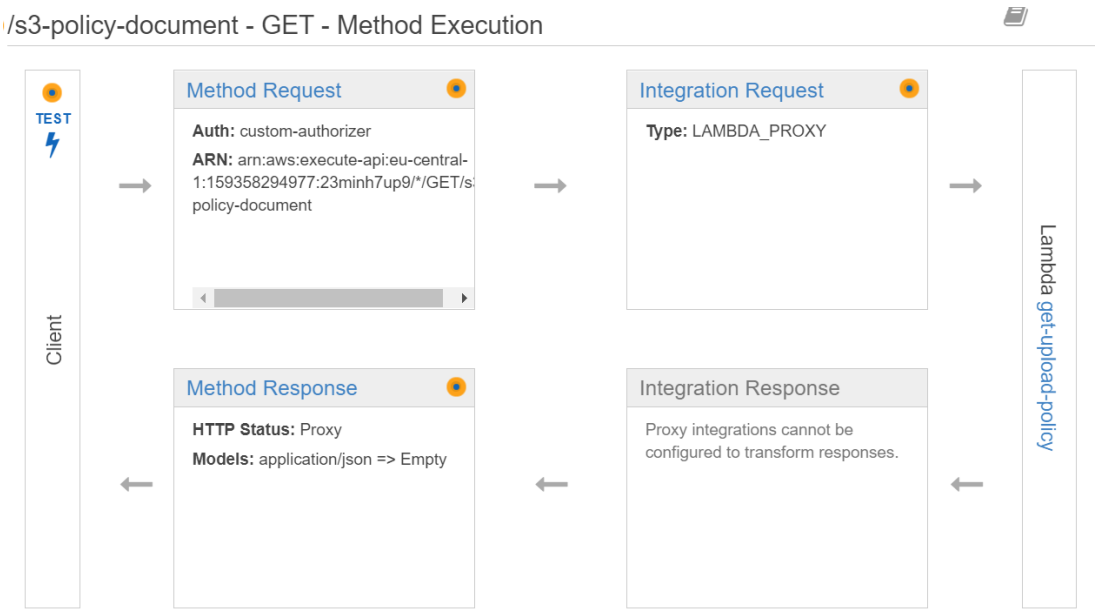

Abbildung 20: get-upload-policy – API Gateway

Client

Der Client bekommt eine Antwort vom API Gateway (siehe Abbildung 21). Aus dieser Antwort wird der POST-Request erstellt. Die Antwort enthält die Signatur und die in Base-64 kodierte Upload-Policy.

JSON

signature: b5f0bd998b68606640c30d81de05b69be87104bfc5162544cdf5181802ae7e6a
encoded_policy: eyJleHBpcmF0aW9uIjoiMjAxOC0wMi0yMlQxMzozNzozNS4wNTVaIiwiY29uZGl0aW9ucyI6W3
sia2V5IjoiMWM2MDAzMzZlNTQ1Y2FhYjQyNjAwYTI0MTUxODQ5ODZlYmQ5YTEwMS90ZXN
0LWltYWdlLTc1MHg1MDBweCAoY29weSkuanBnIn0seyJidWNrZXQiOiJwcmV2ZWxwLWltY
WdlLXVwbG9hZCJ9LHsiYWNsIjoicHJpdmF0ZSJ9LFsic3RhcnRzLXdpdGgiLCIkQ29udGVudC1
UeXBlIiwiIl0seyJ4LWFtei1hbGdvcml0aG0iOiJBV1M0LUhNQUMtU0hBMjU2In0seyJ4LWFtei1j
cmVkZW50aWFsIjoiQUtJQUpSVElFVlpPVEFESUhWVkEvMjAxODAyMjEvZXUtY2VudHJhbC0
xL3MzL2F3czRfcmVxdWVzdCJ9LHsieC1hbXotZGF0ZSI6IjIwMTgwMjIxVDAwMDAwMFoifV19
access_key: AKIAJRTIEVZOTADIHVVA
upload_url: http://s3.eu-central-1.amazonaws.com/prevelp-image-upload
key: 1c600336e545caab42600a2415184986ebd9a101/test-image-750x500px (copy).jpg
date: 20180221
region: eu-central-1
service: s3

Abbildung 21: get-upload-policy – GET-Response

Im Client werden die Informationen aus der GET-Response in ein HTML-Formular eingetragen (siehe Abbildung 22), das im POST-Request mitgeschickt wird. Im Formular muss jeder Eintrag enthalten sein, der auch in der Upload-Policy unter dem Punkt „conditions“ enthalten ist. Als letzter Teil des Formulars muss die Datei hinzugefügt werden.

```
var fd = new FormData();
fd.append('key', data.key)
fd.append('acl', 'private');
fd.append('Content-Type', file.type);
fd.append('policy', data.encoded_policy)
fd.append('x-amz-signature', data.signature);
fd.append('x-amz-algorithm', 'AWS4-HMAC-SHA256');
fd.append('x-amz-credential', data.access_key + '/'
  + data.date + '/' + data.region + '/' + data.service + '/aws4_request');
fd.append('x-amz-date', data.date + "T000000Z")
fd.append('file', file, file.name);

$.ajax({
  url: data.upload_url,
  type: 'POST',
  data: fd,
  processData: false,
  contentType: false,
  xhr: this.progress,
  beforeSend: function(req) {
    req.setRequestHeader('Authorization', ' ');
  }
```

Abbildung 22: get-upload-policy – POST-Request (Client)

Amazon S3

Im S3 CORS (Cross-Origin Resource Sharing) Configuration Editor (siehe Abbildung 23) muss für die POST-Methode erlaubt werden, dass sie von einer anderen Website – also einer fremden Ressource – ausgeführt werden darf. Im Prototyp sind durch die Wildcard „*“ alle Herkünfte erlaubt, da der custom-authorizer den Zugriff schützt. In einem Produktionssystem würde „AllowedOrigins“ auf den Client beschränkt werden.

CORS configuration editor ARN: arn:aws:s3:::prevelp-image-upload
Add a new cors configuration or edit an existing one in the text area below.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
3  <CORSRule>
4      <AllowedOrigin>*</AllowedOrigin>
5      <AllowedMethod>POST</AllowedMethod>
6      <MaxAgeSeconds>3000</MaxAgeSeconds>
7      <AllowedHeader>*</AllowedHeader>
8  </CORSRule>
9  </CORSConfiguration>
```

Abbildung 23: get-upload-policy – S3 CORS-Konfiguration

5.1.4 custom-authorizer

Ein benutzerdefinierter API Gateway-Genehmiger – auch bekannt als „custom-authorizer“ – ist eine Lambda-Funktion, durch welche der Zugriff auf die API-Methoden kontrolliert werden kann.
Es wird das Bearer-Token-Authentifizierungsprotokoll OAuth 2.0 verwendet, es wäre aber auch möglich, Informationen, die von Headern, Pfaden, Query Strings und mehr übergeben wurden, für die Authentifizierung zu verwenden.[33]
Zu Beginn wird die Funktion in AWS Lambda vorgestellt. Im Anschluss werden die erforderlichen Einstellungen im API Gateway erläutert.

AWS Lambda

Der Request-Handler (siehe Abbildung 24) bekommt das event-Objekt übergeben, aus dem der Token entnommen wird. Der Token ist im Format „Bearer <TOKEN>“, weshalb der eigentliche Token zu Beginn von dem String „Bearer“ getrennt wird. Damit die verify-Funktion der jwt-Library mit einem vom RS256-Algorithmus unterzeichnetem JSON-Web-Token funktioniert, wird noch ein PEM kodierter public key (prevelp.pem) mitgegeben, der zusammen mit dem Funktionscode zu AWS Lambda hochgeladen wurde. Bei erfolgreicher Verifikation des Tokens wird eine Policy, die das Aufrufen des API Gateway erlaubt, generiert.

```
  var token = event.authorizationToken.split(' ')[1];

  var cert = fs.readFileSync('prevelp.pem');
  jwt.verify(token, cert, function(err, decoded) {
    if (err) {
      console.log('Failed jwt verification: ', err, 'auth: ', event.authorizationToken)
      callback('Authorization Failed');
    } else {
      callback(null, generatePolicy('user', 'allow', event.methodArn));
    }
  })
}
```

Abbildung 24: custom-authorizer – Requesthandler

Im Anschluss muss eine Policy erzeugt werden, die es erlaubt, dass die angefragte Ressource des API Gateways aufgerufen werden darf. Die Anfrage auf die Ressource ruft dann wiederum die geforderte Lambda-Funktion auf. Erzeugt wird die Standard-Policy aus dem Amazon Dokumentationsbeispiel für einen „custom-authorizer“[34]. Der signifikante Teil ist die Zuweisung von „execute-api:Invoke“ für die Statement-Action der Policy und „allow“ für den Statement-Effect. Dieser Teil erlaubt den Aufruf der API Gateway Ressource.

[33] [AWS n]
[34] [AWS n]

API Gateway

Im API Gateway (siehe Abbildung 25) wird ein Name für den Authorizer vergeben. Dann wird der Typ des Authorizers gewählt, sowie die aufzurufende Lambda-Funktion einschließlich der Region. Als Payload im Event wird der Token angegeben, damit der Token an die Lambda-Funktion übergeben wird. Als Quelle für den Token wird der Punkt Authorization im Header des Requests übergeben.

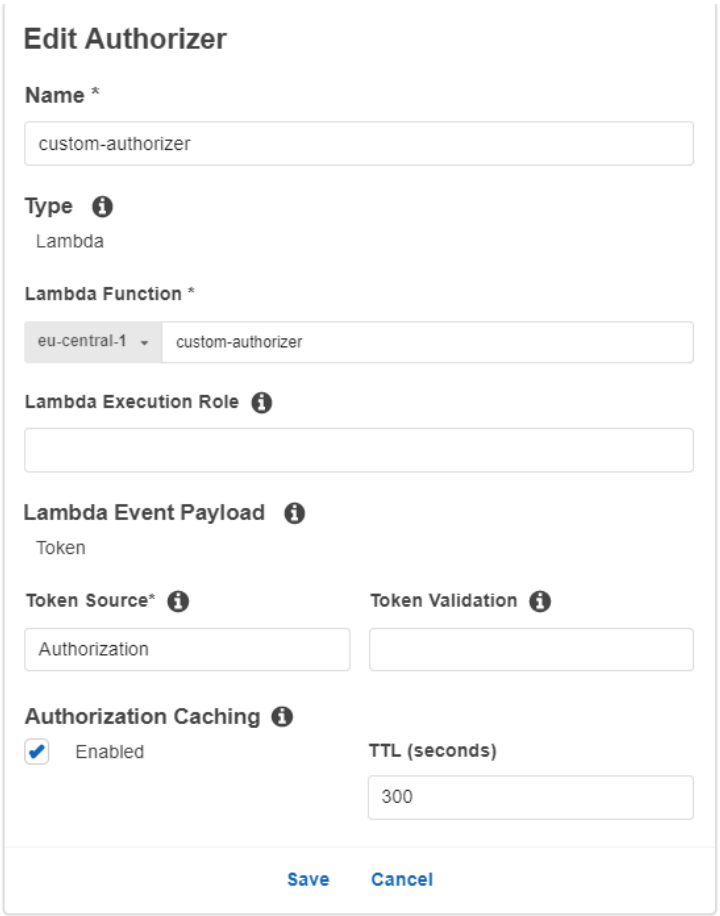

Abbildung 25: custom-authorizer – Edit Authorizer

5.1.5 get-photo-list

Das Betrachten von Fotos soll nur für registrierte und authentifizierte User möglich sein. Deshalb wird eine Lambda-Funktion erstellt, die den User validiert und die notwendigen Informationen für den Upload zu S3 bereitstellt. Dies geschieht über das API Gateway, das den User mit der Lambda-Funktion custom-authorizer autorisiert und dann – ebenfalls per Lambda – eine Upload-Policy für Amazon S3 erstellt. Die Erzeugung der Policy wird in der Lambda-Funktion vorgenommen, damit die Zugangsdaten für den Upload zu Amazon S3 nicht im Client gespeichert werden müssen.
Die Upload-Policy wird dann mit einigen, für einen HTTP POST-Request zu Amazon S3, notwendigen Informationen an den Client gesendet. Nach dem Erhalten dieser Informationen beginnt der Client mit dem Upload per HTTP POST.

AWS Lambda

In dieser Lambda-Funktion wird erneut der async waterfall verwendet, der drei Funktionen (createBucketParams, getPhotosFromBucket, createList) durchgeht, um zuerst den Bucket-Parameter (Name des Buckets) zu erhalten. Im weiteren Verlauf werden die Fotos im Bucket aufgelistet und im letzten Schritt wird aus diesen Daten eine Liste für den Client erzeugt.

Der Parameter für den Bucket wird in der Funktion createBucketParams aus der Umgebungsvariable „BUCKET“ der Lambda-Funktion entnommen und an die nächste Funktion übergeben.

Die Fotos werden in der getPhotosFromBucket-Funktion mit der listObjects-Methode für Amazon S3 aus dem AWS SDK für Javascript S3 aufgelistet. Die Auflistung der Objekte wird an die nächste Funktion im Array des waterfalls übergeben.

Für die Liste werden die Fotos aus den Daten der vorherigen waterfall-Funktion entnommen und in ein Array gepusht. Das Foto-Array wird zusammen mit der Domain (entnommen aus der Umgebungsvariable „BASE_URL“) und dem Bucket (entnommen aus der Umgebungsvariable „BUCKET“) an das API Gateway gegeben, das diese Informationen an den Client übergibt.

API Gateway

Im API Gateway erfolgt ein GET Request des Clients an die /photos Ressource des API Gateways (siehe Abbildung 26). Der Request wird durch den custom-authorizer autorisiert. Der Integration Request mit dem Punkt „Type: LAMBDA_PROXY“ bedeutet, dass der HTTP-Request für die Lambda-Funktion durch das event-Objekt verfügbar gemacht wird. Das event-Objekt geht an die Lambda Funktion get-photo-list, die im vorherigen Teil vorgestellt wurde. Die Antwort mit der Liste wird in der Lambda-Funktion generiert und vom API Gateway zum Client durchgegeben.

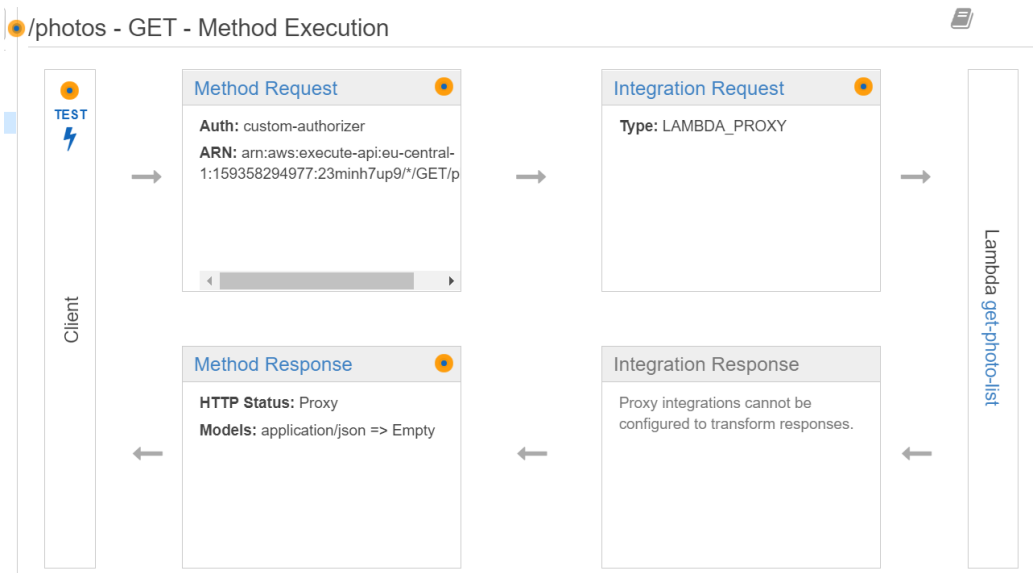

Abbildung 26: get-photo-list – API Gateway

Client

Im Client wird aus den erhaltenen Daten (data) die Domain und der Bucket entnommen, um zusammen mit dem Dateinamen des Fotos die Quelle für das „img“-Tag im HTML-Code zu generieren und einzufügen. Dieser Vorgrang ermöglicht es, zusammen mit der clone-Methode, eine Liste aus den Fotos im Bucket zu generieren, die auf der Website angezeigt wird.

5.2 Authentifizierung

Wie bereits in der Konzeption erwähnt, findet die Authentifizierung über Bearer-Token von Auth0 statt. Hierbei handelt es sich um Token, die auf dem OAuth2.0 Protokoll beruhen.

Im Client wird Lock v11[35] von Auth0 verwendet. Lock ist ein in die Website integriertes Login-Formular, das sich über das Auth0 Dashboard anpassen lässt. Im Client wird zunächst ein Auth0Lock-Objekt initialisiert (siehe Abbildung 27). Der Constructor bekommt dabei die Auth0-ClientID und die Auth0-Domain übergeben, die in der config.js des Clients hinterlegt sind. Als dritter Parameter werden dem Constructor die Konfigurationsoptionen übergeben. Hier lässt sich zum Beispiel einstellen, dass die Anmeldung in einem Popup erfolgt. Es wird weiterhin eingestellt, dass ein ID Token als Response übergeben wird und unter „audience“ wird vermerkt, welche API mit dem Access Token arbeiten soll. Unter „params“ und dort unter „scope“ wird „openid email profile“ eingetragen, was für eine konsistente Ausführung des „Last time you logged in“ feature erforderlich ist, mit dem sich Lock die letzte Login-Methode merkt[36].

```
this.data.auth0Lock = new Auth0Lock(config.auth0.clientId, config.auth0.domain, {
  popup: true,
  auth: {
    responseType: 'token id_token',
    audience: 'https://' + 'prevelp.eu.auth0.com' + '/userinfo',
    params: {
      scope: 'openid email profile'
    }
  }
});
```

Abbildung 27: Authentifizierung – Initialisierung von Lock

Ein Click auf den Login-Button bewirkt, dass das Auth0-Lock auf ein „authenticated“-Event wartet. Nachdem das Event ausgeführt wurde, kann der erhaltene Access Token verwendet werden. Der ID Token wird lokal gespeichert, um in einer Methode verwendet zu werden, die einen authentifizierten Request vorbereitet, indem der ID Token als Bearer-Token im Header von Requests verwendet wird. Dies geschieht, da der Token für die Authentifizierung durch den custom-authorizer benötigt wird, der den Bearer-Token im Header erwartet.

Von Auth0 werden zusätzlich Profilinformationen übergeben, die momentan nur als reine JSON Daten in einem Modal-Popup angezeigt werden, in späteren Versionen des Prototyps aber weitere Verwendung finden sollen.

[35] [Auth0 d]
[36] [Auth0 c]

5.3 Simple Storage Service

In Amazon S3 wurden drei Buckets erstellt. Ein Bucket für den Upload der Fotos, der bei Upload ein Event auslöst, das die Lambda-Funktion ConvertImage auslöst. In einem weiteren Bucket, werden die konvertierten Fotos gespeichert werden und in einem dritten Bucket sind die Zip-Dateien mit den Lambda-Funktionen gespeichert.

5.4 Amazon API Gateway

Im API Gateway gibt es zwei Ressourcen (/photos und /s3-policy-document), auf die per GET-Request zugegriffen werden kann. Die Ressource /photos wird benötigt, um einen GET-Request für die Fotos, im prevelp-image-converted Bucket, zu authentifizieren. Das API Gateway agiert hier als Proxy für Amazon S3. Die zweite Ressource /s3-policy-document wird per GET-Request aufgerufen, um eine Upload Policy zu erstellen. Die Lambda-Funktion könnte auch direkt aufgerufen werden, dann wäre es jedoch nicht möglich den custom-authorizer zu nutzen und die Zugangsdaten müssten im Client gespeichert werden. Das API Gateway wird also genutzt, um den Zugriff auf die Dienste im Hintergrund zu sichern.

5.5 Identity and Access Management (IAM)

In IAM wurden zwei Benutzer angelegt. Der eine Benutzer (prevelp) wird für alle Dienste genutzt. Der andere Benutzer (s3-upload) ist nur für den Upload zu Amazon S3 zuständig.

Der Benutzer prevelp hat die Policies AWSLambdaFullAccess und AmazonS3FullAcces bekommen, um diese Dienste verwalten zu können. Da der User nur zwischen AWS Diensten genutzt wird und niemals im Frontend zum Einsatz kommt, wurden die Berechtigungen an dieser Stelle nicht weiter eingeschränkt. Der Benutzer s3-upload, über den der POST-Request realisiert wird, hat wegen seinem Kontakt zum Frontend nur eingeschränkte Berechtigungen. Ihm ist lediglich erlaubt, ein Objekt in den Bucket „prevelp-image-upload“ zu legen.

Neben den Benutzern gibt es noch Rollen, die von Lambda-Funktionen angenommen werden müssen. So wurde der Lambda-Funktion ConvertImage beispielsweise die AWSLambdaExecute Rolle zugewiesen, die es ihr ermöglicht, Objekte aus dem Upload Bucket zu nehmen und in den Converted Bucket zu legen. Zusätzlich erlaubt die Rolle das Anlegen von Logs, was für die Suche nach Fehlern unverzichtlich ist.

5.6 CloudWatch

CloudWatch kann genutzt werden, um Alarme einzurichten und damit hohe Rechnungen zu vermeiden. Im Prototyp wurde CloudWatch konfiguriert, um Logs für das API Gateway und die Lambda-Funktionen aufzuzeichnen. Bei Problemen kann hier aus den Logs entnommen werden, welche Fehler oder Exceptions geworfen wurden. Bei serverlosen Anwendungen ist das Troubleshooting in der Regel etwas kompliziert, da der Client eine Antwort erhält, die in vielen Fällen nichts mit dem Fehler zu tun hat, der hinter dem API Gateway in einer Lambda-Funktion geworfen wurde. An dieser Stelle schafft ein Blick in die CloudWatch-Logs Klarheit.

5.7 Single-Page-Application

Vor dem Login sind auf der Website weder Upload-Button noch hochgeladene Bilder zu sehen. Nach einem Login sind authentifizierte Anfragen zum API Gateway möglich, da ab diesem Zeitpunkt der Bearer Token im Request Header mitgeschickt werden kann.

Der Login erfolgt über ein Login-Popup von Auth0, wie im Punkt Authentifizierung beschrieben. Standardmäßig wird im Prototyp die Anmeldung per Social Provider (Google) und per E-Mail-Adresse angezeigt oder „Last time you logged in with“, wenn es zuvor einen Login gab.

Nach dem Login ist der Aufruf des User-Profils möglich. Zusätzlich wird der Upload-Button (gekennzeichnet durch ein „+“) angezeigt und die bereits hochgeladenen Fotos sind zu sehen. In dem Screenshot ist das Wasserzeichen „Test“ über dem Bild zu erkennen, das in der ConvertImage-Funktion auf das Bild gezeichnet wurde.

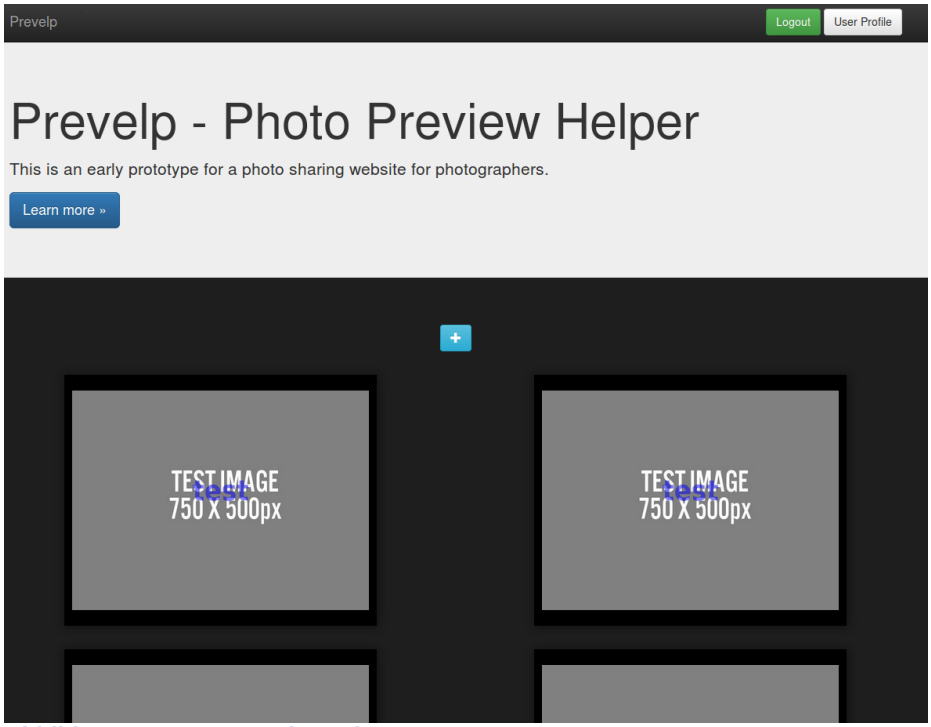

Abbildung 28: SPA – nach Login

Zu Beginn eines Uploads wird eine Fortschrittsanzeige angezeigt. Nach erfolgreichem Upload wird in einem Popup „Upload Finished“ angezeigt.

Der Click auf den „User Profile“ Button öffnet ein Modal-Popup mit den von Auth0 übergebenen Profilinformationen des angemeldeten Benutzers. Im Prototyp werden die Profilinformationen im JSON-Format angezeigt, in weiteren Iterationen der Prototypen sollen die Profilinformationen verwendet werden, um die Fotos dadurch nur dem richtigen Kunden anzuzeigen.

Amazon S3 (Hosting)

Der Prototyp wird in einem S3-Bucket gehostet. Amazon bietet für diesen Bedarf eine simple Möglichkeit einen Bucket für das Website-Hosting zu konfigurieren.[37] Nötige Einstellungen sind zum Ersten, in den Properties des Buckets, eine Angabe des Index-Dokuments. Zweitens muss, in den Berechtigungen des Buckets, eine Bucket-Policy, die es für alle Objekte des Buckets möglich macht vom Browser aufgerufen zu werden, definiert sein.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicRead",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::prevelp-website/*"
        }
    ]
}
```

Abbildung 29: Single-Page-Application – Hosting (Policy)

[37] [AWS m]

6 Evaluierung

Nachdem eine grundlegende Architektur im ersten Prototyp geschaffen wurde, die ersten Komponenten der Anwendung realisiert wurden und der allgemeine Betrieb mit AWS Lambda und den benutzten Diensten erläutert wurde, kann an dieser Stelle ein Fazit über die Entwicklung einer serverlosen Architektur mit AWS Lambda gezogen werden. Im Anschluss an das Fazit wird ein Ausblick auf die Möglichkeiten zur Erweiterung des Prototyps gegeben, der sich auf die im Abschnitt 3.2 erwähnten Ideen bezieht.

6.1 Fazit

Die Entwicklung einer serverlosen Anwendung wird sehr schnell komplex. Aufgrund der Anzahl der verwendeten Dienste und den an unterschiedlichen Stellen genutzten Lambda-Funktionen, ist es sehr einfach, den Überblick zu verlieren. Für eine erfolgreiche Entwicklung wird also eine gute Übersicht benötigt, auf die der Entwickler blicken kann, um bei der Fehlersuche und beim Debugging zu sehen, an welchen Stellen überhaupt etwas passieren sollte. An dieser Stelle sind die Diagramme der Architektur und der Abläufe hilfreich, um sich einen Überblick zu verschaffen. Gleichzeitig bietet eine gut geplante, serverlose Herangehensweise die Möglichkeit, Komplexität in Schranken zu halten und zukünftige Änderungen zu vereinfachen. Die serverlose Herangehensweise und Verwendung der Dienste hat die Menge an benötigtem Programmcode stark reduziert. Auf der anderen Seite gibt es dadurch zwar an manchen Stellen weniger Konfigurationsmöglichkeiten, der geringere Aufwand in der Verwaltung des Programmcodes gleicht dies aber für die meisten Anwendungsfälle aus. Ebenso wird der Bedarf für Layering reduziert. Der klassische Entwurf mit Präsentationsschicht, Datenschicht, und Applikationsschicht wird hier abgelöst durch das mächtigere Frontend, das mit den Diensten direkt kommunizieren kann (Prinzip IV). Im Abschnitt 4.6 wurden die Kosten mit hypothetischen Werten geschätzt. Die geschätzten Kosten für eine serverlose Anwendung waren in der Schätzung sehr gering. Das hat zum einen damit zu tun, dass keiner der teureren Dienste wie Amazon EC2 und CloudFront für den Prototyp verwendet wurde, zum anderen ist dies aber auf die eine große Eigenschaft von serverlosen System zurückzuführen: die Server müssen nicht durchgehend laufen und sie laufen fast nie auf voller Kapazität (nicht-funktionale Anforderung III). Das System kann von alleine skalieren (nicht-funktionale Anforderung II), und es ist nicht erforderlich, einen – in Anschaffung und Betrieb teuren – leistungsstarken Server für den Fall eines Ansturms

an Kunden zu betreiben (Prinzip I). Nicht außer Acht zu lassen ist ebenfalls die Möglichkeit für Entwickler, ihr vorhandenes System um serverlose Komponenten zu erweitern. Dies ist sinnvoll, wenn einige Teile des Systems stärker skalieren müssen als andere.

Die eigene Erfahrung mit der serverlosen Entwicklung war überwiegend positiv. Ein Kritikpunkt ist die schnelle Weiterentwicklung der Systeme und die immense, jedoch unübersichtliche Dokumentation von Amazon Web Services. Durch die schnelle Weiterentwicklung waren gefundene Lösungsansätze bereits überholt und oft war eine Anpassung dieser Lösungsansätze nur nach Stunden der Recherche in der Dokumentation möglich, die leider teilweise viele nicht mehr aktuelle Artikel enthält. Es ist also wie oft mit neuen Diensten noch ein Kampf mit schnell überholtem Wissen.

Zu Beginn der Arbeit mit AWS Lambda war vor allem die Arbeit mit dem Identity and Access Management (IAM) eine Herausforderung. Durch Heranziehen der Dokumentation konnte aber ein Überblick darüber geschaffen werden, was für Rollen und Policies die Lambda-Funktionen für eine erfolgreiche Ausführung benötigen.
Später gab es Schwierigkeiten durch Unklarheiten mit dem API Gateway, da im Browser beispielsweise angezeigt wurde, dass es ein CORS-Problem gibt. Das eigentliche Problem ist jedoch ein anderes gewesen – die CORS-Meldung entstand nur, da bei HTML-Statuscodes im 4XX-5XX Bereich nicht automatisch ein CORS-Header mitgesendet wurde. Durch den CloudWatch-Dienst konnte der wahre Fehler in den Logs erkannt werden.

Mit dem Authentifizierungsdienst Auth0 gab es während der Entwicklung ähnliche Schwierigkeiten wie mit den AWS-Diensten. Ein großes Problem war die große Menge an vorhandener Dokumentation und die dadurch entstehende fehlende Übersicht. Für einen Einsteiger ist dies anfangs überwältigend. Zusätzlich sind Teile der Dokumentation sehr versteckt. So war es über Tage nicht ersichtlich, wieso der custom-authorizer den ID Token nicht verifizieren konnte. Es wurde dann ersichtlich, dass Auth0, unabhängig von den Einstellungen im Dashboard, automatisch einen RS256 ID Token herausgibt anstatt einen mit dem HS256 Algorithmus signierten ID Token. Dies hat vermutlich mit einer Umstellung von Auth0 zu einem höheren Sicherheitsstandard zu tun. In der Dokumentation war dies aber nicht erkennbar, und es erschien wie eine Entscheidung, die der Auth0-Nutzer treffen könnte. Für die weitere Entwicklung der Anwendung musste ab diesem Punkt geklärt werden, wie der RS256 ID Token verifiziert wird.

Zusammengefasst stellen diese Komponenten aber dennoch eine solide Grundlage für eine serverlose Anwendung dar. Dies lässt sich vielleicht auch daraus erkennen, dass die Schwierigkeiten während der Entwicklung zum größten Teil aus fehlender Erfahrung mit den Technologien herrührten. Die Prinzipien der serverlosen Entwicklung haben sich während der Konzeption als sehr hilfreich erwiesen und sind in den fertigen Komponenten erkennbar. Sehr angenehm war auch die

Wiederverwendbarkeit von Lambda-Funktionen. So konnte der custom-authorizer für jede GET-Anfrage unabhängig von den angefragten Ressourcen verwendet werden.

Die eigene Vorgehensweise bei diesem Projekt war im Grunde zufriedenstellend. Gut war die Entscheidung, sich an einer bestehenden Anwendung, bzw. bestehenden Architektur zu orientieren. Auf dieser Grundlage konnte gut aufgebaut werden. Im Rückblick stellte sich aber heraus, dass am Anfang zuviele Punkte auf der Agenda standen. Bei einem weiteren Projekt würde der Prototyp von Anfang an auf die Grundfunktionen reduziert werden. Dies ist bei diesem Projekt schlussendlich auch geschehen, es begann aber nicht so. Die Lektion für die Zukunft ist also, sich der eigenen Kapazitäten bewusst zu sein und dementsprechend zu planen.
Weiterhin wäre es schön gewesen, wenn noch einige weitere Funktionen implementiert worden wären, die aufgrund des Lernaufwands nicht realisiert wurden und nun für spätere Prototypen auf der Liste stehen.

6.2 Ausblick

6.2.1 Projekt

Die Trennung zwischen Fotograf und Kunden des Fotografen, sollte ursprünglich mit einer angepassten SetPermissions Lambda-Funktion realisiert werden. Während der Recherche für diesen Abschnitt wurde leider ersichtlich, dass die Option, über die E-Mail-Adresse die Rechte zu vergeben, für den bisher verwendeten Bucket in der eu-central-1 Region nicht möglich ist.[38] Eine Option wäre also die verwendeten Buckets aus Frankfurt nach Irland zu verlegen. Sinnvoller wäre es aber, das Problem zu umgehen und die get-photo-list, sowie die get-upload-policy so anzupassen, dass sie in einer Datenbank anfragen, welcher Bucket dem Nutzer zugeordnet ist. Gleichzeitig würden im Frontend nur die Optionen angezeigt werden, die dem Nutzer zustehen. Die Anbindung einer Datenbank bereitet im AWS Ökosystem keine Probleme. Hier gibt es gleich mehrere Optionen für die gängigsten Formen der Datenbanksysteme.[39] Dies behebt gleichzeitig das Problem eine CRUD-Funktionalität zu schaffen.

Die Erweiterungen von User Interface, User Experience und Accessibility stellen in dieser Anwendung keine größeren Probleme als in jeder anderen Web-Anwendung.

Die Frage nach dem Archiv erfordert eine Kostenanalyse, die Möglichkeit ist allerdings bereits heute gegeben, da für diese Funktionalität kaum mehr als ein S3-Bucket erforderlich ist.

[38] [AWS a]
[39] [AWS p]

Eine Kennzeichnung der Bilder müsste über die Datenbank vorgenommen werden. Hier würden die Bilder aufgenommen werden und in einem Feld markiert werden, wenn der Kunde sie für die Postproduktion auswählen möchte. Auch das markieren der Bilder für den Fotografen sollte kein Problem darstellen, da ein bearbeitetes Bild sich von den Originalbildern im Dateinamen nur durch den Zusatz „-converted“ am Ende unterscheidet.

Das Entpacken von komprimierten Ordnern lässt sich momentan noch nicht automatisch über S3-Buckets lösen. Es ist allerdings möglich, eine Lambda-Funktion zu nutzen, die einen komprimierten Ordner nach dem Upload entpackt. In den Optionen von S3-Buckets ist es möglich, unterschiedliche Events je nach Objekt-Suffix auszulösen. Damit ist es kein Problem, die Lambda-Funktion zum Entpacken nur im Falle des Suffixes „.zip“ auszulösen, während die ConvertImage-Funktion im Falle von „.jpg“ ausgelöst werden würde.

6.2.2 Serverless

Den serverlosen Anwendungen wird eine rosige Zukunft von verschiedensten Blogs zugesagt. Eine weit verbreitete Meinung ist es z.B. AWS Lambda als nächste Entwicklungsstufe in der Entwicklung von virtuellen Maschinen, zu Containern, zu Kubernetes (Containers at scale) zu sehen. [40]
Technologie entwickelt sich aber stetig weiter und so steht eventuell schon der nächste Nachfolger mit dem Dienst AWS Fargate[41] bereit. Fargate wirbt mit dem Betreiben von Containern, ohne dass Server- oder Cluster-Management erforderlich ist. Mit anderen Worten: Fargate bringt die Zweckmäßigkeit und Skalierung von AWS Lambda in die Container-Welt.[42] Die in Punkt 2.4 erwähnten Restriktionen im Zusammenhang mit Runtimes würden damit wegfallen. Restriktionen im Zusammenhang mit Memory ebenso. Theoretisch bietet es also das Beste aus zwei Welten. Ob AWS Fargate dem auch in der Praxis entsprechen kann, wird die Zukunft zeigen.

[40] [Pyumal 2018]
[41] [AWS g]
[42] [Atchison 2018]

7 Literatur- und Quellenverzeichnis

[async] ASYNC: *Control Flow – waterfall*. https://caolan.github.io/async/docs.html#waterfall, Abruf am 13.07.2018.

[Atchison 2018] ATCHISON, Lee: *Forget AWS Lambda, so long Kubernetes – this is the future of serverless*. 2018, https://diginomica.com/2018/03/23/aws-lambda-kubernetes-future-serverless/, Abruf am 15.07.2018.

[Avram 2016] AVRAM, Abel: *FaaS, PaaS, and the Benefits of the Serverless Architecture.* 2016, https://www.infoq.com/news/2016/06/faas-serverless-architecture, Abruf am 15.07.2018.

[Auth0 a] AUTH0: *ID Token.* https://auth0.com/docs/tokens/id-token, Abruf am 15.07.2018.

[Auth0 b] AUTH0: *Introducing OIDC Conformant Authentication.* https://auth0.com/docs/api-auth/intro#delegation, Abruf am 15.07.2018.

[Auth0 c] AUTH0: *Lock Authentication Parameters.* https://auth0.com/docs/libraries/lock/v11/sending-authentication-parameters, Abruf am 15.07.2018.

[Auth0 d] AUTH0: *Lock v11 for Web.* 2017, https://auth0.com/docs/libraries/lock/v11, Abruf am 15.07.2018.

[AWS a] Amazon Web Services: *Access Control List (ACL) Overview.* https://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html, Abruf am 15.07.2018.

[AWS b] Amazon Web Services: *Amazon API Gateway.* https://aws.amazon.com/de/api-gateway/, Abruf am 15.07.2018.

[AWS c] Amazon Web Services: *Amazon API Gateway – Preise.* https://aws.amazon.com/de/api-gateway/pricing/, Abruf am 15.07.2018.

[AWS d] Amazon Web Services: *Analyse Ihrer Kosten mit Cost Explorer.* https://docs.aws.amazon.com/de_de/awsaccountbilling/latest/aboutv2/cost-explorer-what-is.html, Abruf am 15.07.2018.

[AWS e] Amazon Web Services: *Authenticating Requests: Browser-Based Uploads Using POST (AWS Signature Version 4).* https://docs.aws.amazon.com/de_de/AmazonS3/latest/API/sigv4-authentication-HTTPPOST.html, Abruf am 15.07.2018.

[AWS f] Amazon Web Services: *Authenticating Requests in Browser-Based Uploads Using POST (AWS Signature Version 4).* https://docs.aws.amazon.com/de_de/AmazonS3/latest/API/sigv4-UsingHTTPPOST.html, Abruf am 15.07.2018.

[AWS g] Amazon Web Services: *AWS Fargate.* https://aws.amazon.com/fargate/, Abruf am 15.07.2018.

[AWS h] Amazon Web Services: *AWS Lambda – Preise.* https://aws.amazon.com/de/lambda/pricing/, Abruf am 15.07.2018.

[AWS i] Amazon Web Services: *AWS Trusted Advisor.* https://aws.amazon.com/de/premiumsupport/trustedadvisor/, Abruf am 15.07.2018.

[AWS j] Amazon Web Services: *Erstellen eines Fakturierungsalarms zur Überwachung der geschätzten AWS-Gebühren.* https://docs.aws.amazon.com/de_de/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html, Abruf am 15.07.2018.

[AWS k] Amazon Web Services: *Das Context-Objekt (Java).* https://docs.aws.amazon.com/de_de/lambda/latest/dg/java-context-object.html, Abruf am 15.07.2018.

[AWS l] Amazon Web Services: *Discussion Forums - Details on Pricing.* 2016, https://forums.aws.amazon.com/thread.jspa?threadID=243745, Abruf am 15.07.2018.

[AWS m] Amazon Web Services: *Konfigurieren eines Buckets für das Website-Hosting.* https://docs.aws.amazon.com/de_de/AmazonS3/latest/dev/HowDoIWebsiteConfiguration.html, Abruf am 15.07.2018.

[AWS n] Amazon Web Services: *Verwenden von benutzerdefinierten API Gateway-Genehmigern.* https://docs.aws.amazon.com/de_de/apigateway/latest/developerguide/use-custom-authorizer.html, Abruf am 15.07.2018.

[AWS o] Amazon Web Services: *Was ist IAM?*. https://docs.aws.amazon.com/de_de/IAM/latest/UserGuide/introduction.html, Abruf am 15.07.2018.

[AWS p] Amazon Web Services: *Was ist Amazon Simple Notification Service?*. https://docs.aws.amazon.com/de_de/sns/latest/dg/welcome.html, Abruf am 15.07.2018.

[AWS q] Amazon Web Services: *Speziell entwickelte Datenbanken für sämtliche Anforderungen Ihrer Anwendungen.* https://aws.amazon.com/de/products/databases/, Abruf am 15.07.2018.

[Benetis 2017] BENETIS, Rimantas: *What comes after microservices? Serverless.* 2017, https://www.devbridge.com/articles/what-comes-after-microservices-serverless/, Abruf am 15.07.2018.

[Barr 2015] BARR, Jeff: *AWS Lambda – In Full Production with New Features for Mobile Devs.* 2015, https://aws.amazon.com/de/blogs/aws/aws-lambda-update-production-status-and-a-focus-on-mobile-apps/, Abruf am 15.07.2018.

[Decuyper 2017] DECUYPER, Xavier: *Static website hosting: who's fastest? AWS, Google, Firebase, Netlify or GitHub?.* 2017, https://www.savjee.be/2017/10/Static-website-hosting-who-is-fastest/, Abruf am 15.07.2018.

[Firebase] FIREBASE: *Firebase Realtime Database.* https://firebase.google.com/docs/database/, Abruf am 15.07.2018.

[JMBPSN 2015] JONES, M.; MICROSOFT; BRADLEY, J.; PING IDENTITY; SAKIMURA, N.; NRI: *JSON Web Token (JWT).* 2015, https://tools.ietf.org/html/rfc7519, Abruf am 15.07.2018.

[Kinley 2013] KINLEY, James: *The Lambda architecture: principles for architecting realtime Big Data systems.* 2013, http://jameskinley.tumblr.com/post/37398560534/the-lambda-architecture-principles-for, Abruf am 15.07.2018.

[KrBo 2016] KRYPCZYK, V.; BOCHKOR, O.: *Prototyping – Eine Einführung.* 2016, https://www.informatik-aktuell.de/entwicklung/methoden/prototyping-eine-einfuehrung.html, Abruf am 15.07.2018.

[Mytton 2017] MYTTON, David: *Who has the serverless advantage?.* 2017, https://read.acloud.guru/aws-lambda-vs-google-cloud-functions-vs-azure-functions-who-has-the-serverless-advantage-f6c2535e72f4, Abruf am 15.07.2018.

[Piyumal 2018] PYUMAL, Manjula: *Will serverless be the future of enterprise application development?.* 2018, https://medium.com/@manjulapiyumal/will-serverless-be-the-future-on-enterprice-application-development-9a1901798b0a, Abruf am 15.07.2018.

[Roberts 2016] ROBERTS, Mike: *Serverless Architectures.* 2016, https://martinfowler.com/articles/serverless.html, Abruf am 15.07.2018.

[Sbarski 2017] SBARSKI, Peter: *Serverless Architectures on AWS,* Manning, New York 2017.

[OAuth] OAUTH: *OAuth 2.0.* https://oauth.net/2/, Abruf am 15.07.2018.

[Wolf a] WOLF, Oliver: *Introduction into Microservices.* https://specify.io/concepts/microservices, Abruf am 15.07.2018.

[Wolf b] WOLF, Oliver: *Serverless Architecture in short.* https://specify.io/concepts/serverless-baas-faas, Abruf am 15.07.2018.

8 Abbildungsverzeichnis

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____

Anhang

Anhang 1: Convertimage

```
package de.julesfehr.prevelp.convertimage;

import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLDecoder;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.imageio.IIOImage;
import javax.imageio.ImageIO;
import javax.imageio.ImageWriteParam;
import javax.imageio.ImageWriter;
import javax.imageio.stream.ImageOutputStream;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.event.S3EventNotification.S3EventNotificationRecord;
import com.amazonaws.services.s3.model.ObjectMetadata;

public class ConvertImage implements RequestHandler<S3Event, String> {
    private final String JPG_TYPE = (String) "jpg";
    private final String JPG_MIME = (String) "image/jpeg";
    private final String PNG_TYPE = (String) "png";
    private final String PNG_MIME = (String) "image/png";

    AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

    public String handleRequest(S3Event s3event, Context context) {
        try {
            // Get the S3 notification record.
            S3EventNotificationRecord record = s3event.getRecords().get(0);

            // Extract the source bucket from the record.
            String srcBucket = record.getS3().getBucket().getName();
            // Object key may have spaces or unicode non-ASCII characters.
            String srcKey = record.getS3().getObject().getKey().replace('+', ' ');
            srcKey = URLDecoder.decode(srcKey, "UTF-8");


            // Specify bucket for the converted image.
            String dstBucket = "prevelp-image-converted";
            // Key is the name of the file.
            String dstKey = "converted-" + srcKey;

            // Sanity check: validate that source and destination are different
            buckets.
            if (srcBucket.equals(dstBucket)) {
```

```
                context.getLogger().log("Destination bucket must not match source
bucket.");
                return "";
            }

            // Infer the image type.
            Matcher matcher = Pattern.compile(".*\\.([^\\.]*)").matcher(srcKey);
            if (!matcher.matches()) {
                context.getLogger().log("Unable to infer image type for key " +
srcKey);
                return "";
            }
            String imageType = matcher.group(1);
            if (!(JPG_TYPE.equals(imageType)) && !(PNG_TYPE.equals(imageType))) {
                context.getLogger().log("Skipping non-image " + srcKey);
                return "";
            }

            // Download the image from S3 into a stream.
            InputStream objectData = s3Client.getObject(srcBucket,
srcKey).getObjectContent();

            // Read the source image.
            BufferedImage srcImage = ImageIO.read(objectData);

            // Reduce the image quality.
            reduceQuality(srcImage, imageType, srcKey);

            // Add Watermark.
            addTextWatermark("test", srcImage);

            // Re-encode image to target format.
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            ImageIO.write(srcImage, imageType, os);
            InputStream is = new ByteArrayInputStream(os.toByteArray());

            // Set Content-Length and Content-Type.
            ObjectMetadata meta = new ObjectMetadata();
            meta.setContentLength(os.size());
            if (JPG TYPE.equals(imageType)) {
                meta.setContentType(JPG_MIME);
            }
            if (PNG_TYPE.equals(imageType)) {
                meta.setContentType(PNG_MIME);
            }

            // Uploading to S3 destination bucket.
            context.getLogger().log("Writing to: " + dstBucket + "/" + dstKey);
            s3Client.putObject(dstBucket, dstKey, is, meta);
            context.getLogger().log("Successfully converted " + srcBucket + "/" +
srcKey + " and uploaded to " + dstBucket + "/" + dstKey);
            return "Ok";
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }

    /**
     * Embeds a textual watermark over a source image to produce
     * a watermarked one.
     * @param text The text to be embedded as watermark.
     * @param srcImage The source image file.
```

```
     */
    static BufferedImage addTextWatermark(String text, BufferedImage srcImage) {
        Graphics2D g2d = (Graphics2D) srcImage.getGraphics();

        // Initializes necessary graphic properties.
        AlphaComposite alphaChannel =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f);
        g2d.setComposite(alphaChannel);
        g2d.setColor(Color.BLUE);
        g2d.setFont(new Font("Arial", Font.BOLD, 64));
        FontMetrics fontMetrics = g2d.getFontMetrics();
        Rectangle2D rect = fontMetrics.getStringBounds(text, g2d);

        // Calculates the coordinates where the String is painted.
        int centerX = (srcImage.getWidth() - (int) rect.getWidth()) / 2;
        int centerY = srcImage.getHeight() / 2;

        // Paints the textual watermark.
        g2d.drawString(text, centerX, centerY);

        g2d.dispose();
        return srcImage;
    }

    static BufferedImage reduceQuality(BufferedImage srcImage, String imageType,
String srcKey) throws FileNotFoundException, IOException {
        ImageWriter imageWriter =
ImageIO.getImageWritersByFormatName(imageType).next();
        ImageWriteParam writeParam = imageWriter.getDefaultWriteParam();

        // This call is needed in order to explicitly set the compression level.
        writeParam.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);

        // 1.0f is max quality, min compression; 0.0f is min qual, max comp.
        writeParam.setCompressionQuality(0.1f);

        ByteArrayOutputStream os = new ByteArrayOutputStream();
        ImageOutputStream ios = ImageIO.createImageOutputStream(os);
        imageWriter.setOutput(ios);
        imageWriter.write(null, new IIOImage(srcImage, null, null), writeParam);
        imageWriter.dispose();
        InputStream is = new ByteArrayInputStream(os.toByteArray());
        srcImage = ImageIO.read(is);
        return srcImage;
    }
}
```

Anhang 2: SetPermissions

```
package de.julesfehr.prevelp.setpermissions;

import java.net.URLDecoder;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.s3.model.AccessControlList;
import com.amazonaws.services.s3.model.GroupGrantee;
import com.amazonaws.services.s3.model.Permission;
import com.google.gson.JsonArray;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

public class SetPermissions implements RequestHandler<SNSEvent, String> {

    AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();

    public String handleRequest(SNSEvent event, Context context) {
        try {

            // Get the SNS notification record.
            String record = event.getRecords().get(0).getSNS().getMessage();

            // Read the Json in a Json object to extract the bucket and key
            JsonElement jsonElement = new JsonParser().parse(record);
            JsonObject  jsonObject  = jsonElement.getAsJsonObject();
            JsonArray   jsonArray   = jsonObject.get("Records").getAsJsonArray();
                        jsonObject  =
jsonArray.get(0).getAsJsonObject().get("s3").getAsJsonObject();

            // Extract the bucket from the bucket json object.
            // Bucket name will have quotes when extracted from the json object
whicht will lead to it
            // not being recognized as a valid bucket name.
            String bucket =
jsonObject.get("bucket").getAsJsonObject().get("name").toString().replace("\"", "");

            // Extract the key from the json obect.
            // Object key may have spaces or unicode non-ASCII characters or quotes..
            String key =
jsonObject.get("object").getAsJsonObject().get("key").toString().replace("\"", "");
            // URLDecoder may be redundant because the key should have been decoded
in the ConvertImage function.
            key = URLDecoder.decode(key, "UTF-8");

            // Set the right ACL (make the file publicly accessible).
            AccessControlList acl = s3Client.getObjectAcl(bucket, key);

            // TODO later: add permissions only for the right users (customer should
only see his pictures, not all)
            // Example: acl.grantPermission(new
EmailAddressGrantee("xxx.xxx@gmail.com"), Permission.Read);
            acl.grantPermission(GroupGrantee.AllUsers, Permission.Read);
            s3Client.setObjectAcl(bucket, key, acl);

            return "ok";
```

```
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }
}
```

Anhang 3: get-upload-policy

```
'use strict';

var AWS = require('aws-sdk');
var async = require('async');
var Crypto = require('crypto-js');

var S3 = new AWS.S3();

function createErrorResponse(code, message) {
  var response = {
    'statusCode': code,
    'headers': {'Access-Control-Allow-Origin' : '*'},
    'body': JSON.stringify({'message': message})
  }
  return response;
}

function createSuccessResponse(message) {
  var response = {
    'statusCode': 200,
    'headers': {'Access-Control-Allow-Origin' : '*'},
    'body': JSON.stringify(message)
  }
  return response;
}

function base64encode (value) {
  return new Buffer(value).toString('base64');
}

function generateExpirationDate() {
  var currentDate = new Date();
  currentDate = currentDate.setDate(currentDate.getDate() + 1);
  return new Date(currentDate).toISOString;
}

function getSignatureKey(key, dateStamp, regionName, serviceName) {
  var kDate = Crypto.HmacSHA256(dateStamp, "AWS4" + key);
  var kRegion = Crypto.HmacSHA256(regionName, kDate);
  var kService = Crypto.HmacSHA256(serviceName, kRegion);
  var kSigning = Crypto.HmacSHA256("aws4_request", kService);
  return kSigning;
}

function generatePolicyDocument(filename, next) {
  var directory = crypto.randomBytes(20).toString('hex');
  var key = directory + '/' + filename;
  var expiration = generateExpirationDate();

  var policy = {
    'expiration': expiration,
    'conditions': [
      {key: key},
      {bucket: process.env.UPLOAD_BUCKET},
      {acl: 'private'},
      ['starts-with', '$Content-Type', '']
    ]
  };
  next(null, key, policy);
}
```

```
function encode(key, policy, next) {
  var encoding = base64encode(JSON.stringify(policy)).replace('\n', '');
  next(null, key, policy, encoding);
}

// function sign(key, policy, encoding, next) {
//   var signature = crypto.createHmac('sha1',
process.env.SECRET ACCESS KEY).update(encoding).digest('base64');
//   next(null, key, policy, encoding, signature);
// }


function sign(key, policy, encoding, next) {
  var signatureKey = getSignatureKey(process.env.SECRET_ACCESS_KEY,
moment().format('YYYYMMDD'), 'eu-central-1', 's3');
  var s3signature = Crypto.HmacSHA256(encoding,
signatureKey).toString(Crypto.enc.Hex);
  next(null, key, policy, encoding, s3signature);
}

exports.handler = function(event, context, callback) {
  var filename = null;

  if(event.queryStringParameters && event.queryStringParameters.filename) {
    filename = decodeURIComponent(event.queryStringParameters.filename);
  } else {
    callback(null, createErrorResponse(500, 'Filename must be provided'));
    return;
  }

  async.waterfall([async.apply(generatePolicyDocument, filename), encode, sign],
    function(err, key, policy, encoding, signature) {
      if(err) {
        callback(null, createErrorResponse(500, err));
      } else {
        var result =
        {
          signature: signature,
          encoded policy: encoding,
          access_key: process.env.ACCESS_KEY,
          upload_url: process.env.UPLOAD_URI + process.env.UPLOAD_BUCKET,
          key: key
        }
        callback(null, createSuccessResponse(result));
      }
    }
  )
}
```

Anhang 4: custom-authorizer

```
'use strict';

var jwt = require('jsonwebtoken');

var generatePolicy = function(principalId, effect, resource) {
  var authResponse = {};
  authResponse.principalId = principalId;
  if (effect && resource) {
    var policyDocument = {};
    policyDocument.Version = '2018-02-12';
    policyDocument.Statement = [];
    var statementOne = {};
    statementOne.Action = 'execute-api:Invoke';
    statementOne.Effect = effect;
    statementOne.Resource = resource;
    policyDocument.Statement[0] = statementOne;
    authResponse.policyDocument = policyDocument;
  }
  return authResponse;
}

exports.handler = function(event, context, callback) {
  if (!event.authorizationToken) {
    callback('Could not find authToken');
    return;
  }

  var token = event.authorizationToken.split(' ')[1];

  var secretBuffer = new Buffer(process.env.AUTH0_SECRET);
  jwt.verify(token, secretBuffer, function(err, decoded) {
    if (err) {
      console.log('Failed     jwt     verification:     ',     err,     'auth:     ',
event.authorizationToken);
      callback('Authorization Failed');
    } else {
      callback(null. generatePolicy('user', 'allow', event.methodArn));
    }
  })
}
```

Anhang 5: get-photo-list

```
'use strict';

var AWS = require('aws-sdk');
var async = require('async');

var S3 = new AWS.S3();

function createErrorResponse(code, message, encoding) {
  var response = {
    'statusCode': code,
    'headers': {'Access-Control-Allow-Origin' : '*'},
    'body': JSON.stringify({'code': code, 'message': message, 'encoding': encoding})
  }

  return response;
}

function createSuccessResponse(result) {
  var response = {
    'statusCode': 200,
    'headers': {'Access-Control-Allow-Origin' : '*'},
    'body': JSON.stringify(result)
  }

  return response;
}

function createBucketParams(next) {
  var params = {
    Bucket: process.env.BUCKET
  };

  next(null, params);
}

function getPhotosFromBucket(params, next) {
  s3.listObjects(params, function(err, data) {
    uf (err) {
      next(err);
    } else {
      next(null, data);
    }
  });
}

function createList(encoding, data, next) {
  var files = [];
  for (var i = 0; i < data.Contents.length; i++) {
    var file = data.Contents[i];

    if (encoding) {
      var type = file.Key.substr(file.Key.lastIndexOf('-') + 1);
      if (type !== encoding + '.mp4') {
        continue;
      }
    } else {
      if (file.Key.slice(-4) !== '.mp4') {
        continue;
      }
    }
```

```
    files.push({
      'filename': file.Key,
      'eTag': file.ETag.replace(/"/g, ""),
      'size': file.Size
    });
  }

  var result = {
    domain: process.env.BASE_URL,
    bucket: process.env.BUCKET,
    files: files
  }

  next(null, result);
}

exports.handler = function(event, context, callback) {
  var encoding = null;

  if (event.queryStringParameters && event.queryStringParameters.encoding) {
    encoding = decodeURIComponent(event.queryStringParameters.encoding);
  }

  async.waterfall([createBucketParams,  getVideosFromBucket,  async.apply(createList,
encoding)],
    function(err, result) {
      if (err) {
        callback(null, createErrorResponse(500, err, encoding));
      } else {
        if (result.files.length > 0) {
          callback(null, createSuccessResponse(result));
        } else {
          callback(null, createErrorResponse(404, 'No files were found', encoding));
        }
      }
    });
};
```