Erstellung eines Serverless Content-Upload-Systems für Hugo

12. Juli 2022

Leon.Rauschenbach@fom-net.de

Hochschulzentrum Köln

Praktische Seminararbeit

im Studiengang Informatik

im Rahmen der Lehrveranstaltung

Web Technologien

über das Thema

Erstellung eines Serverless Content-Upload-Systems für Hugo

von

Leon Rauschenbach

Betreuer :	Said Sulaiman Zaheby
Matrikelnummer :	556158
Abgabedatum :	12. Juli 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AWS	Amazon Web Services
CMS	Content Management System
FaaS	Function-as-a-Service
JWT	JSON Web Token
SaaS	Software-as-a-Service
WYSIWYG	What-You-See-Is-What-You-Get
XSS	Cross Site Scripting
YAML	Yet Another Markup Language

1 Einleitung

1.1 Motivation

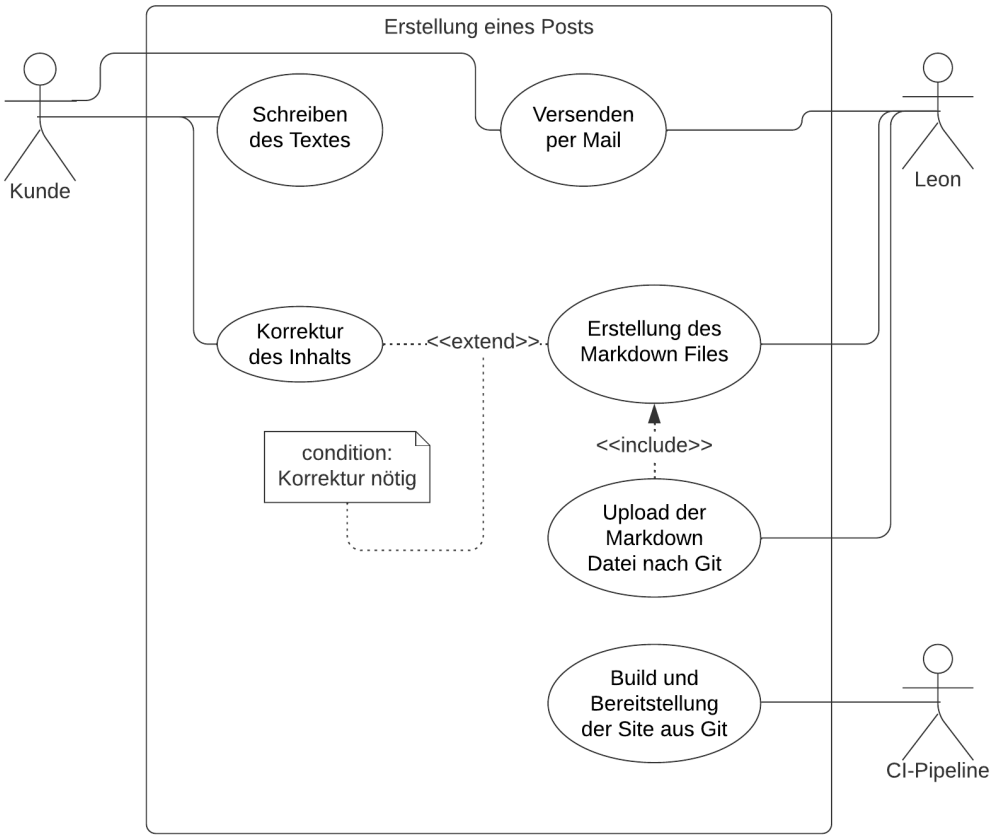
Als meine private Landing-Page hoste ich eine statische HTML-Seite. Diese wird mit dem Static-Site-Generator Hugo erstellt. Hugo generiert statische HTML und CSS Webseiten aus vordefinierten HTML-Templates und Markdown-Dateien. Diese Markdown-Dateien definieren die Inhalte und Metadaten der einzelnen HTML-Seiten [8].

Ich hoste für Freunde/private Kunden Web-Auftritte, welche ich mit Hugo realisiere. In diesen Webseiten ist teilweise eine Blog/Event-Funktion integriert [12]. Diese Blog-Einträge werden in Markdown geschrieben und werden genutzt um Rabattaktionen, Events oder Termine anzukündigen. Hugo ist mein Produkt der Wahl, weil sich extremst günstig hochperformante Webauftritte realisieren lassen. Außerdem müssen bei einer statischen Website keine Sicherheitsupdates wie bei z. B. WordPress eingespielt werden.

1.2 Problemstellung

Aktuell senden die Kunden den zu veröffentlichenden Text manuell an eine E-Mail-Adresse. Von da aus lade ich den Text herunter und formatiere ihn in einer Markdown-Datei. Diese Markdown-Datei wird in ein GitHub-Repository hochgeladen. Von da aus wird ein eine CI-Pipeline ausgelöst, welche den Generierung- und Deployment-Prozess der Website übernimmt.

Abbildung 1: Originaler Upload-Prozess



Quelle: Eigene Darstellung

Dies erfordert jedoch häufig Korrekturen auf meiner Seite. Ich muss den erhaltenen Inhalt manuell in eine Markdown-Datei überführen und die entsprechenden Metadaten wie Titel, Datum und Autor ergänzen. Ziel dieser Arbeit ist es, dass der Kunde eine bereits korrekt formatierte Markdown-Datei über eine Webapplikation hochladen kann. Diese Markdown-Datei soll auf einem leicht zugänglichen Speicherort (im Internet) abgelegt werden, wo Sie von einer CI-Pipeline dann in weiteren Schritten verarbeitet werden kann.

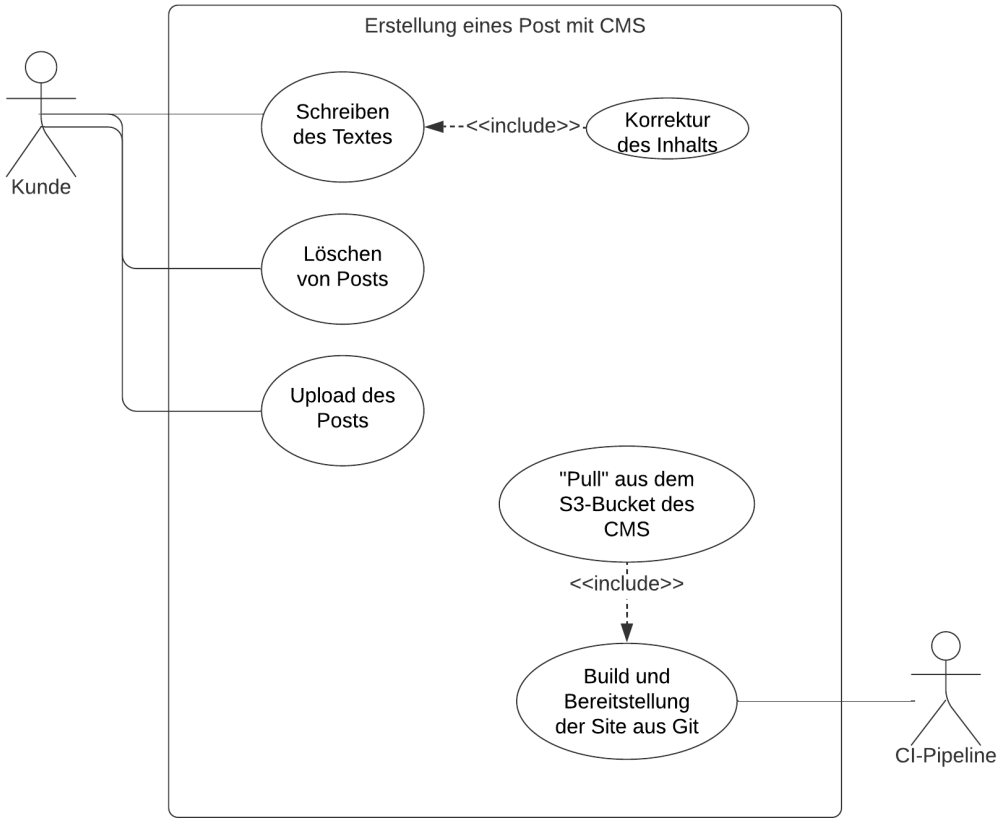
1.3 Lösungsansatz

Hierzu wird eine einfache Form eines Content Management System (CMS) programmiert. Der User wird in eine kleine Web-App eingewiesen, die er aus dem Internet aufrufen kann. Diese Web-App soll mit einem Backend interagieren, um eine vorausgefüllte Markdown-Datei in einem Speicherplatz zu erstellen.

Zur Realisierung soll eine *Serverless* oder auch *Function-as-a-Service (FaaS)* Backend gebaut werden, welches per HTTP von einem Frontend seine Befehle erhält. Das Backend

übernimmt das Parsing und korrekte Speichern der Markdown-Inhalte und ihrer Metadaten in einem Hugo-kompatiblen Format.

Abbildung 2: Neuer Upload-Prozess



Quelle: Eigene Darstellung

Durch diesen neuen Prozess wird die manuelle Arbeit meinerseits stark reduziert. Der Code findet sich im GitHub-Repository unter https://github.com/localleon/hugo-s3-cms. Die Anwendung ist für Firefox 99.0 (64-Bit) optimiert.

Folgende Zugangsdaten wurden für die Bewertung und Testung der Anwendung eingerichtet:

Abbildung 3: Login-Daten der Anwendung

```
Current Deployment in AWS-Cloud:

Frontend-APP: https://daq8101a8qkhh.cloudfront.net/index.html

backend-endpoints:
  POST - https://85tpt5asaa.execute-api.eu-central-1.amazonaws.com/upload
  GET - https://85tpt5asaa.execute-api.eu-central-1.amazonaws.com/list
  DELETE - https://85tpt5asaa.execute-api.eu-central-1.amazonaws.com/delete/{key}
  GET - https://85tpt5asaa.execute-api.eu-central-1.amazonaws.com/get/{key}

User-Accounts:

-------- USER 1 --------
Mail: professortest@lrau.xyz
Password: Y5V8qnrsarLW

-------- USER 2 --------
Mail: studenttest@lrau.xyz
Password: PNN4AK5MDxtc
```

Die Website ist für eine Bildschirmgröße von 1920x1080px und Firefox 99 optimiert. Der Anwendungsfall umfasst explizit keine Nutzer auf mobilen Endgeräten.

2 Technische Dokumentation

2.1 Systemaufbau

2.1.1 Tech Stack

Ziel des Projektes war es zunächst einen Prototyp zu entwerfen, welcher in weitere Arbeit verfeinert und verbessert werden konnte. Für grundlegende Funktionen wie HTTP-Routing, Authentifizierung und ein Storage-Backend sollte auf standardisierte Komponenten zurückgegriffen werden. Deswegen hab ich folgende Komponenten ausgewählt:

Python 3.8 wurde als Programmiersprache für das Backend des Projekts ausgewählt. Im Modul *Konzepte des skriptsprachenorientierten Programmierens* konnte ich mich tief in die Sprache einarbeiten und benutze diese auch auf meiner Arbeit im Kontext der Systemadministration. Die Sprache bietet eine hohe Entwicklerproduktivität (optimal für eine Prüfungsleistung) und umfangreiche Programmbibliotheken aus der Community. Die spezielle Version wird von *AWS Lambda* supportet und es existiert viel Dokumentation für die Nutzung in Amazon Web Services (AWS).

HTML, CSS und Javascript werden im Frontend eingesetzt. Ich habe mich explizit gegen einen JS-Framework wie *Vue.js* entschieden, um den Code möglichst leichtgewichtig zu

halten. Die normalen Features von JS reichen für diese Anwendung, da hier keine komplexe DOM-Manipulation nötig ist. Bei CSS wollte ich aber explizit einen leichtgewichtigen Framework einsetzen, um mir die Arbeit mit Grids und Styling zu erleichtern. *Milligram.css* bietet ein minimales Grundgerüst, um mit hoher Produktivität Frontends/Webseiten zu entwerfen [11].

Serverless ist ein Framework um FaaS Applikationen zu erstellen. Deployment und Konfiguration der Applikation werden in Yet Another Markup Language (YAML) vorgenommen (Infrastructure-as-Code Prinzip). Ein Framework ist nötig um lokal die Funktion testen zu können [9]. Alternativen sind *python-lambda* [10] und *AWS SAM* [7], welche mir aber zu unhandlich waren.

Auth0 ist ein Software-as-a-Service (SaaS) Provider für Authentifizierungslösungen. Mithilfe der SDK lassen sich sehr schnell sehr sicherer Authentifizierungslösungen und Login-Masken bauen. Rechtevergabe für einzelne Dienste, User-Konten oder Anbindungen an andere OpenID-Provider sind möglich [5].

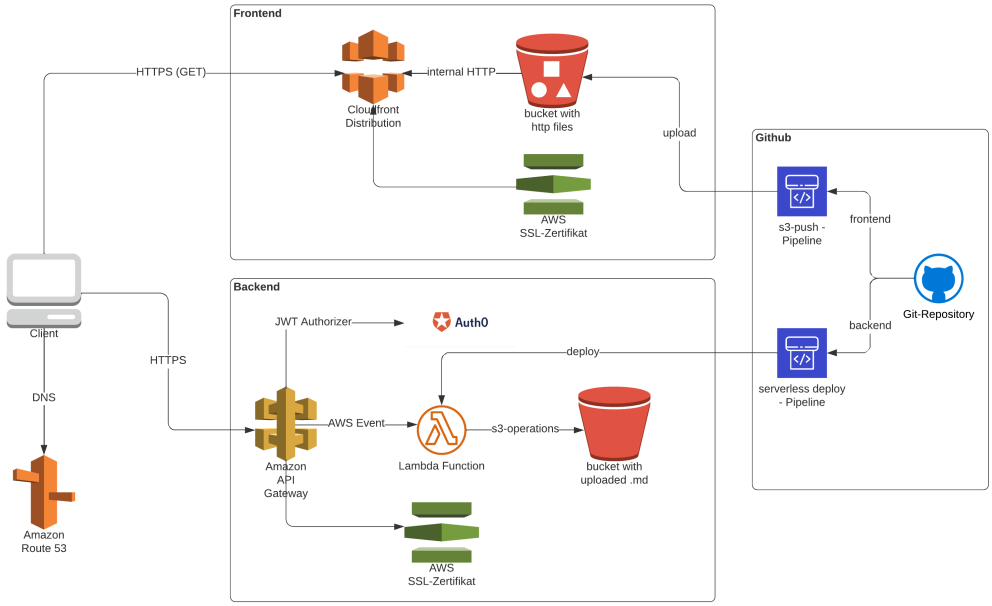
AWS wurde als Cloud/Hosting Anbieter ausgewählt. Als Marktführer bietet AWS die breiteste Palette an Funktionsangeboten.

- Api Gateway (HTTP) [1]
- Lambda Functions (FaaS) [3]
- S3 Object Storage [4]
- Cloudfront CDN [2]

2.1.2 CMS Architektur

Die Web-Applikation wird vollständig in der AWS-Cloud realisiert. Der Benutzer interagiert mit der Web-Applikation über ein Frontend in seinem Browser. Das Frontend besteht aus einer Single-Page-Applikation, die Authentifizierung und Interaktion mit dem Backend übernimmt.

Abbildung 4: CMS Backend in AWS



Quelle: Eigene Darstellung

Über das Backend lassen sich per HTTP-Requests Posts hochladen, anzeigen und löschen. Die HTTP-Requests werden durch ein *API-Gateway* an eine Python-Funktion (Lambda) weitergeleitet, welche die Requests bearbeitet. Die *Lambda-Funktion* nutzt im Hintergrund S3 Objekt-Storage, um die Posts zu speichern.

Der Deployment-Prozess des Frontends ist komplett automatisiert. Bei jedem Commit ins Remote-Repository wird die Applikation in AWS hochgeladen (per Github Action). Das Deployment der Lambda-Funktion erfolgt für bessere lokale Testbarkeit aktuell manuell mit *serverless deploy*.

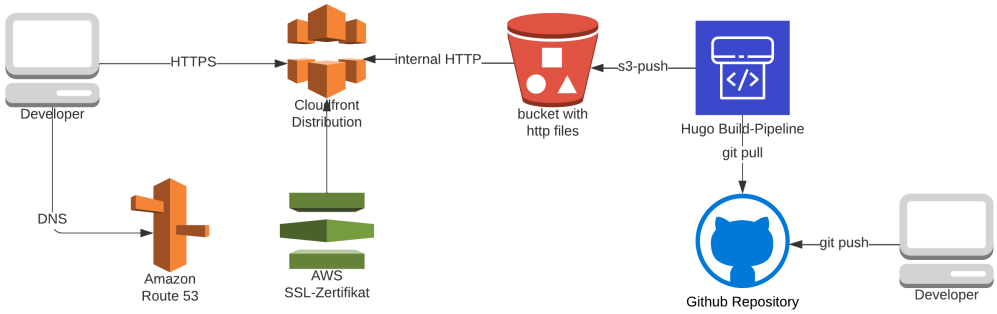
Zur lokalen Entwicklung nutze ich *Visual Studio Code* in Kombination mit einem lokalen HTTP-Server, Python-3.8 Installation und dem Serverless-Framework. Ein lokaler HTTP-Server mit Live-Reload Funktionalität ermöglicht die schnelle Anpassung des Frontends. Die Entwicklung und das Testing der API kann mit dem *Serverless-Framework* lokal getestet werden. Dazu kann mit dem Befehl *serverless invoke -f* die Lambda-Funktion lokal ausgeführt werden und mit entsprechenden Events (HTTP-Requests) getestet werden.

2.1.3 Webseiten Architektur

Die bereits bestehenden statischen Webseiten werden komplett in AWS gehostet. Diese Websites bestehen aus Template und Markdown-Dateien in einem Github-Repository.

Diese werden mithilfe von Hugo (Static-Site-Generator) in einer Build-Pipeline kombiniert. Die Komponenten arbeiten wie folgt miteinander:

Abbildung 5: Hosting einer Webseite in AWS



Quelle: Eigene Darstellung

Über Cloudfront wird SSL-Termination, DDos-Schutz und Caching der Website gesteuert. Der Browser interagiert hier zentral per HTTPS mit der Website. Im Hintergrund lädt die Cloudfront-Distribution die Objekte (Dateien) per HTTP aus einem S3-Bucket. Namensauflösung des Domain-Namens werden über das DNS System Route53 von AWS aufgelöst.

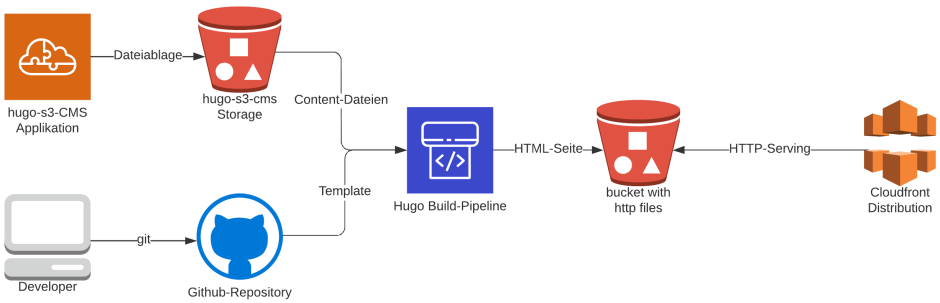
Nach jedem *git push* wird eine CI-Pipeline getriggert, welcher die Seite mithilfe von *hugo* generiert und per AWS-CLI in den S3-Bucket hochlädt. Somit ist der Deployment-Prozess ab dem Git-Repository bereits vollständig automatisiert. In dieser GitHub-Action wird beispielsweise der Content aus dem Hugo-S3-CMS in den Build-Prozess einer Website eingebunden.

Abbildung 6: CI-Pipeline des Deployments

```yaml
name: Publish to s3 fmt-web-test

on:
  push:
    branches:
      - main

jobs:
  deploy:
    name: Upload to Amazon S3
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Setup Hugo
        uses: peaceiris/actions-hugo@v2
        with:
          hugo-version: "0.84.4"

      - name: Configure AWS credentials from Test account
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: eu-central-1

      - name: Copy Markdown Files from CMS to Build-Directory -> content contains the
          input file for the hugo generation
        run: |
          aws s3 sync ./content s3://CMS_BUCKET/auth0|USERNUMBER --acl public-read --
              delete

      - name: Build Website
        run: |
          hugo --minify

      - name: Copy files to web-hosting s3 storage -> public folder contains the
          generated html
        run: |
          aws s3 sync ./public s3://WEBSITE_BUCKET --acl public-read --delete
```

Das CMS agiert als eigenständiges System. Um das CMS mit den entsprechenden Webseiten zu verbinden, muss nur ein Dateitransfer in der Build-Pipeline aus dem CMS integriert werden.

Abbildung 7: Interaktion zwischen CMS und Webseiten



Quelle: Eigene Darstellung

Die Arbeit konzentriert sich nur auf die Erstellung des CMS und behandelt nicht die vollständige Umsetzung dieses Build-Prozesses.

2.1.4 Unit-Tests

Mithilfe des Testing-Frameworks *pytest* und dem Mocking-Framework *moto* können Unit-Tests auf vorgetäuschter AWS Infrastruktur ausgeführt werden.

Abbildung 8: Ausführung der Unit-Tests

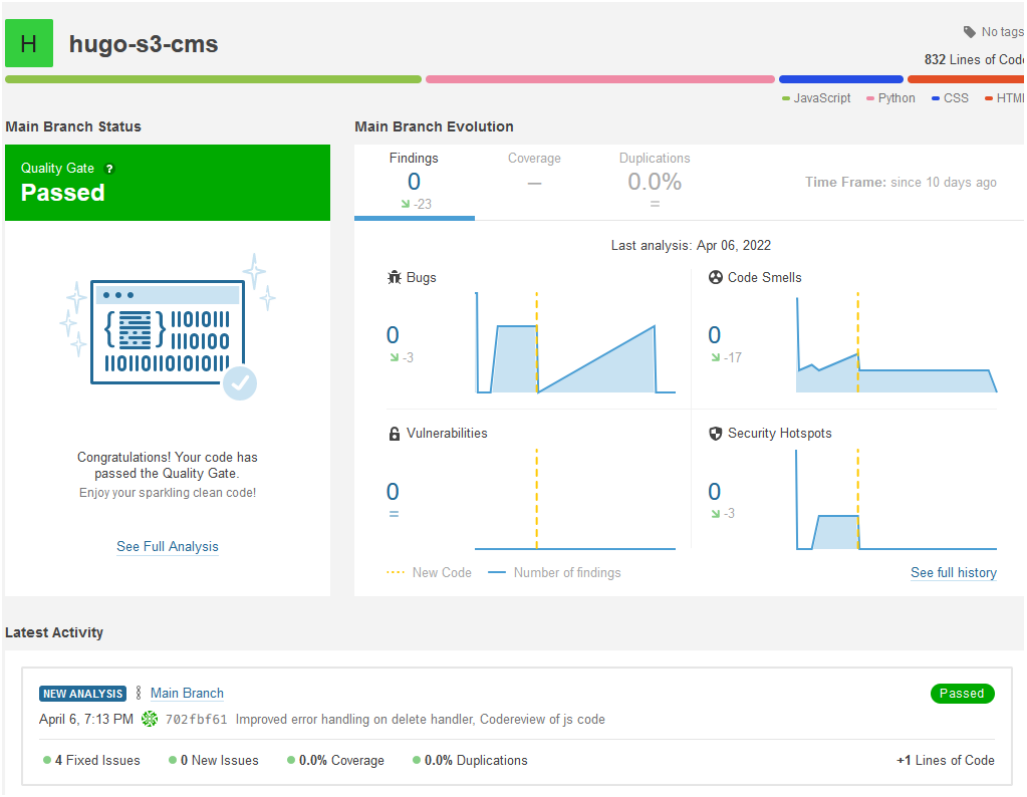
```
lraus@lraus-desk0: ~/python/hugo-cms-backend/hugo-s3-cms/backend
(env) lraus@lraus-desk0:~/python/hugo-cms-backend/hugo-s3-cms/backend$ ls
__pycache__  handler.py    package-lock.json  requirements-dev.txt  sample_events   test_handler.py
env          node_modules  package.json       requirements.txt      serverless.yml
(env) lraus@lraus-desk0:~/python/hugo-cms-backend/hugo-s3-cms/backend$ pytest -v
============================================= test session starts ==============================================
platform linux -- Python 3.8.10, pytest-7.1.1, pluggy-1.0.0 -- /home/lraus/python/hugo-cms-backend/hugo-s3-cms/backend/e
nv/bin/python
cachedir: .pytest_cache
rootdir: /home/lraus/python/hugo-cms-backend/hugo-s3-cms/backend
plugins: cov-3.0.0
collected 10 items

test_handler.py::test_callback PASSED                                                                    [ 10%]
test_handler.py::test_validate_post_json PASSED                                                          [ 20%]
test_handler.py::test_get_filename_from_post PASSED                                                      [ 30%]
test_handler.py::test_get_body_from_event PASSED                                                         [ 40%]
test_handler.py::test_calc_paging_index PASSED                                                           [ 50%]
test_handler.py::test_list_objects_from_bucket_paged_size_only PASSED                                    [ 60%]
test_handler.py::test_list_objects_from_bucket PASSED                                                    [ 70%]
test_handler.py::test_check_correct_user_dir_permissions PASSED                                          [ 80%]
test_handler.py::test_get_file_from_s3 PASSED                                                            [ 90%]
test_handler.py::test_illegal_s3_keys_allowed PASSED                                                     [100%]

============================================== 10 passed in 0.58s ==============================================
(env) lraus@lraus-desk0:~/python/hugo-cms-backend/hugo-s3-cms/backend$
```

Quelle: Eigene Darstellung

In den Unit-Tests werden die grundlegenden Funktionen zur Transformation von Daten, Überprüfung von Werten und simple Dateioperationen der Funktionen getestet. Aufgrund

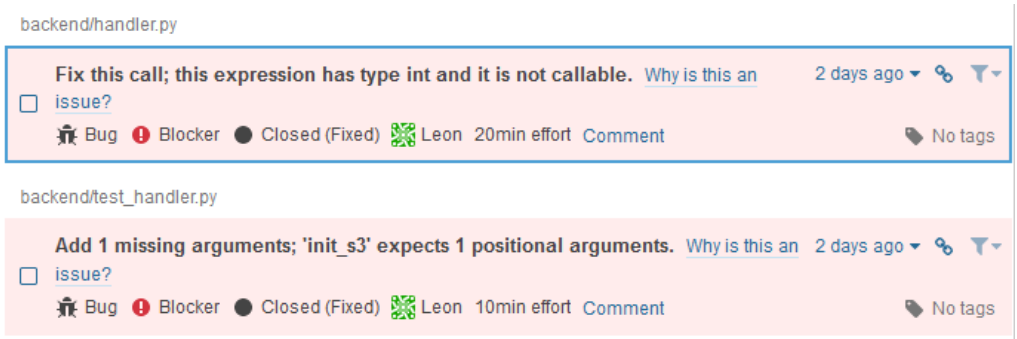
der begrenzten Zeit für die Projektarbeit, war es mir leider nicht möglich den gesamten Codes mit Unit-Tests zu versehen. Innerhalb des Repository werden über GitHub-Action (im Ordner .github/workflows) automatisiert die Unit-Tests ausgeführt.

Abbildung 9: Ausführung der Unit-Tests



Quelle: Eigene Darstellung

Abbildung 10: Empfehlungen/Reports aus Sonarcloud



Quelle: Eigene Darstellung

Gleichzeitig wird über Sonarcloud.io eine statische Code-Analyse des Quellcodes durchgeführt. Hier wird der Code (JS, Python und HTML) mit Regel auf *Code Smells, Anitpattern*

und Secrets geprüft. Konstantes Refactoring wird auf der Basis dieser Analysen durchgeführt. Durch die automatisierten Tests und Code-Analysis gibt es ein konstantes Feedback zur Entwicklung der Code-Qualität [13].

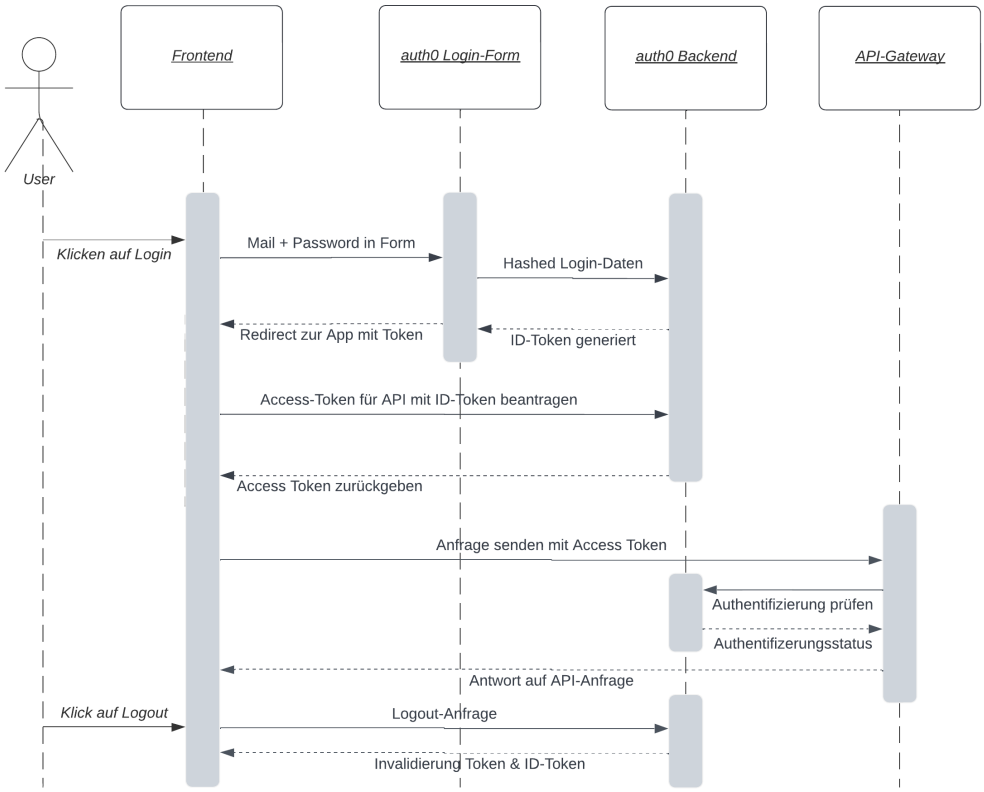
2.2 Zentrale Funktionen

2.2.1 Authentifizierung

Die User-Authentifizierung wird durch JSON Web Tokens realisiert [6]. Mithilfe der SDK des SaaS Providers *auth0* habe ich eine Login-Form mit vollständigem User-Backend implementiert. Die Anwendung lässt sich dadurch durch beliebig viele Nutzer (aktuell zwei konfiguriert, Registrierung deaktiviert) mit eigenständigen Posts nutzen.

Beim Login leitet das Frontend auf eine Login-Form weiter, in welcher korrekte Benutzerdaten eingegeben werden müssen. Bei erfolgreicher Authentifizierung durch E-Mail und Passwort erhält der User einen JSON Web Token (JWT) ID-Token. In diesem ID-Token sind die Metadaten des Nutzers (User ID, E-Mail) hinterlegt. Der Token ist für eine bestimmte Zeit gültig und wird automatisch erneuert. Mithilfe dieses ID-Tokens beantragt das Frontend beim auth0-Backend nun einen speziellen Token der Zugriff auf die Backend-API gewährt.

Abbildung 11: Austausch der JWT-Token zwischen Frontend und Auth-Provider

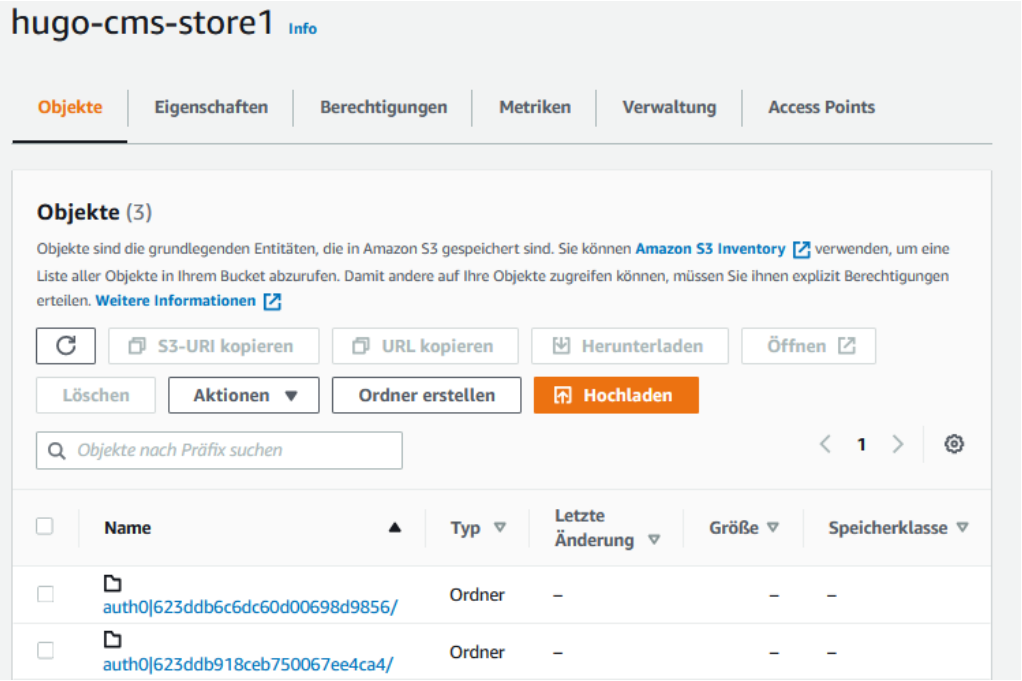


Quelle: Eigene Darstellung

Die JWT-Token Verifizierung der API wird vollständig vom API-Gateway übernommen. Dies kommuniziert über eine JWT-Integration im Hintergrund mit *auth0*. Für die Anwendung ist das komplett transparent und sie erhält nur authentifizierte Requests.

Das Backend verifiziert anhand des Claims (der Berechtigungen im JWT-Token) auf welche User-Objekte der Nutzer zugreifen kann. Die Zuordnung von Objekten zu Nutzern wird mit einem Präfix im Objekt-Storage gelöst. Das Backend dekodiert aus dem Token die Nutzer-ID und nutzt diese als Präfix für sämtliche S3 Operationen. Somit wird dann ein Objekt namens *2020-02-03_Post1.md* im Format *USER-ID/2020-02-03_Post1.md* abgelegt. S3 kennt keine Pfade, sondern nur Dateinamen und Präfixe (ähnlich zu Ordnern).

Abbildung 12: Realisierung der User-Directors im S3-Bucket



Quelle: Eigene Darstellung

2.2.2 Post erstellen

Mithilfe der Web-Anwendung können die User einen Post erstellen. Ein Post enthält Titel, Datum, Autor und Inhalt. Der Inhalt eines Posts kann in Markdown geschrieben werden. Hierzu wird direkt in der Web-Anwendung während des Schreibens ein Preview des Markdown-Inhalts angezeigt, um den User einen Eindruck vom späteren Ergebnis zu schaffen.

Abbildung 13: Generierte Datei eines Posts im S3-Storage

```
---
title: Strand Festival 2022
date: 2022-03-01
author: Party-Phillip
draft: false
---
Zusammen Sommer feiern!

Beste Beats - Musik gibts von Newcomer DJs aller elektronischen Genre aus der Region.
    Bei uns läuft alles was Spaß macht!

Kleiner aber fein - Da das Strand-Festival ein kleines Festival ist, sind die Wege
    kurz und es bleibt mehr Zeit zum Feiern! Auch die Campsite befindet sich direkt
    vor dem Festival-Eingang.

Festival zum Partypreis - Wir buchen keine überteuerten, internationalen Acts, fahren
    keine riesige Marketingkampagnen und uns reicht der See als Kulisse völlig aus.
    Deshalb können wir euch unsere Tickets zu einem super fairen Preis anbieten!

---
```

Das Frontend lässt den User die Post-Felder ausfüllen und schickt eine HTTP-Request (/upload/) an das Backend. Das Backend überprüft die Daten auf Korrektheit und erstellt einen Dateinamen für das Datenobjekt aus dem Titel. Der Post-Inhalt wird in einer Markdown-Datei mit Metadaten auf dem Storage abgelegt.

Abbildung 14: /upload API-Request

```
Request:

POST /upload HTTP/1.1
Host: 85tpt5asaa.execute-api.eu-central-1.amazonaws.com
Accept: */*
Content-Type: application/json
Authorization: Bearer
    eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IkdHeHJEUXh4YUJRM2xLczFiSS1YZyJ9.
    eyJpc3MiOiJodHRwczovL2xvY2FsbGVvbi5ldS5hdXRoMC5jb20vIiwic3ViIjoiYXV0aDB8NjIzZGRiNmM2ZGM2MGQwMDY5
    .F2aeRweoSEtxUmh9RCp4NHa3lDVtdA4pO-
    WJWFehvnsqTbVVQW80reTnK97Hu9Kp215c2bNbnC8MchfzKuFn6cZLpgElcf9lWr2vDWQ4TzaAh3Yo0TRoYN7hnZhObDNhoM
    -cenPt6kBeFhPvEzQ_236FdfcEE8qBQKzU9aeJN37b1ZNPat57GvKQla7YOaZ90XTC-
    kRqBPIx0XPn6OlnS1sdy7wXvuflIyZFfBdVnFiCDdZS9SunHd6cgq55axJylVWZ0i5RYAplcKWbNW37MeJrz9uCuDjCCLJPf


{"title":"Neujahresfest","date":"2022-04-13","author":"Rauschenberg","content":"
    Palimpalim,\n\nich lade zum diesjährigen Neujahresfest ein. \n\nHerzlich
    willkommen "}

Response:

HTTP/2 200 OK
date: Wed, 06 Apr 2022 08:37:13 GMT
content-type: text/plain; charset=utf-8
content-length: 40
access-control-allow-origin: *
apigw-requestid: QJmg-hFEFiAEM2w=

{"msg": "Post was created successfully"}
```

Nach dem Hochladen des Posts über die Web-Anwendung/API, liegt die Datei zur Veröffentlichung bereit und kann vom Admin im S3-Storage eingesehen und weiterverarbeitet werden. Diese Markdown Datei kann nun z. B. von einer CI-Pipeline in den Build-Prozess der Hugo-Webseite kopiert werden, um den Post in der generierten Webseite anzuzeigen.

2.2.3 Post löschen

Um versehentliche erstellte oder fehlerhafte Posts zu löschen, können Dateien aus dem Backend gelöscht werden. Der User kann diese Löschanfrage über den entsprechenden Knopf im Frontend auslösen.

Abbildung 15: /delete API-Request

```
Request

DELETE /delete/2022-04-06_Trashpost.md HTTP/1.1
Host: 85tpt5asaa.execute-api.eu-central-1.amazonaws.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:98.0) Gecko/20100101 Firefox
    /98.0
Accept: */*
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate, br
Authorization: Bearer
    eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IkdHeHJEUXh4YUJRM2xLczFiSS1YZyJ9.
    eyJpc3MiOiJodHRwczovL2xvY2FsbGVvbi5ldS5hdXRoMC5jb20vIiwic3ViIjoiYXV0aDB8NjIzZGRiNmM2ZGM2MGQwMDY5
    .F2aeRweoSEtxUmh9RCp4NHa3lDVtdA4pO-
    WJWFehvnsqTbVVQW80reTnK97Hu9Kp215c2bNbnC8MchfzKuFn6cZLpgElcf9lWr2vDWQ4TzaAh3Yo0TRoYN7hnZhObDNhoM
    -cenPt6kBeFhPvEzQ_236FdfcEE8qBQKzU9aeJN37b1ZNPat57GvKQla7YOaZ90XTC-
    kRqBPIx0XPn6OlnS1sdy7wXvuflIyZFfBdVnFiCDdZS9SunHd6cgq55axJylVWZ0i5RYAplcKWbNW37MeJrz9uCuDjCCLJPf

Content-Type: application/json


Response:

HTTP/2 200 Accepted
date: Wed, 06 Apr 2022 08:39:52 GMT
content-type: text/plain; charset=utf-8
content-length: 65
access-control-allow-origin: *
apigw-requestid: QJm5zj73liAEMxQ=
X-Firefox-Spdy: h2

{"msg": "Successfully deleted your file!"}
```

Das Backend verarbeitet die HTTP-Request und führt eine entsprechende Löschoperation auf dem S3-Storage aus. Löschoperationen werden mithilfe des JWT-Tokens den entsprechenden User-Verzeichnissen zugeordnet.

2.2.4 Post anzeigen

Der User kann alle eigenen Posts im Frontend anzeigen. Dazu wird eine HTTP-Request an das Backend gesendet.

Abbildung 16: /get API-Request

```
Request:

GET /get/2022-04-05_Flohmarkt.md HTTP/1.1
Host: 85tpt5asaa.execute-api.eu-central-1.amazonaws.com
Accept: */*
Content-Type: application/json
Authorization: Bearer
    eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IkdHeHJEUXh4YUJRM2xLczFiSS1YZyJ9.
    eyJpc3MiOiJodHRwczovL2xvY2FsbGVvbi5ldS5hdXRoMC5jb20vIiwic3ViIjoiYXV0aDB8NjIzZGRiNmM2ZGM2MGQwMDY5
    .F2aeRweoSEtxUmh9RCp4NHa3lDVtdA4pO-
    WJWFehvnsqTbVVQW80reTnK97Hu9Kp215c2bNbnC8MchfzKuFn6cZLpgElcf9lWr2vDWQ4TzaAh3Yo0TRoYN7hnZhObDNhoM
    -cenPt6kBeFhPvEzQ_236FdfcEE8qBQKzU9aeJN37b1ZNPat57GvKQla7YOaZ90XTC-
    kRqBPIx0XPn6OlnS1sdy7wXvuflIyZFfBdVnFiCDdZS9SunHd6cgq55axJylVWZ0i5RYAplcKWbNW37MeJrz9uCuDjCCLJPf


Response:

HTTP/2 202 Accepted
date: Wed, 06 Apr 2022 08:42:56 GMT
content-type: text/plain; charset=utf-8
content-length: 308
access-control-allow-origin: *
apigw-requestid: QJnWmg-oFiAEMGA=

{"body": "
    LS0tCnRpdGxlOiBGbG9obWFya3QgCmRhdGU6IDIwMjItMDQtMjkKYXV0aG9yOiBMZW9uCmRyYWZ0OiBmYWxzZQotLS0KIyBG
    "}
```

Dabei empfängt der Endpunkt einen Key, sucht dann mithilfe des JWT-Tokens nach dem korrekten User-Pfad und gibt die entsprechende Datei zurück. Der Text wird Base64 kodiert übertragen.

2.3 Nutzung

Die folgenden Schritte beschreiben eine typische Interaktion eines Nutzers, welche die wichtigsten Funktionen der Anwendung nutzen:

1. Der User klickt auf den Login-Button in der Navigationsleiste und wird auf die Login-Form weitergeleitet. Dort gibt er die erhaltenen Zugangsdaten ein.

2. Der Nutzer schreibt einen Post in der Web-Applikation im Textfeld und füllt alle Metadaten aus. Dann überprüft er seine Eingaben im Preview-Feld daneben und drückt auf den Knopf *Submit* um den Post ins Backend hochzuladen.

3. Der Nutzer scrollt in die Sektion *Manage Posts* und drückt auf der ausgewählten Seite (wechselbar durch die Pagination-Knöpfe) das *Auge-Symbol* aus, um seinen Post anzuzeigen. Eine Markdown-Preview seines hochgeladenen Inhalts wird ihm in einer Preview-Box unter den Auswahlelementen angezeigt.

4. Der Nutzer kann auf das *Papierkorb-Symbol* klicken, um den Post zu löschen, falls er ihm nicht gefällt. Mit Klick auf das Auge erscheint ein Pop-up zur Bestätigung. Nach Bestätigung wird der Post im Backend unwiderruflich gelöscht.

5. Der User klickt den *Logout-Knopf* in der Navigationsleiste, um sich von der Anwendung abzumelden. Das Frontend invalidiert alle Tokens und beendet die Session

Diese Grundfunktionen ermögliche mehreren Nutzern unabhängig voneinander Markdown-Dateien zu erstellen und für die weitere Verwendung durch Build-Pipelines bereitzustellen.

3 Fazit

3.1 Herausforderungen

Durch die Realisierung der Applikation in der Cloud musste ich mich mit vielen neuen Technologien und Providern beschäftigen. Ich wollte möglichst wenig Code schreiben, der nicht mit der *Business Logik* der Applikation zusammenhängt. Dies hat jedoch sehr viel Zeit und kleinere separate Prototypen benötigt, um mit den entsprechenden SDK und API der Provider effektiv zu arbeiten.

Ich habe mich aufgrund der Umsetzung als *AWS Lambda Funktion* gegen einen objektorientierten Programmieransatz entschieden. Lambda Funktionen sind natürlicherweise eher atomare Codeschnipsel als große zusammenhängende Klassenkonstrukte. Durch funktionale Programmierung konnte ich in Python sehr viel Codezeilen einsparen und das Backend sehr kompakt halten (Single-File). Sollte das Projekt um weitere Features erweitert werden, würde ich über das API-Gateway verschiedene Lambda-Funktionen für weitere Endpunkte aufrufen, um den Code und die Ausführung weiter zu separieren und zu optimieren.

In der ersten Version der Anwendung fand sich außerdem eine Cross Site Scripting (XSS) Lücke. Im Prototypen gab es noch keine separaten User-Speicher und die eingegebenen Posts konnten beliebigen Text und Code enthalten. Beim Rendern der Previews, kam es dann zur Ausführung des Codes aus dem Post im Browsern.

Abbildung 17: Ausnutzung der XSS-Lücke in der Anwendung

```
---
title: XSS-Test
date: 2022-03-30
author: Hacker
draft: false
---
# XSS Try

&lt;IMG SRC=/ onerror="alert(String.fromCharCode(88,83,83))"&gt;&lt;/img&gt;
---
```

Die Sicherheitslücke wurde behoben, in dem Code zum Bereinigen (Input Sanitization) des Posts von ausführbarem Code eingeführt wurde. Serverseitig werden alle Felder des Posts einmal bereinigt und dann erst abgespeichert. Im Frontend wird vor dem Rendern der Inhalt vom Server ein zweites Mal bereinigt.

Ein Paging, also eine Seitenfunktion, wurde für das Anzeigen der Post-Objekte eingeführt, um die Daten in den HTTP-Requests und angezeigten Daten im Frontend zu begrenzen. Die Darstellung von mehr als 8 Elementen (Usability Regel: maximal 7+2 angezeigt Elemente kann ein Mensch verarbeiten) gestaltete sich als schwierig.

3.2 Umsetzungserfolg

Mithilfe der Web-App werden die Anforderungen vollständig erfüllt. Benutzer können simple Posts in Textform verfassen und in einem zentralen Storage den Administratoren oder Build-Pipelines bereitstellen. Das System erfüllt seinen Zweck, ohne in die Architektur von anderen Webseiten einzugreifen. Benutzer müssen immer noch grundlegende Markdown-Befehle erlernen, aber durch das Preview und das automatische hochladen wurde der Prozess stark vereinfacht.

Mithilfe der Cloud-Native Technologie und den SaaS-Providern erzielte ich schnell Fortschritte mit wenig Code. Die Integration und Verknüpfung der Pipelines hat mir besonders Spaß bereitet. Das Hosting der Anwendung ist außerdem sehr günstig und kostet ohne Domain etwa 10-20 Cent bei moderater Nutzung, da nur für tatsächlich genutzte Ressourcen bezahlt wird.

3.3 Mögliche Erweiterungen

Die Anwendung könnte man noch um einen What-You-See-Is-What-You-Get (WYSIWYG) Editor und eine Bild-Upload-Funktion erweitert werde. So können multimediale Inhal-

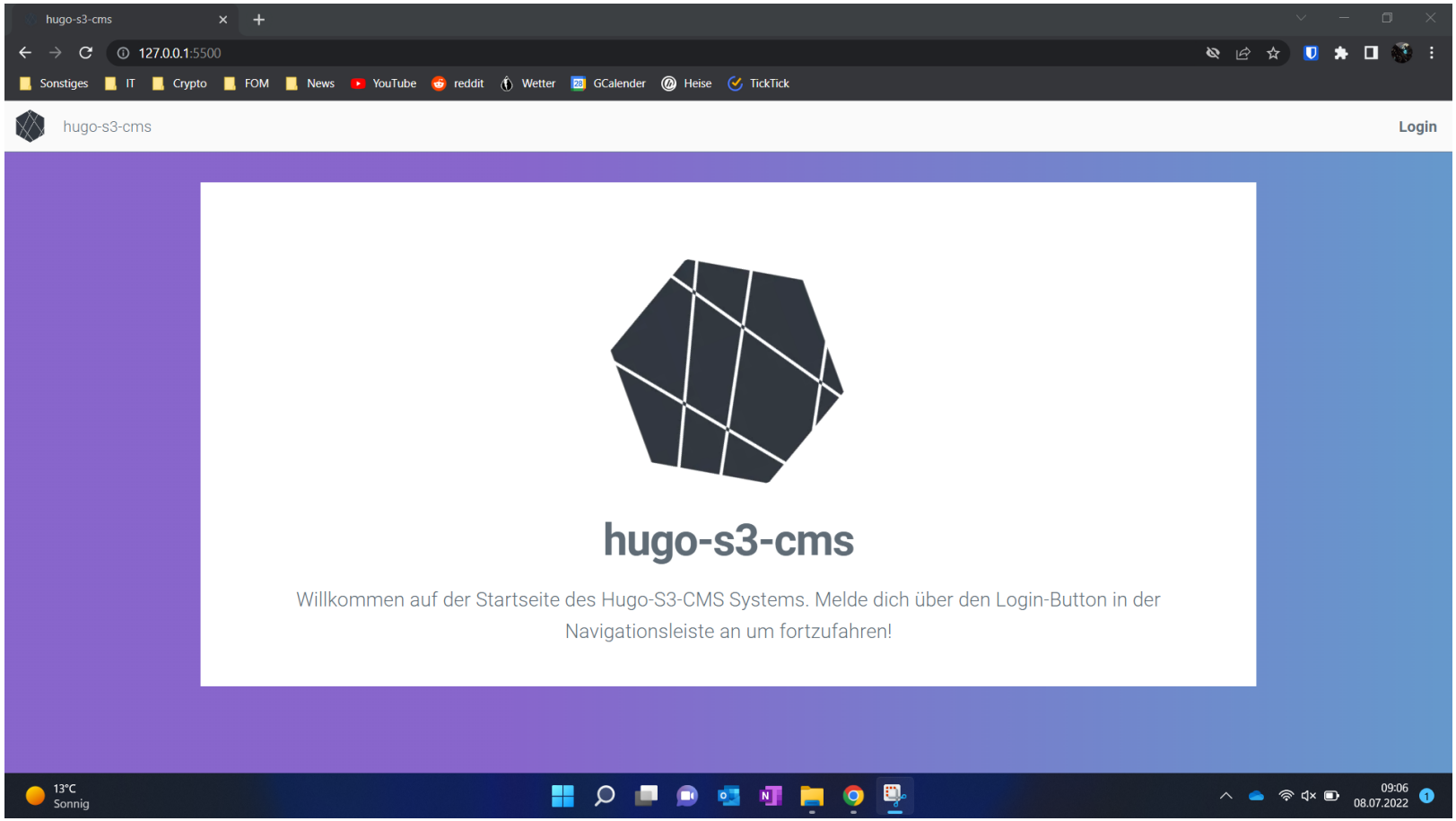
te auch über das CMS erstellt werden. Dies würde jedoch eine weitere Funktion zum Hochladen von Binärdaten und eine Einbindung oder Entwicklung eines browserbasierten Markdown-Editors benötigen. Mit diesen Funktionen könnte der Prozess noch nutzerfreundlich gestaltet werden.

Anhang

Layout der Website

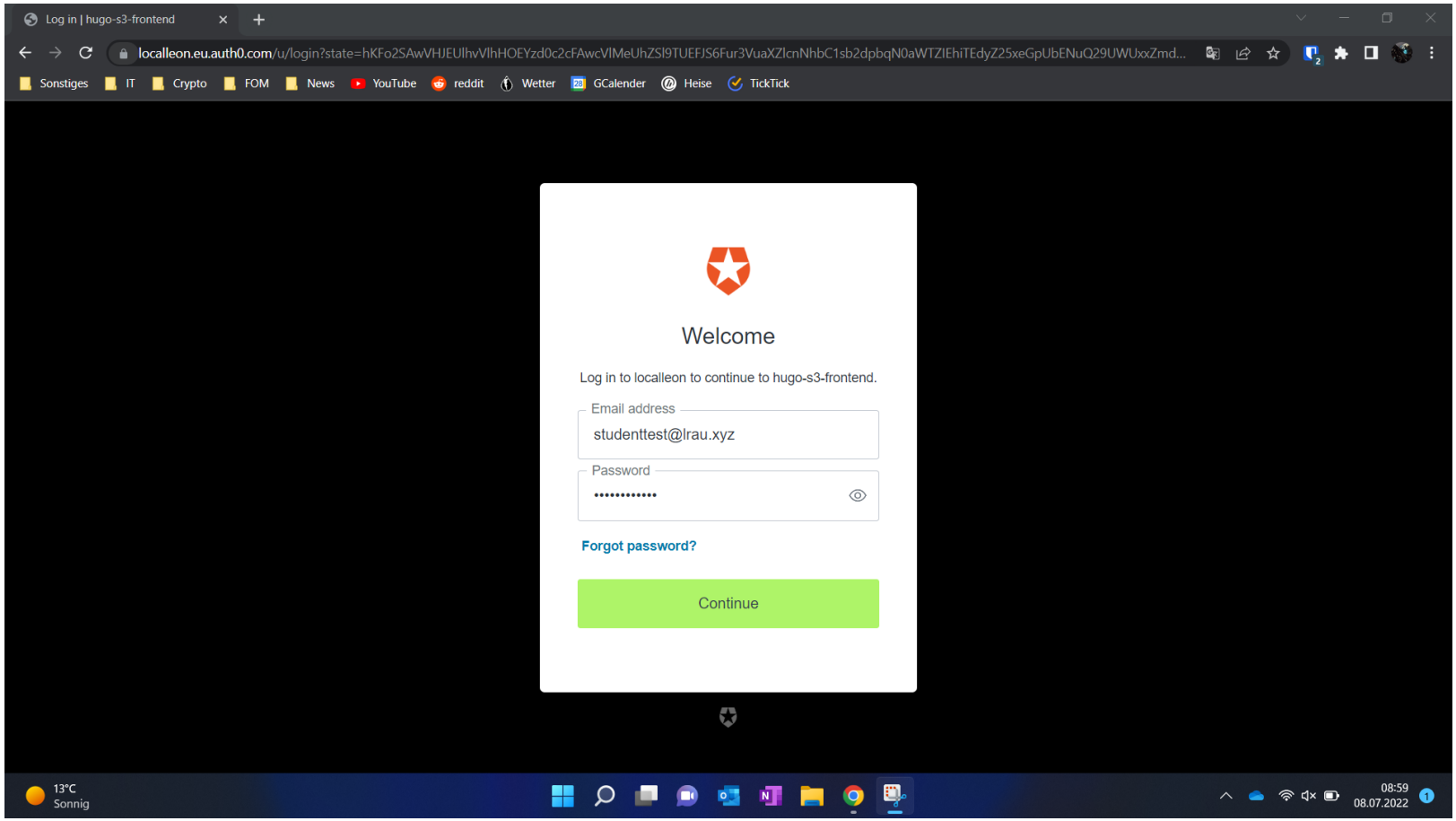
Folgendend finden sich Screenshots aus unterschiedlichen Zuständen der Applikation

Abbildung 18: Welcome-Page der Applikation



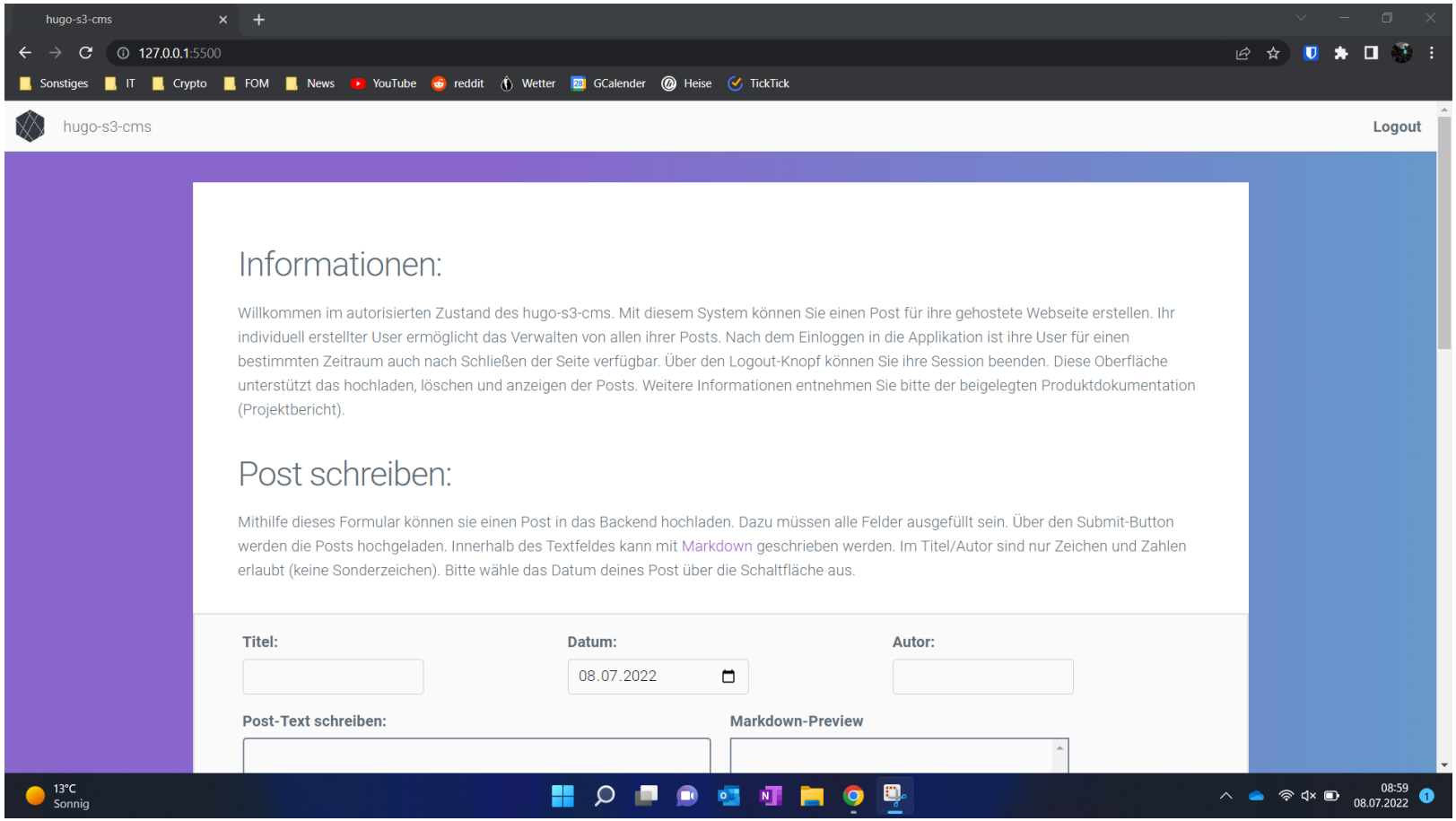
Quelle: Eigene Darstellung

Abbildung 19: Login-Form der Applikation



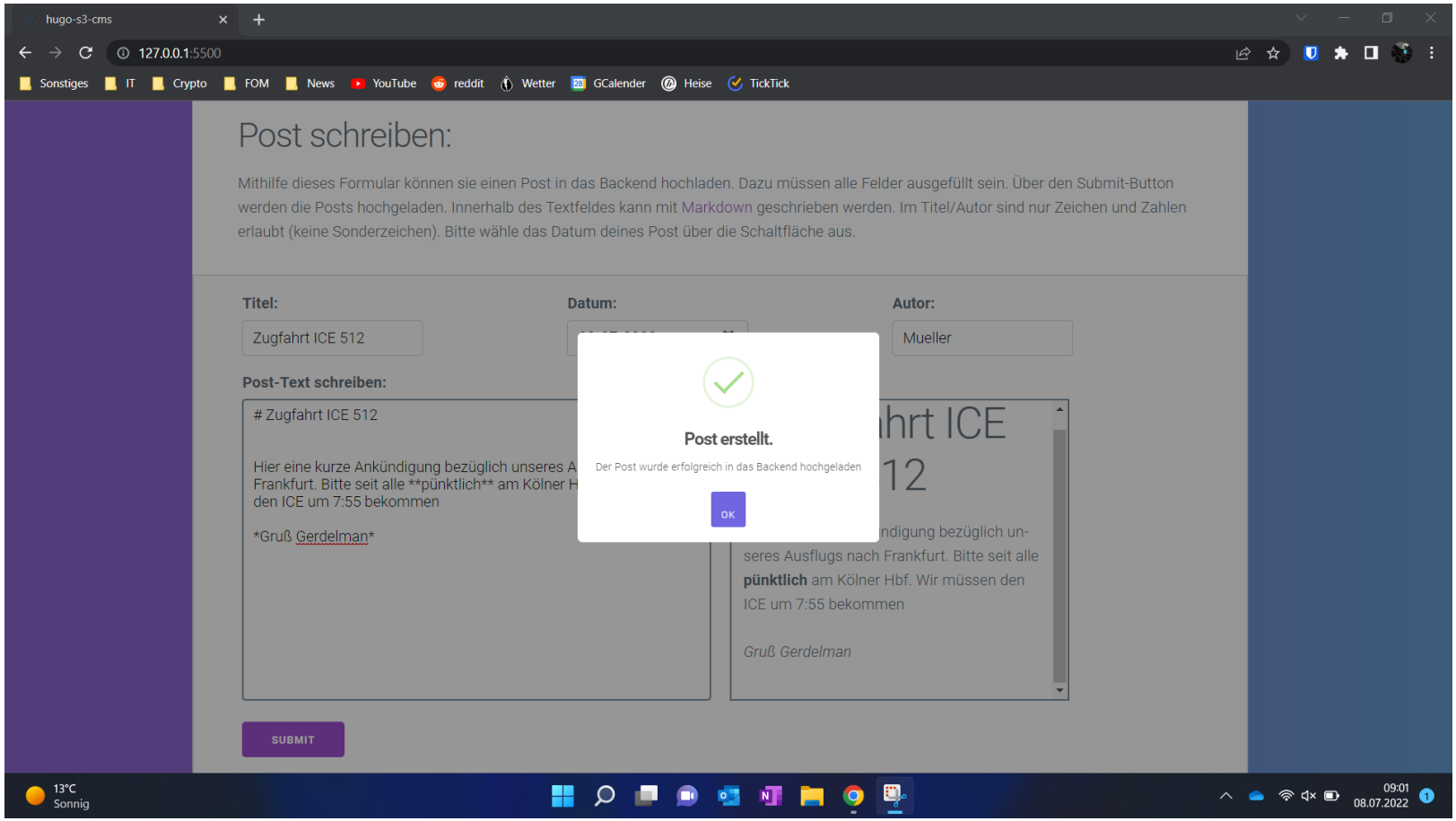
Quelle: Eigene Darstellung

Abbildung 20: Hauptseite der Applikation



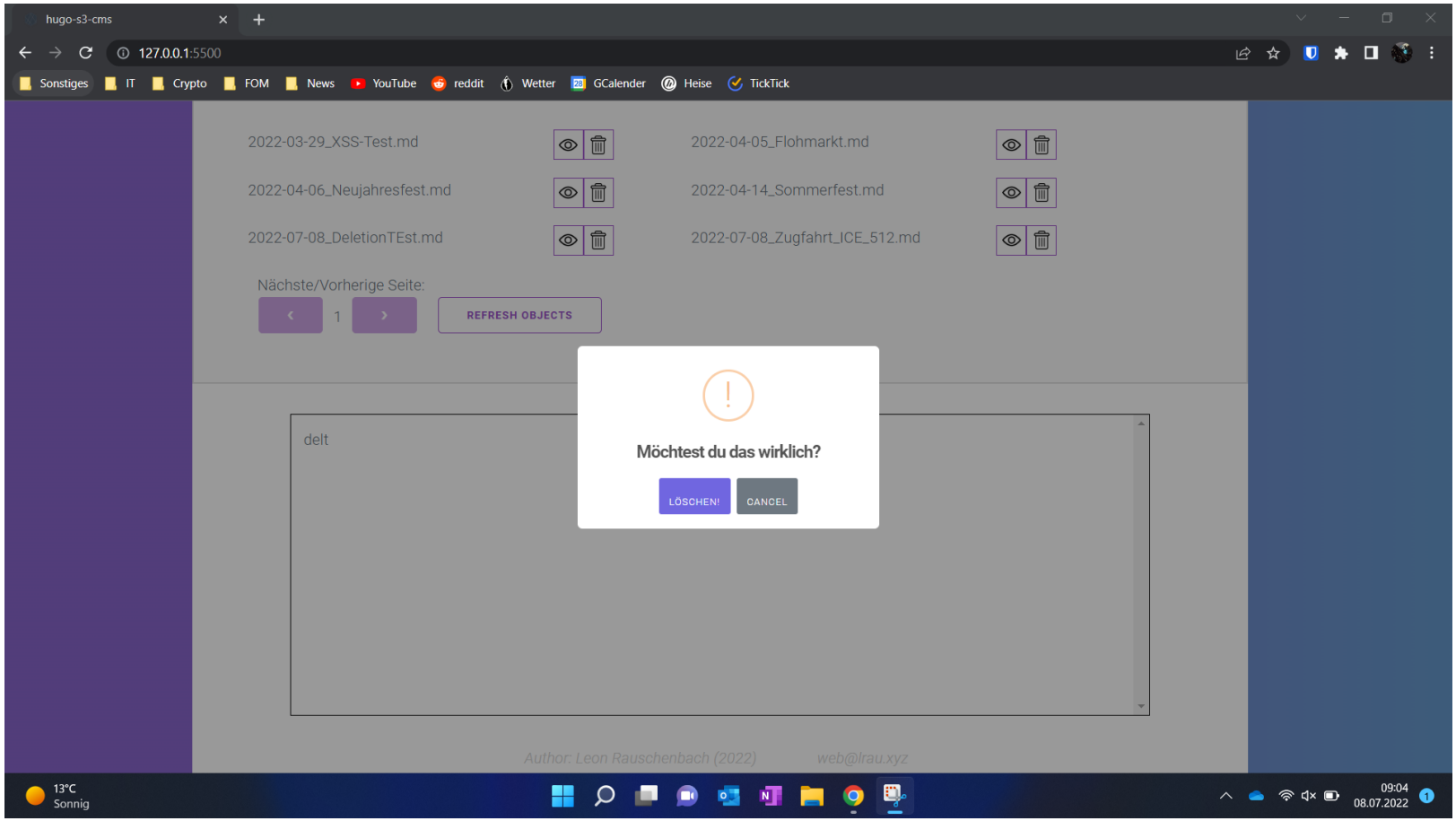
Quelle: Eigene Darstellung

Abbildung 21: Erstellen eines Posts



Quelle: Eigene Darstellung

Abbildung 22: Löschen eines Posts



Quelle: Eigene Darstellung

Internetquellen

[1] Amazon. „API Gateway & API-Entwicklung,“ Amazon Web Services, Inc. (2022), Adresse: https://aws.amazon.com/de/api-gateway/ (besucht am 2022-03-22).

[2] Amazon. „CloudFront CDN (Content Delivery Network),“ Amazon Web Services, Inc. (2022), Adresse: https://aws.amazon.com/de/cloudfront/ (besucht am 2022-03-22).

[3] Amazon. „Lambda Data Processing - Datenverarbeitungsdienste,“ Amazon Web Services, Inc. (2022), Adresse: https://aws.amazon.com/de/lambda/ (besucht am 2022-03-22).

[4] Amazon. „Simple Storage Service S3 – Cloud Online-Speicher,“ Amazon Web Services, Inc. (2022), Adresse: https://aws.amazon.com/de/s3/ (besucht am 2022-03-22).

[5] Auth0. „Auth0: Secure access for everyone. But not just anyone.,“ Auth0. (2022), Adresse: https://auth0.com/ (besucht am 2022-03-31).

[6] auth0.com. „JWT.IO - JSON Web Tokens Introduction.“ (2022), Adresse: http://jwt.io/ (besucht am 2022-04-06).

[7] AWS. „AWS Serverless Application Model (AWS SAM).“ (2022-03-30), Adresse: https://github.com/aws/serverless-application-model (besucht am 2022-03-31).

[8] Hugo. „Hugo: Web-Framework.“ (2013), Adresse: https://gohugo.io/ (besucht am 2022-03-22).

[9] Inc, Serverless. „Serverless: Develop & Monitor Apps On AWS Lambda.“ (2022), Adresse: http://serverless.com// (besucht am 2022-03-31).

[10] nficano. „Python-Lambda: A Toolkit for Developing and Deploying Serverless Python Code in AWS Lambda.“ (2021), Adresse: https://github.com/nficano/python-lambda (besucht am 2022-03-31).

[11] Patoilo, C. J. „Milligram - A minimalist CSS framework,“ Milligram - A minimalist CSS framework. (2020-08-20), Adresse: https://milligram.io/ (besucht am 2022-03-31).

[12] Rauschenbach, Leon. „Posts: Homepage & Blog.“ (2020), Adresse: https://www.lrau.xyz/posts/ (besucht am 2022-03-22).

[13] Sonarcloud. „Automatic Code Review, Testing, Inspection & Auditing | SonarCloud.“ (2022), Adresse: https://sonarcloud.io/ (besucht am 2022-04-07).

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde/Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hochgeladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Düsseldorf, 12.7.2022

(Ort, Datum)

(Eigenhändige Unterschrift)