HOCHSCHULE FURTWANGEN UNIVERSITY HFU

re-lounge. Digitalagentur

Bearbeitungsbeginn: 01.03.2022

Vorgelegt am: 31.08.2022

Thesis

Zur Erlangung des Grades

Bachelor of Science

im Studiengang Medieninformatik

an der Fakultät Digitale Medien

Özkan Yanikbas

Matrikelnummer: 262075

Contentmanagement im Jamstack: redaktionelle Pflege von Websites in der Headless Architektur mit Nuxt 3 und Storyblok

Erstbetreuer: *Thomas Krach*

Zweitbetreuer: *Maximilian Wendt*

Abstract

Für Agenturen und Unternehmen stellt sich die Herausforderung, die passende Architektur bei der Erstellung einer Website auszusuchen. Oft ist eine Migration oder Änderung im Nachhinein durch unterschiedliche technologische Anforderungen aufwendig und teuer. Dabei ist der Aspekt der redaktionellen Pflege einer Website relevant, da je nach Wahl der Architektur völlig neue Arbeitsweisen und Herausforderungen entstehen.

Ziel dieser Arbeit ist es, zu überprüfen, ob der Jamstack hinsichtlich des Funktionsumfangs im Bereich der redaktionellen Pflege von Websites gegenüber der bereits etablierten monolithischen Architektur eine berechtigte Alternative darstellt.

Für diese Arbeit wurde eine intensive Literaturrecherche im Bereich monolithischer- sowie Headless CMS und dem Jamstack betrieben, um eine fundierte Basis für die Theorie zu schaffen. Um die Funktionsweise der Lösungsszenarien zu bestätigen, wurde eine Reihe an praktischen Lösungsszenarien zur redaktionellen Pflege einer realen Website als praktische Ausarbeitung erstellt.

Für jedes der Lösungsszenarien wird die Problemstellung geschildert, worauf eine technologieunabhängige und abstrakte Beschreibung eines möglichen Lösungsweges zur allgemeinen Anwendung im Jamstack folgt. Zuletzt wird ein realer praktischer Lösungsweg mit Storyblok als CMS und Nuxt als Frontend-Framework präsentiert.

Die Lösungsszenarien stellen eine Basis zur Entwicklung von redaktionell pflegbaren Websites im Jamstack dar und können als Grundlage verwendet werden, um weitere Lösungsszenarien zu entwickeln oder um die Workflows in anderen Frameworks oder CMS zu übertragen.

Insgesamt zeigt sich, dass die redaktionellen Aufgaben aus der traditionellen Architektur auch in einer entkoppelten Architektur bestehen können. Der Jamstack ist als anpassungsfähigere Alternative zu sehen, die zusätzlich in Punkten Sicherheit, Skalierbarkeit und Geschwindigkeit Vorteile bietet. Jedoch

kann ein Lösungsweg für die Umsetzung einer redaktionellen Aufgabe je nach Wahl der technologischen Infrastruktur unterschiedlich ausfallen, da der Jamstack eine offene Architektur ist, die es erlaubt, beliebige Technologien zu verwenden. Es lassen sich auch Herausforderungen in Bezug auf die Skalierbarkeit von Websites mit sehr vielen Seiten und der Veröffentlichung von zeitkritischen Inhalten identifizieren. Neue Technologien sind bereits auf dem Weg, um die bestehenden Probleme zu lösen und den Jamstack stets zu verbessern.

Inhaltsverzeichnis

Abstract I
Inhaltsverzeichnis III
Abbildungsverzeichnis VI
Abkürzungsverzeichnis VIII
1. Einleitung 1
1.1 Problemstellung 1
1.2 Beitrag dieser Arbeit 2
1.3 Aufbau dieser Arbeit 2
2. Jamstack 3
2.1 Was ist Jamstack? 3
2.2 SSG 3
2.3 CDN 4
2.4 JavaScript 5
2.5 API 6
2.6 Markup 6
3. Content Management Systeme 7
3.1 Was ist ein CMS? 7
3.1 Aufgaben eines CMS 7
3.2.1 Speicherung 8
3.2.2 Verwaltung 8
3.2.3 Ausgabe 8
3.2.4 Darstellung 9
3.2 Traditionelle / Monolithische CMS 9
3.3 Headless CMS 11
4. Praktische Ausarbeitung 13
4.1 Methodik 13

4.2 Nuxt 3 13
4.2.1 Was ist Nuxt 13
4.2.2 Anmerkungen 14
4.2.3 Rendering Modes 15
4.2.4 SFC / Single-File Components 17
4.2.5 Hooks & Modules 17
4.2.6 Server Routes 18
4.3 Storyblok 18
4.3.1 Was ist Storyblok? 18
4.3.2 Blok-Bibliothek 18
4.3.2 Ordner Struktur 19
4.3.3 Visueller Editor 20
4.4 Netlify 21
4.4.1 Was ist Netlify? 21
4.4.2 Netlify Build 21
4.4.3 Praktische Ausarbeitung 22
5. Lösungsszenarien im Jamstack 23
5.1 Seitengenerierung 23
5.1.1 Problemstellung 23
5.1.2 Lösungsweg 23
5.1.3 Umsetzung in Nuxt 23
5.2 Suchmaschinenoptimierung 25
5.2.1 Problemstellung 25
5.2.2 Lösungsweg 26
5.2.3 Umsetzung in Storyblok 26
5.2.4 Umsetzung in Nuxt 27
5.3 Layouts 28

5.3.1 Problemstellung 28
5.3.2 Lösungsweg 28
5.3.3 Storyblok 29
5.3.4 Umsetzung in Nuxt 29
5.4 Navigation 31
5.4.1 Problemstellung 31
5.4.2 Lösungsweg 31
5.4.3 Umsetzung in Storyblok 31
5.4.4 Umsetzung in Nuxt 33
5.5 Weiterleitungen 33
5.5.1 Problemstellung 33
5.5.2 Lösungsweg 34
5.5.3 Umsetzung in Storyblok 34
5.5.4 Umsetzung in Nuxt 35
5.6 Geschützte Bereiche 36
5.6.1 Problemstellung 36
5.6.2 Lösungsweg 36
5.6.1 Umsetzung in Storyblok 36
5.6.1 Umsetzung in Nuxt 37
5.7 Sitemap 37
5.8 Medienverwaltung 39
5.9 Rollen und Rechte 41
6. Fazit 43
Literaturverzeichnis 45
Bücher 45
Online-Quellen 45
Eidesstattliche Erklärung 51

Abbildungsverzeichnis

Abbildung 1: Funktionsweise SSG (Hawksworth 2020: Abschn. What is a static site generator?) 4

Abbildung 2: Funktionsweise CDN (Was ist ein CDN? | Wie funktionieren CDNs? o. D.: Abschn. Wie funktioniert ein CDN?) 5

Abbildung 3: Aufgabenbereiche eines CMS (vgl. Schürmanns 2021: Abschn. Wie ein Content Management System funktioniert)..... 7

Abbildung 4: vereinfachte Darstellung von einem monolithischen CMS (Angerer 2022: Abschn. What is a headless CMS?)..... 9

Abbildung 5: Funktionsweise API-Driven Headless CMS (Angerer 2022: Abschn. What is a headless CMS?) 12

Abbildung 6: Client-side Rendering (vgl. Nuxt (Guide: Rendering Modes) o. D.: Abschn. Client-side only rendering)..... 16

Abbildung 7: Universal Rendering (Nuxt (Guide: Rendering Modes) o. D.: Abschn. Universal Rendering)..... 16

Abbildung 8: Storyblok - Feldtypen (eigene Aufzeichnung)..... 19

Abbildung 9: Storyblok - beispielhafte Seitenstruktur (eigene Aufzeichnung) 20

Abbildung 10: Storyblok - visueller Editor (eigene Aufzeichnung) 21

Abbildung 11: Storyblok - SEO-Registerkarte aus der Entwicklersicht (eigene Aufzeichnung) 27

Abbildung 12: Storyblok - SEO-Registerkarte aus der Redakteurssicht (eigene Aufzeichnung) 27

Abbildung 13: Storyblok - Layout-Feld (eigene Aufzeichnung)..... 29

Abbildung 14: Storyblok - Navigation (eigene Aufzeichnung)..... 32

Abbildung 15: Storyblok - Navigation Ebene 1 (eigene Aufzeichnung) 32

Abbildung 16: Storyblok - Navigation Ebene 2 (eigene Aufzeichnung) 33

Abbildung 17: Storyblok – 301-Weiterleitung (eigene Aufzeichnung) 35

Abbildung 18: Storyblok – passwortgeschützter Bereich (eigene Aufzeichnung) 37

Abbildung 19: XML-Sitemap (eigene Aufzeichnung) 39

Abbildung 20: Storyblok – Asset-Manager (eigne Aufzeichnung)..... 40

Abbildung 21: Storyblok - Bildeditor (eigene Aufzeichnung) 40

Abbildung 22: Storyblok – Zugriffsrechte einer neuen Rolle (eigene Aufzeichnung) 42

Abkürzungsverzeichnis

API ..... *Application Programming Interface*
CDN ..... *Content Delivery Network*
CI/CD ..... *Continuous Integration, Continuous Delivery*
CMS ..... *Content Management System*
CSS ..... *Cascading Style Sheets*
DOM ..... *Document Object Model*
HTML ..... *Hypertext Markup Language*
ISG ..... *Incremental Static Regeneration*
JSON ..... *JavaScript Object Notation*
OGP ..... *Open Graph protocol*
SEO ..... *Suchmaschinenoptimierung*
SFC ..... *Single-File Components*
SPA ..... *Single Page Application*
SSG ..... *Static Site Generator*
SSR ..... *Server Side Rendering*
URL ..... *Uniform Resource Locator*
WCMS ..... *Web Content Management System*
WWW ..... *World Wide Web*
XML ..... *eXtensible Markup Language*

1. Einleitung

1.1 Problemstellung

Mit der immer weiter fortschreitenden Verbreitung des World Wide Webs (WWW) entstanden Systeme zur Verwaltung von Inhalten (Content) auf Websites (vgl. Spörrer 2019: 1). Solche Systeme werden Content Management System (CMS) genannt und ermöglichen Anwendern den Content, ohne die Voraussetzung von Programmierkenntnissen zu verwalten (vgl. ebd.: 32).

In den frühen 2000ern etablierten sich sogenannte monolithische CMS wie WordPress, Joomla und Drupal. Die monolithische Architektur bietet eine vereinte Lösung durch die Übernahme der Aufgaben des Servers, der Präsentationsschicht (Frontend) und der Integration einer Datenbank innerhalb einer Codebasis (vgl. Sycamore 2021: Abschn. Traditional CMS: WordPress).

Im Kontrast zu den monolithischen Systemen entwickelten sich neue Ansätze. Ein solcher neuer Ansatz ist der Jamstack. Der Jamstack entkoppelt die Aufgaben und Dienste einer Architektur und strukturiert diese in einzelne losgelöste Bausteine, die unabhängig voneinander ausgetauscht und gewartet werden können (vgl. Dillon 2021: Abschn. Decoupling). Diese Architektur setzt auf die Geschwindigkeit, Sicherheit und Skalierbarkeit des WWW (vgl. Enyinnaya 2019: Abschn. Why the JAMstack?).

Für Agenturen und Unternehmen stellt sich die Herausforderung, für die Erstellung einer Website die passende Architektur auszusuchen. Oft ist eine Migration oder Änderungen im Nachhinein durch unterschiedliche technologische Anforderungen und Inkompatibilität aufwendig und teuer (vgl. Nguyen 2018: Abs. 2-3). Dabei ist der Aspekt der redaktionellen Pflege einer Website relevant, da je nach Wahl der Architektur völlig neue Arbeitsweisen und Herausforderungen entstehen (vgl. ebd.).

Der Jamstack ist ein noch neuer Ansatz, der aber die letzten Jahre immer mehr an Beliebtheit gewinnen konnte (vgl. Denysov 2022: Abschn. Adoption of

SSGs). Ziel dieser Arbeit ist es, zu überprüfen, ob dieser Ansatz hinsichtlich des Funktionsumfangs im Bereich der redaktionellen Pflege von Websites gegenüber der bereits etablierten monolithischen Architektur eine legitime Alternative darstellt.

1.2 Beitrag dieser Arbeit

In der vorliegenden Arbeit geht es um die Ausarbeitung praktischer Lösungsszenarien für die redaktionelle Pflege von Websites im Jamstack. Die erarbeiteten Lösungsszenarien sollen darstellen, wie die redaktionellen Aufgaben, die Entwickler und Redakteure aus der monolithischen Architektur gewohnt sind, im Jamstack umzusetzen sind.

1.3 Aufbau dieser Arbeit

Im ersten Teil der Arbeit werden die notwendigen Grundlagen vorgestellt. Darunter fällt die Begriffsklärung „Jamstack“ und dessen Funktionsweisen. Hinzu kommen die unterschiedlichen Ansätze des Contentmanagements. Nachfolgend werden die Schlüsseltechnologien der praktischen Ausarbeitung erklärt.

Im zweiten Teil geht es um die praktische Ausarbeitung. Es enthält die praktischen Lösungsszenarien für die redaktionelle Pflege von Websites im Jamstack. Dabei wird für jedes Lösungsszenario die Vorgehensweise nachvollziehbar und strukturiert präsentiert.

Abschließend gibt es ein Fazit über die erarbeiteten Lösungsszenarien mit der Einordnung der redaktionellen Pflege von Websites im Jamstack.

2. Jamstack

2.1 Was ist Jamstack?

Der Begriff Jamstack wurde von dem Unternehmen Netlify im Jahr 2015 eingeführt, um die von Entwicklern bereits genutzten Konzepte dieses Ansatzes besser zu beschreiben (vgl. Cardoza 2020: Abs. 5). Am 6. April 2016 wurde der Jamstack auf der Smashing Conference von Netlifys Co-Founder und CEO Mathias Biillmann in San Francisco der Öffentlichkeit vorgestellt (vgl. Smashing Conference 2016 o. D.).

Der Jamstack ist eine Art und Weise, wie man Websites und Apps für das WWW entwickelt. Dieser Ansatz besteht aus drei zentralen Bausteinen, die durch unterschiedlichste Technologien und Dienste gelöst werden können (vgl. Camden/Rinaldi 2022: Kap. 1.1). Diese Bausteine sind JavaScript, API (Application Programming Interface) und Markup, die auch in den ersten drei Buchstaben im Wort Jamstack (JAM) wiederzufinden sind (vgl. Camden/Rinaldi 2022: Kap. 1.2.2).

Im Grunde möchte der Jamstack Entwicklern dabei helfen, schnelle, sichere und skalierbare Websites und Apps zu realisieren und gleichzeitig die Entwicklung zu beschleunigen (vgl. Cardoza 2020: Abs. 8). Im Folgenden werden die wichtigsten Bestandteile des Jamstack vorgestellt.

2.2 SSG

Ein zentraler Aspekt im Jamstack ist, dass Websites und Apps bereits zur Anfrage des Besuchers als statische Dateien vorliegen (vgl. Camden/Rinaldi 2022: Kap. 1.1). Inhalte werden nicht von einem klassischen Server erst zur Anfrage dynamisch generiert (vgl. ebd.). Diese Herangehensweise wird durch die Nutzung eines sogenannten Static Site Generators (SSG) ermöglicht. Vereinfacht füllt ein SSG Templates mit Inhalten aus einer Datenquelle und generiert dynamisch das HTML (Hypertext Markup Language), JavaScript und CSS (Cascading Style Sheets) der jeweiligen Seiten im Voraus (vgl. ebd.). Dieser Prozess wird Pre-rendering genannt (vgl. Netlify (Jamstack.org: Pre-

render) o. D.). In Abbildung 1 wird der Pre-rendering Prozess eines SSG dargestellt.

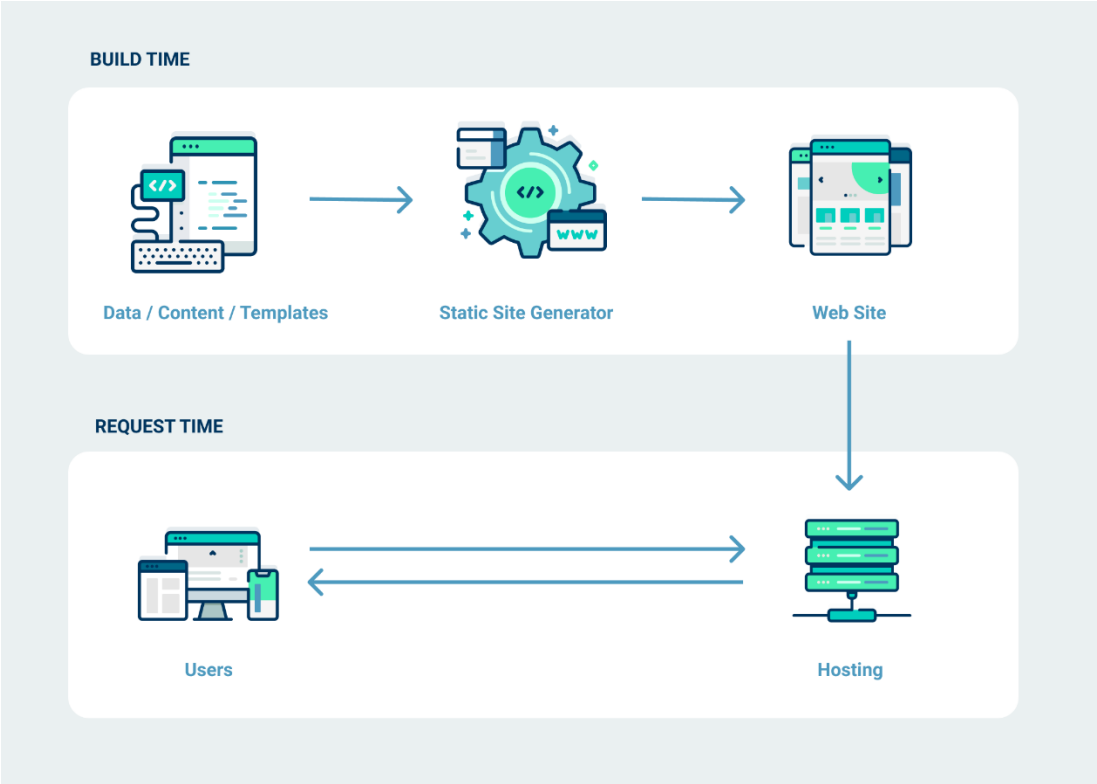

Abbildung 1: Funktionsweise SSG (Hawksworth 2020: Abschn. What is a static site generator?)

2.3 CDN

Da die vom SSG dynamisch generierten Inhalte statisch ausgegeben werden, kann man sie durch ein Content Delivery Network (CDN) weltweit zur Verfügung stellen (vgl. Camden/Rinaldi 2022: Kap. 1.3.1). Ein CDN ist ein globales Netzwerk an Servern zur Bereitstellung von Inhalten im WWW (vgl. Cloudflare o. D.: Abschn. Wie funktioniert ein CDN?). Ladezeiten können stark reduziert werden, da sich die Server näher zum Endnutzer befinden als einzelne oder wenige zentral verwaltete Hosting-Server (vgl. ebd.: Abschn. Welche Vorteile hat die Verwendung eines CDN?). Des Weiteren kann durch ein CDN, aufgrund der hohen Anzahl der abrufbereiten CDN-Server, eine bessere Verfügbarkeit bei einem Hardwaredefekt oder bei erhöhtem Traffic gewährleistet werden (vgl. ebd.). In Abbildung 2 wird die Funktionsweise eines CDN dargestellt.

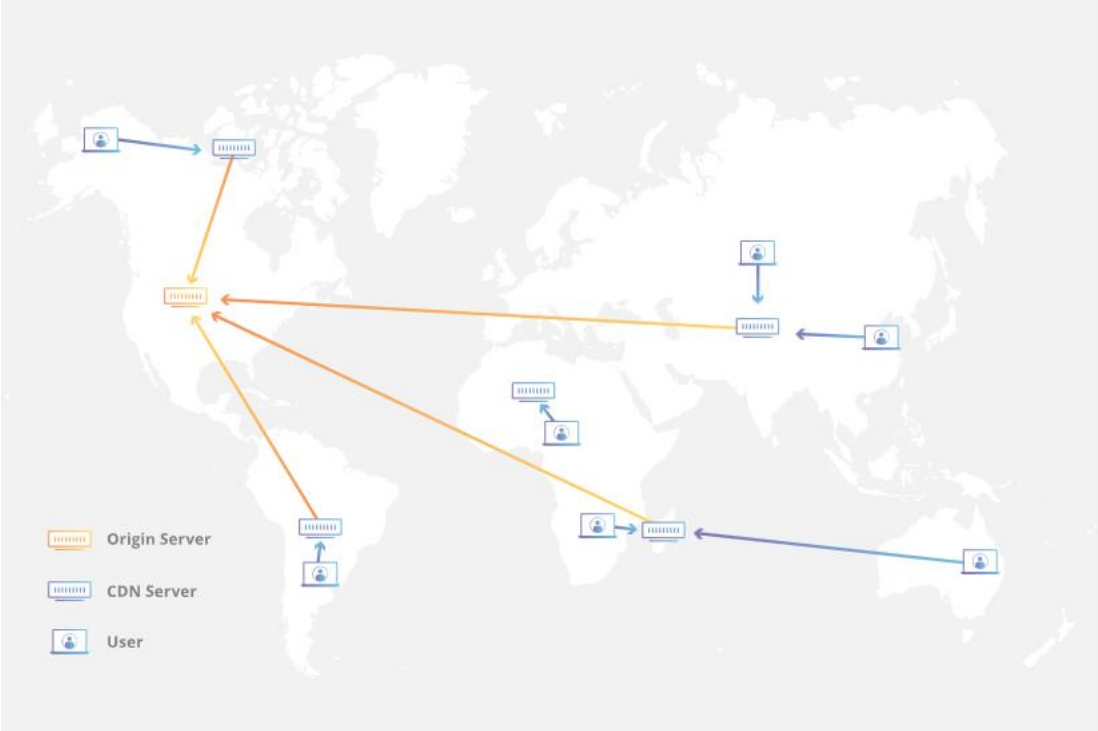

Abbildung 2: Funktionsweise CDN (Was ist ein CDN? | Wie funktionieren CDNs? o. D.: Abschn. Wie funktioniert ein CDN?)

2.4 JavaScript

Die Programmiersprache JavaScript ist in einer berühmten und kurzen Entwicklungsphase von etwa 12 Tagen im Mai 1995 von Brendan Eich entwickelt worden (vgl. Ackermann 2018: 22). Anfangs hieß die Programmiersprache noch Mocha und wurde für den Internetbrowser Netscape Navigator entwickelt (vgl. ebd.).

Die Einsatzgebiete von JavaScript sind heute vielfältig. Heute kommt die Sprache nicht nur im Browser zum Einsatz, sondern auch auf Servern, Desktop- und mobilen Anwendungen (vgl. Ackermann 2018: 24). Man kann sagen, dass JavaScript allgegenwärtig ist und zu einem unverzichtbaren Bestandteil des WWW geworden ist.

Der Jamstack nutzt JavaScript, um die vom SSG statisch hinterlegten Websites und Apps dynamische und interaktiv zu machen (vgl. Camden/Rinaldi 2022: Kap. 1.1). Es wird unter anderem für die Kommunikation mit Programmierschnittstellen und externen Diensten verwendet (vgl. ebd.).

2.5 API

Der grundlegende Unterschied zwischen einfachen statischen Websites und Websites im Jamstack ist der, dass die Inhalte nicht statisch sein müssen (vgl. Camden/Rinaldi 2022: Kap. 1.1). Dynamische Funktionalitäten und Inhalte werden durch die Nutzung von Programmierschnittstellen ermöglicht (vgl. ebd.). APIs kommen im Browser, aber auch bei der Generierung der Seiten beim SSG zum Einsatz (vgl. ebd.). So ist es möglich, komplexe und interaktive Inhalte zu erstellen.

2.6 Markup

HTML ist ein Kernelement des WWW (vgl. Biilmann/Hawksworth 2019: 10). Es wird von jedem Browser verstanden und definiert, wie Inhalte strukturiert sind (vgl. ebd.). Es beschreibt auch, welche Ressourcen einer Seite geladen und präsentiert werden sollen (vgl. ebd.). Besonders elementar ist das Document Object Model (DOM), das Browsern und anderen Geräten ermöglicht, Struktur und Inhalte zu durchsuchen, zu präsentieren und zu manipulieren (vgl. ebd.).

Die Herangehensweise, wie Markup im Jamstack ausgegeben wird, unterscheidet sich zu herkömmlichen Serverarchitekturen, die das Markup erst zur Laufzeit bei eingehendem Seitenaufruf generieren (vgl. Biilmann/Hawksworth 2019: 11). Wie bereits in 2.2 SSG und 2.3 CDN erwähnt, wird im Jamstack das Markup bereits im Voraus durch einen SSG generiert und mit einem CDN global zum Abruf zur Verfügung gestellt (vgl. ebd.). Dadurch kann das Frontend und das Backend (Serverebene) voneinander entkoppelt und isoliert betrachtet und Aspekte wie Skalierung, Sicherheit und Performance besser angegangen werden (vgl. ebd.).

3. Content Management Systeme

3.1 Was ist ein CMS?

Ein CMS ist ein System zur redaktionellen Pflege von Inhalten (vgl. Spörrer 2019: 32). Bei dieser Arbeit geht es primär um das Web Content Management System (WCMS), das sich mit der Verwaltung von Websites beschäftigt und gleichzeitig auch als Synonym für CMS gilt (vgl. ebd.: 26). Ein CMS trennt den Inhalt, die Struktur und das Layout voneinander, wodurch eine klare Arbeitsteilung entsteht (vgl. ebd.: 31). Redakteure sollen sich auf die Pflege von Inhalten fokussieren können, ohne besondere technische Kenntnisse zu benötigen (vgl. ebd.: 32).

3.1 Aufgaben eines CMS

Welche Features ein spezifisches CMS mit sich bringt, schwankt von einem System zum anderen. Um trotzdem einen umfangreichen Überblick schaffen zu können, kann der Aufgabenbereich eines CMS in vier Kategorien unterteilt werden: Inhalte speichern, verwalten, ausliefern und optional darstellen (vgl. Schürmanns 2021: Abschn. Wie ein Content Management System funktioniert).

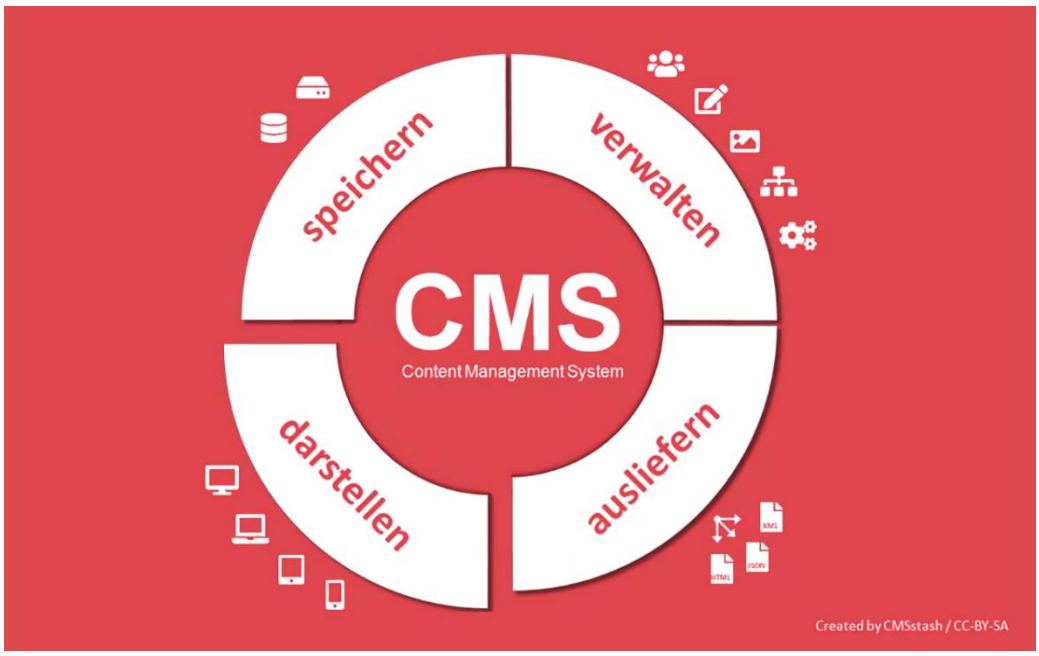

Abbildung 3: Aufgabenbereiche eines CMS (vgl. Schürmanns 2021: Abschn. Wie ein Content Management System funktioniert)

3.2.1 Speicherung

Ein CMS speichert alle Inhalte wie Medien, Dokumente und Konfigurationen ab. Wie die Inhalte genau gespeichert werden, unterscheidet sich je nach System grundlegend. Inhalte werden meistens in relationalen oder unstrukturierten Datenbanken gesichert (vgl. Schürmanns 2021: Abschn. Inhalte mit einem CMS speichern). Es gibt auch CMS, die ohne Datenbanken arbeiten können und als Flat-File-CMS bezeichnet werden (vgl. ebd.). Flat-File-CMS speichern alle Inhalte als Dateien im File-System des Servers ab (vgl. ebd.). Medieninhalte wie Bilder werden für gewöhnlich bei allen Systemen als solche im File-System des Servers gesichert (vgl. ebd.).

3.2.2 Verwaltung

Wie bereits in 3.1 Was ist ein CMS? erwähnt, geht es bei einem CMS um die redaktionelle Pflege von Inhalten. Um Inhalte pflegen zu können, muss das CMS die Möglichkeit bieten, Inhalte zu erstellen, zu editieren oder zu löschen (vgl. Schürmanns 2021: Inhalte mit einem CMS verwalten). Dazu haben CMS eine Benutzeroberfläche, die zur Steuerung des Systems dient (vgl. ebd.). Um Inhalte zu verwalten, benötigt ein CMS bestimmte Funktionalitäten, die sich über die Benutzeroberfläche steuern lassen (vgl. ebd.). Dazu gehört die Nutzerverwaltung, die dafür benötigt wird, um sich als Nutzer anmelden zu können (vgl. ebd.). Darüber hinaus gibt es die Funktionalität, Rollen und Rechte zu verteilen (vgl. ebd.). Beispielsweise haben Redakteure und Administratoren unterschiedliche rollenspezifische Zugriffsrechte, um ungewollte Eingriffe ins System zu vermeiden. Eine weitere Funktionalität ist die Medienverwaltung, in dem Medieninhalte wie Bilder, Videos und Dokumente separat im System gespeichert und verwaltet werden (vgl. ebd.).

3.2.3 Ausgabe

Die Ausgabe von Inhalten stellt eine weitere Kernfunktionalität eines CMS dar. Dabei ist die Zugriffssteuerung ein wichtiger Aspekt, die dabei hilft zu regeln, wer die Berechtigung hat, die zu ausgebenden Inhalte zu sehen (vgl. Schürmanns 2021: Inhalte mit einem CMS ausgeben). Beispielsweise dürfen nicht angemeldete Benutzer auf bestimmte Inhalte nicht zugreifen. Zudem

unterscheidet sich das Ausgabeformat der Inhalte je verwendetem System. Gängige Ausgabeformate sind HTML, XML (eXtensible Markup Language) und JSON (JavaScript Object Notation) (vgl. ebd.).

3.2.4 Darstellung

Wie die sich im CMS befindenden Inhalte präsentiert werden, unterscheidet sich grundsätzlich nach der Wahl der CMS-Architektur. Während es die Aufgabe des CMS sein kann, die Inhalte darzustellen, trifft es nur für das traditionelle CMS zu (vgl. Schürmanns 2021: Inhalte darstellen). In den folgenden Kapiteln 3.2 Traditionelle / Monolithische CMS und 3.3 Headless CMS werden die Unterschiede bezüglich der Präsentation genauer erklärt.

3.2 Traditionelle / Monolithische CMS

Ein monolithisches CMS, auch bekannt als traditionelles CMS, zeichnet sich durch die Vereinigung der Präsentationsschicht, dem Server und der Datenbank innerhalb einer Codebasis als eine ganze Einheit aus (vgl. Sycamore 2021: Abschn. Traditional CMS: WordPress). In Abbildung 4 ist der Aufbau eines monolithischen CMS visualisiert.

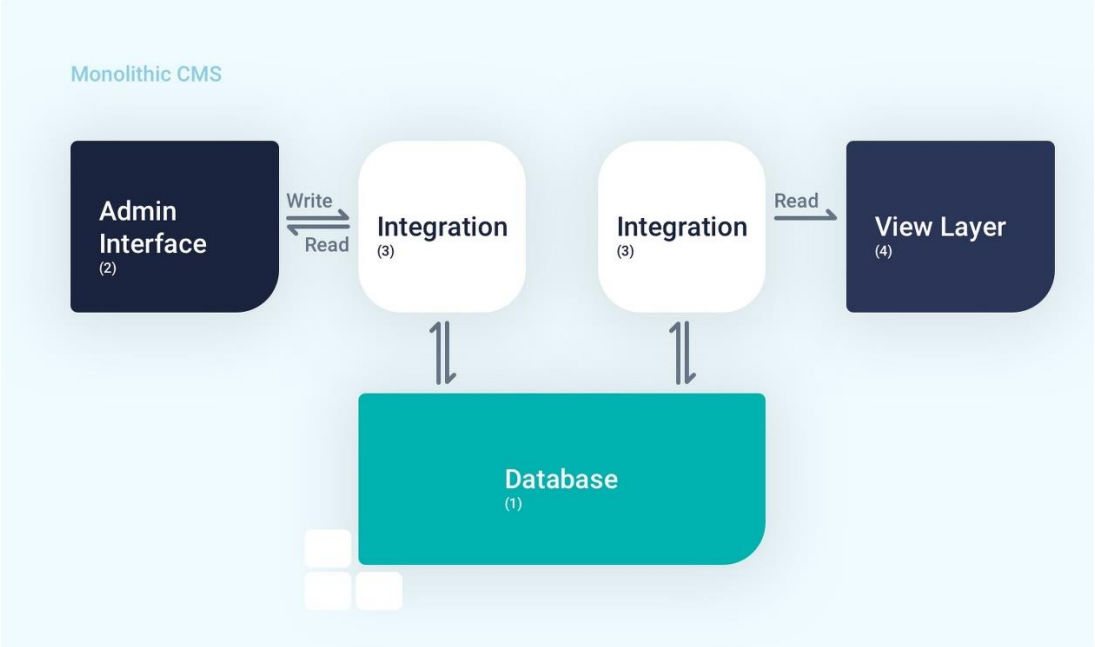

Abbildung 4: vereinfachte Darstellung von einem monolithischen CMS (Angerer 2022: Abschn. What is a headless CMS?)

Ein gutes Beispiel für ein monolithisches CMS ist WordPress. WordPress ist ein kostenloses Open-Source CMS, das Stand Juli 2022 mit 43 % Marktanteil das meistgenutzte CMS für Websites im Internet ist (vgl. Statista 2022). WordPress kommt zur Erstellung und redaktionellen Pflege von Websites jeder Größenordnung zum Einsatz (vgl. iThemes 2020: Abschn. What Is WordPress?).

Eines der Hauptmerkmale von WordPress ist, dass es wenig bis keine technischen Kenntnisse voraussetzt, um es aufzusetzen und Seiten redaktionell zu pflegen (vgl. iThemes 2020: Abschn. Is WordPress right for me?). Hinzu kommt ein Plug-in-System, das dem Benutzer ermöglicht, das System mit neuen Funktionalitäten zu erweitern (vgl. ebd.). Zudem können Nutzer mit wenig Aufwand das Theme einer Seite verändern (vgl. ebd.). Als meistgenutztes CMS gibt es eine große Auswahl an kostenlosen sowie kostenpflichtigen Plug-ins und Themes, die über beliebige Marktplätze erworben werden können (vgl. ebd.).

Auch wenn das monolithische CMS eine bewährte Architektur mit vielen Vorteilen ist, zieht es mit dem Blick der aktuellen Zeit einige Nachteile mit sich (vgl. Contentstack o. D.: Abs. 1–2). Bei traditionellen Architekturen werden Seiten erst zur Zeit der Anfrage vom Server dynamisch erstellt (vgl. Sycamore 2021: Abschn. Traditional CMS: WordPress). Das ist auch der Fall, wenn es sich um eine inhaltlich statische Seite handelt, die nicht bei jeder neuen Anfrage dynamisch erstellt werden müsste (vgl. ebd.). Das führt im Vergleich zu neueren Ansätzen wie dem Jamstack, die die Seiten bereits zur Zeit der Anfrage zur Verfügung stellen, wie in 2.2 SSG vorgestellt, zu einem deutlichen Performance-Nachteil. Hinzu kommt, dass die Sicherheit des Servers eine sehr hohe Priorität haben muss, da alle Anfragen immer direkt mit dem Produktivsystem kommunizieren und somit einen möglichen Eintrittspunkt für Hacker darstellen.

Da alle Bausteine des Systems sehr eng miteinander innerhalb einer Codebasis in Verbindung stehen, sind technische Änderungen und Updates immer mit sehr viel Aufwand und Risiken verbunden (vgl. Karwatka et al. o. D.: Abschn. The disadvantages of monolithic architecture). Es werden auch

systemspezifische Spezialisten benötigt, die sich explizit mit der verwendeten CMS-Technologie auskennen müssen (vgl. ebd.). Zudem besteht die Gefahr, dass wenn ein Teilbereich des Systems, wie z. B. die Verwendung eines Plug-ins einen Fehler erzeugt, es das ganze Produktivsystem außer Betrieb bringen kann. Man sollte auch bedenken, dass eine erhöhte Serverlast für bestimmte Teilbereiche eines monolithischen Systems dazu führt, dass das gesamte System langsamer wird.

3.3 Headless CMS

Ein Headless CMS ist im Vergleich zum monolithischen CMS ein Content Management System, das vollständig von der Präsentationsebene gelöst ist (vgl. Dillon 2021: Abschn. What is a Headless CMS?). Die Kommunikation mit dem Frontend findet durch eine vom CMS bereitgestellte API statt (vgl. Schiel 2020: Abschn. Was ist ein Headless CMS?). Inhalte haben im CMS keine Layouts oder Templates, die das identische Endergebnis darstellen (vgl. ebd.). Da das CMS entkoppelt vom Frontend ist und nur die rohen Inhalte enthält, können Inhalte auch auf mehreren verschiedenen Frontends ausgegeben werden (vgl. Schiel 2020: Abschn. Warum sollte man nun Headless CMS statt einem monolithischen CMS verwenden?). Daraus ergibt sich der Vorteil, dass Entwickler mehr Flexibilität bei der Wahl der passenden Programmiersprache und in der Gestaltung des Frontends haben, weil es nicht wie im monolithischen CMS an die Systemtechnologie gebunden ist (vgl. ebd.).

Diese Art von Headless CMS wird auch API-Driven CMS genannt, da alle Inhalte durch eine API zur Verfügung gestellt werden (vgl. King 2022: Abschn. Git-based vs API-Driven CMS). Eine weitere Möglichkeit ist es, den gesamten Content im versionierten Repository des Frontends zu speichern. Diese Art von Headless CMS wird Git-based CMS genannt (vgl. ebd.). Im Vergleich zum API-Driven CMS wird beim Git-based CMS die Ausgabe des Inhalts auf das eigene Frontend beschränkt (vgl. ebd.). Da der gesamte Content im versionierten Repository gespeichert wird, gibt es keine Anbieterbindung und Inhalte sind sicher in den Händen des Eigentümers (vgl. ebd.). In dieser Arbeit wird es primär um das API-Driven Headless CMS gehen.

Die entkoppelte Natur eines Headless CMS qualifiziert es als passenden Baustein für die redaktionelle Pflege von Inhalten im Jamstack. In Abbildung 5 ist der Aufbau eines API-Driven Headless CMS visualisiert.

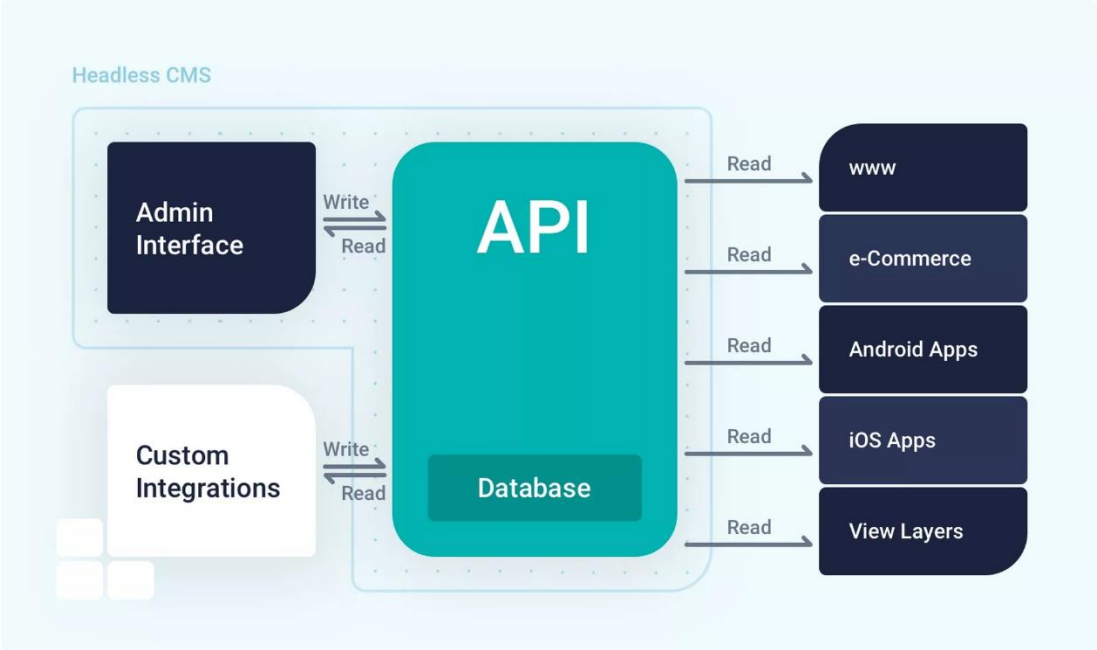

Abbildung 5: Funktionsweise API-Driven Headless CMS (Angerer 2022: Abschn. What is a headless CMS?)

4. Praktische Ausarbeitung

4.1 Methodik

Um eine fundierte Basis für die Theorie zu schaffen, wurde für die in 5. Lösungsszenarien im Jamstack dargestellten Lösungsszenarien eine intensive Literaturrecherche im Bereich monolithischer sowie Headless CMS und dem Jamstack betrieben.

Um die Funktionsweise der Lösungsszenarien zu bestätigen, wurde eine praktische Ausarbeitung entwickelt. Damit alle Prozesse transparent und nachvollziehbar sind, wurde der Quellcode mit dem Versionsverwaltungssystem Git dokumentiert. Der versionierte Quellcode ist als Anhang der Bachelorarbeit oder öffentlich bei GitHub unter der Adresse https://github.com/oezkancodes/bachelorarbeit zugänglich.

Für jedes vorgestellte Lösungsszenario wird zuerst die Problemstellung geschildert. Darauf folgt die technologieunabhängige und abstrakte Beschreibung eines möglichen Lösungsweges für die allgemeine Anwendung im Jamstack. Zuletzt wird ein realer praktischer Lösungsweg mit einem Technologie-Stack, der in den folgenden Unterkapiteln vorgestellt wird, greifbar gemacht.

In den folgenden Kapiteln 4.2 Nuxt 3, 4.3 Storyblok und 4.4 Netlify wird die technische Grundlage, aus der die praktische Ausarbeitung im Kern besteht, vorgestellt.

4.2 Nuxt 3

4.2.1 Was ist Nuxt

Nuxt ist ein modernes und hybrides Web-Framework, das in dieser Arbeit für die Entwicklung der Lösungsszenarien zur redaktionellen Pflege von Websites im Jamstack als Frontend sowie als SSG verwendet wurde. Im Kern basiert Nuxt auf dem populären JavaScript-Framework Vue.js (Vue), das im Februar 2014 von Evan You veröffentlicht wurde (vgl. Prasad 2022: Abs. 2). Vue ist ein komponentenbasiertes JavaScript-Framework, das zur Erstellung von

Benutzeroberflächen und Single Page Applications (SPA), auch bekannt als client-seitige Web-Anwendungen, eingesetzt wird (vgl. Singh 2021: Abschn. Vue to Web: Introduction Part-1). Nuxt erweitert Vue mit stärkeren Konventionen in Bezug auf die Projektstruktur und führt eine Serverseite ein (vgl. Kinsta 2022: Abs. 1–2). Durch den Server ergeben sich neue Möglichkeiten wie SEO-freundliche Seiten oder der statischen Generierung der Seiten durch einen SSG (vgl. Kinsta 2022: Abschn. Statisch generierte Seiten). Hinzu kommt, dass Nuxt keine initiale Konfiguration benötigt. Es bietet ein sogenanntes Zero-Config-Erlebnis mit einer internen Standard-Konfiguration mit bewährten Praktiken aus der JavaScript- und der Vue-Entwicklung an (vgl. Kinsta 2022: Was ist Nuxt.js?).

Darüber hinaus hat Nuxt einen integrierten Router, das client-seitiges Routing ermöglicht. Der Wechsel von Seiten kann beim client-seitigen Routing durch JavaScript ausgeführt werden (vgl. Vue.js o. D.: Kap. Routing). Durch automatisches seitenbasiertes Code-Splitting besitzt jede Seite seinen eigenen JavaScript-Bundle (vgl. Kinsta 2022: Abschn. Automatisches Code-Splitting). Das Code-Splitting ermöglicht es dem Router, zusammenhängende Seiten zu erkennen und sie im Voraus zu laden, wodurch ein Seitenwechsel meist ohne Verzögerung stattfinden kann. Beim clientseitigen Routing führt ein Seitenwechsel dazu, dass nur die Inhalte im Dokument per JavaScript ausgetauscht werden, statt ein neues Dokument vom Server anzufordern und zu initialisieren (vgl. Vue.js o. D.: Kap. Routing).

4.2.2 Anmerkungen

Die Nuxt-Version, die in dieser Arbeit zum Einsatz kam, ist der Release-Kandidat `nuxt@3.0.0-rc.7` der Nuxt 3 Beta. Mehr zur Beta kann dem offiziellen Beitrag von Nuxt entnommen werden: https://nuxtjs.org/announcements/nuxt3-rc/. Die erarbeiteten Ergebnisse für Nuxt 3 sind somit sehr nah am modernsten Geschehen des Frameworks. Es ist zwar unwahrscheinlich, jedoch nicht ausgeschlossen, dass sich zur Veröffentlichung der offiziellen Release-Version an gewissen Stellen noch Syntax oder Konzepte ändern können.

Der Quellcode der im schriftlichen Teil dieser Arbeit vorgestellten Lösungsszenarien bezieht sich immer auf Nuxt 3. Der Code wird teils vereinfacht dargestellt und soll lediglich die Vorgehensweise näherbringen. Es wird immer ein Pfad zur Datei angegeben, der im Quellcode existiert und im vollen Umfang betrachtet werden kann.

4.2.3 Rendering Modes

Nuxt bietet zwei verschiedene Modi an, um Seiten zu rendern (vgl. Nuxt (Rendering Modes) o. D.: Abschn. Rendering Modes). Der erste Modus ist das Client-side Rendering und gleicht dem Rendering einer klassischen, auf Vue basierten Web-Anwendung (vgl. Nuxt (Rendering Modes) o. D.: Abschn. Client-side only rendering). Wenn der Besucher eine Anfrage zum Server stellt, wird ihm ein HTML-Dokument gesendet, das anbei das JavaScript mit der gesamten Logik für die Anwendung beinhaltet (vgl. ebd.). Das Dokument ist in der Regel leer oder enthält einen Ladebildschirm. Im Folgenden initialisiert der Browser das JavaScript, was meist eine kurze Zeit in Anspruch nimmt (vgl. ebd.). Erst nachdem der Browser das JavaScript vollständig ausgeführt hat, werden die tatsächlichen Inhalte der Seite präsentiert und abschließend interaktiv gemacht (vgl. ebd.). Dieser Modus enthält für das Rendering keine serverseitige Logik, da alle Inhalte gewollt im Browser verarbeitet und ausgeführt werden. In Abbildung 6 ist der Prozess des Client-side Renderings in drei Schritten dargestellt.

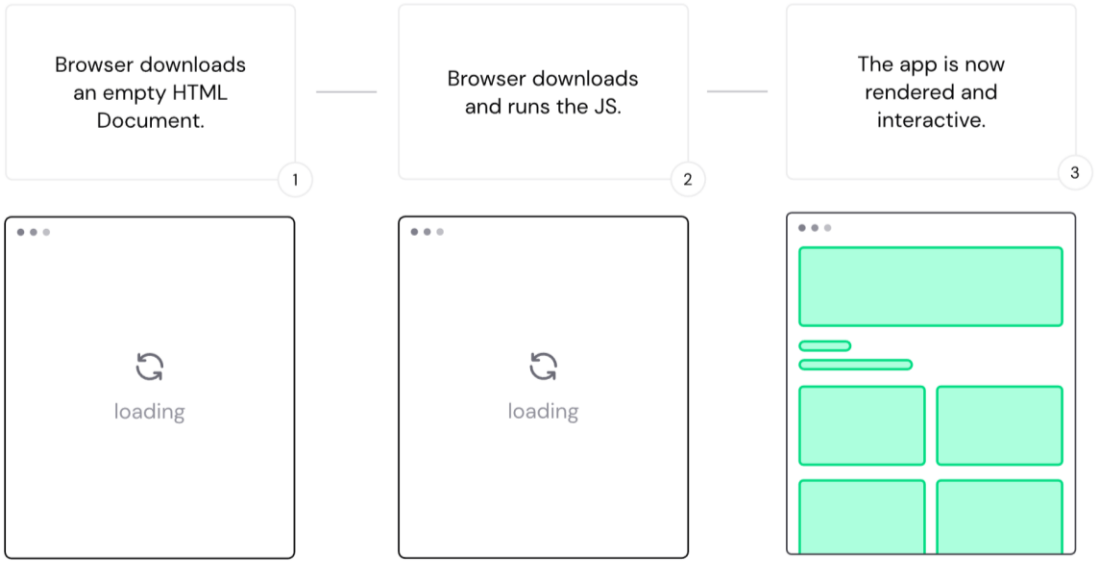

Abbildung 6: Client-side Rendering (vgl. Nuxt (Guide: Rendering Modes) o. D.: Abschn. Client-side only rendering)

Der zweite Modus ist das Universal Rendering, in dem die Stärke des Frameworks steckt. In diesem Modus wird das Rendering vom Server übernommen und es entstehen zwei Möglichkeiten, die Seiten zu rendern: einmal zur Zeit der Anfrage des Servers durch das Server Side Rendering (SSR) oder bereits vor der Anfrage des Servers als statische Dateien hinterlegt durch den integrierten SSG (vgl. Nuxt (Rendering Modes) o. D.: Abschn. Universal Rendering).

Egal ob SSR oder SSG verwendet wird, bei einer Anfrage zum Server erhält der Browser ein vollständig gerendertes HTML-Dokument, das alle Inhalte inklusive der für SEO relevanten Inhalte enthält (vgl. ebd.). Mit dabei ist auch in diesem Modus zusätzliches JavaScript, das der Browser als Nächstes ausführt (vgl. ebd.). Den Prozess, in dem das JavaScript ausgeführt wird, wird Hydration genannt (vgl. ebd.). Nach der erfolgreichen Ausführung *hydriert* die Seite von einem statischen Dokument zu einer SPA (vgl. ebd.). In diesem Zustand ist auch das clientseitige Routing aktiv, wodurch der Seitenwechsel per JavaScript gesteuert werden kann (vgl. ebd.). In Abbildung 7 ist der Prozess des Universal Renderings in drei Schritten dargestellt.

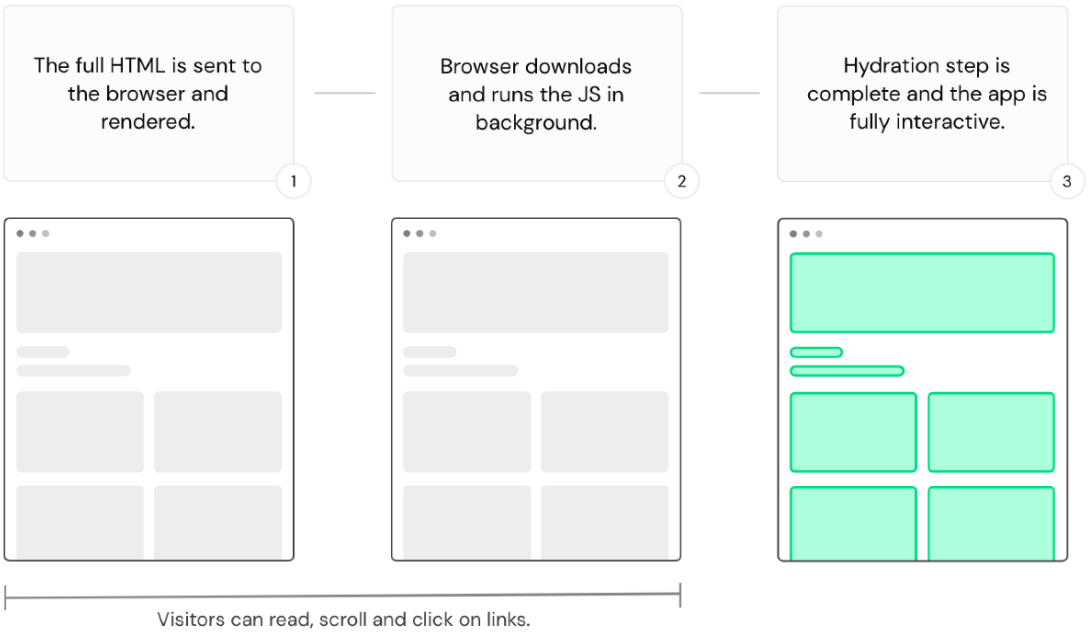

Abbildung 7: Universal Rendering (Nuxt (Guide: Rendering Modes) o. D.: Abschn. Universal Rendering)

Da es in dieser Arbeit um die redaktionelle Pflege von Websites im Jamstack geht, wird in der praktischen Ausarbeitung Nuxt im Universal Rendering Modus als SSG verwendet.

4.2.4 SFC / Single-File Components

Da Nuxt auf Vue basiert, wird auch das Konzept der Single-File Components (SFC) zur Entwicklung von Seiten und Komponenten verwendet. Eine SFC ist ein Vue-spezifisches Dateiformat, das auf `.vue` endet und das Template, die Logik und die Gestaltung in einer Datei vereint (vgl. Vue.js o. D.: Kap. Single-File Components). Die einzelnen Bereiche werden mit den HTML-ähnlichen Blöcken `<template>`, `<script>` und `<style>` getrennt (vgl. ebd.).

Auch wenn die Entwicklung der Komponenten im SFC-Format stark empfohlen wird, bietet Vue die Option an, Komponenten auch in anderen Dateiformaten wie z.B. JSX bzw. TSX zu entwickeln (vgl. Vue.js o. D.: Kap. Render Functions & JSX).

4.2.5 Hooks & Modules

Nuxt stellt sogenannte Hooks bereit, die Entwicklern einen Zugriff auf nahezu alle Prozesse des Systems ermöglichen (vgl. Nuxt (Guide: Lifecycle Hooks) o. D.: Abschn. Lifecycle Hooks). Dadurch lassen sich einzelne Abläufe des Frameworks verändern und beliebig erweitern. Hooks sind in unterschiedliche Kategorien eingeteilt (vgl. Nuxt 3 (API: Lifecycle Hooks) o. D.: Abschn. Lifecycle Hooks). Eine für den Jamstack nützliche Kategorie sind die `generate` Hooks, die den Eingriff in den Generierungsprozess des SSG ermöglichen.

Ein weiteres Merkmal von Nuxt ist, dass es über ein Erweiterungssystem mit Modulen verfügt. Module sind Funktionen, die sich automatisch in das Framework integrieren und der Reihe nach ausgeführt werden (vgl. Nuxt (Guide: Module Author Guide) o. D.: Abs. 1). Durch die Nutzung von Hooks innerhalb von Modulen lässt sich Nuxt beliebig erweitern (vgl. ebd.). Module können auch veröffentlicht und von anderen Entwicklern verwendet werden (vgl. ebd.).

4.2.6 Server Routes

In Nuxt ist die Entwicklung von serverseitigen APIs, Routes und Middlewares sehr einfach gestaltet. Das Framework überprüft die Ordner `~/server/api`, `~/server/routes` und `~/server/middleware` und registriert vorhandene Dateien automatisch als solche (vgl. Nuxt (Guide: Server Routes) o. D.: Abschn. Server Routes). Dafür stellt Nuxt eine Funktion mit dem Namen `defineEventHandler()` bereit, die immer eine Antwort zurückgibt (vgl. ebd.).

4.3 Storyblok

4.3.1 Was ist Storyblok?

Für die redaktionelle Pflege der Inhalte kommt das CMS Storyblok zum Einsatz. Storyblok ist ein API-Driven Headless CMS. Dokumente in Storyblok werden „Story“ genannt und beinhaltet immer genau einen Dokumententypen (vgl. Storyblok (Content Structures) o. D.: Abschn. Story). Ein Dokumenttyp beschreibt stets, welche spezifischen Inhalte eine Story enthalten soll (vgl. ebd.). In Storyblok gibt es verschachtelbare Komponenten, die „Bloks“ genannt werden (vgl. ebd.: Abschn. Blok (nestable Component)). Ein Blok kann entweder ein Dokumenttyp (Content Type) oder eine verschachtelbare Komponente in anderen Bloks sein (vgl. ebd.). Das Konzept der Bloks, gleicht der komponentenbasierten Entwicklung in SPA Frameworks wie Vue.

4.3.2 Blok-Bibliothek

Bloks werden von Entwicklern in der Storyblok Blok-Bibliothek erstellt. Jeder Blok kann entweder ein Dokumenttyp oder ein verschachtelbarer Blok sein (vgl. Storyblok o. D. a: Abschn. Component). Bloks enthalten Felder, die jeweils einen spezifischen Feldtyp besitzen (vgl. ebd.: Abschn. Schema). Je nach Feldtyp wird bestimmt, was der Redakteur in das Feld eingeben kann (vgl. ebd.). In Abbildung 8 sind alle in Storyblok verfügbaren Feldtypen zu sehen.

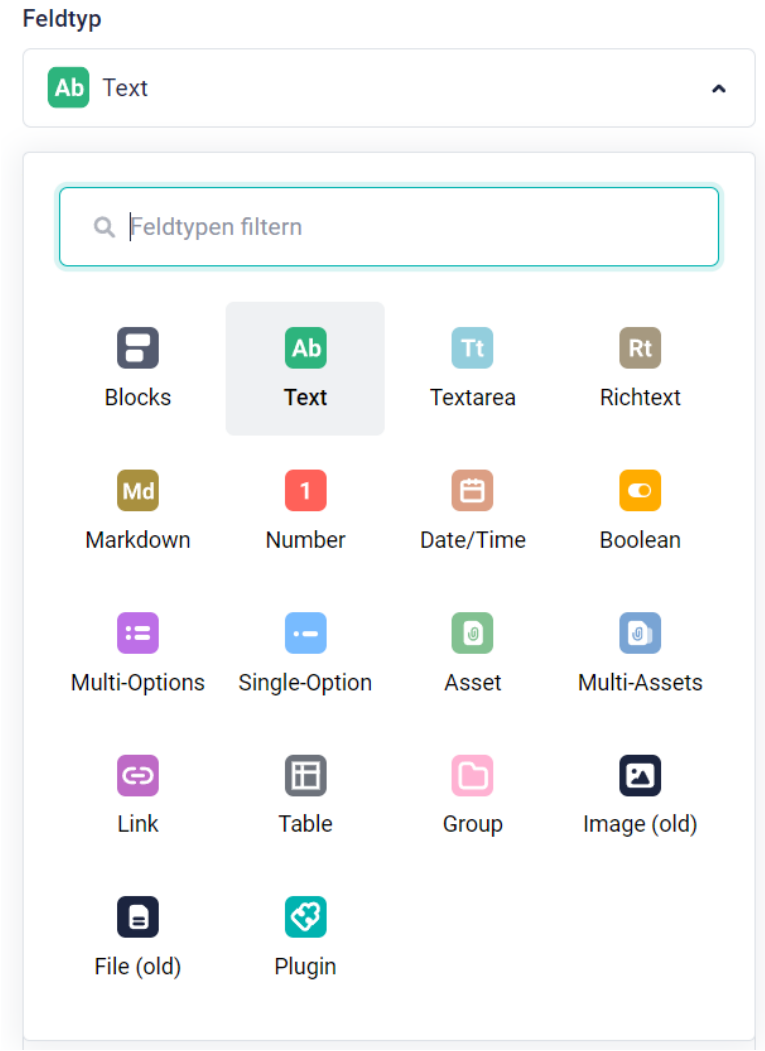

Abbildung 8: Storyblok - Feldtypen (eigene Aufzeichnung)

Alle Bloks in der Blok-Bibliothek sind wiederverwendbare Bausteine, die Redakteure verwenden können, um eine Website mit Inhalten zu füllen. Bloks können beispielsweise Sektionen einer Seite wie der Bühne oder einzelne Elemente wie ein Button darstellen.

4.3.2 Ordner Struktur

Storyblok bietet die Möglichkeit, Dokumente in Ordner zu gruppieren (vgl. Storyblok (Content Structures) o. D.: Abschn. Folder). Durch die Ordnerstruktur können Redakteure die Dokumente so anlegen, dass es die exakte Seitenstruktur der Website abbildet, wodurch die Komplexität in der Entwicklung reduziert wird. In Abbildung 9 ist eine beispielhafte Seitenstruktur einer typischen Website zu sehen.

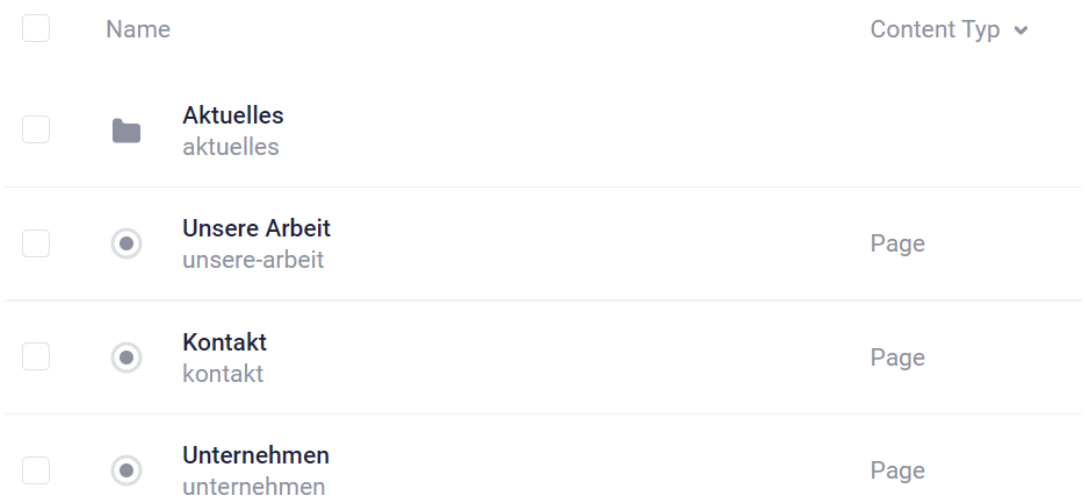

Abbildung 9: Storyblok - beispielhafte Seitenstruktur (eigene Aufzeichnung)

4.3.3 Visueller Editor

Storyblok hat neben den einfachen Feldern zum Editieren der Inhalte einen optionalen visuellen Editor. Der visuelle Editor bietet eine Live-Vorschau der jeweiligen Seite an. Inhalte, die in den Feldern geändert werden, werden auch in der Vorschau aktualisiert (vgl. Storyblok (Visual Editor) o. D.: Abschn. Bridge between the Visual Editor Preview and Content). Das ist durch die Storyblok JS Bridge, ein Skript, das im Frontend eingebunden wird und auf Änderungen in Storyblok hört, möglich (vgl. ebd.). Inhalte, die sich geändert haben, werden automatisch durch Storyblok Bridge client-seitig für die Vorschau aktualisiert (vgl. ebd.). Somit haben unveröffentlichte Änderungen keinen Effekt auf die Inhalte im Produktivsystem und dienen lediglich zur Vorschau (vgl. ebd. Abschn. Understanding the Visual Editor). In Abbildung 10 ist der visuelle Editor zu sehen.

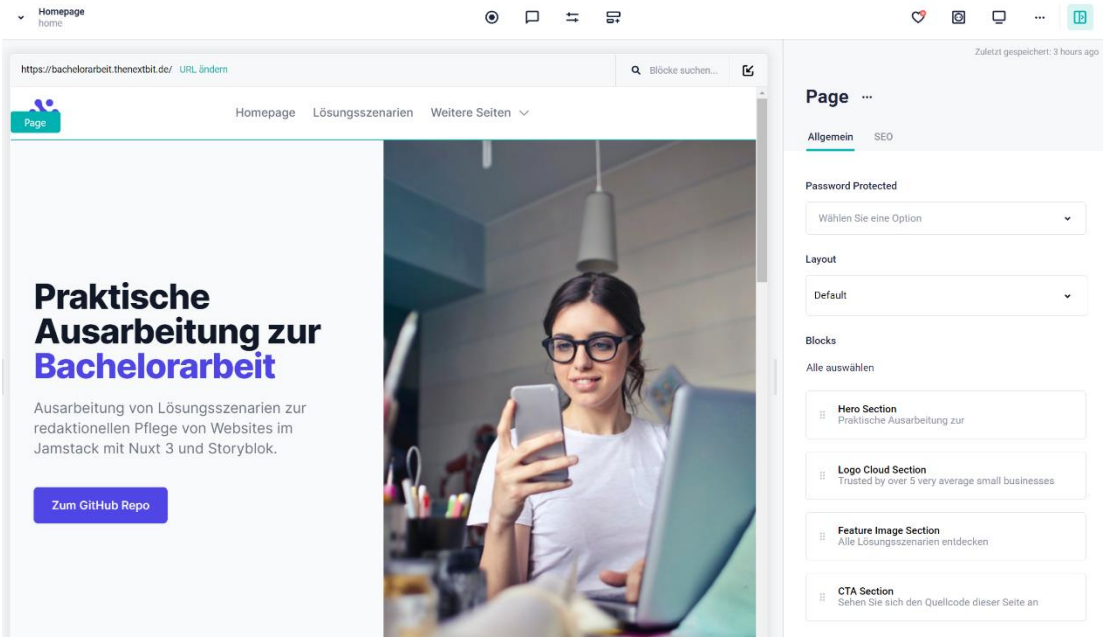

Abbildung 10: Storyblok - visueller Editor (eigene Aufzeichnung)

4.4 Netlify

4.4.1 Was ist Netlify?

Netlify ist ein Hosting-Anbieter für Jamstack-Projekte. Wie bereits in 2.1 Was ist Jamstack? erwähnt, wurde der Begriff Jamstack von Netlify vorgestellt und vorangetrieben. Netlify bietet eine voll automatisierte Entwicklungsumgebung, die keine komplexe Einrichtung benötigt. Mit wenigen Klicks können Entwickler Websites ihre Projekte hosten.

4.4.2 Netlify Build

Netlify Build ist ein Dienst zur kontinuierlichen Integration und Bereitstellung von Webprojekten (engl. Continuous Integration/Continuous Delivery, CD/CI). Um eine Jamstack-Website zu hosten, wird das Repository, in dem sich der Quellcode befindet, mit Netlify verbunden. Die Website wird darauf durch den SSG von der Netlify Infrastruktur gebaut. Wenn dieser Prozess erfolgreich abgeschlossen ist, werden die Inhalte über das CDN von Netlify weltweit zur Verfügung gestellt.

4.4.3 Praktische Ausarbeitung

Netlify wurde für die praktische Ausarbeitung als Hoster sowie als CI/CD Infrastruktur verwendet. Alle Konzepte der in 5. Lösungsszenarien im Jamstack enthaltenen Lösungsszenarien sind somit in einer Produktivumgebung getestet worden. Um das Ergebnis der praktischen Ausarbeitung visuell greifbar zu machen, ist das Endergebnis unter der Adresse https://bachelorarbeit.thenextbit.de/ erreichbar.

5. Lösungsszenarien im Jamstack

5.1 Seitengenerierung

5.1.1 Problemstellung

Seiten, die im CMS gepflegt werden, sollen nicht seitenweise und händisch von Entwicklern im Frontend implementiert werden müssen. Redakteure sollen ohne die Hilfe von Entwicklern beliebig viele neue Seiten anlegen können. Der Workflow sollte dabei voll automatisiert ablaufen. Da im Jamstack das Frontend vom CMS entkoppelt ist, muss hier eine zuverlässige Verbindung zur Seitengenerierung aufgebaut werden.

5.1.2 Lösungsweg

Wenn Redakteure neue Inhalte pflegen und veröffentlichen, muss der Generierungsprozess des SSG vom CMS ausgelöst werden, damit die neuen Inhalte generiert werden können. Um diesen Prozess ins Rollen zu bringen, gibt es Build Hooks (oder auch Deploy Hooks). Build Hooks sind eindeutige URLs (Uniform Resource Locator), die bei der CI/CD-Infrastruktur einen Bauprozess auslösen (vgl. Netlify (Build Hooks) o. D.: Abs. 1). Wird ein Build Hook vom CMS nach der Aktualisierung einer Seite aufgerufen, wird der SSG gestartet. Da der SSG von sich selbst nicht weiß, welche Seiten existieren, muss vor der Generierung ein Prozess zum Abruf aller zu generierenden Seiten durch die API des CMS stattfinden. Wenn dem SSG alle Seiten bekannt sind, kann der Generierungsprozess beginnen.

5.1.3 Umsetzung in Nuxt

Die klassische Vorgehensweise, um in Nuxt Seiten anzulegen, ist im `pages` Verzeichnis eine Datei mit dem Namen der Seite wie z. B. `kontakt.vue` anzulegen. Der Dateiname entspricht dem Pfad, der in der URL eingegeben wird: `meine-domain.de/kontakt`. Was schnell klar wird, ist, dass dieses Verfahren einen Entwickler benötigt, der jede neue Seite manuell anlegen muss.

Um die Seiten dynamischer zu gestalten, gibt es in Nuxt die Möglichkeit, dynamische Seiten zu erstellen, die je nach Pfad unterschiedliche Inhalte

haben können. Dynamische Seiten können durch eckige Klammern im Dateinamen angelegt werden: `pages/[uid].vue` (vgl. Nuxt (Guide: Pages directory) o. D.: Abschn. Dynamic Routes). Die Bezeichnung `uid` innerhalb der Klammern wird der Seite als dynamischer Parameter, der abhängig vom aktuellen Pfad ist, übergeben (vgl. ebd.). Wenn der Pfad `/dieser-pfad` adressiert wird, ist der Wert von dem Parameter `uid` gleich `dieser-pfad`.

Die dynamische Seite `pages/[uid].vue` würde aber nur die Seiten abfangen, die sich hierarchisch auf derselben Ebene befinden. Seiten wie `meine-domain.de/foo/bar` können so nicht berücksichtigt werden, da sie sich auf einer tieferen Ebene (verschachtelt) befinden. Damit die dynamische Seite auf allen Ebenen funktioniert, müssen vor dem Parameter im Dateinamen drei Punkte stehen: `[...uid].vue` (vgl. ebd.).

In der dynamischen Seite ist es möglich, den Pfad der aktuellen Seite durch den Parameter `path` innerhalb der globalen Funktion `useRoute()` abzufragen (vgl. Nuxt (API: useRoute) o. D.: Abschn. Example). Diesen Pfad kann man programmatisch in die Abfrage der benötigten Seite vom CMS integrieren.

```
// ./pages/[...uid].vue

<script setup lang="ts">
  const route = useRoute();
  const storyblokApi = ...;
  const { data } = await useAsyncData(route.path, async () => {
    const res = await storyblokApi.get(
      'cdn/stories/' + route.path,
      { ... }
    );
    return res.data.story;
  });
</script>
```

Um den SSG mitzuteilen, welche Seiten alle im Generierungsprozess miteinbezogen werden sollen, wurde ein Modul mit dem Namen `dynamic-routes` erstellt. Dieses Modul hat die Aufgabe, alle Seiten vom CMS abzufragen und sie im Anschluss in den Generierungsprozess einzubinden. Das Modul

verwendet den `nitro:config` Hook, um bereits vor dem Start des SSG die Informationen ins System einzuspeisen.

```
// ./modules/dynamic-routes.ts

import { defineNuxtModule } from '@nuxt/kit';

export default defineNuxtModule({
  setup(options, nuxt) {
    nuxt.hook('nitro:config', async (nitroConfig) => {
      const storyblokApi = ...;
      const routes: string[] = storyblokApi.get(...).then(...);
      routes.forEach((path) => nitroConfig.prerender.routes.push(path));
    });
  }
});
```

Wenn der Generierungsprozess des SSG startet, werden alle Seiten, die durch das `dynamic-routes` Modul injiziert wurden, in der dynamischen Seite jeweils als Parameter übergeben und erstellt.

5.2 Suchmaschinenoptimierung

5.2.1 Problemstellung

Die Suchmaschinenoptimierung (SEO) ist ein wichtiger Bestandteil einer erfolgreichen Website, die meistens von Marketing-Experten übernommen wird. Es ist zwischen zwei Arten der SEO zu unterscheiden: Onpage-Optimierung und Offpage-Optimierung (vgl. Horster et al. 2022: 98). Onpage-Optimierungen sind inhaltliche und strukturelle Maßnahmen, die direkt an der Website selbst vorgenommen werden können (vgl. ebd.: 99), während die Offpage-Optimierung mehr um Themen wie dem Aufbau von Verlinkungen (Backlinks) im WWW geht (vgl. ebd.: 110). Im Folgenden wird es nur um die Onpage-Optimierung gehen. Es wird gezeigt, wie Entwickler suchmaschinenrelevante Meta-Informationen im CMS und Frontend implementieren und Redakteure einfach und intuitiv Onpage-Optimierungen umsetzen können.

Zu den wichtigsten Meta-Informationen im Head-Bereich einer Website gehört der Titel und die Beschreibung (vgl. Horster et al. 2022: 100-101). Diese haben direkten Einfluss auf die Suchmaschinen (vgl. ebd.). Ein weiterer Aspekt, auf den eingegangen werden soll, ist das Open Graph protocol (OGP). OGP ist besonders wichtig für Seiten, die in den sozialen Medien geteilt werden. OGP Meta-Daten wie Titel, Beschreibung und Vorschaubild werden verwendet, um z. B. eine Vorschau der Seite zu erzeugen.

5.2.2 Lösungsweg

Im CMS legen Entwickler bestimmte suchmaschinenrelevante Felder an, die von Redakteuren seitenweise gepflegt werden können. Beim Pre-rendering des SSG werden die Daten dieser Felder in den Head-Bereich des Markups der entsprechenden Seiten geschrieben.

5.2.3 Umsetzung in Storyblok

Die Felder für die Meta-Daten werden in einem beliebigen, nicht verschachtelbaren Dokumenttyp in Storyblok umgesetzt. Der Grund ist, dass die SEO zentral und nur einmal pro Seite implementiert werden soll. Um die SEO vom Hauptinhalt der Seite zu trennen, kann eine neue Registerkarte angelegt werden, die in diesem Beispiel „SEO“ genannt wird. In dieser Registerkarte werden die in 5.2.1 Problemstellung genannten Felder für Titel, Beschreibung und Vorschaubild angelegt. Für die technischen Namen werden die Felder mit einem `seo_` Präfix versehen, damit sie sich von anderen ähnlichen Feldern wie `title` oder `description` unterscheiden. Das Ergebnis der Registerkarte ist in Abbildung 11 zu sehen.

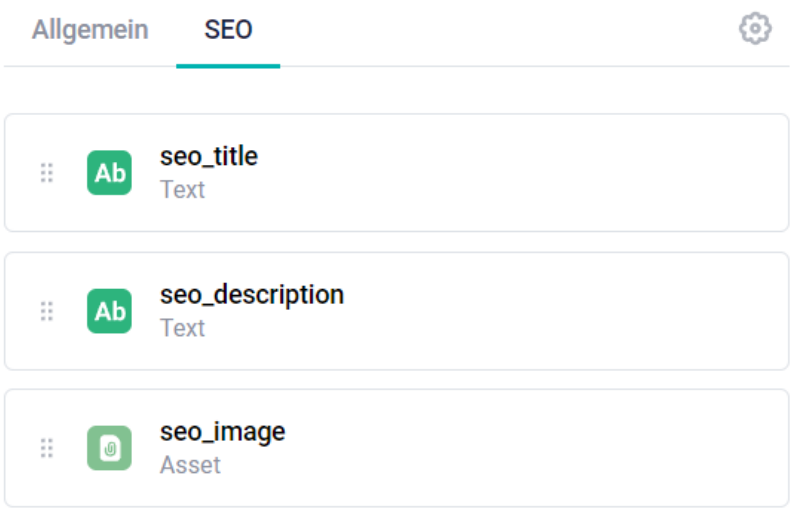

Abbildung 11: Storyblok - SEO-Registerkarte aus der Entwicklersicht (eigene Aufzeichnung)

Wie in Abbildung 12 zu sehen ist, können Redakteure in der Registerkarte „SEO“ die Meta-Informationen der jeweiligen Seiten pflegen.

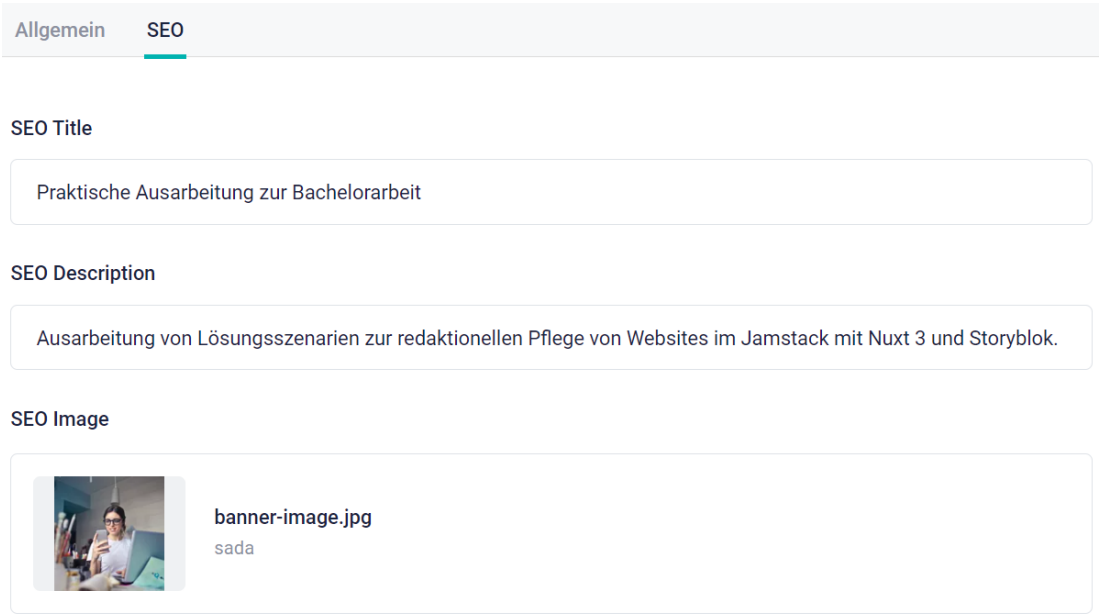

Abbildung 12: Storyblok - SEO-Registerkarte aus der Redakteurssicht (eigene Aufzeichnung)

5.2.4 Umsetzung in Nuxt

Es gibt zwei Wege Meta-Daten einer Seite in Nuxt festzulegen. Eine Möglichkeit ist es, Meta-Komponenten wie `<Title>` und `<Meta>` innerhalb der `<Head>` Komponente einer beliebigen Seite einzusetzen (vgl. Nuxt (Guide: Head Management) o. D.: Abschn. Meta Components). Meta-Komponenten werden von Nuxt global bereitgestellt und sind den nativen HTML Meta-Tags ähnlich

(vgl. ebd.). Für dieses Beispiel kommt jedoch eine andere Methode zum Einsatz. Die SEO relevanten Meta-Daten können auch durch die `useHead()` Methode, die innerhalb von `<script setup>` aufgerufen werden kann, festgelegt werden (vgl. ebd.: useHead Composable). In `useHead()` kommt ein Objekt, dass alle möglichen Meta-Tags wie `title`, `meta`, `link` oder `htmlAttrs` enthalten kann (vgl. ebd.). Im Folgenden die Umsetzung der SEO mit der `useHead()` Methode in der dynamischen Seite.

```
// ./pages/[...uid].vue

<script setup>
  const { data } = await useAsyncData(...); // Story from Storyblok
  useHead({
    title: data.value.content.seo_title,
    meta: [
      {  name: 'description',  content: data.value.content.seo_description  },
      {  name: 'og:title',  content: data.value.content.seo_title  },
      {  name: 'og:description',  content: data.value.content.seo_description },
      {  name: 'og:image',  content: data.value.content.seo_image.filename  }
    ]
  });
</script>
```

5.3 Layouts

5.3.1 Problemstellung

Layouts legen die Gestaltung einer Seite fest. Redakteure sollen die Möglichkeit haben, verschiedene Seitenlayouts anzuwenden.

5.3.2 Lösungsweg

Redakteure sollen aus einer Reihe an vordefinierten Layouts im CMS wählen und sie seitenweise verwenden können. Damit die Layouts im Frontend angewendet werden können, müssen die Entwickler alle vorhandenen Layouts im Vorfeld implementiert haben.

5.3.3 Storyblok

Eine Möglichkeit ist es, Dokumenttypen anzulegen, die einem bestimmten Seitenlayout entsprechen. Das Problem an dieser Vorgehensweise ist, dass Redakteure im Nachhinein den Dokumenttypen einer Seite nicht ändern können und somit auf das eine Layout beschränkt wären.

Die flexibelste Lösung ist die Implementierung eines Auswahlfeldes mit unterschiedlichen Layouts, die in jeder Seite zur Verfügung steht. Daraus ergibt sich die Freiheit, dass jede Seite ein anderes Layout annehmen kann. Hierzu kann der Feldtyp „Single-Option“ verwendet werden, welcher dem Redakteur nur eine Auswahlmöglichkeit bietet. In Abbildung 13 ist das Auswahlfeld mit zwei Layouts aus der praktischen Arbeit zu sehen.

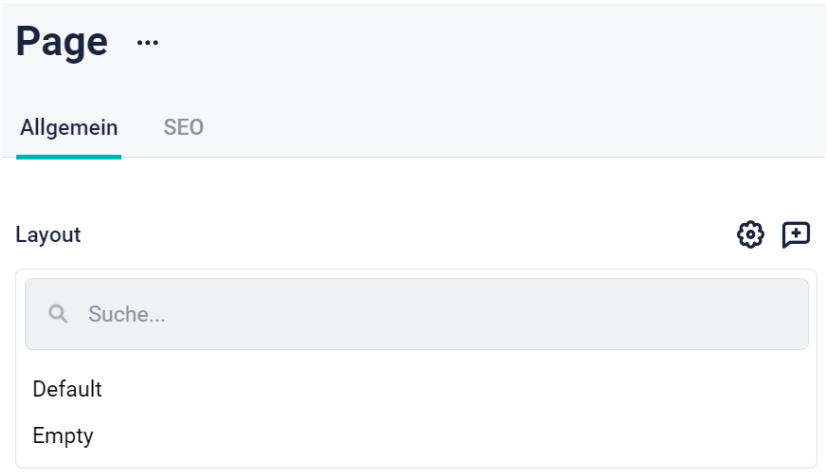

Abbildung 13: Storyblok - Layout-Feld (eigene Aufzeichnung)

Während das Layout „Default“ eine Navigation und ein Footer enthält, ist das Layout „Empty“ einfach leer.

5.3.4 Umsetzung in Nuxt

Um Seitenlayouts dynamisch und seitenweisen anwenden zu können, muss das Standard-Seitenlayout in der `definePageMeta()` Methode, die innerhalb von `<script setup>` einer Seite aufgerufen werden kann, deaktiviert werden.

Um ein Seitenlayout anzuwenden, wird die `<NuxtLayout>` Komponente verwendet, die in Nuxt global zur Verfügung steht. Die Komponente benötigt

das Feld `name`, um ein Layout festlegen zu können, welches von dem Layout-Auswahlfeld aus dem CMS kommt. Wichtig ist, dass der Inhalte der Seite innerhalb von `<NuxtLayout>` platziert wird.

```
// ./pages/[...uid].vue

<template>
  <div>
    <!-- Layout -->
    <NuxtLayout :name="data.content.layout || 'default'">
      <!-- Page Content -->
    </NuxtLayout>
  </div>
</template>

<script setup lang="ts">
  definePageMeta({ layout: false });
  const { data } = await useAsyncData(...);
</script>
```

Die Layouts, die im CMS definiert sind, müssen gleichnamig im Fronentend implementiert werden. Im `layouts` Verzeichnis können Layouts in Form von gewöhnlichen Vue-SFC angelegt werden. Nuxt registriert die Layouts aus diesem Ordner automatisch zur Verwendung in der `<NuxtLayout>` Komponente. Im Folgenden ist der Code des Layouts „Default“ aus der praktischen Arbeit zu sehen.

```
// ./layouts/default.vue

<template>
  <div class="layout--default">
    <Navigation />
    <slot />
    <Footer />
  </div>
</template>
```

5.4 Navigation

5.4.1 Problemstellung

Die Navigation einer Website hilft dem Besucher der Seite dabei, die relevanten Inhalte zu finden und verbindet die unterschiedlichen Seiten miteinander (vgl. Zeta Producer 2018: Abs. 2). Sie kann auch anzeigen, auf welcher Seite sich der Besucher aktuell befindet (vgl. ebd.: Abs. 3).

Eine erfolgreiche Navigation hängt auch von der Anzahl der der Links ab (vgl. ebd.: Abschn. Die Navigation zur Übersicht für die Webseiten-Besucher). Während zu viele Links den Besucher überfordern können, findet er möglicherweise bei zu wenigen das Ziel nicht und verlässt die Seite (vgl. ebd.). In einer flexiblen Architektur sollte der Redakteur die Freiheit haben, die Links einer Navigation redaktionell frei zu pflegen.

5.4.2 Lösungsweg

Für die Navigation wird im CMS ein eigenständiges Dokument zur zentralen Verwaltung einer bestimmten Navigation angelegt. Dieses Dokument wird beim Generierungsprozess des SSG abgerufen und der Inhalt dementsprechend präsentiert. In diesem Dokument können neben den Links weitere Elemente wie z. B. das Logo einer Hauptnavigation redaktionell gepflegt werden.

5.4.3 Umsetzung in Storyblok

Wie bereits in 5.4.2 Lösungsweg erklärt, wird für die Verwaltung der Navigation eine eigenständige Story (Dokument) zur zentralen Verwaltung angelegt.

Damit eine verschachtelte Navigation möglich ist, muss der Entwickler in der Blok-Bibliothek Navigationselemente erstellen, die je nach Szenario weitere Navigationselemente enthalten können. Bei Bedarf können weitere Felder wie das Logo einer typischen Navigation gepflegt werden.

In den Abbildungen Abbildung 14, Abbildung 15 und Abbildung 16 ist die verschachtelte Navigation der praktischen Arbeit aus der Redakteurssicht zu sehen.

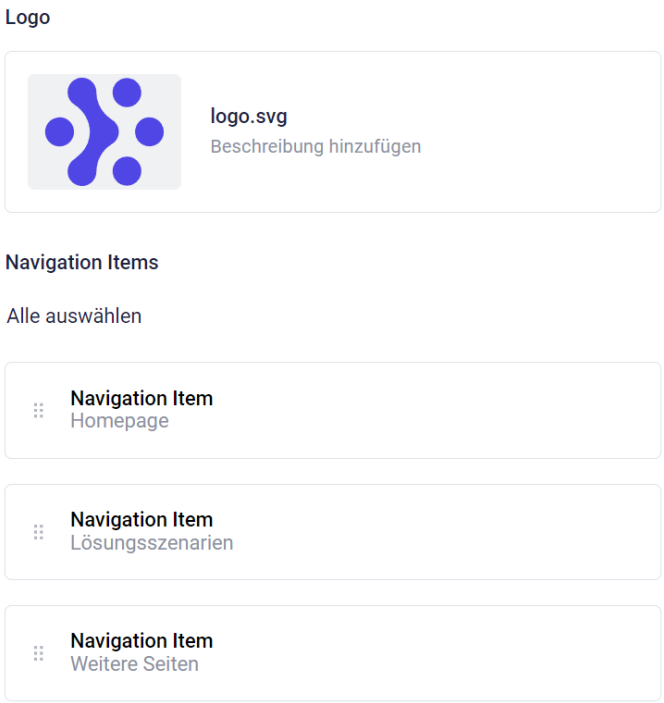

Abbildung 14: Storyblok - Navigation Übersicht (eigene Aufzeichnung)

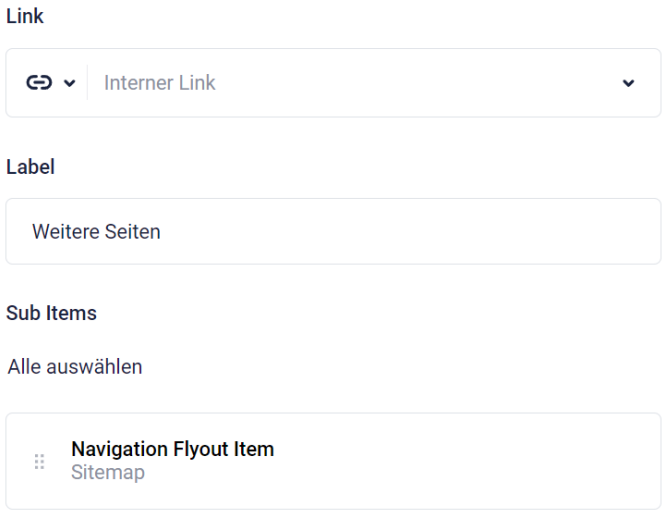

Abbildung 15: Storyblok - Navigation Ebene 1 (eigene Aufzeichnung)

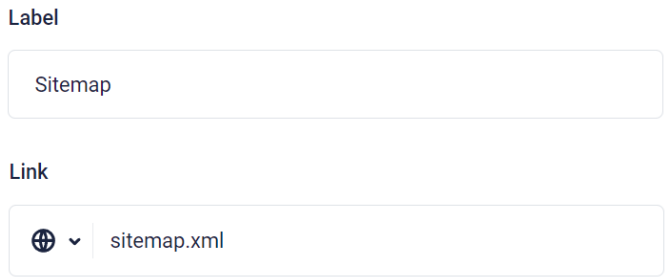

Abbildung 16: Storyblok - Navigation Ebene 2 (eigene Aufzeichnung)

5.4.4 Umsetzung in Nuxt

Für die Navigation wurde eine gleichnamige Komponente `Navigation.vue` im Verzeichnis `/components` erstellt, die in das Standard-Layout `/layouts/default.vue` integriert wurde. Die Komponente lädt das passende Dokument für die Navigation aus dem CMS und stellt die Inhalte dar. Man muss dabei berücksichtigen, dass bei einer verschachtelten Navigation, wie in 5.4.3 Umsetzung in Storyblok erklärt, die zugehörigen verschachtelbaren Komponenten (Bloks) implementiert werden müssen.

5.5 Weiterleitungen

5.5.1 Problemstellung

Es kann sein, dass sich im Laufe der Lebenszeit einer Website bestimmte URLs verändern. Um eine alte URL zu einer neuen umzuleiten, werden sogenannte Weiterleitungen verwendet (vgl. Brockbank 2021: Abs. 1). Weiterleitungen geben Suchmaschinen und dem Browser die Information, wohin sich der Inhalt einer alten URL bewegt hat (vgl. ebd.: Abschn. Was sind URL-Redirects und Weiterleitungen?). Daraus ergibt sich der Vorteil, dass man nicht auf einer Fehlerseite landet, sondern zum neuen Ziel weitergeleitet wird, wodurch die Nutzererfahrung nicht eingeschränkt wird und Suchmaschinen die URL aktualisieren können (vgl. ebd.: Abschn. Weiterleitungen und Nutzerzufriedenheit). Welche Pfade wohin umgeleitet werden sollen, können Entwickler oder Redakteure je nach Bedarf einrichten.

5.5.2 Lösungsweg

Für die Verwaltung von Weiterleitungen wurden zwei Möglichkeiten gefunden. Weiterleitungen können entweder zentral oder dezentral verwaltet werden. Bei der dezentralen Variante müsste jedes Dokument eine Option für Weiterleitungsregeln enthalten. Entweder müsste immer definiert werden, aus welcher oder zu welcher URL weitergeleitet werden soll. Wenn zu einer anderen URL weitergeleitet werden soll, darf das alte Dokument nicht gelöscht werden, um die Weiterleitung weiterhin aufrechterhalten zu können. Daraus lässt sich beim dezentralen Ansatz schließen, dass es schnell unübersichtlich werden kann. Die Verwaltung aller Weiterleitungen in einem zentralen Dokument erscheint als die übersichtlichere Wahl.

Für den zentralen Ansatz wird wie in 5.4.2 Lösungsweg des Lösungsszenarios 5.4 Navigation im CMS ein für den Anwendungsfall spezifisches und zentrales Dokument angelegt. Dieses Dokument wird beim Generierungsprozess des SSG abgerufen und eine Datei für die CD/CI-Infrastruktur bzw. Server erstellt, um die Weiterleitungen serverseitig zu verarbeiten.

5.5.3 Umsetzung in Storyblok

Wie bereits in 5.5.2 Lösungsweg beschrieben, muss für die Verwaltung der Weiterleitungen ein Dokument zur zentralen Verwaltung angelegt werden. Das Dokument enthält ein Feld für die Weiterleitungsregeln für den Server. Eine Weiterleitungsregel hat die Felder für die alte URL, die neue URL und der Statuscode der Weiterleitung. Gängige Statuscodes für Weiterleitungen sind 301 (dauerhaft umgezogene URL) und 302 (vorübergehend umgezogene URL). In Abbildung 17 ist eine beispielhafte Weiterleitung aus der Redakteurssicht zu sehen.

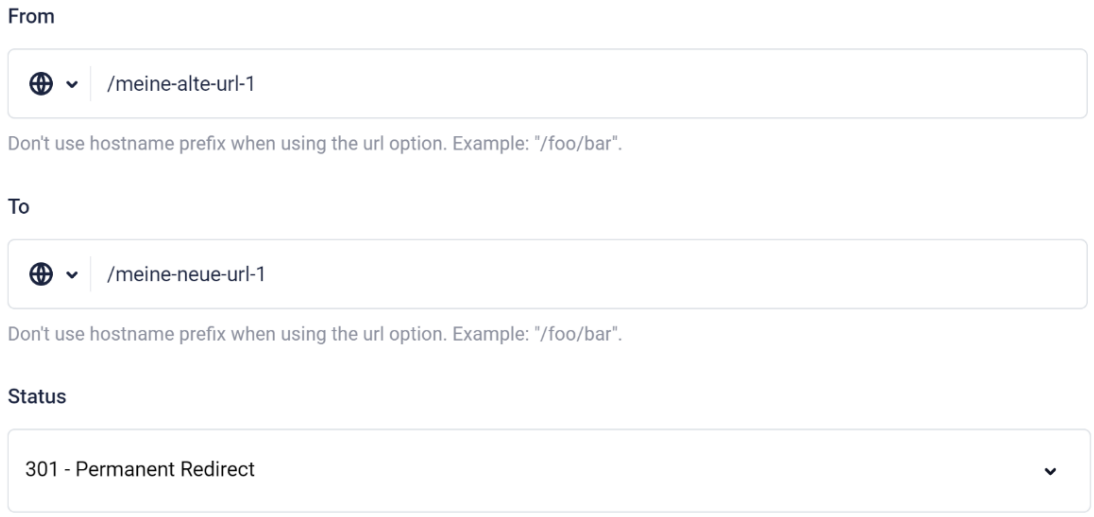

Abbildung 17: Storyblok – 301-Weiterleitung (eigene Aufzeichnung)

5.5.4 Umsetzung in Nuxt

Da in dieser Arbeit Netlify als CD/CI-Infrastruktur verwendet wird, werden die Weiterleitungen durch die Server von Netlify bearbeitet. Netlify sucht hierfür nach dem erfolgreichen Generierungsprozess des SSG automatisch nach einer Datei mit dem Namen `_headers` (vgl. Netlify (Redirects and rewrites) o. D.: Abschn. Redirects and rewrites). Diese Datei enthält Weiterleitungsregeln mit einer alten und neuen URL sowie dem HTTP-Statuscode der Weiterleitung (vgl. ebd.: Abschn. Syntax for the _redirects file).

Um die Datei für die Weiterleitungen zu erstellen, wurde das Modul `netlify-redirects` im Verzeichnis `/modules` entwickelt. Das Modul ruft im Generierungsprozess des SSG das Dokument für die Weiterleitungen aus dem CMS ab, formatiert die Weiterleitungsregeln und speichert sie in einer neuen `_redirects` Datei ab. Im Folgenden ein Beispiel für das Ergebnis einer `_headers` Datei für zwei Weiterleitungen, das von `netlify-redirects` Modul erstellt wurde.

```
// _redirects

/meine-alte-url-1  /meine-neue-url-1  301
/meine-alte-url-2  /meine-neue-url-2  302
```

5.6 Geschützte Bereiche

5.6.1 Problemstellung

Geschützte Bereiche sind Teile einer Website, die von den Besuchern eine Authentifizierung verlangen. Um geschützte Bereiche zu realisieren, kommt im Rahmen dieser Arbeit die Authentifizierungsmethode HTTP Basic Auth zum Einsatz. HTTP Basic Auth ist ein etabliertes Standardverfahren, um bestimmte Verzeichnisse im Server mit einer Authentifizierung zu schützen (vgl. dataliquid GmbH 2018: Abschn. Was ist HTTP Basic Auth?). Bei einer unautorisierten Anfrage auf einen geschützten Bereich sendet der Server den Status Code 401 zurück (vgl. ebd.: Abschn. Die Funktionsweise von Basic Auth). Dieser Code teilt dem Browser mit, dass eine Authentifizierung erforderlich ist. In diesem Schritt wird der Benutzer aufgefordert, Benutzernamen und Passwort einzugeben (vgl. ebd.). Ist die Eingabe richtig, gewährt der Server den Zugriff auf die Inhalte.

5.6.2 Lösungsweg

Redakteure sollen in einem zentralen Dokument im CMS festlegen, welche Bereiche geschützt sein sollen. Um die geschützten Bereiche umzusetzen, muss im Generierungsprozess des SSG eine Datei mit Authentifizierungsregeln für den Server ausgegeben werden. Wie die Datei heißt und wie sie konfiguriert sein muss, hängt von der verwendeten Server-Infrastruktur ab.

5.6.1 Umsetzung in Storyblok

In Storyblok wird ein zentrales Dokument zur Verwaltung der geschützten Bereiche erstellt. Das Dokument enthält ein Feld für die Authentifizierungsregeln. Eine Authentifizierungsregel enthält Felder für den geschützten Bereich, den Benutzernamen und das Passwort. Daten wie Benutzernamen und Passwort lassen sich in Storyblok optional als globale Datenquelle definieren, wodurch Redakteure dann nur einen Benutzer aus einer Liste auswählen müssten. In Abbildung 18 ist ein beispielhafter geschützter Bereich aus der Sicht des Redakteurs zu sehen.

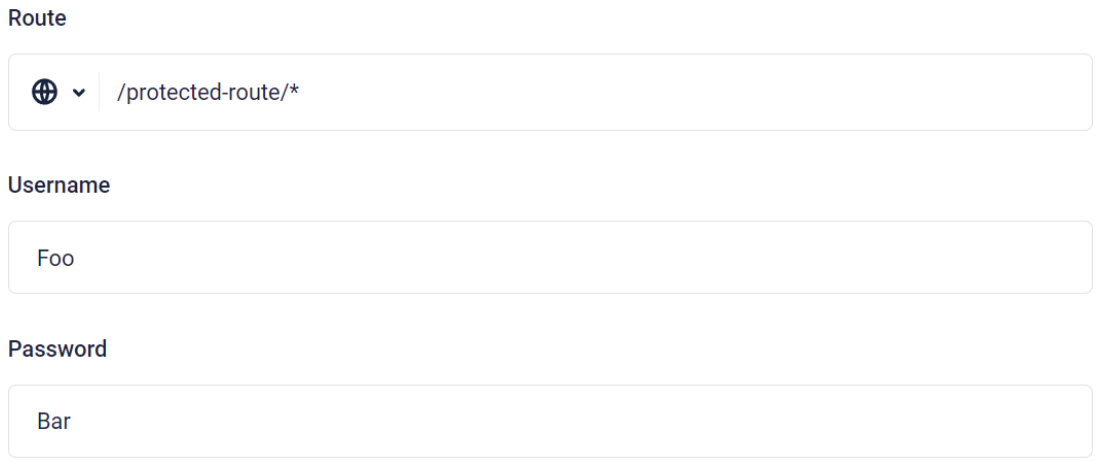

Abbildung 18: Storyblok – passwortgeschützter Bereich (eigene Aufzeichnung)

5.6.1 Umsetzung in Nuxt

Ähnlich wie in 5.5.4 Umsetzung in Nuxt des Lösungsszenarios 5.5 Weiterleitungen muss eine Datei mit Anweisungen für die CD/CI-Infrastruktur, was in diesem Fall Netlify ist, erstellt werden. Diese Datei hat den Namen `_headers` und enthält für jeden geschützten Bereich die URL und mindestens einen Benutzernamen und ein Passwort.

Für die Erstellung der Datei wurde das Modul `netlify-password-protection` erstellt. Das Modul ruft im Generierungsprozess des SSG das Dokument für die geschützten Bereiche aus dem CMS ab, formatiert die Authentifizierungsregeln und speichert es in einer neuen `_headers` Datei ab. Im Folgenden ein Beispiel für das Ergebnis einer `_headers` Datei für einen geschützten Bereich, das von diesem Modul erstellt wurde.

```
// _headers

/geschützer-bereich/*
  Basic-Auth: Foo:Bar
```

5.7 Sitemap

5.7.1 Problemstellung

Eine Sitemap stellt eine Übersicht über welche Inhalte sich auf einer Website befinden dar und zeigt, wie sie miteinander verbunden sind (vgl. Exposure Ninja (Shinobi) 2021: Abschn. What Is a Sitemap?). Es hilft den

Suchmaschinen dabei, eine Website zu durchsuchen und zu indexieren (vgl. Backlinko 2020: Abschn. What Is a Sitemap?). Aus diesem Grund sollte eine Sitemap auch als Teil der SEO berücksichtigt werden.

Die Sitemap wird für gewöhnlich im für Suchmaschinen zugänglichen XML-Format ausgegeben (vgl. Exposure Ninja (Shinobi) 2021: Abschn. What Is a Sitemap?). Es gibt auch Sitemaps im HTML-Format, bei dem der Fokus auf der Visualisierung der Inhalte steht (vgl. ebd.: Abschn. What Is an HTML Sitemap?). Im Folgenden wird es um die Erstellung einer XML-Sitemap gehen, da HTML-Sitemaps nicht häufig vorkommen.

5.7.2 Lösungsweg

Die von Redakteuren veröffentlichten Seiten im CMS können automatisch in die Sitemap miteinbezogen werden. Im Generierungsprozess des SSG müssen die Seiten aus dem CMS aufgerufen und in eine Sitemap verarbeitet werden. Dieser Prozess kann vollständig automatisiert werden.

5.7.1 Umsetzung in Nuxt

Um eine Sitemap in Nuxt umzusetzen, gibt es zwei naheliegende Methoden. Die Sitemap lässt sich als Modul und als Teil des Generierungsprozesses oder als dedizierte Server Route integrieren.

Bei einem Modul wäre die Vorgehensweise ähnlich wie in den vergangenen Lösungsszenarien. Dazu wurde das Modul `sitemap` entwickelt. Das Modul fragt vom CMS alle veröffentlichten Seiten ab und formatiert die Daten zu einer Sitemap. Das Ergebnis wird als Datei beispielsweise mit dem Namen `sitemap.xml` ausgegeben.

Als Alternative wurde die Server Route `/sitemap.xml` erstellt. Eine Server Route ist eine Seite, die nur serverseitig verarbeitet werden kann. Die Server Route macht das gleiche wie das Modul jedoch nur zur Zeit der Anfrage zum Server. Das bietet einen großen Vorteil in der Entwicklererfahrung, da Änderungen im Code direkt getestet werden können. Beim Modul ist die Sitemap nur als Teil des Generierungsprozesses des SSG konzipiert, wodurch nach jeder Änderung ein Generierungsprozess gestartet werden muss, um die

Änderungen zu testen. Damit die Server Route auch beim Generierungsprozess zur statischen Generierung berücksichtigt werden kann, muss der Pfad zur Server Route in den Generierungsprozess integriert werden.

In der Abbildung 19 ist eine Sitemap zu sehen, die durch die Server Route der praktischen Arbeit generiert wurde.

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
xmlns:news="http://www.google.com/schemas/sitemap-news/0.9"
xmlns:xhtml="http://www.w3.org/1999/xhtml" xmlns:image="http://www.google.com/schemas/sitemap-
image/1.1" xmlns:video="http://www.google.com/schemas/sitemap-video/1.1">
   <url>
      <loc>https://bachelorarbeit.thenextbit.de/solutions</loc>
      <priority>1.0</priority>
   </url>
   <url>
      <loc>https://bachelorarbeit.thenextbit.de/</loc>
      <priority>1.0</priority>
   </url>
</urlset>
```

Abbildung 19: XML-Sitemap (eigene Aufzeichnung)

5.8 Medienverwaltung

Dieses Lösungsszenario wird sich nur auf das CMS beziehen, da die Medienverwaltung zur redaktionellen Pflege einer Website zum Aufgabenbereich des CMS gehört.

Die Medienverwaltung ist, wie in 3.2.2 Verwaltung bereits erklärt, ein essenzieller Bestandteil eines CMS. Es ist zu beachten, dass die Medienverwaltung je nach Wahl des CMS vom Funktionsumfang unterschiedlich ausfallen kann. Im Folgenden wird es um die Medienverwaltung in Storyblok gehen.

Storyblok kommt mit einer integrierten Medienverwaltung zur redaktionellen Pflege aller Medieninhalte einer Website. Medieninhalte können eingefügt, editiert und gelöscht werden. In Abbildung 20 ist der Asset-Manager von Storyblok zu sehen.

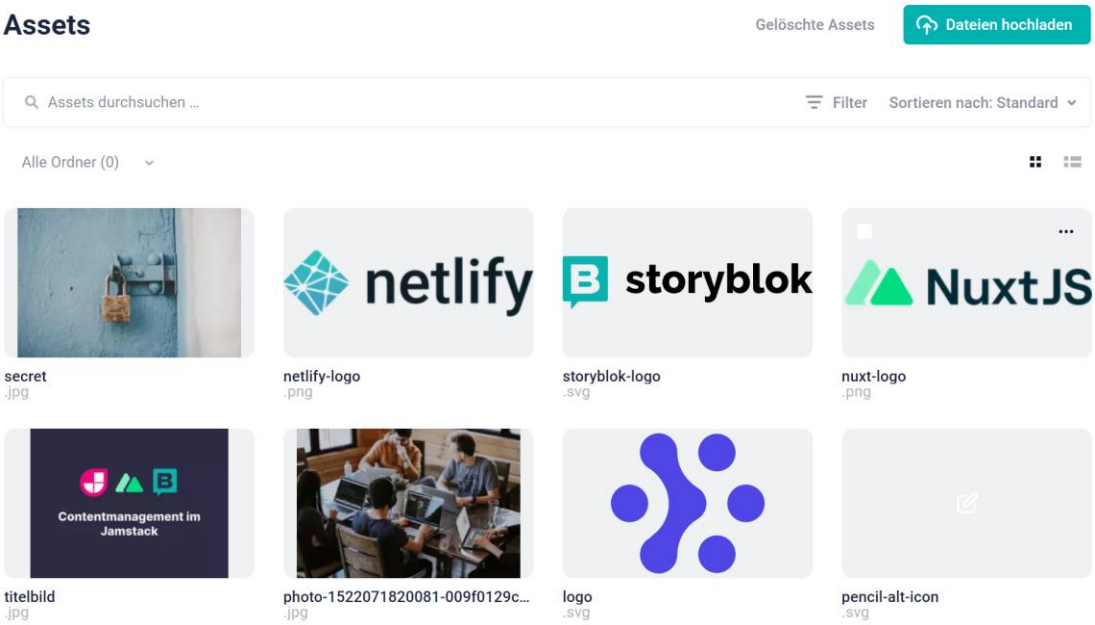

Abbildung 20: Storyblok – Asset-Manager (eigne Aufzeichnung)

Eine zusätzliche Funktion, die Storyblok bietet, ist die Bildmanipulation. Redakteure können beispielsweise die Bilder direkt im Asset-Manager in ein neues Format ausschneiden oder die Helligkeit anpassen. In Abbildung 21 ist der Bildeditor im Einsatz zu sehen.

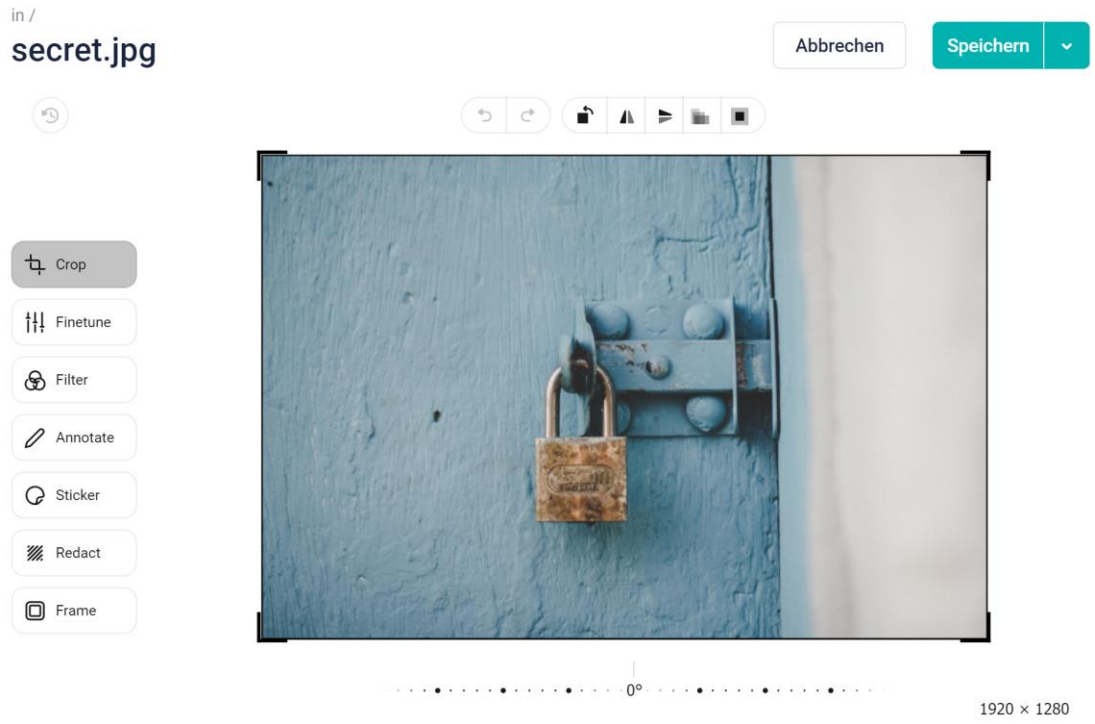

Abbildung 21: Storyblok - Bildeditor (eigene Aufzeichnung)

Eine weitere hilfreiche Funktionalität in Bezug auf die Bildoptimierung ist der Storyblok Image Service. Dieser Dienst ermöglicht es, Bilder bei der Anfrage durch URL-Parameter in ein gewünschtes Format auszugeben (vgl. Storyblok (Image Service) o. D.: Abs. 1). Das hilft besonders bei der SEO, um von einer einzigen Bildquelle verschiedene responsive Bilder für alle Gerätetypen auszugeben. Neben dem Bildformat und der Bildauflösung lässt sich das Bild auch ähnlich wie im Bildeditor in weiteren Bereichen der Bildbearbeitung anpassen (vgl. Storyblok (Image Service) o. D.: Abschn. Contents).

5.9 Rollen und Rechte

Dieses Lösungsszenario wird sich nur auf das CMS beziehen, da die Verwaltung von Rollen und Rechte zur redaktionellen Pflege einer Website zum Aufgabenbereich des CMS gehört.

In Storyblok können bestimmte Aktionen und der Zugang zu bestimmten Bereichen mit Rollen und Rechten auf spezifische Nutzer eingeschränkt werden (vgl. Storyblok (Roles & Permissions) o. D.: Abs. 1). Die Möglichkeit, Rollen und Rechte zu verteilen, gehört zum Funktionsumfang von Storyblok. Diese Funktion kann je nach verwendetem CMS-Anbieter unterschiedlich ausfallen.

Storyblok bietet von Beginn an die zwei Rollen Administrator und Redakteur. Während der Administrator das gesamte Projekt uneingeschränkt verwalten darf, kann der Redakteur nur Inhalte erstellen, aktualisieren und löschen (vgl. ebd.: Abschn. Default Roles). Falls diese zwei Rollen nicht ausreichen, gibt es die Möglichkeit, eigene Rollen anzulegen (vgl. ebd.: Abschn. Custom Roles). Dabei kann man sehr präzise die Zugriffsrechte zu allen Bereichen des CMS einstellen (vgl. ebd.: Abschn. Permission Types). In Abbildung 22 sind all die Einstellmöglichkeiten für eine neue Rolle im Überblick zu sehen.

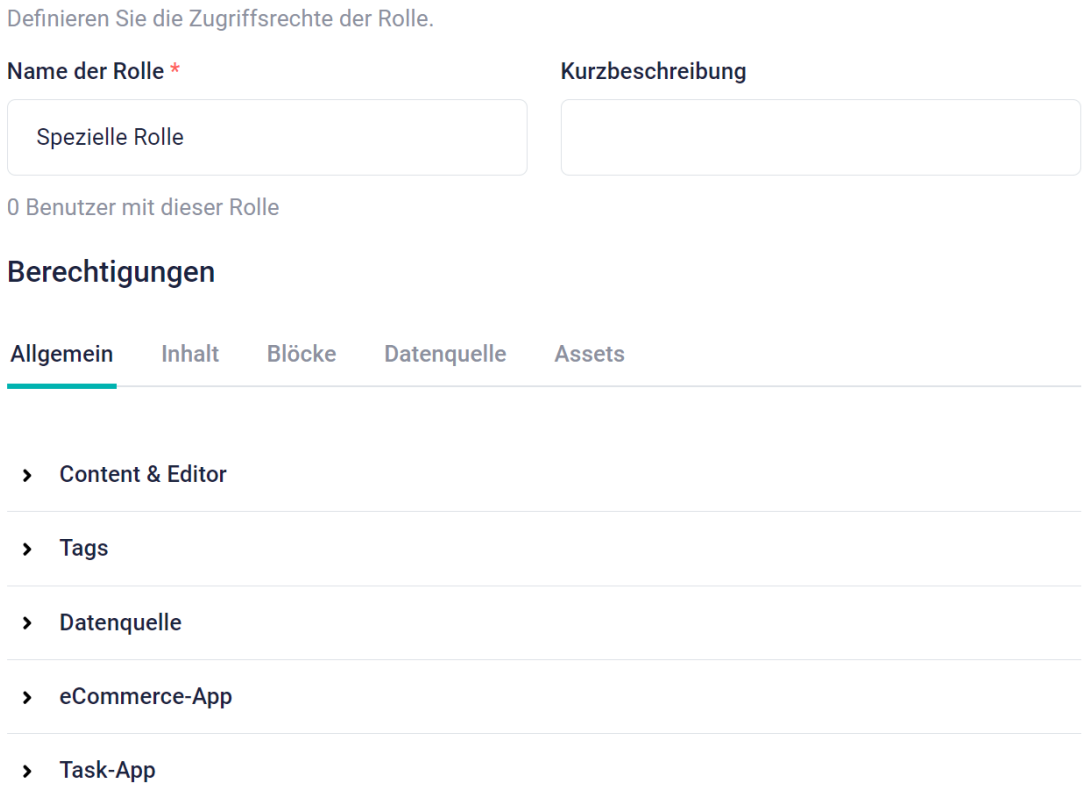

Abbildung 22: Storyblok – Zugriffsrechte einer neuen Rolle (eigene Aufzeichnung)

6. Fazit

Ziel dieser Arbeit ist es, zu überprüfen, ob der Jamstack hinsichtlich des Funktionsumfangs im Bereich der redaktionellen Pflege von Websites gegenüber der bereits etablierten monolithischen Architektur eine berechtigte Alternative darstellt. Um den Jamstack zu überprüfen, wurde eine Reihe an praktischen Lösungsszenarien zur redaktionellen Pflege einer realen Website als praktische Ausarbeitung erstellt.

Die erarbeiteten Lösungsszenarien lassen den Schluss zu, dass die redaktionelle Pflege von Websites im Jamstack im großen Umfang uneingeschränkt möglich ist. Jedoch kann ein Lösungsweg für die Umsetzung einer redaktionellen Aufgabe je nach Wahl der technologischen Infrastruktur unterschiedlich ausfallen, da der Jamstack eine offene Architektur ist, die es erlaubt, beliebige Technologien zu verwenden. Daraus ergibt sich, dass es nicht den einen „perfekten“ Lösungsweg gibt, um ein Lösungsszenario zu entwickeln.

Betrachtet man die genutzten Kerntechnologien Nuxt, Storyblok und Netlify, aus der die praktische Ausarbeitung besteht, lässt sich sagen, dass sie zum Einsatz zur redaktionellen Pflege im Jamstack optimal geeignet sind. Nuxt hat sich als besonders flexibles Framework erwiesen, hinsichtlich der Möglichkeiten, die auf viele Arten lösbare Konzepte umzusetzen. Storyblok hat sich als einfach zu integrierendes CMS erprobt, das vom Aufbau her der komponentenbasierten Natur von Nuxt gleicht, was eine sehr angenehme Entwicklererfahrung mit sich brachte. Als CI/CD-Infrastruktur war Netlify ein leicht bedienbarer und zuverlässiger Dienst zur Überprüfung der Lösungsszenarien in einer Produktivumgebung.

Die praktische Ausarbeitung und die Lösungsszenarien stellen eine Basis zur Entwicklung von redaktionell pflegbaren Websites im Jamstack dar und können als Grundlage verwendet werden, um weitere Lösungsszenarien zu entwickeln oder um die gezeigten Konzepte in anderen Frameworks und CMS zu übertragen. Besonders die entwickelten Module und die Projektstruktur von Nuxt kann in weiteren Nuxt-Projekten wiederverwendet werden.

Durch die Erstellung der Lösungsszenarien und der Entwicklung der praktischen Ausarbeitung wurde gezeigt, dass sich die redaktionelle Pflege von Websites im Jamstack erfolgreich umsetzen lässt. Die redaktionellen Aufgaben aus der traditionellen Architektur können auch in einer entkoppelten Architektur bestehen. Der Jamstack ist als anpassungsfähigere Alternative zu sehen, die zusätzlich in Punkten Sicherheit, Skalierbarkeit und Geschwindigkeit Vorteile gegenüber der traditionellen Architektur bietet.

Folgende Problembereiche haben sich bei der praktischen Ausarbeitung herausgestellt: Es zeigte sich, dass der Jamstack schnell auf seine Grenzen stößt, wenn sich die Inhalte einer Website in einer sehr hohen Frequenz verändern oder kritische Informationen umgehend aktualisiert werden müssen. Das liegt daran, dass der neue Content immer durch einen SSG erst gebaut und im Anschluss veröffentlicht werden muss. Dieser Prozess benötigt Rechenzeit und kann nicht unverzüglich stattfinden. Ein weiteres Problem ist, dass es bei sehr großen Websites mit tausendenden von Seiten zu verlängerten Bauzeiten seitens des SSG kommt. Die Skalierbarkeit solcher Websites kann zur Herausforderung für Entwickler werden. Um diese Einschränkung zu umgehen, kommen neue Technologien wie beispielsweise Incremental Static Regeneration (ISG) zum Einsatz, welches einen enormen Geschwindigkeitszuwachs verspricht, indem nur der neue Inhalt anstatt das gesamte Projekt von Grund aus neu generiert werden muss. Abschließend ist zu sagen, dass der Jamstack eine sehr flexible Architektur hinsichtlich der Erweiterbarkeit ist und es auch zukünftig immer neue Möglichkeiten und Verbesserungen geben wird, die die bestehenden Probleme lösen werden.

Literaturverzeichnis

Bücher

Ackermann, Philip (2018): Professionell entwickeln mit JavaScript: Design, Patterns und Praxistipps für Enterprise-fähigen Code, 2. Aufl., Bonn, Deutschland: Rheinwerk Computing.

Biilmann, Mathias/Phil Hawksworth (2019): Modern Web Development on the JAMstack, 1. Aufl., O'Reilly.

Camden, Raymond/Brian Rinaldi (2022): The Jamstack Book: Beyond static sites with JavaScript, APIs, and markup, 1. Aufl., Manning.

Horster, Eric/Constantin Foltin/Kristine Honig/Elias Kärle/Florian Bauhuber (2022): Digitales Tourismusmarketing: Grundlagen, Suchmaschinenmarketing, User-Experience-Design, Social-Media-Marketing und Mobile Marketing, Wiesbaden, Deutschland: Springer Gabler.

Spörrer, Stefan (2019): Content Management Systeme: Begriffsstruktur und Praxisbeispiel (Edition KWV), 1. Aufl. 2009, Nachdruck 2019, Springer Gabler.

Online-Quellen

Angerer, Dominik (2022): Headless CMS explained in 5 effective minutes, Storyblok, [online] https://www.storyblok.com/tp/headless-cms-explained [abgerufen am 11.06.2022].

Backlinko (2020): What is a Sitemap? How to Create an SEO Optimized Sitemap, Backlinko, [online] https://backlinko.com/hub/seo/sitemaps [abgerufen am 13.08.2022].

Brockbank, James (2021): Weiterleitungen und URL-Redirects: Die große und einfache Erklärung, Semrush Blog, [online] https://de.semrush.com/blog/weiterleitungen-url-redirects/ [abgerufen am 13.08.2022].

Cardoza, Christina (2020): Jamstack brings front-end development back into focus, SD Times, [online] https://sdtimes.com/webdev/jamstack-brings-front-end-development-back-into-focus/ [abgerufen am 15.05.2022].

Cloudflare (o. D.): Was ist ein CDN? | Wie funktionieren CDNs?, Cloudflare.com, [online] https://www.cloudflare.com/de-de/learning/cdn/what-is-a-cdn/ [abgerufen am 17.05.2022].

Contentstack (o. D.): Monolithic vs. Microservices: Why Decoupled and Headless Architectures Are the Future | Contentstack, Contentstack, [online] https://www.contentstack.com/cms-guides/monolithic-vs-microservices-cms-architectures/ [abgerufen am 06.08.2022].

dataliquid GmbH (2018): HTTP Basic Auth, dataliquid, [online] https://www.dataliquid.com/artikel.html?tx_knowledgebase_piknowledgebase%5Baction%5D=detail&tx_knowledgebase_piknowledgebase%5Barticle%5D=1&tx_knowledgebase_piknowledgebase%5Bcontroller%5D=Article&cHash=45c1ab39656f4246ead44b0f49a7328f [abgerufen am 11.08.2022].

Denysov, Artem (2022): Jamstack | 2021 | The Web Almanac by HTTP Archive, HTTP Archive, [online] https://almanac.httparchive.org/en/2021/jamstack [abgerufen am 09.06.2022].

Dillon, Charlotte (2021): Headless CMS and the Jamstack, Explained, bloomreach, [online] https://developers.bloomreach.com/blog/2021/headless-cms-and-the-jamstack-explained.html [abgerufen am 10.06.2022].

Enyinnaya, Chimezie (2019): JAMstack: The What, the Why and the How, DigitalOcean Community, [online] https://www.digitalocean.com/community/tutorials/jamstack-the-what-the-why-and-the-how [abgerufen am 14.05.2022].

Exposure Ninja (Shinobi) (2021): What Is a Sitemap? (and How To Upload One to Search Console), Exposure Ninja, [online] https://exposureninja.com/training/guides/seo/what-is-a-sitemap/ [abgerufen am 13.08.2022].

Hawksworth, Phil (2020): What is a Static Site Generator? How do I find the best one to use?, Netlify, [online] https://www.netlify.com/blog/2020/04/14/what-is-a-static-site-generator-and-3-ways-to-find-the-best-one/ [abgerufen am 06.06.2022].

iThemes (2020): What is WordPress? | WordPress 101 Tutorials, iThemes, [online] https://ithemes.com/tutorials/what-is-wordpress/ [abgerufen am 05.08.2022].

Karwatka, Piotr/Tamara Bolsewicz/Tamara Bolsewicz/Tim Green (o. D.): Monolithic architecture vs microservices: Which is better? | Divante, divante, [online] https://www.divante.com/blog/monolithic-architecture-vs-microservices [abgerufen am 06.08.2022].

King, Ryland (2022): Headless CMS: Definition, Pros/Cons, Use Cases & Tools, Stackbit, [online] https://www.stackbit.com/blog/what-is-a-headless-cms/ [abgerufen am 14.05.2022].

Kinsta (2022): Was ist Nuxt.js? Erfahre mehr über das intuitive Vue-Framework, Kinsta, [online] https://kinsta.com/de/wissensdatenbank/nuxt-js/ [abgerufen am 06.08.2022].

Netlify (Build Hooks) (o. D.): Build hooks, Netlify Docs, [online] https://docs.netlify.com/configure-builds/build-hooks/ [abgerufen am 06.08.2022].

Netlify (Jamstack.org: Pre-render) (o. D.): Pre-render / Pre-generate, Jamstack.org, [online] https://jamstack.org/glossary/pre-render/ [abgerufen am 06.08.2022].

Netlify (Redirects and rewrites) (o. D.): Redirects and rewrites, Netlify Docs, [online] https://docs.netlify.com/routing/redirects/ [abgerufen am 08.08.2022].

Nguyen, Katherine (2018): How to Choose the Right Content Management System (CMS) | Contentstack, contentstack.com, [online] https://www.contentstack.com/blog/all-about-headless/10-tips-on-content-management-systems-cms/ [abgerufen am 15.05.2022].

Nuxt (API: Lifecycle Hooks) (o. D.): Lifecycle Hooks, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/api/advanced/hooks/ [abgerufen am 06.08.2022].

Nuxt (API: useRoute) (o. D.): useRoute, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/api/composables/use-route/ [abgerufen am 06.08.2022].

Nuxt (Guide: Head Management) (o. D.): Head Management, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/guide/features/head-management/ [abgerufen am 06.08.2022].

Nuxt (Guide: Lifecycle Hooks) (o. D.): Lifecycle Hooks, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/guide/going-further/hooks/ [abgerufen am 06.08.2022].

Nuxt (Guide: Module Author Guide) (o. D.): Module Author Guide, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/guide/going-further/modules/ [abgerufen am 06.08.2022].

Nuxt (Guide: Pages directory) (o. D.): Pages directory, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/guide/directory-structure/pages/ [abgerufen am 06.08.2022].

Nuxt (Guide: Rendering Modes) (o. D.): Rendering Modes, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/guide/concepts/rendering [abgerufen am 14.06.2022].

Nuxt (Guide: Server Routes) (o. D.): Server Routes, Nuxt 3 Dokumentation, [online] https://v3.nuxtjs.org/guide/features/server-routes/ [abgerufen am 06.08.2022].

Prasad, Madhusha (2022): Vuejs history, Medium, [online] https://medium.com/sliit-foss/vuejs-history-865eb1bba386 [abgerufen am 24.07.2022].

Schiel, Fabian (2020): Headless CMS: eine Einführung, Liechtenecker, [online] https://liechtenecker.at/blog/headless-cms/ [abgerufen am 10.06.2022].

Schürmanns, Sebastian (2021): CMS - Wie funktioniert ein Content Management System?, CMSstash, [online] https://cmsstash.de/content-management-system/wie-ein-cms-funktioniert [abgerufen am 31.07.2022].

Singh, Vrijraj (2021): Vue to Web: Introduction Part-1 - CodinGurukul, Medium, [online] https://medium.com/codingurukul/vue-to-web-introduction-part-1-42d1aee95822 [abgerufen am 06.08.2022].

Smashing Conference (o. D.): Smashing Conference San Francisco 2016, Smashing Conference 2016, [online] https://archive.smashingconf.com/sf-2016/ [abgerufen am 15.05.2022].

Statista (2022): Nutzungsanteil der Content-Management-Systeme (CMS) weltweit im August 2022, Statista, [online] https://de.statista.com/statistik/daten/studie/320685/umfrage/nutzungsanteil-der-content-management-systeme-cms-weltweit/ [abgerufen am 05.08.2022].

Storyblok (Content Structures) (o. D.): Structures of Content, Storyblok, [online] https://www.storyblok.com/docs/guide/essentials/content-structures [abgerufen am 11.06.2022].

Storyblok (Image Service) (o. D.): Image Service (Legacy), Storyblok, [online] https://www.storyblok.com/docs/image-service [abgerufen am 14.08.2022].

Storyblok (Roles & Permissions) (o. D.): Roles & Permissions, Storyblok, [online] https://www.storyblok.com/docs/guide/in-depth/roles-and-permissions [abgerufen am 14.08.2022].

Storyblok (Visual Editor) (o. D.): Understanding the Visual Editor, Storyblok, [online] https://www.storyblok.com/docs/guide/essentials/visual-editor [abgerufen am 26.07.2022].

Sycamore, Sam (2021): A Clueless Newbie's Guide to Headless CMS & the Jamstack, Sycamore Design Blog, [online] https://blog.sycamore.design/headless-cms-jamstack [abgerufen am 15.05.2022].

Vue.js (o. D.): Vue.js, Vue.js, [online] https://vuejs.org/guide [abgerufen am 24.07.2022].

Zeta Producer (2018): Was ist eine Navigation? - einfach erklärt, Professionelle Websites erstellen, [online] https://blog.zeta-producer.com/navigation/ [abgerufen am 07.08.2022].

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Thesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel, sind angegeben.

Denzlingen, den 31.08.2022