Einsatz von Meta-Frameworks zur Integration heterogener IT-Systeme: Fallstudie eines Self-Service Portals

Masterarbeit von
Jonathan Biren
1201678

Universität der Bundeswehr München
Fakultät für Informatik

der Bundeswehr
Universität München

Einsatz von Meta-Frameworks zur Integration heterogener IT-Systeme: Fallstudie eines Self-Service Portals

Masterarbeit von
Jonathan Biren
1201678

Aufgabensteller:	Univ.-Prof. Dr. Michael Koch
Zweitprüfer:	Prof. Dr.-Ing Mark Minas
Betreuer:	Laura Stojko M.Sc. Dr. Julian Fietkau
Abgabetermin:	01.7.2024

Universität der Bundeswehr München
Fakultät für Informatik

Kurzfassung

Diese Masterarbeit untersucht die Anwendbarkeit und Effektivität moderner Meta-Frameworks in der Webentwicklung am Beispiel der Implementierung eines Self-Service-Portals für die Informatik-Fakultät der Bundeswehr Universität München. Im Fokus der Untersuchung stehen das Component-Framework Svelte und das darauf aufbauende Meta-Framework SvelteKit. Die Arbeit beginnt mit einer Einführung in die theoretischen Grundlagen von Content Management Systemen, LDAP-Authentifizierung und Web-Frameworks. Anschließend wird die Systemarchitektur des Projekts vorgestellt, gefolgt von einer detaillierten Betrachtung der Funktionsweise und Besonderheiten von Svelte und SvelteKit. Der praktische Teil der Arbeit umfasst die Anforderungsanalyse, das Architekturdesign und die Implementierung der Kernfunktionalitäten des Self-Service-Portals. Besonderes Augenmerk liegt dabei auf der Integration mit einem Headless CMS, der Umsetzung verschiedener Authentifizierungsmethoden und der Implementierung der Profilbearbeitung. In der abschließenden Evaluation werden die Ergebnisse kritisch beleuchtet. Dabei werden sowohl die Stärken als auch die Herausforderungen bei der Verwendung von Svelte und SvelteKit analysiert. Die Arbeit kommt zu dem Schluss, dass Meta-Frameworks wie SvelteKit ein erhebliches Potenzial für die effiziente Entwicklung moderner Webanwendungen bieten, insbesondere in Szenarien, die eine enge Integration von Front- und Backend erfordern. Die Erkenntnisse dieser Arbeit liefern wertvolle Einblicke in die praktische Anwendung von Meta-Frameworks und können als Grundlage für zukünftige Entscheidungen bei der Auswahl von Technologien für Webentwicklungsprojekte im akademischen und professionellen Umfeld dienen.

Abstract

This master's thesis examines the applicability and efficacy of modern meta-frameworks in web development through the implementation of a self-service portal for the Faculty of Computer Science at the Bundeswehr University Munich. The study focuses on the component framework Svelte and its associated meta-framework SvelteKit. The thesis commences with an introduction to the theoretical foundations of content management systems, LDAP authentication, and web frameworks. Subsequently, it presents the system architecture of the project, followed by a detailed analysis of the functionality and distinctive features of Svelte and SvelteKit. The practical component of the study encompasses requirements ana-

lysis, architecture design, and the implementation of core functionalities for the self-service portal. Particular emphasis is placed on the integration with a headless CMS, the realization of diverse authentication methodologies, and the implementation of profile editing capabilities. The concluding evaluation critically assesses the outcomes, analyzing both the strengths and challenges encountered in utilizing Svelte and SvelteKit. The thesis concludes that meta-frameworks such as SvelteKit offer substantial potential for the efficient development of modern web applications, particularly in scenarios necessitating close integration of front-end and back-end components. The insights derived from this study provide valuable perspectives on the practical application of meta-frameworks and can serve as a foundation for future decision-making processes in selecting technologies for web development projects in both academic and professional contexts.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Zielsetzung . . . 2
- 1.3 Vorgehen und Aufbau . . . 3

2 Theoretische Grundlagen **5**
- 2.1 Content-Management-Systeme . . . 6
  - 2.1.1 Headless CMS . . . 8
- 2.2 LDAP . . . 10
- 2.3 Web-Frameworks . . . 11
  - 2.3.1 Component-Frameworks . . . 12
  - 2.3.2 Meta-Frameworks . . . 13
- 2.4 REST API . . . 15

3 Gesamtkontext und Systemarchitektur **18**
- 3.1 Architekturübersicht . . . 18
- 3.2 Meta-Frameworks als Lösung . . . 23

4 Svelte und SvelteKit **26**
- 4.1 Svelte . . . 26
  - 4.1.1 Svelte Syntax . . . 29
  - 4.1.2 Svelte Compiler . . . 31
  - 4.1.3 Svelte Reaktivität . . . 33
- 4.2 SvelteKit . . . 35
  - 4.2.1 Routing . . . 35
  - 4.2.2 Laden von Daten . . . 37
  - 4.2.3 Rendering . . . 40

5 Entwicklung des Self-Service-Portals **43**
- 5.1 Anforderungsanalyse . . . 43
  - 5.1.1 Stakeholder-Identifikation . . . 44
  - 5.1.2 Funktionale Anforderungen . . . 45
- 5.2 Architekturdesign . . . 48
- 5.3 Implementierung der Kernfunktionalitäten . . . 54
  - 5.3.1 Implementierung der LDAP-Authentifizierung . . . 54
  - 5.3.2 Implementierung der Custom-Authentifizierung . . . 57
  - 5.3.3 Implementierung der Profilbearbeitung . . . 58

6 Evaluation **63**
- 6.1 Erfüllung der Projektanforderungen . . . 63

6.2 Integration mit externen Diensten 63
6.3 Entwicklungseffizienz und Wartbarkeit 64
6.4 Herausforderungen und Lösungsansätze 65
6.5 Kritische Abwägungen 66

7 Fazit **68**
7.1 Zusammenfassung 68
7.2 Ausblick 69

Abkürzungsverzeichnis **70**

Abbildungsverzeichnis **71**

Literaturverzeichnis **72**

1 Einleitung

In diesem Kapitel soll der Leser einen Eindruck dafür gewinnen, was die Motivation für die Anfertigung dieser Arbeit war. Des Weiteren, werden die zentralen Forschungsfragen der Arbeit herausgestellt, als auch die Vorgehensweise und die gesamtheitliche Struktur der Arbeit dargelegt, um dem Leser ein Fundament für die Kontextualisierung der Arbeit bereitzustellen.

1.1 Motivation

In der heutigen schnelllebigen und digitalisierten Welt gewinnen effiziente und benutzerfreundliche IT-Lösungen, insbesondere Self-Service Portale, im universitären Umfeld zunehmend an Bedeutung [Sof24] . Diese webbasierten Plattformen ermöglichen es Universitätsangehörigen, eigenständig auf Informationen und Dienste zuzugreifen, was die Produktivität steigert und IT-Abteilungen entlastet [ARH+22]. Die vorliegende Arbeit konzentriert sich auf ein konkretes Anwendungsszenario an der Bundeswehr Universität München: die Entwicklung eines Self-Service Portals zur eigenständigen Bearbeitung von Community-Mirror [1] Profilen.

Bislang erforderte die Aktualisierung dieser Profile einen zeitaufwändigen Prozess, bei dem Nutzer Änderungswünsche per E-Mail an das MCI-Team (Mensch Computer Interaktion) übermitteln mussten, der diese dann manuell im Content-Management-System umsetzte. Diese Arbeit zielt darauf ab, diesen Prozess durch ein effizientes, benutzerfreundliches Self-Service Portal zu digitalisieren und zu automatisieren.

Die Entwicklung solcher Portale stellt jedoch häufig eine Herausforderung dar, vorwiegend hinsichtlich der schnellen, kosteneffizienten Umsetzung und nahtlosen Integration in bestehende IT-Infrastrukturen [DD15] . Oft erfordern diese Projekte umfangreiche Ressourcen und spezialisiertes Fachwissen.

Die Hauptmotivation dieser Arbeit liegt in der Untersuchung und Umsetzung eines Self-Service Portals zur Digitalisierung des Community-Mirror Szenarios im universitären Umfeld. Das genaue Szenario wird in Kapitel 3 näher beschrieben. Dabei liegt der Fokus auf der Identifikation und Evaluierung geeigneter Werkzeuge, Technologien und Architekturansätze, um eine effiziente, praktikable und zukunftsorientierte Lösung zu entwickeln.

Durch die Analyse moderner Webtechnologien, Frameworks und Architekturparadigmen zielt diese Arbeit darauf ab, einen optimalen Weg aufzuzeigen, um Self-Service Portale im akademischen Kontext zu realisieren. Hierbei steht insbesondere die spezifische Anwendung auf das beschriebene Community-Mirror Szenario im Vordergrund, um eine maßgeschneiderte

[1] `https://publicwiki.unibw.de/pages/viewpage.action?pageId=3546059`

und praxistaugliche Lösung zu erarbeiten, die den Anforderungen und Herausforderungen dieses speziellen Anwendungsfalls gerecht wird.

Unsere Motivation ist somit, anhand eines konkret im universitären Kontext bestehenden Problems festzustellen, ob sich moderne Webtechnologien (konkret Meta-Frameworks 2.3.2), für die einfache und effiziente Implementierung einer Lösung eignen. Diese Untersuchung soll nicht nur zur Lösung des spezifischen Problems beitragen, sondern auch wertvolle Erkenntnisse für die Entwicklung ähnlicher Systeme in verschiedenen akademischen und organisatorischen Kontexten liefern.

1.2 Zielsetzung

Aufbauend auf der im vorherigen Abschnitt dargelegten Motivation widmet sich diese Arbeit der zentralen Forschungsfrage: Inwieweit sind moderne Webtechnologien, vorwiegend Meta-Frameworks wie SvelteKit[2] , geeignet, um hochgradig integrierte Applikationen wie Self-Service Portale effizient zu implementieren?

Das Hauptziel dieser Untersuchung ist es, die Eignung von SvelteKit als Meta-Framework für die Entwicklung von Self-Service Portalen im akademischen Umfeld zu evaluieren [Par24]. Dabei sollen folgende Aspekte betrachtet werden:

- Analyse der spezifischen Funktionen und Abstraktionen von SvelteKit, die die Entwicklung von Self-Service Portalen erleichtern und beschleunigen können.
- Untersuchung technischer Aspekte wie Komponentenarchitektur, Datenmanagement und Routing.
- Bewertung der Möglichkeiten zur Integration mit bestehenden Systemen und Diensten.
- Vergleich der Vor- und Nachteile von SvelteKit gegenüber anderen Ansätzen und Technologien im Kontext der Entwicklung integrierter Webapplikationen.

Anhand des in Abschnitt 1.1 beschriebenen praktischen Anwendungsfalls - der Entwicklung eines Self-Service Portals für die Community-Mirror Profile - soll die Effizienz und Praktikabilität von SvelteKit empirisch untersucht werden. Hierbei steht die Frage im Vordergrund, wie diese Technologie dazu beitragen kann, die Entwicklung zu vereinfachen, die Integration mit bestehenden Systemen zu gewährleisten und gleichzeitig eine hohe Benutzerfreundlichkeit und Performanz zu erzielen.

Ein weiteres Ziel dieser Arbeit ist es, über den spezifischen Anwendungsfall hinaus generalisierbare Erkenntnisse zu gewinnen. Durch die Identifikation wiederverwendbarer Muster, Komponenten und Architekturprinzipien soll evaluiert werden, inwiefern die entwickelte Lösung auf andere Kontexte und Anwendungsszenarien übertragbar ist. Dies soll dazu beitragen, die Entwicklung integrierter, webbasierter Applikationen in verschiedenen Bereichen des akademischen und organisatorischen Umfelds zu erleichtern [Aut22].

Zusammenfassend zielt diese Arbeit darauf ab:

[2] `https://kit.svelte.dev`

- Das Potenzial von SvelteKit für die effiziente Implementierung von Self-Service Portalen und anderen hochgradig integrierten Applikationen aufzuzeigen.
- Einen praxisorientierten Lösungsansatz zu entwickeln und daraus übertragbare Erkenntnisse abzuleiten.
- Ein Fundament zu schaffen, das zur Vereinfachung und Optimierung von Entwicklungsprozessen in diesem Bereich beiträgt.
- Einen Beitrag zur Gestaltung zukunftsfähiger IT-Landschaften in akademischen Institutionen zu leisten.

Durch die Verfolgung dieser Ziele strebt die Arbeit an, nicht nur eine spezifische Lösung für das vorliegende Problem zu entwickeln, sondern auch wertvolle Einsichten für die breitere Anwendung moderner Webtechnologien in komplexen, integrierten IT-Umgebungen zu liefern.

1.3 Vorgehen und Aufbau

Die vorliegende Arbeit gliedert sich in sieben Kapitel, die systematisch den Forschungsprozess und die Entwicklung der Self-Service-Webapplikation dokumentieren. Dieses Kapitel dient als Einleitung und legt den Grundstein für die gesamte Arbeit. Zunächst wurde in Abschnitt 1.1 die Motivation hinter der Forschung dargelegt, wobei die Bedeutung von Self-Service-Portalen im Kontext von Universitäten und Forschungseinrichtungen sowie die spezifischen Herausforderungen und Bedürfnisse der Informatik-Fakultät erörtert werden. Anschließend werden in Abschnitt 1.2 die zentrale Forschungsfrage und die angestrebten Ziele der Arbeit präzisiert.

Kapitel 2 widmet sich den theoretischen Grundlagen und dem Stand der Forschung. In Abschnitt 2.1 erfolgt eine Einführung in Content Management Systeme (CMS) mit besonderem Fokus auf Headless CMS. Abschnitt 2.2 behandelt die Grundlagen und Bedeutung des Lightweight Directory Access Protocol (LDAP) im Kontext der Authentifizierung für das Portal. Abschnitt 2.3 gibt einen Überblick über Web-Frameworks und deren Rolle in der modernen Webentwicklung.

Kapitel 3 beleuchtet den Gesamtkontext und die Systemarchitektur der zu entwickelnden Webapplikation. Abschnitt 3.1 präsentiert eine detaillierte Übersicht über die beteiligten Komponenten wie CMS, LDAP-Server und deren Interaktion. Eine visuelle Darstellung der Systemarchitektur unterstützt das Verständnis. In Abschnitt 3.2 wird eine Unterscheidung zwischen Component-Frameworks und Meta-Frameworks vorgenommen und verschiedene Meta-Frameworks sowie deren Einsatzmöglichkeiten werden vorgestellt. Kapitel 4 befasst sich eingehend mit Svelte und SvelteKit, den ausgewählten Technologien für die Projektentwicklung. Abschnitt 4.1 bietet eine Einführung in Svelte, gefolgt von einer Erläuterung der Grundlagen und Kernkonzepte von SvelteKit in Abschnitt 4.2. Kapitel5 beschreibt den Prozess der Entwicklung der Self-Service-Webapplikation. In Abschnitt 5.1 werden die funktionalen und nicht-funktionalen Anforderungen durch eine sorgfältige Anforderungsanalyse identifiziert und definiert. Abschnitt 5.2 liefert eine detaillierte Beschreibung des Architekturdesigns, einschließlich Implementierungsdetails und verwendeten Technologien. Abschnitt 5.3

erläutert die Entwicklung ausgewählter Kernfunktionalitäten, beispielweise der Integration des LDAP-Servers zur Authentifizierung der Nutzer.

Kapitel 6 widmet sich der Evaluation und den Ergebnissen der entwickelten Webapplikation. Kapitel 7 schließt die Arbeit mit einem Fazit und einem Ausblick ab. In Abschnitt 7.1 werden die Hauptergebnisse der Arbeit zusammengefasst. Abschnitt 7.2 schließt die Arbeit ab und diskutiert die Limitationen der Arbeit und zeigt mögliche zukünftige Forschungsrichtungen auf. Durch diese strukturierte Vorgehensweise wird eine umfassende und systematische Untersuchung der Forschungsfrage gewährleistet.

Der Aufbau der Arbeit ermöglicht es, die einzelnen Aspekte der Entwicklung einer Self-Service-Webapplikation unter Verwendung von Meta-Frameworks wie SvelteKit schrittweise zu beleuchten und die gewonnenen Erkenntnisse in einen größeren Kontext einzuordnen. Im nachfolgenden Kapitel, wollen wir uns nun den theoretischen Grundlagen widmen, die es dem Leser erleichtern sollen, die Arbeit nachzuvollziehen und einzuordnen.

2 Theoretische Grundlagen

Bevor wir uns der konkreten Implementierung des Self-Service Portals zuwenden, ist es essenziell, die theoretischen Grundlagen zu betrachten, auf denen unsere Lösung aufbaut. Dieses Kapitel führt den Leser in vier zentrale Konzepte ein, die für das Verständnis und die Entwicklung des Portals von entscheidender Bedeutung sind: Content-Management-Systeme, LDAP (Lightweight Directory Access Protocol), Web-Frameworks und REST APIs.

Die Auseinandersetzung mit Content Management Systemen in Abschnitt 2.1 ist fundamental, da unser Self-Service Portal mit einem bestehenden CMS interagieren muss. Ein tieferes Verständnis dieser Systeme ermöglicht es uns, die Integration effizient zu gestalten und potenzielle Herausforderungen frühzeitig zu erkennen.

LDAP, das in Abschnitt 2.2 behandelt wird, spielt eine Schlüsselrolle in der Authentifizierung und Autorisierung von Benutzern. Da wir die bestehende LDAP-Infrastruktur der Universität nutzen werden, ist es wichtig, die Funktionsweise und Besonderheiten dieses Protokolls zu verstehen, um eine nahtlose und sichere Benutzerauthentifizierung zu gewährleisten.

In Abschnitt 2.3 widmen wir uns den Web-Frameworks, insbesondere den Meta-Frameworks wie SvelteKit. Diese modernen Technologien bilden das Rückgrat unserer Implementierung und ermöglichen eine effiziente und skalierbare Entwicklung des Self-Service Portals. Ein fundiertes Verständnis ihrer Konzepte und Funktionsweisen ist unerlässlich, um ihre Vorteile optimal nutzen zu können.

Schließlich betrachten wir in Abschnitt 2.4 die Grundlagen von REST APIs. Diese spielen eine entscheidende Rolle bei der Kommunikation zwischen unserem Portal und dem CMS sowie anderen Diensten. Das Verständnis von REST-Prinzipien entscheidend für die Gestaltung einer robusten und effizienten Datenübertragung innerhalb unserer Architektur.

Es ist wichtig zu betonen, dass die hier vorgestellte Lösung in ihrem Wesen integrativ ist. Sie bringt die verschiedenen Elemente - CMS, LDAP, Web-Frameworks und REST APIs - in einer kohärenten Architektur zusammen. Diese Integration ermöglicht es uns, die Stärken jeder Komponente zu nutzen und gleichzeitig eine nahtlose Benutzererfahrung zu schaffen. Unser Self-Service Portal fungiert dabei als zentraler Knotenpunkt, der diese unterschiedlichen Technologien orchestriert und zu einem funktionalen Ganzen verbindet. Die integrative Natur der in dieser Arbeit vorgestellten Lösung wird in Abbildung 2.1 nochmals verdeutlicht.

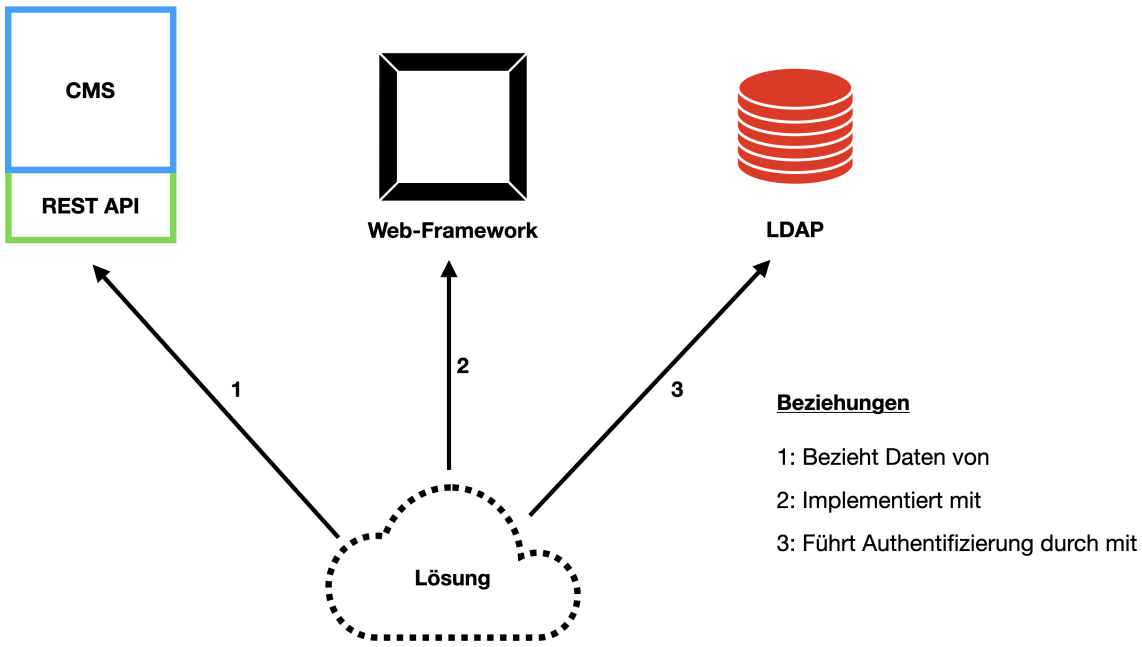

Abbildung 2.1: Illustration der Integration der Elemente LDAP, REST API, Web-Framework und CMS

Durch die Betrachtung dieser vier Themenbereiche schaffen wir eine solide Wissensbasis, die es uns ermöglicht, informierte Entscheidungen während des Entwicklungsprozesses zu treffen und die verschiedenen Komponenten unseres Systems effektiv zu integrieren. Diese Grundlagen bilden das Fundament für die nachfolgenden Kapitel, in denen wir uns der praktischen Umsetzung des Self-Service Portals widmen werden. Das Verständnis dieser Konzepte und ihrer Zusammenhänge ist entscheidend, um die Komplexität des Gesamtsystems zu beherrschen und eine Lösung zu entwickeln.

2.1 Content-Management-Systeme

Zunächst wollen wir uns einmal vergegenwärtigen, was ein Content-Management-System ist und wozu es verwendet wird. Ein Content-Management-System (CMS) ist eine Softwareanwendung, die es Nutzern ermöglicht, digitale Inhalte zu erstellen, zu verwalten und zu modifizieren, ohne tiefgehende technische Kenntnisse zu benötigen [Opt24]. Ein CMS trennt den Inhalt von der Darstellung und bietet Benutzern eine intuitive Benutzeroberfläche zur Verwaltung von Webseiteninhalten. Ein populäres Beispiel hierfür ist das auch für das Community-Mirrors Projekt[1] verwendete WordPress [2].

[1] https://publicwiki.unibw.de/display/MCI/CommunityMirror
[2] https://wordpress.org

Vorteile der Verwendung eines CMS

Die Nutzung eines CMS bietet zahlreiche Vorteile, die für die Verwaltung von Webseiteninhalten sowohl im akademischen als auch im unternehmerischen Kontext wichtig sind [Opt24]. Diese Vorteile lassen sich in verschiedene Kategorien einteilen:

1. Effizienz und Benutzerfreundlichkeit:
    - CMS ermöglicht es auch nicht-technischen Benutzern, Inhalte einfach und effizient zu aktualisieren. Dies reduziert die Abhängigkeit von technischen Fachkräften und verringert die damit verbundenen Kosten und Zeitaufwände.
    - Durch die Verwendung von Vorlagen und WYSIWYG-Editoren (What You See Is What You Get) können Inhalte schnell und ohne technische Fehler aktualisiert werden.
2. Kollaboration und Workflow-Management:
    - Ein CMS unterstützt kollaboratives Arbeiten, indem es mehreren Benutzern ermöglicht, gleichzeitig an Inhalten zu arbeiten. Zudem bieten viele CMS integrierte Workflows, die sicherstellen, dass Inhalte vor der Veröffentlichung von mehreren Parteien geprüft und genehmigt werden können.
3. Flexibilität und Skalierbarkeit:
    - CMS-Lösungen sind hoch skalierbar und können von kleinen Blogs bis hin zu großen Unternehmenswebseiten eingesetzt werden. Sie bieten eine Vielzahl von Plugins und Erweiterungen, die zusätzliche Funktionen die beispielweise SEO-Optimierung und Social Media Integration ermöglichen.
4. Sicherheit und Updates:
    - Ein CMS bietet regelmäßig Sicherheitsupdates und Patches, die helfen, die Webseite vor potenziellen Bedrohungen zu schützen. Diese Updates können automatisch oder manuell installiert werden, um sicherzustellen, dass die Webseite stets auf dem neuesten Stand ist.
5. Kosteneffizienz:
    - Durch die Reduktion der Notwendigkeit, einen dedizierten Webentwickler für jede Inhaltsänderung einzusetzen, können Unternehmen erhebliche Kosten einsparen. Ein CMS ermöglicht es Unternehmen, ihre Inhalte intern zu verwalten, wodurch langfristige Kosteneinsparungen erzielt werden können.

Unterschied zwischen CMS und traditionellen Datenbanken

Während sowohl CMS als auch Datenbanken zur Verwaltung von Daten verwendet werden, gibt es wesentliche Unterschiede in ihrer Funktionalität und Anwendung:

1. Datenverwaltung vs. Inhaltsverwaltung:
    - Eine Datenbank ist ein System zur Speicherung und Organisation von Daten, das in der Regel von technischen Fachkräften verwaltet wird. Sie ist ideal für

strukturierte Daten, wie Kundeninformationen oder Transaktionsdaten.

- Ein CMS hingegen konzentriert sich auf die Verwaltung von Inhalten, die auf einer Webseite präsentiert werden. Es bietet Werkzeuge zur Erstellung, Bearbeitung und Veröffentlichung von Inhalten ohne tiefgehende Datenbankkenntnisse.

2. Benutzerfreundlichkeit:
   - CMS-Plattformen sind darauf ausgelegt, benutzerfreundlich zu sein, mit grafischen Benutzeroberflächen, die das Erstellen und Bearbeiten von Inhalten erleichtern. Datenbanken erfordern in der Regel spezielle Kenntnisse in SQL oder anderen Abfragesprachen.
3. Integration und Erweiterbarkeit:
   - CMS-Systeme bieten oft integrierte Lösungen und Plugins, die die Funktionalität erweitern und die Integration mit anderen Systemen erleichtern. Datenbanken erfordern oft maßgeschneiderte Lösungen und zusätzliche Programmierung für ähnliche Funktionen.

Wir halten also fest: Ein CMS bietet eine benutzerfreundliche Plattform, die es ermöglicht, Inhalte schnell und mit geringen technischen Kenntnisse zu aktualisieren. Im folgenden Abschnitt, wollen wir nun auf den spezifischeren Begriff 'Headless CMS' eingehen, und warum dieser für uns relevant ist.

2.1.1 Headless CMS

Traditionelle CMS und Headless CMS sind zwei unterschiedliche Ansätze, die sich in ihrer Architektur und Funktionalität unterscheiden. Ein traditionelles CMS, auch bekannt als gekoppeltes CMS, ist ein System, bei dem die Verwaltung der Inhalte und die Präsentation der Inhalte eng miteinander gekoppelt sind. Es bietet eine Benutzeroberfläche zur Erstellung, Bearbeitung und Verwaltung von Inhalten sowie zur Gestaltung des Frontend-Designs der Website. Auch Wordpress bietet einen solchen Website-Builder[3] an. Bei einem gekoppelten CMS sind Backend und Frontend also untrennbar miteinander verbunden, was bedeutet, dass Änderungen im Backend direkte Auswirkungen auf das Frontend haben. Diese Art von CMS eignet sich ideal für Websites mit einer einzigen Präsentationsschicht, bei denen Inhalt und Design eng miteinander verknüpft sind.

Im Gegensatz dazu trennt ein Headless CMS, auch bekannt als entkoppeltes CMS, die Inhaltsverwaltung von der Präsentationsschicht. Es fungiert als reines Backend-System, das sich auf die Verwaltung und Bereitstellung von Inhalten über eine oder mehrere APIs konzentriert. Dies erlaubt, beziehungsweise erfordert, das Frontend separat zu entwickeln. Die Präsentationsschicht wird also separat entwickelt und kann mit verschiedenen Technologien und Frameworks erstellt werden. Daher stammt auch der Begriff Headless CMS, da der Körper (Die Inhalte) gleich bleiben, aber der Kopf (die Präsentation) beliebig ausgetauscht werden kann. Diese Entkopplung zwischen Inhalten und Präsentation ermöglicht eine größere Flexibilität und Skalierbarkeit, da die Inhalte über APIs an verschiedene Frontend-Anwendungen, wie Websites, mobile Apps oder andere digitale Plattformen, geliefert wer-

[3] `https://wordpress.com/website-builder/`

den können. WordPress kann sowohl als traditionelles gekoppeltes CMS als auch als Headless CMS eingesetzt werden. In seiner Standardkonfiguration ist WordPress ein gekoppeltes CMS, bei dem Inhaltsverwaltung und Präsentation eng miteinander verbunden sind. Mit der Einführung der WordPress REST API hat WordPress jedoch auch die Möglichkeit, als Headless CMS zu fungieren. Durch die Nutzung der REST API können Entwickler auf die Inhalte des WordPress Backend zugreifen und diese in verschiedene Frontend-Anwendungen integrieren, ohne an das Standard-Frontend von WordPress gebunden zu sein [Con24] . Dadurch eröffnen sich neue Möglichkeiten für die Erstellung von maßgeschneiderten Websites (wie unsere Applikation eine sein wird), mobilen Apps und anderen digitalen Erlebnissen, die auf WordPress als Backend-System aufbauen.

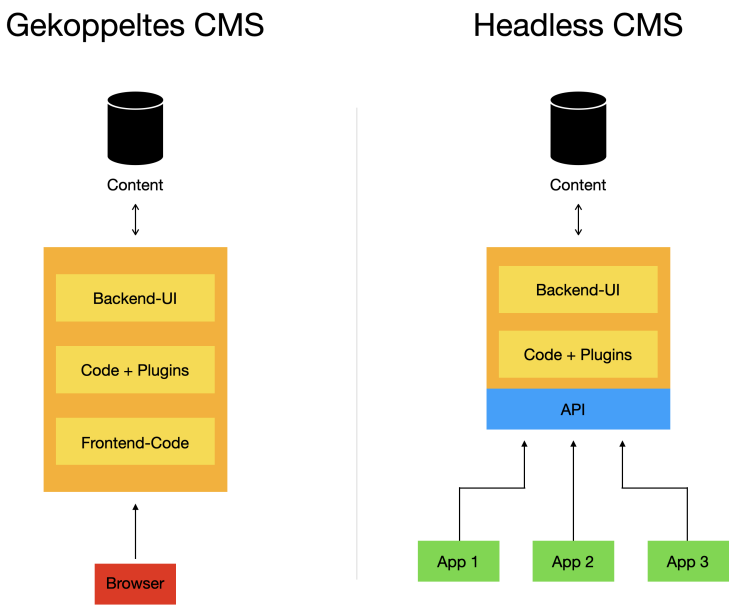

Abbildung 2.2: Vereinfachte Darstellung zur Illustration des Unterschieds zwischen gekoppeltem und Headless CMS

Abbildung 2.2 illustriert den Unterschied zwischen einem gekoppelten CMS und einem Headless CMS in vereinfachter Form. Der fundamentale Unterschied besteht darin, dass bei einem gekoppelten CMS die HTML-Templates vom CMS selbst dem Client (in der Regel, aber nicht zwingend, ein Browser-Programm auf dem Endgerät des Nutzers) bereitgestellt werden. Im Gegensatz dazu nutzen Web-Applikationen bei einem Headless CMS die vom CMS zur Verfügung gestellte REST API, um auf die gespeicherten Inhalte zuzugreifen. Die Applikationen sind dann selbst dafür verantwortlich, diese Daten in entsprechende HTML-Templates oder mobile Views einzubetten.

Dieser Zusammenhang wird dem Leser in den späteren Kapiteln, wenn wir uns der konkreten Architektur unserer Applikation widmen, noch deutlicher werden.

Zusammenfassend lässt sich konstatieren, dass traditionelle CMS eine enge Kopplung zwischen Inhaltsverwaltung und Präsentation aufweisen, während Headless CMS eine Entkopplung dieser Komponenten ermöglichen. WordPress, obwohl ursprünglich als gekoppeltes CMS

konzipiert, kann dank seiner REST API [Con24] auch als Headless CMS eingesetzt werden. Dies bietet erhebliche Flexibilität für diverse Anforderungen in der modernen Webentwicklung.

2.2 LDAP

LDAP[4] (Lightweight Directory Access Protocol) nimmt eine zentrale Rolle bei der Benutzerauthentifizierung innerhalb unserer Applikation ein und bildet darüber hinaus zum gegenwärtigen Zeitpunkt (Mai 2024) das Fundament für die Authentifizierung von Benutzern in der gesamten Service-Landschaft der Bundeswehr Universität München. Um ein grundlegendes Verständnis dieser Technologie zu vermitteln , werden wir im Folgenden näher auf das Wesen und die Funktionsweise von LDAP eingehen.

LDAP ist ein offener und herstellerunabhängiger Protokollstandard, der die Kommunikation zwischen Verzeichnisdiensten und Client-Anwendungen ermöglicht. Es wurde entwickelt, um den Zugriff auf hierarchisch strukturierte Verzeichnisinformationen über ein IP-Netzwerk zu vereinfachen und zu standardisieren. Ein LDAP-Verzeichnis ist eine spezialisierte Datenbank, die zur Speicherung und Organisation von Informationen über Ressourcen in einem Netzwerk dient. Diese Ressourcen können Benutzer, Gruppen, Geräte, Anwendungen und andere Objekte sein.

Die Daten in einem LDAP-Verzeichnis sind in einer baumartigen Struktur organisiert, die als Directory Information Tree (DIT) bezeichnet wird. Jeder Eintrag im DIT besteht aus einem eindeutigen Identifikator, dem Distinguished Name (DN), und einer Reihe von Attributen, die die Eigenschaften des Objekts beschreiben. Dieser Eigenschaft werden wir uns im Rahmen des Authentifizierungsworkflows der Applikation noch näher zuwenden.

LDAP bietet eine standardisierte Möglichkeit, auf diese Verzeichnisinformationen zuzugreifen und sie zu verwalten. Client-Anwendungen können mithilfe von LDAP-Anfragen Daten aus dem Verzeichnis abrufen, ändern, hinzufügen oder löschen. Das Protokoll unterstützt verschiedene Operationen wie Suche, Vergleich, Authentifizierung und Modifikation von Einträgen.

Eine der Hauptaufgaben von LDAP besteht darin, eine zentrale Verwaltung von Benutzern, Gruppen und anderen Ressourcen in einer Organisation zu ermöglichen. Durch die Verwendung eines LDAP-Verzeichnisses können Administratoren Benutzerkonten, Berechtigungen und Zugriffssteuerungen an einer zentralen Stelle verwalten, anstatt sie für jede Anwendung oder jedes System separat zu konfigurieren. Dies vereinfacht die Administration und reduziert den Aufwand bei der Verwaltung von Benutzerdaten in großen Netzwerken.

LDAP wird häufig in Unternehmensumgebungen, Bildungseinrichtungen und anderen Organisationen eingesetzt, um die Benutzerverwaltung zu zentralisieren und zu vereinheitlichen. Es dient als zentrale Anlaufstelle für die Authentifizierung und Autorisierung von Benutzern und ermöglicht eine konsistente und effiziente Verwaltung von Identitäten und Zugriffsrechten [SWAH19].

Darüber hinaus kann LDAP auch zur Integration verschiedener Anwendungen und Systeme

[4] `https://openldap.org/doc/`

innerhalb einer Organisation verwendet werden. Durch die Nutzung von LDAP als gemeinsame Verzeichnisquelle können Anwendungen Benutzerdaten abrufen und synchronisieren, ohne separate Benutzerdatenbanken verwalten zu müssen. Dies fördert die Interoperabilität und reduziert die Komplexität der Systemlandschaft.

Insgesamt ist LDAP ein leistungsfähiges Protokoll, das eine standardisierte Methode zur Verwaltung und zum Zugriff auf Verzeichnisinformationen bietet. Es spielt eine wichtige Rolle bei der Zentralisierung der Benutzerverwaltung, der Vereinfachung der Systemadministration und der Integration verschiedener Anwendungen und Dienste in einer Organisation.

In Kapitel 6 werden wir, wie bereits angesprochen, noch näher auf die exakte Rolle von LDAP im Kontext unserer Web-Applikation eingehen. Das dritte und letzte hoch aggregierte Konzept, welches dem Leser mit an die Hand gegeben werden soll, bevor er sich dem Kern der Arbeit widmet, ist das Konzept der Frameworks, speziell im Kontext des Web.

2.3 Web-Frameworks

Bevor wir uns in die spezifischen Kategorien der Web-Frameworks vertiefen, ist es essentiell, ein grundlegendes Verständnis dafür zu schaffen, was ein Framework im Allgemeinen und ein Web-Framework im Besonderen ausmacht. Frameworks, zu Deutsch Rahmenwerke, sind mächtige Werkzeuge in der Softwareentwicklung, die dazu dienen, den Entwicklungsprozess zu strukturieren, zu vereinfachen und zu beschleunigen [Dev22].

Im Kern ist ein Framework eine Sammlung von Bibliotheken, Tools und Konventionen, die einen vordefinierten Rahmen für die Entwicklung von Anwendungen bieten. Es stellt wiederverwendbare Komponenten, Entwurfsmuster und Best Practices bereit, die Entwickler nutzen können, um effizient und konsistent komplexe Softwaresysteme zu erstellen. Frameworks abstrahieren häufig wiederkehrende Aufgaben und Problemstellungen, sodass sich Entwickler auf die spezifischen Anforderungen ihrer Anwendung konzentrieren können, anstatt sich mit grundlegenden Infrastrukturkomponenten auseinandersetzen zu müssen.

Im Bereich der Webentwicklung haben sich Web-Frameworks als unverzichtbare Hilfsmittel etabliert. Sie adressieren die besonderen Herausforderungen und Anforderungen, die bei der Erstellung interaktiver und dynamischer Webanwendungen auftreten. Web-Frameworks bieten eine Strukturierung und Modularisierung des Codes, erleichtern die Implementierung gängiger Funktionalitäten wie Routing, Templating und Datenbankzugriff und fördern die Einhaltung bewährter Entwurfsprinzipien wie MVC[5] (Model-View-Controller).

Durch den Einsatz von Web-Frameworks können Entwickler ihre Produktivität steigern, indem sie auf robuste und getestete Codebibliotheken zurückgreifen, die häufige Probleme lösen und bewährte Muster implementieren. Frameworks bieten oft auch umfangreiche Dokumentationen, lebendige Communitys und Ökosysteme von Erweiterungen und Plugins, die den Entwicklungsprozess weiter unterstützen und beschleunigen.

Es ist jedoch wichtig zu beachten, dass die Wahl eines geeigneten Web-Frameworks sorgfältig abgewogen werden muss. Jedes Framework hat seine eigenen Stärken, Schwächen und Philosophien, die mit den spezifischen Anforderungen und Zielen des jeweiligen Projekts in

[5] `https://www.geeksforgeeks.org/mvc-design-pattern/`

Einklang gebracht werden müssen. Faktoren wie Lernkurve, Performanz, Skalierbarkeit, Community-Unterstützung und Kompatibilität mit anderen Technologien sollten bei der Entscheidung für ein Framework berücksichtigt werden.

Um die Vielfalt der verfügbaren Web-Frameworks zu strukturieren, werden wir sie in zwei Hauptkategorien unterteilen: Component-Frameworks und Meta-Frameworks. Meta-Frameworks sind hierbei eine recht neue Entwicklung, welche die Grenzen zwischen einem Backend- und einem Frontend-Framework verschwimmen lässt [Cai23]. Diese Kategorien spiegeln die unterschiedlichen Schwerpunkte und Anwendungsbereiche wider und ermöglichen eine gezieltere Betrachtung der Frameworks in ihrem jeweiligen Kontext. In den folgenden Abschnitten werden wir uns eingehend mit beiden Kategorien befassen und ihre Bedeutung für die moderne Webentwicklung diskutieren.

2.3.1 Component-Frameworks

In der traditionellen Webentwicklung wurden HTML, CSS und JavaScript oft als separate Einheiten betrachtet und in unterschiedlichen Dateien organisiert. HTML diente der Strukturierung des Inhalts, CSS der visuellen Gestaltung und JavaScript der Interaktivität und Dynamik. Mit dem Aufkommen von Component-Frameworks hat sich jedoch ein Paradigmenwechsel vollzogen, der diese strikte Trennung aufhebt und eine komponentenbasierte Entwicklung in den Vordergrund stellt [Den24].

Component-Frameworks basieren auf dem Konzept der Komponenten, die HTML, CSS und JavaScript in einer einzigen Einheit zusammenfassen. Eine Komponente ist ein wiederverwendbarer und modularer Baustein einer Benutzeroberfläche, der sowohl die Struktur als auch das Aussehen und das Verhalten enthält. Anstatt die verschiedenen Aspekte einer Komponente auf mehrere Dateien zu verteilen, ermöglichen Component-Frameworks die Erstellung von Komponenten, die alle relevanten Bestandteile in einer einzigen Datei vereinen. Komponenten bieten eine Strukturierung und Abstraktion für die Erstellung dynamischer und reaktiver Benutzererlebnisse.

Dieser Ansatz bietet mehrere Vorteile. Erstens fördert er eine bessere Kapselung und Modularität des Codes. Alle Teile einer Komponente sind an einem Ort zentralisiert, was die Lesbarkeit und Wartbarkeit verbessert. Entwickler können sich auf eine Komponente konzentrieren, ohne zwischen verschiedenen Dateien navigieren zu müssen. Zweitens erleichtert die Zusammenführung von HTML, CSS und JavaScript in einer Datei die Wiederverwendbarkeit von Komponenten. Sie können einfach in verschiedene Teile der Anwendung importiert und integriert werden, ohne dass zusätzliche Schritte erforderlich sind.

Im Gegensatz zu Backend-Frameworks, die sich auf die serverseitige Logik und Datenverarbeitung fokussieren, sind Component-Frameworks primär für die Gestaltung und Funktionalität der Benutzeroberfläche zuständig, die sich aus eben diesen Komponenten zusammensetzt. Es ist dienlich, sich eine Website als eine Komposition von Web-Komponenten vorzustellen, und Frameworks vereinfachen das Programmieren und die Entwicklung eben dieser Komponenten.

Auf den Endnutzer der Seite hat dies jedoch keinen direkten Einfluss, sondern der Hauptvorteil der Verwendung solcher Frameworks liegt in dem Produktivitätszuwachs der Entwickler, welche diese beherrschen und nutzen. Die Begriffe Frontend-Framework und Component-

Framework sind heutzutage austauschbar geworden, wobei die Bezeichnung Component-Framework daher rührt, dass diese Frameworks der Strukturierung von Komponenten dienen [Den24]. Das Component-Framework, welches wir im Rahmen dieser Arbeit verwenden werden, nennt sich Svelte [6]. Auf die besonderen Eigenschaften von Svelte als Component-Framework werden wir noch in 4.1 eingehen.

2.3.2 Meta-Frameworks

In der Welt der modernen Webentwicklung haben sich Component-Frameworks wie Svelte als leistungsstarke Werkzeuge etabliert, um interaktive und reaktive Benutzeroberflächen zu erstellen [Dev22] . Sie ermöglichen die Entwicklung modularer und wiederverwendbarer Komponenten, die HTML, CSS und JavaScript in einer einzigen Einheit zusammenfassen. Obwohl diese Component-Frameworks einen großen Schritt nach vorn darstellen, reichen sie allein oft nicht aus, um eine vollumfängliche Webanwendung bereitzustellen.

Traditionell war die Softwarearchitektur von Webanwendungen so strukturiert, dass Frontend- und Backend-Code klar voneinander getrennt waren und in unterschiedlichen Codebases lagen [Win23]. Das Frontend konzentrierte sich auf die Benutzeroberfläche und die Interaktion mit dem Benutzer, während das Backend für die Verarbeitung von Anfragen, die Datenbankinteraktion und die Geschäftslogik zuständig war. Diese Trennung ermöglichte eine klare Zuordnung von Verantwortlichkeiten, führte jedoch auch zu einer gewissen Komplexität bei der Entwicklung und Wartung von Webanwendungen.

An diesem Punkt kommen Meta-Frameworks ins Spiel [Pri23]. Meta-Frameworks sind eine Abstraktionsschicht oberhalb von Component-Frameworks und bieten zusätzliche Funktionalitäten und Strukturen, um die Entwicklung vollständiger Webanwendungen zu erleichtern. Sie kombinieren die Vorteile von Component-Frameworks mit serverseitigen Fähigkeiten und bieten eine ganzheitliche Lösung für die Erstellung moderner Webanwendungen.

Ein wichtiger Aspekt von Meta-Frameworks ist die Integration von Frontend- und Backend-Funktionalitäten in einem einzigen Framework. Sie ermöglichen es Entwicklern, den gesamten Stack einer Webanwendung mit einem konsistenten und einheitlichen Ansatz zu entwickeln. Dies soll durch die nachfolgende Abbildung veranschaulicht werden.

[6] `https://svelte.dev/`

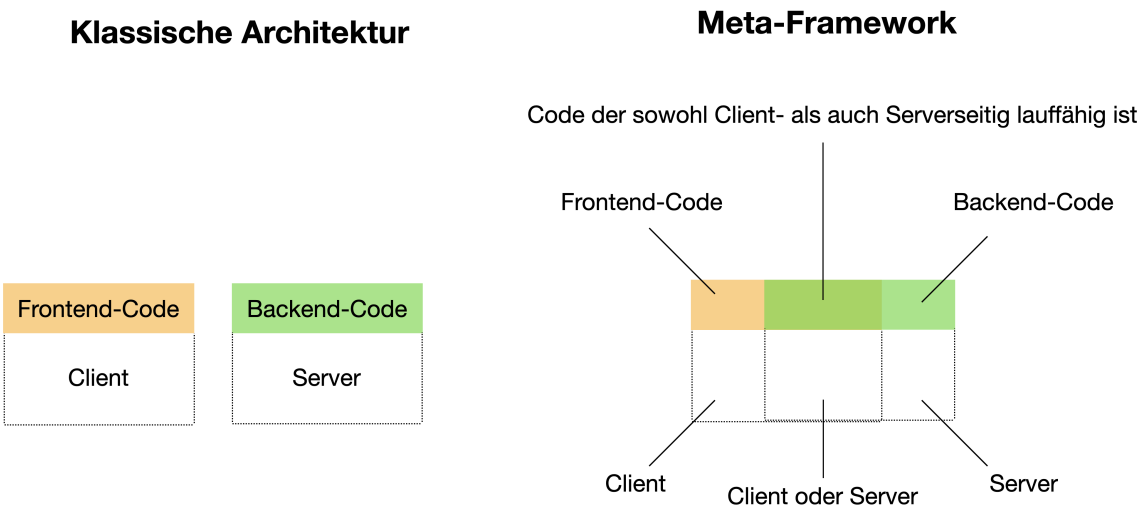

Abbildung 2.3: Abgrenzung des Unterschieds zwischen der klassischen Web-Architektur und Meta-Frameworks

Um die Abbildung 2.3 nicht zu überladen, wurde hier darauf verzichtet, die Aufteilung zwischen Codebases zu visualisieren. Jedoch ist dies ein elementarer Aspekt, den der Leser in jedem Fall verstanden haben sollte:

- **Klassische Architektur** : Der Frontend-Code der Applikation wird in seiner eigenen Codebase gehalten und der Backend-Code wird ebenfalls in seiner eigenen Codebase gehalten.
- **Meta-Framework** : Sowohl der Frontend-Code als auch der Backend-Code werden in der gleichen Codebase gehalten.

Denn anstatt separate Codebases für Frontend und Backend zu verwalten, bieten Meta-Frameworks eine nahtlose Verbindung zwischen den beiden Bereichen. Meta-Frameworks erweitern also die Funktionalität von Component-Frameworks, indem sie zusätzliche Funktionen bereitstellen, die für die Entwicklung vollständiger Webanwendungen erforderlich sind. Dazu gehören beispielsweise:

- **Routing**: Meta-Frameworks bieten leistungsstarke Routing-Mechanismen, mit denen Entwickler Anwendungs-Routes definieren und die Navigation zwischen verschiedenen Seiten und Komponenten der Anwendung verwalten können. Serverseitiges Rendering (SSR): Mit Meta-Frameworks können Entwickler ihre Webanwendungen serverseitig rendern lassen, was zu einer verbesserten Performanz, einer besseren Suchmaschinenoptimierung (SEO) und einer schnelleren Erstladezeit führt.
- **API-Integration** : Meta-Frameworks erleichtern die Integration von Backend-APIs, indem sie Werkzeuge und Konventionen bereitstellen, um mit serverseitigen Diensten zu kommunizieren und Daten abzurufen. Auch die sonst für das Backend vorbehaltene Fähigkeit mit einer Datenbank zu kommunizieren ist in einem Meta-Framework möglich.

- **Zustandsverwaltung**: Meta-Frameworks bieten oft integrierte Lösungen zur Verwaltung des Anwendungszustands, sei es durch eigene Zustandsverwaltungssysteme oder durch die Integration mit externen Bibliotheken wie Redux [7]. Svelte und SvelteKit benötigen jedoch kein solches Zustandsverwaltungssystem, was einer der Aspekte ist, der diese Kombination so attraktiv macht. Darauf werden wir jedoch noch in Kapitel 4 eingehen
- **Build-Prozess und Deployment**: Meta-Frameworks vereinfachen den Build-Prozess und das Deployment von Webanwendungen, indem sie vorkonfigurierte Build-Werkzeuge, Optimierungen und Bereitstellungsoptionen bereitstellen.

Beispiele für Meta-Frameworks sind Next.js[8] für React [9] (Component-Framework), Nuxt.js [10] für Vue.js[11] (Ebenfalls ein Component Framework) und SvelteKit für Svelte. Auch hier soll dazu gesagt sein, dass die Funktionalität welche durch Meta-Frameworks bereitgestellt wird, auch mit der klassischen Aufteilung einer Web-Applikation in separate HTML, CSS und JavaScript Dateien umsetzbar ist. Doch genau wie es bei Component-Frameworks der Fall ist, stellen die Meta-Frameworks eine produktivere Entwicklungsumgebung für den Programmierer zur Verfügung [Pri23]. Die genannten Meta-Frameworks bauen auf den jeweiligen Component-Frameworks auf und erweitern sie um zusätzliche Funktionen. Wie genau sich Svelte und SvelteKit ergänzen, werden wir jedoch noch ausführlich in 4 besprechen. Wir halten also nochmals fest: Durch die Kombination von Component-Frameworks mit den Fähigkeiten von Meta-Frameworks können Programmierer produktiver arbeiten, indem sie sich auf die Erstellung von Funktionen konzentrieren, anstatt sich mit der Komplexität der Infrastruktur der Applikation, und der Kommunikation zwischen Front- und Backend, sowie der Pflege beider damit assoziierten Codebases, auseinanderzusetzen. Meta-Frameworks bieten somit eine vollintegrierte Lösung, die Frontend- und Backend-Aspekte miteinander verbindet und es Entwicklern ermöglicht, moderne und leistungsstarke Webanwendungen schneller und effizienter zu erstellen. Um besser zu verstehen, wie Web-Applikationen mit Ressourcen integriert werden können, wollen wir im folgenden auf REST APIs eingehen.

2.4 REST API

Im Kontext der Entwicklung moderner Webanwendungen spielt das Verständnis von REST (Representational State Transfer) APIs eine zentrale Rolle, da sie für die Kommunikation zwischen verschiedenen Komponenten von entscheidender Bedeutung sind. Eine API (Application Programming Interface) ist bekanntlich eine nach außen verfügbare Schnittstelle, über welche eine Softwarekomponente mit einer anderen Softwarekomponente oder dem Benutzer interagieren kann [Ama24]. Eine REST API ist dabei eine Verfeinerung des API-Begriffs [Ama24] , welche den folgenden Anforderungen genügt:

- **Ressourcenorientiert**: Jede Ressource (z.B. Datenbankeinträge) hat eine eindeutige URL.

[7] `https://redux.js.org/`
[8] `https://nextjs.org/`
[9] `https://react.dev/`
[10] `https://nuxt.com/`
[11] `https://vuejs.org/`

- **Verben und HTTP-Methoden**: Nutzt HTTP-Methoden (GET, POST, PUT, DELETE), um Operationen durchzuführen.
- **Zustandslos** : Jede Anfrage vom Client an den Server muss alle notwendigen Informationen enthalten, und der Server speichert keine Client-Zustände zwischen den Anfragen.
- **Repräsentationen**: Ressourcen können in verschiedenen Formaten dargestellt werden (z.B. JSON, XML).

Eine REST API ist also im Grunde genommen eine Softwareschicht, die als Schnittstelle zwischen der Webanwendung und den zugrunde liegenden Daten dient. Dies sei in der folgenden Abbildung schematisch dargestellt:

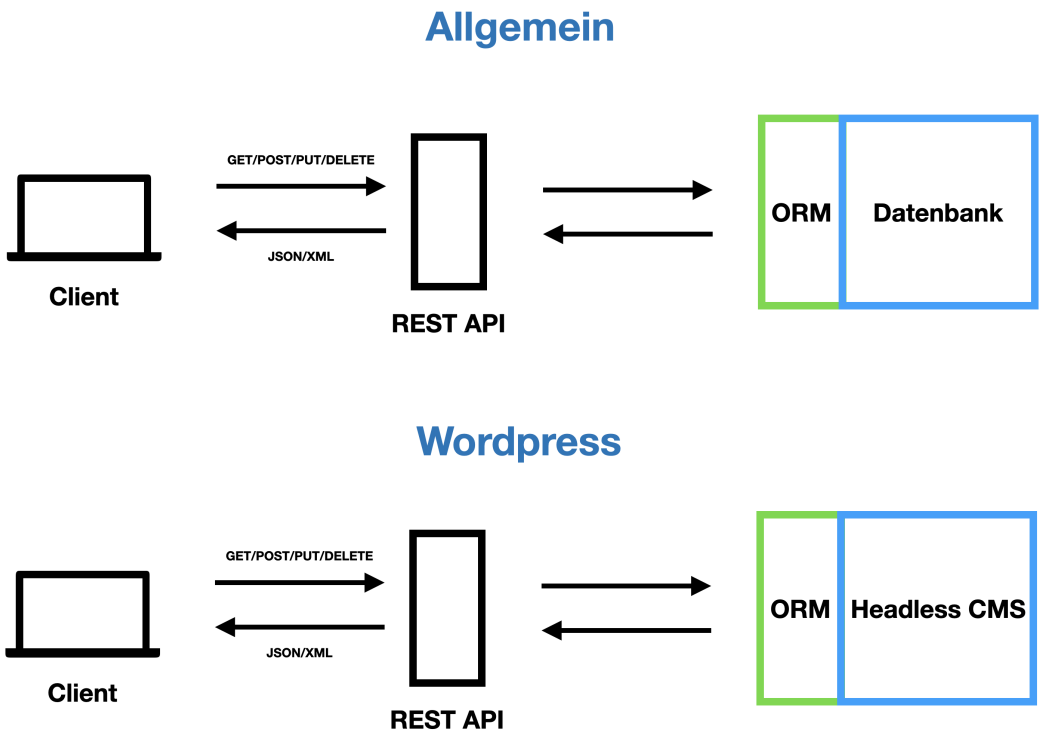

Abbildung 2.4: Illustration der Rolle einer REST API

Die REST API selbst ist nicht an eine bestimmte Maschine oder einen bestimmten Ort gebunden. Sie kann auf demselben Server wie die Datenbank laufen, muss es aber nicht zwingend. In vielen Fällen wird die REST API auf einem separaten Anwendungsserver bereitgestellt, während die Datenbank auf einem dedizierten Datenbankserver läuft. Verwendet man WordPress als Headless CMS, befinden sich die Datenbank und die REST API in der Regel auf derselben Maschine oder demselben Server. Dies liegt daran, dass WordPress selbst eine vollständige Webanwendung ist, die sowohl die Datenbank als auch die REST API bereitstellt.

Die Hauptaufgabe der REST API besteht darin, die eingehenden HTTP-Anfragen zu interpretieren, die erforderlichen Daten aus der Datenbank abzurufen oder zu manipulieren und die Ergebnisse in einem geeigneten Format (z.B. JSON) an die Webanwendung zurückzusenden. Dabei fungiert die REST API als eine Art Übersetzer zwischen den HTTP-Anfragen und den Datenbankoperationen. Es ist jedoch wichtig zu beachten, dass die REST API nicht direkt SQL-Abfragen ausführt. Stattdessen verwendet sie oft eine Datenbankabstraktionsschicht (In Abbildung 2.4 grün markiert) oder ein ORM [12] (Object-Relational Mapping), um mit der Datenbank zu interagieren. Diese Abstraktionsschicht übersetzt die Anfragen der REST API in die entsprechenden Datenbankoperationen (z.B. SQL-Abfragen) und stellt eine objektorientierte Schnittstelle für den Zugriff auf die Daten bereit.

Die Verwendung einer Abstraktionsschicht hat mehrere Vorteile. Sie entkoppelt die REST API von den spezifischen Details der Datenbank und ermöglicht eine einheitliche Schnittstelle für den Datenzugriff. Dadurch wird die Wartbarkeit und Flexibilität der Anwendung verbessert, da Änderungen an der Datenbank keine direkten Auswirkungen auf die REST API haben [Ama24]. Dies ist im Kontext dieser Arbeit nicht direkt relevant, da wir die bereits voll implementierte WordPress REST API verwenden, jedoch schadet es nicht, die zugrunde liegenden Komponenten und deren lose Kopplung nachvollzogen zu haben.

Zusammenfassend lässt sich sagen: Die WordPress REST API bietet eine leistungsstarke Schnittstelle für den Zugriff auf WordPress-Inhalte und -Funktionalitäten und ermöglicht die nahtlose Integration in die zu entwickelnde Anwendung. Durch die Verwendung von REST-Prinzipien und die Verarbeitung von JSON-Daten können Entwickler effizient mit der API interagieren und die Inhalte von WordPress in ihre Anwendung einbinden.

Das Einführen des Lesers in die Grundlagen ist somit abgeschlossen. Das Fundament für das weitere Verständnis der Arbeit ist gelegt, und wir wenden uns im Folgenden dem Verständnis der Problemstellung zu.

[12] `https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/`

3 Gesamtkontext und Systemarchitektur

Innerhalb dieses Kapitels, wollen wir nun ein Verständnis für die Problemstellung aufbauen. Aus dem Verständnis der bestehenden Problemstellung soll dann das Verständnis des gewählten Lösungsansatzes erwachsen. Für das initiale Verständnis der Problemstellung wollen wir uns dem Kontext und der IT-Architektur des Problems widmen. Im Anschluss daran werden wir diskutieren, warum ein Meta-Framework eine probate Lösung für die vorliegende Problemstellung ist.

3.1 Architekturübersicht

Wir wollen jetzt die in Kapitel 2 besprochenen Grundlagen als Teilkomponenten eines Gesamtsystems betrachten, und somit auch die Beziehung zwischen diesen Komponenten beleuchten. Die Komponenten, wie sie bereits eingeführt wurden, sind:

1. Headless Wordpress CMS
2. LDAP
3. Web-Applikation

Die Web-Applikation wird, zur Vereinfachung, zu diesem Zeitpunkt als in sich geschlossene Entität angenommen und dargestellt. Eine feinkörnige Detaillierung der Architektur der Web-Applikation wird in Kapitel 3 vorgenommen. Darüber hinaus werden wir auch den User selbst als Komponente betrachten. Um eine Diskussionsgrundlage zu schaffen, ist im Folgenden eine vereinfachte Darstellung der Komponenten und deren Relationen gegeben.

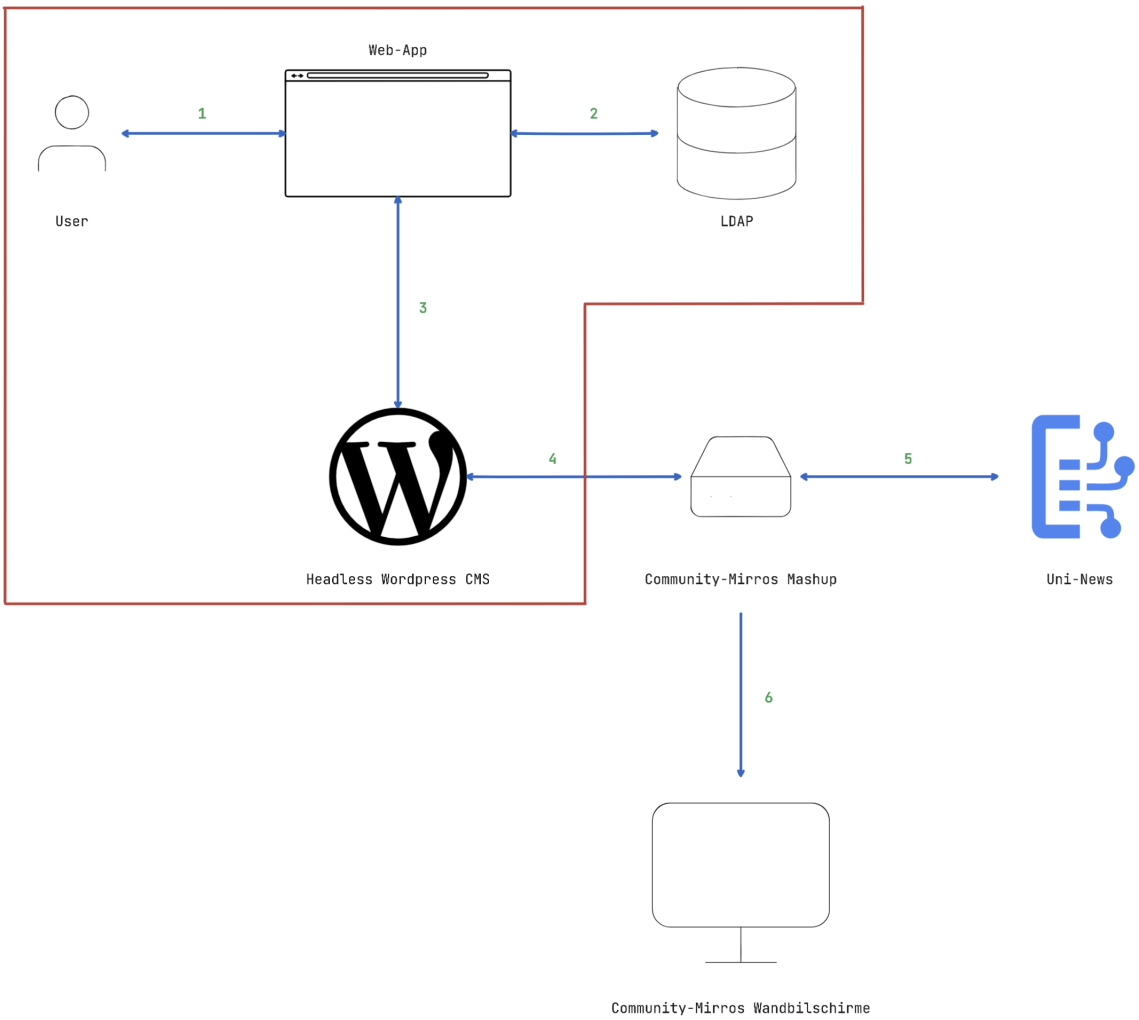

Abbildung 3.1: Vereinfachte Illustration der Gesamtarchitektur der Problemstellung

In der Darstellung sind damit die Komponenten, Mashup, Uni-News und Wandbildschirme hinzugekommen. Diese Komponenten sind jedoch nicht direkt für diese Arbeit von Relevanz, sondern wurde für eine bessere Kontextualisierung mit aufgeführt. Dem Leser soll somit ermöglicht werden, den gesamtheitlichen Sinn hinter der in dieser Arbeit behandelten Web-Applikation verstehen. Die für die Arbeit relevantesten Kern-Komponenten befinden sich innerhalb der roten Grenze. Alle sich innerhalb dieser Grenze befindenden Komponenten spielen somit im Kontext der Arbeit eine elementare Rolle. Wir werden als Erstes die Komponenten innerhalb der Markierung nochmals näher im Kontext der vorliegenden Problemstellung erläutern, und anschließend die weniger relevanten, außerhalb der Grenze liegenden Komponenten. Daraufhin, werden die Relationen 1–6 (In Abbildung 3.1 grün markiert) zwischen den Komponenten näher erläutert.

Komponenten innerhalb

Die Komponenten denen eine besondere Relevanz im Kontext der Arbeit zukommt, und visuell dadurch hervorgehoben sind, dass sie innerhalb der roten Grenze liegen:

- **User**: Der User, welcher mit der Web-Applikation interagiert. Da es sich bei der Applikation um ein Self-Service Portal handelt, beinhaltet die Gruppe der User alle Personen, welche ein Profil innerhalb des Wordpress CMS besitzen, und dieses verwalten wollen. Hierbei handelt es sich vordergründing um Angehörige der Universität, welche von der Informatik Fakultät [1] beschäftigt werden.
- **Web-App**: Die Web-App, hier zu einer einfachen Komponente abstrahiert, stellt dem User ein Interface zur Verfügung, über welches er, nach erfolgreicher Authentifikation, lesenden und schreibenden Zugriff auf sein Nutzerprofil hat.
- **LDAP**: Der LDAP-Server der Universität, welcher für die Authentifikation des Users genutzt wird. Auf den genauen Fluss der User-Authentifikation wird in Kapitel 5.3.1 eingegangen.
- **Headless Wordpress CMS**: Das Headless WordPress CMS dient als primäre Datenquelle für die Benutzerprofile. Es speichert die Meta-Daten der Benutzer, wie beispielsweise Namen, akademische Titel und Kontaktinformationen. Diese Daten bilden die Grundlage für die Darstellung der Benutzerprofile auf den Wandbildschirmen, als auch die Datengrundlage für die Web-App.

Komponenten außerhalb

Diese Komponenten sind für die Arbeit zwar nicht direkt relevant, doch werden dennoch oberflächlich behandelt, da sie zum gesamtheitlichen Verständnis der Arbeit beitragen. Wir halten kurz fest, was ein Mashup ist, und welche Rolle es einnimmt, bevor wir es als Komponente in der Architektur kontextualisieren. Das Mashup ist ein Service, welcher die Aufgabe übernimmt, die Daten aus dem Headless WordPress CMS und der Uni-News API intelligent zu kombinieren und aufzubereiten [NFK+10]. Es führt die Meta-Daten der Benutzer mit den entsprechenden Veröffentlichungsdaten zusammen und erstellt somit eine kohärente und aussagekräftige Darstellung der Benutzerprofile. Durch die Verknüpfung und Anreicherung der Daten schafft das Mashup einen Mehrwert, der über die einzelnen Datenquellen hinausgeht.

- **Community-Mirrors Mashup**: In der dargestellten Systemarchitektur nimmt das Mashup eine zentrale Rolle ein, indem es als Vermittler zwischen den verschiedenen Datenquellen und den Wandbildschirmen fungiert. Das Mashup ist verantwortlich für die Aggregation, Verarbeitung und Zusammenführung der Daten aus dem Headless WordPress CMS und der Uni-News API.
- **Uni-News** : Die Uni-News API hingegen liefert aktuelle Informationen zu den wissenschaftlichen Veröffentlichungen der Benutzer. Diese API fungiert als zusätzliche Datenquelle, um die Benutzerprofile mit relevanten und zeitnahen Daten anzureichern. Durch

[1] `https://www.unibw.de/inf`

die Integration der Veröffentlichungsdaten können die Benutzerprofile um wichtige Informationen wie Publikationstitel, Erscheinungsdatum und Zitierhäufigkeit erweitert werden.

- **Community-Mirrors Wandbildschirme** : Die Wandbildschirme dienen schließlich als Präsentationsmedium für die aufbereiteten Benutzerprofile. Sie visualisieren die durch das Mashup generierten Informationen in einem ansprechenden und leicht verständlichen Format. Die Wandbildschirme ermöglichen es den Betrachtern, sich schnell einen Überblick über die Profile der Benutzer zu verschaffen und deren aktuelle wissenschaftliche Aktivitäten zu verfolgen. Durch die Darstellung auf den Wandbildschirmen wird die Sichtbarkeit und Präsenz der Benutzer und ihrer Forschungsarbeit innerhalb des Informatik-Instituts erhöht[2].

Relationen

Nun wollen wir auf die Relationen zwischen den einzelnen Komponenten eingehen. Die Nummerierungen entsprechen dabei der in Abbildung 3.1 enthaltenen grünen Nummerierungen.

- **1: Relation zwischen User und Web-App**: Die Interaktion zwischen dem User und der Web-App ist bidirektional und ermöglicht einen dynamischen Austausch von Informationen. In der Richtung vom User zur Web-App hat der User die Möglichkeit, über eine Schnittstelle auf die Inhalte des Content-Management-Systems (CMS) schreibend zuzugreifen. Dies erlaubt dem User, seine Profilinformationen eigenständig zu pflegen und zu aktualisieren. In der entgegengesetzten Richtung, von der Web-App zum User, stellt die Web-App dem User seine individuellen Profilinformationen bereit. Durch diese Bereitstellung erhält der User eine transparente Einsicht in seine hinterlegten Daten und kann diese bei Bedarf direkt über die Web-App bearbeiten. Diese bidirektionale Kommunikation schafft eine nahtlose und benutzerfreundliche Interaktion zwischen dem User und der Web-App, die eine effiziente Verwaltung der Profildaten ermöglicht.
- **2: Relation Web-App LDAP**: Die Relation zwischen der Web-App und dem LDAP (Lightweight Directory Access Protocol) Server ist von entscheidender Bedeutung für die Authentifizierung der User. Wenn ein User versucht, sich über die Web-App anzumelden, übermittelt die Web-App die eingegebene User-ID und das entsprechende Passwort an den LDAP Server. Der LDAP Server fungiert als zentrale Authentifizierungsinstanz und überprüft die übermittelten Anmeldedaten gegen die im Verzeichnis hinterlegten Informationen. Je nachdem, ob die ID-Passwort-Kombination im LDAP Server vorliegt, sendet der LDAP Server entweder eine Bestätigung oder eine Ablehnung der Authentifizierungsanfrage an die Web-App zurück. Durch diesen gesicherten Austausch von Anmeldeinformationen zwischen der Web-App und dem LDAP Server wird sichergestellt, dass nur autorisierte User Zugriff auf die geschützten Ressourcen der Web-App erhalten. Diese Authentifizierungsmethode gewährleistet die Integrität und Vertraulichkeit der Benutzerdaten und schützt die Web-App vor unbefugtem Zugriff.
- **3: Relation zwischen Wordpress und Web-App**: Die Kommunikation zwischen

[2] `https://publicwiki.unibw.de/pages/viewpage.action?pageId=3546059`

der Web-App und dem Headless WordPress CMS basiert auf einer REST API. (Representational State Transfer). Die Web-App sendet HTTP-Anfragen an das CMS, um lesend oder schreibend auf die Inhalte zuzugreifen. Bei lesenden Anfragen (GET Requests[3]) fordert die Web-App spezifische Daten vom CMS an, wie beispielsweise Benutzerprofile oder Metadaten. Das CMS verarbeitet diese Anfragen und stellt die angeforderten Daten über die REST API bereit. Die Web-App empfängt die Daten und kann sie dann entsprechend aufbereiten und darstellen. Bei schreibenden Anfragen (POST Requests[4]) übermittelt die Web-App Daten an das CMS, um Inhalte zu erstellen, zu aktualisieren oder zu löschen. Das CMS empfängt diese Anfragen, verarbeitet sie und führt die entsprechenden Operationen auf der Datenbank aus. Durch diese bidirektionale Kommunikation zwischen der Web-App und dem Headless WordPress CMS wird eine flexible und entkoppelte Architektur ermöglicht, die eine effiziente Verwaltung und Bereitstellung von Inhalten erlaubt. Die Verwendung der REST API gewährleistet eine standardisierte und plattformunabhängige Schnittstelle, die eine nahtlose Integration zwischen der Web-App und dem CMS ermöglicht.

- **4: Relation zwischen Mashup und Headless WordPress CMS**: Das Mashup kommuniziert mit dem Headless WordPress CMS, um Meta-Daten zu den Nutzern abzufragen. Diese Meta-Daten umfassen den Status der Nutzer (Professor, wissenschaftlicher Mitarbeiter, Teamassistenz etc.), deren Vollnamen, die Adresse des Büros sowie eine kurze Beschreibung zur Person und ihrer Arbeit. Die Kommunikation zwischen dem Mashup und dem CMS erfolgt bidirektional. In regelmäßigen Intervallen, üblicherweise alle 24 Stunden, sendet das Mashup eine Anfrage an das CMS, um die aktuellen Daten abzurufen. Das CMS verarbeitet diese Anfrage und stellt dem Mashup die angeforderten Meta-Daten bereit. Durch diesen regelmäßigen Datenaustausch stellt das Mashup sicher, dass es stets über die aktuellsten Informationen zu den Nutzern verfügt. Die Verwendung eines Headless CMS ermöglicht eine flexible und skalierbare Bereitstellung der Daten, während das Mashup die Daten aggregiert und für die weitere Verarbeitung und Darstellung auf den Wandbildschirmen aufbereitet.
- **5: Relation zwischen Mashup und Uni-News**: Uni-News fungiert als externe Datenquelle, die dem Mashup aktuelle Informationen zu wissenschaftlichen Veröffentlichungen der Nutzer bereitstellt. Die Kommunikation zwischen dem Mashup und Uni-News erfolgt unidirektional, wobei das Mashup Daten von Uni-News abruft. Uni-News aggregiert und strukturiert wissenschaftliche Veröffentlichungen aus verschiedenen Quellen und stellt diese Daten über eine Schnittstelle bereit. Das Mashup nutzt diese Schnittstelle, um regelmäßig die neuesten Veröffentlichungsdaten abzufragen. Die abgerufenen Daten können beispielsweise den Titel der Veröffentlichung, die Autoren, das Veröffentlichungsdatum und weitere Metadaten enthalten. Durch die Integration der Daten von Uni-News erweitert das Mashup die Benutzerprofile um aktuelle Informationen zu deren wissenschaftlichen Publikationen. Diese Anreicherung der Profile ermöglicht es, ein umfassenderes Bild der akademischen Aktivitäten und Leistungen der Nutzer zu präsentieren.
- **6: Relation zwischen Mashup und Community-Mirrors Wandbildschirme**: Das Mashup bereitet die aggregierten Daten aus dem Headless WordPress CMS und

[3] `https://www.w3schools.com/tags/ref_httpmethods.asp`
[4] `https://www.w3schools.com/tags/ref_httpmethods.asp`

Uni-News auf und überträgt sie an die Community-Mirrors Wandbildschirme zur Anzeige.

In der vorangegangenen Analyse haben wir die Systemarchitektur eingehend betrachtet und die einzelnen Komponenten sowie deren Relationen detailliert beschrieben. Durch die Untersuchung der spezifischen Funktionen und Interaktionen der beteiligten Systeme konnten wir ein tieferes Verständnis für die Komplexität und die Herausforderungen der vorliegenden Problemstellung erlangen. Darüber hinaus haben wir eine Kategorisierung der Komponenten nach ihrer individuellen Wichtigkeit im Kontext dieser Arbeit vorgenommen. Diese Priorisierung ermöglicht es uns, den Fokus auf die kritischen Elemente der Architektur zu legen und deren Bedeutung für das Gesamtsystem hervorzuheben. Durch die Identifikation der Schlüsselkomponenten und deren Zusammenspiel können wir nun gezielter nach Lösungsansätzen suchen, die den spezifischen Anforderungen und Herausforderungen gerecht werden. An dieser Stelle richten wir unseren Blick auf Meta-Frameworks als vielversprechende Lösung für die vorliegende Problemstellung.

3.2 Meta-Frameworks als Lösung

Im Unterkapitel 2.3.1 haben wir bereits erläutert, was ein Component-Framework ist. In Unterkapitel 2.3.2 haben wir oberflächlich erklärt, was ein Meta-Framework ist und den Zusammenhang zwischen beiden Frameworks behandelt. Ebenfalls wurden innerhalb dieses Kapitel ein paar der technischen Eigenschaften oberflächlich behandelt. Da der Leser also ein grobes Verständnis der technischen Vorteile hat, wollen wir uns jetzt näher mit den entwicklungsbezogenen Vorteilen eines Meta-Frameworks im Kontext unserer Problemstellung beschäftigen. Diese Vorteile ergeben sich teilweise aus den technischen Eigenschaften. Am Ende dieses Unterkapitels, wir der Leser somit in der Lage sein, die detaillierte Behandlung der technischen Aspekte in Kapitel 4, besser einzuordnen.

Effizienz und Skalierbarkeit

Meta-Frameworks bieten eine Abstraktionsschicht und einheitliche Strukturen, die es Entwicklern ermöglichen, effizient und skalierbar Webanwendungen zu erstellen. Ein wesentlicher Vorteil ist die Fähigkeit von Meta-Frameworks, eine umfassende Entwicklungsumgebung bereitzustellen, die sowohl Frontend- als auch Backend-Funktionalitäten abdeckt. In traditionellen Webentwicklungsansätzen wird der Backend-Code oft in Sprachen wie Java (beispielsweise Spring Boot [5]) oder Python (beispielsweise Django[6]) geschrieben. Meta-Frameworks hingegen, wie SvelteKit, setzen auf JavaScript oder TypeScript als einheitliche Sprache für die gesamte Anwendung, sowohl für das Frontend als auch für das Backend.

Einheitliche Entwicklungsumgebung

Die Verwendung von JavaScript auf der Serverseite, typischerweise in einer Node.js-Umgebung, ermöglicht es Entwicklern, eine konsistente Codebasis zu pflegen und nahtlos zwischen Frontend- und Backend-Komponenten zu wechseln. Dieser Ansatz fördert die Einheitlichkeit und Kon-

[5] `https://spring.io/projects/spring-boot`
[6] `https://www.djangoproject.com`

sistenz der Anwendung und erleichtert sowohl die Entwicklung als auch die Wartung [Fil23]. Ein zentraler Aspekt, den der Leser unbedingt verstehen sollte, ist die Fähigkeit von Meta-Frameworks, Backend-Code auszuführen, der Zugriff auf Backend-Funktionalitäten hat. Diese Funktionalitäten sind essenziell für die Erfüllung der Anforderungen der Web-Applikation.

Backend-Funktionalitäten

Im weiteren Verlauf dieser Arbeit werden wir detailliert auf die spezifischen Backend-Funktionalitäten eingehen, die für die Realisierung der gestellten Anforderungen benötigt werden. Bereits an dieser Stelle sei jedoch hervorgehoben, dass die Web-Applikation mehrere Backend-Komponenten erfordert. Dazu gehört ein LDAP-Client für die Interaktion mit dem LDAP-Server, der für die Authentifizierung und Autorisierung der Benutzer zuständig ist. Zudem werden Umgebungsvariablen (Environment variables) verwendet, um sensible Konfigurationsdaten sicher zu speichern und zu verwalten. Ein Datenbank-Client ermöglicht den Zugriff auf eine SQLite-Datenbank[7] und stellt die persistente Speicherung von Daten sicher. Diese Backend-Funktionalitäten erfordern eine serverseitige Ausführungsumgebung, die über die Möglichkeiten von JavaScript in einer reinen Browser-Umgebung hinausgeht.

Bedeutung der Node.js-Umgebung

Hier kommt die Bedeutung einer Node.js-Umgebung zum Tragen. Node.js[8] ist eine serverseitige JavaScript-Laufzeitumgebung, die es ermöglicht, JavaScript-Code außerhalb des Browsers auszuführen und auf Systemressourcen und APIs zuzugreifen. Durch die Verwendung von Node.js können die benötigten Backend-Funktionalitäten, wie der LDAP-Client , die Verwendung von Umgebungsvariablen und der Datenbank-Client, effizient implementiert und in die Web-Applikation integriert werden.

Konsistenz und Produktivität

Meta-Frameworks wie SvelteKit bieten durch die Verwendung von JavaScript oder TypeScript als einheitliche Sprache für das Frontend und Backend eine konsistente und nahtlose Entwicklungserfahrung. Diese einheitliche Entwicklungsumgebung hat einen positiven Einfluss auf die Entwicklerproduktivität. Durch die Verwendung einer einzigen Sprache für die gesamte Anwendung entfällt der Kontextwechsel zwischen verschiedenen Programmiersprachen und -paradigmen. Entwickler können sich auf eine einzige Technologie konzentrieren und müssen sich nicht ständig an unterschiedliche Syntax, Konventionen und Best Practices anpassen. Diese Konsistenz reduziert die kognitive Belastung und ermöglicht es Entwicklern, schneller und effizienter zu arbeiten [Win23].

Unterstützung durch Dokumentation und Community

Darüber hinaus bieten Meta-Frameworks oft eine umfangreiche Dokumentation, vorgefertigte Komponenten und bewährte Muster, die den Entwicklungsprozess weiter rationalisieren. Entwickler können auf eine solide Grundlage aufbauen und müssen das Rad nicht jedes Mal

[7] `https://sqlite.org`
[8] `https://nodejs.org/en`

neu erfinden. Die Wiederverwendung von Code und die Nutzung von Best Practices tragen dazu bei, die Entwicklungszeit zu verkürzen und die Qualität der Anwendung zu verbessern [Cai23].

Im Kontext dieser Arbeit, die von einem einzelnen Entwickler mit begrenzter Weberfahrung durchgeführt wurde, spielte die Entwicklerproduktivität eine entscheidende Rolle.

Scaffolding und Automatisierung

Ein weiterer produktivitätsfördernder Aspekt sind die Werkzeuge zur Scaffolding, die von vielen Meta-Frameworks bereitgestellt werden. Scaffolding bezieht sich auf die automatische Generierung von Codestrukturen, Konfigurationen und Boilerplate-Code. Durch die Verwendung dieser Werkzeuge können Entwickler schnell ein Grundgerüst für ihre Anwendung erstellen, ohne sich mit repetitiven und zeitaufwendigen Aufgaben aufhalten zu müssen. Zuletzt sei die umfassende Dokumentation und die aktive Community erwähnt, die viele Meta-Frameworks begleiten. Sie tragen ebenfalls zur Entwicklerproduktivität bei, da Entwickler Zugriff auf detaillierte Anleitungen, Tutorials und Beispiele haben , die den Einstieg erleichtern und beste Praktiken vermitteln.

Zusammenfassung

Insgesamt ermöglichen es Meta-Frameworks selbst einem einzelnen Entwickler mit begrenzter Erfahrung, in kürzerer Zeit funktionsreiche und robuste Web-Anwendungen mit Backend-Funktionalität zu erstellen. Die Abstraktion von komplexen Details, die Bereitstellung vorgefertigter Komponenten und die Automatisierung von repetitiven Aufgaben reduzieren den Aufwand und die Komplexität der Entwicklung. Dies macht Meta-Frameworks zu einer wertvollen Wahl für die effiziente und skalierbare Webentwicklung.

Ob ein Tool sich tatsächlich für etwas eignet, kann jedoch immer erst festgestellt werden, wenn es auch zum praktischen Einsatz in einem reellen Szenario kommt. Nachdem wir in den Kapiteln 2.3.1 und 2.3.2 , wie auch in diesem Kapitel, über die allgemeinen Vorteile von Component- und Meta-Frameworks gesprochen haben, wollen wir uns im kommenden Kapitel mit zwei konkreten Ausprägungen solcher Frameworks beschäftigen.

Angesichts der Vielzahl verfügbarer Component- und Meta-Frameworks ist eine sorgfältige Auswahl für den Erfolg eines Projekts entscheidend. Wir werden daher nicht nur die spezifischen Eigenschaften und Funktionalitäten der ausgewählten Frameworks untersuchen, sondern auch die Gründe für ihre Wahl im Kontext unseres Projekts erläutern. Diese Begründung wird sowohl technische Aspekte als auch projektspezifische Anforderungen berücksichtigen, um ein umfassendes Verständnis für die Entscheidungsfindung zu vermitteln. Durch diese detaillierte Betrachtung und Begründung soll dem Leser ein tieferer Einblick in den Auswahlprozess und die Eignung der gewählten Frameworks für unser spezifisches Vorhaben gegeben werden.

4 Svelte und SvelteKit

In den vorangegangenen Kapiteln haben wir uns eingehend mit den Konzepten von Component-Frameworks und Meta-Frameworks beschäftigt. Wir haben untersucht, wie diese Frameworks die Entwicklung von modernen Webanwendungen erleichtern und welche Vorteile sie bieten. Dabei haben wir auch die enge Beziehung zwischen Component-Frameworks und Meta-Frameworks beleuchtet und gezeigt, wie sie zusammenarbeiten, um eine effiziente und strukturierte Entwicklungsumgebung zu schaffen. In diesem Kapitel wollen wir nun einen Schritt weiter gehen und uns mit zwei konkreten Ausprägungen solcher Frameworks auseinandersetzen: Svelte als Component-Framework und SvelteKit als zugehöriges Meta-Framework. Wir werden untersuchen,was diese Frameworks auszeichnet und wie sie sich gegenüber vergleichbaren Lösungen abheben.

In den folgenden Abschnitten werden wir die Besonderheiten von Svelte und SvelteKit genauer betrachten und ihre Stärken und Unterschiede im Vergleich zu anderen populären Frameworks wie React[1], Vue [2] oder Angular[3] herausarbeiten. Wir werden untersuchen, wie diese Frameworks die Entwicklung von modernen Webanwendungen prägen und welche Vor- und Nachteile sie mit sich bringen.

Durch die detaillierte Analyse von Svelte und SvelteKit werden wir ein tieferes Verständnis dafür erlangen, wie diese spezifischen Frameworks die Prinzipien von Component-Frameworks und Meta-Frameworks umsetzen und welchen Mehrwert sie für Entwickler bieten.

4.1 Svelte

Wir wollen im Folgenden einen kurzen Abriss über Svelte, und die Historie dieses Frameworks, geben. Svelte ist ein modernes Component-Framework, das in den letzten Jahren immer mehr an Bedeutung gewonnen hat [Sta23b]. Es wurde entwickelt, um die Erstellung von interaktiven Benutzeroberflächen zu vereinfachen und die Performanz von Webanwendungen zu verbessern. In dieser Hinsicht unterscheidet es sich nicht von anderen Component-Frameworks. Im Gegensatz zu anderen populären Frameworks wie React oder Vue.js, die auf eine Laufzeitumgebung im Browser setzen, verfolgt Svelte jedoch einen innovativen Ansatz, der auf einem Compiler basiert [Dig21]. Die Geschichte von Svelte begann im Jahr 2016, als Rich Harris, ein Entwickler bei der Firma Vercel [4], mit der Arbeit an diesem neuen Framework begann [Har23b]. Vercel ist bekannt für seine Beiträge zur Webentwicklung und ist auch das Unternehmen hinter dem beliebten Meta-Framework Next.js[5]. Mit Svelte wollte

[1] `https://react.dev/`
[2] `https://vuejs.org/`
[3] `https://angular.dev/`
[4] `https://vercel.com/`
[5] `https://nextjs.org/`

Rich Harris einen neuen Weg einschlagen, um die Komplexität und die Performanceprobleme zu adressieren, die oft mit modernen JavaScript-Frameworks einhergehen. In den frühen Tagen von Svelte lag der Fokus darauf, ein schlankes und effizientes Framework zu schaffen, das sich auf die Kernfunktionalitäten konzentriert [Har23a]. Die erste Version von Svelte (2016 erschienen) wurde von der Community mit Interesse aufgenommen, da es einen erfrischenden Ansatz bot, der sich von den etablierten Frameworks abhob. Mit jeder neuen Version wurden weitere Verbesserungen und Funktionen hinzugefügt, um den wachsenden Anforderungen der Webentwicklung gerecht zu werden. Ein Meilenstein in der Entwicklung von Svelte war dann die Einführung des bereits erwähnten Compilers in Version 3. Der Compiler analysiert den Svelte-Code und generiert hochoptimierten JavaScript-Code, der direkt im Browser ausgeführt werden kann [Bep23]. Dieser Ansatz führt zu kleineren Bündel-Größen und einer verbesserten Performanz, da der generierte Code speziell auf die Anforderungen der Anwendung zugeschnitten ist. Dadurch hebt sich Svelte von anderen Frameworks ab, die oft eine zusätzliche Laufzeitumgebung im Browser benötigen.

Heute steht Svelte als leistungsstarkes und innovatives Component-Framework zur Verfügung, das Entwicklern eine einfache und effiziente Möglichkeit bietet, moderne Webanwendungen zu erstellen. Es hat sich als ernstzunehmende Alternative zu etablierten Frameworks wie React und Vue.js etabliert und gewinnt immer mehr an Popularität, wie auch eine von Stack Overflow[6] unter professionellen Entwicklern durchgeführte Umfrage [7] in Abbildung 4.1 zeigt.

[6] `https://try.stackoverflow.co`

[7] `https://survey.stackoverflow.co/2023/#section-admired-and-desired-web-frameworks-and-technologies`

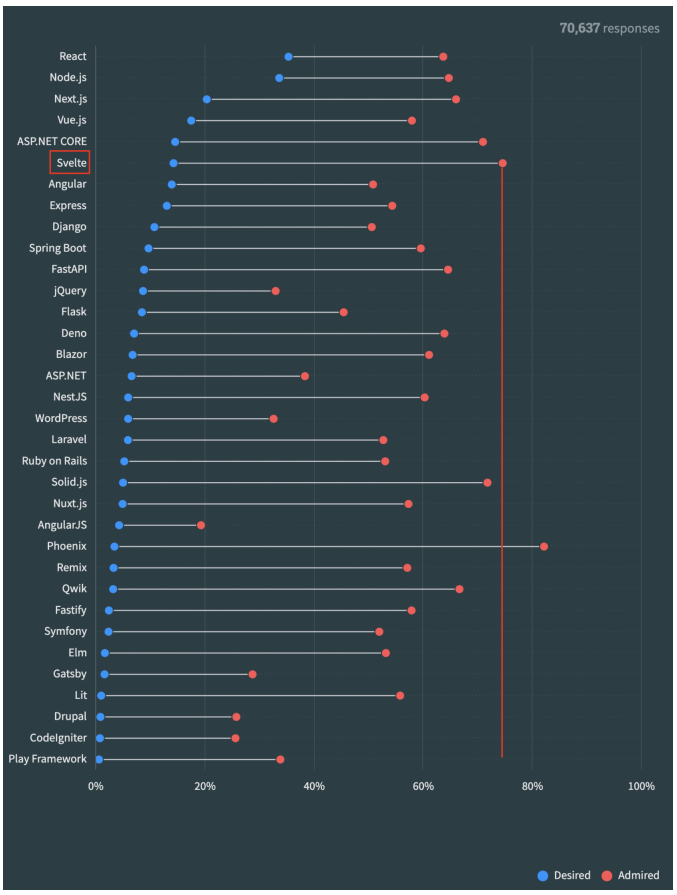

Abbildung 4.1: Von Stack Overflow durchfgeführte Developer Survey zur Beliebtheit von Web-Frameworks [Sta23a].

In den folgenden Abschnitten werden wir uns genauer mit den Besonderheiten von Svelte befassen und untersuchen, wie es sich von anderen Frameworks unterscheidet. Zu diesen Besonderheiten gehören die Syntax, der Compiler, und das State Management, da sich Svelte in diesen Punkten am klarsten von anderen Component-Frameworks unterscheidet.

Es ist wichtig zu beachten, dass sich Svelte kontinuierlich weiterentwickelt und verbessert. Zum Zeitpunkt der Erstellung dieser Arbeit steht die Version Svelte 5 kurz vor ihrem offiziellen Release. Obwohl sich Svelte 5 noch in der "Release Candidate"-Phase[8] befindet, haben wir uns bewusst dafür entschieden, auch die neuen Funktionen und Verbesserungen dieser Version zu berücksichtigen. Der Grund für diese Entscheidung liegt darin, dass wir sicherstellen wollen, dass die Inhalte dieser Arbeit auch nach der Veröffentlichung von Svelte 5 relevant und aktuell bleiben. Da Svelte 5 unmittelbar vor dem offiziellen Release (Status kann hier [9] eingesehen werden) steht und bereits umfangreiche Dokumentation[10] und Ressourcen zur Verfügung stehen, ist es sinnvoll, diese Version als Grundlage für unsere Betrachtungen zu verwenden. Es ist jedoch wichtig zu beachten, dass einige der vorgestellten Funktionen und

[8] `https://svelte.dev/blog/svelte-5-release-candidate`
[9] `https://svelte-5-preview.vercel.app/status`
[10] `https://svelte-5-preview.vercel.app/docs/introduction`

Beispiele möglicherweise noch nicht in der offiziellen Dokumentation[11] von Svelte enthalten sind, da sich diese derzeit noch auf die stabile Version, Svelte 4, bezieht. Stattdessen werden wir uns auf die Svelte 5 beziehen, da diese Version auch für die Entwicklung unserer Web-Anwendung verwendet wurde. Daher werden wir, wo erforderlich, entsprechende Hinweise und Anmerkungen einfügen, um klarzustellen, dass sich bestimmte Funktionen oder Beispiele auf die Version 5 beziehen und möglicherweise noch Änderungen unterliegen.

Ziel der folgenden vier Unterkapitel ist es dem Leser einen Eindruck für die Besonderheiten des Svelte Component-Frameworks zu vermitteln.

4.1.1 Svelte Syntax

Die Syntax hat einen direkten Einfluss auf die Lesbarkeit, Verständlichkeit und Entwicklungsfreundlichkeit des Codes. Svelte, ein modernes Component-Framework, hebt sich in diesem Bereich besonders durch seine zugängliche und native HTML-ähnliche Syntax ab. In diesem Kapitel werden wir kurz die Syntax von Svelte im Vergleich zu anderen gängigen Component-Frameworks analysieren.

Svelte hat eine native HTML-ähnliche Syntax. Es unterscheidet sich grundlegend von Frameworks wie React durch den Verzicht auf JSX [12] (JavaScript XML). Während React-Komponenten in JSX geschrieben werden, was eine Mischung aus JavaScript und XML-ähnlicher Syntax darstellt , bleibt Svelte näher an reinem HTML. Dies reduziert die kognitive Belastung für Entwickler, insbesondere für solche, die bereits mit HTML vertraut sind. Die Svelte Syntax stellt somit eine geringere Abweichung von etablierten Web-Standards dar als die JSX-Syntax es tut. Ein einfaches Beispiel einer Svelte-Komponente zeigt dies deutlich:

Abbildung 4.2: Darstellung der gleichen Komponente in React und in Svelte

Die Verwendung von JSX kann für Entwickler, die HTML gewohnt sind, verwirrend sein, da sie die Syntax lernen müssen, die eine Mischung aus HTML und JavaScript ist. Svelte hingegen verwendet direktes HTML, was den Einstieg erleichtert und die Lesbarkeit des

[11] https://svelte.dev/docs/introduction
[12] https://legacy.reactjs.org/docs/introducing-jsx.html

Codes verbessert. Dies ist angesichts der Tatsache, dass in dieser Arbeit die effiziente Umsetzung von Web-Apps betrachtet wird, ein nicht von der Hand zu weisender Vorteil. Des Weiteren folgt eine Svelte-Komponente einem klaren "Separation of Concerns" Prinzip, indem Funktionalität, Darstellung und Inhalt klar voneinander getrennt werden. Dies wird in Abbildung 4.3 deutlich.

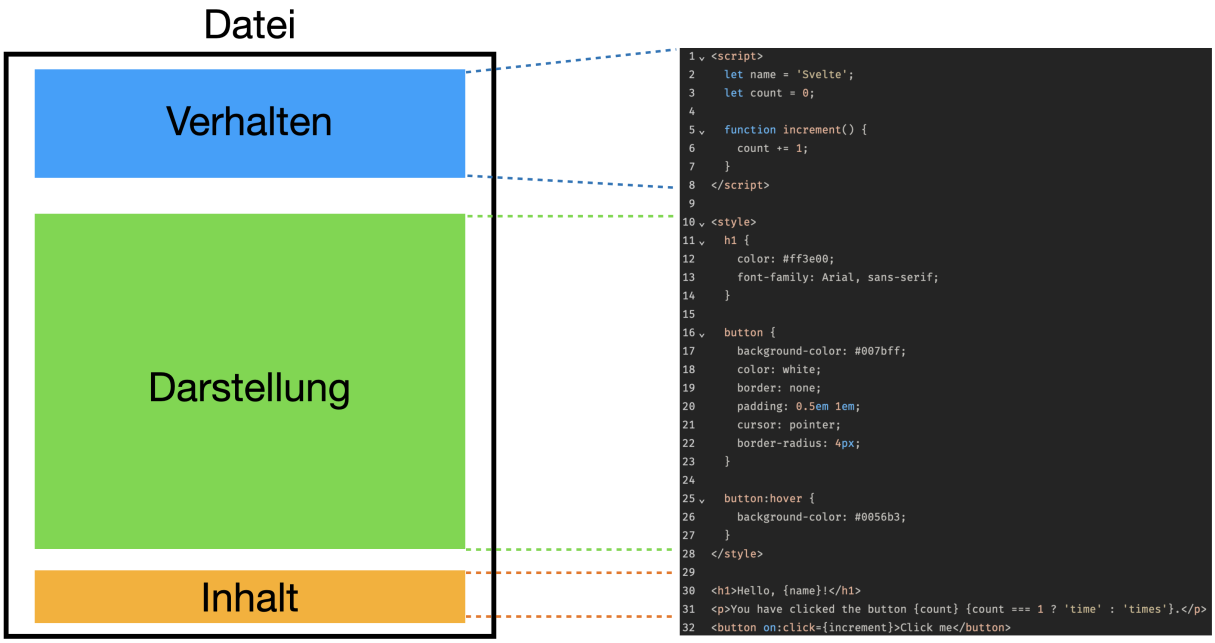

Abbildung 4.3: Darstellung der Struktur einer Svelte Komponente

Der Code einer Komponente ist in Svelte somit sehr organisiert. Es sei zusätzlich angemerkt, dass das Styling der Komponente nur die Komponente selbst als Scope hat. Da somit das Styling der Komponenten nicht innerhalb einer Datei erfolgt, müssen keine komplexen Naming Conventions eingehalten werden, die vonnöten sind, wenn das gesamte CSS eine Applikation in einer einzigen Datei enthalten ist.

Der Inhalt der Komponente, also der HTML-Anteil, kann mit programmatischen Konstrukten unterstützt werden, wodurch das Templating von HTML stark vereinfacht wird, und deklarativer erfolgt. In Abbidlung 4.4 sind zwei in sich geschlossene Beispiele gegeben, welche die Verwendung dieser Konstrukte darstellen.

{#if}-Block

```svelte
<script>
  let isLoggedIn = false;

  function toggleLogin() {
    isLoggedIn = !isLoggedIn;
  }
</script>

<style>
  button {
    margin-top: 1em;
    padding: 0.5em 1em;
    background-color: #007bff;
    color: white;
    border: none;
    cursor: pointer;
    border-radius: 4px;
  }
</style>

{#if isLoggedIn}
  <p>Welcome back, user!</p>
{:else}
  <p>Please log in.</p>
{/if}

<button onclick={toggleLogin}>
  {isLoggedIn ? 'Log out' : 'Log in'}
</button>
```

{#each}-Block

```svelte
<script>
  let items = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    { id: 3, name: 'Item 3' }
  ];
</script>

<style>
  ul {
    list-style: none;
    padding: 0;
  }

  li {
    background: #f4f4f4;
    margin: 0.5em 0;
    padding: 0.5em;
    border-radius: 4px;
  }
</style>

<ul>
  {#each items as item}
    <li>{item.name}</li>
  {/each}
</ul>
```

Abbildung 4.4: Illustration des deklarativen HTML-Templating in Svelte

Wie wir der Abbildung entnehmen, können für das HTML-Templating in Svelte geläufige Programmierkonstrukte wie if-Abfragen und for-each-Schleifen verwendet werden.

Dieses Unterkapitel stellt selbstverständlich in keinster Weise eine exhaustive Behandlung der Syntax von Svelte dar. Hierfür ist die Dokumentation zuständig. Es soll für den Leser jedoch ein Grundlagenverständnis für spätere Beispiele schaffen, als auch die Möglichkeit geben den Code der Web-Applikation einfacher nachvollziehen zu können, sollte er sich entscheiden diesen zu lesen.

4.1.2 Svelte Compiler

Die Verwendung eines Compilers in Svelte ist das Merkmal dieses Component-Frameworks, in welchem es sich am klarsten von anderen Component-Frameworks unterscheidet. Im Gegensatz zu Frameworks wie React und Vue, die auf ein virtuelles DOM (Document Object Model) [13] setzen, verwendet Svelte einen Compiler, der während der Build-Zeit arbeitet. Dieser Ansatz bringt mehrere Vorteile mit sich, insbesondere hinsichtlich Performance und Code-Effizienz [Har19]. Dieser Abschnitt beleuchtet die Funktionsweise des Svelte-Compilers, und die Vorteile der Nutzung eines Compilers.

[13] `https://legacy.reactjs.org/docs/faq-internals.html`

Funktionsweise des Svelte-Compilers

Der Svelte-Compiler transformiert Svelte-Komponenten in effizienten, direkt ausführbaren JavaScript-Code [Bep23]. Dieser Prozess lässt sich in mehrere Schritte unterteilen:

1. **Parsing**: Der Compiler beginnt mit dem Einlesen der Svelte-Komponente und wandelt den Quellcode in einen abstrakten Syntaxbaum (AST) um. Dieser Baum repräsentiert die Struktur des Codes und ermöglicht eine systematische Analyse und Transformation.
2. **Transformation**: Basierend auf dem AST analysiert der Compiler die Komponentenstruktur und wandelt sie in imperativen JavaScript-Code um. Dieser Schritt beinhaltet die Erkennung von reaktiven Variablen, die Implementierung von Zustandsmanagement und die Handhabung von Event-Listeners [14].
3. **Code-Generierung**: Der transformierte Code wird schließlich in JavaScript umgewandelt, das direkt in den Browser geladen werden kann. Dieser Code enthält optimierte DOM-Manipulationslogik, die ohne die Zwischenstufe eines virtuellen DOM auskommt.
4. **CSS-Verarbeitung**: Stile, die innerhalb der Svelte-Komponente definiert sind, werden ebenfalls verarbeitet. Der Compiler generiert spezifische CSS-Klassen, die auf die Komponente beschränkt sind, um Stilkonflikte zu vermeiden.

In Abbildung 4.5 sind sinnbildlich die Artefakte, welche während der Entwicklungszeit beziehungsweise der Build-Zeit entstehen, dargestellt.

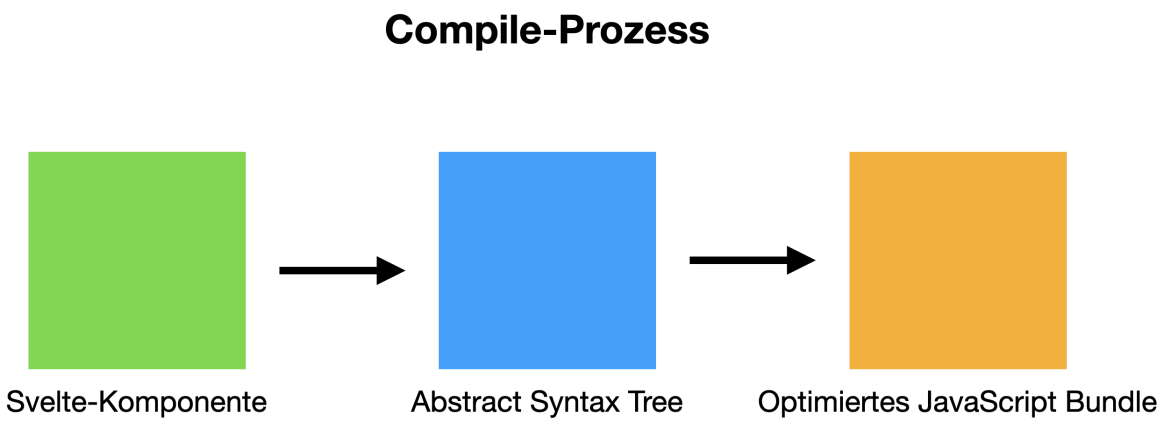

Abbildung 4.5: Illustration der entstehenden Komponenten während des Kompilierens

Doch welche Vorteile sind nun durch die Verwendung eines Compilers gegeben?

Vorteile des Einsatzes eines Compilers

Der Einsatz eines Compilers wie in Svelte bietet mehrere signifikante Vorteile [Bep23]:

[14] `https://www.w3schools.com/js/js_htmldom_eventlistener.asp`

1. Performanz: Durch die Kompilierung zur Build-Zeit wird der Overhead zur Laufzeit minimiert. Der generierte JavaScript-Code führt direkte DOM-Manipulationen durch, was zu schnelleren Updates und einer geringeren Rechenlast führt.
2. Kleinere Bundle-Größe: Der Compiler entfernt unnötigen Code und erzeugt hoch optimierten JavaScript-Code. Dies reduziert die Größe des ausgelieferten Bundles und verbessert die Ladezeiten der Anwendung [Sch21].
3. Statische Analyse: Durch die Analyse zur Build-Zeit können potenzielle Fehler frühzeitig erkannt und gemeldet werden. Dies verbessert die Code-Qualität und reduziert die Anzahl von Laufzeitfehlern.

Fazit

Der Svelte-Compiler stellt eine innovative Lösung zur Erstellung performanter und wartbarer Webanwendungen dar. Durch die Verlagerung der Arbeit auf die Build-Zeit und die Erzeugung von effizientem, imperativem Code bietet Svelte signifikante Vorteile gegenüber traditionellen Frameworks, die auf ein virtuelles DOM setzen, wie der Marktführer React[15] es tut [Win23].

4.1.3 Svelte Reaktivität

Im folgenden wollen wir auf Reactivity (zu deutsch Reaktivität) eingehen. Bevor wir auf das Konzept an sich eingehen, ist es wichtig hervorzuheben, dass die Art und Weise auf welche Reactivity in Svelte 5 umgesetzt wurde, sich erheblich vom im Svelte 4 verwendeten Ansatz entscheidet, womöglich sogar den Hauptunterschied zwischen den Versionen darstellt.

Was ist Reaktivität?

In der Welt der Webentwicklung bezieht sich der Begriff "Reaktivität" auf die Fähigkeit eines Systems, die Benutzeroberfläche (UI) automatisch als Reaktion auf Änderungen im zugrunde liegenden Datenmodell zu aktualisieren [Dig21]. Das reaktive Paradigma ermöglicht es Entwicklern, Beziehungen zwischen dem Datenmodell und der Benutzeroberfläche deklarativ zu definieren. Dadurch wird sichergestellt, dass jede Änderung des Datenzustands automatisch auf die entsprechenden UI-Elemente übertragen wird. Dieses Paradigma steht im Gegensatz zur imperativen Programmierung, bei der Entwickler den Zustand und die Synchronisierung der Benutzeroberfläche explizit verwalten müssen , was oft zu komplexem und fehleranfälligem Code führt. Das Konzept der Reaktivität basiert auf den Grundlagen von Beobachtern (Observers) und Beobachtbaren (Observables). Observables repräsentieren Datenquellen, während Observers Entitäten sind, die auf Änderungen in diesen Datenquellen reagieren. Wenn sich der Zustand eines Observables ändert, werden die Observers benachrichtigt und handeln entsprechend, um sicherzustellen, dass die Benutzeroberfläche mit dem Datenzustand konsistent bleibt [Dig21].

[15] `https://react.dev/`

Die Notwendigkeit der Reaktivität

Die Notwendigkeit der Reaktivität ergibt sich aus der dynamischen Natur moderner Webanwendungen. Benutzer erwarten Echtzeit-Updates und nahtlose Interaktionen, was eine reaktionsschnelle Benutzeroberfläche erfordert, die sich sofort an Datenänderungen anpasst. Reaktivität vereinfacht den Entwicklungsprozess, indem sie die komplexen Details der Zustandsverwaltung und der UI-Synchronisierung abstrahiert. Diese Abstraktion führt zu mehreren Vorteilen [Con23]:

1. Deklarative Syntax: Reaktivität ermöglicht es Entwicklern, zu beschreiben, wie die Benutzeroberfläche für einen bestimmten Zustand aussehen soll, ohne im Detail zu erläutern, wie die Benutzeroberfläche aktualisiert werden soll, wenn sich der Zustand ändert.
2. Verbesserte Leistung: Reaktive Systeme können Updates optimieren, indem sie nur die betroffenen Teile der Benutzeroberfläche neu berechnen und neu rendern, was zu einer besseren Leistung führt, insbesondere bei komplexen Anwendungen.
3. Verbesserte Benutzererfahrung: Eine reaktive Benutzeroberfläche bietet eine reibungslose und intuitive Benutzererfahrung, indem sie sicherstellt, dass die Oberfläche immer mit den zugrunde liegenden Daten synchronisiert ist.

Svelte Ansatz zur Reaktivität

Das Reaktivitätsmodell von Svelte hebt sich aufgrund seines im vorigen Unterkapitel dargestellten Compiler basierten Ansatzes von anderen populären Frontend-Frameworks ab. Im Gegensatz zu Frameworks wie React und Vue, die sich auf ein virtuelles DOM verlassen, um Updates zu verwalten, kompiliert Svelte Komponenten zu hocheffizientem JavaScript-Code, der das DOM direkt manipuliert. Dieser Ansatz eliminiert den Laufzeit-Overhead, der mit dem Vergleich virtueller DOMs verbunden ist, was zu einer schnelleren Leistung und kleineren Bundle-Größen führt [Dig21]. Hauptunterschiede:

- **React**: Verwendet ein virtuelles DOM und Hooks für die Zustandsverwaltung. Die Reaktivität von React wird durch die Hooks useState und useEffect verwaltet, die aufgrund der Notwendigkeit der Abstimmung des virtuellen DOMs eine gewisse Komplexität und einen Leistungs-Overhead mit sich bringen können.
- **Vue**: Verwendet ein reaktives Datenverbindungssystem mit einem virtuellen DOM. Die Reaktivität von Vue wird durch reaktive Eigenschaften und Watcher verwaltet, die einfacher sein können als die Hooks von React, aber dennoch virtuelle DOM-Operationen beinhalten.
- **Svelte**: Kompiliert Komponenten zu minimalem JavaScript-Code, der das DOM direkt aktualisiert.

Svelte 5 führt signifikante Änderungen in seinem Reaktivitätsmodell ein und geht von den in Svelte 4 verwendeten reaktiven Variablen zu einem neuen System über, das auf "Runen"[16] und JavaScript Signals[17] basiert. Runen sind hierbei das für Svelte 5 proprietäre

[16] `https://svelte.dev/blog/runes`
[17] `https://millermedeiros.github.io/js-signals/docs/symbols/Signal.html`

Konzept, während Signals ein JavaScript spezifisches Konzept ist, welches auch in anderen Component-Frameworks Anwendung findet [Con23]. Diese Verschiebung zielt darauf ab, einige der Einschränkungen und Komplexitäten des vorherigen Modells zu beseitigen und gleichzeitig die allgemeine Entwicklererfahrung zu verbessern. Detailliert auf den Unterschied zwischen dem Reactvity Model zwischen Svelte 4 und Svelte 5 einzugehen liegt jedoch außerhalb des Fokus dieser Arbeit. Es sei nur gesagt, dass der Grundsatz der Reaktivität im Web immer ist, so präzise wie möglich zu sein. Sobald sich der Zustand einer Variablen ändert, sollen im besten Fall nur exakt die Variablen geändert werden die von der geänderten Variablen abhängen. Diese feingranulare Reaktivität ist was Svelte 5 im Vergleich zu Svelte 4 in diesem Kontext abhebt.

Somit hat der Leser einen oberflächlichen Eindruck dafür bekommen, was Svelte als Component-Framework ausmacht. Da wir nun ein grundlegendes Verständnis von Svelte haben, wenden wir uns im Folgenden dem zugehörigen Meta-Framework SvelteKit zu [Con23].

4.2 SvelteKit

Wie bereits besprochen, handelt es sich bei SvelteKit um das zu Svelte zugehörige Meta-Framework. Im Unterkapitel 2.3.2 haben wir bereits ein einige Funktionalitäten eines Meta-Frameworks betrachtet. Darunter Routing, API-Integration und Deployment. Wie in 4.1, wollen wir jetzt auf einige Funktionalitäten spezifischer eingehen, um dem Leser ein Eindruck von SvelteKit zu vermitteln. Wir werden spezifischer die Themen Routing, Rendering und Data Loading eingehen, das diese Konzepte bei der Entwicklung der in Kapitel 5 behandelten Web-Applikation eine tragende Rolle spielen.

4.2.1 Routing

In Web-Applikationen bezieht sich der Begriff “Routing” auf den Prozess der Zuordnung von URLs zu spezifischen Komponenten oder Seiten innerhalb der Anwendung, wodurch der Applikation Struktur verliehen wird. Routen definieren also die Navigationsstruktur einer Web-Applikation und ermöglichen es Benutzern, durch verschiedene Ansichten und Funktionen zu navigieren, ohne die Seite neu laden zu müssen. Das Routing-System interpretiert die URL und rendert die entsprechenden Komponenten basierend auf den definierten Routen. Sie ermöglichen eine nahtlose Navigation, verbessern die Benutzerfreundlichkeit und ermöglichen eine effiziente Verwaltung des Anwendungszustands. Durch die Verwendung von aussagekräftigen URLs können Routen auch die Lesbarkeit und Suchmaschinenoptimierung (SEO) verbessern, indem sie den Inhalt der Anwendung strukturieren und beschreiben [Kys20]. In SvelteKit, einem Meta-Framework für die Entwicklung von Svelte-basierten Anwendungen, wird das Routing-Konzept durch ein verzeichnisbasierten Ansatz[18] umgesetzt. Dieser Ansatz, auch bekannt als “directory-based routing”, ermöglicht eine intuitive und deklarative Definition von Routen basierend auf der Verzeichnisstruktur der Anwendung. Bei dateibasiertem Routing in SvelteKit entsprechen die Verzeichnisse und Dateien im src/routes-Verzeichnis der Anwendung direkt den URLs der Routen. Zum Beispiel würde eine Datei unter src/routes/about.svelte automatisch der Route /about zugeordnet werden. Die Abbildung 4.6 verdeutlicht dieses Prinzip.

[18] `https://kit.svelte.dev/docs/routing`

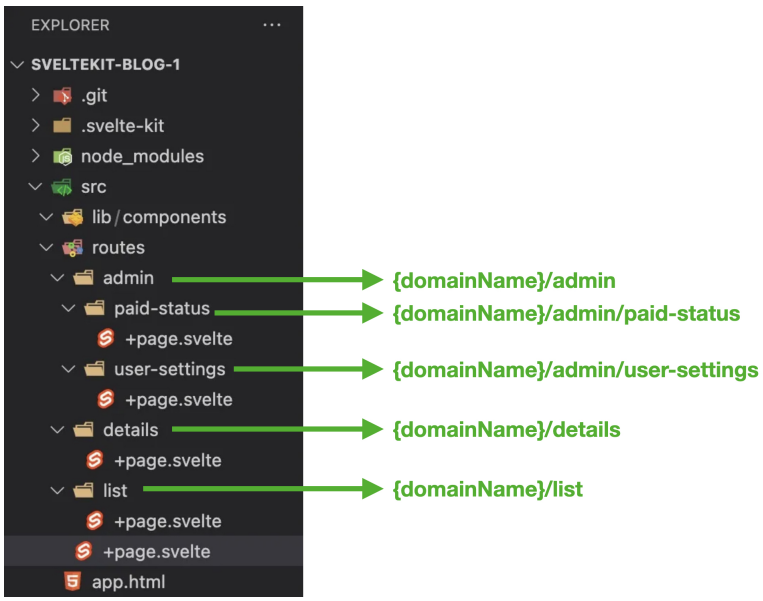

Abbildung 4.6: SvelteKit Routing Beispiel - Quelle[19]

Der domainName stellt hierbei natürlich die Domain dar, mit welcher die Applikation schlussendlich ausgebracht wird. Durch die Verwendung von Parametern und dynamischen Segmenten in den Dateinamen können auch variable Teile einer URL erfasst und an die entsprechende Komponente übergeben werden. Der Inhalt der geladenen Route ist also von dem übergebenen Parameter abhängig, und somit dynamisch. Man stelle sich beispielweise vor, dass eine Route in Abhängigkeit einer übergebenen ID, die Profilseite eines Users lädt. Hier wäre es natürlich von Vorteil, wenn der eingeloggte User auf Basis seiner ID auch den für Ihn relevanten Inhalt auf der Seite angezeigt bekommen kriegt. Man nennt diese Funktionalität “Dynamic Routing”.

Das Routing in SvelteKit ist sehr facettenreich und bietet noch weitere Funktionalitäten, die hier aber nur kurz Erwähnung finden soll. Es können beispielsweise Layoutkomponenten verwendet werden, um gemeinsame Elemente wie Navigationsleisten oder Fußzeilen auf mehreren Routen wiederzuverwenden. Zusätzlich unterstützt SvelteKit das Code-Splitting auf Routenebene, um die Leistung zu optimieren, indem nur der Code geladen wird, der für die aktuelle Route benötigt wird. Insgesamt bietet das dateibasierte Routing in SvelteKit einen eleganten und effizienten Ansatz für die Definition und Verwaltung von Routen in Web-Applikationen. Es verbessert die Entwicklererfahrung, fördert eine saubere Codeorganisation und ermöglicht eine nahtlose Navigation für Benutzer.

Wir halten fest: Der verzeichnisbasierte Ansatz bietet mehrere Vorteile. Erstens ermöglicht er eine intuitive und übersichtliche Organisation der Anwendungskomponenten, da die Verzeichnisstruktur die Navigationsstruktur der Anwendung widerspiegelt. Entwickler können auf einen Blick erkennen, welche Komponenten für welche Routen verantwortlich sind, was die Wartbarkeit und Skalierbarkeit der Anwendung verbessert. Zweitens erleichtert das dateibasierte Routing die Definition und Verwaltung von Routen erheblich. Anstatt Routen manuell in einer zentralen Konfigurationsdatei zu definieren, werden sie automatisch basierend auf der Verzeichnisstruktur generiert. Dies reduziert den Konfigurationsaufwand und

minimiert die Wahrscheinlichkeit von Fehlern bei der Routendefinition. Durch die Kombination von dateibasiertem Routing mit den leistungsstarken Funktionen von SvelteKit können Entwickler skalierbare und leicht wartbare Web-Applikationen erstellen. Selbstverständlich müssen die innerhalb der Routen enthaltenen Svelte Komponenten auch mit Daten befüllt werden können. Deshalb werden wir im Folgenden darauf eingehen, wie das Laden von Daten in SvelteKit vonstattengeht.

4.2.2 Laden von Daten

In der heutigen Zeit der dynamischen Webanwendungen spielt das effiziente Laden von Daten eine entscheidende Rolle für die Benutzerfreundlichkeit und Performanz. Daten-Laden, auch bekannt als Data Loading, bezieht sich auf den Prozess des Abrufs von Daten aus verschiedenen Quellen, wie APIs oder Datenbanken, und deren Bereitstellung für die Verwendung innerhalb einer Webanwendung. Dieser Vorgang ist besonders relevant für Single-Page-Anwendungen [20] (SPAs) und Frameworks wie SvelteKit, die auf die Erstellung reaktiver und interaktiver Benutzererfahrungen abzielen.

Die Notwendigkeit des Daten-Ladens ergibt sich aus der Natur dynamischer Webanwendungen. Im Gegensatz zu statischen Webseiten, bei denen alle Inhalte bereits beim ersten Laden vollständig zur Verfügung stehen, erfordern dynamische Anwendungen die Fähigkeit, Daten nach Bedarf zu laden und die Benutzeroberfläche entsprechend zu aktualisieren. Dieses Konzept ermöglicht es, große Datenmengen in kleinere, verwaltbare Teile aufzuteilen und nur die benötigten Informationen zu laden, um die Ladezeiten zu optimieren und die Benutzerfreundlichkeit zu verbessern. Darüber hinaus ermöglicht das Daten-Laden die Integration von Echtzeit-Updates und interaktiven Funktionen. Durch den Abruf aktueller Daten können Anwendungen Benutzern stets die neuesten Informationen präsentieren und auf Benutzerinteraktionen reagieren, ohne dass eine vollständige Seitenaktualisierung erforderlich ist. Diese Fähigkeit trägt dazu bei, ein nahtloses und reaktionsschnelles Benutzererlebnis zu schaffen, das den Erwartungen moderner Webanwendungen entspricht.

Die soeben genannten Punkte treffen jedoch auf alle Web-Anwendungen und Meta-Frameworks zu. Wodurch hebt sich SvelteKit also ab? Jede Route in SvelteKit hat die Möglichkeit eine spezifisch ihr zugehörige +page.ts oder +page.server.ts Datei zu definieren.

[20] `https://developer.mozilla.org/en-US/docs/Glossary/SPA`

Abbildung 4.7: Darstellung der +page.ts und +page.server.ts Datei

Innerhalb dieser Datei wird dann die 'load'-Funktion definiert. Diese Funktion ist der grundlegende Mechanismus, der es einer Route ermöglicht, Daten zu beziehen. Alle in load Funktion geladenen Daten, können innerhalb der zugehörigen +page.svelte Datei der Route verwendet werden, und auf Basis dessen, der Inhalt der Route gerendert werden. Doch worin liegt nun der Unterschied zwischen einer +page.ts und einer +page.server.ts Datei? Hier kommt eine kritische Funktionalität von SvelteKit zu tragen.

Load-Funktion in +page.ts Dateien

In einer +page.ts Datei wird die load-Funktion clientseitig ausgeführt. Das bedeutet, dass die Funktion nach der Auslieferung der Seite an den Browser des Benutzers ausgeführt wird. Hierbei handelt es sich um eine typische clientseitige Fetch-Operation, bei der Daten über APIs oder andere öffentliche Endpunkte abgerufen werden. Ein Beispiel für eine solche load-Funktion ist in 4.1 aufgeführt.

```
1 // src/routes/example/+page.ts
2 export async function load({ fetch }) {
3   const res = await fetch('/api/data');
4   if (!res.ok) {
5     throw new Error('Failed to fetch data');
6   }
7   const data = await res.json();
8   return { props: { data } };
9 }
```

Listing 4.1: Exemplarische +page.ts load Funktion

Diese clientseitige load-Funktion hat keinen Zugriff auf serverseitige Ressourcen wie Datenbanken oder private APIs, die normalerweise nur vom Backend zugänglich sind. Sie ist ideal für Daten, die über öffentliche APIs abgerufen werden und keinen besonderen Schutz erfordern.

Load-Funktion in +page.server.ts Dateien

Im Gegensatz dazu wird die load-Funktion in einer +page.server.ts Datei auf dem Server ausgeführt. Dies bietet erhebliche Vorteile, da diese Funktion direkten Zugriff auf serverseitige Ressourcen wie Datenbanken, geschützte APIs und andere Backend-Dienste hat. Ein Beispiel für eine serverseitige load-Funktion ist in 4.2 aufgezeigt.

```
    // src/routes/example/+page.server.ts
import { db } from '$lib/database';

export async function load({ params }) {
  const data = await db.getData(params.id);
  if (!data) {
    return { status: 404, error: new Error('Data not found') };
  }
  return { props: { data } };
}
```

Listing 4.2: Exemplarische +page.server.ts load Funktion mit Zugriff auf Datenbank Klienten

Wie man sehen kann, greifen wir innerhalb dieser load Funktion auf einen Datenbankklienten. Dabei handelt es sich um eine Funktionalität, welche in der Regel Server-seitigem Code vorbehalten ist, um sensible Daten zu schützen. Die load Funktion wird in +page.server.ts also serverseitig ausgeführt, bevor die Route an der Client gesendet wird. Dadurch können sensible Daten sicher abgerufen und verarbeitet werden, ohne dass sie dem Benutzer direkt zugänglich gemacht werden. Diese serverseitige Ausführung ermöglicht auch eine verbesserte Performance, da die Daten bereits geladen sind, wenn die Seite den Benutzer erreicht.

Unterschiede und Vorteile

Der entscheidende Unterschied zwischen +page.ts und +page.server.ts liegt also in der Ausführungsumgebung der load-Funktion:

- +page.ts: Clientseitige Ausführung. Daten werden nach dem Laden der Seite im Browser abgerufen. Kein Zugriff auf Backend-Ressourcen. Geeignet für öffentliche API-Aufrufe.
- +page.server.ts: Serverseitige Ausführung. Daten werden auf dem Server abgerufen und die Seite wird bereits mit den notwendigen Daten ausgeliefert. Direkter Zugriff auf geschützte Backend-Ressourcen wie Datenbanken und private APIs.

Die Möglichkeit, load-Funktionen sowohl clientseitig als auch serverseitig zu definieren, macht SvelteKit besonders flexibel und leistungsfähig. Entwickler können genau entscheiden, wo und wie ihre Daten geladen werden, basierend auf den spezifischen Anforderungen ihrer Anwendung. Diese Flexibilität, kombiniert mit der Sicherheit und Performance-Vorteilen der serverseitigen Ausführung, hebt SvelteKit von anderen Frameworks ab und bietet eine robuste Lösung für moderne Webentwicklungsanforderungen.

SvelteKit bietet noch viele weitere Funktionalitäten, wie das Laden von Daten in einer “+layout.ts” bzw. “ +layout.server.ts” Datei, deren Daten dann allen Routen, die sich innerhalb

des Verzeichnisses der Route mit der Layout-Datei befinden, zur Verfügung stehen. Des Weiteren ist die Definition einer "+server.ts" Datei möglich, in welcher dedizierte API-Endpoints definiert werden können. Da die Behandlung dieser Funktionalität jedoch nicht in erheblicher Weise zum Verständnis dieser Arbeit beiträgt, wird diese im vorliegenden Rahmen nicht vorgenommen. Es sei dennoch auf die Dokumentation verwiesen[21] [22]. Insgesamt ist das Laden von Daten in SvelteKit sehr flexibel, und die nahezu mühelose Integration von client- und serverseitiger Funktionalität für das Laden von Daten ist eine Eigenschaft, welche SvelteKit zum Zeitpunkt des Schreibens dieser Arbeit klar von anderen Meta-Frameworks abgrenzt und für unsere im folgenden Kapitel behandelte Web-Applikation eine erhebliche Rolle spielt.

4.2.3 Rendering

Der letzte Aspekt, indem sich SvelteKit klar von anderen Meta-Frameworks abgrenzt, ist die Flexibilität des Rendering-Prozesses, der für das Rendern des HTML verwendet wird.

Was ist Rendering im Allgemeinen im Kontext einer Web-App?

Rendering in einer Webanwendung bezieht sich auf den Prozess, bei dem HTML, CSS und JavaScript verwendet werden, um eine visuelle Darstellung von Inhalten und Daten im Browser zu erzeugen. Sinnbildlich kann man sich diesen Prozess als die Umwandlung von deklarativem Code, in den konkreten HTML Code, welches vom Browser interpretierbar und darstellbar ist, vorstellen. Dies kann auf verschiedene Arten erfolgen, abhängig davon, wann und wo der HTML-Code generiert wird. Die zwei Hauptansätze sind Client-Side Rendering (CSR) und Server-Side Rendering (SSR).

Client-Side Rendering

Beim Client-Side Rendering wird der Großteil der Verarbeitung auf dem Client, also im Browser des Benutzers, durchgeführt. Die initiale HTML-Seite, die vom Server gesendet wird, enthält meist wenig oder gar keinen Inhalt, sondern lediglich Referenzen zu JavaScript-Dateien. Diese JavaScript-Dateien werden dann im Browser ausgeführt, um den vollständigen Inhalt der Seite zu laden und zu rendern.

Vorteile von CSR:

- Interaktivität: CSR ermöglicht eine reiche Benutzerinteraktion, da der Browser dynamisch Inhalte aktualisieren kann, ohne die gesamte Seite neu zu laden.
- Schnelle Subsequenzielle Updates: Einmal geladene Anwendungen können schnellere Interaktionen und Datenaktualisierungen ermöglichen, da nur spezifische Teile der Seite neu gerendert werden.

Nachteile von CSR:

[21] `https://kit.svelte.dev/docs/routing#layout`
[22] `https://kit.svelte.dev/docs/routing#server`

- Initiale Ladezeit: Die initiale Ladezeit kann länger sein, da der Browser zuerst die JavaScript-Dateien laden und ausführen muss, bevor er den Inhalt anzeigen kann.
- SEO: CSR kann Nachteile bei der Suchmaschinenoptimierung (SEO) haben, da Suchmaschinen -Bots möglicherweise Schwierigkeiten haben, JavaScript-Inhalte zu indexieren.

Beim CSR wird also der Großteil der Arbeit des Render-Prozesses auf der Client-Maschine ausgeführt.

Server-Side Rendering

Beim Server-Side Rendering wird der HTML-Code auf dem Server generiert und dann als vollständige Seite an den Client gesendet. Der Browser zeigt den empfangenen HTML-Inhalt direkt an, was zu einer schnelleren initialen Ladezeit führt. SSR kann durch SvelteKit nativ unterstützt werden, indem die Daten auf dem Server geladen und verarbeitet werden, bevor die Seite an den Client gesendet wird.

Vorteile von SSR:

- Schnellere initiale Ladezeit: Da der vollständige HTML-Inhalt bereits auf dem Server generiert wurde, kann der Benutzer den Inhalt schneller sehen.
- SEO: Suchmaschinen können den Inhalt einfacher indexieren, da er bereits im HTML-Dokument vorhanden ist.

Nachteile von SSR:

- Serverlast: Der Server muss für jede Anfrage die Seite neu rendern, was zu einer höheren Serverlast führen kann.
- Interaktivität: SSR erfordert zusätzliche Schritte, um eine reiche Benutzerinteraktion wie bei CSR zu ermöglichen,da die Seite zunächst statisch geladen wird.

In vielen modernen Anwendungen ist eine Mischung aus CSR und SSR sinnvoll. Diese hybride Methode nutzt die Vorteile beider Ansätze, indem sie initiale Inhalte serverseitig rendert, um schnelle Ladezeiten und bessere SEO zu gewährleisten , und dann clientseitige Rendering-Techniken verwendet, um die Interaktivität zu erhöhen und dynamische Inhalte nachzuladen.

Stärke von SvelteKit im Rendering

SvelteKit ist besonders stark im Bereich des Renderings aufgrund seiner Flexibilität und Leistungsfähigkeit. Es bietet eine nahtlose Integration von SSR und CSR und ermöglicht Entwicklern, die besten Ansätze basierend auf den spezifischen Anforderungen ihrer Anwendung zu wählen.

- Einfachheit und Effizienz: SvelteKit ermöglicht es, SSR und CSR ohne großen Aufwand zu implementieren. Die APIs sind intuitiv und unterstützen Entwickler bei der Erstellung effizienter Anwendungen.

- Kompilierungsansatz: Durch die Kompilierung von Svelte-Komponenten in hocheffizienten JavaScript-Code wird die Performance weiter optimiert. Dieser Code kann direkt im Browser ausgeführt werden, was die Ladezeiten reduziert und die Interaktivität verbessert.
- Hydratation: SvelteKit unterstützt Hydratation [23] , bei der serverseitig gerenderte HTML-Seiten mit interaktiven clientseitigen JavaScript-Komponenten angereichert werden. Dies kombiniert die Vorteile von SSR (schnelle initiale Ladezeiten) und CSR (reichhaltige Interaktivität).

SvelteKit zeichnet sich durch seine Fähigkeit aus, die Vorteile von Client Side Rendering und Server Side Rendering nahtlos zu kombinieren und so die Nachteile beider Ansätze zu minimieren. Während CSR eine hohe Interaktivität und Dynamik ermöglicht, kann es zu längeren initialen Ladezeiten führen. SSR hingegen liefert schnelle erste Seitenladezeiten und verbessert die Suchmaschinenoptimierung, kann aber die Interaktivität beeinträchtigen. SvelteKit hebt sich von anderen Frameworks ab, indem es Entwicklern die Flexibilität bietet, die am besten geeignete Rendering-Methode für ihre spezifischen Anforderungen zu wählen. Durch die intuitive API und den effizienten Kompilierungsprozess von Svelte-Komponenten können Entwickler mühelos zwischen CSR und SSR wechseln oder beide Ansätze kombinieren.

Damit beenden wir unsere Einführung in SvelteKit. Genau wie für das Unterkapitel 4.1 gilt auch hier, dass es sich keineswegs um eine exhaustive Behandlung der Funktionalität von SvelteKit handelt. SvelteKit ist ein enorm facettenreiches Meta-Framework, mit teils hochgradig spezifischer und komplexer Funktionalität. Es ging in diesem Kapitel darum, dem Leser einen Eindruck dafür zu verleihen, was die Kernfunktionalität von SvelteKit ist, wie es sich von anderen Meta-Frameworks abgrenzt, und warum es sich für die Entwicklung unserer Web-Applikation eignet. Im Folgenden wollen wir nun auf die konkrete Entwicklung der in Kapitel 3 beschriebenen Applikation eingehen.

[23] `https://dev.to/costamatheus97/hydration-what-is-it-1lgl`

5 Entwicklung des Self-Service-Portals

Nach der Betrachtung der theoretischen Grundlagen, der Analyse der Systemarchitektur und einen Überblick über Svelte und SvelteKit wenden wir uns nun der konkreten Entwicklung des Self-Service-Portals zu. Dieses Kapitel bildet das Herzstück der Arbeit, indem es den Prozess der Umsetzung der Konzepte in eine funktionale Anwendung in Teilen beschreibt.

Wir beginnen in Abschnitt 5.1 mit einer Anforderungsanalyse. Hier werden wir die spezifischen Bedürfnisse der verschiedenen Stakeholder, insbesondere der Universitätsangehörigen und der IT-Abteilung, identifizieren und in konkrete funktionale Anforderungen übersetzen. Diese Analyse bildet das Fundament für alle nachfolgenden Entwicklungsschritte.

Anschließend widmen wir uns in Abschnitt 5.2 dem Architekturdesign des Portals. Basierend auf den ermittelten Anforderungen, werden wir die Architektur unserer Anwendung definieren. Dabei legen wir besonderen Wert auf die Integration der verschiedenen Komponenten wie das Content Management System, die LDAP-Authentifizierung und die Nutzung des Meta-Frameworks SvelteKit.

Der Hauptteil dieses Kapitels, Abschnitt 5.3 , befasst sich mit der konkreten Implementierung der Kernfunktionalitäten. Hier werden wir Schritt für Schritt die Umsetzung wichtiger Features wie die LDAP-Authentifizierung, die Profilbearbeitung und die Interaktion mit dem Headless CMS über die REST API erläutern. Dabei werden wir praktische Codebeispiele präsentieren und die Entscheidungen hinter unseren Implementierungsansätzen diskutieren.

Durch dieses Kapitel ziehen sich als roter Faden die in den vorherigen Abschnitten erarbeiteten Konzepte und Technologien. Wir werden zeigen, wie die theoretischen Grundlagen in die Praxis umgesetzt worden sind und wie das Self-Service-Portal als integratives Element die verschiedenen Komponenten zu einer kohärenten Lösung verbindet.

5.1 Anforderungsanalyse

Die Anforderungsanalyse bildet das Fundament für die Entwicklung eines erfolgreichen Software-Systems. In diesem Kapitel werden wir die Anforderungen an das Self-Service-Portal für die Angehörigen der Informatik Fakultät identifizieren, analysieren und dokumentieren. Dabei ist hervorzuheben, dass die Anforderungen in einem iterativen Prozess durch regelmäßige Gespräche und Abstimmungen mit den Betreuern der Arbeit ermittelt und verfeinert wurden. Dieser kollaborative Ansatz ermöglichte es, ein tiefgreifendes Verständnis der Bedürfnisse und Erwartungen zu entwickeln und sicherzustellen, dass das resultierende System den realen Anforderungen der Fakultät entspricht.

Ziel dieses Kapitels ist es, ein klares Bild der Stakeholder-Bedürfnisse, der gewünschten Funktionalitäten und der relevanten nicht-funktionalen Anforderungen zu zeichnen. Wir be-

ginnen mit der Identifikation der verschiedenen Stakeholder-Gruppen, die mit dem Self-Service-Portal interagieren werden. Diese Analyse bildet die Grundlage für das Verständnis der diversen Nutzerperspektiven und -bedürfnisse.

Es ist anzumerken, dass den nicht-funktionalen Anforderungen im Rahmen dieser Arbeit eine geringere Priorität eingeräumt wurde. Dies ist hauptsächlich auf die überschaubare Anzahl potenzieller Nutzer und den begrenzten zeitlichen Rahmen der Arbeit zurückzuführen. Dennoch werden relevante nicht-funktionale Aspekte, soweit sie für die Qualität und Akzeptanz des Systems von Bedeutung sind, berücksichtigt.

Durch diese strukturierte Anforderungsanalyse schaffen wir eine solide Basis für die nachfolgenden Phasen der Entwicklung und stellen sicher, dass das resultierende Self-Service-Portal den Bedürfnissen und Erwartungen der Informatik Fakultät gerecht wird.

5.1.1 Stakeholder-Identifikation

Bei der Entwicklung des Self-Service-Portals für die Informatik Fakultät ist es von entscheidender Bedeutung, die verschiedenen Stakeholder zu identifizieren und ihre Bedürfnisse und Anforderungen zu verstehen. In diesem Abschnitt werden die Hauptakteursgruppen vorgestellt, die von der Einführung des Portals betroffen sind.

Angehörige der Informatik Fakultät

Die Angehörigen der Informatik Fakultät sind die primären Nutzer des Self-Service-Portals. Bisher mussten sie eine E-Mail an das MCI-Team (Mensch Computer Interaktion) senden, um Änderungen an ihren im Content-Management-System (CMS) hinterlegten Profilen vorzunehmen. Mit dem neuen Portal erhalten sie die Möglichkeit, ihre eigenen Profile direkt zu aktualisieren, ohne auf die Unterstützung des MCI-Teams angewiesen zu sein. Dies ermöglicht eine schnellere und effizientere Verwaltung ihrer Profilinformationen.

Teilnehmer mit der Rolle “Teamass”

Eine spezielle Benutzergruppe innerhalb der Informatik Fakultät sind die Teilnehmer mit der Rolle “Teamass”, sprich die Teamassistenz. Diese Benutzer erhalten erweiterte Berechtigungen im Self-Service-Portal, die es ihnen ermöglichen, mehrere Profile innerhalb ihres zugewiesenen Instituts zu verwalten. Dadurch können sie die Profilinformationen ihrer Teammitglieder zentral pflegen und aktuell halten, ohne dass jedes Teammitglied einzeln Änderungen vornehmen muss. Dies verbessert die Konsistenz und Vollständigkeit der Profilinformationen auf Institutsebene.

Angehörige der Universität

Neben den direkt involvierten Angehörigen der Informatik Fakultät gibt es eine breitere Gruppe von Stakeholdern: die Angehörigen der Universität im Allgemeinen. Sie haben ein Interesse daran, auf die Profile der Informatik Fakultät zuzugreifen, um Informationen über Forschungsgebiete, Kontaktdaten und die akademische Laufbahn der Fakultätsmitglieder zu erhalten. Bisher waren diese Informationen nur auf den Community-Mirrors innerhalb des

Informatik Instituts verfügbar. Durch das Self-Service-Portal erhalten alle Angehörigen der Universität die Möglichkeit, die Profile explorativ zu durchsuchen und einzusehen, allerdings nur mit lesendem Zugriff und ohne die Möglichkeit zur Bearbeitung.

MCI-Team

Obwohl das MCI-Team durch die Einführung des Self-Service-Portals entlastet wird, spielen sie weiterhin eine unterstützende Rolle. Sie sind dafür verantwortlich, neue Profile im CMS anzulegen, da das Portal in seiner aktuellen Version nur das Aktualisieren bestehender Profile ermöglicht. Darüber hinaus können sie bei Fragen, technischen Problemen oder Ausnahmefällen, die über die Funktionalität des Portals hinausgehen, weiterhin als Ansprechpartner fungieren.

Durch die Identifikation dieser Stakeholder-Gruppen und ihrer spezifischen Anforderungen schaffen wir die Grundlage für die Entwicklung eines Self-Service-Portals, das den Bedürfnissen aller Beteiligten gerecht wird. Im nächsten Abschnitt werden wir die funktionalen Anforderungen näher betrachten, die sich aus den Stakeholder-Anforderungen ableiten.

5.1.2 Funktionale Anforderungen

Nachdem wir die Stakeholder identifiziert und ihre Bedürfnisse verstanden haben, konzentrieren wir uns nun auf die funktionalen Anforderungen des Self-Service-Portals. Funktionale Anforderungen beschreiben die Fähigkeiten und Funktionen, die das System erfüllen muss, um den Anforderungen der Stakeholder gerecht zu werden. Sie definieren das gewünschte Verhalten des Portals und dienen als Grundlage für die Entwicklung und Überprüfung des Systems. In diesem Abschnitt werden wir die funktionalen Anforderungen in vier Hauptkategorien unterteilen: Authentifizierung, Profilbearbeitung, erweiterte Berechtigungen für die Rolle “Teamass” und explorativer Zugriff für nicht authentifizierte Benutzer. Jede dieser Kategorien adressiert spezifische Funktionalitäten, die das Self-Service-Portal bieten soll.

Authentifizierung

Die Authentifizierung ist ein kritischer Aspekt des Self-Service-Portals, da sie sicherstellt, dass nur berechtigte Benutzer Zugriff auf die Bearbeitungsfunktionen haben. Um eine nahtlose Integration in die bestehende IT-Infrastruktur der Universität zu gewährleisten und gleichzeitig Flexibilität für zukünftige Änderungen zu bieten, soll das Portal zwei Authentifizierungsmethoden unterstützen.

LDAP-Integration

Die primäre Authentifizierungsmethode soll die Integration mit dem bestehenden LDAP-Server der Universität sein. Der LDAP-Server wird bereits für die Authentifizierung anderer Dienste wie “ILIAS” und “HisinOne” verwendet, so dass die Benutzer ihre bestehenden Anmeldeinformationen auch für das Self-Service-Portal nutzen können. Durch die Verwendung des LDAP-Servers können wir sicherstellen, dass nur authorisierte Benutzer der Universität Zugriff auf das Portal haben und dass die Authentifizierungsinformationen konsistent und

sicher verwaltet werden. Die Integration mit dem LDAP-Server soll nahtlos erfolgen, sodass die Benutzer beim Zugriff auf das Self-Service-Portal automatisch zur LDAP-Anmeldeseite weitergeleitet werden, falls sie noch nicht authentifiziert sind. Nach erfolgreicher Eingabe ihrer LDAP-Anmeldeinformationen sollen die Benutzer dann Zugriff auf die für sie freigegebenen Funktionen des Portals erhalten.

Manuelle Anmeldeinformationen

Da absehbar ist, dass die LDAP-Authentifizierung in Zukunft möglicherweise abgelöst wird, soll das Self-Service-Portal auch die Möglichkeit bieten, manuelle Anmeldeinformationen für Benutzer zu hinterlegen. Diese Funktion soll jedoch nicht direkt über das Portal selbst zugänglich sein, sondern den Administratoren der Web-Applikation vorbehalten bleiben. Die Administratoren sollen in der Lage sein, Benutzernamen und Passwörter für spezifische Benutzer manuell anzulegen und zu verwalten. Dazu soll eine separate Administrationsoberfläche oder ein Backend-System entwickelt werden, das nur für autorisierte Administratoren zugänglich ist. Die manuell angelegten Anmeldeinformationen sollen sicher gespeichert und verschlüsselt werden, um unbefugten Zugriff zu verhindern.

Profilbearbeitung

Die Profilbearbeitung ist eine zentrale Funktionalität des Self-Service-Portals, die es authentifizierten Nutzern ermöglicht, ihre im WordPress Headless CMS hinterlegten Informationen selbstständig zu aktualisieren. Durch die Bereitstellung einer benutzerfreundlichen Oberfläche zur Verwaltung der eigenen Profildaten wird der manuelle Aufwand für das MCI-Team reduziert und die Datenqualität verbessert. Bearbeitbare Profilfelder Authentifizierte Nutzer sollen in der Lage sein, fast alle ihre im CMS gespeicherten Profilinformationen direkt über das Self-Service-Portal zu bearbeiten. Dazu gehören Felder wie:

- Name
- Forschungsgebiet
- Gebäude und Raum an der Universität
- Rolle (Professor, wissenschaftlicher Mitarbeiter, Teamassistenz, etc.)
- Inhaltsfeld für Informationen zur akademischen Laufbahn

Diese Felder sollen in einer übersichtlichen Formularansicht dargestellt werden, die eine einfache und intuitive Bearbeitung ermöglicht. Die Nutzer sollen ihre Änderungen in den entsprechenden Feldern vornehmen und speichern können. Eine Ausnahme bildet die E-Mail-Adresse des Nutzers. Da sich diese Information selten und nur aus triftigen Gründen ändert, soll die Möglichkeit zur Bearbeitung der E-Mail-Adresse nicht direkt im Self-Service-Portal verfügbar sein. Stattdessen müssen Nutzer weiterhin eine E-Mail an das MCI-Team senden, um eine Änderung ihrer E-Mail-Adresse zu beantragen. Dies stellt sicher, dass Änderungen an dieser wichtigen Information kontrolliert und nachvollziehbar durchgeführt werden. Im Self-Service-Portal soll die E-Mail-Adresse des Nutzers dennoch sichtbar sein, jedoch als nicht bearbeitbares Feld dargestellt werden. Ein erklärender Text soll die Nutzer darauf hinweisen, dass sie sich für Änderungen der E-Mail-Adresse an das MCI-Team wenden müssen.

Um den Nutzern eine klare Rückmeldung über den Erfolg ihrer Profilaktualisierungen zu geben, soll das Self-Service-Portal entsprechende Benachrichtigungen anzeigen. Nach dem Speichern der Änderungen soll eine positive Meldung erscheinen, die dem Nutzer bestätigt, dass sein Profil erfolgreich aktualisiert wurde. Diese Meldung kann beispielsweise in Form einer grünen Erfolgsmeldung oder eines Bestätigungsdialogs dargestellt werden. Im Falle eines Fehlers während der Profilaktualisierung, z.B. aufgrund von Netzwerkproblemen oder Serverausfällen, soll das Portal eine klare Fehlermeldung anzeigen. Diese Meldung soll den Nutzer darüber informieren, dass ein Problem aufgetreten ist und die Änderungen möglicherweise nicht gespeichert wurden. Eine Empfehlung, es zu einem späteren Zeitpunkt erneut zu versuchen oder sich bei anhaltenden Problemen an das MCI-Team zu wenden, soll ebenfalls enthalten sein. Die Fehlermeldung soll in einem auffälligen Format, z.B. einer roten Box, dargestellt werden, um die Aufmerksamkeit des Nutzers zu erlangen.

Erweiterte Profilbearbeitung für Teamassistenten

Zusätzlich zu den Standardfunktionen der Profilbearbeitung soll das Self-Service-Portal eine erweiterte Funktionalität für Benutzer mit der Rolle “Teamassistenz” (Teamass) bieten. Teamassistenten übernehmen administrative Aufgaben innerhalb ihrer Institute und benötigen daher die Möglichkeit, nicht nur ihr eigenes Profil, sondern auch die Profile anderer Personen innerhalb ihres Instituts zu bearbeiten.

Im WordPress Headless CMS sind die Benutzer mit verschiedenen Rollen versehen, darunter auch die Rolle “Teamass”. Zusätzlich zur Rolle ist für jeden Benutzer auch die Zugehörigkeit zu einem bestimmten Institut hinterlegt, wie beispielsweise “INF2”. Das Self-Service-Portal soll diese Informationen nutzen, um den Zugriff auf die Profilbearbeitung basierend auf der Rolle und der Institutszugehörigkeit zu steuern. Benutzer mit der Rolle “Teamass” sollen erweiterte Berechtigungen erhalten, die es ihnen ermöglichen, die Profile aller Personen zu bearbeiten, die demselben Institut zugeordnet sind. Wenn eine Person im CMS beispielsweise die Rolle “Teamass” und die Institutszugehörigkeit “INF2” hat, soll sie in der Lage sein, alle Profile von Personen zu bearbeiten,die ebenfalls dem Institut “INF2” zugeordnet sind, unabhängig von deren individueller Rolle.

Für Teamassistenten soll eine spezielle Benutzeroberfläche im Self-Service-Portal bereitgestellt werden, die ihre erweiterten Berechtigungen widerspiegelt. Nach der Anmeldung sollen Teamassistenten eine übersichtliche Liste aller Profile sehen, die zu ihrem Institut gehören. Die Liste kann beispielsweise in Form einer Tabelle oder Karten dargestellt werden und Informationen wie Namen, Rollen und Kontaktdaten der Personen enthalten. Durch Auswahl eines Profils aus der Liste soll der Teamassistent in die Lage versetzt werden, die Profildetails der entsprechenden Person zu bearbeiten. Die Bearbeitungsansicht soll ähnlich gestaltet sein wie die Ansicht für die Bearbeitung des eigenen Profils, mit den gleichen Feldern und Einschränkungen (z.B. keine Bearbeitung der E-Mail-Adresse).

Explorativer Profilzugriff

Neben den Funktionen für authentifizierte Nutzer und Teamassistenten soll das Self-Service-Portal auch nicht authentifizierten Nutzern die Möglichkeit bieten, auf die Profile der Institutsangehörigen lesend zuzugreifen. Dieser explorative Zugriff ermöglicht es allen Interessier-

ten, sich über die Mitglieder des Instituts zu informieren und deren Profildaten einzusehen, ohne sich anmelden zu müssen.

Um den Nutzern eine einfache Möglichkeit zu bieten, gezielt nach bestimmten Personen zu suchen, soll das Self-Service-Portal eine namensbasierte Textsuche bereitstellen. Nutzer können den Vor- und/oder Nachnamen der gesuchten Person in das Suchfeld eingeben. Das System soll dann eine Volltextsuche in den Namen aller vorhandenen Profile durchführen und die entsprechenden Suchergebnisse anzeigen.

Zusätzlich zur namensbasierten Suche soll das Self-Service-Portal auch ein seitenbasiertes Durchklicken durch die Profile ermöglichen. Dies ist besonders hilfreich, wenn Nutzer einen allgemeinen Überblick über die Mitglieder des Instituts erhalten möchten oder wenn sie sich nicht an den genauen Namen einer Person erinnern. Die öffentliche Profilansicht soll in mehrere Seiten unterteilt sein, wobei jede Seite eine begrenzte Anzahl von Profilen anzeigt. Am unteren Ende der Seite sollen Navigationsschaltflächen oder eine Seitennummerierung bereitgestellt werden, mit denen die Nutzer zur nächsten oder vorherigen Seite blättern können. So können sie bequem durch die Profile navigieren und sich einen Überblick verschaffen.

Damit schließen wir das Unterkapitel zur Anforderungsanalyse ab, und wenden uns im folgenden einem Überblick der Anwendungsarchitektur zu, welche uns die Umsetzung der innerhalb dieses Kapitels besprochenen Kernfunktionalitäten ermöglicht.

5.2 Architekturdesign

In diesem Kapitel werden wir das Architekturdesign der Self-Service-Applikation näher betrachten. Das Architekturdesign bildet das Fundament für die Entwicklung einer robusten, skalierbaren und wartbaren Anwendung. Es definiert die Struktur, die Komponenten und die Interaktionen zwischen den verschiedenen Teilen der Applikation.

Durch die Kombination von Svelte und SvelteKit können wir eine Architektur entwickeln, die sowohl eine reaktive und interaktive Benutzererfahrung im Frontend bietet als auch eine effiziente Verarbeitung und Datenintegration im Backend ermöglicht. Die Architektur wird so gestaltet, dass sie den Anforderungen der Self-Service-Applikation gerecht wird und gleichzeitig Flexibilität und Erweiterbarkeit für zukünftige Anpassungen bietet. Im Folgenden werden wir die verschiedenen Aspekte des Architekturdesigns genauer beleuchten. Der Leser kann dabei natürlich jederzeit das Repository der Applikation referenzieren [Bir24].

Überblick

Wie im Unterkapitel 4.2.1 und der Abbildung 4.6 erklärt, beziehungsweise gezeigt, wurde, ist ein zentrales Konzept der Architektur die Verwendung von Routen als Organisationseinheit, Diese spiegeln sich direkt in der Verzeichnisstruktur der Applikation wieder.

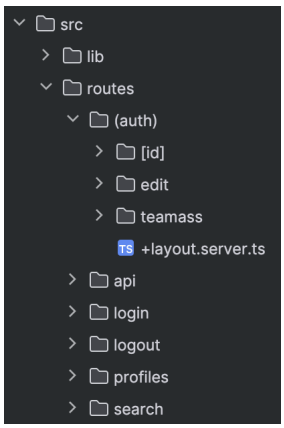

Abbildung 5.1: Überblick der Verzeichnisstruktur

Anhand der bereitgestellten Verzeichnisstruktur lässt sich erkennen, dass die Applikation in verschiedene Bereiche unterteilt ist. Der (auth)-Ordner enthält Routen, die eine Authentifizierung des Benutzers erfordern. Dazu gehören beispielsweise die Routen für die Bearbeitung des eigenen Profils (/edit) oder für den Zugriff auf Teamassistenz-Funktionalitäten (/teamass). Nur authentifizierte Benutzer haben Zugriff auf diese geschützten Bereiche der Applikation.

Im Gegensatz dazu stehen die Routen außerhalb des (auth)-Ordners, die auch für nicht authentifizierte Benutzer zugänglich sind. Dazu zählen beispielsweise die Startseite (/), die Anmeldeseite (/login) und die öffentliche Profilansicht (/profiles). Diese Routen ermöglichen den Zugriff auf allgemeine Informationen und Funktionen, ohne dass eine Authentifizierung erforderlich ist. Wir führen uns folgende Grafik vor Augen:

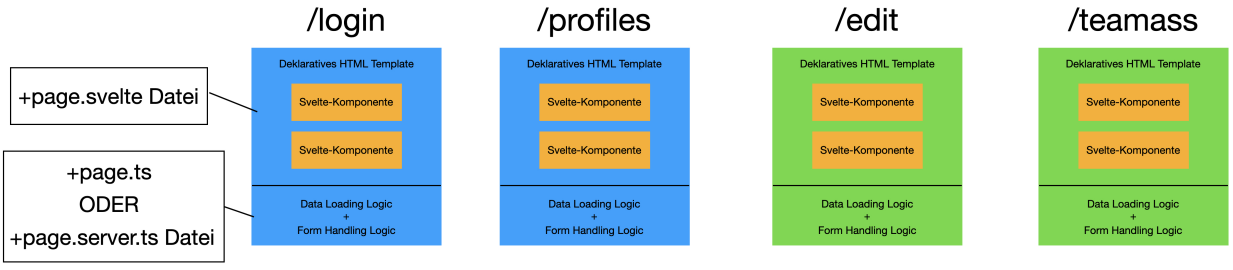

Abbildung 5.2: Sinnbildliche Visualisierung der Routen-Organisation der Applikation

Die Grafik 5.2 beinhaltet einige Aspekte, die sich der Leser nochmals bewusst machen sollte. Jeder Ordner welcher eine Route darstellt, besteht aus zwei Dateien, einer Datei für das HTML Template, und einer Datei welche für das Laden von Daten und sonstige Backend-Funktionalität (Beispielweise Form Handling) zuständig ist. Die Datei welche das HTML Template darstellt, die +page.svelte Datei, ist eine Svelte-Komponente, die wiederum andere Svelte-Komponenten beinhalten kann. Die blaue und die grüne Unterscheidung zwischen den Routen soll darstellen, dass die Routen keinen, beziehungsweise einen, authentifizierten

Status des Nutzers benötigen um aufgerufen werden zu können und im Browser dargestellt zu werden.

Durch die Verwendung von Routen als Organisationseinheit und die Trennung zwischen authentifizierten und nicht authentifizierten Bereichen wird eine klare Struktur und Zugriffskontrolle erreicht. Diese Architektur fördert die Modularität, Wiederverwendbarkeit und Skalierbarkeit der Applikation und ermöglicht eine effiziente Entwicklung und Wartung.

Frontend-Architektur

Die Frontend-Architektur der Self-Service-Applikation basiert auf den in 4.1 dargelegten Prinzipien von Svelte und nutzt eine komponentenbasierte Struktur. Jede Route der Applikation wird durch eine spezifische Kombination von HTML-Templates (Das HTML Template der Route ist ebenfalls eine Svelte-Komponente), den darin enthaltenen Svelte-Komponenten und zugehöriger Logik definiert. Wie in Abbildung 5.2 dargestellt, besteht jede Route aus zwei Dateien. Die oberste Datei bildet das HTML-Template, das die grundlegende Struktur und das Layout der Seite definiert. Innerhalb des Templates werden Svelte-Komponenten eingebettet, die wiederverwendbare und modulare Bestandteile der Benutzeroberfläche darstellen. Die Svelte-Komponenten bilden das Herzstück der Frontend-Architektur. Sie ermöglichen eine deklarative Beschreibung der Benutzeroberfläche und kapselt die zugehörige Logik und den Zustand. Komponenten können geschachtelt und wiederverwendet werden, um komplexe Strukturen aufzubauen und die Wartbarkeit und Skalierbarkeit der Applikation zu verbessern. Wir haben also Komponenten als Organisationseinheit für Svelte, und Routen als Organisationseinheit für SvelteKit. Auch wenn die Grenze zwischen Front- und Backend bei einem Meta-Framework oft verschwimmt, kann man dennoch sagen, dass Komponenten vordergründig die Architektur des Frontends bestimmen, und Routen selbiges für das Backend tun. Vollständig trennscharf ist diese Unterteilung jedoch nicht.

Ein weiterer wichtiger Aspekt der Frontend-Architektur ist der Datenfluss zwischen den Komponenten. Svelte verwendet einen unidirektionalen Datenfluss, bei dem Daten von über geordneten Komponenten an untergeordnete Komponenten weitergegeben werden. Dieser Ansatz sorgt für eine klare Struktur und erleichtert das Verständnis und die Verwaltung des Datenflusses innerhalb der Applikation. In der Self-Service-Applikation werden verschiedene Arten von Daten verwendet und dargestellt. Dazu gehören beispielsweise Benutzerprofile, Authentifizierungsinformationen und Suchergebnisse. Die Daten werden von den entsprechenden Routen geladen, verarbeitet und über Props (Properties) an untergeordnete Komponenten weitergegeben und somit schlussendlich innerhalb des HTML-Templates gerendert und durch den Browser dargestellt. Dadurch wird eine klare Trennung zwischen Daten und Darstellung erreicht und die Wiederverwendbarkeit der Komponenten gefördert.

Backend-Architektur

Die Backend-Architektur der Self-Service-Applikation baut auf den Prinzipien von SvelteKit auf und nutzt die Möglichkeit, Backend-Code innerhalb von +page.server.ts-Dateien auszuführen. Ähnlich wie bei der Frontend-Architektur spielt die Aufteilung der Applikation in dedizierte Routen auch im Backend eine zentrale Rolle. SvelteKit bietet mehrere

Möglichkeiten zur Implementierung von Backend-Funktionalität, aber die Verwendung von +page.server.ts-Dateien ist für unsere Applikation von besonderer Bedeutung.

Jede Route, die durch ein Verzeichnis repräsentiert wird, kann entweder eine +page.ts- oder eine +page.server.ts-Datei enthalten. Die Entscheidung, welche Datei verwendet wird, hängt davon ab, ob die Route Backend-Funktionalität benötigt oder nicht. Ein wesentlicher Aspekt der Backend-Architektur besteht darin, sorgfältig zu analysieren, welche Routen Backend-Funktionalität erfordern und welche nicht. Diese Entscheidung hat Auswirkungen auf die Leistung, Sicherheit und Skalierbarkeit der Applikation. Betrachten wir ein konkretes Beispiel: Die /profiles-Route der Applikation greift auf einen öffentlich verfügbaren Endpunkt der WordPress-API zu. Da dieser Endpunkt keine Authentifizierung erfordert, benötigen wir für diese Route keine Backend-Funktionalität. In diesem Fall kann die Route eine +page.ts-Datei verwenden und von den Vorteilen des dynamischen Renderings profitieren. Das bedeutet, dass die Seite nach dem initialen Laden auch auf dem Client gerendert werden kann, was die Leistung und Interaktivität verbessert. Im Gegensatz dazu erfordert die /edit-Route den Zugriff auf einen Endpunkt der WordPress REST API, der nur authentifizierte HTTP-Anfragen akzeptiert. In diesem Fall benötigen wir eine +page.server.ts-Datei, um die notwendige Backend-Funktionalität bereitzustellen. Innerhalb der +page.server.ts-Datei haben wir Zugriff auf die Backend-Umgebung, einschließlich der .env-Datei, in der die erforderlichen Authentifizierungsinformationen (z.B. ein Application-Password) abgelegt sind. Durch die Verwendung von Backend-Code können wir sensible Informationen wie API-Keys sicher verwenden, ohne sie im Client-Code offenzulegen.

Die Aufteilung der Backend-Funktionalität in separate +page.server.ts-Dateien bietet mehrere Vorteile. Zunächst ermöglicht sie eine klare Trennung von Frontend und Backend, was die Wartbarkeit und Skalierbarkeit der Applikation verbessert. Durch die Ausführung von Backend-Code auf dem Server können wir sicherheitskritische Operationen durchführen, ohne sensible Daten im Client offenzulegen. Dies trägt dazu bei, potenzielle Sicherheitsrisiken zu minimieren und die Integrität der Applikation zu gewährleisten. Ein weiterer Vorteil der Verwendung von +page.server.ts-Dateien besteht darin, dass wir serverseitige Logik und Datenverarbeitung von der Benutzeroberfläche trennen können. Dadurch können wir komplexe Operationen auf dem Server durchführen, ohne die Leistung des Clients zu beeinträchtigen. Dies ist besonders wichtig für rechenintensive Aufgaben oder Operationen, die eine Interaktion mit externen Diensten oder Datenbanken erfordern.

Datenmodell und Wordpress REST API

Das Datenmodell der Self-Service-Applikation basiert auf einem sogenannten “Custom Post Type” [1] im Headless WordPress CMS. Für die Verwaltung der Institutsmitglieder wurde ein spezifischer Custom Post Type namens “Person” im CMS angelegt. Jedes Institutsmitglied wird somit als ein einzelner Post vom Typ “Person” modelliert. Dieser Ansatz ermöglicht eine strukturierte und konsistente Speicherung der Mitgliederdaten. Ein Custom Post Type erweitert die Funktionalität von WordPress über die Standardbeiträge und Seiten hinaus und ermöglicht die Definition benutzerdefinierter Inhaltstypen. Durch die Verwendung des “Person” Custom Post Types können wir spezifische Felder und Metadaten definieren, die für die Beschreibung eines Institutsmitglieds relevant sind, wie z.B. Name, Rolle, Kontakt-

[1] `https://wordpress.org/plugins/custom-post-types/`

informationen und Profilbild. Diese Felder können im WordPress Backend bearbeitet und verwaltet werden, was eine zentrale Pflege der Mitgliederdaten ermöglicht.

Um mit der Sammlung (Collection) der Posts vom Typ “Person” zu interagieren und CRUD-Operationen (Create, Read, Update, Delete) auszuführen, nutzen wir die WordPress REST API. Die REST API bietet eine standardisierte Schnittstelle, um auf die Daten im WordPress CMS zuzugreifen und diese zu manipulieren. Durch die Verwendung von HTTP-Anfragen an spezifische Endpunkte können wir Mitgliederdaten abrufen und bestehende Mitglieder aktualisieren. Das Löschen und Erstellen von Mitglieder wurde im Rahmen dieser Arbeit nicht umgesetzt.

Zum Beispiel können wir durch eine GET-Anfrage an den Endpunkt /wp-json/wp/v2/person/id die Person mit der zugehörigen ID abrufen. Dies ist in Abbildung 5.3 veranschaulicht.

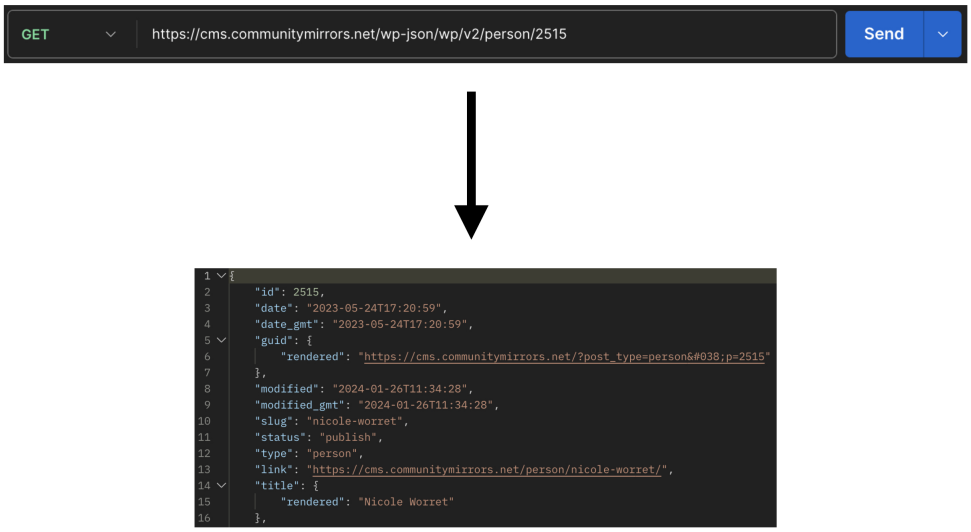

Abbildung 5.3: Senden einer GET-Request an den API-Endpoint der ”Person“ Collection

Die REST API gibt die Daten in einem standardisierten JSON-Format zurück, was eine einfache Integration in unsere Applikation ermöglicht.

Ein wichtiger Aspekt bei der Interaktion mit der WordPress REST API ist die Authentifizierung. Alle Operationen auf der Collection, die nicht ausschließlich lesend sind, erfordern eine Authentifizierung. Dies bedeutet, dass Anfragen, die Daten ändern, nur von authentifizierten Benutzern oder Systemen durchgeführt werden können. In unserem Fall werden authentifizierte Anfragen ausschließlich vom Server ausgeführt, um die Sicherheit zu gewährleisten.

Der Grund dafür liegt darin, dass für die Authentifizierung gegenüber der WordPress REST API ein sogenanntes Application Password (im Wesentlichen ein API-Schlüssel[2]) verwendet

[2]https://wordpress.com/support/security/two-step-authentication/application-specific-passwords

wird. Dieses Application Password darf aus Sicherheitsgründen nicht im Client gespeichert oder übertragen werden, da es sonst von unbefugten Personen abgefangen werden könnte. Stattdessen wird das Application Password sicher auf dem Server als Umgebungsvariable gespeichert, und bei Bedarf für authentifizierte Anfragen verwendet. Durch die Verwendung von serverseitiger Authentifizierung stellen wir sicher, dass sensible Operationen nur von autorisierten Systemen durchgeführt werden können, wodurch die Sicherheit und Integrität der Daten gewährleistet wird.

Verwendete Technologien

Während Svelte und SvelteKit den Kern der Self-Service-Applikation bilden, kommen auch weitere unterstützende Technologien zum Einsatz, die in der heutigen Entwicklung von modernen Web-Applikationen unverzichtbar sind. Diese Technologien ergänzen und erweitern die Funktionalität von Svelte und SvelteKit und tragen zu einer effizienteren Entwicklung, besseren Benutzerfreundlichkeit und Skalierbarkeit bei.

- **TypeScript**[3] : TypeScript ist eine typisierte Obermenge von JavaScript, die es ermöglicht, statische Typen zu definieren und somit die Codequalität und Wartbarkeit zu verbessern. während der Entwicklung erkannt und behoben werden, was die Stabilität und Zuverlässigkeit der Applikation erhöht.
- **Tailwind CSS**[4] : Tailwind CSS ist ein Utility-First-CSS-Framework, das eine schnelle und effiziente Gestaltung von Benutzeroberflächen ermöglicht. Anstatt vordefinierte Komponenten zu verwenden, bietet Tailwind CSS eine Vielzahl von Utility-Klassen, mit denen Entwickler ihre eigenen Designs erstellen können. können wir eine konsistente und ansprechende Benutzeroberfläche für die Self-Service-Applikation gestalten.
- **DaisyUI**[5] : DaisyUI ist eine Komponentenbibliothek, die auf Tailwind CSS aufbaut und vorgefertigte UI-Komponenten bereitstellt. ohne jede Komponente von Grund auf neu zu gestalten. Die Verwendung von DaisyUI beschleunigt die Entwicklung und sorgt für ein konsistentes Erscheinungsbild der Applikation.
- **SQLite**[6] : SQLite ist eine leichtgewichtige, dateibasierte Datenbank, die sich nahtlos in die Self-Service-Applikation integrieren lässt. In unserem Fall verwenden wir SQLite speziell für die persistente Speicherung der in der Anforderungsanalyse erwähnten benutzerdefinierten Anmeldeinformationen (custom user credentials). SQLite ermöglicht es uns, diese sensiblen Daten sicher und zuverlässig zu speichern und bei Bedarf abzurufen.
- **Prisma**[7] : Prisma ist ein modernes Datenbanktoolkit, das die Arbeit mit Datenbanken in Node.js und TypeScript vereinfacht. Es bietet eine typsichere und intuitive API für die Interaktion mit der Datenbank und unterstützt verschiedene Datenbanktypen wie PostgreSQL, MySQL und SQLite. Mit Prisma können wir die Datenbankoperationen in unserer Applikation effizient und sicher durchführen.

[3] `https://www.typescriptlang.org`
[4] `https://tailwindcss.com`
[5] `https://daisyui.com`
[6] `https://www.sqlite.org`
[7] `https://www.prisma.io`

Durch die Kombination dieser Technologien mit Svelte und SvelteKit schaffen wir eine leistungsstarke und moderne Technologieplattform für die Entwicklung der Self-Service-Applikation. TypeScript erhöht die Codequalität und Wartbarkeit, während Tailwind CSS und DaisyUI eine effiziente Gestaltung ansprechender Benutzeroberflächen ermöglichen. Prisma vereinfacht die Datenbankinteraktion und SQLite dient als zuverlässiger Speicher für Anmeldeinformationen.

Somit wurde dem Leser ein Eindruck dafür verliehen, wie sich die Architektur der Applikation darstellt. Für präzisere Eindrücke, sei auf das Repository verwiesen [Bir24]. Aufbauend auf unserem grundlegenden Verständnis von Svelte, SvelteKit und der Architektur der Applikation, wollen wir im folgenden auf die konkrete Umsetzung von einer Teilmenge der Kernfunktionalitäten eingehen.

5.3 Implementierung der Kernfunktionalitäten

Nach der eingehenden Analyse der Anforderungen und dem sorgfältigen Entwurf der Architektur kommen wir nun zum Kernstück der Entwicklung der Self-Service-Applikation: der Implementierung der zentralen Funktionalitäten. In diesem Kapitel werden wir uns detailliert mit der technischen Umsetzung der Schlüsselfunktionen befassen, die für den erfolgreichen Betrieb und die Benutzerfreundlichkeit der Applikation von entscheidender Bedeutung sind. Die Implementierung der Kernfunktionalitäten gliedert sich in drei wesentliche Bereiche: die LDAP-Authentifizierung, die Custom-Authentifizierung und die Profilbearbeitung.

Zunächst werden wir uns der Implementierung der LDAP-Authentifizierung widmen (5.3.1). Die Integration der Self-Service-Applikation mit dem bestehenden LDAP-System der Universität ist von zentraler Bedeutung, um den Benutzern einen sicheren und nahtlosen Zugang zu ermöglichen.

Darüber hinaus werden wir die Implementierung der Custom-Authentifizierung behandeln (5.3.2). Da nicht alle Benutzer über LDAP-Anmeldeinformationen verfügen, ist es erforderlich, eine alternative Authentifizierungsmethode anzubieten.

Ein weiterer Schwerpunkt dieses Kapitels liegt auf der Implementierung der Profilbearbeitung (5.3.3). Die Möglichkeit für Benutzer, ihre eigenen Profilinformationen zu aktualisieren und zu verwalten, ist ein wesentlicher Bestandteil der Self-Service-Funktionalität. Wir werden die Integration mit dem Headless CMS über die WordPress REST API eingehend erläutern und die notwendigen Erweiterungen der API diskutieren, um den spezifischen Anforderungen der Profilbearbeitung gerecht zu werden.

5.3.1 Implementierung der LDAP-Authentifizierung

Die LDAP-Authentifizierung stellt eine der beiden zentralen Authentifizierungsmethoden dar, die in der Self-Service-Applikation implementiert wurden. Diese Methode ermöglicht es Benutzern, sich mit ihren bestehenden Universitäts-Anmeldedaten anzumelden und gewährleistet somit eine nahtlose Integration in die bestehende IT-Infrastruktur der Universität.

Der Authentifizierungsprozess läuft dabei folgendermaßen ab:

1. **Eingabe der Anmeldedaten**: Der Benutzer gibt seine Anmeldedaten (Benutzername und Passwort) in das Login-Formular der Self-Service-Applikation ein.
2. **Übermittlung an das Backend** : Der Client sendet die eingegebenen Anmeldedaten sicher an das Backend der SvelteKit-Applikation.
3. **LDAP-Überprüfung**: Das Backend verwendet einen LDAP-Client[8] , der in der Node.js-Umgebung ausgeführt wird, um die übermittelten Anmeldedaten gegen den LDAP-Server der Universität zu prüfen. Backend gewährleistet die Sicherheit der Anmeldedaten, da diese nicht im Client-seitigen Code verarbeitet werden.
4. **Authentifizierungsantwort**: Der DAP-Server sendet eine Antwort zurück, die angibt, ob die Kombination aus Benutzername und Passwort gültig ist. Diese Antwort wird vom LDAP-Client im Backend entgegengenommen und verarbeitet.
5. **Token-Generierung**: Bei erfolgreicher Authentifizierung generiert das Backend einen JSON Web Token [9] (JWT). Ein JWT ist ein kompaktes und selbstständiges Mittel zur sicheren Übermittlung von Informationen zwischen Parteien als JSON-Objekt. Diese Tokens können digital signiert werden, um ihre Integrität zu gewährleisten.
6. **Token-Speicherung**: Der generierte JWT wird anschließend in einem sicheren, HTTP-only Cookie [10] gespeichert. Dies bietet einen zusätzlichen Schutz gegen Cross-Site Scripting [11] (XSS) Angriffe, da das Cookie nicht über JavaScript zugänglich ist.
7. **Token-Validierung**: Bei nachfolgenden Anfragen wird der JWT mithilfe eines privaten Schlüssels (PRIVATE_KEY) validiert, der als Umgebungsvariable im Backend gespeichert ist. Dieser Prozess stellt sicher, dass der Token nicht manipuliert wurde und noch gültig ist.
8. **Authentifizierungsstatus**: Basierend auf der Validität des Tokens wird der Authentifizierungsstatus des Benutzers in der Applikation gesetzt. Dies ermöglicht es der Anwendung, den Zugriff auf geschützte Ressourcen und Funktionen entsprechend zu steuern.

Die Verwendung von JWTs in Kombination mit sicheren Cookies bietet mehrere Vorteile:

- Zustandslosigkeit: JWTs enthalten alle notwendigen Informationen, um den Benutzer zu authentifizieren, wodurch serverseitige Sitzungsspeicherung vermieden wird.
- Sicherheit: Durch die Verwendung von digitalen Signaturen und sicheren Cookies wird ein hohes Maß an Sicherheit gewährleistet.
- Flexibilität: JWTs können zusätzliche Informationen wie Benutzerrollen enthalten, was die Autorisierung vereinfacht.

Die Implementierung der LDAP-Authentifizierung in Kombination mit JWTs stellt sicher, dass die Self-Service-Applikation sowohl sicher als auch benutzerfreundlich ist. Sie ermöglicht es Benutzern, ihre vertrauten Universitäts-Anmeldedaten zu verwenden, während gleichzeitig

[8] `https://github.com/ldapts/ldapts`
[9] `https://jwt.io`
[10] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies`
[11] `https://www.kaspersky.de/resource-center/definitions/what-is-a-cross-site-scripting-attack`

moderne Sicherheitsstandards eingehalten werden. Diese Methode bildet eine solide Grundlage für die Authentifizierung und Autorisierung innerhalb der Applikation und gewährleistet, dass nur berechtigte Benutzer Zugriff auf sensible Funktionen und Daten erhalten. Diese spielt sich dann wie folgt ab:

Wie in Abbildung 5.1 dargestellt, haben wir eine spezielle Route-Gruppe namens (auth) implementiert. Diese Gruppe spielt eine zentrale Rolle bei der Durchsetzung der Authentifizierung für bestimmte Routen der Applikation. Innerhalb der (auth)-Route-Gruppe befindet sich eine +layout.server.ts Datei. Diese Datei enthält eine load-Funktion,die einen entscheidenden Teil des Authentifizierungsprozesses darstellt. Die load-Funktion wird bei jeder Navigation zu einer Route innerhalb der (auth)-Gruppe automatisch ausgeführt. Ihre Hauptaufgabe besteht darin, den JWT (JSON Web Token) des Benutzers zu überprüfen und sicherzustellen, dass nur authentifizierte Benutzer Zugriff auf die geschützten Bereiche der Applikation erhalten. Der Ablauf dieser Überprüfung ist in dem Code-Ausschnitt 5.1 dargestellt.

```
1  import { redirect } from '@sveltejs/kit';
2  import { verifyToken } from '$lib/utils/jwtAuth.js';
3
4  // The load function only checks whether the token is present and valid
5  // This part of the code is not responsible for generating the token
6  export function load({ cookies }) {
7    const token = cookies.get('jwt');
8
9    //If no token is present, redirect to login page
10   if (!token) {
11     throw redirect(303, '/login');
12   }
13   const { valid } = verifyToken(token);
14   if (!valid) {
15     throw redirect(303, '/login');
16   }
17 }
```

Listing 5.1: Auszug der +layout.server.ts

Es werden in 5.1 folgende Schritte durchlaufen:

1. Token-Extraktion: Die Funktion extrahiert den JWT aus dem sicheren Cookie, in dem er gespeichert ist.
2. Token-Validierung: Der extrahierte Token wird mithilfe des PRIVATE_KEY überprüft.
3. Entscheidung über den Zugriff: Wenn der Token gültig ist, wird der Zugriff auf die angeforderte Route gewährt, und sonst zur /login Seite umgeleitet.

Durch die Verwendung dieser Route-Gruppe und der serverseitigen load-Funktion wird ein robuster und effizienter Mechanismus zur Durchsetzung der Authentifizierung in der Self-Service-Applikation implementiert. Dies stellt sicher, dass nur berechtigte Benutzer Zugriff auf sensible Bereiche und Funktionen der Applikation erhalten, während gleichzeitig eine nahtlose und benutzerfreundliche Erfahrung gewährleistet wird.

5.3.2 Implementierung der Custom-Authentifizierung

Wie bereits in der Anforderungsanalyse diskutiert, wurde die Implementierung einer Custom-Authentifizierung als wichtige Ergänzung zur LDAP-Authentifizierung identifiziert. Der Hauptgrund dafür ist die Erkenntnis, dass die Authentifizierung über den LDAP-Server mittelfristig nicht die primäre Methode für die IT-Dienste der Universität bleiben wird. Die Einführung einer Custom-Authentifizierung gewährleistet somit die Zukunftsfähigkeit und Flexibilität des Self-Service Portals.

Eine zentrale Anforderung war es, den Administratoren des Self-Service Portals, Dr. Julian Fietkau und Laura Stojko, die Möglichkeit zu geben, im Backend neue Benutzer anzulegen. Um den Entwicklungsaufwand zu minimieren und die Sicherheit zu erhöhen, wurde entschieden, diese Funktionalität nicht über eine neue Benutzeroberfläche, sondern durch die Ausführung eines Skripts im Backend bereitzustellen.

Für die Umsetzung der Custom-Authentifizierung wurden folgende Technologien gewählt:

1. **SQLite als Datenbanklösung**: SQLite wurde als Datenbanklösung für die persistente Speicherung der Benutzerdaten gewählt. Im Gegensatz zu anderen Datenbanksystemen wie MySQL[12] oder PostgreSQL[13] , die als separate Server-Prozesse laufen, ist SQLite eine serverlose, selbstständige Datenbank. Die gesamte Datenbank wird als einzelne Datei auf der lokalen Maschine gespeichert, auf der auch die Web-Applikation läuft. Dies bietet mehrere Vorteile:
    - Einfache Konfiguration und Wartung
    - Keine zusätzliche Server-Software erforderlich
    - Hohe Portabilität, da die gesamte Datenbank in einer Datei enthalten ist
    - Geringer Ressourcenverbrauch, ideal für kleinere bis mittelgroße Anwendungen
2. **Prisma als ORM** (Object-Relational Mapping): Prisma wurde als ORM gewählt, um eine typsichere Interaktion mit der Datenbank in der TypeScript-Entwicklungsumgebung zu ermöglichen. Prisma bietet mehrere Vorteile:
    - Definition des Datenbankschemas in einer deklarativen und intuitiven Syntax
    - Automatische Generierung von TypeScript-Typen basierend auf dem Datenbankschema
    - Typsichere Datenbankabfragen, die Laufzeitfehler reduzieren
    - Migrations-Management für einfache Schemaänderungen

Die Implementierung der Custom-Authentifizierung umfasst folgende Hauptkomponenten:

1. Datenbankschema: Definition des Schemas für Benutzer mit relevanten Feldern wie Benutzername, gehashtes Passwort und Rollen.
2. Prisma-Client: Generierung und Verwendung des Prisma-Clients für typsichere Datenbankoperationen im Backend.

[12] `https://www.mysql.com`
[13] `https://www.postgresql.org`

3. Authentifizierungslogik: Implementierung der Logik zum Vergleichen von eingegebenen Anmeldedaten mit den in der Datenbank gespeicherten Informationen.
4. Token-Generierung: Bei erfolgreicher Authentifizierung Generierung eines JWT, ähnlich wie bei der LDAP-Authentifizierung.

Für die Administration der Benutzerdaten bietet Prisma eine besonders nützliche Funktion: Prisma Studio. Dies ist ein lokaler Entwicklungsserver mit einer intuitiven grafischen Benutzeroberfläche, die es den Administratoren ermöglicht, Benutzer einfach und effizient anzulegen, zu bearbeiten oder zu löschen. Die Administratoren können Prisma Studio über ein Kommandozeilenskript starten und dann über einen Webbrowser auf die Benutzeroberfläche zugreifen. Die GUI ist in 5.4 dargestellt.

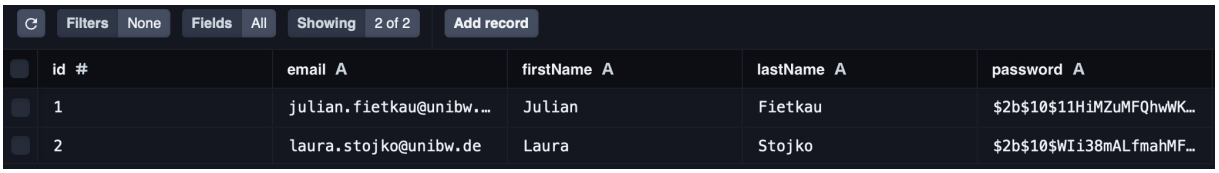

Abbildung 5.4: Darstellung der Prisma GUI für intuitive Nutzerverwaltung

Die Verwendung von Prisma Studio bietet mehrere Vorteile:

- Benutzerfreundliche Oberfläche für Datenbankoperationen
- Direkte Visualisierung der Datenbankstruktur und -inhalte
- Möglichkeit zur einfachen Dateneingabe und -bearbeitung ohne SQL-Kenntnisse
- Automatische Validierung der Eingaben basierend auf dem Datenbankschema

Durch die Kombination von SQLite für die Datenspeicherung, Prisma als ORM und Prisma Studio für die Administration wurde eine robuste, flexible und benutzerfreundliche Lösung für die Custom-Authentifizierung geschaffen. Diese Implementierung ermöglicht es den Administratoren, das System effektiv zu verwalten, während gleichzeitig eine sichere und skalierbare Authentifizierungsmethode für das Self-Service Portal bereitgestellt wird.

5.3.3 Implementierung der Profilbearbeitung

Die Implementierung der Profilbearbeitung ist ein zentraler Aspekt der Self-Service-Applikation und erfordert eine sorgfältige Integration verschiedener Komponenten. Wie in den vorherigen Abschnitten dieser Arbeit erläutert, erfolgt die Kommunikation zwischen dem Headless WordPress CMS und den Clients über die WordPress REST API. Diese Architektur ermöglicht eine flexible und effiziente Verwaltung der Profilinformationen.

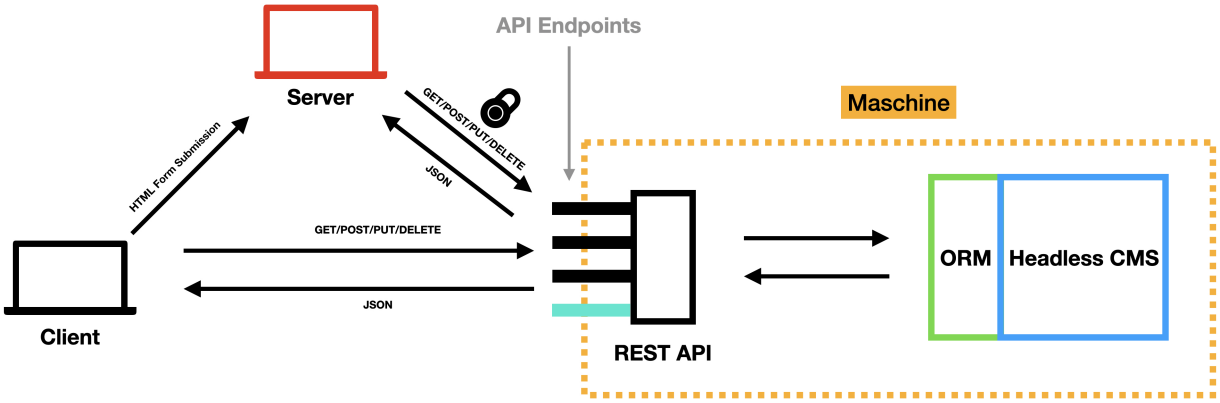

Abbildung 5.5: Illustration der Interaktion zwischen den zugehörigen Komponenten im Kontext der Profilaktualisierung

Die WordPress REST API stellt verschiedene Endpoints bereit, über die auf die im CMS gespeicherten Daten zugegriffen werden kann. Diese Endpoints lassen sich in zwei Kategorien unterteilen:

1. Unauthentifizierte Endpoints: Diese akzeptieren HTTP-Requests ohne Authentifizierung. In der Regel handelt es sich hierbei um lesende (GET) Anfragen, die öffentlich zugängliche Informationen abrufen.
2. Authentifizierte Endpoints: Diese erfordern eine Authentifizierung und werden hauptsächlich für schreibende Operationen(POST, PUT, DELETE) verwendet, die Änderungen am CMS-Inhalt vornehmen.

Wie in Kapitel 5.2 erwähnt, verwendet WordPress für die Authentifizierung von API-Anfragen sogenannte Application Passwords. Diese funktionieren ähnlich wie API-Keys und ermöglichen eine sichere Kommunikation zwischen der Self-Service-Applikation und dem WordPress CMS. Diese Unterscheidung ist entscheidend für die Sicherheit und Integrität der Daten im CMS.

Das Application Password wird als Umgebungsvariable auf dem Server gespeichert, um die Sicherheit zu gewährleisten und unbefugten Zugriff zu verhindern. Diese Implementierung hat eine wichtige Konsequenz: Authentifizierte Requests müssen immer vom Server aus durchgeführt werden, da das Application Password nicht im Client-Code verfügbar sein darf. Um die Anforderung zu erfüllen, dass authentifizierte Requests vom Server ausgehen müssen, nutzt die SvelteKit-Applikation server-seitige Form Actions. Diese Funktionalität ermöglicht es, HTML-Formulare, die auf dem Client ausgefüllt und abgesendet werden, auf dem Server zu verarbeiten.

Um die Anforderung zu erfüllen, dass authentifizierte Requests vom Server ausgehen müssen, nutzt die SvelteKit-Applikation server-seitige Form Actions. Diese Funktionalität ermöglicht es, HTML-Formulare, die auf dem Client ausgefüllt und abgesendet werden, auf dem Server zu verarbeiten.

Der Prozess läuft wie folgt ab (siehe Abbildung 5.5):

- Der Client sendet ein ausgefülltes HTML-Formular an den Server.
- Der Server empfängt die Formulardaten über die Form Action.
- Die Form Action auf dem Server verarbeitet die Daten und fügt die notwendige Authentifizierung (Application Password) hinzu.
- Der Server sendet die authentifizierte Anfrage an die WordPress REST API.
- Die REST API verarbeitet die Anfrage und aktualisiert die Daten im Headless CMS.
- Die API sendet eine Antwort zurück an den Server.
- Der Server verarbeitet die Antwort und sendet gegebenenfalls eine Bestätigung oder Fehlermeldung an den Client.

Dieser Ansatz gewährleistet, dass sensible Informationen wie das Application Password niemals den Server verlassen und gleichzeitig eine nahtlose Benutzerinteraktion ermöglicht wird. Die Verwendung von server-seitigen Form Actions in SvelteKit bietet mehrere Vorteile:

1. Sicherheit: Sensible Daten und Authentifizierungsinformationen bleiben auf dem Server.
2. Flexibilität: Komplexe Verarbeitungslogik kann auf dem Server implementiert werden, ohne den Client zu belasten.
3. Konsistenz: Die Datenverarbeitung erfolgt zentral auf dem Server, was die Wartung und Fehlerbehandlung vereinfacht.

Durch diese Implementierung der Profilbearbeitung wird eine sichere, effiziente und benutzerfreundliche Lösung geschaffen, die es Benutzern ermöglicht, ihre Profilinformationen zu aktualisieren, während gleichzeitig die Integrität und Sicherheit des CMS gewährleistet bleibt. Bevor wir dieses Kapitel beenden soll noch ein elementar wichtiger Punkt Erwähnung finden. Nämlich die Erweiterung der Wordpress REST API. Denn diese Aspekt spielt im Rahmen der Profilbearbeitung und Profildarstellung eine tragende, aber dennoch versteckte Rolle.

Erweiterung der WordPress API

Die WordPress REST API bietet in ihrer Standardkonfiguration eine breite Palette von Funktionen für die Verwaltung von Inhalten. Allerdings reicht diese Standardfunktionalität nicht immer aus, um den spezifischen Anforderungen von benutzerdefinierten Anwendungen wie unserem Self-Service Portal gerecht zu werden. In solchen Fällen ist es notwendig,die API um zusätzliche Endpunkte zu erweitern, um maßgeschneiderte Funktionen bereitzustellen.

Im Rahmen der Entwicklung unseres Self-Service Portals wurde deutlich, dass für bestimmte Funktionen, insbesondere für die Profilselektion und Profilbearbeitung, zusätzliche Endpunkte erforderlich waren. Ein konkretes Beispiel hierfür ist die Implementierung eines Endpunkts, der, gegeben ein bestimmtes Institut, alle Personen-Objekte zurückgibt, die diesem Institut zugeordnet sind. Diese spezifische Funktionalität war in der Standardkonfiguration der WordPress REST API nicht vorhanden.

WordPress bietet glücklicherweise eine flexible Möglichkeit, die REST API zu erweitern [Dev24]. Dies geschieht durch das Hinzufügen von benutzerdefiniertem Code in der functions.php Datei des WordPress-Themes. In dieser Datei können neue Endpunkte definiert und die Logik für ihre Funktionalität implementiert werden.

```
    function get_people_by_org($request) {
    $org_substring = $request['org'];
    $args = array(
        'post_type' => 'person',
        'posts_per_page' => -1,  // Retrieve all matching posts
        'meta_query' => array(
            array(
                'key' => 'org',
                'value' => $org_substring,
                'compare' => 'LIKE'
            )
        )
    );

    $query = new WP_Query($args);
    $posts = array();

    if ($query->have_posts()) {
        while ($query->have_posts()) {
            $query->the_post();
            $posts[] = array(
                'id' => get_the_ID(),
                'title' => get_the_title(),
                'link' => get_permalink(),
                'org' => get_field('org')
            );
        }
    }

    return new WP_REST_Response($posts, 200);
}
```

Listing 5.2: Code für die Erstellung eines custom End Points innerhalb der functions.php Datei

Der Code-Abschnitt 5.2 implementiert die Funktion get_people_by_org, die als Callback für den benutzerdefinierten Endpunkt der WordPress REST API dient. Diese Funktion erweitert die Standardfunktionalität der API, um Personen basierend auf ihrer Organisationszugehörigkeit abzufragen.

Hier sind die Hauptaspekte:

1. Die Funktion akzeptiert einen request Parameter, der den Organisations-Substring enthält.
2. Es wird eine WordPress-Abfrage (WP_Query) erstellt, die spezifisch nach Posts vom Typ 'person' sucht. Dies entspricht dem im Kapitel erwähnten Custom Post Type für Personen.
3. Die Abfrage verwendet eine Meta-Query, um Personen zu finden, deren 'org' Feld (Ein

benutzerdefiniertes Feld, implementiert mit Advanced Custom Fields[14]) den gesuchten Substring enthält.

4. Die Funktion durchläuft alle gefundenen Posts und extrahiert relevante Informationen wie ID, Titel, Permalink und das 'org' Feld.
5. Schließlich werden die gesammelten Daten als JSON-Response zurückgegeben.

Die Implementierung dieses benutzerdefinierten Endpunkts in der functions.php des WordPress-Themes, wie im vorherigen Kapitel beschrieben, würde es der SvelteKit-Anwendung ermöglichen, diese maßgeschneiderte Abfrage direkt über die API durchzuführen, ohne komplexe client-seitige Filterung oder mehrfache API-Aufrufe.

Es ist wichtig zu beachten, dass sich dieser Code nicht im Repository der SvelteKit-Anwendung befindet. Stattdessen muss er direkt im WordPress Admin-Dashboard konfiguriert werden. Dies stellt einen wichtigen Aspekt in der Verwaltung und Wartung des Systems dar, da Änderungen an der API-Erweiterung eine separate Aktualisierung des WordPress-Backends erfordern.

In der Abbildung 5.5 ist der benutzerdefinierte Endpunkt zur Hervorhebung türkis eingefärbt, was die Erweiterung der Standardfunktionalität der WordPress REST API verdeutlicht.

Die Implementierung solcher benutzerdefinierten Endpunkte ermöglicht es uns, die Funktionalität des Self-Service Portals genau an unsere Bedürfnisse anzupassen. Sie erlaubt effizientere Datenabfragen und -manipulationen, die sonst mehrere Anfragen oder komplexe Client-seitige Verarbeitung erfordern würden.

Diese Erweiterung der WordPress REST API ist ein entscheidender Faktor für die Flexibilität und Leistungsfähigkeit unseres Self-Service Portals und zeigt, wie Standard-CMS-Lösungen an spezifische Anforderungen angepasst werden können.

In diesem Kapitel haben wir einige der Kernfunktionalitäten unseres Self-Service Portals beleuchtet und deren Implementierung skizziert. Dabei war es nicht das Ziel, eine detaillierte technische Dokumentation zu liefern, sondern vielmehr dem Leser ein Gefühl dafür zu vermitteln, wie spezifische Aspekte einer Anwendung in einem Meta-Framework wie SvelteKit umgesetzt werden können. Die Beispiele der LDAP-Authentifizierung, der Custom-Authentifizierung und der Profilbearbeitung zeigen exemplarisch, wie moderne Webtechnologien genutzt werden können, um komplexe Anforderungen effizient zu erfüllen. Diese Einblicke dienen als Grundlage für Kapitel 6, in dem wir, basierend auf den Ergebnissen dieser Arbeit, eine breitere Perspektive einnehmen werden. Wir werden untersuchen und extrapolieren, inwiefern sich Meta-Frameworks wie SvelteKit allgemein für die effiziente und effektive Entwicklung von Diensten in einer heterogenen IT-Infrastruktur eignen. Diese Betrachtung wird es uns ermöglichen, die Potenziale und Herausforderungen solcher Frameworks im Kontext komplexer, organisationsweiter IT-Landschaften zu bewerten und Schlussfolgerungen für zukünftige Entwicklungsprojekte zu ziehen.

[14] `https://www.advancedcustomfields.com`

6 Evaluation

Die Entwicklung des Self-Service Portals für das MCI-Institut bot eine wertvolle Gelegenheit, die Umsetzbarkeit und Praktikabilität unseres Ansatzes in einem realen Projektkontext zu evaluieren. In diesem Kapitel betrachten wir kritisch die Ergebnisse unserer Implementierung, beleuchten die Stärken und potenziellen Schwächen des gewählten Ansatzes und ziehen Schlussfolgerungen für zukünftige Projekte.

6.1 Erfüllung der Projektanforderungen

Die primäre Zielsetzung des Projekts war die Erfüllung der definierten Anforderungen für das Self-Service Portal. In dieser Hinsicht kann die Entwicklung als erfolgreich betrachtet werden. Folgende Hauptanforderungen wurden vollständig umgesetzt:

- Implementierung einer dualen Authentifizierungsmethode (LDAP und Custom)
- Entwicklung einer benutzerfreundlichen Oberfläche zur Profilbearbeitung
- Integration mit dem bestehenden Headless WordPress CMS
- Bereitstellung erweiterter Bearbeitungsmöglichkeiten für Teamassistenten
- Implementierung eines explorativen Zugriffs für nicht authentifizierte Nutzer

Die erfolgreiche Umsetzung dieser Anforderungen demonstriert die Eignung des gewählten Ansatzes für die Entwicklung von Webanwendungen mit verschiedenen Integrationen und Funktionalitäten im universitären Kontext.

6.2 Integration mit externen Diensten

Ein Hauptfokus des Projekts lag auf der erfolgreichen Integration mit bestehenden externen Diensten. Hier betrachten wir die wichtigsten Integrationen und ihre Umsetzung:

LDAP-Integration

Die Anbindung an den universitären LDAP-Server erwies sich als effektiv. Durch die Nutzung einer LDAP-Bibliothek [1] konnten wir eine robuste Authentifizierung implementieren. Herausforderungen bestanden hauptsächlich in der korrekten Konfiguration und dem sicheren Umgang mit den Anmeldedaten. Die gewählte Lösung ermöglicht eine nahtlose Authentifizierung für Universitätsangehörige, ohne zusätzliche Anmeldeinformationen zu erfordern.

[1] `https://www.npmjs.com/package/ldapts`

Headless WordPress CMS

Die Integration mit dem bestehenden Headless WordPress CMS war ein zentraler Aspekt des Projekts, da das CMS die Datenbasis für die Applikation darstellt. Durch die Nutzung der WordPress REST API konnten wir eine effiziente Schnittstelle für den Datenaustausch nutzen und für unsere Zwecke erweitern. Herausforderungen lagen hier in der Handhabung von Authentifizierung und Autorisierung bei API-Anfragen, in der Strukturierung der Daten für die Darstellung im Portal, sowie der richtigen Konfiguration und Erweiterung der REST API.

Benutzerdefinierte Authentifizierung

Neben LDAP implementierten wir eine benutzerdefinierte Authentifizierungsmethode unter Verwendung von SQLite und Prisma ORM. Diese Lösung bietet die Flexibilität auf zukünftige Änderungen in der Authentifizierungsstruktur der Universität zu reagieren und ermöglicht eine effiziente Verwaltung von Benutzerkonten außerhalb des LDAP-Systems.

6.3 Entwicklungseffizienz und Wartbarkeit

Der gewählte Ansatz eines Meta-Frameworks erwies sich als vorteilhaft für die Entwicklungseffizienz und Wartbarkeit des Projekts:

Einheitliche Codebase

Die Möglichkeit, Frontend- und Backend-Code in einer einzigen Codebase zu verwalten, reduzierte die Komplexität erheblich. Dies führte zu verbesserter Übersichtlichkeit und erleichterte Wartung und Debugging.

Typsicherheit durch TypeScript

Die Verwendung von TypeScript trug wesentlich zur Codequalität bei. Obwohl der native Support für TypeScript nicht exklusiv für SvelteKit ist, sondern auch in anderen modernen Frameworks wie React, Angular oder Vue zu finden ist, erwies sich diese Funktionalität in unserem Projekt als besonders wertvoll. Die statische Typisierung half, Fehler frühzeitig im Entwicklungsprozess zu erkennen und zu beheben, was die Stabilität und Zuverlässigkeit der Applikation erhöhte.

Ein besonders nützlicher Aspekt war die Möglichkeit, einen typsicheren Datenaustausch zwischen unserer Web-Applikation und der WordPress REST API zu implementieren. Durch die Definition von TypeScript-Interfaces, die die Struktur der von der API zurückgegebenen Daten abbilden, konnten wir eine zusätzliche Ebene der Typsicherheit in unsere Applikation einführen. Dies führte nicht nur zu einer Reduktion von Laufzeitfehlern, sondern verbesserte auch die Entwicklererfahrung durch präzise Autovervollständigung und Typüberprüfung in der Entwicklungsumgebung.

Diese Typisierung der API-Responses ermöglichte es uns, potenzielle Inkonsistenzen zwischen den erwarteten und tatsächlichen Datenstrukturen frühzeitig zu erkennen. Dadurch konn-

ten wir robustere Komponenten entwickeln, die besser mit verschiedenen Datenzuständen umgehen können, was wiederum zu einer stabileren und wartbaren Anwendung führte.

Modulare Struktur

Die komponenten- und routenbasierte Organisation des Codes, ein mittlerweile verbreitetes Konzept in modernen Component- und Meta-Frameworks, erwies sich als äußerst vorteilhaft für unser Projekt. Dieser Ansatz bietet nicht nur eine klare Struktur, die gut wartbar ist, sondern stellt auch ein intuitives mentales Modell dar, das sowohl für erfahrene Entwickler als auch für Neueinsteiger leicht verständlich ist.

Die Aufteilung in Komponenten ermöglicht eine hohe Wiederverwendbarkeit und Modularität des Codes. Jede Komponente kapselt ihre eigene Logik, Darstellung und oft auch ihren eigenen Zustand, was die Komplexität reduziert und die Testbarkeit verbessert. Die routenbasierte Organisation spiegelt direkt die Struktur der Anwendung wider, was die Navigation und das Verständnis des Gesamtsystems erleichtert.

6.4 Herausforderungen und Lösungsansätze

Während der Entwicklung traten verschiedene Herausforderungen auf, die spezifische Lösungsansätze erforderten:

Sicherheit bei der API-Kommunikation

Die sichere Handhabung von API-Schlüsseln und Zugangsdaten, insbesondere bei der Kommunikation mit dem WordPress CMS, erforderte sorgfältige Implementierung von Sicherheitsmaßnahmen. Wir lösten dies durch die Verwendung von Umgebungsvariablen und sicheren Server-seitigen Anfragen.

Benutzerfreundlichkeit vs. Sicherheit

Die Balance zwischen einer benutzerfreundlichen Oberfläche und strengen Sicherheitsanforderungen war eine kontinuierliche Herausforderung. Unser Ansatz fokussierte sich auf intuitive Benutzerführung bei gleichzeitiger Implementierung robuster Sicherheitsmaßnahmen im Hintergrund.

Schlussfolgerungen

Das entwickelte Self-Service Portal erfüllt die spezifischen Anforderungen der Informatik Fakultät und demonstriert die Machbarkeit einer integrierten Lösung in einer heterogenen IT-Umgebung. Die gewählte Architektur und der Entwicklungsansatz ermöglichten eine effiziente Umsetzung und bieten gute Voraussetzungen für zukünftige Erweiterungen und Anpassungen.

Für zukünftige Projekte ähnlicher Art empfehlen wir:

- Frühzeitige und detaillierte Planung der Integration mit bestehenden Systemen

- Fokus auf Modularität und Wiederverwendbarkeit von Komponenten
- Kontinuierliche Abstimmung mit Stakeholdern zur Sicherstellung der Anforderungserfüllung
- Berücksichtigung von und Erweiterbarkeit in der initialen Architektur

Es lässt sich somit sagen, dass der gewählte Ansatz sich als effektiv für die Entwicklung eines maßgeschneiderten Self-Service Portals im universitären Umfeld erwiesen hat. Die Erfahrungen und Erkenntnisse aus diesem Projekt bieten wertvolle Einblicke für ähnliche Vorhaben in der Zukunft.

6.5 Kritische Abwägungen

Während die Umsetzung des Self-Service Portals im Rahmen dieser Arbeit erfolgreich war und die Integration der verschiedenen Elemente gelungen ist, ist es wichtig, die Ergebnisse im Kontext der spezifischen Projektanforderungen und -umstände zu betrachten. Eine kritische Reflexion ermöglicht es uns, die Grenzen und potenziellen Einschränkungen des gewählten Ansatzes zu erkennen.

Skalierbarkeit und Komplexität

Es ist hervorzuheben, dass das entwickelte Self-Service Portal eine Applikation von überschaubarem Umfang und Komplexität darstellt, die primär für eine begrenzte Nutzerbasis konzipiert wurde. In diesem Kontext erwies sich der gewählte Ansatz mit SvelteKit und der Organisation durch Routen als effizient und zweckmäßig. Jedoch ist es durchaus vorstellbar, dass bei deutlich komplexeren Anwendungen oder solchen mit einer signifikant größeren Nutzerbasis die Organisationseinheit “Route” an ihre Grenzen stoßen könnte.

Bei Projekten mit erheblich höherer Komplexität oder umfangreicheren Anforderungen an die Geschäftslogik könnte eine klassische Client-Server-Aufteilung in Verbindung mit einer dedizierten Backend-Architektur möglicherweise eine besser skalierbare und wartbarere Lösung darstellen. Dies würde eine klarere Trennung von Verantwortlichkeiten ermöglichen und potenziell die Entwicklung und Wartung umfangreicher Systeme erleichtern.

Performanz-Überlegungen

Ein weiterer kritischer Aspekt betrifft die Performanz-Anforderungen von Webanwendungen. Während das entwickelte Self-Service Portal für seine Zwecke ausreichende Leistung bietet, könnten Dienste mit hohen Performanz-Ansprüchen an die Grenzen der JavaScript-basierten Architektur stoßen.

Ein Beispiel aus dem universitären Kontext wäre ein System zur Echtzeitverarbeitung und -visualisierung von Forschungsdaten. Aufgrund der inhärenten Performanz-Limitationen von JavaScript als Interpreter-Sprache wären derartige Services wahrscheinlich besser in dedi-

zierten Backend-Sprachen wie Java[2], Go[3] oder Rust[4] umzusetzen, die eine effizientere Ressourcennutzung und höhere Rechenleistung ermöglichen.

Abwägung zwischen Entwicklungseffizienz und Spezialisierung

Die Wahl eines Meta-Frameworks wie SvelteKit bietet zweifellos Vorteile in Bezug auf Entwicklungsgeschwindigkeit und die nahtlose Integration von Frontend und Backend. Dies ist besonders vorteilhaft für Projekte mittlerer Größe und Komplexität, wie das hier entwickelte Self-Service Portal. Allerdings muss bei der Entscheidung für einen solchen Ansatz sorgfältig abgewogen werden zwischen der Entwicklungseffizienz und den potenziellen Vorteilen einer spezialisierten Architektur.

Für zukünftige Projekte empfiehlt es sich daher, eine gründliche Analyse der spezifischen Anforderungen, der erwarteten Skalierung und der Performanz-Bedürfnisse durchzuführen. In einigen Fällen könnte eine Hybrid-Lösung optimal sein, bei der ein Meta-Framework für den Großteil der Anwendung verwendet wird, während besonders ressourcenintensive oder komplexe Komponenten in spezialisierten Backend-Services implementiert werden.

Zusammenfassend lässt sich sagen, dass der in dieser Arbeit gewählte Ansatz für das spezifische Projekt des Self-Service Portals angemessen und erfolgreich war. Für zukünftige Entwicklungen, insbesondere solche mit höheren Anforderungen an Skalierbarkeit, Komplexität oder Performanz, sollten jedoch die hier diskutierten Aspekte sorgfältig in Betracht gezogen werden, um die optimale Architektur und Technologie-Stack zu wählen.

[2] `https://www.java.com/en/`
[3] `https://go.dev`
[4] `https://www.rust-lang.org`

7 Fazit

Die rasante Entwicklung von Web-Technologien stellt Entwickler und Organisationen vor die ständige Herausforderung, neue Ansätze und Frameworks zu evaluieren und deren Potenzial für reale Anwendungsfälle zu erschließen. Diese Masterarbeit hat sich dieser Aufgabe gewidmet, indem sie die aufkommende Technologie der Meta-Frameworks am Beispiel eines konkreten Projekts untersucht hat.

7.1 Zusammenfassung

Das primäre Ziel dieser Arbeit war es, die Eignung und Effektivität von Meta-Frameworks für die Lösung realer Probleme in der Webentwicklung zu evaluieren. Zu diesem Zweck wurde ein Self-Service-Portal für die Profilverwaltung der Community-Mirrors der Informatik-Fakultät entwickelt, das als Testfall für den Einsatz moderner Web-Technologien diente.

Im Zentrum der Untersuchung standen das Component-Framework Svelte und das darauf aufbauende Meta-Framework SvelteKit. Die Wahl dieser Technologien wurde sorgfältig begründet, und wesentliche Aspekte beider Frameworks wurden näher beleuchtet. Dies umfasste eine Einführung in die Svelte-Syntax, den innovativen Kompilierungsansatz und das Reaktivitätskonzept von Svelte, sowie die Routing-Mechanismen, das Data-Loading und die Rendering-Strategien von SvelteKit. Um dem Leser ein umfassendes Verständnis zu vermitteln, wurde zunächst die Problemstellung des Self-Service-Portals erläutert. Dabei wurden die spezifischen Anforderungen und Herausforderungen des Projekts dargestellt, insbesondere im Kontext der Integration mit bestehenden Systemen wie dem Headless CMS und der LDAP-Authentifizierung. Parallel dazu wurden die für die Problemlösung ausgewählten Tools und Technologien vorgestellt und ihre Relevanz für das Projekt erklärt.

Ein wesentlicher Teil der Arbeit widmete sich der Architektur einer SvelteKit-Applikation. Hier wurde dem Leser ein Einblick in den strukturellen Aufbau und die Funktionsweise von SvelteKit-Anwendungen gegeben. Dies bildete die Grundlage für das Verständnis der nachfolgenden Implementierungsschritte. Aufbauend auf dieser architektonischen Basis wurde eine repräsentative Auswahl von Entwicklungsaufgaben präsentiert. Anhand dieser Beispiele wurde demonstriert, wie konkrete Funktionalitäten wie die LDAP-Authentifizierung, die Custom-Authentifizierung und die Profilbearbeitung innerhalb der SvelteKit-Architektur implementiert wurden. Diese praktischen Beispiele dienten dazu, die theoretischen Konzepte in einen realen Entwicklungskontext zu setzen und die Anwendbarkeit von SvelteKit für Webapplikationen zu veranschaulichen. Abschließend wurden die Ergebnisse des Projekts kritisch eingeordnet und bewertet. Dabei wurden sowohl die Stärken als auch die potenziellen Herausforderungen bei der Verwendung von Svelte und SvelteKit diskutiert. Die Evaluation umfasste Aspekte wie die Erfüllung der Anforderungen, die Effizienz der Entwicklung,

die intuitive Bedienbarkeit und Zukunftssicherheit sowie Überlegungen zur Performanz und Skalierbarkeit.

Diese umfassende Betrachtung ermöglicht es, fundierte Schlüsse über die Eignung von Meta-Frameworks wie SvelteKit für die Entwicklung moderner Webapplikationen zu ziehen und bietet wertvolle Einblicke für Entwickler und Entscheidungsträger, die vor ähnlichen technologischen Herausforderungen stehen.

7.2 Ausblick

Die Untersuchung und Implementierung des Self-Service-Portals mit Svelte und SvelteKit hat wertvolle Einblicke in die Möglichkeiten und Herausforderungen moderner Meta-Frameworks geliefert. Dennoch ist es wichtig anzuerkennen, dass eine umfassende Bewertung einer Technologie erst durch ihre Anwendung in einer Vielzahl von Echtweltszenarien mit unterschiedlichen Anforderungsprofilen möglich ist.

Die vorliegende Arbeit stellt einen wichtigen ersten Schritt in diese Richtung dar, kann aber naturgemäß nur einen Ausschnitt des gesamten Potenzials und der möglichen Einsatzgebiete von Meta-Frameworks beleuchten. Trotz dieser Einschränkung hat die Arbeit einen wertvollen Beitrag zur Validierung von Meta-Frameworks im Kontext realer Anwendungsentwicklung geleistet. Sie hat gezeigt, dass auch weniger bekannte Frameworks wie Svelte und SvelteKit bedeutende Innovationen und Effizienzsteigerungen in den Entwicklungsprozess einbringen können. Die Ergebnisse unterstreichen, dass es sich lohnt, über den Tellerrand der etablierten Lösungen hinauszublicken und neue Ansätze in Betracht zu ziehen.

Ein wichtiger nächster Schritt wird die Ausbringung der im Rahmen dieser Arbeit entwickelten Applikation auf einem Server der Bundeswehr Universität München sein. Dieser finale Test unter realen Betriebsbedingungen wird zusätzliche Erkenntnisse über die Leistungsfähigkeit, Skalierbarkeit und Wartbarkeit der Lösung liefern. Er wird auch die Möglichkeit bieten, die Interaktion mit bestehenden Systemen und die Benutzererfahrung unter echten Bedingungen zu evaluieren. Es ist zu hoffen, dass diese Arbeit dazu beiträgt, den Blick für innovative Lösungsansätze bei der Implementierung zukünftiger Services an der Universität zu weiten. Meta-Frameworks wie SvelteKit sollten als valide Option in Erwägung gezogen werden, insbesondere für Applikationen, die eine Kombination aus Backend-Funktionalität und effizienter Implementierung erfordern. Sie bieten das Potenzial, den Entwicklungsprozess zu beschleunigen, die Codebase übersichtlicher zu gestalten und gleichzeitig leistungsfähige und skalierbare Anwendungen zu erstellen.

Abschließend lässt sich sagen, dass die Welt der Webentwicklung sich ständig weiterentwickelt, und es von entscheidender Bedeutung ist, offen für neue Technologien und Ansätze zu bleiben. Die Erfahrungen aus dieser Arbeit können als Ausgangspunkt für weitere Untersuchungen und Projekte dienen, die das volle Potenzial von Meta-Frameworks in verschiedenen Anwendungsbereichen erforschen. Indem wir kontinuierlich neue Technologien evaluieren und adaptieren, können wir sicherstellen, dass die IT-Infrastruktur der Universität zukunftsfähig bleibt und den sich wandelnden Anforderungen der akademischen Gemeinschaft gerecht wird.

Abkürzungsverzeichnis

CMS Content Management System

LDAP Lightweight Directory Access Protocol

DIT Directory Information Tree

DN Distinguished Name

SSR Server Side Rendering

CSR Client Side Rendering

REST Representational State Transfer

API Application Programming Interface

DOM Document Object Model

JWT JSON Web Token

MCI Mensch Computer Interaktion

Abbildungsverzeichnis

2.1 Illustration der Integration der Elemente LDAP, REST API, Web-Framework und CMS 6
2.2 Vereinfachte Darstellung zur Illustration des Unterschieds zwischen gekoppeltem und Headless CMS 9
2.3 Abgrenzung des Unterschieds zwischen der klassischen Web-Architektur und Meta-Frameworks 14
2.4 Illustration der Rolle einer REST API 16

3.1 Vereinfachte Illustration der Gesamtarchitektur der Problemstellung 19

4.1 Von Stack Overflow durchfgeführte Developer Survey zur Beliebtheit von Web-Frameworks [Sta23a]. 28
4.2 Darstellung der gleichen Komponente in React und in Svelte 29
4.3 Darstellung der Struktur einer Svelte Komponente 30
4.4 Illustration des deklarativen HTML-Templating in Svelte 31
4.5 Illustration der entstehenden Komponenten während des Kompilierens 32
36fig4.7 Darstellung der +page.ts und +page.server.ts Datei 38

5.1 Überblick der Verzeichnisstruktur 49
5.2 Sinnbildliche Visualisierung der Routen-Organisation der Applikation 49
5.3 Senden einer GET-Request an den API-Endpoint der "Person“ Collection . . 52
5.4 Darstellung der Prisma GUI für intuitive Nutzerverwaltung 58
5.5 Illustration der Interaktion zwischen den zugehörigen Komponenten im Kontext der Profilaktualisierung 59

Literaturverzeichnis

[Ama24] Amazon Web Services: *What is RESTful API?* https://aws.amazon.com/what-is/restful-api/. Version: 2024. – Accessed: 2024-06-12

[ARH+22] Abdullah, Sharifah Latifah S. ; Razali, Nazirah ; Hassan, Nor A. ; Khalid, Noor S. ; Ismail, Mohd Fairuz H.: The Role of Self-Service Technology and Graduates' Perceived Job Performance in Assessing University Service Quality. In: *ResearchGate* (2022). https://www.researchgate.net/publication/363838852_The_role_of_self-service_technology_and_graduates'_perceived_job_performance_in_assessing_universit – Accessed: 2024-06-05

[Aut22] Author(s): Higher Education Future in the Era of Digital Transformation. In: *Education Sciences* 12 (2022), Nr. 11, 784. http://dx.doi.org/10.3390/educsci12110784. – DOI 10.3390/educsci12110784

[Bep23] Bepyan: How Does the Svelte Compiler Work? (2023). https://bepyan.me/en/post/svelte-compiler-operation/. – Accessed: 2024-06-15

[Bir24] Biren, Jonathan: *SvelteKit Self Service Portal.* https://github.com/jonathanbiren/SvelteKitStarter. Version: 2024. – Accessed: 2024-06-19

[Cai23] Caisy.io: *What is a Meta-Framework?* https://caisy.io/blog/what-is-a-meta-framework. Version: 2023. – Accessed: 2024-06-14

[Con23] Contributors, SitePoint: Signals: Fine-Grained Reactivity for JavaScript Frameworks. (2023). https://www.sitepoint.com/signals-fine-grained-javascript-framework-reactivity/. – Accessed: 2024-06-17

[Con24] Contributors, WordPress: *WordPress REST API.* https://developer.wordpress.org/rest-api/, 2024. – Accessed: 2024-06-05

[DD15] Delaney, Rob ; D'Agostino, Robert: *The Challenges of Integrating New Technology into an Organization*, La Salle University, Mathematics and Computer Science Capstones, 2015. https://digitalcommons.lasalle.edu/cgi/viewcontent.cgi?article=1024&context=mathcompcapstones

[Den24] Deneire, Tom: *Understanding Front-End Frameworks: Component-Based Frameworks.* https://tomdeneire.medium.com/understanding-front-end-frameworks-component-based-frameworks-d72a84c87745, 2024. – Accessed: 2024-06-05

[Dev22] Developer Nation Community: The Rise and Fall of Web Frameworks. In: *Developer Nation* (2022), November. https://www.developernation.net/blog/the-rise-and-fall-of-web-frameworks/. – Accessed: 2024-06-23

[Dev24] Developers, WordPress: *Extending the REST API – REST API Handbook*. https://developer.wordpress.org/rest-api/extending-the-rest-api/. Version: 2024. – Accessed: 2024-06-22

[Dig21] Digital, Lloyds: Comparing Reactivity Models: React vs Vue vs Svelte vs MobX vs Solid. (2021). https://dev.to/lloyds-digital/comparing-reactivity-models-react-vs-vue-vs-svelte-vs-mobx-vs-solid-29m8. – Accessed: 2024-06-17

[Fil23] Filippov, Vlad: *Building Your Own JavaScript Framework*. Birmingham : Packt Publishing, 2023 https://www.packtpub.com/product/building-your-own-javascript-framework/9781804617403. – ISBN 978–1–80461–740–3

[Har19] Harris, Rich: Virtual DOM is pure overhead. (2019). https://svelte.dev/blog/virtual-dom-is-pure-overhead. – Accessed: 2024-06-17

[Har23a] Harris, Rich: Rich Harris on why he created Svelte. (2023). https://www.offerzen.com/blog/rich-harris-on-why-he-created-svelte. – Accessed: 2024-06-15

[Har23b] Harris, Rich: The story of Svelte. (2023). https://www.offerzen.com/blog/rich-harris-the-story-of-svelte. – Accessed: 2024-06-15

[Kys20] Kyslova, Kseniia: *Main SPA SEO challenges and ways to make your web app discoverable in search*. https://proxify.io/articles/single-page-app-spa-seo. Version: 2020. – Accessed: 2024-06-14

[NFK+10] Nauerz, Andreas ; Falb, Jürgen ; Kern, Heiko ; Klein, Jürgen ; Strauss, Jürgen: CommunityMashup: A Service Oriented Approach. In: *Proceedings of the 1st International Workshop on Lightweight Integration on the Web (ComposableWeb 2010)* Bd. 705, 2010 (CEUR Workshop Proceedings)

[Opt24] Optimizely: What is a Content Management System (CMS)? In: *Optimizely Optimization Glossary* (2024). https://www.optimizely.com/optimization-glossary/content-management-system/

[Par24] PartnerHero: *4 Ways Self-Service Impacts Customer Experience*. https://www.partnerhero.com/blog/4-ways-self-service-impacts-customer-experience, 2024. – Accessed: 2024-06-05

[Pri23] Prismic: Understanding the JavaScript Meta-Framework Ecosystem. In: *Prismic Blog* (2023). https://prismic.io/blog/javascript-meta-frameworks-ecosystem. – Accessed: 2024-06-23

[Sch21] Schwarz, Ben: Small Bundles, Fast Pages: What To Do With Too Much JavaScript. (2021). https://calibreapp.com/blog/bundle-size-optimization. – Accessed: 2024-06-17

[Sof24] Softnet, Focus: *Empowering Productivity with Self-Service Portal*. https://www.focussoftnet.com/blogs/empowering-productivity-with-self-service-portal, 2024. – Accessed: 2024-06-05

[Sta23a] Stack Overflow: *Stack Overflow Developer Survey 2023*. https://survey.stackoverflow.co/2023/. Version: 2023. – Section: Admired and Desired Web Frameworks and Technologies

[Sta23b] Statista: *Most used web frameworks among developers worldwide 2023*. Version: 2023. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

[SWAH19] Sari, Riri F. ; Wibisono, Ari ; Adlina, Dea ; Harwahyu, Ruki: Systematic Literature Review on the LDAP Protocol As a Centralized Mechanism for the Authentication of Users in Multiple Systems. In: *KnE Engineering* 2019 (2019), 248-259. http://dx.doi.org/10.18502/keg.v4i2.5823. – DOI 10.18502/keg.v4i2.5823

[Win23] Wingravity: *The Rise of Meta-Frameworks*. https://www.wingravity.com/blog/the-rise-of-meta-frameworks. Version: 2023. – Accessed: 2024-06-14

Hiermit versichere ich, dass die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden.

Ferner habe ich vom Merkblatt über die Verwendung von Masterarbeit Kenntnis genommen und räume das einfache Nutzungsrecht an meiner Masterarbeit der Universität der Bundeswehr München ein.

Neubiberg, den 01.7.2024

.....

(Unterschrift des Kandidaten)