FOM Hochschule für Oekonomie & Management

Hochschulstandort Bonn

Seminararbeit

im Studiengang Wirtschaftsinformatik

als Teil des Moduls
Web Technologie

über das Thema

Webprojekt

von

Mads Fuchs, Nils Rüber, Janis Wiesen, Clara Tombrink

Betreuer: Julian Schröter
Matrikelnummer: 675314, 674514, 670300, 660710
Eingereicht am: 06.01.2025

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Ein paar einleitende Worte.

1.1 Anforderungen an moderne Webanwendungen

Die Entwicklung moderner Webanwendungen stellt Entwickler vor verschiedene Herausforderungen. Dabei geht es nicht nur um die reine Funktionalität, wie zum Beispiel das Umsetzen eines funktionierenden Webshops, der Webseite an sich. Bei der Entwicklung einer Webseite geht es darüber hinaus auch um die Performance, Benutzerfreundlichkeit, Flexibilität und Skalierbarkeit. Dabei können die unterschiedlichen Aspekte, je nach Anforderung, unterschiedlich stark gewichtet werden, wobei die reine Performance und die Benutzerfreundlichkeit der Webseite für den Nutzer meistens von hoher Bedeutung sind. Die Performance spielt eine große Rolle, da sie den Erfolg der Webseite beeinflussen kann. Nutzer erwarten kurze Ladezeiten und einen einwandfreien Ablauf der Funktionalität[1]. Die Benutzerfreundlichkeit einer Webseite ist deshalb so wichtig, da sie überhaupt erst die Akzeptanz zur Nutzung einer Webseite beeinflusst. Eine Webseite muss grundsätzlich gut lesbar, navigierbar und anpassbar für viele Verschiedene Geräte und Browser sein[2]. Auch die Flexibilität einer Webseite spielt eine übergeordnete Rolle. Sowohl ständig wechselnde Geschäftsanforderungen als auch fortschreitende Technologien setzen voraus, dass Webseiten in der Lage sein müssen auf diese Veränderungen beispielsweise mithilfe von modularen Technologien wie React reagieren zu können[3]. Zuletzt ist die Skalierbarkeit einer Webseite insofern wichtig, dass durch unterschiedliche Nutzerzahlen, der Ressourcenbedarf einer Webseite stark schwanken und dynamisch anpassbar sein sollte[4]. Um die genannten Anforderungen möglichst gut umzusetzen, ist es essenziell wichtig, Frontend und Backend grundsätzlich voneinander zu trennen. Die modulare Vorgehensweise wird hierdurch noch einmal bestätigt und ermöglicht eine flexible Vorgehensweise in der Entwicklung. Die Trennung von Frontend und Backend ist außerdem hilfreich, damit die Entwicklung der beiden Komponenten parallel und unabhängig voneinander erfolgen kann[5]. Dadurch kann bei der Entwicklung Zeit gespart werden und es können schneller Funktionalitäten der Webseite fertiggestellt und getestet oder veröffentlicht werden.

[1] **deinhard_uberblick_2024**.
[2] **siever_responsive_2024**.
[3] **wiedermayer_webentwicklung_2024**.
[4] **annacherniavska_erstellung_2023**.
[5] **hort_warum_2018**.

1.2 Konzept & Workflows

RF InnoTrade stellt eine Dachfirma im E-Commerce-Bereich dar, die sich durch eine starke Diversifizierung Ihrer Produkte auszeichnet. Der primäre Fokus des Unternehmens liegt darauf, schnell und flexibel auf neue Trends sowie Produkte zu reagieren, die dem Reselling-Markt zur Verfügung gestellt werden. Diese Agilität ist wichtig, um schnellstmöglich auf einen sehr dynamischen Markt reagieren zu können.

Um die technische Umsetzung dieses Ansatzes effizient zu gestalten, ohne bei jeder neuen Marke oder Produkt eine komplett neue Webseite erstellen zu müssen, war eine Lösung erforderlich, die sowohl den Programmieraufwand als auch die damit verbundenen Kosten so gering hält wie möglich. Beispielsweise entfallen typische Aufwendungen für die Registrierung neuer Domains, das Hosting sowie teure Upgrades für Subsysteme wie Shopify (auf dessen Plattform ein Ähnliches, aber sehr kostenintensives, Resultat erzielt werden könnte). Die Lösung sollte einen einfachen, schnellen und kostengünstigen Prozess bieten, der es dem Unternehmen ermöglicht, neue Webseiten für Marken oder Produkte mit minimalem Aufwand zu generieren.

In diesem Kontext wurde die Webseite www.rf-innotrade.de um ein neues Admin Center erweitert. Dieses Admin Center ermöglicht eine effiziente Verwaltung und Erstellung neuer Webseiten mit einem vereinfachten Konfigurationsprozess. Es enthält wesentliche Funktionen, die zur Automatisierung und Skalierung der operativen Abläufe beitragen:

1. Die Möglichkeit, mit nur einem Klick eine neue Marke oder Webseite zu erstellen, inklusive eines vollständigen Konfigurationsprozesses.
2. Eine KI-unterstützte Generierung von Beschreibungsbildern und Texten, die den Content-Erstellungsprozess beschleunigt.
3. Eine umfassende Übersicht aller erstellten Marken und Webseiten zur effizienten Verwaltung.
4. Die Möglichkeit, einzelne Marken oder Webseiten on- und offline zu schalten.
5. Die Option, bestehende Marken oder Webseiten hinsichtlich ihrer Rahmenparameter zu bearbeiten.

Technisch gesehen wird für jede neue Webseite oder Marke ein separater Docker-Container gestartet, der ein vorgefertigtes Image enthält. Vor dem Build-Prozess werden spezifische Variablen zugewiesen, die den Inhalt der jeweiligen Webseite definieren. Diese Variablen sowie der zugehörige Inhalt werden zentral im Content-Management-System Strapi gespeichert, was eine effiziente Verwaltung und Anpassung der Daten ermöglicht.

Das übergeordnete Ziel dieser Erweiterung ist es, RF InnoTrade sowohl organisatorisch als auch technisch in die Lage zu versetzen, mit hoher Geschwindigkeit und Flexibilität zu skalieren. Die Lösung trägt somit zur Optimierung von Prozessen bei und unterstützt das Unternehmen dabei, sich schnell an wechselnde Marktanforderungen anzupassen.

Um ein besseres Verständnis für die Funktionalität des Admin Centers zu erhalten wurden drei beispielhafte Workflows erstellt. Diese Workflows zeigen, wie das Admin Center in der Praxis genutzt werden kann, um neue Marken oder Webseiten zu erstellen, zu verwalten und zu bearbeiten. Die Workflows sind in den folgenden Abschnitten detailliert beschrieben.

1.2.1 Workflow 1

Neue Marke mit Webauftritt und Webshop erstellen

1. Der Nutzer öffnet die Webseite www.rf-innotrade.de und navigiert zum Admin Center.
2. Im Admin Center erfolgt die Anmeldung mit den persönlichen Zugangsdaten.
3. Nach erfolgreichem Login befindet sich der Nutzer im Dashboard des Admin Centers. In der oberen rechten Ecke findet er den Button "+ New Page", den er anklickt.
4. Ein Formular öffnet sich, das den Nutzer durch den Konfigurationsprozess leitet.
5. Folgende Pflichtfelder müssen für die initiale Erstellung der Webseite ausgefüllt werden:
    - BrandName (Name der Marke)
    - Slug (URL-freundlicher Name)
    - Logo (Markenlogo)
    - Font (Schriftart)
    - Primärfarbe
    - Sekundärfarbe
6. Nach Eingabe aller relevanten Informationen klickt der Nutzer auf SSeite erstellen".
7. Der Docker Build Prozess startet automatisch. Der Nutzer erhält eine Benachrichtigung, sobald der Build-Vorgang abgeschlossen ist und die Seite zur Verfügung steht.

8. Nach Fertigstellung hat der Nutzer die Möglichkeit, die neue Seite zu verwalten, zu bearbeiten oder aufzurufen.

1.2.2 Workflow 2

Einer bestehenden Marke einen Shop hinzufügen

1. Vorarbeit in der RF InnoTrade Shopify-Instanz:
    - Gewünschte Produkte anlegen
    - Produkte einer neuen Kollektion hinzufügen
    - Aus der neuen Kollektion einen "Buy Buttonërzeugen (ein vorgefertigtes JavaScript-Skript von Shopify)
    - Den Code des Buy Buttons kopieren
2. Der Nutzer loggt sich im Admin Center ein.
3. In der Übersicht wählt der Nutzer die Webseite aus, der er einen Shop hinzufügen möchte, und klickt auf den Ëdit"Button.
4. Es öffnet sich ein Formular. Hier fügt der Nutzer den kopierten Shopify-Code in das Feld "CodeBuyButtonëin.
5. Abschließend klickt der Nutzer auf SSave Changes", um die Änderungen zu speichern und den Shop zu integrieren.

1.2.3 Workflow 3

Webseite offline nehmen oder löschen

1. Der Nutzer navigiert zum Admin Center und meldet sich an.
2. In der Übersicht aller existierenden Webseiten sucht er die betroffene Seite.
3. Um die Webseite temporär offline zu nehmen, klickt der Nutzer auf den Button SStop Container"neben der entsprechenden Webseite.
4. Soll die Webseite dauerhaft entfernt werden, klickt der Nutzer stattdessen auf den "Löschen"Button neben SStop Container".
5. Bei Klick auf "Löschen"wird die Webseite unwiderruflich gelöscht und aus dem System entfernt.

1.3 Projektaufbau

Das Projekt besteht aus mehreren wichtigen Komponenten, die eng miteinander verzahnt sind:

GitHub Repository rfinnotrade

Das Herzstück des Projekts ist das GitHub Repository rfinnotrade. Hier befindet sich die gesamte Projektstruktur mit dem zugehörigen Code. Das Repository gliedert sich in drei zentrale Bereiche. Im Frontend-Bereich ist die gesamte Benutzeroberfläche sowie alle clientseitigen Komponenten implementiert. Das Backend, basierend auf Strapi, verwaltet die Datenbank und stellt die notwendigen REST-APIs bereit. Der Subbrands-Bereich enthält sämtliche Templates und Konfigurationen für die Webseiten, die über das Admin Center erstellt werden. Die Entwicklung erfolgte in Visual Studio Code mit Remote-SSH-Zugriff auf das Repository, was eine sichere und effiziente Zusammenarbeit ermöglichte.

GitHub Repository WebPaper

Für die Dokumentation wurde ein separates LaTeX-Repository eingerichtet. Die Entscheidung für LaTeX als Dokumentationswerkzeug basierte auf mehreren Vorteilen. Zunächst ermöglicht es eine erhebliche Kosteneinsparung im Vergleich zu kommerziellen Lösungen wie Overleaf. Durch die Git-Integration wird eine präzise Versionskontrolle gewährleistet. Die Projektgruppe konnte außerdem auf eine bewährte Projektstruktur aus vorherigen Arbeiten zurückgreifen. Nicht zuletzt überzeugt LaTeX durch das professionelle Erscheinungsbild der generierten PDF-Dokumente. Die gesamte Dokumentation konnte so effizient im Team erstellt und verwaltet werden.

Server

Der Server spielt eine duale Rolle im Projekt. In seiner Funktion als Produktivumgebung ist er für das Hosting der Live-Webseiten und Dienste zuständig. Als Entwicklungsumgebung ermöglicht er die zentrale Verwaltung verschiedener Aspekte: Die Entwicklungsdependenzen werden hier einheitlich verwaltet, alle notwendigen Umgebungsvariablen sind zentral konfiguriert, Build-Prozesse werden standardisiert durchgeführt und Deployment-Workflows sind klar definiert. Diese Zentralisierung hat die Teamarbeit deutlich vereinfacht und Inkonsistenzen zwischen verschiedenen Entwicklungsumgebungen verhindert.

Figma

Figma diente als kollaborative Design-Plattform für verschiedene Projektbereiche. Für die Hauptwebseite www.rf-innotrade.de wurde ein responsives Layout entwickelt, das sich an

verschiedene Bildschirmgrößen anpasst. Das Admin Interface wurde als benutzerfreundliche Verwaltungsoberfläche mit intuitiver Navigation konzipiert. Für den Authentifizierungsbereich wurde ein sicheres und modernes Login-Design erstellt. Das Kontaktformular wurde für optimale Benutzerfreundlichkeit bei Kundenanfragen gestaltet. Besonders wichtig war die Entwicklung eines flexiblen Design-Systems für die automatisch generierten Unterseiten. Durch die Echtzeitkollaboration in Figma konnte das Team Design-Entscheidungen schnell treffen und iterativ verbessern.

2 Grundlagen

2.1 Überblick der eingesetzten Technologien

2.1.1 React

React ist das von uns bevorzugte Frontend. Im Folgenden werden wir erklären, was React ist und wie es grundsätzlich aufgebaut ist. React wurde von Facebook entwickelt und ist eine leistungsstarke Open-Source-Bibliothek, basierend auf JavaScript. Dabei ist React auf die Erstellung von Benutzeroberflächen, also auf das sogenannte Frontend, spezialisiert[6]. Darüber hinaus ist React eine komponentenbasierte Bibliothek, welche einen effizienten und strukturierten Ansatz zur Gestaltung von interaktiven Benutzeroberflächen bietet[7]. Aus diesen Gründen eignet sich die Bibliothek hervorragend für komplexe Benutzeroberflächen, welche eben aus diesen kleinen und modularen Komponenten bestehen. Für die Benutzer bedeutet der Einsatz von React anwenderfreundliche Oberflächen und für die Entwickler eine verbesserte Code-Organisation, was unter anderem zu einer erleichterten Wartung führt[8]. Zudem verwendet React ein virtuelles Document Object Model (DOM). Das virtuelle DOM ist ähnlich zum tatsächlichen DOM. Es ist allerdings wesentlich leichtgewichtiger und wird als JavaScript-Objekt dargestellt. Auf Grund der Darstellung als JavaScript-Objekt werden lediglich die wirklich notwendigen Änderungen am tatsächlichen DOM durchgeführt, wodurch wiederum die Performance optimiert wird[9]. React Komponenten verwenden grundsätzlich zwei verschiedene Konzepte zur Datenverwaltung. Diese beiden Konzepte nennen sich States und Props und arbeiten im Zusammenspiel, um dynamische und interaktive Benutzeroberflächen zu erstellen[10]. States repräsentieren den internen Zustand einer Komponente und können sich im Laufe der Zeit verändern. Sie werden von der React Komponente selber verwaltet und können sowohl durch Benutzeraktionen als auch durch interne Ereignisse verändert werden[11]. Dies ermöglicht überhaupt erst die interaktive Nutzung der Webseite im laufenden Betrieb. Props dagegen sind als Kurzform für Properties, zu Deutsch Eigenschaften, zu verstehen und sind unveränderlich. Sie dienen dazu, bestimmte Funktionen und Daten von einer sogenannten Elternkomponente an andere Komponenten zu übertragen[12].

[6] **acharya_15_2023**.
[7] **p_was_2024**.
[8] **laurent_komponentenbasierte_2023**.
[9] **chauhan_mastering_2023**.
[10] **bhimani_react_2024**.
[11] **nestorowicz_state_2021**.
[12] **platforms_state_nodate**.

2.1.2 Strapi

Als nächstes gehen wir auf unsere verwendete Backendstruktur ein. Die von uns gewählte Backend Technologie, nennt sich Strapi. Strapi ist ein Headless Content Management System (CMS), das Open-Source betrieben wird. Strapi zeichnet sich durch seine Flexibilität und durch seine Entwicklerfreundlichkeit aus. Die Idee hinter Strapi ist die Trennung der Inhaltsverwaltung vom Frontend[13]. Es ermöglicht eine effiziente Erstellung, Verwaltung und Bereitstellung der Inhalte über eine integrierte und benutzerfreundliche Oberfläche. Der Headless-CMS-Ansatz bedeutet, dass alle Inhalte von der Präsentationsschicht getrennt sind. Alle Daten die in Strapi abgelegt und verwaltet werden, werden per API an das Frontend bereitgestellt[14]. Durch die Bereitstellung der Daten per API, ist Strapi in der Wahl des Frontends flexibel, da APIs grundsätzlich einem standardisierten Format folgen und jeweils leicht angepasst werden können. Im Folgenden werde ich etwas konkreter auf die einzelnen Vorteile von Strapi eingehen. Ein wichtiger Aspekt von Strapi ist die vorhandene Flexibilität. Strapi ermöglicht es, Inhalte und Strukturen präzise an die eigenen Projektanforderungen anzupassen. Dabei erlaubt es die No-Code-Konfiguration, das Backend nahezu ohne eigene Programmierung einzurichten. Durch die mitgelieferte, grafische Benutzeroberfläche, können sowohl Datenfelder und Inhaltstypen als auch Relationen zwischen den Daten bequem erstellt werden[15]. In diesem Zusammenhang ist die einfache Integration von Strapi in das Gesamtprojekt zu erwähnen. Es können unterschiedlichste Datenbanksysteme mit Strapi genutzt werden. Darunter SQL-basierte, aber auch NoSQL-basierte. Die Daten wiederum können ganz einfach per API an das unabhängige Frontend übermittelt werden[16]. Ein weiterer äußerst wichtiger Aspekt, der von Strapi abgedeckt wird, ist die IT-Sicherheit. Strapi verfügt bereits von Haus aus, über grundlegende Sicherheitsfunktionen wie Authentifizierung und Authorisierung. Zudem liefert Strapi die Möglichkeit zur Konfiguration von Rollen- und Berechtigungskonzepten direkt mit[17].

2.1.3 JavaScript, HTML und CSS

Neben dem eigentlichen Front- und Backend, gibt es weitere Programmiersprachen, die bei der Entwicklung einer Webseite eine Rolle spielen. In diesem Kapitel gehen wir genauer auf diese Sprachen und deren Funktionen ein.

[13] **autor_strapi_2024**.
[14] **behrens_headless-cms_nodate**.
[15] **viehmann_headless_2024**.
[16] **autor_strapi_nodate**.
[17] **siever_basics_2024**.

JavaScript ist eine diversifizierte und leistungsfähige Programmiersprache, die in vielen Webentwicklungen eine Rolle spielt. JavaScript wird dabei clientseitig, also lokal auf dem jeweiligen Endgerät des Nutzers, ausgeführt. Dies ermöglicht unter Anderem den Einsatz von clientseitiger Validierung, wodurch Daten bereits lokal überprüft und somit die Serverlast reduziert werden kann[18]. Der Einsatz von JavaScript ermöglicht außerdem die Implementierung von Logik und Interaktivität, wodurch die dynamischen Benutzeroberflächen im Frontend erst geschaffen und funktional gestaltet werden können. Konkret wird mit JavaScript die Manipulation des DOM ermöglicht[19]. Darüber hinaus können mithilfe von JavaScript Benutzerinteraktionen wie Klicks, Tastatureingaben oder Mausbewegungen verarbeitet werden. Durch den gleichzeitigen Einsatz von AJAX-Technologien können Daten asynchron mit dem Server ausgetauscht werden, wodurch die nahtlose Aktualisierung von Webseiteninhalten und eine echte und dynamische Interaktivität sichergestellt wird[20]. Die grundsätzliche Konzeption von JavaScript sieht vor, dass wiederverwendbare Funktionen als Kernbaustein genutzt und bei Bedarf aufgerufen werden. JavaScript unterstützt verschiedene, gängige Datentypen wie Zahlen, Zeichenketten und Objekte. Dadurch ist es auch möglich, einen objektorientierten Ansatz zu verfolgen.

Auch HTML spielt bei der Entwicklung von modernen Webseiten eine Rolle. Über HTML wird die grundlegende Strukturierung der Benutzeroberfläche vorgenommen. Anders als bei klassischen Webseiten, wird diese Strukturierung allerdings in JavaScript XML (JSX), eine Syntaxerweiterung für JavaScript, vorgenommen. JSX ermöglicht es Entwicklern, HTML-artige Elemente direkt in JavaScript umzusetzen, ohne explizite HTML-Syntax zu nutzen[21].

Bei Cascading Style Sheets (CSS) handelt es sich um eine deklarative Sprache, die es ermöglicht Webseiten und deren HTML-Elemente visuell zu gestalten. Dabei gibt es verschiedene Ansätze und Möglichkeiten, wie CSS implementiert und verwaltet wird. Zum einen gibt es die klassische CSS-Variante, bei der eine separate .css-Datei erstellt wird. In dieser Datei werden alle Styles für die Webseite definiert, was bei größeren Projekten allerdings zu Problemen bei der Wartbarkeit führen kann[22]. Eine weitere Möglichkeit bieten CSS Module, die beliebig oft wiederverwendet werden können. Die dritte Möglichkeit bieten CSS Frameworks. In diesem Fall handelt es sich um vorgefertigte Komponenten und Stile, die ebenfalls wiederverwendet werden können. Auch hier gibt es wieder verschiedene Frameworks wie zum Beispiel Bootstrap, Tailwind CSS oder Foundation.

[18] **sikora_professionelles_nodate**.
[19] **autor_funktionen_nodate**.
[20] **autor_javascript_nodate**.
[21] **w3schools_react_nodate**.
[22] **autor_welche_2019**.

2.1.4 Docker

Docker ist eine Open-Source-Plattform, welche zur Entwicklung, Bereitstellung und Ausführung von Anwendungen in isolierten und sogenannten Containern dient. Ein Container wiederum basiert auf einer standardisierten Einheit von Software, der alle benötigten Abhängigkeiten, Bibliotheken und Konfigurationen in einem sogenannten Image kapselt. Dadurch wird eine unabhängige Ausführung der in dem Container befindlichen Software ermöglicht, egal in welcher Umgebung sich dieser Container gerade verbindet. Die wichtigsten Komponenten von Docker umfassen das beschriebene Docker Image, den Docker Container – einer laufenden Instanz des erstellten Imgages – und der Dockerfile bestehend aus Anweisungen zum erstellen des Docker Images[23].

2.2 Zusammenspiel der Technologien

Für eine funktionsfähige und moderne Webseite ist das Zusammenspiel der zuvor genannten Technologien unerlässlich. Für unser Projekt bilden die klare Trennung von Backend und Frontend sowie die Integration von HTML, CSS und JavaScript, das grundlegende Fundament.

Auf Grund der genannten klaren Trennung von Frontend und Backend, können die Arbeiten an den beiden Teilprojekten unabhängig voneinander und parallel stattfinden. Außerdem bewirkt die Trennung eine gewisse Flexibilität, da Änderungen in einem der beiden Bereiche, nur geringe Auswirkungen auf den anderen Bereich haben. Darüber hinaus können beide Bereiche ebenfalls unabhängig voneinander erweitert oder skaliert werden. Die Kommunikation zwischen den beiden Bereichen erfolgt über RESTFUL APIs. Diese APIs werden von Strapi im Backend automatisch generiert und von React im Frontend genutzt. Durch diese Vorgehensweise wird eine effiziente Datenverwaltung ermöglicht. Strapi verwaltet die Inhalte und React präsentiert sie. Die in Strapi gespeicherten Inhalte können dabei mehrfach von React im Frontend wiederverwendet werden. HTML per JSX, CSS und JavaScript müssen im Gesamtkontext eingegliedert werden. HTML wird in React per JSX umgesetzt, um die Arbeit für Entwickler zu vereinfachen und um die grundlegende Webseitenstruktur zu bauen. CSS setzt an dem Punkt an, an dem das Design angepasst und verfeinert wird, um eine anschauliche Benutzeroberfläche zu bauen. JavaScript ist das Bindeglied zwischen den verschiedenen Technologien. Insbesondere in Form von React, wird per JavaScript der Datenfluss und die dynamische Aktualisierung der Benutzeroberfläche logisch umgesetzt und verwaltet. Docker fungiert hier als Isolation für die ausge-

[23] **beck_lokale_2023**.

führten Komponenten. Sowohl das Backend als auch das Frontend sollen innerhalb der einzelnen Container laufen.

2.3 Vorteile der gewählten Architektur

Im Folgenden werden die Vorteile unserer gewählten Struktur und Technologien hervorgehoben.

Ein Vorteil von der von uns gewählten Struktur, ist die Modularität und Wiederverwendbarkeit. Diese wird vor allem durch die Trennung von Front- und Backend erreicht. Hinzu kommt die Verwendung einzelner Komponenten innerhalb von React, wodurch Wartung und Erweiterung der Webseite verbessert werden[24].

Darüber hinaus ist, ebenfalls durch die Trennung von Front- und Backend, aber auch durch die Nutzung von Strapi als headless CMS, eine hohe Flexibilität bei der Entwicklung gegeben. Dadurch kann jederzeit auf äußere Einflüsse wie neue Technologien oder Kundenanforderungen eingegangen werden[25].

Die Verwendung von APIs für den Datenfluss zwischen Front- und Backend, bietet den Vorteil, dass dieser effizient und standardisiert ablaufen kann. Strapi stellt die standardisierten APIs bereit und React greift darüber auf die benötigten und wiederverwendbaren Daten zu[26].

Strapi bietet zudem einen großen Vorteil in Sachen Sicherheit. Die integrierten Sicherheitsfunktionen in Strapi schützen das Backend und im Frontend können unabhängig davon weitere Sicherheitsmaßnahmen gezielt getroffen werden.

[24] **acharya_15_2023**.
[25] **autor_warum_nodate**.
[26] **autor_headless-cms_nodate**.

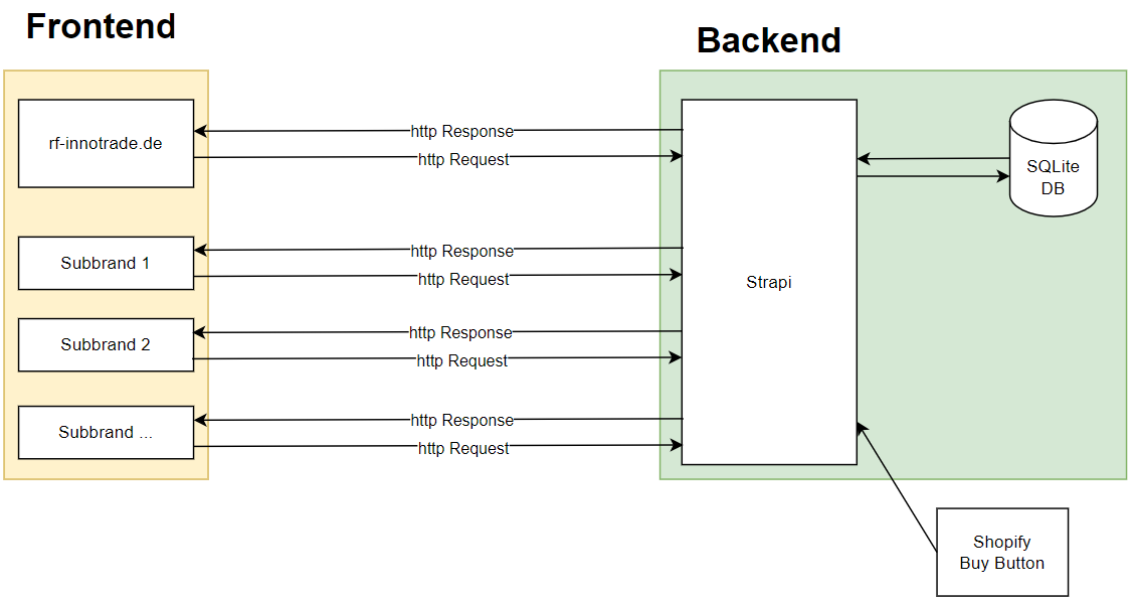
3 Hauptteil

Einleitende Worte zum Hauptteil.

3.1 Architektur

Das Architekturdiagramm zeigt eine klassische Client-Server-Architektur, die in zwei Hauptbereiche unterteilt ist: Frontend und Backend. Im Frontend-Bereich befinden sich mehrere Komponenten: die Hauptwebsite "rf-innotrade.deßowie beliebig viele Unterwebseiten (hier: Subbrands). Jede dieser Frontend-Komponenten kommuniziert mit dem Backend über HTTP-Requests und erhält entsprechende HTTP-Responses zurück. Das Backend, besteht aus einer Strapi-Anwendung als zentraler Komponente. Diese ist mit einer SQLite-Datenbank verbunden, die für die Datenspeicherung zuständig ist. Zusätzlich ist eine Shopify-Instanz eingebunden. Diese einbindung erfolgt über einen sogenannten Buy Button, welcher durch ein von Shopify selbst erzugtes Skript auf die Unterwebseiten eingebettet wird. Dieses Skript kommuniziert mit Shopify und bildet darüber diverse Shopfunktiuonen ab. Die gesamte Architektur folgt einem einheitlichen Kommunikationsmuster, bei dem alle Frontend-Komponenten über standardisierte HTTP-Verbindungen mit dem Strapi-Backend interagieren. Diese klare Trennung zwischen Frontend und Backend ermöglicht eine skalierbare und leicht zu wartende Systemarchitektur.

Abbildung 1: Architektur Schaubild, Quelle: Eigene Darstellung



3.2 Strapi als Backend

Strapi bietet von Haus aus eine gute Grundstruktur. Alle notwendigen Dateien werden von Strapi grundlegend mitgeliefert und müssen im Anschluss nur noch an die individuellen Anforderungen angepasst werden. In diesem Kapitel werden wir auf die von uns veränderten und auf unsere Bedürfnisse zugeschnittenen Dateien eingehen und deren Funktionalitäten erläutern.

3.2.1 Vite.config.js

Diese Datei ist abgelegt unter: „rfinnotrade/strapi/src/admin/vite.config.js“. Um zu erläutern, was in dieser Datei passiert, muss das grundlegende Prinzip von Cross-Origin Ressource Sharing (CORS) verstanden werden. CORS ist ein Sicherheitsmechanismus, welcher in Webbrowsern verwendet werden kann, um Zugriff auf Ressourcen von einer anderen Domain kontrolliert zulassen zu können. Daher auch der Begriff „Cross-Origin“. Es handelt sich hierbei um eine Erweiterung der standardmäßig verwendeten Same-Origin-Policy (SOP), welche Zugriffe nur von der selben Domain zulässt (Vgl. Cross-Origin Resource Sharing – Wikipedia, o. J.). In unserem Fall erlaubt diese Struktur also den Fremdzugriff auf Ressourcen innerhalb von Strapi. Diese Datei wird ebenfalls grundsätzlich von Strapi mitgeliefert und wurde von uns leicht angepasst.

Listing 1: Vite.config.js

```
const { mergeConfig } = require('vite');

module.exports = (config) => {
  // Important: always return the modified config
  return mergeConfig(config, {
    resolve: {
      alias: {
        '@': '/src',
      },
    },
    server: {
      cors: {
        origin: true,
        credentials: true,
      },
    },
  });
```

```
18 };
```

In Zeile 13 und 14 erlauben wir den Zugriff auf Strapi und legen zudem fest, dass credentials weitergeleitet werden sollen. Hier könnte man auch weitere Restriktionen vornehmen, was von uns allerdings in der Datei „Middleware.js“ vorgenommen wurde.

3.2.2 Middlewares.js

Diese Datei liegt unter „rfinnotrade/strapi/config/middleware.js“. Der Begriff Middleware in unserem Webprojekt ist als eine Softwareschicht zwischen unserer Webanwendung und dem Server zu verstehen. Die Middleware verarbeitet dabei eingehende Anfragen und ausgehende Antworten, um zum Beispiel Login- und Authentifizierungsfunktionen zu ermöglichen.

Listing 2: Middlewares.js

```
1  module.exports = [
2    'strapi::logger',
3    'strapi::errors',
4    'strapi::security',
5    {
6    name: 'strapi::cors',
7      config:
8      {
9        origin: [
10         'http://www.rf-innotrade.de',
11         'http://rf-innotrade.de',
12         'http://www.rf-innotrade.de:82',
13         'http://rf-innotrade.de:82',
14         'http://www.rf-innotrade.de:1338',
15         'http://rf-innotrade.de:1338',
16         'http://www.rf-innotrade.de:8080',
17         'http://rf-innotrade.de:8080'
18       ],
19       methods: ['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD',
             'OPTIONS'],
20       headers: ['Content-Type', 'Authorization', 'Origin', '
            Accept'],
21       keepHeaderOnError: true,
22       credentials: true,
23       maxAge: 86400,
```

```
24      },
25    },
26    'strapi::poweredBy',
27    'strapi::query',
28    'strapi::body',
29    'strapi::session',
30    'strapi::favicon',
31    'strapi::public',
32    'global::Token',
33  ];
```

Die hier gezeigte Middleware.js, arbeitet die aufgeführten Middleware-Funktionen nacheinander und in genau dieser Reihenfolge ab. Somit wird sichergestellt, dass aufeinander aufbauende Funktionen auch in der richtigen Reihenfolge ausgeführt werden. Es ist außerdem noch wichtig zu erwähnen, dass dies Strapi Middleware-Funktionen sind, da wir uns nach wie vor innerhalb unserer Strapi Konfiguration befinden. Neben der Anmeldung unter unserer rf-innotrade.de Admin Seite oder für andere API-Calls, welche nur von einem angemeldeten Admin durchgeführt werden dürfen, ist diese Datei für die im vorherigen Kapitel erwähnte CORS-Konfiguration zuständig. Diese Konfiguration sorgt dafür, dass Authentifizierungsanfragen immer nur über die in Zeile 14 bis 21 genannten Domains erlaubt sind und die Anfrage nur dann weiterverarbeitet werden darf.

3.2.3 Token.js

Die nächste Datei ist unter dem Pfad „rfinnotrade/strapi/src/middlewares/Token.js“ abgelegt. Diese Datei ist ebenfalls sehr wichtig, da Strapi von Haus aus keine Cookies verarbeiten kann. Die Verarbeitung von Cookies ist in unserem Fall notwendig, damit Informationen wie der aktuelle Login-Status oder welcher User, mit welchen Berechtigungen, startet gerade eine Anfrage an Strapi, abgefragt werden können.

Listing 3: Token.js

```
1   module.exports = () => {
2  return async (ctx, next) => {
3    const cookies = ctx.request.header.cookie || false;
4    if (cookies) {
5      let token = cookies
6        .split(";")
7        .find((c) => c.trim().startsWith("jwt="))
8        .split("=")[1];
```

```
9       if (token) {
10        ctx.request.header.authorization = `Bearer ${token}`;
11      }
12    }
13    await next();
14  };
15 };
```

Realisiert wird diese Funktion im ersten Schritt durch das Auslesen des Cookies aus dem mitgelieferten Gesamtkontext der Webseite in Zeile 2 und 3. Im nächsten Schritt wird in den Zeilen 6 bis 8 der JSON Web Token (JWT) extrahiert. Sollte die Abfrage für ein JWT erfolgreich sein, wird in Zeile 9 und 10 das Token um „Bearer“ erweitert und dem „Authorization-Header“ des Webseitenkontextes wieder hinzugefügt. Durch diese Middleware Funktion versteht Strapi, ob ein User berechtigter Weise eingeloggt ist oder Serveranfragen durchführen darf.

3.2.4 Custom.js

Die nachfolgende Datei liegt unter „rfinnotrade/strapi/src/api/custom/controllers/custom.js“ und beinhaltet verschiedene Funktionen. Im Folgenden werde ich auf die Funktionen „login“, „logout“ und „checkAuthStatus“ eingehen, da diese im Kontext von Strapi und den bisher genannten Funktionen am relevantesten sind.

Als erstes gehen wir auf die Login-Funktion ein. Diese Funktion erledigt die Aufgabe, den eigentlichen Login bei einer Login-Anfrage von der Admin Seite zu verifizieren, Daten aus Strapi abzufragen und darüber hinaus alle notwendigen Daten in den Cookies zu setzen.

Listing 4: Login-Funktion

```
1  async login(ctx) {
2      const { body } = ctx.request;
3      const hostname = "localhost";
4      const absoluteURL = `http://${hostname}:${strapi.config.
         server.port}`;
5      const sanitizeOutput = (user) => {
6        const {
7          password,
8          resetPasswordToken,
9          confirmationToken,
10         ...sanitizedUser
```

```
    } = user; // be careful, you need to omit other private attributes yourself
    return sanitizedUser;
  };
  try {
    console.log("Tryin to login");
    // Now submit the credentials to Strapi's default login endpoint
    let { data } = await axios.post(`${absoluteURL}/api/auth/local`, body);
    const populatedUser = await strapi.entityService.findOne(
      "plugin::users-permissions.user",
      data.user.id,
      {
        populate: {
          role: {
            fields: ["type"],
          },
        },
      }
    );
    data.user = sanitizeOutput(populatedUser);
    // Set the secure cookie
    if (data && data.jwt) {
      ctx.cookies.set("jwt", data.jwt, {
        httpOnly: true,
        secure: false,
        SameSite: "None",
        maxAge: 1000 * 60 * 60 * 24 * 7, // 7 Day Age
      });
    }
    // Respond with the jwt + user data, but now this response also sets the JWT as a secure cookie
    return ctx.send(data);
  } catch (error) {
    console.log("An error occurred:", error.response);
    return ctx.badRequest(null, error);
  }
},
```

Um diesen Vorgang zu realisieren, wird erneut der gesamte Webseitenkontext in Zeile 1 übergeben und daraus alle „request“-Daten in Zeile 2 geladen. Hier sind unter anderem die Eingaben des Benutzers gespeichert. Von Zeile 5 bis Zeile 13, wird die Funktion „sanitize-Output“ deklariert, welche erst später angewendet wird. Ab Zeile 14 wird versucht mithilfe der übergebenen Login-Daten, eine Anfrage an Strapis eigentliche Login API durchzuführen. Ist diese Anfrage erfolgreich, werden die entsprechenden Benutzerdaten aus der Strapi Datenbank geladen, unter anderem die Benutzerrollen abgefragt. Ist alles erledigt, wird eine Cookie mit allen notwendigen Informationen zurückgegeben und in den Webseitenkontext geladen. Somit wurde die Login Abfrage erfolgreich über eine API durchgeführt und alle notwendigen Login Informationen anschließend in den Cookies der Webseite für weiter Abfragen gespeichert. Eine automatische Logout Funktion wurde in Zeile 68 ebenfalls großzügig implementiert. Diese würde spätestens nach 7 Tagen greifen, das Cookie auslaufen und der Logout somit automatisch erfolgen.

Der ebenfalls implementierten Logout-Funktion wird zu Beginn erneut der Webseitenkontext übergeben.

Listing 5: Logout-Funktion

```
1  async logout(ctx) {
2      try {
3        // Clear the JWT cookie
4        ctx.cookies.set('jwt', null, {
5          httpOnly: true,
6          secure: false,
7          maxAge: 0,
8        });
9
10       return ctx.send({
11         message: 'Successfully logged out',
12       });
13     } catch (error) {
14       return ctx.badRequest(null, error);
15     }
16   },
```

In Zeile 4 wird dann zunächst das Cookie aus dem Kontext ausgelesen und und das JWT auf „NULL“ gesetzt, wodurch der Logout theoretisch bereits erfolgt ist. Zusätzlich wird der automatische Logout, welcher in der Login Funktion auf 7 Tage gesetzt wird, auf den Wert 0 gesetzt. Dadurch löst sich das Cookie selber auf und der Logout würde ebenfalls erfolgen. Das Cookie wird am Ende wieder an den Webseitenkontext zurückgegeben.

Die Funktion checkAuthStauts hat die Aufgabe, den aktuellen Login Status bei Aufruf dieser Funktion abzufragen. Diese Funktion ergänzt die Login- und Logout-Funktion also um eine weiter wichtige Funktionalität. Die Funktion ist ähnlich aufgebaut wie Login und Logout.

Listing 6: checkAuthStatus-Funktion

```
async checkAuthStatus(ctx) {
    // Check for JWT in cookies first
    const jwtCookie = ctx.cookies.get('jwt');

    if (!jwtCookie) {
      return ctx.unauthorized('No JWT cookie found');
    }

    try {
      const decoded = verify(jwtCookie, strapi.config.get('
          plugin::users-permissions.jwtSecret'));
      return ctx.send({ isAuthenticated: true, user: decoded });
    } catch (err) {
      return ctx.unauthorized('Invalid token');
    }
  },
```

Diese Funktion überprüft zunächst, ob überhaupt ein JWT vorhanden ist. Sollte das nicht so sein, wird die Anfrage abgelehnt, da kein aktueller Login erfolgt ist. Sollte ein JWT vorhanden sein, wird dieses erneut verifiziert und der Benutzer kann, entsprechend seinen Berechtigungen, Anfragen durchführen.

3.3 React als Frontend

React bildet das Herzstück unseres Frontends und ermöglicht die Erstellung einer dynamischen und interaktiven Benutzeroberfläche. In den folgenden Abschnitten werden wir die wichtigsten React-Komponenten unseres Projekts genauer betrachten und ihre Funktionen erläutern.

3.3.1 App.jsx Frontend

Betrachten wir als nächstes unser Frontend. Die Hauptkomponente im Frontend ist die Datei App.jsx unter „rfinnotrade/frontend/src/App.jsx“. Diese Datei organisiert die gesamte

Struktur der Webseite, steuert die Navigation und verwaltet darüber hinaus auch, in Verbindung mit den Backendkomponenten, den aktuellen Authentifizierungsstatus des Benutzers.

Listing 7: App.jsx Router

```
return (
    <Router>
      <div className="App">
        {!isOpen && <Header isOpen={isOpen} toggleSidebar={
           toggleSidebar} isAuthenticated={isAuthenticated}
           setIsAuthenticated={setIsAuthenticated} />}
        <Sidebar isOpen={isOpen} toggleSidebar={toggleSidebar}
           />
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<AboutUs />} />
          <Route path="/impressum" element={<Impressum />} />
          <Route path="/agb" element={<AGB />} />
          <Route path="/kontakt" element={<Kontakt />} />
          <Route path="/login" element={isAuthenticated ? <
             AdminCenter /> : <Login onLoginSuccess={checkAuth}
             />} />
          <Route path="/admin-center" element={isAuthenticated ?
              <AdminCenter /> : <Login onLoginSuccess={checkAuth
             } />} />
        </Routes>
        <Footer />
      </div>
    </Router>
  );
}
```

Der „Router“ Bereich wird verwendet, um die Navigation zwischen verschiedenen Seiten zu ermöglichen. Die Routes innerhalb dieser Sektion definieren die konkret verfügbaren Seiten. Außerdem ist hier die Sidebar- und Header-Steuerung zu finden. Der aktuelle Zustand der Sidebar wird über „isOpen“ gesteuert und über die „toggleSidebar“ Funktion, kann zwischen den Zuständen der Sidebar gewechselt werden.

Listing 8: App.jsx checkAuth

```
function App() {
```

```
2     const [isAuthenticated, setIsAuthenticated] = useState(false
          );
3
4     const checkAuth = async () => {
5       try {
6         const response = await axios.get(import.meta.env.
              VITE_STRAPI_BASE_URL + 'api/auth/status', {
              withCredentials: true });
7         if (response.data.isAuthenticated) {
8           setIsAuthenticated(true);
9         } else {
10          setIsAuthenticated(false);
11        }
12      } catch (err) {
13        setIsAuthenticated(false);
14      }
15    };
16
17    useEffect(() => {
18      checkAuth();
19    }, []);
20    const [isOpen, setIsOpen] = useState(false);
21
22    const toggleSidebar = () => {
23      setIsOpen(!isOpen);
24    };
```

Außerdem wird in Zeile 2 der state „isAuthenticated“, also ein veränderbarer Zustand, erzeugt und mit false initialisiert. Dazu gehört die checkAuth Funktion, welche eine Anfrage an Strapi ins Backend zur Authentifizierungsüberprüfung sendet. Diese Funktion wird bei jedem Neuladen der Seite automatisch über die ebenfalls implementiert Funktion useEffect aufgerufen. Sollte der Nutzer abgemeldet werden oder für gewisse Funktionen der Webanwendung nicht freigeschaltet sein, wird über die hier implementierten Funktionen die Authentifizierung stets überprüft und in diesem Fall auf False gesetzt. Die App.jsx Datei kommt immer beim Laden der Webseite zum Einsatz. Darüber hinaus wird sie genutzt, wenn der Benutzer zwischen verschiedenen Seiten der Webandwendung navigiert, damit die jeweiligen Komponenten auch geladen werden. Damit der Benutzer auch nur auf jene Bereiche und Komponenten zugreifen kann, auf die er auch zugreifen darf, sind die Authentifizierungsfunktionen implementiert.

3.3.2 Login.jsx

Eine Seite, bzw. Komponente ist die Login Seite, welche unter „rfinnotrade/frontend/src/-components/pages/Login.jsx“ abgelegt ist. Die Login.jsx Datei definiert die Login Seite unseres Frontends. Sie hat grundsätzlich die Funktion, den Benutzern die Anmeldung per E-Mail oder Benutzernamen mit dem dazugehörigen Passwort zu ermöglichen.

Listing 9: Login.jsx Frontendaufbau

```
return (
    <div className="login-container">
            <div className="login-header">
            <h1 className="login-title">RF InnoTrade Admin
               Center</h1>
        </div>
        <div className="login-background">
            <div className="login-box">
                <h2 className="welcome-back">Welcome Back</h2>
                <p className="instructions">Gib deine
                   Zugangsdaten ein, um Zugang zu deinem Account
                    zu erhalten</p>
                <form onSubmit={handleSubmit} className="login-
                   form">
                    <div className="form-group">
                        <input
                            type="text"
                            placeholder="Gib deine Email oder
                               Username ein"
                            value={identifier}
                            onChange={(e) => setIdentifier(e.
                               target.value)}
                            required
                            className="form-input"
                        />
                    </div>
                    <div className="form-group">
                        <input
                            type="password"
                            placeholder="Gib dein Passwort ein"
                            value={password}
                            onChange={(e) => setPassword(e.
                               target.value)}
```

```
                                required
                                className="form-input"
                            />
                        </div>
                        <button type="submit" className="login-
                            button">Anmelden</button>
                    </form>
                    <p className="reset-password">
                        Passwort vergessen?
                        {/* <Link to="/password-reset">Passwort zurü
                            cksetzen </Link> */}
                    </p>
                    {error && <p className="login-error">{error}</p
                        >}
                </div>
            </div>
        </div>
);
```

Das User-Interface (UI) ist ebenfalls hier implentiert und besteht aus einem Eingabefeld für die E-Mail und den Benutzernamen, einem Eingabefeld für das Passwort und einer Schaltfläche zum Absenden der Login-Daten. Wenn der Login fehlschlät, erscheint eine Fehlermeldung unterhalb des Formulars. Das Styling der Seite wird über eine ausgelagerte Login.css Datei gesteuert. Um den Login eines Benutzers durchzuführen, werden in der Login Komponente verschiedene Informationen verwaltet

Listing 10: Login.jsx states

```
const [password, setPassword] = useState('');
const [identifier, setIdentifier] = useState('');
const [error, setError] = useState('');
```

Wie hier zu sehen ist, verwaltet die Login Komponente password, identifiert und error, um die Logindaten oder Fehler bei der Anmeldung innerhalb der Komponente zu speichern und verarbeiten zu können. Sendet der Benutzer den Login per Schaltfläche ab, wird die Funktion „handleSubmit“ aufgerufen. Diese sendet einen POST-Request an die Login-API von Strapi, überprüft dort die Anmeldedaten und der Benutzer wird bei erfolg an die Admin-Seite der Webseite weitergeleitet.

Listing 11: Login.jsx Login-Funktion

```
try {
```

```
    const response = await axios.post(import.meta.env.
        VITE_STRAPI_BASE_URL+'api/auth/login', {
         identifier,
         password,
    }, {
         withCredentials: true // This is important for including
             cookies in the request
    });
    console.log('User profile', response.data.user);
    navigate('/Admin-Center'); // Redirect to the home page
    props.onLoginSuccess();
    } catch (error) {
    setError('Login fehlgeschlagen bitte überprüfe deine Eingabe
        .');
    }
};
```

Außerdem wird die Funktion „onLoginSuccess“ ausgeführt, damit der Authentifizierungsstatus des Benutzers aktualisiert wird.

3.3.3 Sidebar.jsx

Die Sidebar ist der Dreh- und Angelpunkt um auf der Webseite zu anderen Komponenten zu navigieren und ist unter „rfinnotrade/frontend/src/components/Sidebar.jsx“ abgelegt. Die Sidebar enthält Links zu den anderen Seiten der Anwendung und bietet eine interaktive Funktion, um ein- und ausgeblendet zu werden.

Listing 12: Sidebar.jsx

```
const Sidebar = ({ isOpen, toggleSidebar }) => {

return (
    <>
    <div className={`menu-icon ${isOpen ? 'open' : ''}`} onClick
        ={toggleSidebar}>
         <span className="menu-icon-bar"></span>
         <span className="menu-icon-bar"></span>
         <span className="menu-icon-bar"></span>
    </div>

    <div className={`sidebar ${isOpen ? 'open' : ''}`}>
```

```
12          <ul className="sidebar-links">
13          <li>
14              <Link to="/" onClick={toggleSidebar}>Home</Link>
15          </li>
16          <li>
17              <Link to="/brands" onClick={toggleSidebar}>Our
                    Brands</Link>
18          </li>
19          <li>
20              <Link to="/about" onClick={toggleSidebar}>About Us</
                    Link>
21          </li>
22          <li>
23              <Link to="/impressum" onClick={toggleSidebar}>
                    Impressum</Link>
24          </li>
25          <li>
26              <Link to="/agb" onClick={toggleSidebar}>AGB</Link>
27          </li>
28          <li>
29              <Link to="/kontakt" onClick={toggleSidebar}>Kontakt
                    </Link>
30          </li>
31
32          <div className="divider"></div>
33          <div className="login-section">
34              <ul>
35              <Link to="/admin-center" onClick={toggleSidebar}>
                    Admin Center</Link>
36              </ul>
37          </div>
38          </ul>
39      </div>
40      </>
41  );
42  };
```

Über die von der Elternkomponente „App.jsx“ übergebene isOpen-Eigenschaft, wird die sidebar grundlegend gesteuert. Die Funktion togglesidbar wird dabei jedes Mal aufgerufen, wenn der Benutzer auf das Sidebarsymbol oder einen Link innerhalb der Sidebar klickt. Mit jedem Aufruf der Funktion, wird die Sidebar dann geöffnet oder geschlossen.

3.3.4 AdminCenter

Als nächstes gehen wir auf das Admin-Center ein, welches unter „rfinnotrade/frontend/src/-components/pages/admincenter.jsx“ gespeichert ist. Die Admin-Center Komponente stellt das Hauptverwaltungscenter für die Brands dar. Hier können neue Seiten erstellt, Seiteninhalte bearbeitet und laufende Docker-Container kontrolliert werden. Auch diese Komponente nutzt wieder verschiedene States.

Listing 13: AdminCenter.jsx states

```
function AdminCenter() {
    const [brands, setBrands] = useState([]);
    const [error, setError] = useState('');
    const [isModalOpen, setIsModalOpen] = useState(false);
    const [newPage, setNewPage] = useState({});
    const [activeContainers, setActiveContainers] = useState([])
        ;
    const [pageFields, setPageFields] = useState({});
```

Brands speichert die Liste aller erstellten Brands. Der State isModalOpen steuert die Sichtbarkeit des Modals zur Erstellung von neuen Seiten und newPage speichert die Daten der neuen Seite die erstellt werden soll ab. ActiveContainers enthält die Informationen zu aktuell laufenden Docker-Containern und pageFields speichert die Felder des content-types für Seiten. Innerhalb der Komponente sind außerdem verschiedenste Funktionen implementiert. Im Folgenden gehen wir auf drei wichtige Funktionen ein.

Listing 14: fetchBrands-Funktion

```
const fetchBrands = async () => {
try {
    const response = await axios.get(
    `${import.meta.env.VITE_STRAPI_BASE_URL}api/pages?populate=
       ColorPalette`,
    { withCredentials: true }
    );
    setBrands(response.data.data);
} catch (err) {
    setError('Failed to fetch brands');
    console.error('Error fetching brands:', err);
}
};
```

Diese Funktion ruft die Liste der Brands über eine API ab und speichert diese dann im state brands. Nach dieser Vorgehensweise funktioniert auch fetchActiveContainers, zum abrufen und speichern der laufenden Container.

Listing 15: fetchActiveContainers-Funktion

```
1  const fetchActiveContainers = async () => {
2      try {
3          const response = await axios.get(
4          `${import.meta.env.VITE_STRAPI_BASE_URL}api/docker/list
               `,
5          { withCredentials: true }
6          );
7          setActiveContainers(response.data);
8      } catch (err) {
9          setError('Failed to fetch running containers');
10         console.error('Error fetching running containers:', err)
               ;
11     }
12     };
```

In Zeile 4 ist ein beispielhafter API-Endpunkt zu sehen. Das Frontend greift über diese API auf das Backend in Strapi zu und lädt von dort aus die laufenden Docker-Container.

Listing 16: fetchPageFields-Funktion

```
1  const fetchPageFields = async () => {
2      try {
3          // Fetch the main content type
4          const mainFields = await fetchContentType('api::page.
               page');
5          if (!mainFields) return;
6
7          // Process components recursively
8          const processedFields = {};
9          for (const [key, field] of Object.entries(mainFields)) {
10         if (field.type === 'component') {
11             // Fetch and process component fields
12             const componentFields = await fetchComponent(field.
                   component);
13             processedFields[key] = {
14             ...field,
15             fields: componentFields
```

```
            };
        } else {
            processedFields[key] = field;
        }
        }
```

Diese Funktion sorgt dafür, dass Felddefinitionen des Seiten-Content-Types sowie zugehörige Komponenten geladen und die Struktur der neuen Seite initialisiert werden. Dies ist also die Hauptfunktion, um die in den Feldern vom Benutzer eingetragenen Daten für die neue Seite zu übernehmen und die Initialisierung der neuen Brand zu starten. Auch im admin-center ist wieder der eigentliche Webseitenaufbau in der html-nahen jsx-Syntax zu finden.

Listing 17: AdminCenter.jsx Aufbau

```
return (
    <div className="admin-center">
        <div className="brand-management">
        <div className="subheader">
            <h2 className="center-title" > Admin Center </h2>
            <button onClick={openModal} className="add-brand-btn
               ">+ New Page</button>
        </div>

        {isModalOpen && (
            <div className="modal-overlay">
            <div className="modal-content">
                <h3>Neue Seite erstellen</h3>
                <form onSubmit={handleSubmit}>
                {Object.entries(pageFields).map(([key, field])
                   =>
                    renderField(key, field, '', newPage,
                       setNewPage, setError)
                )}
                <div className="modal-actions">
                    <button type="button" onClick={closeModal}>
                       Cancel</button>
                    <button type="submit">Seite erstellen</
                       button>
                </div>
                </form>
            </div>
```

```
            </div>
        )}

        <div className="brand-list">
            {brands.map((brand, index) => {
            const isRunning = activeContainers.some(container =>
                container.name === `subbrand-${brand.documentId
                }`);
            return (
                <BrandItem
                key={brand.id || index}
                name={brand.BrandName || `Brand ${index + 1}`}
                page={brand}
                refreshBrands={() => fetchBrands()}
                isRunning={isRunning}
                fetchActiveContainers={fetchActiveContainers}
                pageFields={pageFields}
                isEditing={activeEditBrandId === brand.
                    documentId} // Check if this brand is being
                    edited
                toggleEdit={() => toggleEditBrand(brand.
                    documentId)} // Toggle edit mode
                />
            );
            })}
        </div>
        </div>
    </div>
    );
}
```

Der Aufbau der Webseite ist dabei grundsätzlich in zwei Teile unterteilt. Die eigentliche admin-center Seite und die darin integrierte brand-list.

3.3.5 BrandItem.jsx

Der Code in dieser Datei, welche unter „rfinnotrade/frontend/src/components/BrandItems.jsx“ abgelegt ist, definiert eine weitere React-Komponente, die für die Anzeige und Verwaltung einzelner Brand- oder Seiteninformationen genutzt wird. In der Komponente werden Funktionen wie das Starten und Stoppen von Containern, das Bearbeiten von

Branddaten und das Löschen einer Brand bereitgestellt. Die Datei bietet also wichtige Verwaltungsfunktionen für den Nutzer. Eine sehr nützliche Funktion im Admin Center bietet die Editierfunktion für die einzelnen Brands, welche über den Edit Button aufgerufen werden kann. Zu Beginn des Codes wird editedPage als state definiert.

Listing 18: branditem.jsx

```
const [editedPage, setEditedPage] = useState(page);
```

Über diesen state werden Änderungen an den Daten einer Brand, wie zum Beispiel der Name oder ein Beschreibungstext, gespeichert. Sollten tatsächlich Veränderungen an einer Seite vorgenommen worden sein, wird beim Speichern der geänderte Inhalt mit dem vorherigen Inhalt verglichen und nur die veränderten Daten per API-PUT Anfrage ans Backend gesendet.

Listing 19: branditem.jsx Editierung

```
if (Object.keys(updateData).length > 0) {
    await axios.put(
        `${import.meta.env.VITE_STRAPI_BASE_URL}api/pages/${page
            .documentId}`,
        { data: updateData },
        { withCredentials: true }
    );
    toggleEdit();
    refreshBrands();
```

Im Anschluss daran wird der editedPage Status über toggleEdit verändert und über refreshBrands wird lediglich die fetchBrands Funktion, die wiederum im admin-center definiert ist, ausgeführt und somit die Brandlist aktualisiert. Eine wichtige Funktion im Code übernimmt isEditing. IsEditing steuert, ob sich die Brand-Komponente im Bearbeitungsmodus befinet oder nicht. Über diese Funktion kann also der Status des Containers abgefragt werden. Passend dazu, kann der Container über zwei Button gestartet oder gestoppt werden.

Listing 20: branditem.jsx Start/Stop Container-Funktionen

```
const handleStartContainer = async () => {
    try {
        setIsLoading(true);

        // If container info is empty or status is empty, create
            container first
```

```
        if (!containerInfo || !containerInfo.status) {
        await handleCreateContainer();
        }

        const response = await axios.post(
        `${import.meta.env.VITE_STRAPI_BASE_URL}api/docker/start`,
        {
            documentId: page.documentId,
            brandName: page.BrandName
        },
        { withCredentials: true }
        );
        if (response.data.success) {
        alert(
            `Container started successfully!\n` +
            `Name: ${response.data.containerName}\n` +
            `Port: ${response.data.port}`
        );
        fetchActiveContainers();
        }
    } catch (err) {
        setError('Failed to start container');
        console.error('Error starting container:', err);
    } finally {
        setIsLoading(false);
    }
    };

    const handleStopContainer = async () => {
    try {
        setIsLoading(true);
        const response = await axios.post(
        `${import.meta.env.VITE_STRAPI_BASE_URL}api/docker/stop`,
        {
            documentId: page.documentId,
            brandName: page.BrandName
        },
        { withCredentials: true }
        );
```

```
45          if (response.data.success) {
46          alert(`Container stopped successfully!`);
47          fetchActiveContainers();
48          }
49      } catch (err) {
50          setError('Failed to stop container');
51          console.error('Error stopping container:', err);
52      } finally {
53          setIsLoading(false);
54      }
55      };
```

Hinter den beiden Buttons, sind diese beiden Funktionen hinterlegt. Wenn der Container läuft, wird nur der Stop Container Button angezeigt und wenn der Container nicht läuft, wird nur der Start Container Button angezeigt. Der Delete Button ist ähnlich implementiert, wobei in diesem Fall nicht nur der Container gestoppt, sondern die ganze Brandpage gelöscht wird. Dies bietet dem User die einfache und schnelle Möglichkeit, die Brandpage und dessen Containerinstanz zu verwalten. Darüber hinaus ist auch eine visuelle Anzeige in die Brand List innerhalb des Admin Centers integriert. Hier wird über isRunning einfach der aktuelle Status abgefragt und für den Benutzer sichtbar gemacht.

3.3.6 FormUtils.jsx

FormUtils.jsx ist ein Hilfsmodul, das verschiedene nützliche Funktionen für den Umgang mit Formularen und den daraus resultierenden Daten bereitstellt. Sie ist unter „rfinnotrade/-frontend/src/utils/formUtils.jsx“ zu finden. Die Datei unterstützt bei der Initialisierung und Bearbeitung von Eingabefeldern. Zwei wichtige implementierte Funktionen sind fetchContentType und fetchComponent.

Listing 21: formutils.jsx fetch-Funktionen

```
1  export const fetchContentType = async (contentType) => {
2      try {
3          const response = await axios.get(
4          `${import.meta.env.VITE_STRAPI_BASE_URL}api/content-type
               -builder/content-types/${contentType}?populate=*`,
5          { withCredentials: true }
6          );
7          return response.data.data.schema.attributes;
8      } catch (err) {
```

```
        console.error(`Error fetching content type ${contentType
            }:`, err);
        return null;
    }
    };

    export const fetchComponent = async (componentName) => {
    try {
        const response = await axios.get(
        `${import.meta.env.VITE_STRAPI_BASE_URL}api/content-type
            -builder/components/${componentName}?populate=*`,
        { withCredentials: true }
        );
        return response.data.data.schema.attributes;
    } catch (err) {
        console.error(`Error fetching component ${componentName
            }:`, err);
        return null;
    }
    };
```

Sie sind darauf ausgelegt, die Datenstrukturen für Formulare im Backend per API abzurufen, und mithilfe der dort hinterlegten Strukturen dynamische Formulare zu ermöglichen. Die dafür genutzte API ist die Content-Type-Builder API von Strapi. Anknüpfend an diese beiden Funktionen, gibt es die Funktion renderField.

Listing 22: formutils.jsx renderField-Funktion

```
export const renderField = (key, field, parentPath = '', newPage
    , setNewPage, setError, isEditing = false) => {
    const fieldPath = parentPath ? `${parentPath}.${key}` : key;

    if (field.type === 'component') {
        return (
        <div key={fieldPath} className="component-group">
            <h4>{field.displayName || key}</h4>
            {Object.entries(field.fields).map(([subKey, subField
                ]) =>
            renderField(subKey, subField, fieldPath, newPage,
                setNewPage, setError, isEditing)
            )}
        </div>
```

```
        );
    }
```

Diese Funktion nutzt die geladenen Strukturen und erstellt dann tatsächlich und basierend auf den geladenen Strukturen die Eingabefelder im Formular. RenderFields wird zum Beispiel in BrandItem.jsx verwendet. Außerdem wird in dieser Datei der Upload von Dateien über ein Formular ermöglicht.

Listing 23: formutils.jsx Datei-Upload-Funktion

```
export const handleFileChange = async (e, fieldName, setNewPage,
    setError) => {
    const file = e.target.files[0];
    if (!file) return;

    try {
        const formData = new FormData();
        formData.append('files', file);

        const uploadResponse = await axios.post(
        `${import.meta.env.VITE_STRAPI_BASE_URL}api/upload`,
        formData,
        {
            withCredentials: true,
            headers: {
            'Content-Type': 'multipart/form-data',
            },
        }
        );
```

Auch hier erfolgt wieder die Kommunikation mit Strapi. Dieses Mal über die Strapi-Upload-API. Ebenso wird hier eine Funktion zur Initialisierung von vollständig neuen Seiten, bzw. Brands realisiert.

Listing 24: formutils.jsx Neue Seite initialisieren

```
export const initializeNewPage = (fields) => {
    const initialPage = {};
    for (const [key, field] of Object.entries(fields)) {
        if (field.required) {
        if (field.type === 'component') {
            initialPage[key] = initializeNewPage(field.fields);
        } else {
```

```
8               switch (field.type) {
9               case 'boolean':
10                  initialPage[key] = field.default || false;
11                  break;
12              case 'media':
13                  initialPage[key] = null;
14                  break;
15              case 'text':
16              case 'string':
17              case 'uid':
18                  if (field.regex === '^#?([A-Fa-f0-9]{6}|[A-Fa-f0
                        -9]{3})$') {
19                  initialPage[key] = field.default || '#f7f7f7';
20                  } else {
21                  initialPage[key] = '';
22                  }
23                  break;
24              default:
25                  initialPage[key] = null;
26              }
27          }
28          }
29      }
30      return initialPage;
31      };
```

Mithilfe dieser Funktion wird ein neues Objekt für eine Seite erstellt, das auf den Felddefinitionen aus der Strapi-API basiert. Diese Felder sind erstmal inhaltslos und werden später durch die Eingaben des Users mit Inhalt gefüllt.

3.4 Subbrands

Im Folgenden gehen wir auf den Subbrand Bereich ein. In diesem Kapitel geht es konkret um die jeweils erstellten Subbrandseiten, wobei wir hier auch teilweise auf Frontend- und Stylingelemente eingehen.

3.4.1 App.jsx Subbrands

Diese Datei, abgelegt unter „rfinnotrade/subbrand/src/App.jsx“ ist die zentrale Komponente des Subbrand Bereichs. Sie dient als Einstiegspunkt für React, definiert das Grundgerüst und steuert das Laden der dynamischen Inhalte. Sie ist also zum Beispiel für das Laden von Farben, Schriftarten oder Beschreibungen der einzelnen Brands verwantwortlich.

Listing 25: API.jsx

```
const STRAPI_BASE_URL = import.meta.env.VITE_STRAPI_BASE_URL;
const documentId = import.meta.env.VITE_DOCUMENTID

export const fetchPageByContentId = async () => {
    try {
    const response = await fetch(`${STRAPI_BASE_URL}/api/pages/$
       {documentId}?populate=*`);
    const result = await response.json();
    console.log(result)
    return result.data; // Assuming contentId is unique
    } catch (error) {
    console.error('Error fetching page data:', error);
    throw error;
    }
};
```

Die Funktion fetchPageByContentID ist aus der Datei API.jsx im selben Ordner importiert und dient zum Laden der gewünschten Seiteninhalte per API.

Listing 26: App.jsx state

```
try {
    const data = await fetchPageByContentId();
    setPageData(data);
```

Im Anschluss daran werden die Daten der Seite in den state pageData geladen.

Listing 27: App.jsx Imports

```
import Header from './components/HeaderBrands';
import ShopifyCollection from './components/ShopifyCollection';
import AboutUs from './components/AboutUs';
import Footer from './components/FooterBrands';
import Home from './pages/Home';
```

Außerdem werden die einzelnen Komponenten der Seite ebenfalls hier importiert. Sie sind der eigentliche Bestandteil der Subbrands und werden hier wieder vereint und strukturiert. Wenn man sich nun den gesamten useEffect-Hook anschaut, kann man darüber hinaus die dynamische Integration der CSS-Inhalte sehen.

Listing 28: App.jsx useEffect-Funktion

```
1  useEffect(() => {
2      const loadPage = async () => {
3          try {
4          const data = await fetchPageByContentId();
5          setPageData(data);
6          // Dynamically set CSS variables
7          if (data?.ColorPalette) {
8              const { primarycolor, secondarycolor, tertiarycolor
                   } = data.ColorPalette;
9              document.documentElement.style.setProperty('--
                   primary-color', primarycolor);
10             document.documentElement.style.setProperty('--
                   secondary-color', secondarycolor);
11             document.documentElement.style.setProperty('--
                   tertiary-color', tertiarycolor);
12         }
13         document.documentElement.style.setProperty('--font',
               data?.Font || '');
14         document.documentElement.style.setProperty('--font-url',
               data?.FontURL || '');
15
16         setLoading(false);
17         } catch (error) {
18         console.error("Failed to fetch page", error);
19         setLoading(false);
20         }
21     };
```

Hier zu sehen in Zeile 9 bis 11. Dies ermöglicht, dass jede Subbrand ihre eigenen Gestaltungsmöglichkeiten erhält. Damit Inhalte der Seite erst dann angezeigt werden, wenn alle Daten vollständig geladen sind, gibt es den loading state und dessen setLoading Funktion.

Listing 29: App.jsx setLoading

```
1  setLoading(false);
2  } catch (error) {
```

```
    console.error("Failed to fetch page", error);
    setLoading(false);
}
};
```

Mithilfe dieses states und der zugehörigen Funktion, kann der Ladestatus jederzeit abgefragt und der Status, wie im Beispiel zu sehen ist, neu gesetzt werden. Der Router in der Datei verwaltet wiederum, wie schon zuvor in den anderen Bereichen unseres Projekts, die Navigation zwischen den verschiedenen Bereichen einer Subbrand und wie diese erreichbar sind.

Listing 30: App.jsx Router

```
<Router>
<div className="App">
    <Header />
    <Routes>
    <Route path="/" element={<Home pageData={pageData} />} />
    <Route path="/about" element={<AboutUs />} />
    </Routes>
    <Footer />
</div>
</Router>
```

3.4.2 Home.jsx

Die Home.jsx Datei, abgespeichert unter „rfinnotrade/subbrand/src/pages/Home.jsx“, definiert die Haupt- und Startseite der Subbrands. Sie kombiniert wieder verschiedene visuelle Komponenten, um den Inhalt und das gewünschte Layout der Hauptseite darzustellen.

Listing 31: Home.jsx

```
import HeroSection from '../components/HeroSection';
import WavyBanner from '../components/WavyBanner';
import StraightBanner from '../components/StraightBanner';
import HighlightSection from '../components/HighlightSection';
import ShopifyCollection from '../components/ShopifyCollection';
import '../stylesheets/Home.css';

const Home = ({ pageData }) => {
    return (
```

```
    <div className="App" >
        <div className='homecontainer'>
        <HeroSection />
        {pageData?.Banner1 === 'true' && (
            <StraightBanner />
        )}
        {pageData?.Banner2 === 'true' && (
            <WavyBanner />
        )}
        <div className='shopify-collection'>
            <ShopifyCollection collectionId="646106448221"
               domain="1jc1dr-i4.myshopify.com"
               storefrontAccessToken="20858
               f71b76e268f82b0f33ee6773838" />
        </div>
        <HighlightSection />
        </div>
    </div>
    )
}

export default Home;
```

Für unsere Subbrands sind die Komponenten HeroSection, WavyBanner, StraightBanner und eine HighlightSection integriert.

3.4.3 HeroSection.jsx

Die HeroSection ist eine Komponente, abgelegt unter „rfinnotrade/subbrand/src/components/HeroSection.jsx“, welche als ein sinnvolles Beispiel einer Komponente im Folgenden beschrieben wird. In der Datei werden zunächst die notwendigen Daten mithilfe der fetchPageByContentId-Funktion geladen.

Listing 32: HeroSection.jsx

```
useEffect(() => {
    const loadPage = async () => {
        try {
        const data = await fetchPageByContentId();
        setPageData(data);
        setLoading(false);
```

```
7           } catch (error) {
8           console.error("Failed to fetch page", error);
9           setLoading(false);
10          }
11      };
```

Dieser typische API-Aufruf kann grundsätzlich in allen Komponenten vorgenommen werden, um gewünschte Daten aus dem Strapi Backend zu laden. Bis alle Daten geladen sind, wird ein Ladezustand ausgegeben.

Listing 33: HeroSection.jsx

```
1  return (
2      <div className="hero-section">
3          {loading ? (
4          <h1>Loading...</h1>
5          ) : (
6          <>
7          {pageData?.HeroPicture && <img src={`${import.meta.env.
               VITE_STRAPI_BASE_URL}${pageData?.HeroPicture?.url}`}
               alt="HeaderBild" className="hero-image" />}
8          </>
9          )}
10     </div>
11     );
12 };
```

Der Ladezustand wird angezeigt, bis alle Inhalte erfolgreich geladen wurden. Erst dann werden die Inhalte gerendert und angezeigt.

3.5 Design

3.6 Docker

Für Docker und zur Erstellung von Containern ist jeweils im Frontend, Backend und in Subbrands eine Dockerfile geschrieben, über die Docker Images erstellt werden. Alle einzelnen Komponenten laufen in einem eigens kreierten Container. Strapi im Backend, das admin Center im Frontend und auch die einzelnen Subbrands werden in einem eigenen Docker Container gestartet. Die Container für Strapi und das Admin Center laufen jeweils

automatisch und dauerhaft. Lediglich die Subbrands können über das Admin Center eigenhändig gestartet und gestoppt werden, wobei der initialie Start des Containers einer Subband immer manuell ausgeführt werden muss.

Listing 34: Middlewares.js

```
# Use the official Node.js image for building the React app
FROM node:alpine AS build

# Set the working directory for the build
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies with increased memory limit
RUN node --max-old-space-size=2048 $(which npm) install

# Copy the rest of the application code
COPY . .
ARG VITE_DOCUMENTID
ENV VITE_DOCUMENTID=${VITE_DOCUMENTID}
ENV VITE_STRAPI_BASE_URL=http://www.rf-innotrade.de:1337
# Build the React app
RUN npm run build

# Use the official Nginx image for serving static files
FROM nginx:alpine

# Set the working directory in the container
WORKDIR /usr/share/nginx/html

# Remove the default Nginx static assets
RUN rm -rf ./*

# Copy the prebuilt React files from the build stage
COPY --from=build /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Expose port 80
EXPOSE 80

```

```
37 # Start Nginx in the foreground
38 CMD ["nginx", "-g", "daemon off;"]
```

Die Dockerfile ist in einem zweistufigen Build- und Deployment-Prozess aufgebaut. Der erste Prozess ist die Build-Phase, in der ein Node.js-Image genutzt wird, um die React-App zu bauen. Darüber hinaus werden alle notwendigen Abhängigkeiten installiert und vor-allem werden die dynamischen Umgebungsvariablen gesetzt, um spezifische Subbrand-daten wie Ids, oder API-URLs zu definieren. Am Ende wird die React-App gebaut und im Ordner dist abgelegt. Der zweite Prozess ist die Deployment-Phase. Hier wird Nginx als Webserver genutzt und alle generierten Dateien aus der Build-Phase in den Nginx-Container kopiert. Zuletzt wird der Container bereitgestellt und über Port 80 erreichbar gemacht. Mithilfe dieser Vorgehensweise werden die einzelnen Subbrands in Container gekapselt und können trotzdem über API-Aufrufe verändert und verwaltet werden.

4 Fazit

Hier kommt das Fazit hin.

4.1 Ausblick

Durch die zeitlich begrenzte Natur dieses Projekts war es nicht möglich, alle gewünschten Funktionen im Rahmen des Projekts umzusetzen. Dennoch möchten wir drei Erweiterungen für ein zukünftiges Projekt hier erwähnen.

Monitoring

Zu einer Webseite, insbesondere einem Webshop, fallen viele nützliche Informationen an. Diese können aus unterschiedlichen Quellen kommen, wie z.B. aus Serverlogs oder von Shopify (Warenkorbinformationen). Da das Admin Center bereits eine zentrale Stelle für die Verwaltung aller Webseiten darstellt, ist es durchaus sinnvoll, alle Traffic-Informationen zu den Webseiten auch dort zentral abzubilden. Darüber hinaus könnte man dort noch weitere, typische Nutzerinteraktions-Metriken abbilden, wie z.B. die Absprungrate, die durchschnittliche Verweildauer oder die Scroll-Rate. Außerdem wäre es auch durchaus sinnvoll, weitere technische und performancerelevante Metriken abzubilden.

Upgrade einer Webseite

Das hier vorgestellte Konzept basiert darauf, möglichst schnell neue Webseiten bereitzustellen. Diese damit erstellten Webseiten sind im ersten Schritt immer erstmal Unterseiten der Hauptseite www.rf-innotrade.de, passend zur geschäftlichen Abbildung der Marken (welche als nicht eigenständige Entitäten unterhalb der RF InnoTrade angesiedelt sind). Sollte eine dieser neu erstellten Marken Erfolg haben und eine bestimmte Umsatzgrenze überschreiten, wird diese zu einer geschäftlich und rechtlich eigenständigen Entität umgewandelt. Dieser Schritt soll dann auch im Admin Center durchführbar sein in Form eines „Upgrades". Damit einher geht dann zum Beispiel das Vergeben einer komplett eigenständigen URL und, je nach Traffic auf der Webseite, auch das Auslagern des Containers auf einen eigenen Server.

Integration von weiteren Systemen

Um den Admin Center von RF InnoTrade noch umfassender bzw. vielseitiger zu gestalten, ist es eine Überlegung wert, in gewissem Maße andere benutzte Systeme mit in den Admin Center einzubinden. Das kann entweder durch eine simple Verknüpfung einer URL als

Absprungpunkt in ein anderes System abgebildet werden (bspw. Absprung in ein Buchhaltungsprogramm) oder durch die tatsächliche Integration von Funktionen aus anderen Systemen. Ein Beispiel für Letzteres ist die Nutzung der Shopify API. Hiermit könnten beispielsweise die pro Marke vertriebenen Produkte direkt im Admin Center angezeigt werden mit der Option, kleinere Anpassungen an den Produkten vorzunehmen (z.B. Änderung des Preises). Auch das zentrale Steuern von Versandoptionen, Rabattaktionen usw. könnte dann über den Admin Center erfolgen.

Appendices

Anhang 1: Literature appendices

Together with the paper, the following literature appendices are submitted:

- Sunarto et al. - 2023 - A Systematic Review of WebAssembly VS Javascript P.pdf
- De Macedo et al. - 2022 - WebAssembly versus JavaScript Energy and Runtime.pdf
- javascript-issues-and-solutions.html
- 10-most-common-javascript-mistakes.html
- what-to-understand-callback-and-callback-hell-in-javascript.html
- introduction.html
- what-is-the-dom-explained-in-plain-english.html
- vanilla-javascript-libraries-quest-stateful-dom-rendering.html
- javascript-performance.html
- js_performance.html
- was-ist-ein-assembler-a-756636.html
- using-git-and-github-for-latex-writting.html
- a-git-workflow-for-writing-papers-in-latex-4cfb31be4b06.html
- _Einleitung.html
- ITInfraPaper.html
- WASMvsJS.html
- webassembly-solving-performance-problems.html
- experimental-design.html
- hitlist.html
- experiment.html
- Sieve_of_Eratosthenes.html

- webassembly-mehr-als-nur-ein-web-standard.html
- was-ist-webassembly-a-4243591d831cc5a12d425ede224f5e5b.html
- WebAssembly-Wasm.html
- javascript-vs-webassembly.html
- downsides-of-rust-programming-language.html
- rust-worlds-fastest-growing-programming-language.html
- 2023.html
- was-ist-ein-compiler.html
- was-ist-ein-interpreter.html
- Concepts.html
- FOM-LaTeX-Template.html
- Rust_to_wasm.html

They can also be found in the literature appendices folder of the GitHub repository.

Anhang 2: GitHub Repositories

Here are the links to the GitHub repositories that contain the code and data from the LaTeXProject used in this paper:

- ITInfraPaper
- WASMvsJS

Declaration of Authenticity

We hereby declare that we produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that we have marked as quotations all passages which are reproduced verbatim or near-verbatim from publications. Further, we declare that this thesis has never been submitted before to any examination board in either its present form or in any other similar version. we herewith disagree that this thesis may be published. We herewith consent that this thesis may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this thesis for the purpose of submitting it to plagiarism detection systems is not a form of publication.