Konzeption und Prototyp eines Storefrontsystems mit Plugin- und Theme-Architektur

Ablösung des Abacus E-Commerce-Systems AbaShop

Bachelorarbeit

Frühjahrssemester 2024

Autoren: Ramon Ebneter
Gian-Luca Vogel

Betreuer: Stefan Kapferer

Experte: Roman Blum

Gegenleser: Nikolaus Heners

Industriepartner: Customize AG
Neuwiesenstrasse 20
8400 Winterthur

Inhaltsverzeichnis

Abstract **iii**

1 Management Summary **1**

1.1 Ausgangslage . . . 1
1.2 Vorgehen . . . 2
1.3 Ergebnis . . . 2
1.4 Ausblick . . . 3

2 Kontext **4**

2.1 Ausgangslage . . . 5
2.2 Vision . . . 6
2.3 Dokumentation . . . 8
2.4 Abacus . . . 9

3 Anforderungen **14**

3.1 Vorgehen . . . 15
3.2 Akteure . . . 15
3.3 Kernfunktionalitäten . . . 16
3.4 Konfiguration . . . 20
3.5 Theming System . . . 24
3.6 Plugin System . . . 28
3.7 Headless Nutzung . . . 34
3.8 Nichtfunktionale Anforderungen . . . 34

4 Verwandte Arbeiten **37**

4.1 Plugin . . . 37
4.2 Theme . . . 40

5 Design und Implementation **43**

5.1 Schnittstelle nopCommerce . . . 44
5.2 Erweiterungen nopCommerce . . . 49
5.3 Auswahl Framework . . . 59
5.4 Dynamisches Laden und Teilen von Source-Code . . . 63
5.5 Aufbau des Monorepos . . . 71
5.6 Theming System . . . 72

5.7 Plugin System . . . 85
5.8 Architektur des Storefrontsystems . . . 98
5.9 Migration und Entwicklung . . . 99

6 Ergebnisdiskussion **103**
6.1 Kritische Erfolgsfaktoren . . . 103
6.2 Bewertung der funktionalen Anforderungen . . . 105
6.3 Bewertung der definierten NFAs . . . 109
6.4 Optimierungen . . . 113
6.5 Abschluss und Ausblick . . . 117

A Benutzertest **119**
A.1 Einleitung . . . 120
A.2 Ausgangslage . . . 120
A.3 Testaufbau und Ablauf . . . 121
A.4 Aufgaben . . . 121
A.5 Bewertung . . . 126
A.6 Massnahmen . . . 127

B Developer Dokumentation **128**
B.1 Erste Schritte . . . 130
B.2 Deployment nopCommerce . . . 134
B.3 Deployment Storefrontsystem . . . 142
B.4 Theme Entwicklung . . . 144
B.5 Plugin Entwicklung . . . 153

C Aufgabenstellung **158**
C.1 Ausgangslage . . . 159
C.2 Ziele der Arbeit und Liefergegenstände . . . 159

Literaturverzeichnis **161**

Abkürzungsverzeichnis **165**

Abbildungsverzeichnis **166**

Tabellenverzeichnis **168**

Abstract

Die Abacus Research AG hat im Rahmen ihrer ERP-Lösung 'Abacus' die E-Commerce-Applikation 'AbaShop' entwickelt, deren Wartung Ende 2025 eingestellt wird. Aus diesem Grund muss die Customize AG, ein Vertriebspartner von Abacus, eine Nachfolgelösung für ihre Shop-Kunden finden. In einer vorangegangenen Studienarbeit wurde das E-Commerce-System 'nopCommerce' evaluiert und über das entwickelte Plugin 'Abacus Connector' an das ERP angebunden. Um die individuellen Anforderungen der breiten Kundschaft mit einem System bedienen zu können, soll ein Storefrontsystem entwickelt werden, das die Services von nopCommerce nutzt und den Consultants von Customize vielfältige Möglichkeiten zur Konfiguration des Webshop-Frontends bietet.

In dieser Bachelorarbeit wurden gemeinsam mit den Consultants die Anforderungen an die Konfigurationsmöglichkeiten des Storefrontsystems anhand von Kunden-Use-Cases erfasst. Die Implementierung eines Prototyps zeigt, wie individuelle Anforderungen durch die erhobenen Use-Cases abgedeckt werden können. Durch einen Benutzertest wurde sichergestellt, dass die Consultants die Kundenanforderungen über die Konfigurationsmöglichkeiten erfüllen können. Die Entwicklung eines Migrationspfades zeigt auf, wie Kunden von 'AbaShop' auf das neue Storefrontsystem migrieren können.

Die Implementierung einer RESTful HTTP-Schnittstelle als Plugin für nopCommerce ermöglicht einen 'headless' Zugriff auf dessen Services. Der entwickelte Storefront Prototyp bietet ein Kernsystem mit Standard-Webshop-Funktionalitäten für alle Kunden. Die Integration einer Plugin- und Theme-Architektur ermöglicht umfangreiche kundenspezifische Anpassungen des Kernsystems. Themes können das visuelle Erscheinungsbild des Storefronts komplett verändern, während Plugins eigene Seiten und Logik hinzufügen. Die Entwicklung von zwei Themes und mehreren Beispiel-Plugins zeigt, wie Consultants das Storefrontsystem individualisieren können. Darüber hinaus ermöglicht ein Admin-Bereich umfassende Konfigurationen, um einfache Kundenanforderungen schnell umzusetzen, ohne ein Theme oder Plugin entwickeln zu müssen.

Kapitel 1

Management Summary

1.1 Ausgangslage

Im Abacus ERP, nachfolgend als Abacus bezeichnet, wird eine E-Commerce-Lösung namens 'AbaShop' angeboten, die vom Softwarehersteller des ERP-Systems entwickelt und gewartet wird. Mit der Ankündigung des Softwareherstellers, dass die Weiterentwicklung und Wartung von 'AbaShop' Ende 2025 eingestellt wird, steht die Customize AG, der Industriepartner dieser Arbeit, vor der Herausforderung, sowohl für bestehende als auch zukünftige Shop-Kunden eine geeignete Nachfolgelösung zu finden. Als eine mögliche Alternative bietet Abacus eine neue RESTful HTTP-Schnittstelle an, welche die Entwicklung einer integrierten E-Commerce-Lösung als Nachfolger von 'AbaShop' ermöglicht.

In der vorangegangen Studienarbeit [8] von Ramon Ebneter wurde die Machbarkeit einer integrierten Webshoplösung auf Basis der neuen Schnittstellen von Abacus untersucht. Darüber hinaus wurden bestehende E-Commerce-Systeme evaluiert, um zu entscheiden, ob ein komplett eigenständiges System entwickelt oder ein existierendes E-Commerce-System angebunden wird. Die Wahl fiel auf das System 'nopCommerce', das über die RESTful HTTP-Schnittstelle mit Abacus verbunden wurde. Zu diesem Zweck wurde ein Plugin namens 'Abacus Connector' entwickelt, das beide Systeme über einen bidirektionalen Datenaustausch verbindet.

Das in der Studienarbeit verwendete E-Commerce-System bietet ein Standard-Frontend, welches jedoch nicht alle Anfroderungen der Kunden des Industriepartners erfüllt. Ziel ist es daher, nopCommerce zur Bereitstellung von E-Commerce-Kernfunktionalitäten über eine Web-Schnittstelle zu nutzen. In dieser Bachelorarbeit soll ein Prototyp eines Storefrontsystems entwickelt und über eine Web-Schnittstelle an nopCommerce angebunden werden. Dieses Storefront soll ein umfangreich konfigurierbares Frontend für eine Webshop-Lösung mit einem integrierten Adminbereich anbieten. Dies ist notwendig, um die vielfältigen Kunden der Customize AG mit einem System bedienen zu können, während den Consultants gleichzeitig ermöglicht wird, die unterschiedlichen Anforderungen an das visuelle Erscheinungsbild und die Funktionalität der Webshops ihrer Kunden zu entwickeln.

1.2 Vorgehen

In dieser Bachelorarbeit wurden die Anforderungen an ein Theming- und Plugin-System für das Storefrontsystem mit drei Consultants erarbeitet. Anhand von fünf Customize-Kunden mit komplexen Anforderungen wurde untersucht, welche speziellen Anforderungen diese Kunden haben, die zukünftig über das geplante Theming- und Plugin-System von den Consultants entwickelt werden können. Darauf aufbauend wurden allgemeingültige Use Cases definiert, die entweder in einem Theme oder Plugin umgesetzt werden.

Um das Storefrontsystem an nopCommerce anzubinden, wurden geeignete Schnittstellentechnologien evaluiert und prototypisch mit dem ausgewählten Framework für das Storefrontsystem, Next.js, implementiert. Es wurde beschlossen, eine RESTful HTTP-Schnittstelle als zusätzliches Plugin für nopCommerce zu entwickeln. Der Schwerpunkt dieser Arbeit lag in der Entwicklung eines Prototypen des Storefrontsystems, der die umfangreichen Konfigurationsmöglichkeiten über ein Theme oder Plugin demonstriert. Ein Benutzertest wurde mit einem Consultant der Customize durchgeführt, um sicherzustellen, dass das System in der Lage ist, mögliche Anforderungen eines Kunden über ein Theme oder Plugin abzudecken. Zusätzlich dokumentiert diese Arbeit, wie die Migration eines bestehenden Kunden von 'AbaShop' auf das neue Storefrontsystem erfolgen kann. Dies wird durch eine umfangreiche Entwicklerdokumentation unterstützt, die als gehostete Webseite zur Verfügung gestellt wird.

1.3 Ergebnis

Die vorliegende Arbeit und der entwickelte Prototyp zeigen, dass den Consultants mit dem Plugin- und Theming-System ein umfassendes Werkzeug zur Verfügung steht, um die vielfältigen Spezialanforderungen ihrer Kunden zu erfüllen. Diese Systeme ermöglichen es, allen Kunden ein identisches Kernsystem zur Verfügung zu stellen, wodurch eine kontinuierliche Updatefähigkeit des Storefrontsystems gewährleistet ist. Der durchgeführte Benutzertest hat gezeigt, dass das Theming-System aufgrund der Erfahrungen mit 'AbaShop' für die Consultants besonders intuitiv gestaltet wurde und sich als wertvolles Werkzeug für die laufende Anpassung an die Kundenwünsche erweist. Der im Kernsystem integrierte Administrationsbereich beinhaltet zusätzlich einen Administrationsbereich, der eine umfassende Konfiguration ohne das Schreiben von Source-Code ermöglicht. Der empfohlene Migrationsprozess zeigt, dass die Stammdaten eines bestehenden Kunden automatisiert übernommen werden können. Eine automatische Migration der Themes ist nicht möglich, da 'AbaShop' und das Storefrontsystem technologisch unterschiedlich aufgebaut sind.

1.4 Ausblick

Die neu entwickelte RESTful HTTP-Schnittstelle für nopCommerce eröffnet die Möglichkeit, dieses System zukünftig als zentrale Schnittstelle für neue Systeme zu nutzen. Der entwickelte Prototyp des Storefrontsystems hat einen ausgereiften Entwicklungsstand erreicht, so dass im nächsten Schritt ein Pilotkunde das neue Storefrontsystem im Rahmen eines Proof of Concept einführen kann. Auf Basis dieser Pilotimplementierung plant die Customize AG zu entscheiden, ob das aus der Studienarbeit und dieser Bachelorarbeit entwickelte Gesamtsystem für alle Kunden implementiert werden soll.

Kapitel 2

Kontext

Industriepartner dieser Bachelorarbeit ist die Customize AG, welche als Vertriebspartner des Softwareherstellers Abacus Research AG auftritt. Die Abacus Research AG hat bisher im Rahmen ihrer Enterprise-Resource-Planning (ERP)-Lösung 'Abacus' eine selbstentwickelte E-Commerce-Applikation namens 'AbaShop' im Einsatz. Damit konnte die Customize ihren Kunden eine individuell auf die Anforderungen zugeschnittene und in das ERP integrierte Webshop-Lösung anbieten. Da die Wartung dieser Shop-Lösung per Ende des Jahres 2025 eingestellt wird, steht die Customize vor der Aufgabe eine Nachfolgelösung für ihre bestehenden und zukünftigen Shop-Kunden zu finden. Um dies zu ermöglichen, stellt die Abacus eine neue RESTful HTTP-Schnittstelle zur Verfügung, über welche Kunden in Zukunft ihre eigenen Shop-Lösungen an das ERP anbinden können.

In der Lösung 'AbaShop' stellt Abacus dem Kunden auf ihrer Plattform Abacuscity einen Server zur Verfügung, der ausserhalb des Abacus ERP liegt und unabhängig agiert. Mittels eines Publikationsmechanismus werden in definierten Intervallen geänderte Stammdaten vom Abacus ERP an Abacuscity gesendet. In Abacuscity werden diese Stammdaten redundant in einer zusätzlichen Datenbank gespeichert. Ebenso holt das Abacus ERP in festgelegten Intervallen die neu eingetroffenen Bestellungen von Abacuscity ab und integriert diese in das ERP-System. Auf einem File Transfer Protocol (FTP)-Server[1] von Abacuscity befinden sich Jakarta Server Pages (JSP)-Templates[2], die es dem Industriepartner ermöglichen, das 'AbaShop'-System an die individuellen Kundenbedürfnisse anzupassen und neue Funktionalitäten einzufügen. Abacuscity bietet dabei eine dokumentierte Schnittstelle, die über JSP genutzt werden kann. Der zentrale Unterschied bei Einsatz der neuen RESTful HTTP-Schnittstellen liegt darin, dass das Abacus ERP nicht mehr aktiv Änderungen an das E-Commerce-System übermittelt, sondern das System die Daten selbstständig vom ERP abrufen muss.

[1]Ein FTP-Server ist ein Netzwerkserver, der das File Transfer Protocol nutzt, um den Austausch von Dateien zwischen Computern über ein TCP/IP-Netzwerk zu ermöglichen.

[2]JSP-Templates sind Vorlagen in Jakarta Server Pages, die es Entwicklern ermöglichen, dynamischen Inhalt in Webseiten zu integrieren, indem sie Java-Code in HTML einbetten.

2.1 Ausgangslage

In der vorangegangenen Studienarbeit von Ramon Ebneter [8] wurde eine Analyse verschiedener existierender E-Commerce-Frameworks durchgeführt. Die Auswahl fiel auf nopCommerce, ein im .NET-Ökosystem entwickeltes System [35]. Für die Anbindung an Abacus wurde ein Connector entwickelt, der den automatisierten Austausch von Stammdaten zwischen dem ERP und dem E-Commerce-System ermöglicht. Der Connector, genannt Abacus Connector, wurde als Plugin für nopCommerce entwickelt. Abbildung 2.1 zeigt in einem C4-Kontextdiagramm die aus der Studienarbeit resultierende Architektur.

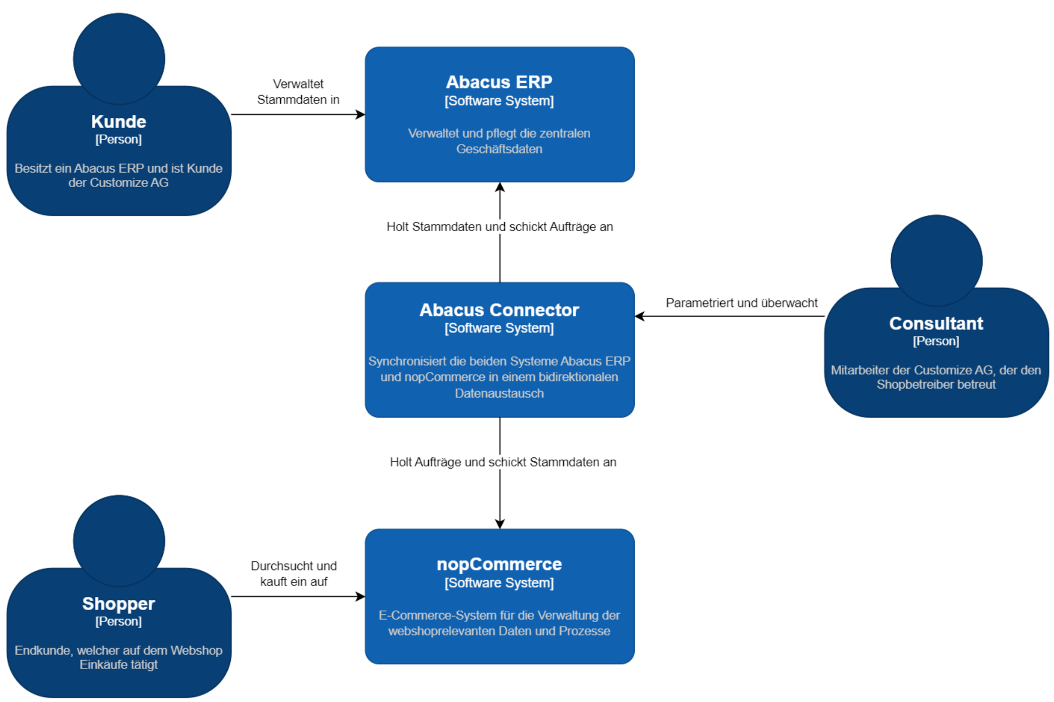


Abbildung 2.1: Kontextdiagramm Studienarbeit (C4) [8]

Das evaluierte System, nopCommerce, bietet standardmässig ein Storefrontsystem an. Im Kontext dieser Arbeit bezeichnet ein Storefrontsystem den Teil eines E-Commerce-Systems, den die Endkunden (Shopper) verwenden, oder anders ausgedrückt, das Frontend eines Webshop-Systems.

Da der Industriepartner Kunden aus unterschiedlichen Branchen mit teilweise sehr spezifischen und komplexen Anforderungen an das Shopsystem betreut, kann das Frontend von nopCommerce diesen Anforderungen nicht gerecht werden. Der Industriepartner hat bewusst darauf verzichtet, die Machbarkeit einer Integration der fehlenden Funktionalitäten in das bestehende Frontend zu prüfen. Dies liegt zum einen daran, dass der verwendete Technologiestack, Razor Pages[3], nicht dem Know-how der Mitarbeiter entspricht. Zum anderen besteht der Wunsch, eine massgeschneiderte Lösung von Grund auf

[3]Razor Pages ist ein serverseitiges Webanwendungs-Framework von ASP.NET, das eine seitenorientierte Codierungsstruktur verwendet, um Entwicklern die Gestaltung dynamischer Webseiten mithilfe von HTML, CSS und C# zu ermöglichen.

zu entwickeln, um sie optimal auf die Kundenbedürfnisse auszurichten. Zudem eröffnen die modernen JavaScript-Frameworks die Möglichkeit, ein performanteres und interaktiveres Webshop-System zu entwickeln.

Das angestrebte Ziel ist es, nopCommerce als zentrales Backend-System für die Bereitstellung von E-Commerce-Funktionalitäten einzusetzen. Unter E-Commerce-Funktionalitäten verstehen sich die Bereitstellung von Produktstammdaten, die Verwaltung der Online-Kunden und die Abwicklung der Verkaufsaufträge. Darüber hinaus spielt nopCommerce eine zentrale Rolle bei der Integration des Abacus ERP-Systems, ermöglicht durch den Einsatz des aus der Vorarbeit stammenden Connectors. Ein Vorteil dieser Zielarchitektur ist, dass später auch andere Kanäle die Dienste von nopCommerce nutzen können, beispielsweise ein Kassensystem.
In der Fachliteratur wird dieser Ansatz häufig als 'headless Commerce' bezeichnet [45]. Der Ausdruck 'headless' findet bereits seit geraumer Zeit Anwendung in der Beschreibung von Softwarearchitekturen, bei denen die Datenverwaltung und Geschäftslogik (das Backend) sowie die Benutzeroberfläche (das Frontend) technisch separiert sind. Diese Trennung ermöglicht es einem oder mehreren Frontendsystemen über die zur Verfügung gestellten Schnittstellen auf Backend-Funktionalitäten zuzugreifen.

2.2 Vision

Das Hauptziel dieser Arbeit ist die Konzeption und Umsetzung eines Prototypen des Storefrontsystems, welches den 'AbaShop' ersetzen soll. Dieses Storefrontsystem wird durch die Verwendung der Dienste von nopCommerce in das Abacus ERP-System integriert. Mit Hilfe der Bereitstellung von nopCommerce als 'headless-Commerce'-System wird die Möglichkeit eröffnet, zukünftig auch weitere Lösungen neben dem Storefrontsystem anzubieten.
Eine zentrale Herausforderung besteht darin, das System under Development (SuD) an die vielfältige Kundenbasis von Customize anzupassen. Jeder Kunde hat individuelle Bedürfnisse bezüglich Funktionalität und visuellem Design des Storefrontsystems. Da die Mitarbeiter der Customize technisch versiert, jedoch unterschiedlich gute Programmierfähigkeiten besitzen, sollen grundlegende Anpassungen möglichst einfach ermöglicht werden. Um die Updatefähigkeit des vielfach eingesetzten Systems zu gewährleisten, müssen kundenspezifische Konfigurationen und Erweiterungen möglich sein, ohne Änderungen am Source-Code der Kernfunktionalitäten vornehmen zu müssen.
Im weiteren Verlauf dieser Arbeit bezeichnet SuD das Storefrontsystem, welches sowohl das für den Endkunden bestimmte Webshop-Frontend als auch den für die Konfiguration von Logik und Design des Webshops notwendigen Backend-/Adminbereich umfasst.

Abbildung 2.2 veranschaulicht die Zielarchitektur des in dieser Arbeit entwickelten Prototypen anhand eines C4-Kontextdiagramms. Im Gegensatz zum vorher eingeführten Kontextdiagramm (siehe Abbildung 2.1) greift der Shopper nicht mehr auf nopCommerce zu, sondern auf das SuD dieser Arbeit, das die Dienste von nopCommerce nutzt.

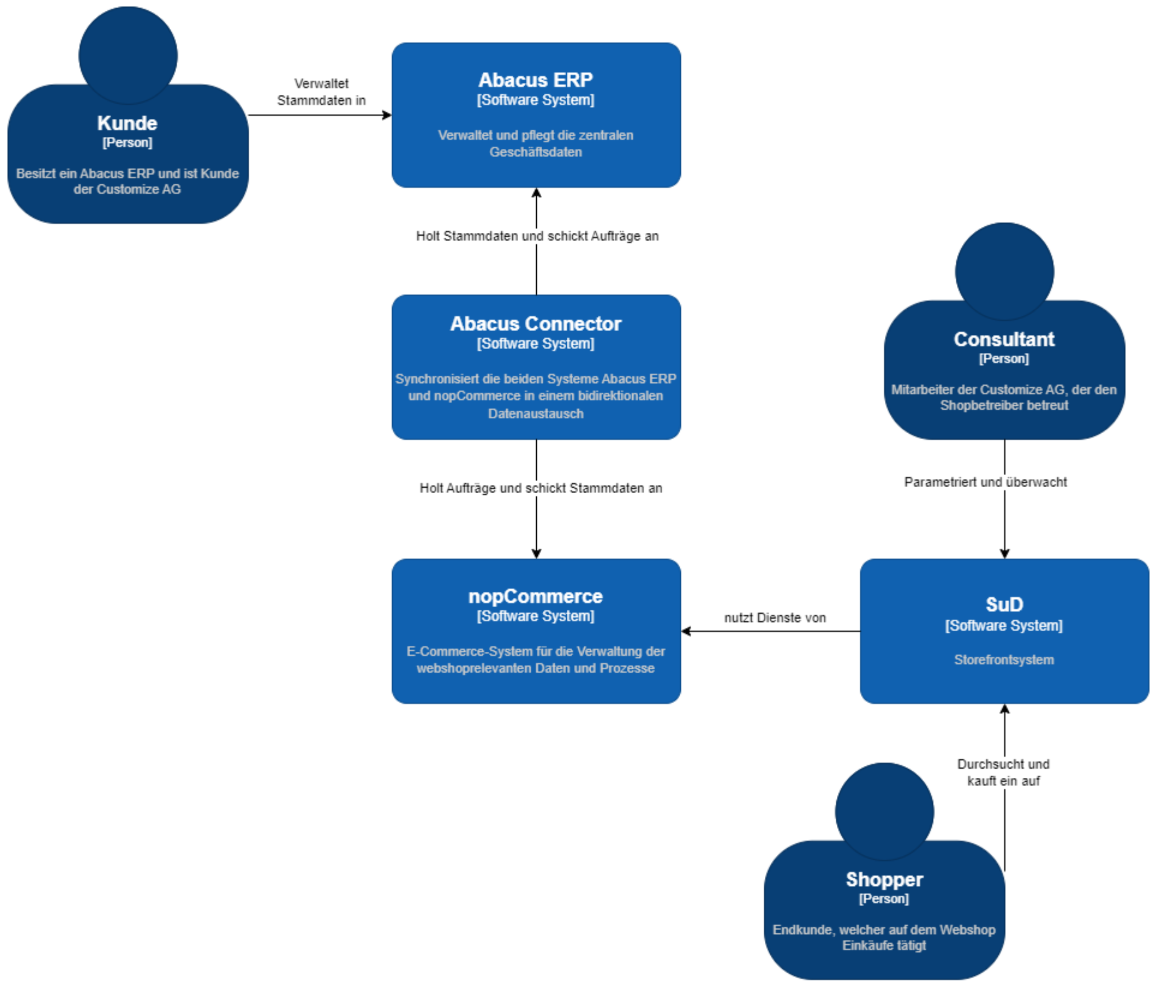


Abbildung 2.2: Kontextdiagramm Zielarchitektur der Bachelorarbeit (C4)

Mit dem Prototypen dieser Arbeit möchte der Industriepartner die Realisierbarkeit seiner Vision einer selbst entwickelten Nachfolgelösung für 'AbaShop' überprüfen. Die Lösung soll die Möglichkeiten zur kundenspezifischen Individualisierung vom 'AbaShop' wieder zur Verfügung stellen und mit neuen, oft gewünschten Möglichkeiten erweitern. Der Prototyp soll helfen, die auf Ende 2024 terminierte Make or Buy Entscheidung zu treffen. Spätestens zu diesem Zeitpunkt muss entschieden werden, ob auf Grundlage des Prototypen eine eigene Lösung entwickelt oder ob eine Nachfolgelösung basierend auf anderen Systemen oder Entwicklungen von Konkurrenten für seine Kunden erworben wird.

Abbildung 2.3 zeigt den groben Zeitplan vom Projekt 'Ablösung AbaShop' auf.

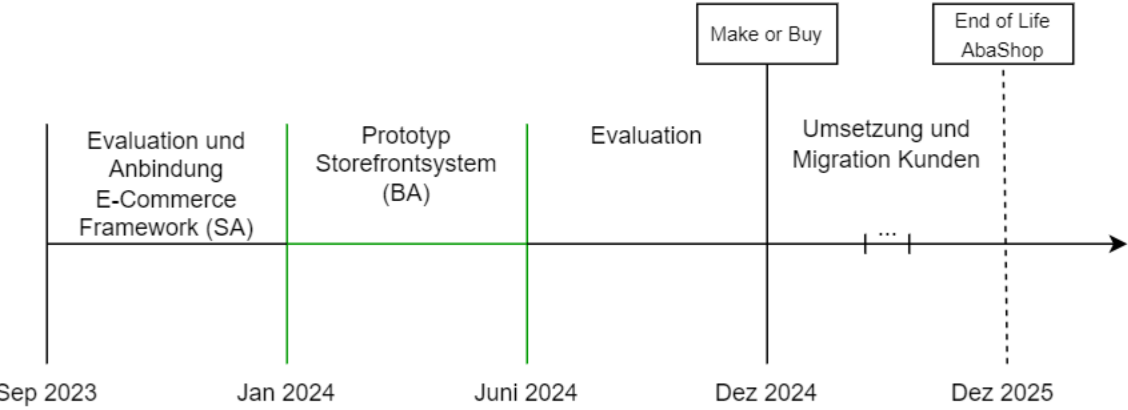


Abbildung 2.3: Projektplan Ablösung 'AbaShop'

2.3 Dokumentation

In diesem Dokument wird die anonymisierte Firma 'Sportartikel AG' als Modellunternehmen verwendet, um konzeptionelle Themen anhand eines realen Beispiels zu veranschaulichen. Solche Beispiele sind mit der folgenden Box gekennzeichnet:

Abacus

Die Sportartikel AG nutzt das Abacus ERP um ihre Geschäftsprozesse digital abzubilden.

Die anonymisierte Firma Sportartikel AG ist ein realer Kunde von Customize und nutzt einen umfassenden 'AbaShop'. Sie ist ein wichtiger Stakeholder einer potentiellen Nachfolgelösung vom 'AbaShop' und wird in dieser Arbeit als Beispielkunde für die Anforderungserhebung und konkreter Beispiele herangezogen.

2.4 Abacus

In diesem Abschnitt werden die fachlichen Domänenkonzepte von Abacus vorgestellt. Die vorgestellten Konzepte beschränken sich auf jene, die für die Kernfunktionalitäten eines Webshop-Systems relevant sind. Abacus ist ein modular aufgebautes ERP-System, wobei von Applikationen gesprochen wird. Jede Applikation deckt einen spezifischen fachlichen Bereich des ERP ab. Diese Applikationen sind innerhalb von Abacus über Standard-Schnittstellen miteinander verknüpft. Im Rahmen einer Webshop-Lösung sind drei Applikationen aus dem Portfolio von Abacus beteiligt: Auftragsbearbeitung, Customer Relationship Management (CRM) und E-Business. In Abbildung 2.4 wird dargestellt, wie diese Applikationen interagieren und wo die Schnittstellen innerhalb von Abacus sowie zum Webshop liegen. Innerhalb einer Gruppe werden die wichtigsten Domänenkonzepte als Klassen dargestellt. Im Domänenmodell in Abschnitt 2.4.1 werden diese Konzepte weiter ausgeführt.

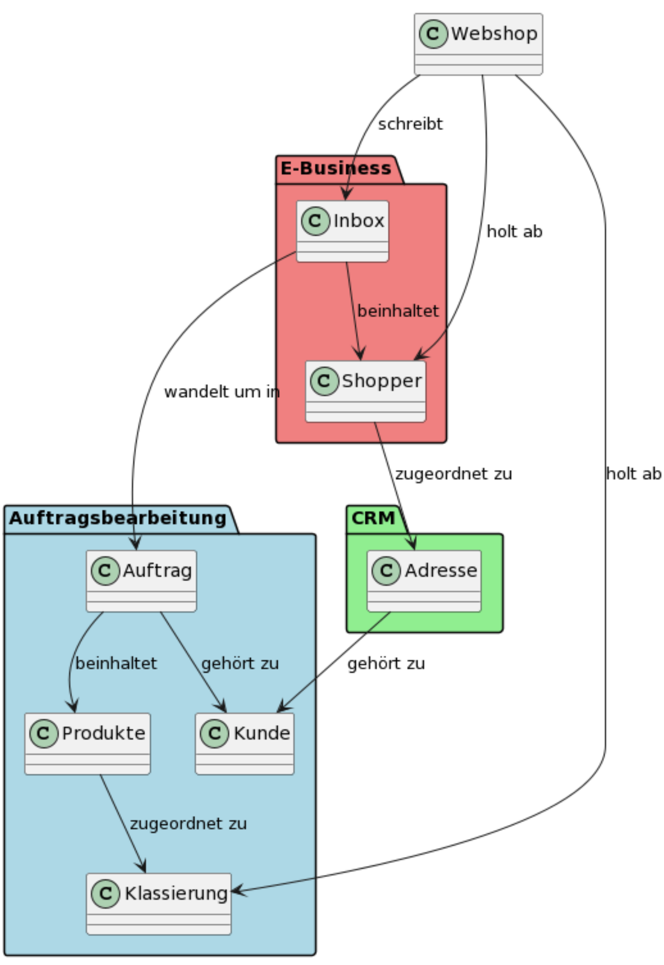


Abbildung 2.4: Übersicht Applikationen Abacus [8]

Die Abbildung der Schnittstellen zu einem Webshop ist vereinfacht dargestellt und wurde in der vorangegangen Studienarbeit vertieft, da in vorliegender Arbeit die Anbindung der Schnittstellen an nopCommerce implementiert wurde.

2.4.1 Domänenmodell

In Abbildung 2.5 ist das Domänenmodell von Abacus aufgeführt.

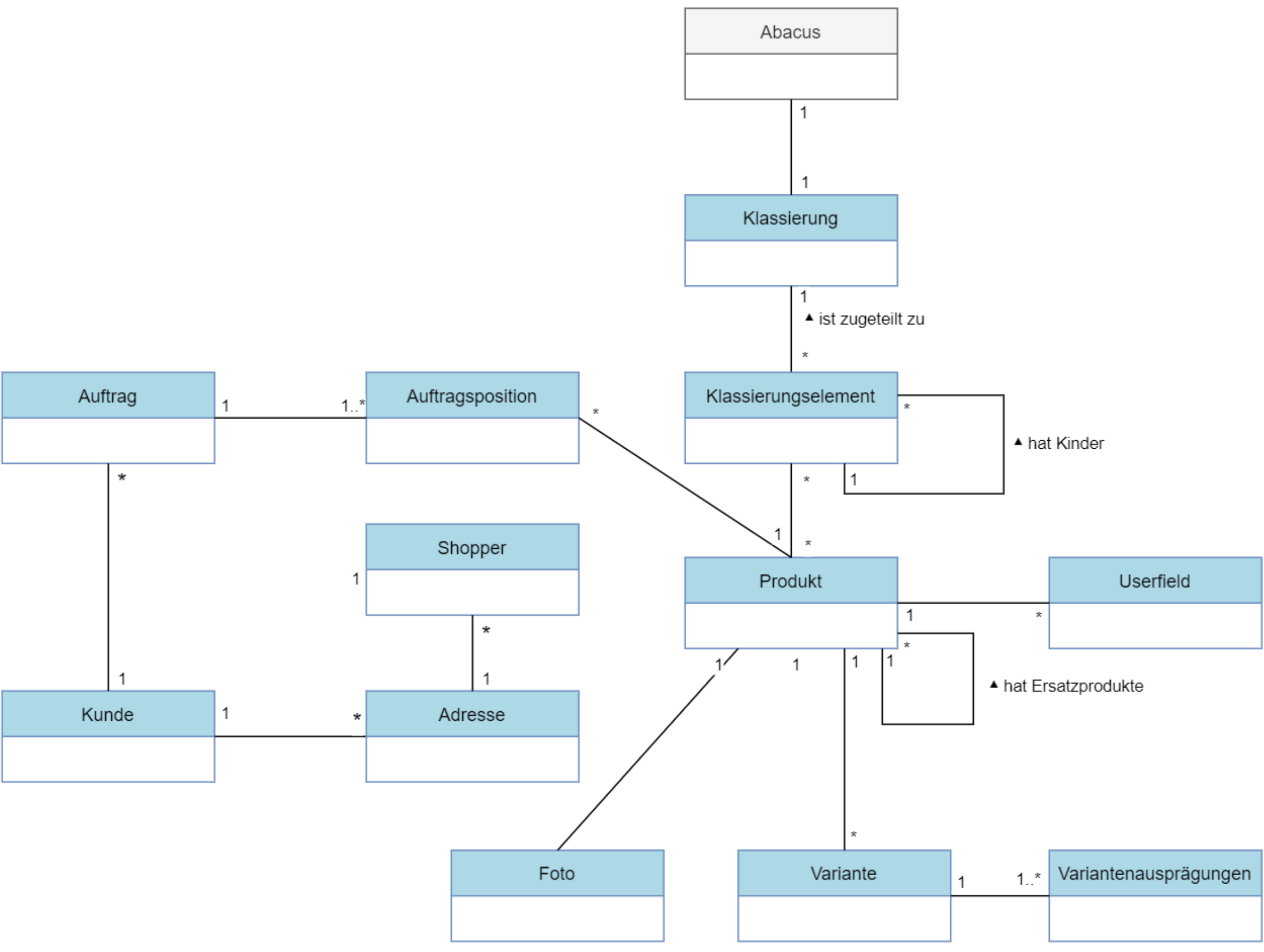


Abbildung 2.5: Domänenmodell Abacus

In den nachfolgenden Abschnitten folgen detaillierte Ausführungen zum Domänenmodell von Abacus, soweit diese für den Kontext dieser Arbeit von Bedeutung sind.

2.4.2 Klassierung

Eine Klassierung, auch Produktklassierung genannt, stellt eine hierarchische Gliederung des gesamten Produktsortiments einer Abacus-Installation dar. Sie umfasst eine unbegrenzte Anzahl an Klassierungselementen, die in einer Baumstruktur bis zu neun Stufen tief miteinander verknüpft werden können. Den Klassierungselementen werden Produkte zugewiesen, wobei ein Produkt mehreren Klassierungselementen zugeordnet sein kann. Obwohl bei den meisten Customize-Kunden die Produktzuordnung auf der untersten Ebene erfolgt, ist eine Zuweisung auf jeder beliebigen Ebene möglich.

Klassierung

Die Sportartikel AG strukturiert ihr gesamtes Produktsortiment in einer Klassierung, um eine klare Kategorisierung der Artikel zu gewährleisten. Durch die Strukturierung können Artikel gruppiert werden um die Auffindbarkeit zu verbessern. Abbildung 2.6 zeigt, wie eine solche Klassierung aussehen kann.

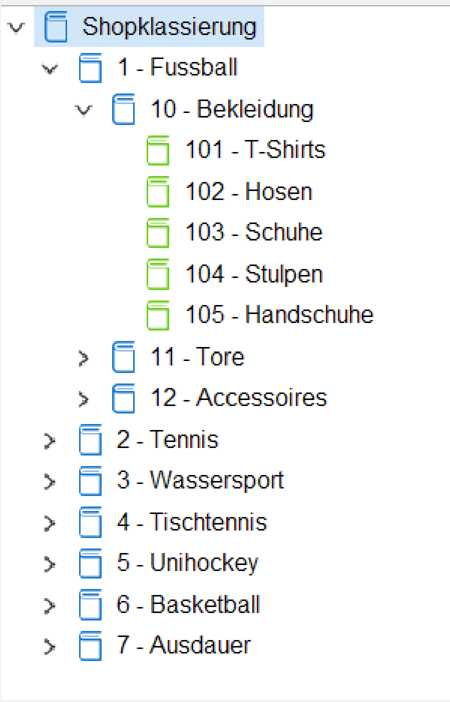


Abbildung 2.6: Beispiel Klassierung in Abacus

Im Webshop der Sportartikel AG wird die Klassierung als Navigationsbaum dargestellt, damit ein Kunde den gewünschten Artikel besser finden kann.

2.4.3 Userfield

Abacus bietet die Möglichkeit, mittels Userfields, die auch oft als Benutzerfelder bezeichnet werden, auf der Ebene einer Entität individuelle Attribute zu definieren. Im Rahmen eines Webshops kommen diese Userfields bei der Entität 'Produkt' zum Einsatz, um Kunden aus verschiedenen Branchen die Definition spezifischer Produkteigenschaften zu ermöglichen. Abhängig von den spezifischen Anforderungen eines Kunden können Userfields auch bei Aufträgen oder Adressen Anwendung finden.

Userfields

Die Sportartikel AG implementiert für den Verkauf ihrer Tischtennisschläger zwei Userfields auf den Produkten: 'Griffart' und 'Gewicht'. Diese Attribute dienen dazu, Kunden detaillierte Informationen über die Griffart und das Gewicht des jeweiligen Tischtennisschlägers zu geben. Die Mitarbeiter der Sportartikel AG haben die Möglichkeit, bei jedem Tischtennisschläger spezifische Werte für 'Griffart' und 'Gewicht' festzulegen. Durch die systematische Erfassung dieser Daten in den Produktattributen lassen sich Filter im Webshop realisieren. Diese Filter ermöglichen es den Kunden, gezielt nach Tischtennisschlägern mit den gewünschten Eigenschaften zu suchen, was zu einer deutlich verbesserten Einkaufserfahrung führt.

2.4.4 Variante

Ein Artikel kann als Variantenartikel angelegt werden, wodurch Kunden die Möglichkeit erhalten, aus verschiedenen Ausprägungen eines Produkts zu wählen. Jede Variante ist durch ihre einzigartigen Merkmale (Variantenausprägungen) charakterisiert und definiert.

Variante

Die Sportartikel AG bietet Fussball-T-Shirts in unterschiedlichen Designs an. Jede Variante definiert sich dabei durch die beiden Variantenausprägungen Design (z.B. Heim-, Auswärts-, Dritttrikot) und Grösse (S, M, L, XL). Somit ergeben sich mehrere Varianten des Artikels, mit beispielsweise einer Variante mit Design = Heimtrikot und Grösse = M.

Für jede einzelne Variante lassen sich Lagerbestand und Preis spezifisch festlegen. Aus diesem Grund erfolgt die eindeutige Identifikation eines Variantenartikels im Abacus ERP nicht nur über die Artikelnummer, sondern zusätzlich durch die Variantennummer.

2.4.5 Kunde / Adresse / Shopper

Die Verwaltung von Adressdaten in Kombination mit Online-Kunden, die auch als Shopper bezeichnet werden, wird im Abacus-System hierarchisch gehandhabt. Ein Auftrag wird immer einem Kunden zugeordnet, der auch als Debitor bekannt ist. Ein Kunde kann über mehrere Lieferadressen verfügen, an die Bestellungen versendet werden. Jeder dieser Adressen können wiederum mehrere Shopper zugeordnet werden, die ihre Einkäufe im Webshop tätigen. Diese Struktur ermöglicht es, dass mehrere Shopper im ERP Einkäufe für denselben Kunden durchführen können. Diese Struktur wurde bereits in nopCommerce implementiert.

Shopper

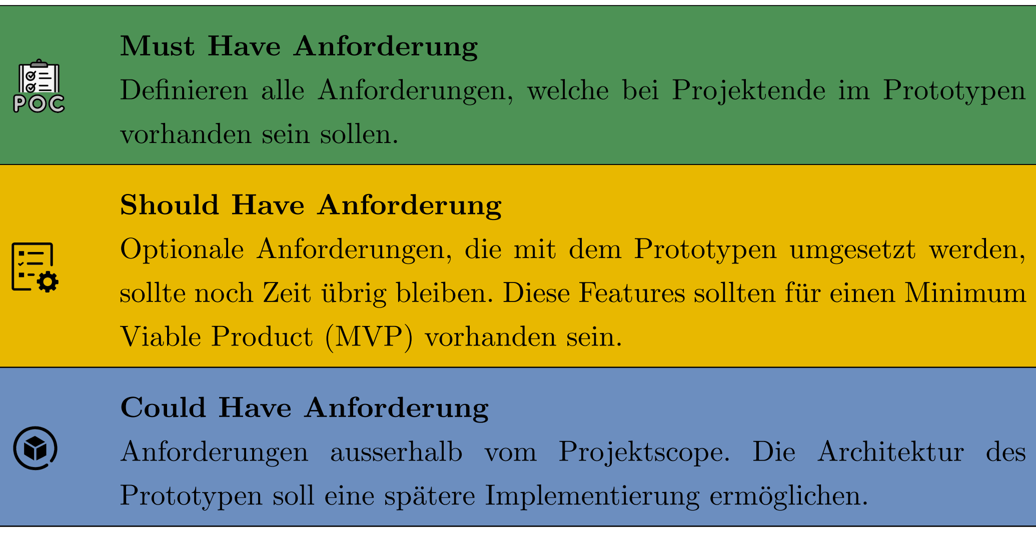
Die Sportartikel AG versorgt ein grosses Tenniscenter, das über zwei Standorte verfügt. An jedem dieser Standorte arbeiten mehrere Mitarbeiter, die für ihren Bedarf an Tennismaterialien bei der Sportartikel AG einkaufen. Im Abacus der Sportartikel AG ist das Tenniscenter als Kunde mit zwei unterschiedlichen Adressen geführt. Jeder Mitarbeiter des Tenniscenters verfügt über einen individuellen Shopper-Account, welcher der Adresse seines spezifischen Standorts zugeordnet ist. Dadurch kann die Sportartikel AG Rechnungen stets demselben Kunden, dem Tenniscenter, zuordnen, und die Lieferung der Tennismaterialien erfolgt an die jeweilige Adresse des zugeordneten Shoppers.

Kapitel 3

Anforderungen

Ein zentrales Ergebnis dieser Bachelorarbeit ist die detaillierte Erfassung der Anforderungen an das neue Storefrontsystem hinsichtlich seiner Erweiterbarkeit und der Parametrierbarkeit von Aussehen sowie Logik. Das Wissen dieser Anforderungen ist unter den Consultants des Industriepartner verteilt, da jeder die umgesetzten Spezialfunktionalitäten seiner zugeteilten Kunden kennt. Für den Industriepartner Customize ist es wichtig, die vorhandenen Spezialanforderungen der breiten Kundenbasis zu kennen und allfällige zusätzliche Wünsche der Kunden zu berücksichtigen. Viele der erfassten Anforderungen dieser Arbeit sind in Zusammenarbeit mit den Mitarbeitern von Customize entstanden, da diese die Kunden am besten kennen.

Die aufgenommenen funktionalen Anforderungen wurden priorisiert, um den 'Make or Buy'-Entscheidungsprozess des Industriepartners im Zusammenhang mit der Ablösung des bestehenden 'AbaShop'-Systems zu erleichtern. Die Methode der Priorisierung sowie die resultierenden Prioritäten sind in diesem Kapitel entsprechend der Tabelle 3.1 schematisch dargestellt.



Symbol	Beschreibung
POC	**Must Have Anforderung** Definieren alle Anforderungen, welche bei Projektende im Prototypen vorhanden sein sollen.
	Should Have Anforderung Optionale Anforderungen, die mit dem Prototypen umgesetzt werden, sollte noch Zeit übrig bleiben. Diese Features sollten für einen Minimum Viable Product (MVP) vorhanden sein.
	Could Have Anforderung Anforderungen ausserhalb vom Projektscope. Die Architektur des Prototypen soll eine spätere Implementierung ermöglichen.

Tabelle 3.1: Priorisierungschema Anforderungen

3.1 Vorgehen

Der im Abschnitt 2.3 vorgestellte Musterkunde Sportartikel AG wurde gezielt für die Anforderungserhebung ausgewählt. Die Sportartikel AG ist ein wichtiger und repräsentativer Customize-Kunde, der einen 'AbaShop' im Einsatz hat und somit ein potentieller Anwender des zu entwickelnden Systems ist. Bei diesem Kunden sind viele Spezialanforderungen umgesetzt, weshalb er als Beispielkunde ausgewählt wurde. In Zusammenarbeit mit dem zuständigen Consultant dieses Kunden wurden die Anforderungen erhoben. Obwohl dieser Kunde viele der Anforderungen für die Erweiterung der Standardfunktionalitäten abdeckt, sind diese nicht vollständig. Daher wurden auch Anforderungen anderer Kunden aufgenommen und aus Sicht des Musterkunden dokumentiert. Dies ermöglicht eine umfassende und repräsentative Erfassung der Anforderungen für die gesamte Kundenbasis von Customize.

3.2 Akteure

In Tabelle 3.2 sind alle Akteure aufgeführt, die zur Beschreibung der funktionalen und nichtfunktionalen Anforderungen in diesem Kapitel herangezogen werden.

Akteur	**Beschreibung**
Consultant	Ein Mitarbeiter der Customize betreut als technischer Berater einen Kunden, der Besitzer eines Abacus ERP-Systems mit angebundenem Webshop ist. Der Consultant beabsichtigt, das Storefrontsystem einzuführen und es an die Bedürfnisse des Kunden anzupassen. Es ist dabei wichtig zu beachten, dass die Kenntnisse in der Softwareentwicklung unter den Consultants variieren können, da dies nicht zum Hauptumfang ihrer täglichen Arbeit gehört. Daraus folgt, dass die Parametrierung und Anpassung des Systems so gestaltet sein sollte, dass sie auch für einen Junior-Entwickler mit grundlegenden Kenntnissen gut durchführbar ist.
Entwickler	Kunden der Customize AG haben teilweise intern oder bei externen Firmen Entwickler angestellt, die zusätzliche Anforderungen ausserhalb der Standardfunktionalitäten eines Webshops umsetzen. Diese Anpassungen können, je nach Wunsch des Kunden, auch von einem Consultant der Customize AG mit den entsprechenden Fähigkeiten entwickelt werden.
Kunde	Der Akteur Kunde ist der Besitzer eines Storefrontsystems und entsprechend Kunde beim Industriepartner Customize. Der Kunde arbeitet mit Abacus als ERP-System und hat einen 'AbaShop' als Webshoplösung im Einsatz oder möchte neu einen ERP-integrierten Webshop einführen.

Shopper	Ein Shopper ist wiederum Kunde des oben eingeführten Akteurs Kunde. Dies bedeutet, dass ein Shopper als Endkunde auf dem Storefrontsystem die angebotenen Produkte sichtet und schliesslich einkauft.
Abacus ERP	Jeder Kunde der Customize setzt das Enterprise Ressource Planning (ERP) System von Abacus ein. Das Abacus ERP und nopCommerce tauschen über eine bidirektionale Schnittstelle Daten aus. Mithilfe eines Connectors [8] werden in einem festgelegten Zeitintervall Produkt- und Kundendaten in nopCommerce übertragen. Im Gegenzug übergibt nopCommerce neue Bestellungen an Abacus zur Weiterverarbeitung und Fakturierung.
nopCommerce	NopCommerce ist ein Open-Source E-Commerce-System, das als 'headless-Commerce'-System zur Verwaltung von Stammdaten und Transaktionsdaten eingesetzt wird. Das zu entwickelnde Storefrontsystem soll dieses System nutzen, um die erforderlichen Stammdaten abzurufen und neue Transaktionen zu erstellen.

Tabelle 3.2: Akteure

3.3 Kernfunktionalitäten

Kernfunktionalitäten sind Anforderungen, die zu den Standardfunktionalitäten eines Webshop-Systems gehören und somit allen Kunden als Kernsystem identisch zur Verfügung stehen. Das Ziel dieser Arbeit ist es nicht, diese Anforderungen vollständig zu erfassen und zu entwickeln. Eine ausführliche Erfassung wurde bereits in der Studienarbeit [8] von Ramon Ebneter durchgeführt. Um die evaluierten Anforderungen an die Konfigurationsmöglichtkeiten wie auch an das Theming- und Plugin-System zu demonstrieren, werden ausgewählte Kernfunktionalitäten umgesetzt. Diese werden in Form von User Stories dokumentiert, welche im nachfolgenden Format gemäss der Vorlage von Mike Cohn erfasst werden [5].

`Als <Art des Benutzers> möchte ich <ein bestimmtes Ziel>, sodass <ein bestimmter Grund>.`

3.3.1 US1: Produktklassierung durchsuchen

`Als Shopper möchte ich über einen Navigationsbaum durch die verschachtelte Produktklassierung navigieren und dabei das gewünschte Klassierungselement selektieren, sodass ich die zugeordneten Artikel aus dem ausgewählten Klassierungselement angezeigt erhalte, um die Auswahl der gewünschten Artikel gezielt einzuschränken.`

In der Detailansicht gibt es zwei unterschiedliche Darstellungsmöglichkeiten für die angezeigten Daten:

- **Produktansicht:** Diese Ansicht präsentiert jeden Artikel, der dem ausgewählten Klassierungselement zugeordnet ist.
- **Klassierungsansicht:** Hier werden alle Klassierungselemente dargestellt, die dem ausgewählten Element als Unterkategorien zugeordnet sind.

Die Auswahl zwischen diesen Ansichten hängt in der Regel davon ab, wie tief der Benutzer in der Navigation vorgedrungen ist. Normalerweise wird auf oberen Ebenen die Klassierungsansicht angezeigt, da hier die Anzahl der zugeordneten Artikel sehr hoch sein kann. Die Parametrierung dieser Ansichten wird in Abschnitt 3.4.6 behandelt.

3.3.2 US2: Filter anwenden

`Als Shopper möchte ich auf einem Klassierungselement Filter anwenden, sodass ich die Auswahl der angezeigten Artikel weiter einschränken kann.`

Diese User Story folgt auf *US1: Produktklassierung durchsuchen* und wird ausgeführt, sobald ein Klassierungselement ausgewählt wurde. Die Produktfilter basieren auf Attributen, die bei den Artikeln hinterlegt sind. In Abacus sind dies benutzerdefinierte Felder (Userfields) auf den Produkten, die zur Erfassung beliebiger Eigenschaften dienen. Es existieren zwei verschiedene Filtertypen:

- **Auswahlliste:** Nutzer können aus verschiedenen Ausprägungen eines Attributs in einer Liste mit Möglichkeit zur Mehrfachauswahl wählen.
- **Rangepicker:** Der Shopper kann Werte eines Attributs innerhalb eines bestimmten Bereichs (von-bis) filtern. Dies ist ausschliesslich bei Attributen des Datentyps `number` sinnvoll.

Auf einem spezifischen Klassierungselement werden nur jene Filter angezeigt, die bei den zugehörigen Artikeln mindestens zwei verschiedene Ausprägungen aufzeigen. Filter, die keinen Mehrwert für die weitere Einschränkung der Produktauswahl bieten, werden nicht zur Auswahl angeboten. Die Parametrierung dieser Filter wird in Abschnitt 3.4.4 behandelt.

In Abbildung 3.1 wird anhand eines Wireframes aufgezeigt, wie solche Filter aussehen könnten. In diesem Fall zeigt das Storefrontsystem lediglich die Artikel an, welche auf dem Userfield 'Farbe' den Wert 'Blau' geführt haben.

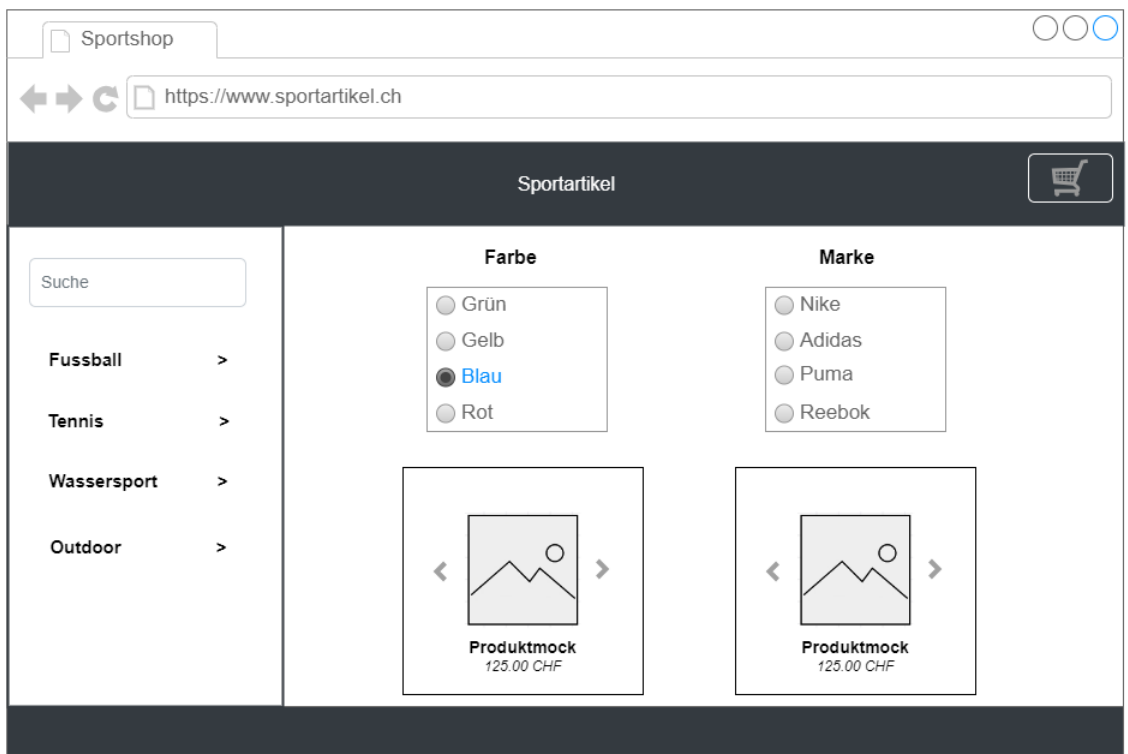


Abbildung 3.1: Wireframe Filter

3.3.3 US3: Produktdetails einsehen

`Als Shopper möchte ich auf die Detailansicht eines Produktes zugreifen können, sodass ich weitere Informationen zum Artikel einsehen kann.`

Bei einem durchschnittlichen Kunden zeigt diese Ansicht den Preis, eine detaillierte Produktbeschreibung in Hypertext Markup Language (HTML), alle Fotos sowie die individuellen Produktattribute an. Die angezeigten Informationen hängen jedoch oft stark vom Kunden ab. Das Theming-System in Abschnitt 3.5 führt die Anforderungen an diese kundenspezifische Parametrierung ein.

3.3.4 US4: Produkt in den Warenkorb legen

`Als Shopper möchte ich einen Artikel in den Warenkorb legen können, sodass ich meine Auswahl für den späteren Kauf speichern kann.`

Die Persistierung eines Warenkorbes setzt eine Login-Funktionalität voraus, damit der Warenkorb in nopCommerce einem Shopper zugeordnet und bei einem späteren Login erneut abgerufen werden kann. Zur Umsetzung dieser User Story soll der Login implementiert werden, jedoch ohne einen Registrierungsmechanismus.

3.3.5 US5: Bestellung tätigen

`Als Shopper möchte ich meine ausgewählten Artikel bestellen können, sodass ich meinen Einkauf abschliessen und die Produkte zu mir liefern lassen kann.`

Diese User Story basiert auf *US4: Produkt in den Warenkorb legen*, in dem der erstellte Warenkorb vom Shopper als Verkaufsauftrag über nopCommerce an Abacus geschickt wird. Eine Bestellung kann dabei als Gastbenutzer (ohne Login) wie auch als eingeloggter Shopper abgesetzt werden.

3.3.6 US6: Sprache im Shop ändern

`Als Shopper möchte ich die Sprache im Shop ändern können, sodass ich die Inhalte in meiner bevorzugten Sprache lesen kann.`

Die Sprache im Shop sollte über ein Dropdown-Menü im Storefrontsystem geändert werden können. Die verfügbaren Sprachen umfassen in der Regel Deutsch, Französisch, Italienisch und Englisch, entsprechend den Sprachen, die das Abacus ERP bereitstellt. Die Übersetzungen sollen aus einer textbasierten Datei geladen werden, um eine einfache Anpassung und Erweiterung der Texte zu ermöglichen.

3.4 Konfiguration

In diesem Abschnitt werden Anforderungen an die Konfigurationsmöglichkeiten des Storefrontsystems beschrieben. Diese User Stories sind aus Sicht des Consultants formuliert, können jedoch auch vom Kunden selbst umgesetzt werden. In der Regel hängt es stark vom Kunden ab, durch welchen Akteur diese User Stories durchgeführt werden, da manche Kunden vieles selbst übernehmen möchten, während andere alles vom Consultant erledigen lassen. Die beschriebenen Anforderungen decken die Konfigurationsmöglichkeiten ab, die Personen mit wenig bis keinen Programmierkenntnissen zur Verfügung gestellt werden sollen.

3.4.1 US7: Farbgebung editieren

> `Als Consultant möchte ich auf einer Administrationsseite die Farbe der Buttons anpassen, sodass ich das Storefrontsystem ohne Schreiben von Source-Code auf das Corporate Design (CD) von meinem Kunden anpassen kann.`

Die eingeführte User Story, die sich auf die Anpassung der Buttonfarbe konzentriert, soll auf alle anderen Komponenten des Storefrontsystems anwendbar sein. Durch die Einführung eines Farbkonzeptes soll ermöglicht werden, alle Farben des Systems an einem zentralen Ort zu konfigurieren. Unter einem Farbkonzept versteht sich die Nutzung einer begrenzten Anzahl vordefinierter Farben innerhalb einer Anwendung nach festgelegten Kriterien. Dies ermöglicht dem Consultant, das gesamte Storefrontsystem gezielt gemäss dem Farbschema des Corporate Designs des Kunden anzupassen.

3.4.2 US8: Logo und Stammdaten editieren

> `Als Consultant möchte ich auf einer Administrationsseite Stammdaten erfassen und mutieren können, sodass ich statische Daten im laufenden Betrieb in das Storefrontsystem einpflegen kann.`

Diese User Story soll die minimal notwendige Konfiguration ermöglichen, damit ein Kunde mit einem einfachen Webshop ohne Anforderungen an die visuelle Erscheinung in Betrieb genommen werden kann. Als Minimalkonfiguration wurden folgende Informationen identifiziert:

- Logo
- Favicon
- Adresse mit Kontaktdaten (Telefon und E-Mail)
- Öffnungszeiten der Telefonhotline
- Verbindung zu nopCommerce

3.4.3 US9: Informationstexte verwalten

> Als Consultant möchte ich auf einer Administrationsseite formatierte Informationstexte erfassen und editieren, sodass die rechtlich benötigten statischen Texte im Fussbereich des Webshops angezeigt werden.

In einem Webshop müssen im Fussbereich bestimmte Texte aus rechtlichen Gründen im unteren Bereich (Footer) angezeigt werden. Je nach Kunde werden verschiedene Texte benötigt, die im Administrationsbereich gepflegt werden sollen.
Im Normallfall betrifft dies folgende Texte:

- Allgemeine Geschäftsbedingungen (AGB)
- Impressum
- Datenschutzerklärung

3.4.4 US10: Filter aktivieren

> Als Consultant möchte ich auf einer Administrationsseite parametrieren, welche Filter im Webshop zur Verfügung stehen sowie den Filtertyp definieren (Auswahlliste oder Rangepicker), sodass ich diese vielgefragte Funktionalität ohne Schreiben von Source-Code in kurzer Zeit einführen kann.

In *US2: Filteranwendung* wird erläutert, dass ein Shopper die Möglichkeit haben möchte, innerhalb eines Klassierungselements Filter anzuwenden, um die Auswahl der angezeigten Produkte weiter einzuschränken. Diese Filter sollen durch diese User Story aktiviert werden können. Normalerweise wird ein Filter als individuelles Attribut eines Produkts im Abacus ERP geführt. Es gibt jedoch Attribute, die lediglich als beschreibendes Merkmal bei einem Produkt angezeigt werden, aber nicht als Filter zur Verfügung stehen sollen. Der Consultant konfiguriert auf der Administrationsseite, welche Produktattribute als Filter verfügbar gemacht werden. Zudem wird festgelegt, ob der Filter als *Auswahlliste* oder *Rangepicker* zur Verfügung steht. Sobald ein Shopper gemäss US2 verschiedene Filter anwenden möchte, sollen die parametrierten Filter zur Verfügung stehen.

3.4.5 US11: Filter pro Klassierungselement parametrieren

`Als Consultant möchte ich auf einer Administrationsseite auf Stufe Klassierungselement definieren, welche Filter zur Verfügung stehen, sodass für jedes Klassierungselement unterschiedliche Filter zur Verfügung stehen und folglich eine spezifischere Konfiguration der Filter ermöglicht wird.`

Diese User Story soll die Möglichkeiten von *US9: Filter aktivieren* erweitern. Ein grosser Teil der Customize-Kunden möchte global definieren, welche Benutzerfelder als Filter zur Verfügung stehen. Es gibt jedoch einige, die je nach ausgewähltem Klassierungselement unterschiedliche Filter anzeigen möchten, was eine Funktion erfordert, im hierarchischen Klassierungsbaum pro Element die Filter zu aktivieren.

3.4.6 US12: Klassierungsnavigation parametrieren

`Als Consultant möchte ich auf einer Administrationsseite parametrieren, welche Details ein Shopper bei der Auswahl eines Klassierungselementes sieht, sodass eine zielgerichtete Produktsuche ermöglicht wird.`

In *US1: Produktklassierung durchsuchen* wird beschrieben, wie ein Shopper durch den Baum der Produktklassierung navigiert, um gezielt nach gewünschten Artikeln zu suchen. Der Consultant soll die Möglichkeit haben, auf der Administrationsseite zu parametrieren, welche Detailansicht nach der Auswahl eines Klassierungselements angezeigt wird. Wie bereits in US1 dargelegt, werden entweder die Unterkategorien des ausgewählten Elements (*Klassierungsansicht*) oder die darin verknüpften Artikel (*Produktansicht*) angezeigt.

Die Kunden von Customize haben folgende Optionen als mögliche Anforderung:

- **Klassierungstiefe:** Ein definierter Schwellenwert für die Klassierungstiefe bestimmt, ab welchem Punkt das *Produktdetail* angezeigt wird. Die Klassierungstiefe bezieht sich auf die Ebene innerhalb des gesamten Klassierungsbaums.
- **Anzahl der Artikel:** Für ein ausgewähltes Klassierungselement wird basierend auf der Anzahl der zugeordneten Artikel und einem festgelegten Schwellenwert entschieden, welche Ansicht gezeigt wird.

Nachdem in den beiden vorangegangenen Abschnitten die Standardfunktionalitäten aus Sicht des Shoppers und die Anforderungen an die Konfigurationsmöglichkeiten definiert wurden, werden in den beiden folgenden Abschnitten die Anforderungen an das Theming-System und das Plugin-System behandelt. Mit diesen beiden Konzepten soll die Möglichkeit geschaffen werden, alle Customize-Kunden mit einem identischen Kernsystem zu bedienen. Individuelle Anforderungen an das optische Erscheinungsbild (Theming) und an die Funktionalität (Plugin) sollen möglich sein, ohne Anpassungen am Source-Code des Kernsystems vornehmen zu müssen. Da diese Konzepte auch zukünftige, heute noch unbekannte Anforderungen abdecken sollen, werden sie als Use Cases beschrieben, die besser als User Stories die Interaktionen zwischen Systemen darstellen. Um zu verdeutlichen, wie diese beiden Konzepte mit dem Kernsystem interagieren, wird in der Beschreibung der Use Cases das Kernsystem gemäss der Definition in Tabelle 3.3 als eigener Akteur eingeführt.

Akteur	**Beschreibung**
Kernsystem	Das Kernsystem umfasst Funktionen des Storefrontsystems, die allen Kunden gleichermassen bereitgestellt werden. Diese entsprechen den Standardfunktionalitäten, die jeder Kunde von Customize im Storefrontsystem erwartet.

Tabelle 3.3: Zusätzlicher Akteur Use Cases

Die Notation der Unified Modeling Language (UML) für Use-Case-Diagramme basiert auf der Vorlage von Craig Larman [28]. Um die abstrakt gehaltenen Use Cases zu veranschaulichen, wird bei einigen eine reale Anforderung vom Musterkunden Sportartikel AG in Form eines Use Case Szenarios aufgeführt.

3.5 Theming System

Als beispielhafte Anforderung, welche durch das Theming-System abgedeckt werden kann, wurde folge User Story definiert:

US13: Angezeigte Daten im Produktdetail steuern

```
Als Consultant möchte ich in einem Theme konfigurieren, dass auf der
Detailansicht eines Produktes ein im Abacus ERP neu eingeführtes
Produktattribut angezeigt wird, sodass der Shopper eine bessere Beschreibung
des Produktes erhält.
```

3.5.1 Einleitung

Das Theming-System hat die zentrale Aufgabe, die individuellen Anforderungen eines Kunden an die visuelle Erscheinung des Storefrontsystemes abzudecken. Es ermöglicht von aussen ohne Anpassung des Kernsystems zu steuern, wie einzelne Komponenten erscheinen. Aus Sicht des Theming-Systems wurden die in Abbildung 3.2 dargestellten Use Cases identifiziert.

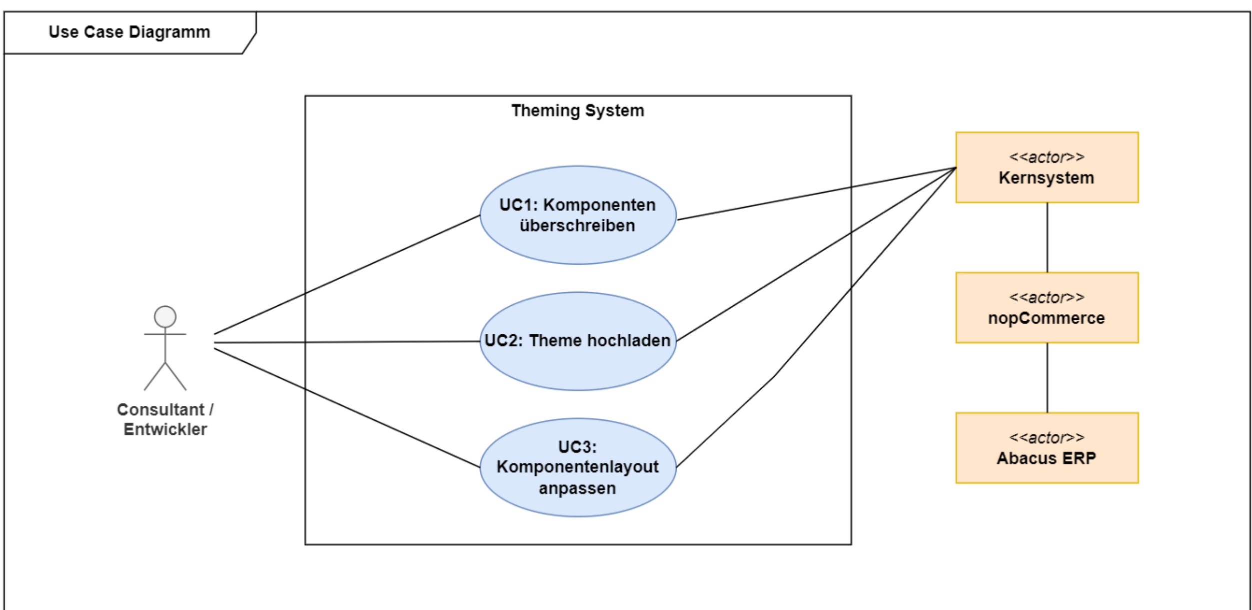


Abbildung 3.2: Use Cases des Theming-Systems (UML Use Case Diagramm)

3.5.2 UC1: Komponenten überschreiben

Komponenten werden in der Frontendentwicklung als wiederverwendbare Elemente für Webseiten und Anwendungen definiert, die mit HTML, CSS und JavaScript erstellt werden. Diese Komponenten sind gekapselt, was bedeutet, dass ihre interne Struktur von anderen Seitenelementen getrennt bleibt, um Konflikte mit dem restlichen Code zu vermeiden [13].

Use Case Brief
Dieser Use Case deckt den Kern des Theming-Systems ab. Entwickler sollen die Möglichkeit erhalten, Komponenten zu entwickeln, die Standardkomponenten des Kernsystems überschreiben können. Ziel ist es, nicht nur das Design anzupassen, sondern auch zu kontrollieren, welche Daten dargestellt werden. Wird ein Theme in einem Storefrontsystem aktiviert, prüft das Kernsystem bei jedem Aufruf einer Komponente, ob eine Überschreibung durch das Theme existiert. In einem solchen Fall übergibt das Kernsystem den Kontext an die Theme-Komponente und präsentiert sie gemäss der entwickelten Darstellung. Es soll eine klar definierte Schnittstelle bestehen, die angibt, welche Komponenten überschrieben werden können und welche Informationen der Entwickler im Kontext einer Komponente verwenden kann.

3.5.3 UC2: Theme hochladen

Use Case Brief
Ein einfaches Theme besteht aus mehreren Komponenten gemäss *UC1: Komponenten überschreiben*, wobei auch andere Konzepte wie zum Beispiel ein Farbschema aus *US6: Farbgebung editieren* integriert werden können. Nachdem der Entwickler ein Theme erstellt hat, kann er es auf der Administrationsseite für den Kunden hochladen. Das Kernsystem führt schematische und strukturelle Prüfungen durch, um sicherzustellen, dass die bereitgestellte Schnittstelle für ein Theme korrekt verwendet wurde. Nach erfolgreicher Prüfung kann das Theme aktiviert werden. Langfristiges Ziel dieses Use Cases ist die Integration eines Marktplatzes für den Austausch von entwickelten Standard-Themes.

3.5.4 UC3: Komponentenlayout anpassen

Use Case Brief
Die Integration von *UC1: Komponenten überschreiben* ermöglicht es, die Darstellung einer Komponente zu definieren. Sie erlaubt es auch zu bestimmen, welche Daten innerhalb einer Komponente angezeigt werden. Darüber hinaus soll dem Entwickler die Möglichkeit gegeben werden, die Platzierung einer bestimmten Komponente eines Themes innerhalb der gesamten Seite zu definieren. Entsprechend dieser Definition positioniert das Kernsystem die Komponenten auf der Seite.

Use Case Szenario:
Die meisten Customize-Kunden platzieren die Produktklassierung für die Navigation oberhalb der Artikel im Webshop. Eine mögliche Darstellung ist in einem vereinfachten Wireframe in Abbildung 3.3 dargestellt.

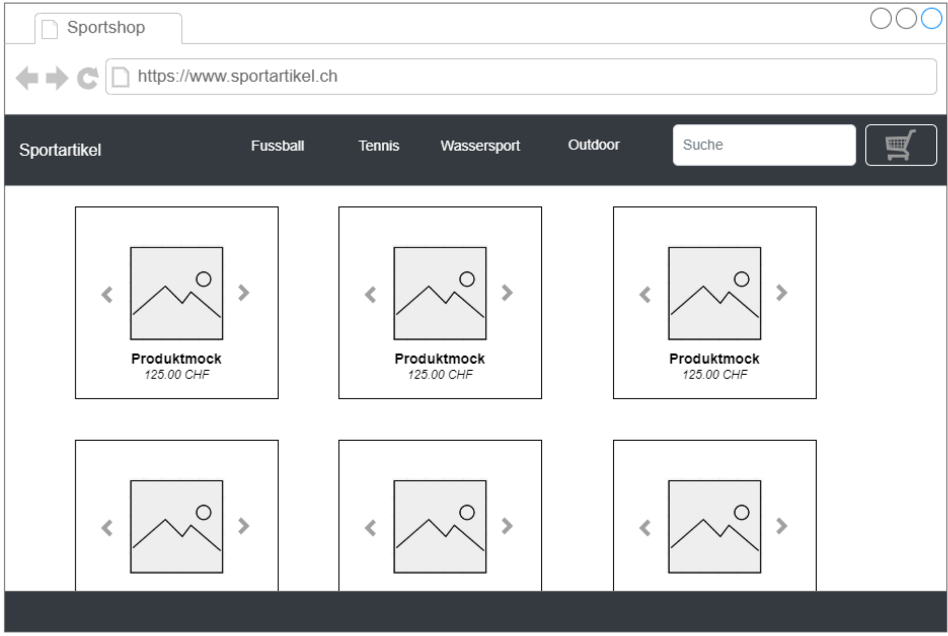


Abbildung 3.3: Wireframe Darstellung Klassierung oben

Der Kunde Sportartikel AG möchte diese Komponente jedoch auf auf der linken Seite neben den Produkten gemäss Darstellung in Abbildung 3.4 platzieren.

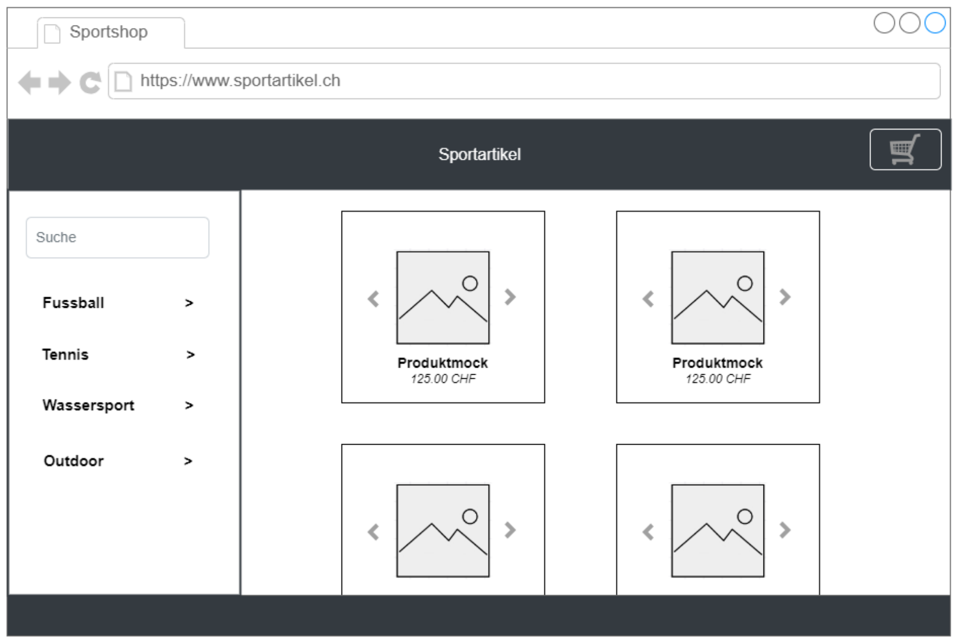


Abbildung 3.4: Wireframe Darstellung Klassierung links

3.5.5 Priorisierung

In der Tabelle 3.4 wird die Priorisierung der Use Cases des vorherigen Abschnittes aufgeführt und begründet.

Must Have Anforderungen	
Use Case	**Begründung**
UC1 - Komponenten überschreiben	Dieser Use Case deckt die Kernanforderung an das Theming-System ab und muss umgesetzt werden, um einen aussagekräftigen Prototypen zu erhalten, da das Überschreiben von Komponenten in einem Theme die meisten technischen Risiken birgt und einen grossen Einfluss auf die Architektur des Zielsystems hat.
UC3 - Komponentenlayout anpassen	Diese Anforderung wird von vielen Kunden gestellt und hat einen Einfluss auf die Architektur vom Theming-System.
Could Have Anforderungen	
Use Case	**Begründung**
UC2 - Theme hochladen	Für den Prototyp ist es grundsätzlich ausreichend, wenn ein Theme manuell in ein Repository geladen wird. Langfristig soll es jedoch eine Möglichkeit geben, ein neues Theme im Administrationsbereich hochzuladen und zu aktivieren. Damit soll es in Zukunft möglich sein, Standard-Themes über einen Marktplatz zur Verfügung zu stellen.

Tabelle 3.4: Priorisierung Anforderungen Theming-System

3.6 Plugin System

Als beispielhafte Anwendung, die durch das Plugin-System abgedeckt werden könnte, wurde folgende User Story definiert:

US14: Downloadcenter entwickeln

`Als Consultant oder Entwickler möchte ich in Form eines Plugins ein Downloadcenter entwickeln und im Webshop zur Verfügung stellen, sodass der eingeloggte Shopper Zugang zu seinen eingekauften digitalen Produkten erhält, die er herunterladen kann.`

3.6.1 Einleitung

Das Plugin-System hat die zentrale Aufgabe, den Entwicklern eine Möglichkeit zu bieten, das Storefrontsystem um zusätzliche Funktionalitäten zu erweitern. Es adressiert kundenspezifische Anforderungen, die vom Kernsystem nicht standardmässig bereitgestellt werden. Zusätzlich soll das Plugin-System die Möglichkeit bieten, an definierten Punkten die Logik des Kernsystems extern zu beeinflussen. Aus der Perspektive des Plugin-Systems wurden die in Abbildung 3.5 dargestellten Use Cases identifiziert.

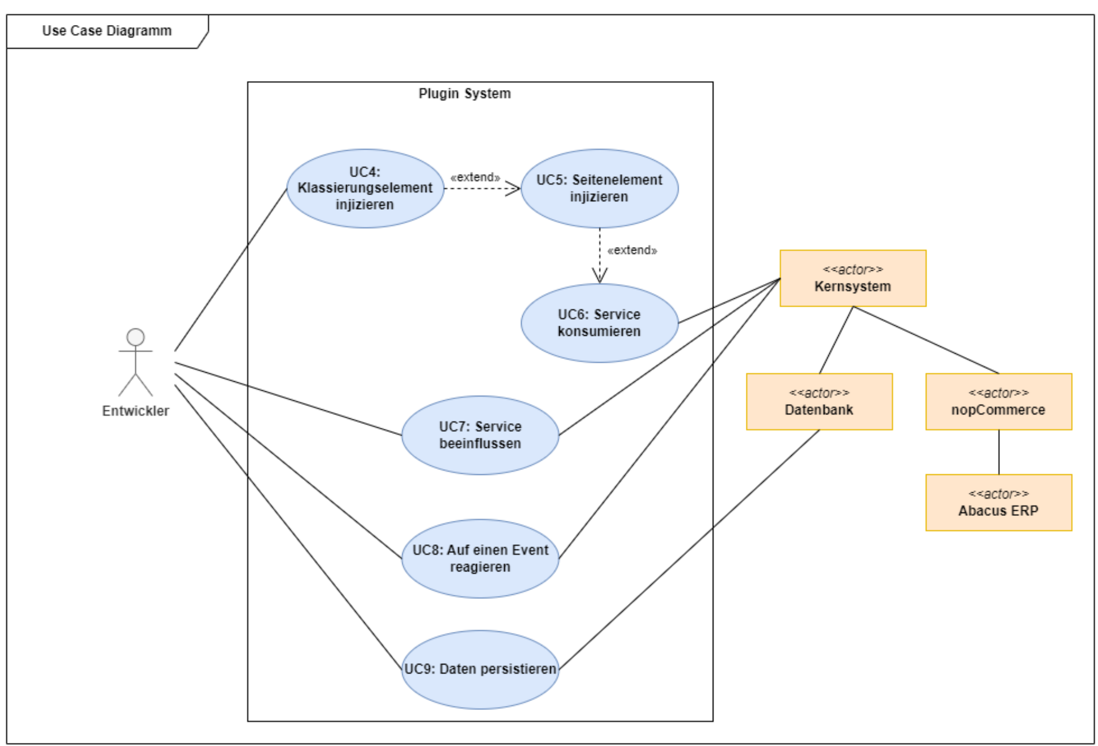


Abbildung 3.5: Use Cases des Plugin Systems (UML Use Case Diagramm)

3.6.2 UC4: Klassierungselement injizieren

Use Case Brief

Die Produktklassierung wird im Kernsystem von nopCommerce abgerufen und aufgebaut. Ein Plugin-Entwickler soll die Möglichkeit haben, eigene Klassierungselemente an der gewünschten Stelle in die Produktklassierung zu integrieren. Ein Klassierungselement besteht aus einem Namen und einem Uniform Resource Locator (URL)-Slug.[1] Bei einem Aufruf der Produktklassierung zeigt das Kernsystem auch die vom Plugin eingefügten Elemente an der definierten Stelle im Klassierungsbaum an. Bei der Auswahl des Elements wird auf den definierten Slug weitergeleitet. Neben der Produktklassierung sollen auch in der Navigation des Adminbereichs eigene Elemente injiziert werden können.

3.6.3 UC5: Seitenelement injizieren

Ein Seitenelement ist die ganze Seite, welche beim Aufruf eines Slugs dargestellt wird. In Abbildung 3.6 ist ein Wireframe eines Seitenelementes dargestellt.

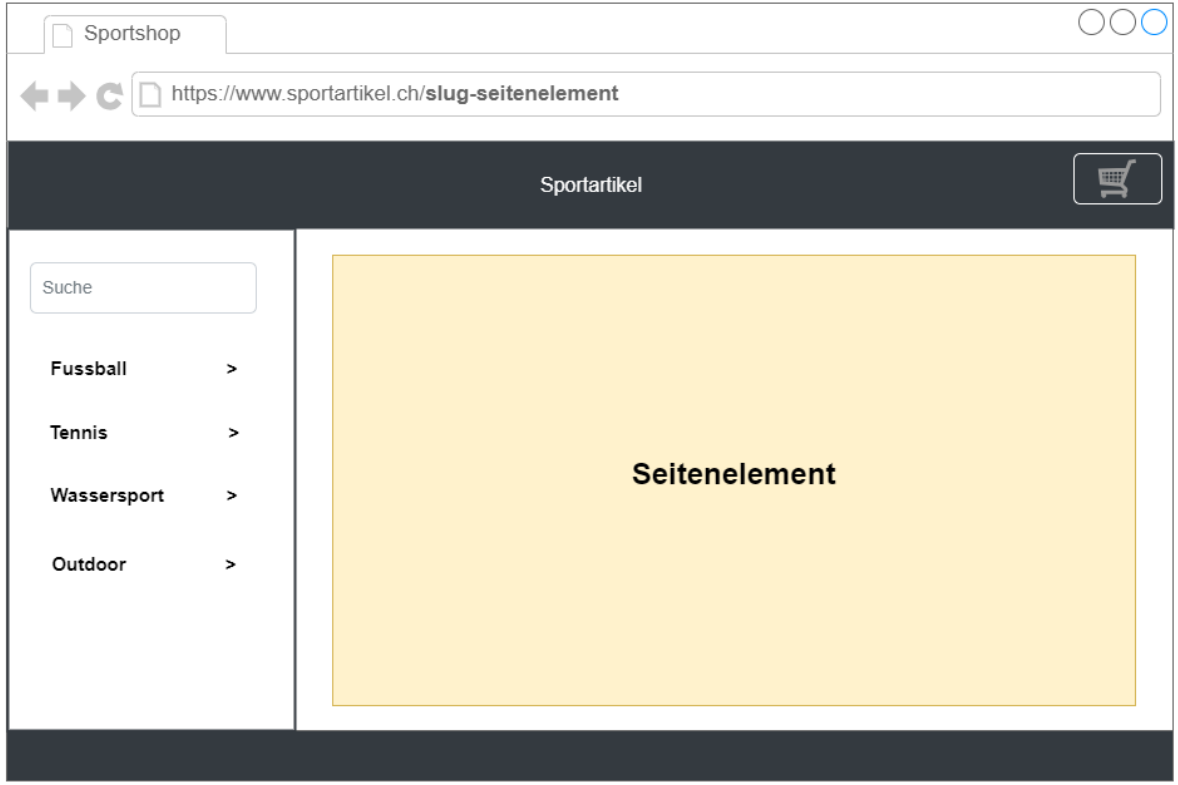


Abbildung 3.6: Wireframe Seitenelement

[1]Ein Slug ist der letzte Teil des URL-Pfades, der auf eine bestimmte Seite im Storefrontsystem verweist, die beim Aufruf dieser URL angezeigt wird [44].

Use Case Brief
Eine zentrale Funktionalität des Plugin-Systems ist die Möglichkeit für Entwickler, eigene Seiten in das Kernsystem zu integrieren. Das Plugin legt fest, unter welchem Slug die eingefügte Seite angezeigt wird. Sobald dieser Slug aufgerufen wird, zeigt das Kernsystem die injizierte Seite an. Auf dieser Seite kann der Entwickler verschiedene Funktionen implementieren, die von den Kunden als Erweiterungen gewünscht werden. Es soll möglich sein, eigene Seiten sowohl im öffentlich zugänglichen Webshop als auch auf der Administrationsseite einzufügen. Der zuvor beschriebene Use Case *UC4: Klassierungselement injizieren* beschreibt die Möglichkeit für Plugin-Entwickler, ein eigenes Klassierungselement in die Baumstruktur der Produktklassierung oder in die Adminnavigation zu integrieren. Eine Standardanwendung wäre, dass der Slug auf dem Klassierungselement auf eine eingefügte Seite dieses Use Cases verweist.

3.6.4 UC6: Service konsumieren

Use Case Brief
In *UC5: Seitenelement injizieren* wird die Anforderung beschrieben, eigene Seitenelemente in das Storefrontsystem zu integrieren. Ein Plugin-Entwickler soll dabei die Möglichkeit haben, die Standard-Services des Kernsystems zu nutzen. Beim Aufruf eines Seitenelements stellt das Kernsystem dem Plugin eine Schnittstelle zur Verfügung, damit die benötigten Services genutzt werden können.

Die drei zuletzt beschriebenen Anforderungen UC4 bis UC6 sind eng miteinander verknüpft, können aber auch unabhängig voneinander realisiert werden. Durch ihr Zusammenspiel ist es möglich, dass ein Plugin ein Klassierungselement in die Produktnavigation integriert (*UC4: Klassierungselement injizieren*), das einen Slug enthält. Dieser Slug verweist wiederum auf ein Seitenelement, welches durch das Plugin hinzugefügt wurde (*UC5: Seitenelement injizieren*). Innerhalb dieses Seitenelements werden die vom Kernsystem bereitgestellten Services genutzt, um die spezifischen Ziele des Plugins zu erreichen (*UC6: Service konsumieren*).
Das folgende Use-Case-Szenario konkretisiert die beispielhafte User Story *US13: Produktkonfigurator entwickeln* an einem konkreten Beispiel des Musterkunden anhand der vorangegangenen Use Cases des Plugin-Systems.

Use Case Szenario: Konfigurator
Die Sportartikel AG bietet einen speziellen Konfigurator für Fussballtrikots an. Der Kunde kann ein Trikot nach seinen Wünschen gestalten, indem er die Farbe des Trikots auswählt und beliebige Texte oder Bilder auf vordefinierten Flächen platziert. Das Trikot wird entsprechend der Konfiguration des Kunden hergestellt und ausgeliefert. Ein Entwickler möchte diese Anforderung mit einem Plugin umsetzen. Er injiziert auf Basis von *UC4: Klassierungselement injizieren* ein Klassierungselement unter dem Pfad *Fussball / Kleidung* mit dem Namen *Trikot gestalten* und dem Slug */configurator*. Der Entwickler programmiert den Konfigurator und teilt dem Kernsystem mit, dass die Seite über */configurator* (UC5) aufgerufen werden kann. Um das Produkt in den Warenkorb legen zu können, verwendet er den entsprechenden Service, den das Kernsystem bereitstellt (UC6).

3.6.5 UC7: Verhalten Kernsystem beeinflussen

Use Case Brief

Ein Entwickler sollte in der Lage sein, das Verhalten des Kernsystems an vordefinierten Punkten, sogenannten Erweiterungspunkten, zu beeinflussen. Sobald im Kernsystem ein Service ausgeführt wird, der von einem Plugin beeinflusst wird, führt das Kernsystem die Logik des Plugins anstelle der Standardfunktionalität aus. Folgende Erweiterungspunkte sind mögliche Stellen im Kernsystem, die überschrieben werden könnten:

- Preisermittlung eines Produktes
- Rabattermittlung eines Produktes
- Berechnung der Mehrwertsteuer (MwSt)
- Hinzufügen eines Artikels zum Warenkorb

Use-Case-Szenario: Umsatzsteuer Deutschland verrechnen

Da die Sportartikel AG einen erheblichen Teil ihres Umsatzes in Deutschland erzielt, soll für diese Kunden die deutsche Mehrwertsteuer beim Verkauf berücksichtigt werden, damit sie beim Empfang der Produkte keine zusätzlichen Zahlungen leisten müssen. Durch die Anmeldung der Sendung bei einem externen Dienstleister kann die Sportartikel AG sicherstellen, dass die Sendung letztendlich ohne zusätzliche Kosten für den Empfänger verzollt wird. Am Ende erhält die Sportartikel AG eine Rechnung mit deutscher Mehrwertsteuer. Standardmässig werden Exportverkäufe im Kernsystem jedoch mit 0% Mehrwertsteuer fakturiert, sodass der Empfänger die Steuer im Bestimmungsland entrichtet. Um diese spezifische Kundenanforderung zu erfüllen, entwickelt der Entwickler ein Plugin, das die Anmeldung der Sendung beim externen Dienstleister durchführt. Innerhalb dieses Plugins muss der Entwickler die Möglichkeit haben, die MwSt-Berechnung des Kernsystems so anzupassen, dass anstelle von 0% der korrekte Steuersatz aus Deutschland berechnet wird.

3.6.6 UC8: Auf einen Event reagieren

Use Case Brief
Ein Plugin soll die Möglichkeit erhalten, über bestimmte Ereignisse innerhalb des Kernsystems informiert zu werden. Ein Beispiel hierfür könnte der Event *Kunde X hat Artikel Y in den Warenkorb gelegt* sein. Der Entwickler des Plugins registriert sein Plugin für ein Ereignis. Sobald der entsprechende Vorgang im Kernsystem stattfindet, wird das Ereignis ausgelöst und das Plugin informiert. Der Entwickler des Plugins kann dann individuell mit seiner definierten Logik darauf reagieren.

Use-Case-Szenario: Werbeartikel
Die Sportartikel AG hat einen neuen Proteinriegel im Angebot. Um diesen Artikel zu bewerben, möchten sie bestimmten Kunden diesen Riegel gratis zu ihrer Bestellung hinzufügen. Das Plugin prüft bei jedem Artikel, der in den Warenkorb gelegt wird, um welchen Artikel es sich handelt. Damit möchte die Sportartikel AG sicherstellen, dass nur Kunden der gewünschten Zielgruppe den Riegel erhalten. Wenn der Artikel *Y* zum Produktklassierungselement *Fitness* gehört, wird der Riegel mit einem Verkaufspreis von 0 CHF in den Warenkorb gelegt. Dies könnte durch die Anwendung von *UC6: Service konsumieren* durchgeführt werden.

3.6.7 UC9: Daten persistieren

Use Case Brief
Ein Entwickler des Plugins benötigt eine Möglichkeit, die Daten seines Plugins zu persistieren. Das Kernsystem stellt dem Plugin seine Infrastruktur dazu zur Verfügung.

Use-Case-Szenario: Anmeldung deutsche Umsatzsteuer
Im Use-Case-Szenario *Umsatzsteuer Deutschland verrechnen* wird die Anforderung thematisiert, dass mittels eines Plugins die Mehrwertsteuer für Deutschland verrechnet wird. Die Sportartikel AG muss die verrechneten Steuern in einem bestimmten Intervall einem externen Dienstleister melden, damit dieser die verrechnete Mehrwertsteuer an die Sportartikel AG zurückführen kann. Die Sportartikel AG möchte diese Meldung nun automatisiert aus dem Storefrontsystem generieren lassen. Diese Anforderung kann mit den vorgestellten Use Cases des Plugin Systems erfüllt werden. Der Entwickler des Plugins benötigt jedoch eine Möglichkeit zu verfolgen, welche Bestellungen bereits an den deutschen Dienstleister übermittelt wurden. Mit der Anwendung von *UC9: Daten persistieren* kann der Entwickler im Plugin mit Hilfe vom Kernsystem die benötigten Daten persistieren.

3.6.8 Priorisierung

In der Tabelle 3.5 wird die Priorisierung der Use Cases des vorangegangen Abschnittes aufgeführt und begründet.

Must Have Anforderungen	
Use Case	**Begründung**
UC4 - Klassierungselement injizieren	Die Anforderung *UC5: Seitenelement injizieren* beschreibt die Möglichkeit, zusätzliche Seitenelemente als Plugin in das Storefrontsystem einzufügen. Das Einfügen eines zusätzlichen Klassierungselements ermöglicht es dem Shopper, diese Seite über den Link auf dem Klassierungselement zu finden. Dieser Use Case wird als wichtiger Unterstützungsprozess für UC5 priorisiert.
UC5 - Seitenelement injizieren	Dieser Use Case deckt die wichtigste Funktionalität des Plugin-Systems ab.
UC6 - Service konsumieren	Die Verwendung von Services aus dem Kernsystem ermöglicht einem Plugin, Funktionalitäten des Kernsystems zu nutzen. Dazu gehört beispielsweise die Interaktion mit nopCommerce. Dieser Use Case eröffnet dem Entwickler eines Plugins die Möglichkeit, individuelle Anforderungen breiter abzudecken.
Should Have Anforderungen	
Use Case	**Begründung**
UC7 - Service beeinflussen	Die Beeinflussung der Funktionalität des Kernsystems erhöht die Individualisierungsmöglichkeiten des Storefrontsystems und ist folglich ein wichtiger Use Case. Dieser Use Case wurde als 'Should Have' eingestuft, um den Umfang dieser Arbeit einzugrenzen.
UC8 - Auf einen Event reagieren	Dieser Use Case ermöglicht viele zusätzliche Möglichkeiten mit dem Plugin-System.
Could Have Anforderungen	
Use Case	**Begründung**
UC9 - Daten persistieren	Wird für ein Minimum Viable Product (MVP) nicht benötigt.

Tabelle 3.5: Priorisierung Anforderungen Plugin System

3.7 Headless Nutzung

Als Grundlage für das gesamte Storefrontsystem muss gemäss der definierten Aufgabenstellung (siehe Anhang C) eine Lösung gefunden werden, wie das in der Studienarbeit [8] evaluierte E-Commerce-System nopCommerce 'headless' zur Verfügung gestellt werden kann. Die Nutzung als 'headless' System bezeichnet den Betrieb eines Systems ohne eine vorgegebene Präsentationsschicht, was eine hohe Flexibilität in der Art und Weise ermöglicht, wie Inhalte ausgeliefert und dargestellt werden. Bei einem 'headless' System werden die Inhalte über Application Programming Interfaces (APIs) bereitgestellt, was Entwicklern erlaubt, diese Inhalte in beliebigen Frontend-Frameworks oder Plattformen zu verwenden [45]. Dies ist die Voraussetzung dafür, dass das Storefrontsystem die Stammdaten aus nopCommerce beziehen kann, welches über einen Connector in das Abacus ERP des Kunden integriert ist.

3.8 Nichtfunktionale Anforderungen

Nachdem in den vorangegangen Abschnitten die fachlichen Anforderungen an das Storefrontsystem aufgezeigt wurden, sind in diesem Abschnitt die nichtfunktionalen Anforderungen (NFA) an das zukünftige System aufgeführt. Diese Anforderungen werden von Customize gestellt, um die Qualität des zukünftigen Storefrontsystems für die Kunden und Consultants sicherzustellen.

3.8.1 NFA1: Performance

Performance ist ein wichtiges Qualitätsmerkmal des neuen Storefrontsystems. Dies gilt insbesondere für alle Systemteile, die für die Endkunden der Customize-Kunden (Shopper) öffentlich zugänglich sind. In Absprache mit dem Industriepartner ist es nicht vorgesehen, ein Performancedach im Vergleich zur heutigen 'AbaShop'-Lösung zu definieren. Die Architektur der beiden Systeme ist zu unterschiedlich, als dass dies sinnvoll verglichen werden könnte.
Es wird gefordert, dass bei architekturrelevanten Entscheidungen Performancevergleiche durchgeführt werden, um die Performanceeinbussen einer möglichen Entscheidung zu gewichten. Im System sollen alle Seiten, welche die Shopper aufrufen, innerhalb von maximal zwei Sekunden geladen werden.

3.8.2 NFA2: Zukunftssicherheit

Abacus hat als Enterprise Resource Planning (ERP)-System eine ausserordentlich lange Lebensdauer. Auch wenn das Storefrontsystem über die RESTful HTTP-Schnittstellen von Abacus entkoppelt ist, verlangen die Kunden aufgrund der getätigten Investition in das Abacus ERP eine entsprechend lange Lebensdauer des Webshop-Systems. Das System soll mindestens zwei Updatezyklen des ERP überleben, was einer Lebensdauer von sechs Jahren entspricht. Der Industriepartner macht keine Vorgaben bezüglich der eingesetzten Bibliotheken. Zur Sicherstellung dieser nichtfunktionalen Anforderung soll jedoch möglichst nicht direkt gegen die Implementierung programmiert werden, um die Abhängigkeit zu reduzieren und einen Austausch der verwendeten Bibliothek bei Bedarf zu erleichtern. Weiterhin soll darauf geachtet werden, dass die eingesetzten Bibliotheken, Programmiersprachen und Programmierkonzepte eine möglichst hohe Verbreitung in der Softwareindustrie haben.

3.8.3 NFA3: Einfachheit eines Themes

Im Abschnitt 3.2 wurde der Akteur Consultant, ein Mitarbeiter von Customize, eingeführt. Ein Consultant eines Kunden mit einem Storefrontsystem ist in der Regel technisch versiert. Die Programmierfähigkeiten eines einzelnen Consultants können jedoch variieren. Aus diesem Grund müssen einfache Anpassungen an einem Theme auch für Consultants mit geringeren Programmierkenntnissen möglich sein. Die Verifikation soll durch einen Usertest mit ausgewählten Customize-Mitarbeitern erfolgen. Ein Consultant, der mit dem entwickelten System nicht vertraut ist, sollte nach einer kurzen Einführung in der Lage sein, innerhalb von zehn Minuten eine einfache Kundenanforderung in einem Theme umzusetzen. Beispiele für einfache Aufgaben sind das Hinzufügen einer neuen Variable zu einer Komponente oder die visuelle Anpassung eines Textes.

3.8.4 NFA4: Search Engine Optimierung

Ein wichtiges Qualitätsmerkmal ist ein Search Engine optimiertes System. Suchmaschinenoptimierung (SEO) bezieht sich auf den Prozess der Verbesserung der Sichtbarkeit einer Website oder einer Webseite in den unbezahlten Ergebnissen einer Suchmaschine, oft als organische, natürliche oder verdiente Ergebnisse bezeichnet. Ziel ist es, die Position der Website in den Suchergebnissen zu erhöhen, um mehr Besucher anzuziehen, was durch die Optimierung von Inhalten, die Verbesserung der technischen Performance und das Erreichen von Relevanz durch Schlüsselwörter und Backlinks erreicht wird [12]. Für die Kunden der Customize ist es essentiell, dass ihr Storefrontsystem in den Suchmaschinen einfach gefunden wird.
Die Suchmaschinenoptimierung ist ein aktives Forschungsgebiet, weshalb im Kontext dieser Arbeit auf die Basisanforderungen für ein SEO-optimiertes System geachtet werden soll. Zur Verifikation sollen in der Google Lighthouse Analyse in der Kategorie SEO mindest 80 von 100 Punkten erreicht werden [19].

3.8.5 NFA5: Verfügbarkeit

Die Kunden des Industriepartners erwarten eine hohe Verfügbarkeit des Storefrontsystems, sodass ihre Kunden jederzeit Einkäufe tätigen können. Ein Update des Abacus ERP kann beispielsweise bei einem grösseren Kunden zu Unterbrechungen von bis zu zwei Tagen führen. Das Storefrontsystem sollte während Wartungsarbeiten für maximal eine Stunde nicht erreichbar sein. Zudem ist es wichtig, dass die Funktionsfähigkeit des Systems auch bei einem Ausfall des Abacus gewährleistet bleibt. Es kann vorkommen, dass unter diesen Umständen bestimmte Funktionalitäten nicht verfügbar sind, jedoch sollte dies nur in begründeten Fällen akzeptiert werden.

3.8.6 NFA6: Updatefähigkeit

Das Storefrontsystem wird potenziell bei einer Vielzahl unterschiedlicher Kunden eingesetzt. Mit der Integration eines Theming- und Plugin-Systems wird sichergestellt, dass die individuellen Anforderungen jedes Kunden erfüllt werden. Um eine zukünftige Updatefähigkeit mit möglichst geringem Aufwand für jeden Kunden zu ermöglichen, ist darauf zu achten, dass der Kern des Systems trotz individueller Anforderungen bei jedem Kunden identisch bleibt. Die Verifikation erfolgt durch Benutzertests. Dabei werden die konkreten Anforderungen der Kunden durch einen Customize-Consultant im Prototyp umgesetzt.

Kapitel 4

Verwandte Arbeiten

In Kapitel 3 wurde in der Dokumentation der funktionalen Anforderungen die Einführung eines Theming- und Plugin-Systems als zentrale Funktionalitäten des Storefrontsystems erläutert. Solche Konzepte sind nicht neu und bereits in vielen aktuellen Systemen integriert. In diesem Kapitel werden bestehende Lösungen für derartige Konzepte untersucht, mit dem Ziel, theoretische Lösungsansätze oder praktische Architekturkonzepte zu identifizieren, die als Basis für die eigene Implementierung dienen können. Die technischen Details der untersuchten Lösungen, insbesondere jene, die für die Entwicklung ähnlicher Konzepte im System notwendig sind, werden in Kapitel 5 vertieft, um gezielt die für die Umsetzung relevanten Aspekte herauszuarbeiten.

4.1 Plugin

Nach einer Einführung in den Begriff 'Plugin' werden in diesem Abschnitt bestehende Plugin-Systeme untersucht, um daraus relevante Schlüsse für die Umsetzung im Storefrontsystem zu ziehen.

4.1.1 Definition

Der Begriff *Plugin* leitet sich vom englischen Ausdruck *to plug in* ab, was so viel bedeutet wie 'einstecken' oder 'anschliessen'. In der Softwareentwicklung beschreibt ein Plugin eine Softwarekomponente, die in ein bestehendes System 'eingesteckt' wird um diese zu erweitern oder zu verändern.

Plugins werden in vielen Softwareprodukten verwendet, wobei oft von *pluginfähig* gesprochen wird. Es gibt dabei verschiedene Definitionen von *Plugin* und keine genaue Begriffsbestimmung, die von jedem Hersteller identisch verwendet wird. Eichler definiert in seinem Paper [11] ein Plugin als dynamische Komponente, die Funktionalität zur Verfügung stellt, um ein Softwaresystem zu erweitern. Der Begriff Dynamik bedeutet in diesem Kontext, dass eine Komponente zur Laufzeit geladen werden kann. Dies entspricht jedoch nicht der Realität in vielen Plugin-Systemen. Beispielsweise wurden in der integrierten Entwicklungsumgebung (IDE) von *Eclipse* Plugins lange Zeit lediglich zum Start der Software geladen. Durch Weiterentwicklungen im Projekt *equinox* [9] besteht mittlerweile die Möglichkeit, Eclipse-Plugins auch zur Laufzeit zu laden. Eclipse wurde jedoch schon vor diesem Projekt als Plugin-System betrachtet, weshalb die Dynamik der Komponenten kein integraler Bestandteil ihres Plugin-Systems ist. Zu dieser Aussage gelang auch Duvigneau in seiner Dissertation [7], indem er feststellt, dass ein Plugin-System

dynamisch rekonfigurierbar sein kann, aber nicht muss. Im vorgesehenen Plugin-System dieser Arbeit ist die Dynamik ebenfalls nicht der entscheidende Teil, vielmehr ist die von Eichler erwähnte Möglichkeit der Erweiterung eines Softwaresystems durch zusätzliche Funktionalität von Interesse.

Eine passende Definition für den vorgesehenen Anwendungsfall liefern Marquardt und Völter [29]:

> *'Plug-Ins sind Ergänzungen zu einer Applikation, die von dieser vorhergesehen sind, die sie aber nicht selbst mitbringen kann oder will. Ein Plug-In deckt dabei eine fachliche Funktion so vollständig ab, dass keine andere Komponente des Systems irgendetwas spezifisches darüber wissen müsste. Für die dazu nötigen Schnittstellen ist die Applikation verantwortlich.'*

Sie sehen Plugins als Ergänzung von Funktionalität, die eine Hostapplikation nicht mitbringt. Die Hostapplikation entspricht dem Akteur Kernsystem, der in Abschnitt 3.4 eingeführt wird. Im Gegensatz zu dieser Definition soll im Storefrontsystem ein Plugin nicht nur vollständige fachliche Funktionalitäten abdecken, sondern auch das Verhalten von Komponenten der Hostapplikation übersteuern. Dies wird benötigt, um Use Case *Service beeinflussen* umzusetzen.

4.1.2 Eclipse

Eclipse ist eine weit verbreitete integrierte Entwicklungsumgebung[1], die vor allem für Java-Entwicklung genutzt wird, aber auch für viele andere Programmiersprachen durch eine Vielzahl von Plugins erweiterbar ist. Im Folgenden werden die wichtigsten Konzepte und Strukturen vom Eclipse Plugin-System erläutert. Die verwendeten Informationen stammen aus der offiziellen Eclipse-Dokumentation [10].

Das Plugin-System von Eclipse basiert auf einem modularen Architekturprinzip, das die Erweiterbarkeit und Anpassungsfähigkeit der Entwicklungsumgebung ermöglicht. Im Kern dieser Architektur steht das Konzept der **Extension Points** und **Extensions**. Ein Extension Point ist eine definierte Stelle innerhalb des Kernsystems, an der Plugins ihre Funktionalitäten anbinden können. Ein Extension Point stellt eine definierte Schnitttelle in Form eines Interfaces zur Verfügung, welche ein Plugin implementieren muss. Die Definition eines Extension Points und die Bindung von Extensions erfolgt in der *plugin.xml*-Datei des Plugins. Interessanterweise können Plugins selbst Extension Points definieren, welche von anderen Plugins genutzt werden können. Zur Anwendung einer Extension muss in der Regel ein von Eclipse definiertes Interface implementiert werden, das die jeweilige Funktionalität bereitstellt.

Eclipse bietet verschiedene Erweiterungspunkte zur Integration und Erweiterung der Benutzeroberfläche. Dazu gehören `org.eclipse.ui.menus`, um Menüpunkte und Werkzeugleisten hinzuzufügen, `org.eclipse.ui.views`, um Ansichten zu integrieren, und `org.eclipse.ui.editors`, um Editoren bereitzustellen. Durch diese Extension Points können Plugins die Benutzeroberfläche dynamisch erweitern und anpassen.

[1] Eine integrierte Entwicklungsumgebung (IDE) ist eine Software-Anwendung, die Entwicklern eine umfassende Sammlung von Werkzeugen in einer einzigen Benutzeroberfläche bietet, um den Prozess der Softwareerstellung durch Funktionen wie Code-Editor, Compiler, Debugger und automatisches Build-Management zu vereinfachen und zu beschleunigen.

4.1.3 WordPress

WordPress ist ein weit verbreitetes Open-Source-Content-Management-System (CMS)[2], das in PHP geschrieben ist und MySQL als Datenbank verwendet. Es ermöglicht Benutzern, Webseiten und Blogs einfach zu erstellen und zu verwalten, und bietet umfangreiche Anpassungsmöglichkeiten durch Plugins. Im Folgenden werden die wichtigsten Konzepte und Strukturen erläutert. Diese Informationen stammen aus der offiziellen WordPress-Dokumentation [57].

Actions und Filters sind zwei Arten von Hooks[3], welche zentrale Bestandteile des Plugin-Systems darstellen. Actions erlauben es, benutzerdefinierte Funktionen an bestimmten Punkten im WordPress-Kern auszuführen, während Filters dazu genutzt werden, Daten zu modifizieren, bevor sie gespeichert oder angezeigt werden. Diese Mechanismen bieten eine leistungsstarke Möglichkeit, auf Ereignisse zu reagieren und Daten dynamisch zu bearbeiten.

Shortcodes bieten eine einfache Möglichkeit, komplexere Inhalte in Beiträge oder Seiten einzubinden, indem sie es Benutzern ermöglichen, spezielle Tags zu verwenden, die WordPress veranlassen, bestimmte Funktionen auszuführen. Diese können von der Darstellung einer Fotogalerie bis hin zum Einbetten von Videos reichen und verbessern so die Interaktivität und Benutzerfreundlichkeit von Webseiten.

Die Widget API ist ein weiteres wichtiges Konzept im WordPress Plugin-System. Sie ermöglicht Entwicklern, Widgets[4] zu entwickeln, welche Benutzer über ein einfaches Drag-and-Drop-Interface in ihre Seiten oder Beiträge integrieren können. Widgets können Funktionen wie Suchfelder, neueste Kommentare, Kalender und mehr beinhalten, die zur Erweiterung der Interaktion und Funktionalität von WordPress-Seiten beitragen.

Ein weiteres wichtiges Feature von WordPress Plugins ist die Möglichkeit, eigene Menüpunkte in der WordPress-Admin-Navigation zu erstellen. Dies ermöglicht es Entwicklern, ihre Plugin-Einstellungen oder benutzerdefinierte Admin-Seiten direkt in das Dashboard zu integrieren.

[2] Ein Content-Management-System (CMS) ist eine Softwareanwendung, die es Benutzern ermöglicht, digitale Inhalte zu erstellen, zu verwalten und zu veröffentlichen, oft ohne tiefgehende Programmierkenntnisse zu benötigen.

[3] Ein Hook in der Softwareentwicklung ist ein Mechanismus, der es Entwicklern ermöglicht, den Ablauf eines Programms durch benutzerdefinierte Funktionen zu modifizieren oder zu erweitern, indem sie an spezifischen Punkten in das Verhalten des Programms eingreifen.

[4] Widgets in WordPress sind visuelle Komponenten, die Benutzern ermöglichen, Funktionen und Inhalte wie Kalender, Suchfelder oder Textblöcke interaktiv in Seitenleisten oder Fusszeilen der Website einzufügen.

4.1.4 Eignung für das Storefrontsystem

Das Plugin-System von Eclipse ermöglicht es, dass Plugins über definierte Erweiterungspunkte Funktionalität in das Kernsystem einbringen können. Durch festgelegte Schnittstellen ist exakt definiert, welche Implementation ein Plugin erfüllen muss, um einen bestimmten Erweiterungspunkt nutzen zu dürfen. Dieses Prinzip wird auch in der Implementierung des Storefrontsystems angewendet, wobei die Definition der erforderlichen Schnittstellen mittels TypeScript erfolgt (siehe Abschnitt 5.7).
Ein weiteres interessantes Konzept ist das Hook-System von WordPress, das zwischen Filtern und Actions unterscheidet. Ein Filter-Hook ermöglicht es einem Plugin, Daten zu modifizieren, bevor diese angezeigt oder gespeichert werden, was dem Use Case *Service beeinflussen* entspricht. Im Gegensatz dazu liegt der Schwerpunkt bei WordPress auf der Modifikation von Daten für die visuelle Darstellung. WordPress erlaubt es auch, eigene Einträge in der Admin-Navigation zu erstellen, eine Funktionalität, die gemäss Abschnitt 5.7.5 im Storefrontsystem implementiert wird (Use Case*Klassierungselement injizieren*). Die Umsetzung dieses Use Cases ist vergleichbar mit einem Filter-Hook in WordPress. Ein Plugin erhält beim Aufruf die gesamte Navigation und kann diese modifizieren, indem es eigene Navigationselemente einfügt und diese dann zurückgibt.

4.2 Theme

Nach einer Einführung in den Begriff 'Theme' wird in diesem Abschnitt das Theming-System von WordPress untersucht, um daraus relevante Schlüsse für die Umsetzung im Storefrontsystem zu ziehen.

4.2.1 Definition

Eine allgemeine Definition des Begriffs *Theme* ist schwierig zu finden. Die grösste Verbreitung finden Themes in UI-Bibliotheken[5], beispielsweise die Bibliothek Material UI (MUI), welche ein Theme folgendermassen definiert [51]:

> *'The theme specifies the color of the components, darkness of the surfaces, level of shadow, appropriate opacity of ink elements, etc. Themes let you apply a consistent tone to your app. It allows you to customize all design aspects of your project in order to meet the specific needs of your business or brand.'*

Diese Definition entspricht jedoch nicht vollständig dem Verständnis eines Themes im Kontext dieser Arbeit. Während ein Theme in MUI hauptsächlich Attribute wie Farben, Schatten oder Abstände einzelner Komponenten steuert, geht das Konzept eines Themes in einem Storefrontsystem deutlich darüber hinaus. Ein Theme im Storefrontsystem beeinflusst das gesamte Layout und die Struktur der Anwendung. Dies umfasst nicht nur die visuellen Attribute wie Farben und Abstände, sondern auch die Anordnung und Auswahl der anzuzeigenden Daten. Ein solches Theme bestimmt, welche Daten auf welchen Komponenten dargestellt werden und wie diese Komponenten innerhalb des gesamten Layouts positioniert sind.

[5] Eine UI-Bibliothek ist eine Sammlung von vorgefertigten Benutzeroberflächenkomponenten und Tools, die Entwicklern dabei hilft, konsistente und wiederverwendbare Designs und Funktionalitäten in Anwendungen zu erstellen.

Der wesentliche Unterschied besteht also darin, dass ein Theme in MUI zentrale Einstellungen zur Vereinheitlichung der Erscheinung einzelner Komponenten bietet, während ein Theme im Kontext dieser Arbeit das gesamte Layout und die Datenpräsentation steuert. Es legt nicht nur die optischen Eigenschaften fest, sondern definiert auch, welche Informationen angezeigt werden und wo sie positioniert sind.

Eine ähnliche Definition eines Themes trifft WordPress [56]:

> *'A WordPress theme represents the design of your website. It can control everything from colors, to fonts, to the entire layout. In essence, what you see when viewing the front-end of your site is shaped by the theme.'*

Im nachfolgenden Abschnitt wird aufgezeigt, wie das Theming-System von WordPress funktioniert und welche Möglichkeiten dort zur Verfügung stehen.

4.2.2 WordPress-Themes

Das Theming-System in WordPress ermöglicht es, das Erscheinungsbild und das Layout einer Website zu verändern, ohne den Kerncode der Anwendung zu modifizieren. Die Hauptkomponenten eines WordPress-Themes sind Template-Dateien und Stylesheets. Im Folgenden werden die wichtigsten Konzepte und Strukturen erläutert. Diese Informationen stammen aus der offiziellen WordPress-Dokumentation [57].

Ein WordPress-Theme setzt sich typischerweise aus verschiedenen Template-Dateien zusammen, wie beispielsweise *header.php* für den Kopfbereich, *footer.php* für den Fussbereich und *index.php* für die Hauptseite. Interessant ist dabei, dass ein Template wiederum die anderen Templates nutzen kann, womit ein Theme die Möglichkeit erhält, die Darstellung der gesamten Seite zu beeinflussen. Zudem sind Stylesheets entscheidend (*style.css*), die das visuelle Design durch CSS-Regeln[6] definieren.

Das Theme-System ermöglicht es Designern und Entwicklern, sogenannte 'Child Themes' zu erstellen. Diese erben die Funktionalität eines 'Parent Themes', ermöglichen jedoch Modifikationen und Anpassungen ohne Änderungen am ursprünglichen Code des 'Parent Themes'. Dies ist besonders nützlich, um Anpassungen vorzunehmen, die über Updates des 'Parent Themes' hinweg bestehen bleiben.

Darüber hinaus bietet das Theme-System Funktionen wie Template-Tags und Conditional-Tags, die Entwicklern helfen, Inhalte dynamisch zu laden und spezifische Bedingungen für die Anzeige von Inhalten zu setzen. Beispielsweise können Conditional-Tags verwendet werden, um unterschiedliche Header-Dateien für verschiedene Seiten oder Beiträge zu laden.

[6]CSS-Regeln sind Anweisungen in Cascading Style Sheets, die festlegen, wie HTML-Elemente auf einer Webseite in Bezug auf Layout, Farben und Schriftarten gestaltet werden sollen.

4.2.3 Eignung für das Storefrontsystem

WordPress nutzt ein Templating-System, das die Übersteuerung spezifischer Webseitenbereiche innerhalb eines Themes ermöglicht. Ein ähnliches Konzept ist auch für das Storefrontsystem erforderlich. Der Use Case *Komponenten überschreiben* spezifiziert diese Anforderung, da ein Template die Darstellung einer Komponente bestimmt. Die entsprechende Implementierung wird in Abschnitt 5.6.2 erläutert. Das Templating-Konzept von WordPress ermöglicht es darüber hinaus, dass eine Template-Datei andere Templates einbinden kann. Dies ermöglicht zum Beispiel, dass das Template *index.php* andere Plugin-Templates an den gewünschten Stellen innerhalb der gesamten Webseite einfügt. Diese Funktionalität wird im Storefrontsystem durch den Use Case *Komponentenlayout anpassen* abgedeckt, dessen Implementierungsdetails in Abschnitt 5.6.4 beschrieben werden.

Kapitel 5

Design und Implementation

Nachdem in Kapitel 3 die funktionalen und nichtfunktionalen Anforderungen an das neue Storefrontsystem erläutert wurden, widmet sich dieses Kapitel dem im Rahmen dieser Bachelorarbeit entwickelten Prototypen. Es werden sowohl die getroffenen Architekturentscheidungen begründet als auch die zentralen Konzepte der Umsetzung detailliert dokumentiert. Abschliessend wird in Abschnitt 5.9 ein möglicher Migrationspfad aufgezeigt, der es Kunden von 'AbaShop' ermöglicht, auf das neue Storefrontsystem zu migrieren.

Abbildung 5.1 zeigt anhand eines C4-Kontextdiagramms [4], welche Akteure und Softwaresysteme in das Gesamtsystem involviert sind.

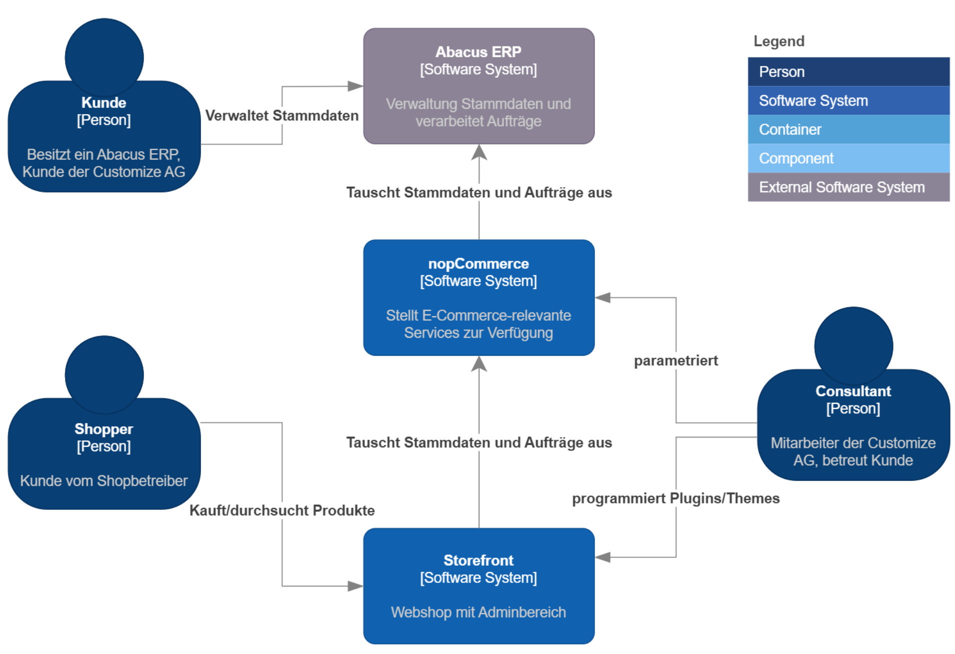


Abbildung 5.1: Kontextdiagramm Gesamtsystem (C4)

Um die Kommunikation zwischen dem Storefrontsystem und nopCommerce zu ermöglichen, wird eine Schnittstelle benötigt. Der nachfolgende Abschnitt dokumentiert die Evaluation der Schnittstellentechnologie, die anschliessend als Plugin für nopCommerce implementiert wurde.

5.1 Schnittstelle nopCommerce

In der vorangegangenen Studienarbeit wurde das E-Commerce-System nopCommerce mittels eines Connectors mit dem Abacus ERP zwecks automatisierten Datenaustausch verbunden. Das Storefrontsystem muss an nopCommerce angebunden werden, um die E-Commerce-relevanten Stammdaten und Services zu beziehen. In Abschnitt 3.7 wird die Anforderung beschrieben, dass nopCommerce 'headless' zur Verfügung gestellt werden soll. Auf dem Marktplatz von nopCommerce gibt es ein Plugin, das eine Schnittstelle für nopCommerce zur Verfügung stellt und für 990 Dollar pro Instanz erworben werden kann [34]. Diese Variante wird jedoch nicht weiterverfolgt, wofür zwei Hauptgründe identifiziert wurden:

- **Zukunftssicherheit:** NFA2 beschreibt, dass das System under Development (SuD) eine Lebensdauer von mindestens sechs Jahren haben soll. Mit einer Abhängigkeit von einem externen Plugin, dessen Wartung jederzeit eingestellt werden kann, ist die Erfüllung dieses NFA nicht garantiert.
- **Funktionalität:** Ein eingekauftes Plugin ist nicht auf die konkreten Bedürfnisse des Industriepartners ausgerichtet. Ein eigener Aufbau der Schnittstelle ermöglicht es, diese gemäss den Anforderungen zu entwickeln und beispielsweise weitere Funktionalitäten des Abacus Connectors aus der Studienarbeit zur Verfügung zu stellen.

5.1.1 Vorgehen

In einem ersten Schritt wurden verschiedene Möglichkeiten der Anbindung des Storefrontsystems evaluiert. Auf Basis der evaluierten Optionen, gRPC und RESTful HTTP, wurde für jede Schnittstelle eine prototypische Anbindung des Storefrontsystems an nopCommerce realisiert. In diesem Prototyp wurde nopCommerce über den Abacus Connector an einen Abacus-Mustermandanten angebunden. Basierend darauf wurde eine Schnittstelle für die Abfrage aller Produkte und Bilder von diesem Mandanten aus nopCommerce implementiert und ein Performancevergleich durchgeführt. Die abgefragten Produkte wurden in einem React-Frontend dargestellt, um mit einer prototypischen End-to-End-Anbindung eine optimale Entscheidungsgrundlage zu schaffen.

5.1.2 Remote Procedure Call gRPC

Remote Procedure Call (gRPC) ist ein Kommunikationsprotokoll, das von Google entwickelt wurde, um Geschwindigkeit und plattformübergreifende Sprachunterstützung in der Entwicklung von Schnittstellen zu verbessern. Dieses Protokoll wird häufig in der Server-zu-Server-Kommunikation eingesetzt, beispielsweise zwischen verschiedenen Microservices[1]. Ein Schlüsselelement, das gRPC besonders leistungsfähig macht, ist die binäre Datenübertragung mittels Protobuf (Protocol Buffers) zur Datenformatierung. Diese binäre Serialisierung führt zu einer signifikanten Reduzierung der Bandbreitennutzung im Vergleich zu textbasierten Formaten wie JavaScript Object Notation (JSON). Protobuf trägt zudem zur

[1] Ein Microservice ist eine unabhängige, klein gehaltene Komponente einer Softwareanwendung, die spezifische Aufgaben ausführt und typischerweise über gut definierte Schnittstellen mit anderen Microservices kommuniziert.

Entwicklungseffizienz bei, da diese Schemadateien sprachunabhängig zwischen Server und Client geteilt werden können [23].

Abbildung 5.2 zeigt schematisch die Funktionsweise von gRPC auf.

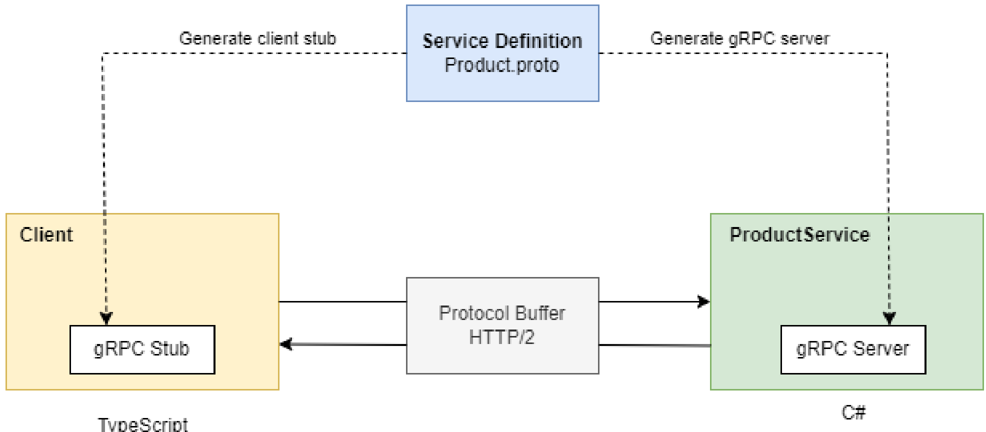


Abbildung 5.2: Funktionsweise von gRPC

Die *.proto*-Datei kann auf dem C#-Server und Typescript-Client verteilt werden, womit der Client die Methode *GetHomePageProducts* auf dem Server ausführen kann und das entsprechende Resultat als Typescript-Objekt zurückerhält. In Listing 5.1 ist eine gekürzte Protobuf-Implementation aus dem Prototypen aufgeführt.

```
syntax = "proto3";
option csharp_namespace = "grpcService";

service ProductService {
  rpc GetHomePageProducts(Empty) returns (HomePageProductResponse);
}

message Empty {}
message ProductResponse {
  repeated Product products = 1;
}
message Product {
  int64 Id = 1;
  string Name = 2;
  string ShortDescription = 3;
  double Price = 4;
  repeated string Images = 5;
}
```

Listing 5.1: Protobuf Implementation Prototyp

Die gRPC-Schnittstelle zwischen den React-Client und nopCommerce konnte erfolgreich implementiert werden. Eine Herausforderung besteht darin, dass nopCommerce im *PictureService* den *HttpContextAccessor* verwendet um den Unified Ressource Locator (URL) der Bilder zu bauen. Dieser Kontext steht aber durch die Anwendung von gRPC nicht standardmässig zur Verfügung. Microsoft schreibt dazu in der Dokumentation [30]:
'*ServerCallContext doesn't provide full access to HttpContext in all ASP.NET APIs.*'

Es steht eine Erweiterungsmethode *GetHttpContext* zur Verfügung, mit welcher mittels Anpassung des *PictureService* diese Problematik gelöst werden kann. Dies benötigt jedoch Anpassungen im Core-Code von nopCommerce, was gemäss NFA7 der vorangegangenen Studienarbeit [8] verhindert werden soll.

5.1.3 RESTful HTTP-Schnittstelle

RESTful HTTP-Schnittstellen sind inspiriert von den Prinzipien, die Fielding in seinem Paper 'Representational State Transfer (REST)' beschreibt [48]. Diese Prinzipien definieren eine einfache und flexible Methode zur Kommunikation zwischen Client und Server im Internet, welche die grundlegenden Operationen des HTTP-Protokolls nutzt. Moderne Implementierungen von RESTful-Schnittstellen verwenden häufig JSON für den Datenaustausch, obwohl REST formatunabhängig konzipiert ist und daher auch andere Datenformate wie die Extensible Markup Language (XML) unterstützen kann.

Dank der weiten Verbreitung von HTTP-Schnittstellen konnte der Prototyp ebenfalls problemlos implementiert werden. Ein Nachteil im Vergleich zu gRPC besteht darin, dass die Datentransfer-Modelle aufgrund der Kommunikation über JSON nicht zwischen Server und Client geteilt werden und daher doppelt in beiden Programmiersprachen geführt werden müssen. Durch die Implementierung der Schnittstelle gemäss der OpenApi Specification (OAS)[2] kann jedoch aus der RESTful HTTP-Schnittstelle ein Schema generiert werden [47]. Mithilfe dieses Schemas kann die Client-Implementation automatisiert mit einem Tool wie NSwag[3] erstellt werden.

[2]Die OpenAPI Specification (OAS) ist ein Industriestandard zur Beschreibung von RESTful APIs, der es ermöglicht, die Struktur einer API inklusive verfügbarer Endpunkte, Operationen und Parameter sowie die dazugehörigen Datenmodelle in einem standardisierten Format zu spezifizieren.

[3]NSwag ist ein Werkzeug zur Codegenerierung für das .NET-Framework, das die Erstellung von API-Client-Bibliotheken, Server-Stubs und API-Dokumentationen aus einer Swagger/OpenAPI-Spezifikation (OAS) automatisiert [18].

5.1.4 Entscheidung

Die prototypische Implementierung hat gezeigt, dass mit beiden evaluierten Schnittstellentechnologien eine End-zu-End-Anbindung zwischen nopCommerce und dem Storefrontsystem erreicht werden kann.
Mittels einer Performanceanalyse wurden 50 Aufrufe mit beiden Schnittstellen durchgeführt, um diese miteinander zu vergleichen. Abbildung 5.3 zeigt einen Boxplot dieses Performancevergleichs, der die Verteilung der Messwerte inklusive Median, Quartilen und Ausreissern visualisiert [49]. Die gemessene Zeit bezieht sich dabei auf den End-zu-End-Zeitraum, was den Zeitabstand vom Start des Aufrufs bis zur Anzeige aller Artikel mit Bildern im Browser bedeutet.

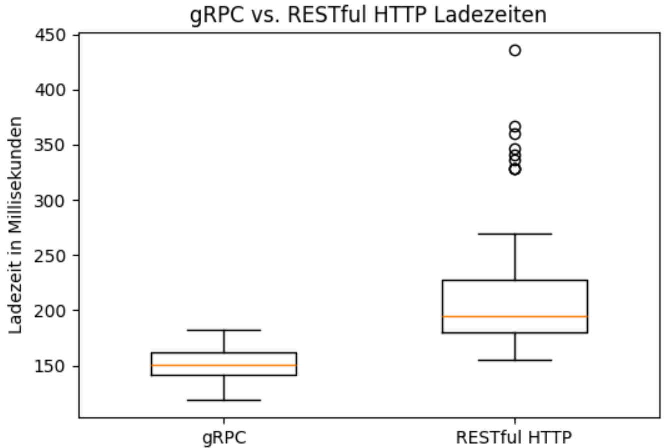


Abbildung 5.3: Boxplot Performance gRPC vs. RESTful HTTP

Wie erwartet ist gRPC performanter, wobei die Performance-Schwankungen der HTTP-Schnittstelle besonders auffällig sind. Die in Abschnitt 5.1.2 diskutierten Anpassungen am *PictureService* verstossen jedoch gegen die nichtfunktionale Anforderung 7 (*Plugin-System*) der Studienarbeit [8], welche Anpassungen am Source-Code des E-Commerce-Systems verhindert, um dessen Updatefähigkeit zu gewährleisten.

Das Buch *Designing Data-Intensive Applications* hebt in der Frage gRPC vs. RESTful HTTP hervor, dass RESTful APIs über JSON einfacher zu experimentieren und zu debuggen sind, was sie für öffentlich zugängliche Schnittstellen prädestiniert. Die Nutzung eines Browser oder Tools wie cURL[4] ermöglicht einfache Tests dieser Schnittstellen, eine Möglichkeit, die bei gRPC nicht gegeben ist. JSON-basierte RESTful APIs sind zudem weit verbreitet und werden von den meisten Programmiersprachen und Plattformen unterstützt. Aus diesem Grund liegt der Hauptanwendungsbereich von gRPC in der internen Kommunikation zwischen Services innerhalb desselben Unternehmens [26].

[4] cURL ist ein vielseitiges Kommandozeilen-Tool und eine Bibliothek, die das Übertragen von Daten mit verschiedenen Protokollen wie HTTP, FTP und SMTP unterstützt und häufig für das Testen und Interagieren mit Web-APIs verwendet wird.

In der Anforderungsanalyse in Abschnitt 3.7 wird erläutert, dass nopCommerce 'headless' bereitgestellt werden soll, um nicht nur das Storefrontsystem, sondern auch potentiell zukünftige Systeme zu unterstützen. Die Schnittstelle wird daher als öffentlich zugängliche API betrachtet, auch wenn sie im Kontext des Storefrontsystems nicht öffentlich sein muss. Da gRPC nur von Server-Applikationen genutzt wird und keine direkte Unterstützung für Browser-Applikationen bietet, sowie die weitreichende Verbreitung von HTTP-Schnittstellen eine breitere Integration mit zukünftigen Applikationen ermöglicht, fiel die Entscheidung auf eine RESTful HTTP-Schnittstelle. Trotz geringerer Performance im Vergleich zu gRPC werden die Vorteile von einer RESTful Schnittstelle, insbesondere die einfache Handhabung und breite Unterstützung, als ausschlaggebend betrachtet. Die Implementierung dieser Schnittstelle als Plugin für nopCommerce erfolgt nach dem OpenAPI-Standard (OAS), um eine zukünftige automatische Generierung von Clients zu ermöglichen.

Die Entscheidung lässt sich gemäss der Struktur des Y-Template [58] von Prof. Olaf Zimmermann folgendermassen zusammenfassen:
'Im Kontext der Integration von nopCommerce in das Storefrontsystem, mit der Anforderung die Services von nopCommerce auch potentiell anderen System 'headless' zur Verfügung zu stellen, fiel die Entscheidung auf die Implementation einer RESTful HTTP-Schnittstelle und gegen einen gRPC Service, da durch die grössere Verbreitung von HTTP-Schnittstellen potentiell mehr Systeme die Services von nopCommerce nutzen können und Anpassungen am Source-Code von nopCommerce verhindert werden können, wobei als Konsequenz die Performanceeinbussen im Vergleich zu gRPC akzeptiert wird.'

5.2 Erweiterungen nopCommerce

In der vorangegangenen Studienarbeit wurde das E-Commerce-System nopCommerce evaluiert, für das ein Plugin (Abacus Connector) entwickelt wurde. Um das entwickelte Storefrontsystem in die zentralen E-Commerce-Services von nopCommerce zu integrieren, entstand in dieser Bachelorarbeit ein weiteres Plugin (Webapi). Dieses stellt die benötigten Services von nopCommerce gemäss Designentscheid im vorangegangen Abschnitt 'headless' über eine RESTful HTTP-Schnittstelle zur Verfügung. Dieser Abschnitt beschreibt Anpassungen am Abacus Connector der Vorarbeit, die zur Erfüllung der funktionalen und nichtfunktionalen Anforderungen dieser Arbeit notwendig waren, sowie wichtige Konzepte aus dem neuen Webapi-Plugin.

5.2.1 Einführung

Der Abacus Connector gewährleistet den bidirektionalen Stammdatenaustausch mit dem Abacus ERP, indem alle shoprelevanten Stammdaten in nopCommerce persistiert und neue Shopper sowie Verkaufsaufträge ins Abacus zurückgeschrieben werden. In Abbildung 5.4 ist das C4-Containerdiagramm des Abacus Connectors dargestellt, das aus der vorangegangenen Studienarbeit [8] resultiert.

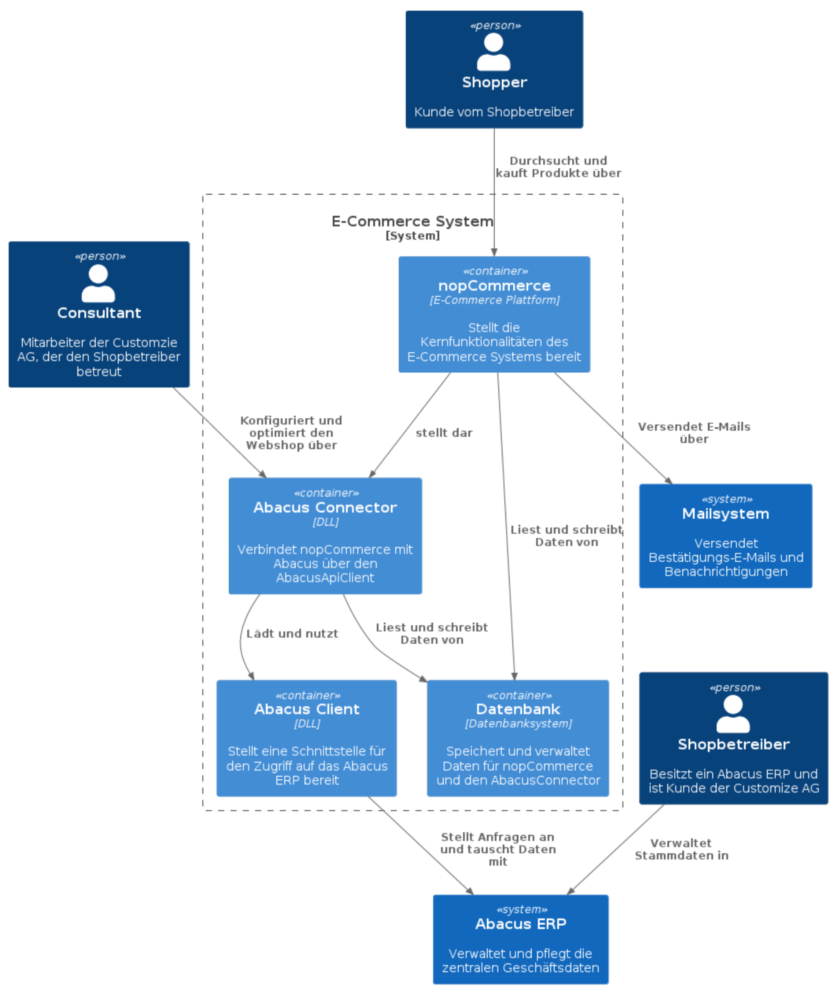


Abbildung 5.4: Containerdiagramm Abacus Connector (C4) [8]

5.2.2 Übersicht Container

Das in Abbildung 5.5 gezeigte C4-Containerdiagramm illustriert die Integration des neuen Plugins 'Webapi' in nopCommerce. Das Storefrontsystem wird dabei als externes System abstrahiert, da das Ziel der Webapi darin besteht, nopCommerce mithilfe des Abacus Connectors 'headless' bereitzustellen. Zukünftig könnte das Webapi-Plugin auch zur Anbindung anderer Systeme an nopCommerce genutzt werden.

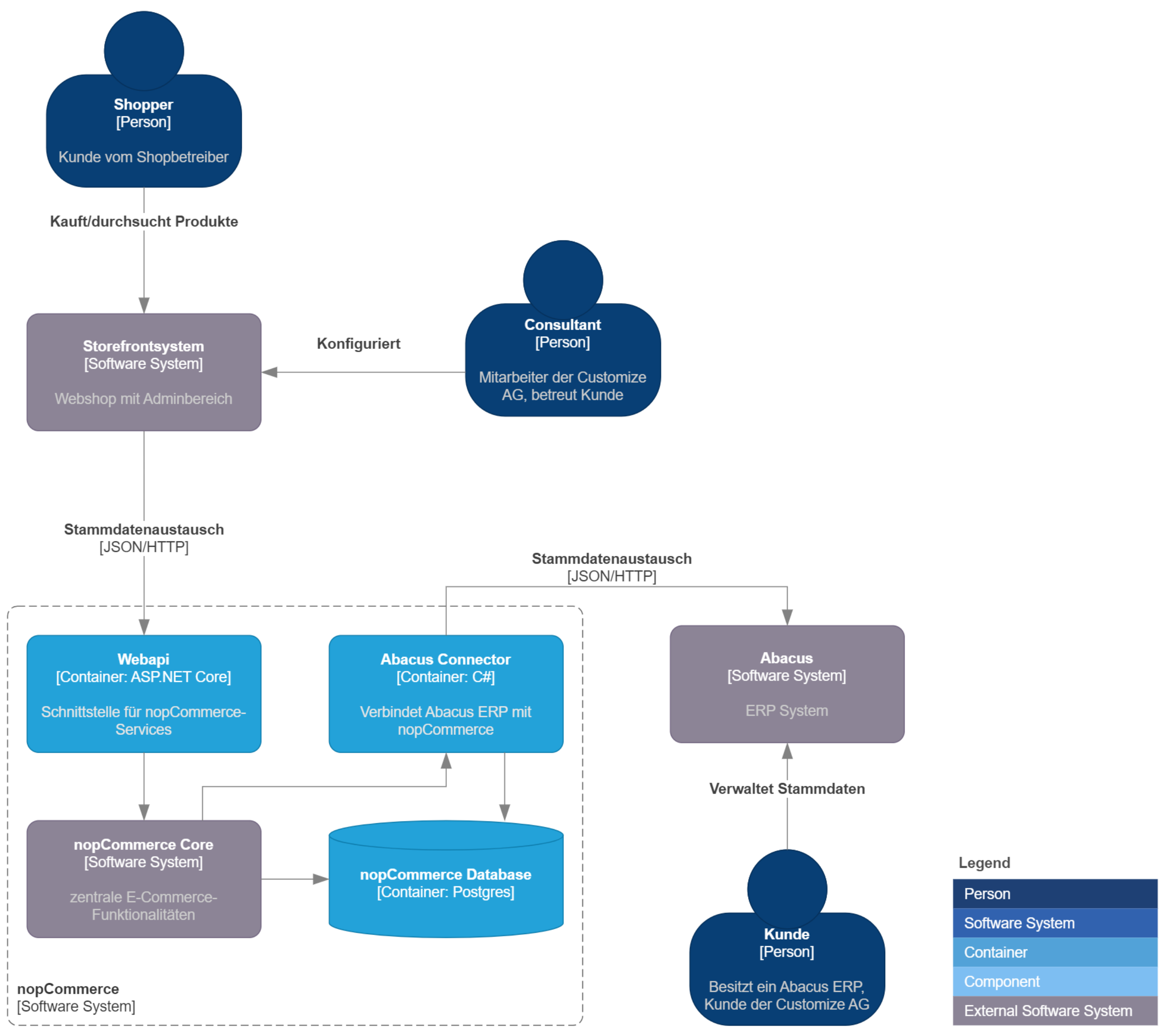


Abbildung 5.5: Containerdiagramm nopCommerce (C4)

In den nachfolgenden Abschnitten wird aufgezeigt, wie der Container *Webapi* implementiert ist und welche Anpassungen am Container *Abacus Connector* im Vergleich zur vorangegangenen Studienarbeit vorgenommen wurden.

5.2.3 Webapi Plugin

In der Designentscheidung in Abschnitt 5.1 wurde erläutert, warum die Schnittstelle für nopCommerce als RESTful HTTP-Schnittstelle implementiert wurde. Analog zum Abacus Connector kann die Webapi als Plugin in nopCommerce installiert und in Betrieb genommen werden. Die Schnittstelle ist nach dem OpenAPI-Standard konzipiert, was eine automatische Generierung der Schnittstellendokumentation ermöglicht. In Abbildung 5.6 sind alle implementierten Endpunkte dargestellt.

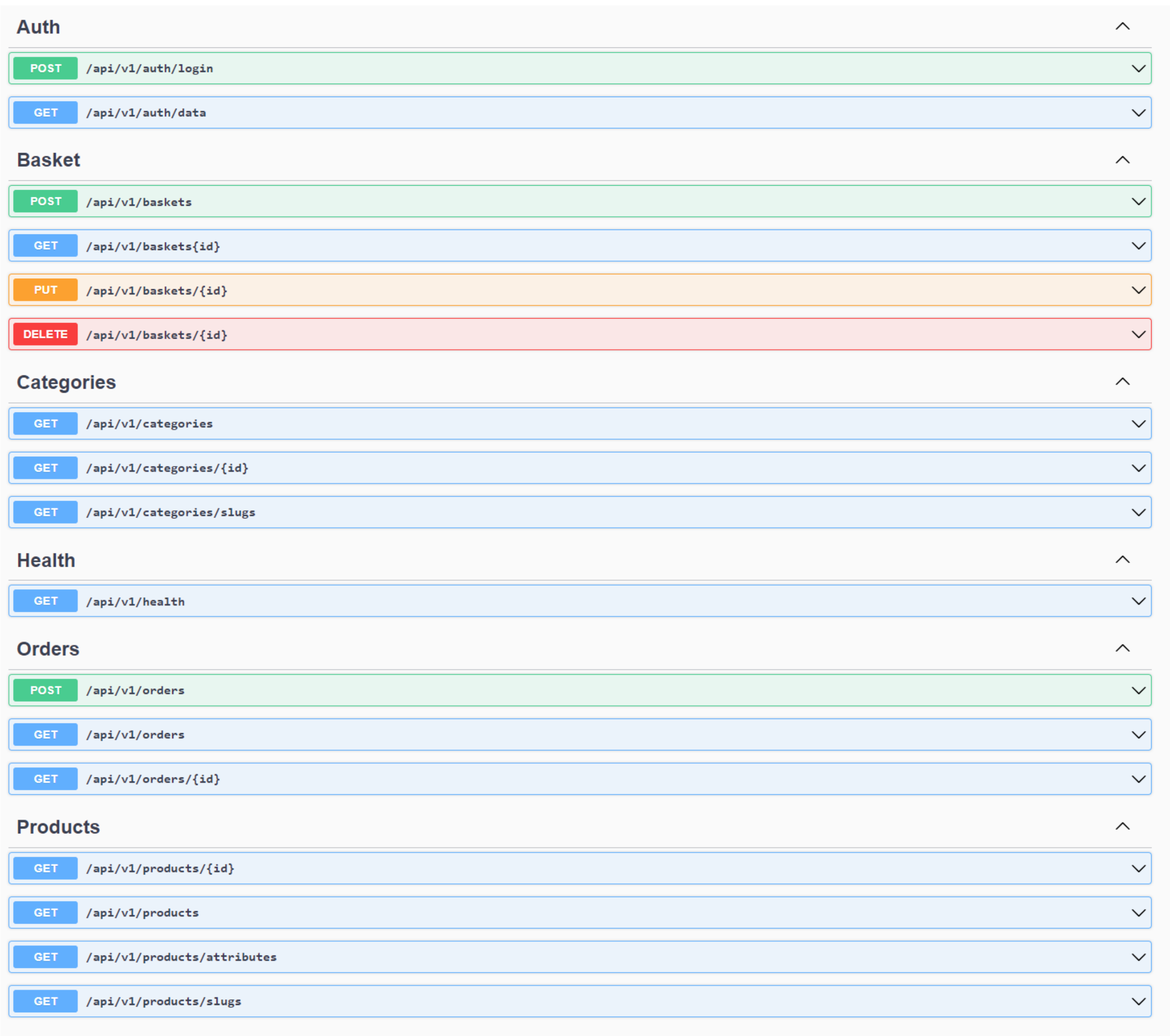


Abbildung 5.6: Dokumentation Webapi Plugin (Openapi)

Authentifizierung

Einige Endpoints sind nur für registrierte Shopper zugänglich und für Gastbenutzer gesperrt. Zur Absicherung bietet der Endpoint */login* die Möglichkeit, sich mit E-Mail und Passwort anzumelden. Das Webapi-Plugin generiert einen JWT-Token (JSON Web Token), der E-Mailadresse und Kundennummer als Claim[5] umfasst. Nach erfolgreicher Authentifizierung muss der Client den JWT-Token bei jeder Anfrage im HTTP-Header mitsenden. Das Webapi-Plugin prüft vor jedem Controller-Aufruf mithilfe einer Middleware den Header. Bei erfolgreicher Verifizierung des Claims fügt die Middleware die Kundennummer in den Kontext des aktuellen Aufrufs ein, der dann für alle Services verfügbar ist. Listing 5.2 zeigt ein Beispiel für eine erfolgreiche Antwort des Login-Endpoints.

```
{
  "Email": "tester@customize.ch",
  "CustomerId": 99,
  "CustomerGuid": "81329716-0de0-4800-9378-6f552445fcd9",
  "Token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "IsAdmin": true
}
```

Listing 5.2: Antwort Aufruf *login* im Webapi Plugin

Es ist ersichtlich, dass in der Antwort eine Variable *isAdmin* zurückgegeben wird. Da nopCommerce ein Berechtigungskonzept umfasst, kann dieses auch für das Storefrontsystem genutzt werden. Die Variable steuert den Zugang zum Adminbereich.

Für die Generierung eines JWT-Tokens ist ein Secret-Key erforderlich, der zur Verschlüsselung und Signatur des Tokens dient, um die Authentizität und Integrität der darin enthaltenen Informationen zu gewährleisten. Dieser Schlüssel kann gemäss Abbildung 5.7 im Administrationsbereich konfiguriert und neu generiert werden. In diesem Fall werden alle bestehenden Token ungültig. Die generelle Lebensdauer eines Tokens lässt sich ebenfalls über eine Variable konfigurieren, wobei standardmässig eine Gültigkeit von 7 Tagen besteht.

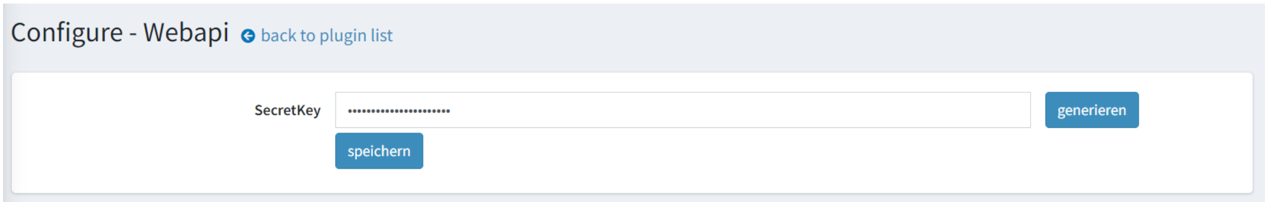


Abbildung 5.7: Konfiguration Secret-Key Webapi Plugin

[5] Ein Claim ist ein Informationsfragment, das für die Authentifizierung und Autorisierung notwendige Benutzer- oder Systemdaten enthält. Claims verifizieren im Rahmen der Authentifizierung die Identität eines Nutzers und ermöglichen die sichere Überprüfung dieser Identität in späteren Serverkommunikationen, ohne erneute Authentifizierung.

API-Versionierung

Das Webapi-Plugin ist bei mehreren Kunden parallel im Einsatz, weshalb eine Versionierung der implementierten Schnittstelle essentiell ist. Dies ermöglicht es jedem Kunden, selbst zu entscheiden, welche Version der Schnittstelle genutzt wird und erlaubt somit die Auslieferung von Updates mit Breaking Changes[6]. Im Buch *Patterns for API Design* [59] werden im Pattern *Version Identifier* mehrere Möglichkeiten zur Implementierung einer Versionierung beschrieben:

- Als HTTP Type Negotiation Header: `Accept: text/json; version=2.0`
- Als Teil des Ressourcen-Identifikators: `GET v2/products/1234`
- Als Domain-Name: `https.v2.api.storefront.com`
- Im JSON-Body: `"version" : "2.0"`

Eine Implementierung über den Domain-Namen wurde aufgrund der dadurch entstehenden Komplexität in Deployment und Wartung der Infrastruktur ausgeschlossen. Alle anderen Methoden sind praktikabel und grundsätzlich mit der vorhandenen Ausgangslage umsetzbar. Die Entscheidung fiel auf die Implementierung mittels Ressourcen-Identifikator, da dadurch die Endpunkte aussagekräftig sind und dieses Verfahren in ASP.NET Core nativ unterstützt wird. Ein weiterer Vorteil dieser Methode ist die einfache Integration in die OpenAPI-Dokumentation gemäss Abbildung 5.8.

Abbildung 5.8: API-Version Auswahl in Swagger

Eine Herausforderung bei der Versionierung von Schnittstellen ist die Definition, welcher Code für die Erstellung einer neuen Version dupliziert wird. Es wurde festgelegt, dass lediglich die Controller, Data Transfer Objects (DTOs) und die DTO-Mapper dupliziert werden. Das Kopieren von Services sollte möglichst vermieden werden, da diese Use Cases abdecken und bei einer neuen Version in der Regel neue Services hinzugefügt werden, statt bestehende Services zu ändern. In den drei definierten Komponenten wird pro Version ein Subverzeichnis `v{x}` erstellt, in dem auch zur vorherigen Version identische Klassen dupliziert werden. Im Storefrontsystem kann im Root-Verzeichnis in der Datei *.env* gemäss Listing 5.3 über eine Umgehungsvariable die gewünschte Version der Schnittstelle gewählt werden.

```
1 NOPCOMMERCE_BASE_URL="http://localhost:5000"
2 NOPCOMMERCE_API_VERSION="v1"
```

Listing 5.3: Umgebungsvariablen Storefrontsystem

[6]Breaking Changes sind Änderungen in einer API, die bestehende Funktionalitäten ändern oder entfernen und dadurch die Kompatibilität mit älteren Versionen beeinträchtigen.

5.2.4 Preis Cache

Die nichtfunktionale Anforderung *Verfügbarkeit* erfordert, dass das Storefrontsystem auch bei Nichterreichbarkeit des Abacus ERP funktionsfähig bleibt. Dies kann im Bereich der kundenindividuellen Preise mit der Implementierung der Studienarbeit nicht garantiert werden.

Das Abacus ERP bietet spezifische Rabatt- und Preiskonditionen für Kunden und Kundengruppen in Kombination mit Produkten und Produktgruppen. Eingeloggte Shopper im Storefrontsystem müssen auf diese Preise zugreifen können. Aufgrund der komplexen Preisgestaltung im Abacus und des hohen Fehlerrisikos sollte die Preisberechnung direkt im Abacus stattfinden, anstelle einer Neuimplementation in nopCommerce. Der Abacus Connector reagiert auf ein Event von nopCommerce, das bei jeder Änderung im Warenkorb ausgelöst wird. Der aktualisierte Warenkorb wird daraufhin an die RESTful HTTP-Schnittstelle von Abacus gesendet, welche die Konditionen berechnet und die Ergebnisse zurückschickt. Das Plugin aktualisiert dann den Warenkorb in nopCommerce. Diese Methode steht jedoch in Konflikt mit nachfolgenden nichtfunktionalen Anforderungen der Arbeit:

- **NFA1: Performance:** Um Shoppern personalisierte Preise direkt in der Produktübersicht anzeigen zu können, müsste der Abacus Connector in nopCommerce für jedes Produkt zusätzlich die Abacus-Schnittstelle abfragen. Die Ladezeiten der komplexen Preiskalkulation im Abacus ERP belastet die Performance erheblich und beeinträchtigt die Ladezeiten des Storefrontsystems. Ein Performancevergleich mit und ohne diese zusätzlichen Preisabfragen wird in Tabelle 5.1 dargestellt.
- **NFA5: Verfügbarkeit:** Wenn das Abacus ERP nicht verfügbar ist, können keine Schnittstellenaufrufe erfolgen, folglich sind individuelle Preise für Shopper nicht abrufbar.

Aufruf	ohne Preisfindung (ms)	mit Preisfindung (ms)
1	829.2	2074.4
2	802.9	2157.6
3	806.8	2101.4
4	842.6	2072.4
5	757.1	2011.8
6	758.7	2122.9
7	752.0	2018.8
8	833.3	2245.9
9	827.8	2281.8
10	837.0	1991.7
Durchschnitt	804.7	2107.9

Tabelle 5.1: Antwortzeiten Webapi Endpoint */products*

Zur Bewältigung dieser Herausforderung wurde eine Erweiterung des Abacus Connectors implementiert. Mittels eines 'Preis Caches' persistiert das Plugin die kundenindividuellen Preise in einer eigenen Tabelle in der Datenbank des Abacus Connector Plugins. Der Synchronisationsdienst, der in einem konfigurierten Intervall die Stammdaten zwischen nopCommerce und Abacus synchronisiert, übermittelt für jeden bekannten Kunden einen Warenkorb mit allen Artikeln an Abacus, um anschliessend die spezifischen Preise in der Datenbank zu persistieren. Die Domain-Entität vom 'Preis-Cache' ist in Listing 5.4 aufgeführt.

```
public class PriceCache : BaseEntity
{
    public int CustomerId { get; set; }
    public int ProductId { get; set; }
    public DateTime LastUpdate { get; set; }

    private double PiceWithTax { get; set; };
    private double PriceWithOutTax { get; set; };
}
```

Listing 5.4: Domain-Entität 'Preis-Cache' im Abacus Connector

Damit auch nicht eingeloggte Kunden von Aktionen profitieren können, existiert im 'Preis-Cache' ein Kunde 0, der die kundenunabhängigen Preise führt. Um die Menge der Daten einzuschränken, werden lediglich Preise geführt, die vom Listenpreis abweichen. Es ist zu beachten, dass der 'Preis-Cache' nicht aktualisiert werden kann, wenn Abacus nicht erreichbar ist. Aus diesem Grund gibt es eine zusätzliche Konfiguration, die jeder Kunde vornehmen kann. Diese legt fest, wie lange ein Preis aus dem 'Preis-Cache' verwendet wird, bevor wieder auf den Listenpreis zurückgegriffen wird. Abbildung 5.9 zeigt die zusätzlichen Konfigurationen, die in der Bachelorarbeit beim Abacus Connector eingeführt wurden.

Abbildung 5.9: Konfiguration Abacus Connector

Die nachfolgende Abbildung 5.10 zeigt in einem UML-Sequenzdiagramm auf, wie beim Aufruf vom Endpoint */products* im Webapi-Plugin der 'Preis-Cache' genutzt wird, um die kundenspezifischen Preise in die Antwort zu integrieren.

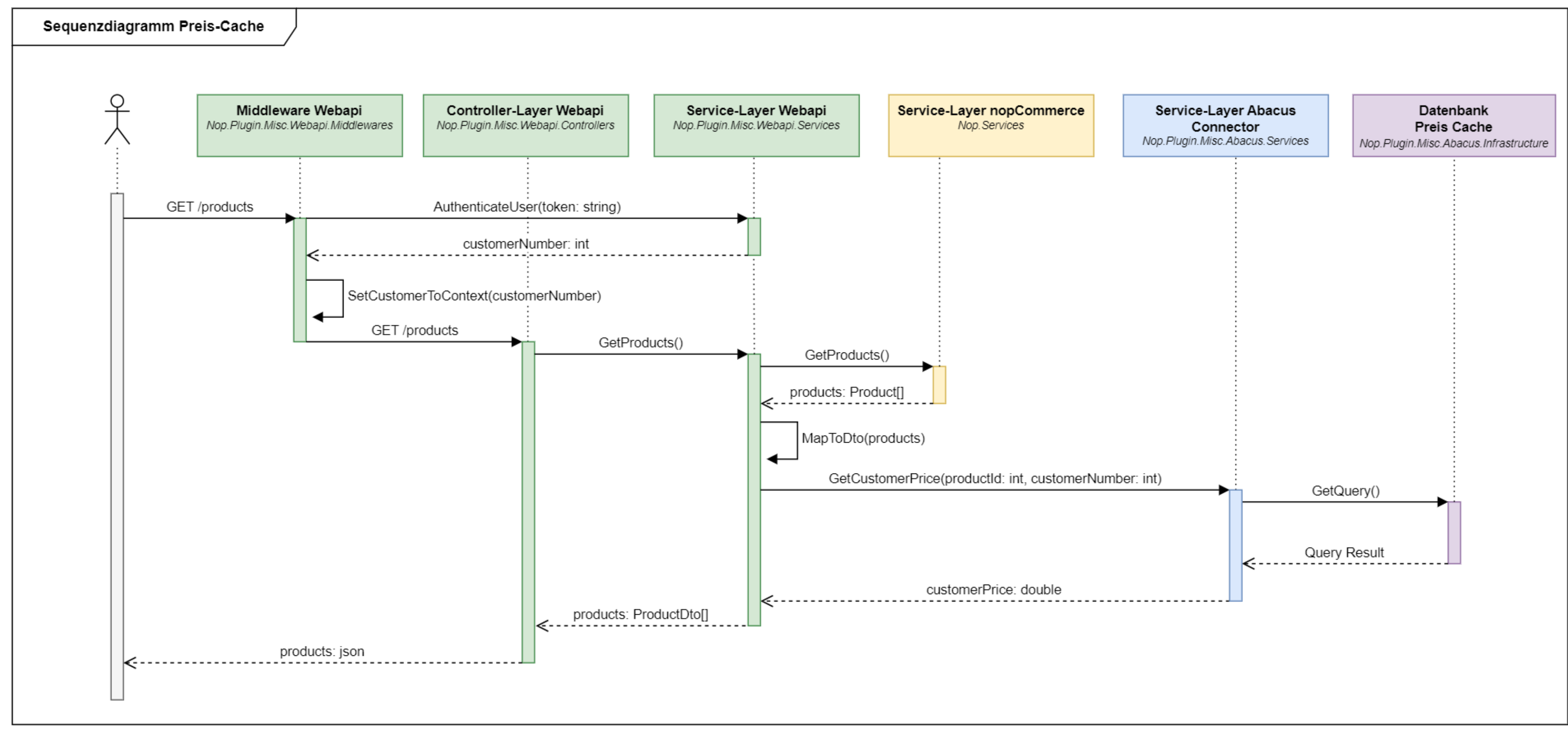


Abbildung 5.10: Ablauf 'Preis-Cache' (UML-Sequenzdiagramm)

Der Client der Webapi kann selbst entscheiden, ob er den 'Preis-Cache' in die Antwort integrieren möchte, indem *customerprice=true* als Query-Parameter[7] mitgegeben wird.

Tabelle 5.2 zeigt auf, wie stark sich die Performance mit der Implementation vom 'Preis-Cache' verglichen zur Implementation mit einem zusätzlichen Aufruf der Abacus Schnittstelle verbessert.

Aufruf	ohne Preisfindung (ms)	mit Preisfindung (ms)	mit 'Preis-Cache' (ms)
1	829.2	2074.4	1039.0
2	802.9	2157.6	1010.9
3	806.8	2101.4	1011.4
4	842.6	2072.4	1050.4
5	757.1	2011.8	958.3
6	758.7	2122.9	965.1
7	752.0	2018.8	953.5
8	833.3	2245.9	1042.7
9	827.8	2281.8	1033.0
10	837.0	1991.7	1041.1
Durchschnitt	804.7	2107.9	1010.5

Tabelle 5.2: Antwortzeiten Webapi Endpoint */products* mit 'Preis-Cache'

Es sollte beachtet werden, dass der 'Preis-Cache' in der aktuellen Implementation lediglich die Kundenkonditionen für die Menge eins persistiert. Dies bedeutet, dass keine an Staffel-Konditionen[8] gebundene Preise vom 'Preis-Cache' ausgegeben werden. Aus diesem Grund wird gemäss der Implementierung in der vorangegangenen Studienarbeit [8] der Warenkorb bei jeder Änderung erneut an Abacus übermittelt, sofern dieses zur Verfügung steht. Dies steht im Einklang mit der NFA *Verfügbarkeit*, da das Gesamtsystem auch dann funktionsfähig bleibt, wenn Abacus nicht erreichbar ist. In solchen Fällen werden jedoch mögliche kundenspezifische Konditionen, die mengenabhängig sind, nicht dargestellt, da in diesem Fall die Kundenkondition gemäss 'Preis-Cache' für die Menge eins gewährt wird.

[7] Query-Parameter sind Schlüssel-Wert-Paare, die an die URL einer HTTP-Anfrage angehängt werden, um spezifische Daten oder Anweisungen an den Server zu übermitteln, wodurch der Server gezielte Informationen zurücksenden oder bestimmte Operationen basierend auf den angegebenen Parametern durchführen kann.

[8] Staffel-Konditionen im Abacus beziehen sich auf Preisnachlässe oder Konditionen, die je nach der gekauften Menge eines Artikels variieren.

5.2.5 Architektur nopCommerce

In Abbildung 5.11 ist das C4-Komponentendiagramm [4] der beiden entwickelten Plugins, Abacus Connector und Webapi, für nopCommerce dargestellt. Der Kerncode von nopCommerce wird als externes System visualisiert, um zu verdeutlichen, dass bei der Entwicklung darauf geachtet wurde, den Source-Code von nopCommerce nicht zu modifizieren. Dies gewährleistet die Kompatibilität mit zukünftigen Updates. Beide Plugins interagieren ausschliesslich mit dem Service-Layer von nopCommerce.

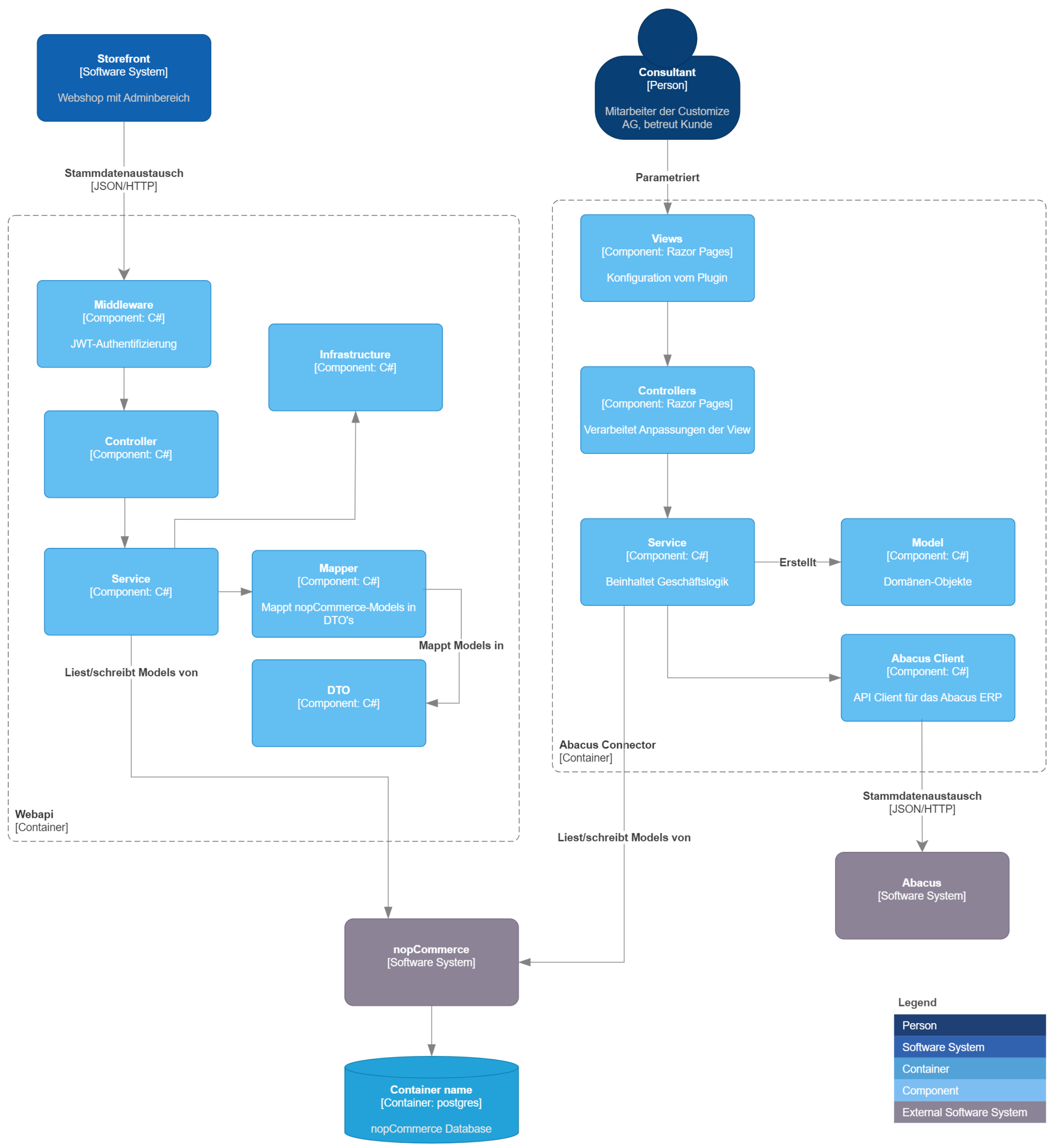


Abbildung 5.11: Architektur: Komponentendiagramm nopCommerce (C4)

5.3 Auswahl Framework

Das im Kontextdiagramm (Abbildung 5.1) des Gesamtsystems dargestellte Storefrontsystem wird in dieser Arbeit als Prototyp entwickelt. Die Aufgabenstellung (siehe Anhang C) gibt vor, dass das Storefrontsystem mit einem modernen JavaScript-Framework umgesetzt werden soll. In diesem Abschnitt wird beschrieben, welche Frameworks evaluiert wurden und mit welchem schliesslich gearbeitet wurde.

5.3.1 Vorgehen

Im JavaScript-Ökosystem existieren viele Frameworks zur Entwicklung von interaktiven Benutzeroberflächen, wobei regelmässig neue hinzukommen. Für die Auswahl potenzieller Frameworks werden lediglich die am weitesten verbreiteten in Betracht gezogen. Als Indikator dient hierbei die Anzahl der wöchentlichen Downloads auf dem JavaScript-Paketmanager npm [37]. Andere Frameworks wie Svelte, Qwik oder Remix werden nicht berücksichtigt, da das Ziel darin besteht, eine möglichst grosse Anzahl von Entwicklern für die Entwicklung von Plugins und Themes zu erreichen und diese Frameworks noch eine geringe Verbreitung haben. Ebenfalls lassen sich diese unbekannteren Frameworks nicht mit dem NFA *Zukunftssicherheit* in Einklang bringen.

Die Abbildung 5.12 illustriert einen Vergleich der wöchentlichen Downloadzahlen der evaluierten Frameworks im Verlauf des letzten Jahres. Dabei ist anzumerken, dass Next.js ein React-Framework ist, folglich wird mit jedem Download von Next.js auch React heruntergeladen.

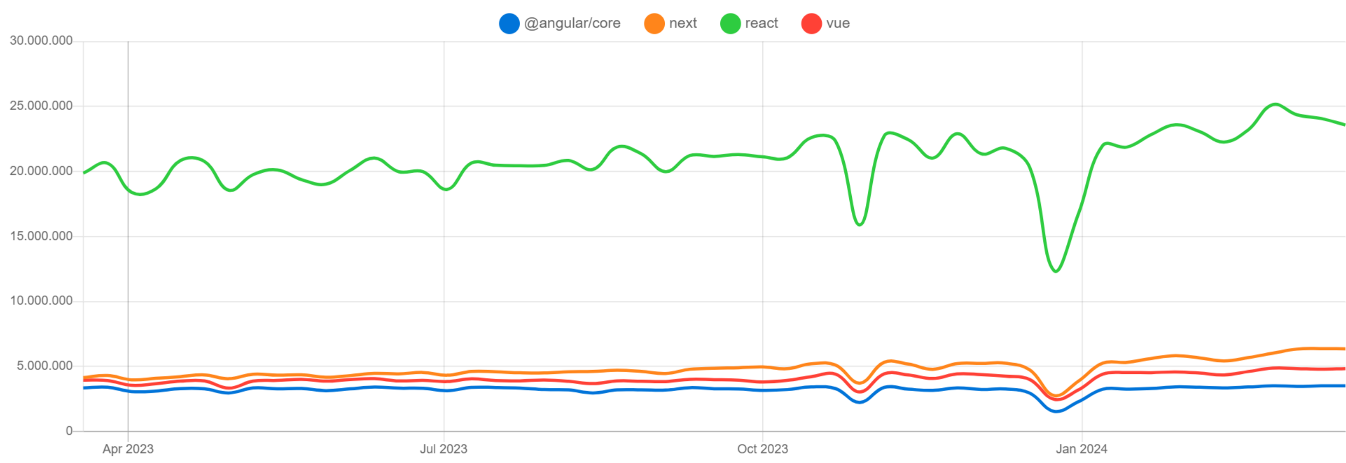


Abbildung 5.12: Wöchentliche Downloadzahlen der evaluierten JavaScript Frameworks (npm)

Um die Anforderungen der *NFA4 - Suchmaschinenoptimierung* zu erfüllen, ist die Verwendung von Server-Side Rendering (SSR)[9] im Storefrontsystem unerlässlich. Im Gegensatz zu traditionellen Single-Page-Applications (SPAs)[10], die nur ein HTML-Dokument als Einstiegspunkt ausliefern und das Rendering anschliessend im Browser mittels JavaScript durchführen, wird beim serverseitigen Rendering (SSR) für jeden Aufruf ein fertiges HTML-Dokument ausgeliefert. Die Verwendung von SSR kann sich positiv auf die Suchmaschinenoptimierung auswirken, da sie den Suchmaschinen eine schnellere und zuverlässigere Indexierung des Inhalts ermöglicht. Dies ist darauf zurückzuführen, dass die Fähigkeit der Suchmaschinen, JavaScript auszuführen, begrenzt oder langsamer ist. Obwohl moderne Suchmaschinen wie Google JavaScript ausführen und SPAs indexieren können, stellt das serverseitige Rendering sicher, dass der gesamte Inhaltsindex unmittelbar nach dem Laden der Seite verfügbar ist, was insbesondere bei zeitkritischen und leistungsorientierten SEO-Strategien hilfreich sein kann. Obwohl die heutigen JavaScript-Frameworks ursprünglich für traditionelle SPAs entwickelt wurden, bieten sie in den letzten Jahren zunehmend Möglichkeiten, auch für das serverseitige Generieren von Markup eingesetzt zu werden. Die folgenden Abschnitte stellen die evaluierten Frameworks vor und untersuchen, inwiefern sie SSR ermöglichen.

5.3.2 Angular

Angular, ein von Google entwickeltes Frontendframework für Webanwendungen, basiert auf TypeScript und ist heute als Open-Source-Software frei verfügbar. Es geniesst besondere Beliebtheit in grossen Projekten, da sein modulares Konzept das Gruppieren spezifischer Bereiche einer Applikation ermöglicht, die dann von einzelnen Teams unabhängig entwickelt werden können. Aufgrund dieser Flexibilität ist das Framework jedoch umfangreich und bringt einen entsprechend grösseren Overhead mit sich. Angular ist komponentenbasiert, wobei jede Komponente als wiederverwendbares Element fungiert, das sowohl das Verhalten als auch die Darstellung eines Teils der Benutzeroberfläche definiert. Eine Komponente besteht in der Regel aus einem HTML-Template, das Struktur und Layout festlegt, einer TypeScript-Klasse, die das Verhalten definiert, und einer CSS-Datei, die das Aussehen gestaltet. Durch das Two-Way-Binding zwischen HTML-Template und TypeScript-Klasse werden Änderungen in der Benutzeroberfläche automatisch im Datenmodell reflektiert und umgekehrt, was eine effiziente Interaktion zwischen Anwendung und Benutzer ermöglicht [2].

Mit der Einführung des *Angular Universal Package* in Angular 4 wurde Server-Side Rendering (SSR) unter Angular möglich. Nachdem diese Bibliothek stabilisiert wurde, integrierten die Entwickler das Universal-Paket direkt in den Angular Client. Es kann mit dem Befehl `ng add @angular/ssr` zu einem Projekt hinzugefügt werden. Durch die Konfiguration in der Datei `server.ts` erstellt Angular einen Node.js-Express-Server, der als Laufzeitumgebung für das serverseitige Rendering dient [6]. Obwohl Angular SSR unterstützt, liegt der Fokus weiterhin auf dem clientseitigen Rendering.

[9]Serverseitiges Rendering (SSR) ist die Technik, bei der Webseiteninhalte auf dem Server generiert und als vollständiges HTML-Dokument an den Browser gesendet werden, was die initiale Ladezeit erhöhen kann, aber die Suchmaschinenoptimierung verbessert, indem es den sofortigen Zugriff auf den Seiteninhalt ermöglicht.

[10]Eine Single Page Application (SPA) ist eine Webanwendung, die durch Laden einer einzelnen HTML-Seite funktioniert und dynamisch Inhalte aktualisiert, indem sie Daten mit dem Webserver austauscht, ohne die Seite neu zu laden.

5.3.3 React

React ist eine von Facebook entwickelte JavaScript-Bibliothek zur Erstellung von Benutzeroberflächen, die heute als Open-Source-Software frei verfügbar ist. React verwendet JSX, eine Syntaxerweiterung für JavaScript, die es Entwicklern ermöglicht, HTML-Strukturen innerhalb des JavaScript-Codes zu implementieren. Dies ermöglicht es, Rendering-Logik und Markup in einer Komponente zu vereinen [43]. Im Unterschied zu Angular verwendet React eine Einweg-Datenbindung. Dabei ist der Datenfluss stets von Eltern- zu Kindkomponenten gerichtet.
In der Praxis wird React oft mit darauf aufbauenden Frameworks verwendet. Bis 2023 galt das vom React-Team entwickelte Framework Create-React-App (CRA) als bevorzugte Wahl für viele Projekte. Mittlerweile wird CRA jedoch als veraltet betrachtet, und die offizielle Empfehlung lautet, auf extern entwickelte Frameworks umzusteigen [42]. Zum Zeitpunkt dieser Arbeit (2024) dominieren Vite und Next.js als Werkzeuge für die Arbeit mit React. Vite ist lediglich ein Build-Tool, das den Bau von React-Applikationen ermöglicht. Dabei wird eine Low-Level-API für serverseitiges Rendering (SSR) zur Verfügung gestellt, die allerdings primär für Framework-Entwickler konzipiert ist und nicht direkt für die Entwicklung spezifischer Anwendungen eingesetzt werden sollte [52]. Da dies nicht Teil dieser Arbeit ist, wird Vite als potentielles Tool für das Storefrontsystem nicht weiterverfolgt.

Next.js

Next.js ist ein leistungsfähiges React-Framework, das von Vercel entwickelt wurde, einer Firma, die Hosting-Lösungen anbietet, unter anderem spezialisiert auf Next.js-Applikationen. Es ist optimiert für serverseitiges Rendering (SSR), die Generierung statischer Seiten (SSG)[11] sowie die Entwicklung von Single Page Applications (SPA). Zu den Features zählen einfaches, dateibasiertes Routing, das es ermöglicht, anhand der Dateistruktur im Projektverzeichnis Routen zu definieren. Automatisches Code-Splitting verkürzt die Ladezeiten, indem nur der jeweils benötigte Code geladen wird. Integrierte API-Routen ermöglichen serverlose Funktionen[12]. Diese erlauben es, serverseitige Logik direkt innerhalb des Next.js-Projekts zu implementieren, ohne separate Server-Software bereitzustellen. Next.js verfolgt einen 'SSR-first'-Ansatz, bei dem standardmässig alle Seiten serverseitig gerendert werden. Um eine Komponente clientseitig zu rendern, muss dies explizit mit der `"use client"`-Direktive am Anfang der Komponente angegeben werden [31].

[11] Statisches Seitengenerieren (SSG) ist ein Ansatz in der Webentwicklung, bei dem Webseiten zur Build-Zeit vorab generiert und als fertige HTML-Dateien gespeichert werden, was schnelle Ladezeiten und verbesserte Sicherheit bietet, da dynamische Serverabfragen minimiert werden.

[12] In Next.js ermöglichen API-Routen die Erstellung von serverseitigen Funktionen und Endpunkten direkt innerhalb des Next.js-Projekts, indem Dateien im *pages/api* Verzeichnis platziert werden, die dann als serverlose Funktionen behandelt und ausgeführt werden.

5.3.4 Vue.js

Vue.js ist ein progressives JavaScript-Framework, das vorrangig für den Aufbau interaktiver Benutzeroberflächen und Single Page Applications (SPAs) von Evan You entwickelt wurde und als Open-Source Software frei zur Verfügung steht. Vue.js baut auf Standardtechnologien wie HTML, CSS und JavaScript auf, was den Einstieg besonders zugänglich macht. Trotz dieser Zugänglichkeit bietet Vue.js eine reaktive und modulare Architektur. Typischerweise besteht eine Komponente in Vue.js aus einem `<script>`-Tag, das die Logik in JavaScript enthält, einem `<template>`-Tag für das HTML-Markup und einem `<style>`-Tag für das CSS. Seit Version 3 unterstützt Vue.js das serverseitige Rendering, das durch den Import `import { createSSRApp } from 'vue'` in ein Projekt integriert werden kann. Dabei wird im Hintergrund ein Node.js-Express-Server genutzt, um eine Komponente auf dem Server zu rendern [53].

5.3.5 Entscheidung

Der 'SSR-first'-Ansatz von Next.js bietet für dieses Projekt einen grossen Vorteil, unterstützt durch die weite Verbreitung und die sehr aktive Community auf GitHub. Next.js bietet zudem viele nützliche Zusatzfeatures, wie beispielsweise API-Routen. Vite scheidet als Option aus, da es keine stabile Lösung für SSR bietet. Ein möglicher Nachteil von Next.js könnte der 'Vendor-Lock-in' sein, da der Hersteller des Frameworks auch Hosting-Lösungen anbietet. Obwohl eigenes Hosting grundsätzlich möglich ist, ist das gesamte Ökosystem auf das Deployment bei Vercel optimiert. Angular bietet ebenfalls ein umfassendes Ökosystem und unterstützt SSR innerhalb der Kernbibliothek. Allerdings bringt Angular für die Grösse dieses Projekts einen vergleichsweise grossen Overhead mit sich. Neben Next.js wird Vue.js als eine sehr gut geeignete Option für dieses Projekt angesehen.
Aufgrund der Erfahrungen der Mitarbeiter von Customize mit React und dem umfassenden Ökosystem von Next.js wurde entschieden, dass die Entwicklung mit dem React-Framework Next.js durchgeführt wird.

Die Entscheidung lässt sich gemäss der Struktur des Y-Template [58] folgendermassen zusammenfassen:
'Im Kontext der Auswahl eines JavaScript-Frameworks für die Entwicklung des Storefrontsystems, mit der Anforderung, serverseitiges Rendering zu unterstützen, fiel die Entscheidung auf React mit dem Framework Next.js und gegen Angular, Vue.js oder Vite, da dieses Framework einen 'SSR-first' Ansatz bietet und mit zusätzlichen Features wie API-Routen die Entwicklung des Storefrontsystems erleichtert und durch die grosse Verbreitung ein umfassendes Ökosystem zur Verfügung stellt, wobei als Konsequenz die mögliche Gefahr eines 'Vendor-Lock-ins' durch die Hosting-Plattform des Frameworkproduzenten Vercel akzeptiert wird.'

5.4 Dynamisches Laden und Teilen von Source-Code

Das angestrebte Theming- und Plugin-Konzept im Storefrontsystem erfordert die Fähigkeit des Kernsystems, dynamisch Source-Code zu laden und zwischen den Applikationen zu teilen. Dies ermöglicht es, im Storefrontsystem das zur Buildzeit noch unbekannte Theme und die aktivierten Plugins auszuführen und anzuzeigen.

Die Implementierung einer solchen Architektur in der Entwicklung von React- sowie generell JavaScript-Anwendungen stellt eine Herausforderung dar, da diese typischerweise 'gebündelt' werden. Das evaluierte Framework Next.js nutzt hierfür Webpack[13]. Zur Buildzeit kompiliert der Bundler die gesamte Anwendung und führt dabei dem Importpfad folgend alle Dateien in eine einzige Datei zusammen. Diese Datei wird als 'Bundle' bezeichnet und ermöglicht die Auslieferung einer gesamten Anwendung in nur einer einzelnen Datei. Eine Konsequenz davon ist, dass Webpack zur Buildzeit den gesamten Importpfad indexiert und Modulidentifikatoren vergibt, was das dynamische Laden von Modulen zur Laufzeit erschwert [22].

5.4.1 Vorgehen

Die in diesem Kontext getroffene Designentscheidung hat einen erheblichen Einfluss auf die Architektur des zukünftigen Systems. Deshalb wurden verschiedene Lösungsansätze prototypisch verifiziert. Der Prototyp soll einen kleinen Teil des vorgesehenen Theming-Systems abdecken. Hierbei wird versucht, die einführende User Story (*US13: Angezeigte Daten im Produktdetail steuern*) aus Abschnitt 3.5 minimal umzusetzen. In diesem Kontext bedeutet minimal, dass aufgrund einer Variable zwischen den beiden Komponenten gewechselt werden kann, ohne weitere Anpassungen am Source-Code vorzunehmen. Ziel ist es, in der Themekomponente *productDetail* jeweils unterschiedliche Detaildaten eines Produkts anzuzeigen. Die Steuerung soll dynamisch erfolgen, um festzulegen, welches Theme aktiv ist und welche Komponente entsprechend angezeigt wird.
Die in diesem Abschnitt evaluierten Konzepte lassen sich auch auf das Plugin-System übertragen. Auch hier wird ebenfalls die Möglichkeit benötigt, dynamisch Komponenten des Plugins anzuzeigen. Weiterhin soll es möglich sein, anstelle einer Komponente eine Plugin-Klasse zu laden, deren Logik anschliessend im Kernsystem ausgeführt wird.

Die nachfolgenden Optionen modellieren verschiedene Ansätze, die für diese Designentscheidung evaluiert wurden.

[13]Webpack ist ein populäres Modulbündelungs-Tool, das in modernen Webentwicklungsprojekten eingesetzt wird. Es ermöglicht Entwicklern, JavaScript-Dateien und andere Ressourcen wie CSS und Bilder zu bündeln, was die Ladezeiten von Webanwendungen erheblich verbessern kann.

5.4.2 Dynamic Import

Ein Nachteil des Bundlings besteht in der potenziell langen Dauer des initialen Aufrufs einer React-Applikation, da bei einer umfangreichen Applikation das Bundle entsprechend gross ist und im ersten Aufruf komplett geladen wird. Webpack hat zu diesem Zweck das Code Splitting[14] eingeführt, welches es ermöglicht, mehrere Bundles zu erstellen und diese zur Laufzeit dynamisch nachzuladen.
Next.js implementiert das Webpack-Konzept des Code Splitting unter dem Namen Lazy Loading [33]. Durch Anwendung von `dynamic()` wird ermöglicht, gewünschte Komponenten und importierte Bibliotheken ins Bundle nachzuladen, sobald diese benötigt werden. Eine mögliche Lösung wäre, dies zu verwenden, damit das Kernsystem gemäss Listing 5.5 dynamisch zur Laufzeit die benötigte Komponente vom aktivierten Theme lädt.

```
1 import dynamic from "next/dynamic"
2 const productDetail = dynamic(() => import(‘../${theme}/productDetail‘))
```

Listing 5.5: Dynamischer Import einer React Komponente

Die Problematik besteht darin, dass die Anwendung eines Template-Strings (`${theme}`) nicht funktioniert. Der Pfad muss explizit geschrieben werden, damit Next.js zur Buildzeit den Import mit dem Modulidentifikator von Webpack abstimmen kann [33]. Dies hat zur Konsequenz, dass beim Umschalten eines Themes oder beim Hinzufügen eines neuen Plugins diese Imports innerhalb des Kernsystems angepasst werden müssen.

5.4.3 Module Federation

Im Jahr 2020 hat Webpack mit der Version 5.0 ein neues Konzept namens Module Federation eingeführt. Dieses Konzept ermöglicht Webpack durch die Integration eines Plugins, dass Applikationen zur Laufzeit Source-Code von anderen JavaScript-basierten Applikationen laden können. Für jede Anwendung wird eine eindeutige Eingabedatei generiert, die externen Anwendungen den Zugriff auf ihre Komponenten ermöglicht. Neben der Möglichkeit, Source-Code zu teilen, bietet Module Federation auch die Möglichkeit, externe Abhängigkeiten zwischen Applikationen zu teilen. [55].
Das Ziel von Module Federation besteht in der Realisierung des Konzepts von Microservices im Frontend, wobei in diesem Kontext meist von Microfrontends gesprochen wird [16]. Dazu wird ein Frontend in kleinere, eigenständige Applikationen unterteilt (Remotes), welche unabhängig entwickelt und 'deployed' werden. Die Host-Applikation importiert die Remotes und fügt sie zu einer ganzen Applikation zusammen.

[14] Code Splitting in Webpack ist eine Technik, die es ermöglicht, den JavaScript-Code einer Anwendung in mehrere kleinere Bündel aufzuteilen, die dann basierend auf der Anforderung zur Laufzeit geladen werden können, um die anfängliche Ladezeit zu reduzieren und die Performance zu verbessern[54].

In Abbildung 5.13 wird schematisch aufgezeigt, wie der Host und die Remotes zusammenspielen.

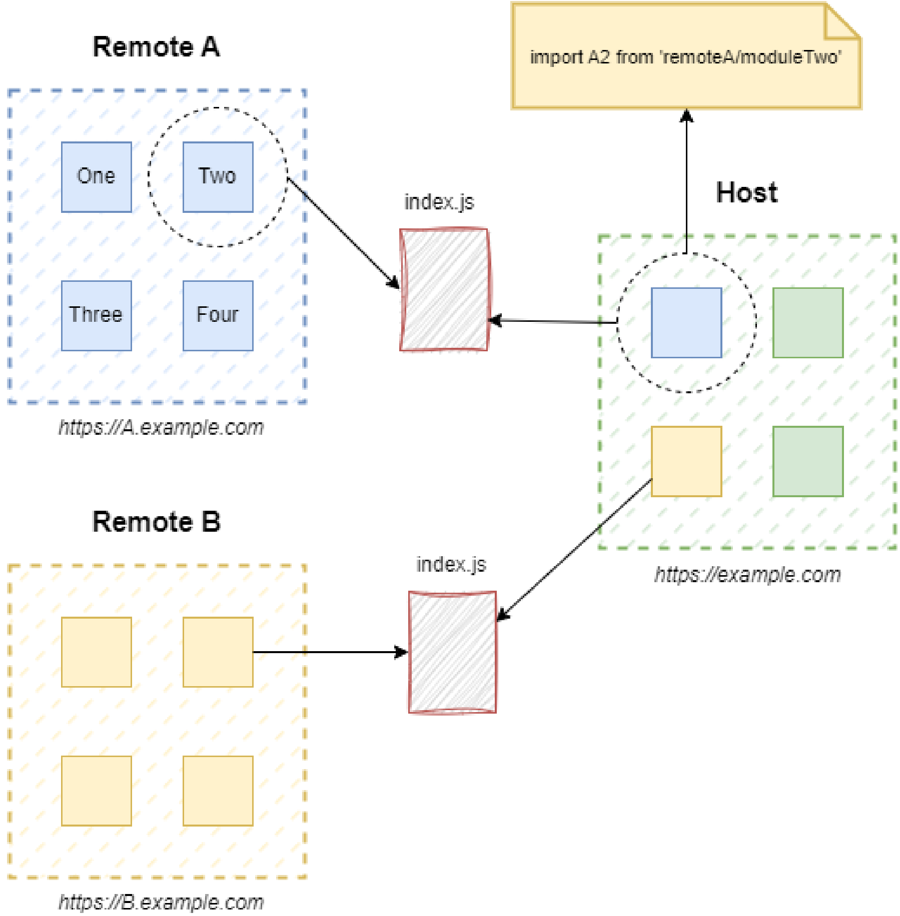


Abbildung 5.13: Konzept von Module Federation

Im Normalfall sieht die Anwendung von Module Federation vor, dass beispielsweise für jede Seite einer Applikation ein Team zuständig ist, welches in einem Remote entwickelt und entsprechend unabhängig deployed wird. Im Kontext dieser Arbeit ist von Interesse, den Host als Kernapplikation zu nutzen und über ein Remote das Theme oder ein Plugin zur Verfügung zu stellen. Der wesentliche Vorteil besteht darin, dass ein Theme angepasst werden kann, ohne im Kernsystem ein neuer Build durchzuführen.

Für die prototypische Umsetzung wurde ein Remote (`theme`) und eine Hostapplikation in Next.js als Kernsystem erstellt.

Listing 5.6 zeigt die Konfiguration in der Datei `next.config.js`, mit der die Hostapplikation eingerichtet wurde, um den Remote zu konsumieren.

```
NextFederationPlugin = require('@module-federation/nextjs-mf');

module.exports = {
  webpack(config, options) {
    if (!options.isServer) {
      config.plugins.push(
        new NextFederationPlugin({
          name: 'shop',
          filename: 'static/chunks/remoteEntry.js',
          remotes: {
            theme: 'http://localhost:3000/_next/static/chunks/remoteEntry.js',
            plugins: 'http://localhost:3001/_next/static/chunks/remoteEntry.js',
          },
        }),
      );
    }
    return config;
  },
};
```

Listing 5.6: Einrichtung Module Federation Host

Innerhalb des Hosts kann nun gemäss Listing 5.7 die gewünschte Komponente konsumiert werden.

```
const productDetail = dynamic(() => import('theme/productDetail'))
```

Listing 5.7: Module Federation dynamischer Import

Um den dynamischen Import vom Remote ausführen zu können, muss der Remote die Komponenten gemäss Listing 5.8 exponieren.

```
const NextFederationPlugin = require('@module-federation/nextjs-mf');

module.exports = {
  webpack(config, options) {
    const { webpack } = options;
    filename: 'static/chunks/remoteEntry.js',
    exposes: {
      './productDetail': './theme1/productDetail.js',
      './productCard': './theme1/productCard.js',
    },
    return config;
  },
};
```

Listing 5.8: Einrichtung Module Federation Remote

Beim Umschalten von 'Theme1' auf 'Theme2' kann der Export an dieser zentralen Stelle im Remote angepasst werden, ohne dass eine Änderung an der Hostapplikation erforderlich ist. Ein weiterer grosser Vorteil ist, dass bei einem Update des aktiven Themes kein neuer Build des Kernsystems erforderlich ist und der Remote unabhängig 'deployed' werden kann. Diese Vorteile gelten ebenfalls für die Implementierung des Plugin-Systems, da mit Module Federation nicht nur Komponenten, sondern auch Klassen geteilt werden können.

Als Nachteil von Module Federation kann angeführt werden, dass es sich noch in einem instabilen Zustand befindet und die Adaption durch Frameworks wie Next.js nicht vorhanden ist. Im Falle von Next.js gibt es keine offizielle Lösung, diese Technologie anzuwenden. Bis Mitte Juli 2021 hatte der Erfinder von Module Federation, Zack Jackson, ein Paket verkauft, welches mittels eines Next.js Plugins die Anwendung von Module Federation ermöglicht. Dieses Paket wurde danach als Open-Source-Software zur Verfügung gestellt. Gemäss diversen GitHub-Diskussionen [24] sowie den Erfahrungen aus der prototypischen Umsetzung sind jedoch noch diverse Probleme zu verzeichnen, weshalb die Integration in Next.js noch als instabil erachtet wird. Diese Aussagen lassen sich durch die folgende Aussage im Readme des npm-Pakets bestätigen [25]:
'This plugin attempts to make the experience as seamless as possible, but it is not perfect.'

5.4.4 Monorepo

Ein Monorepo ist eine Softwareentwicklungsstrategie, bei der im Gegensatz zu einem Polyrepo mehrere Projekte in einem einzigen Repository verwaltet werden. Diese Methode bietet im Kontext dieser Designentscheidung den wesentlichen Vorteil, dass der Source-Code einfacher zwischen Projekten geteilt und der Umgang mit Abhängigkeiten vereinfacht wird. Monorepos werden häufig verwendet, wie zum Beispiel von Google, die ein Monorepo nutzen, das etwa 95% ihres Source-Codes enthält [39].

Der in Abschnitt 5.4.1 vorgestellte Prototyp der Produktdetailkomponente wurde ebenfalls mit einem Monorepo als Basis umgesetzt, um die Machbarkeit zu verifizieren und dann final eine Entscheidungsbasis der drei evaluierten Konzepte zu schaffen.

Im Kontext von Monorepos wird meist von 'Apps' und 'Paketen' gesprochen [50]. Pakete beinhalten eine oder mehrere Softwarekomponenten, Bibliotheken oder Module, die gemeinsam genutzt und in verschiedenen Anwendungen wiederverwendet werden können. Pakete können untereinander Abhängigkeiten haben und werden schliesslich von einem oder mehreren Apps im Monorepo genutzt.
Der im Rahmen des Prototypen verfolgte Ansatz sieht vor, das Kernsystem als Applikation mit Next.js umzusetzen. Das Theme wird als Paket implementiert, welches dann von der Applikation als Abhängigkeit konsumiert wird. Das Paket wird dabei wie ein gewöhnliches npm-Paket aufgebaut, jedoch mit dem Unterschied, dass dieses nicht via npm zur Verfügung gestellt wird, sondern von der Applikation innerhalb des Repositorys als Abhängigkeit konsumiert wird. Der Vorteil dieser Vorgehensweise besteht darin, dass zu einem späteren Zeitpunkt ohne grossen Mehraufwand eine Publikation von einem Theme oder Plugin über eine private npm-Registry[15] möglich ist.

[15] Eine npm-Registry ist eine öffentliche oder private Datenbank, die JavaScript-Pakete speichert und diese für die Installation und Verwaltung über den Node Package Manager (NPM) verfügbar macht.

Das Paket *theme* beinhaltet ein *package.json*, welches die Komponenten vom aktivierten Theme gemäss Listing 5.9 exportiert.

```
{
  "name": "@repo/theme",
  "version": "1.0.0",
  "private": true,
  "exports": {
    "./ProductDetail": "./src/theme1/ProductDetail.tsx"
    "./ProductCard": "./src/theme1/ProductCard.tsx"
  }
}
```

Listing 5.9: Definition vom Theme-Paket im package.json

Um das *package.json* übersichtlich zu gestalten, können alle Exports in eine Datei *index.ts* verlagert werden, die dann als zentrale Schnittstelle für den Paket-Export dient.

Die konsumierende App kann nun das Theme-Paket wie eine gewöhnliche npm-Abhängigkeit im *package.json* importieren. Die Anwendung der Komponente erfolgt gemäss Listing 5.10.

```
import { ProductDetail } from "@repo/theme";
```

Listing 5.10: Anwendung Theme Komponente

Der wesentliche Vorteil gegenüber dem Ansatz 'Dynamic Import' aus Abschnitt 5.4.2 besteht darin, dass in diesem Fall das Kernsystem vollständig unabhängig vom aktivierten Theme ist und letztlich die Komponente darstellt, welche vom Paket als Export zur Verfügung gestellt wird. Dies bedeutet beispielsweise, dass eine Umschaltung von 'Theme1' auf 'Theme2' keine Anpassung des Kernsystems erfordert. Dies ist von entscheidender Bedeutung, um das System im Einklang mit NFA6 entwickeln zu können.

5.4.5 Entscheidung

Es wurde entscheiden, nicht mit Module Federation zu arbeiten, um die Einhaltung von NFA2 (Zukunftssicherheit) zu gewährleisten. Da es keine offizielle Integration von Next.js gibt und das aktuelle Plugin aus privater Initiative entwickelt wurde, ist nicht garantiert, dass diese Lösung weiterhin zur Verfügung steht. Der Entwickler Zack Jackson schreibt im Github-Issue zur Integration von Module Fedaration in Next.js folgendes [24]:
'Update: There will never be official support for Module Federation from Vercel. They believe everything should be available at build time. It will never be supported by Next.js'

Um Zukunftssicherheit zu garantieren ist vorausgesetzt, dass der Framework-Hersteller Interesse an der Integration von Module Federation hat.

Der Entscheid fiel deshalb darauf, das dynamische Laden und Teilen von Source-Code als Basis für ein Theming- und Plugin-System mit einem Monorepo und internen npm-Paketen umzusetzen. Der einzige Nachteil bei diesem Konzept ist, dass bei einem Update oder beim Hinzufügen von einem Theme/Plugin ein neuer Build der gesamten Applikation benötigt wird, da Webpack zur Buildzeit den gesamten Source-Code kennen muss. Der vorgesehene Dev-Prozess für die Entwicklung der Themes/Plugins wird in Abschnitt 5.9.6 ausgeführt.

Die Entscheidung lässt sich gemäss der Struktur des Y-Template [58] folgendermassen zusammenfassen:
'Im Kontext der Evaluation eines Architekturkonzeptes für das dynamische Laden und Teilen von Sourcode als Basis des Theming- und Plugin-Systems mit der Anforderung keine Anpassungen am Source-Code vom Kernsystem vornehmen zu müssen, fiel die Entscheidung mit einem Monorepo und internen npm-Paketen zu arbeiten und gegen den Einsatz von Module Federation, da Module Federation noch keine Stabilität garantieren kann und folglich nicht in Einklang mit NFA2 (*Zukunftssicherheit*) gebracht werden kann, wobei als Konsequenz akzeptiert wird, dass bei einem Update eines Themes/Plugins ein Rebuild der gesamten Applikation benötigt wird.'

5.5 Aufbau des Monorepos

Die in dem vorherigen Abschnitt beschriebene Designentscheidung legt fest, dass jedes Theme oder Plugin seine Komponenten als internes npm-Paket bereitstellt, das vom Kernsystem verwendet wird. Abbildung 5.14 zeigt die Struktur des Monorepos in einem C4-Containerdiagramm [4], wobei jeder Container einem internen npm-Paket entspricht. Es ist zu beachten, dass für jedes Kundenprojekt ein eigenes Monorepo angelegt wird.

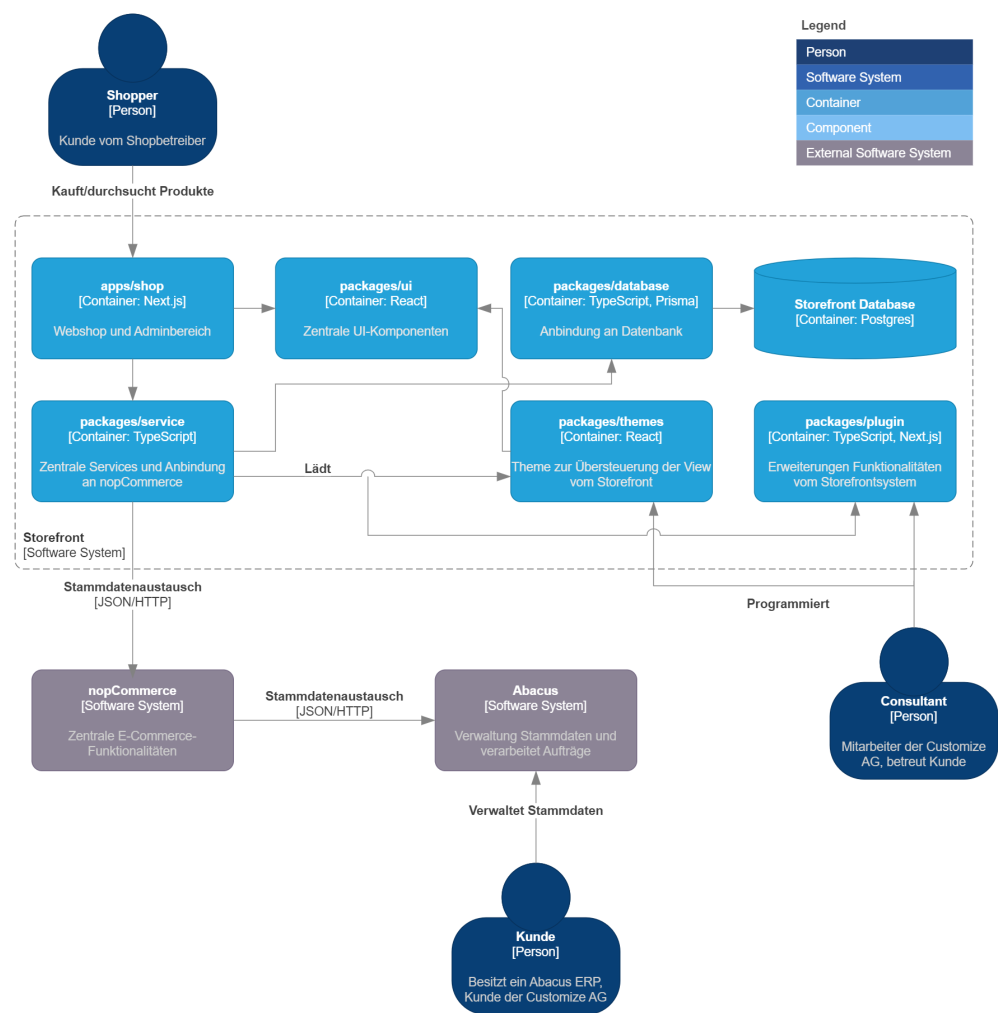


Abbildung 5.14: Struktur des Monorepos (C4-Containerdiagramm)

Die Container *packages/themes* und *packages/plugins* umfassen das Theming- und Plugin-System, die jeweils an die spezifischen Anforderungen jedes Kunden angepasst werden. Alle übrigen Container definieren das Kernsystem und sind für jede Kundeninstanz identisch.

In den beiden nachfolgenden Abschnitten werden die Struktur der Container *packages/themes* und *packages/plugins* aufgezeigt, die es ermöglichen, das Storefrontsystem an die individuellen Bedürfnisse der Kunden anzupassen.

5.6 Theming System

Aufbauend auf dem in Abschnitt 5.4 beschriebenen Designentscheid, der ein Monorepo mit internen npm-Paketen als Grundlage für das Theming- und Plugin-System festlegt, widmet sich dieser Abschnitt der Implementierung des Theming-Systems. Der im Zuge des Designentscheids entwickelte Prototyp konnte durch iterative Verfeinerungen weiter verbessert werden. Dabei wurden insbesondere die Implementierungsdetails optimiert.

5.6.1 Einführung

Um das Verständnis der Ziele vom Theming-System zu vertiefen, wird in Abbildung 5.15 anhand eines Beispiels der Komponente *ProductCard* aufgezeigt, was mit der Implementation eines Themes erreicht werden kann. Im Zuge dieser Arbeit wurden zwei Themes entwickelt, welche dem Industriepartner zur Verfügung gestellt werden. Das 'Standardtheme' wird bei der Installation einer neuen Instanz standardmässig ausgeliefert und bestimmt entsprechend das initiale Design vom Storefrontsystem. Damit kann der Industriepartner für kleine Kunden ohne spezielle Anforderungen innerhalb kurzer Zeit einen Webshop aufsetzen. Mit der Implementation vom 'Moderntheme' wird das Ziel verfolgt, anhand eines Beispiels die Konfigurationsmöglichkeiten über ein Theme vom Storefrontsystem aufzuzeigen.

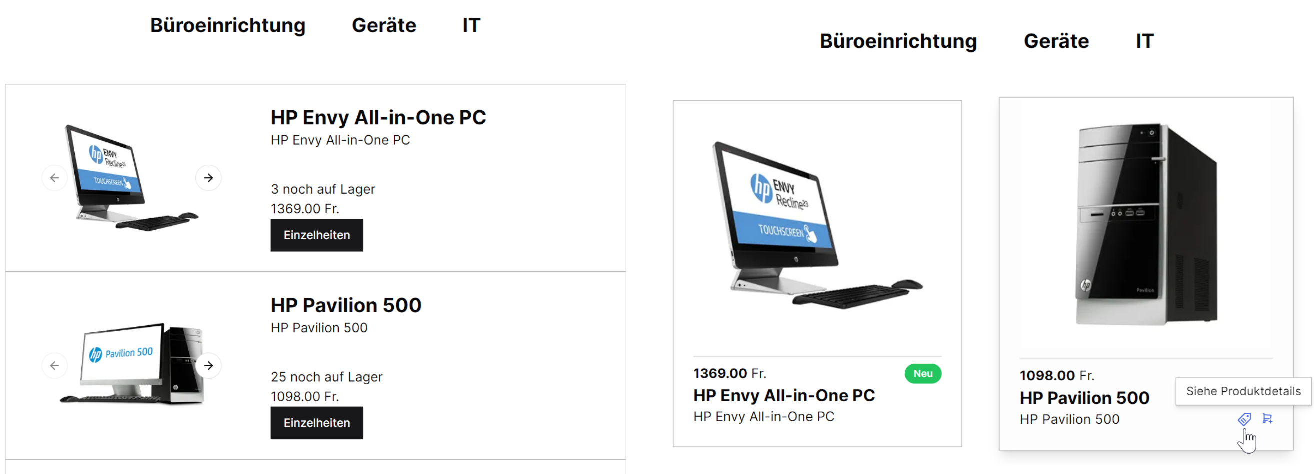


Abbildung 5.15: Vergleich Produktdetail 'Standardtheme' vs. 'Moderntheme'

Im 'Standardtheme' wird die Komponente *ProductCard* als Liste dargestellt. Durch die Implementation vom 'Moderntheme' kann dies nun beispielsweise in eine komplett andere Darstellung als viereckige Card umgebaut werden.

Der nächste Abschnitt dokumentiert, wie eine Theme-Komponente aufgebaut ist und wie ein Consultant diese implementieren kann.

5.6.2 Aufbau der Theme-Komponenten

Bei der Erhebung der funktionalen Anforderungen wurde der Akteur Consultant vorgestellt (siehe Abschnitt 3.2). Dabei wurde erwähnt, dass es auch Consultants gibt, welche lediglich grundlegende Programmierkenntnisse besitzen. Es ist für den Industriepartner ausserordentlich wichtig, dass auch diese Personen Anpassungen am Theme vornehmen können. Daraus folgend wurde das NFA *Einfachheit eines Themes* definiert, welche dies sicherstellen soll. Die abzulösende Lösung 'AbaShop' erfüllte dies, indem HTML-Templates mit der Möglichkeit über die Technologie Jakarta Server Page (JSP)[16] serverseitige Aufrufe auf die Stammdaten durchzuführen. Alle Consultants sind entsprechend geübt, Kundenanforderungen mit dem Anpassen von HTML-Templates und dem Schreiben von Cascading Style Sheets (CSS) umzusetzen.

Das Storefrontsystem wurde in einzelne Komponenten aufgeteilt, die durch die Implementierung von Themes individuell angepasst werden können. Um die nichtfunktionale Anforderung (NFA) der *Einfachheit eines Themes* zu erfüllen, bestehen die Theme-Komponenten ausschliesslich aus präsentierenden Elementen ohne eigene Logik. Diese Strukturierung orientiert sich am *Container and Presentation Pattern*, wie es von Michele Bertoli eingeführt wird [3]. Dabei fungieren die Theme-Komponenten als präsentierende Elemente, die ausschliesslich für die Darstellung zuständig sind, während die Geschäftslogik innerhalb des Kernsystems in separaten Container-Komponenten gekapselt ist. Abbildung 5.16 veranschaulicht die Funktionsweise dieses Patterns durch ein UML-Komponentendiagramm einer einfachen Todo-Anwendung.

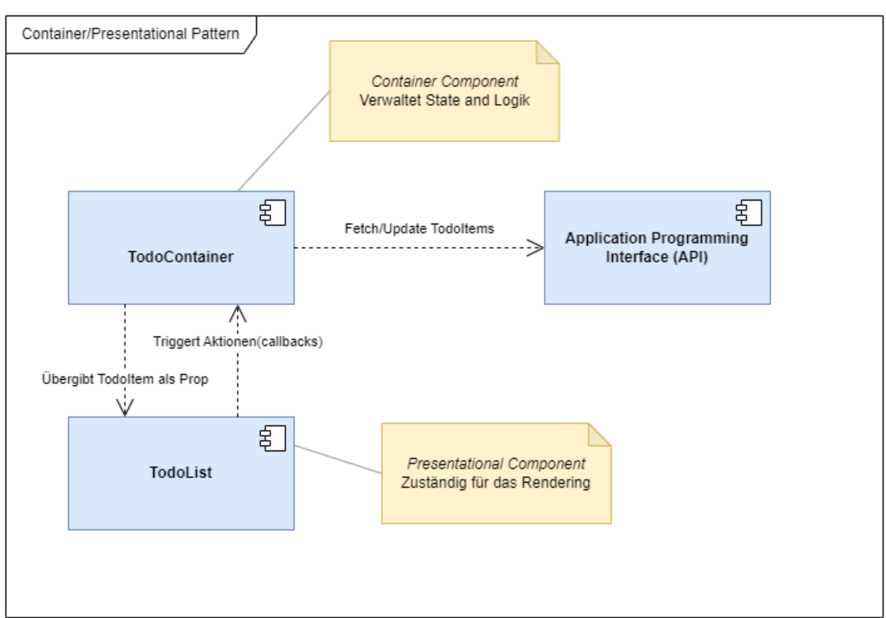


Abbildung 5.16: Container/Representational Pattern (UML-Komponentendiagramm)

[16] JSP-Templates sind Vorlagen in Jakarta Server Pages, die es Entwicklern ermöglichen, dynamischen Inhalt in Webseiten zu integrieren, indem sie Java-Code in HTML einbetten.

Das folgende Listing 5.11 zeigt ein vereinfachtes Beispiel der Produktdetailkomponente, bei dem alle Daten und Logiken als Argumente (Props) an die *Presentational* Theme-Komponente übergeben werden. Der einfache Aufbau der Komponenten erlaubt es Consultants, basierend auf ihrer Erfahrung mit dem 'AbaShop' und ihren vorhandenen Kenntnissen, die Kundenbedürfnisse effizient umzusetzen. Für Consultants mit fortgeschrittenen Programmierkenntnissen besteht zudem die Möglichkeit, auch komplexere Anforderungen zu realisieren.

```
function ProductDetail({product, translate, addToCart}: ProductDetailProps) {
  return (
    <main className="container mb-20">
      <h1 className="font-bold text-5xl mb-1">{product.Name}</h1>
      <p>{product.ShortDescription}</p>
      <section className="flex flex-row justify-between mt-5">
        <h2 className="font-bold text-2xl mb-3">
          {translate("informations")}
        </h2>
        <div dangerouslySetInnerHtml={{ __html: product.FullDescription }} />
        <div className="felx flex-row mt-2">
          <span>{product.Price.toFixed(2)} Fr.</span>
          <Button onClick={() => addToCart(product.Id, 1)}>
            {translate("buy")}
          </Button>
        </div>
      </section>
      <section className="mt-5">
        <h2>{translate("productSpecification")}</h2>
        <div className="grid grid-cols-[auto,1fr]">
          {product.CustomAttributes.map((attribute, index) => (
              <strong key={index}>{attribute.Name}:</strong>
              <span>{attribute.Value}</span>
          ))}
        </div>
      </section>
    </main>
  );
}
```

Listing 5.11: Beispiel einer Theme-Komponente (*ProductDetail*)

5.6.3 Farbkonzept

In der User Story *US7 - Fargebung editieren* (siehe Abschnitt 3.4) wird die Anforderung beschrieben, dass ein Consultant oder auch ein Kunde die Farbgebung des Systems ohne Schreiben von Source-Code anpassen kann. Um diese Möglichkeit zur Verfügung zu stellen, wurde im gesamten System ein Farbkonzept integriert, welches im Kernsystem wie auch im 'Standardtheme' durchgängig genutzt wird. Da die Farbgebung thematisch zu einem Theme gehört, wurde diese darin integriert, so dass ein Theme-Entwickler die gewünschten Standardfarben im Theme definieren kann.

In der modernen CSS-Entwicklung sind Custom Properties, auch bekannt als CSS-Variablen, ein wichtiges Instrument, um eine einheitliche Farbgestaltung innerhalb von Benutzeroberflächen zentral zu steuern. Grant betont in seinem Buch 'CSS in Depth' [20], dass CSS-Variablen die Manipulation und Wiederverwendung von Werten in einem Stylesheet ermöglichen, was einen flexibleren und dynamischeren Designansatz fördert. Ausserdem wird die Möglichkeit erläutert, ein Theming-System mithilfe von CSS-Variablen zu implementieren. Die vorgestellte Lösung setzt jedoch voraus, dass die CSS-Variablen zur Buildzeit definiert sind. Diese Idee wurde in der Umsetzung erweitert, so dass die Variablen zur Laufzeit über eine Benutzeroberfläche im Adminbereich angepasst werden können.

Aufbau CSS-Variablen

Für die Farbgebung wird HSL[17] verwendet, was für Hue, Saturation und Lightness (Farbton, Sättigung und Helligkeit) steht. In der Umsetzung vom Farbkonzept wurde eine bestimmte Anzahl von Variablen definiert, welche durch ein Theme mit einem HSL-Wert versehen werden kann. Listing 5.12 zeigt, in welcher Form ein Theme die Farben über eine Variable definiert.

```
export const baseThemeColors : Color[] = [
  { name: "--background", hue: 0, saturation: 0, lightness: 100, alpha: 1 },
  { name: "--foreground", hue: 240, saturation: 10, lightness: 3.9, alpha: 1 }
]
```

Listing 5.12: Definition CSS-Variable einer Farbe

Die verfügbaren Variablen sind durch das Kernsystem vordefiniert. Sollten künftig zusätzliche Variablen benötigt werden, können diese in der Datei *tailwind.config.js* hinzugefügt und in Base-Theme mit einem Standardwert versehen werden. Dieses Vorgehen ist rückwärtskompatibel für bestehende Themes, da nicht alle Farben definiert werden müssen und fehlende Farben automatisch vom Base-Theme übernommen werden. Es ist besonders wichtig, dass die Consultants bei der Entwicklung neuer Themes konsequent CSS-Variablen verwenden und darauf achten, möglichst keine fest codierten Farbwerte zu benutzen, da ansonsten die Anpassung der Farben zur Laufzeit im Adminbereich nicht die gewünschte Wirkung erzielt.

[17] HSL ist ein Farbmodell, das Farben durch drei Komponenten beschreibt: Hue (Farbton), Saturation (Sättigung) und Lightness (Helligkeit). Hue wird als Winkel auf dem Farbkreis angegeben, Saturation beschreibt die Reinheit oder Intensität der Farbe, und Lightness repräsentiert die Helligkeit von dunkel bis hell [14].

In Tabelle 5.3 sind alle verfügbaren CSS-Variablen zusammen mit ihrem vorgesehenen Einsatzzweck dokumentiert. Die Anwendung dieser Variablen liegt letztlich in der Verantwortung des Theme-Entwicklers, der sie je nach Kontext beliebig nutzen kann. Für eine zuverlässige manuelle Anpassung der Farben im Adminbereich durch den Kunden ist ein zielgerichteter Einsatz gemäss Definition dieser Variablen unerlässlich. Jede Variable verfügt über eine *--foreground* Instanz, die es ermöglicht, einen passenden Vordergrund für die Farben festzulegen, was normalerweise die Definition der Schriftfarbe betrifft. Die Tabelle zeigt allerdings nicht, dass jede Variable zusätzlich durch Voranstellen von *dark* modifiziert werden kann, um die Farbeinstellungen für den Dark Mode anzupassen. So lässt sich beispielsweise mit *--dark-primary* die Hauptfarbe im Dark Mode festlegen.

Name	**Vorgesehener Einsatz**
`--background` / `--foreground`	Bestimmt den Hintergrund und die Schriftfarbe der gesamten Applikation.
`--primary` / `--primary-foreground`	Zweite Hauptfarbe, die meist im Corporate Design einer Firma definiert ist.
`--secondary` / `--secondary-foreground`	Dritte Hauptfarbe, die meist im Corporate Design einer Firma definiert ist.
`--card` / `--card-foreground`	Bestimmt die Farbe von Card-Komponenten, beispielsweise die Product-Card aus Abschnitt 5.6.1.
`--destructive` / `--destructive-foreground`	Signalisiert eine Operation, welche mit einer gewünschten Vorsicht ausgeführt werden soll.
`--border` / `--border-foreground`	Bestimmt die Farbe der Umrandung von Komponenten.
`--popover` / `--popover-foreground`	Popover-Komponenten werden bei Benutzerinteraktion über die ganze Applikation gesetzt/aufgeklappt.
`--input` / `--input-foreground`	Bestimmt die Farbe von Inputelementen, die meist in Formularen eingesetzt werden.
`--accent` / `--accent-foreground`	Accent wird verwendet, sobald etwas hervorgehoben werden soll, beispielsweise ein wichtiger Link.

Tabelle 5.3: Übersicht verfügbare CSS-Variablen im Theming-System

Anpassung der Farben zur Laufzeit

Damit die Farben in einer Benutzeroberfläche zur Laufzeit angepasst werden können, wird eine Persistierung der aktiven Farben benötigt, da im Gegensatz zum Lösungsansatz von Grant [20] die Farben nicht zur Buildzeit der Applikation in einer CSS-Datei stehen. Anstelle einer CSS-Datei werden die Farben deshalb zur Laufzeit aus einer Datenbank gelesen und im Root-Layout[18] über einen `<style>`-Tag in das Storefrontsystem eingefügt. Listing 5.13 zeigt eine gekürzte Implementation der Root-Layout Komponente.

```
export default async function RootLayout({children, lang}) {

  const colors = await new ColorService().getFormattedColors();
  const dynamicStyles = `
    :root {
      ${Object.entries(colors)
        .filter(([key]) => !key.startsWith("--dark-"))
        .map(([key, value]) => `${key}: ${value};`)
        .join("\n")}
    }`

  return (
    <html lang={lang} dir={dir(lang)}>
      <head>
        <style>{dynamicStyles}</style>
      </head>
      <body className={`${inter.className}`}>
        <main>{children}</main>
      </body>
    </html>
  );
}
```

Listing 5.13: Root-Layout Komponente

Damit die Datenbankaufrufe im `ColorService.ts` nicht bei jedem Aufruf einer Seite ausgeführt werden, sind die Farben mit Hilfe der Infrastruktur von Next.js in einem Cache zwischengespeichert. Wird eine Farbe durch die Installation eines Plugins oder manuelle Anpassung im Adminbereich aktualisiert, wird der Cache invalidiert, so dass die Farben beim nächsten Aufruf aktualisiert werden.

[18] Das Root-Layout ist die oberste Komponente einer Next.js Applikation und umfasst die gesamte Applikation, weshalb die eingefügten CSS-Variablen auf allen Komponenten angewendet werden.

Abbildung 5.17 zeigt, wie ein Consultant oder Kunde im Adminbereich über eine Benutzeroberfläche die Standardfarben des aktivierten Themes bearbeiten kann.

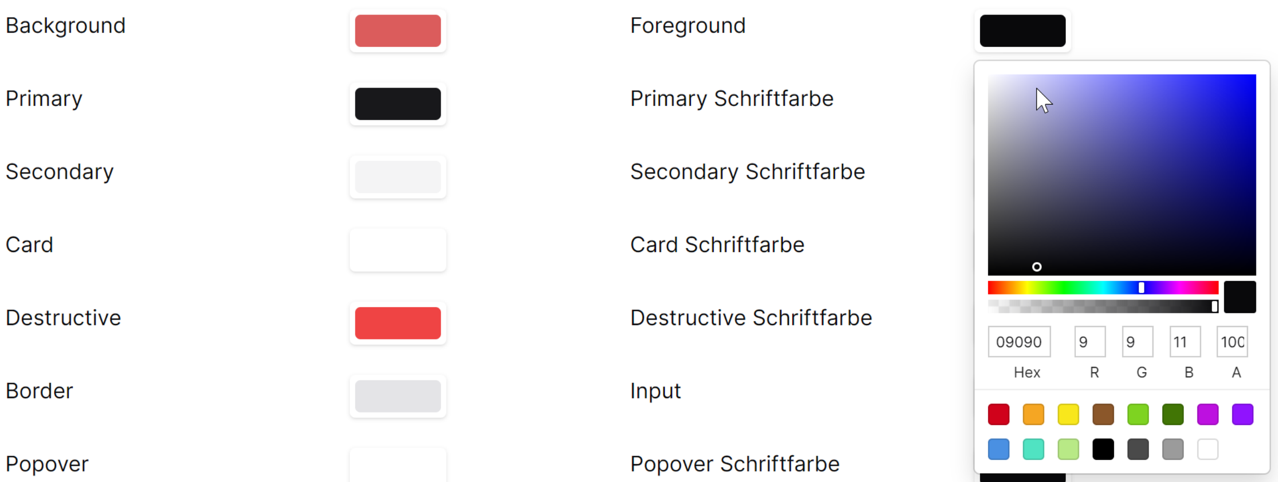


Abbildung 5.17: Anpassen Farben vom Theme im Adminbereich

Bei einer Installation eines neuen Themes werden manuelle Anpassungen durch die Farben im neuen Theme überschrieben. Der Anwender hat jederzeit die Möglichkeit, über den Button *Theme Standard wiederherstellen* seine Anpassungen zurückzusetzen.

5.6.4 Gesamtlayout definieren

Die bis anhin vorgestellten Konzepte ermöglichen es, einzelne Komponenten und die Farbgebung vom gesamten Storefrontsystem gemäss den Kundenanforderungen zu definieren. Allerdings fehlt die Möglichkeit, die Positionierung der Komponenten im gesamten Storefrontsystem festzulegen. Diese Anforderung ist im Use Case *Komponentenlayout anpassen* dokumentiert (siehe Abschnitt 3.5.4).

WordPress bietet im Theming-System diese Möglichkeit (siehe Abschnitt 4.2.2). Innerhalb eines Themes können verschiedene Template-Dateien überschrieben werden:

- `index.php`: Die Haupt-Template-Datei, die als Fallback dient, wenn keine spezifischere Template-Datei vorhanden ist.
- `header.php`: Enthält den HTML-Code für den Kopfbereich der Website, einschliesslich Meta-Tags, Skripten und dem Öffnen des `<body>`-Tags.
- `footer.php`: Enthält den HTML-Code für den Fussbereich der Website, einschliesslich des Schliessens des `<body>`- und `<html>`-Tags.
- `single.php`: Wird verwendet, um einzelne Beiträge anzuzeigen, oft inklusive des vollständigen Inhalts eines Beitrags, Kommentare und Metadaten.
- `page.php`: Wird verwendet, um einzelne Seiten anzuzeigen, ähnlich wie `single.php`, aber speziell für statische Seiten.
- `sidebar.php`: Enthält den Code für die Seitenleiste, die oft Widgets, Navigationslinks oder andere sekundäre Inhalte enthält.

Das Template *index.php* ist verantwortlich, die Platzierung der anderen Templates auf der gesamten Seite zu definieren. Dies wird ermöglicht, indem es über Template-Tags die anderen Templates einbindet.
Die Umsetzung im Storefrontsystem ist durch das im Abschnitt 4.2.2 untersuchte Theming-System von WordPress inspiriert. WordPress verwendet Templates, die in dieser Arbeit Komponenten entsprechen. Im Theming-System von WordPress kann ein Template durch die Verwendung von Template-Tags andere Komponenten anzeigen. Ein Beispiel hierfür ist der Aufruf `<?php get_footer()` innerhalb des Templates *index.php*.

Listing 5.14 zeigt ein Beispiel einer *index.php*-Datei, die beispielsweise durch die Anwendung vom Template-Tag[19] `get_footer()` das Template *footer.php* einbindet.

```
<!DOCTYPE html>
<html <?php language_attributes(); ?>>
<head>
    <title><?php bloginfo('name'); ?></title>
    <?php wp_head(); ?>
</head>
<body <?php body_class(); ?>>
    <?php get_header(); ?>
    <div id="content">
        <?php while (have_posts()) : the_post(); ?>
            <article id="post-<?php the_ID(); ?>" <?php post_class(); ?>>
                <h2><?php the_title(); ?></h2>
                <div class="entry-content">
                    <?php the_content(); ?>
                </div>
            </article>
        <?php endwhile; ?>
    </div>
    <?php get_sidebar(); get_footer(); ?>
</body>
</html>
```

Listing 5.14: Beispiel für eine `index.php`-Datei

Dieser Ansatz diente als Grundlage, um den Use Case *Komponentenlayout anpassen* in angepasster Form im Storefrontsystem zu implementieren. Das Theming-System bietet Layout-Komponenten an, die innerhalb eines Themes überschrieben werden können. Diese Komponenten empfangen als Prop-Argumente[20] andere Komponenten. Falls das Theme eine der Komponenten im Prop selbst definiert, wird in diesem Fall die eigene übergeben, andernfalls die Komponente aus dem *BaseTheme*. Somit ermöglicht das Theme die Definition der Platzierung der 'normalen' Komponenten im gesamten Storefrontsystem.

[19] Template-Tags in PHP sind spezielle Ausdrücke, die in HTML-Dokumenten verwendet werden, um PHP-Code auszuführen und dynamische Inhalte direkt in die Webseite zu integrieren.

[20] In React ist eine Prop (Kurzform für 'Property') ein Argument, das an eine Komponente übergeben wird. Props ermöglichen es, benutzerdefinierte Daten von einer übergeordneten Komponente zu einer untergeordneten Komponente zu übertragen.

Listing 5.15 demonstriert, wie eine Layoutkomponente im Theme definiert ist.

```
export default function FooterLayout({
  AddressSection,
  ContactSection,
  InformationSection,
  LanguageChanger
}: FooterLayoutProps) {
  return (
    <div className="w-full bg-secondary py-5">
      <div className="flex flex-row flex-wrap text-secondary-foreground">
        {InformationSection}
        {AddressSection}
        {ContactSection}
        <section>
          <h2 className="font-bold flex flex-row items-center gap-x-1">
            <IconLanguage />
            <span>Sprache</span>
          </h2>
          {LanguageChanger}
        </section>
      </div>
    </div>
  );
}
```

Listing 5.15: Layout-Komponente im Theme

Der wesentliche Unterschied zum Konzept von WordPress besteht darin, dass in Storefrontsystemen die Layout-Komponenten ihre verfügbaren Komponenten als Argumente vom Kernsystem erhalten, anstatt Template-Tags zur freien Nutzung aller Komponenten zu verwenden.

5.6.5 Übersicht Klassen

In der Abbildung 5.18 wird in einem UML-Klassendiagramm der gesamte Aufbau vom Theming-System dargestellt.

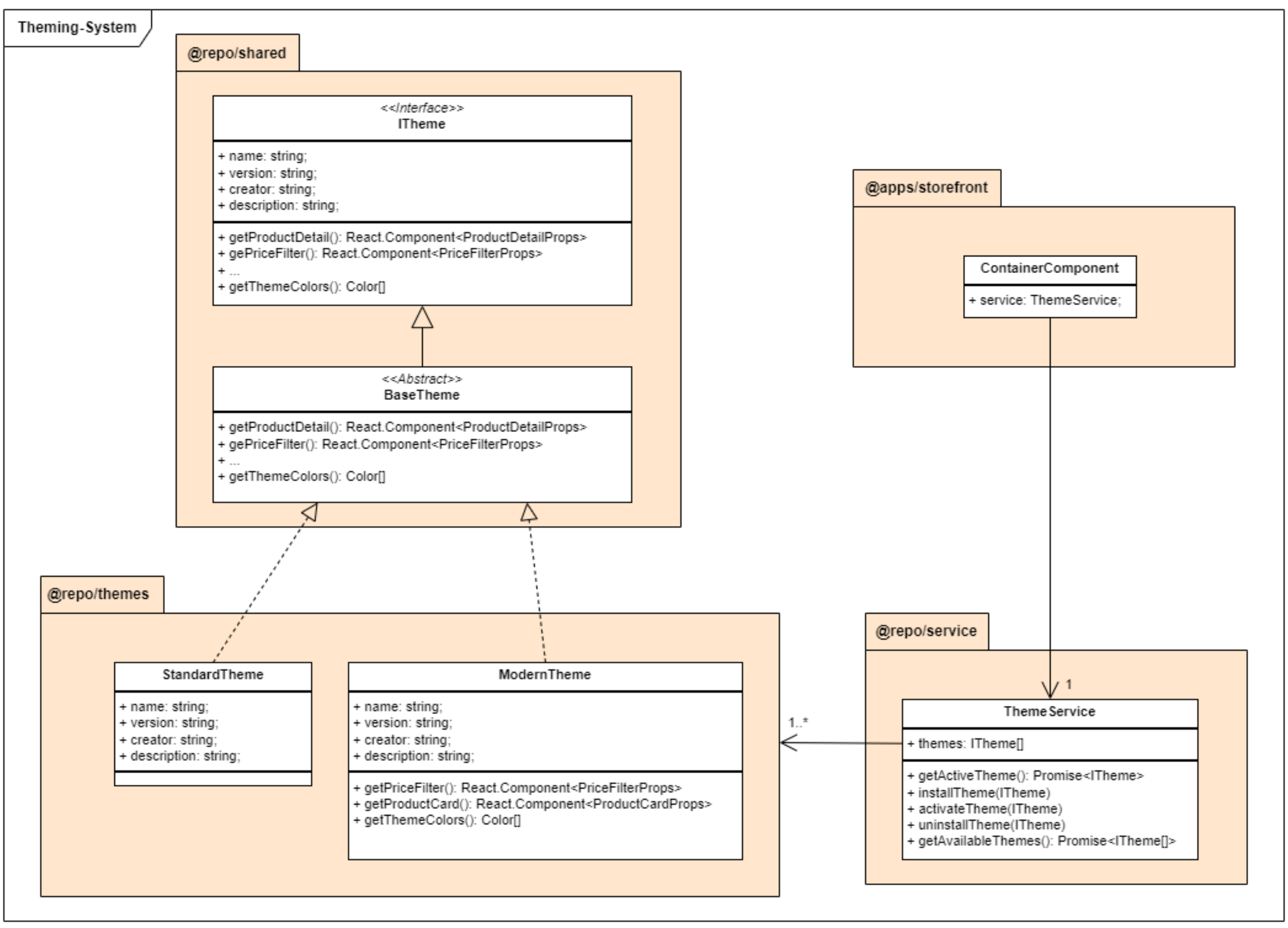


Abbildung 5.18: Aufbau Theming-System (UML-Klassendiagramm)

Das Interface *ITheme* definiert eine Schnittstelle, die von jedem Theme implementiert werden muss. Die darin enthaltenen Instanzvariablen spezifizieren die Metadaten eines Themes, die für dessen Verwaltung und für die Darstellung im Adminbereich erforderlich sind. Die definierten Methoden liefern die Presentational React-Komponenten (siehe Abschnitt 5.6.2), die vom Kernsystem letztlich angezeigt werden. Diese dargestellten Methoden werden in Abbildung 5.18 mit ... abgekürzt, da eine Viezahl von Komponenten überschrieben werden können. Die Methode *getThemeColors()* gibt eine Liste der im vorherigen Abschnitt beschriebenen CSS-Variablen zurück. Zusätzlich definiert das Paket *@repo/shared* eine abstrakte Klasse, die alle Methoden des Interfaces *ITheme* implementiert. Dies ermöglicht es, dass in einem Theme lediglich die vom Entwickler zu überschreibenden Komponenten spezifiziert werden müssen. Das 'Standardtheme' überschreibt daher keine Komponenten, was bedeutet, dass im Kernsystem alle Standardkomponenten angezeigt werden. Bei der Implementierung eines eigenen Themes, wie beispielsweise *ModernTheme*, kann der Entwickler selbst wählen, welche Komponenten er durch

Überschreiben der entsprechenden Methoden anpassen möchte. Die Klasse *ThemeService* Importiert alle Implementationen vom Paket *@repo/themes* und stellt dem Kernsystem das aktivierte Theme zur Verfügung. In den Container-Komponenten vom Kernsystem werden über diesen Service die darin benötigten Komponenten gemäss Listing 5.16 abgeholt und dargestellt.

```
export function ProductPage({ params: { slug }}: ProductPageProps) {

const product = await new ProductsService().getProductById(slug)
const theme = await new ThemeService().getActiveTheme();
const ProductDetail = theme.getProductDetailComponent();

return (
  <ProductDetail product={product} />
);
}
```

Listing 5.16: Anwendung Theme im Kernsystem

5.6.6 Export der Themes

In der im Abschnitt 5.4 dokumentierten Designentscheidung wird erläutert, weshalb das dynamische Laden von Source-Code als Basis für das Theming-System mit einem Monorepo durchgeführt wird. Im erstellten Prototypen als Basis für den getroffen Designentscheid wird aufgezeigt, dass im Paket *@repo/themes* in der Datei *package.json* der Exportpfad aller Komponenten des aktivierten Themes definiert ist. Diese Datei ist der zentrale Einstiegspunkt für das Paket und folglich entscheidend, welche Komponenten vom Bundler inkludiert werden. Der zentrale Schwachpunkt des in Abschnitt 5.4.4 beschriebenen Prototypen des Theming-Systems ist, dass eine Umschaltung von 'Theme1' auf 'Theme2' zur Laufzeit nicht möglich ist. Da in der Exportdefinition der Pfad auf das aktivierte Theme definiert ist, werden vom Paket auch nur diese Komponenten exportiert und schliesslich vom Bundler aufgenommen. Um diese Umschaltung zu ermöglichen, wurde das Konzept des erstellten Prototypen verfeinert.
Im UML-Klassendiagramm in Abbildung 5.18 wird aufgezeigt, dass ein Theme eine Implementation vom Interface *ITheme* ist. Anstelle von Komponenten exportiert das Paket *@repo/themes* Implementationen von diesem Interface. Die Datei *index.ts* definiert gemäss Listing 5.17 zentral die exportieren Theme-Klassen in einem Array. Als Konsequenz nimmt der Bundler alle Komponenten auf, welche von den Objekten des Arrays retourniert werden.

```
import { ITheme } from "@repo/shared";
import { ModernTheme } from "./src/modern/ModernTheme";
import { StandardTheme } from "./src/standard/StandardTheme";

export const themes: ITheme[] = [new StandardTheme(), new ModernTheme()];
```

Listing 5.17: Export Theme-Klassen als Array in index.ts (Paket *@repo/themes*)

Die Klasse *ThemeService* hat eine Abhängigkeit auf dieses Paket und verwaltet die exportierten Themes für das Kernsystem und ermöglicht eine Umschaltung zur Laufzeit. Wird ein komplett neues Theme hinzugefügt, wird lediglich eine Anpassung des Export-Arrays in der Datei *index.ts* als zentrale Schnittstelle benötigt, wobei die weiteren Konfigurationen zur Laufzeit im Adminbereich vorgenommen werden können.

5.7 Plugin System

Dieser Abschnitt beschreibt das Plugin-System, welches zur Erweiterung der Standardfunktionalitäten des Kernsystems eingesetzt wird. Das Ziel des Systems ist es, eine Anpassung des Storefrontsystems an kundenspezifische Spezialanforderungen zu ermöglichen, ohne Änderungen am Kernsystem vornehmen zu müssen. Dabei wurden einige Konzepte des Theming-Systems übernommen, da auch im Plugin-System dynamisch geladene Komponenten dargestellt werden.

5.7.1 Einführung

In der Dokumentation der funktionalen Anforderungen (siehe Abschnitt 3.6) wird das Plugin-System anhand einer User Story eingeführt, die auf einer realen Anforderung eines Kunden des Industriepartners basiert. Der Kunde, ein branchenspezifischer Verband, bietet Gesamtarbeitsverträge an, die von den angeschlossenen Mitgliedsfirmen erworben werden können. Diese Verträge können auch digital erworben werden. Daher möchte der Verband seinen eingeloggten Kunden im Webshop ein Downloadcenter bereitstellen, über das digital erworbene Artikel direkt heruntergeladen werden können. Für dieses Plugin wurden diverse Use Cases im Rahmen der Erhebung der funktionalen Anforderungen an das Plugin-System spezifiziert.
Damit der Consultant konfigurieren kann, bei welchen Produkten ein Download möglich ist, muss das Plugin in der Lage sein, einen neuen Menüpunkt im Administrationsbereich zu erstellen. Unter diesem Menüpunkt wird die Konfigurationsseite des Plugins für Administratoren zugänglich gemacht. Zudem benötigt das Plugin Zugriff auf die Bestellinformationen der aktuell eingeloggten Kunden aus dem Kernsystem, um die Downloadfunktionalität zu unterstützen. Das Downloadcenter soll an einer spezifisch festgelegten Stelle innerhalb der Shopnavigation für Endkunden sichtbar gemacht werden.

Um die vielfältigen Möglichkeiten des Plugin-Systems zu demonstrieren, wurden neben dem oben genannten Downloadcenter weitere Beispiel-Plugins entwickelt, die ebenfalls auf realen Kundenanforderungen des Industriepartners basieren.

- **Statistikplugin:** Dieses Plugin zeigt dem Kunden verschiedene Statistiken des Storefrontsystems an, die nicht aus dem Abacus ERP-System stammen. Ein Beispiel ist eine Auswertung, die zeigt, wie oft ein Artikel in der Detailansicht betrachtet wurde.
- **Promotionplugin:** Dieses Plugin ermöglicht die Verwaltung von Shop-Promotionen. Basierend auf festgelegten Kriterien fügt es automatisch einen Promotionsartikel kostenlos in den Warenkorb des Kunden hinzu. Dies erfolgt beispielsweise, wenn ein Kunde den Artikel X in den Warenkorb legt und daraufhin das Plugin den Promotionsartikel Y automatisch hinzufügt.

In der Analyse des Eclipse-Plugin-Systems (siehe Abschnitt 4.1.2) wird deutlich, dass Erweiterungspunkte zur Verfügung stehen, die von einem Plugin in der Datei *plugin.xml* registriert werden. Hierbei wird eine Klasse eingetragen, die ein von Eclipse spezifiziertes Interface einhalten muss. Ein ähnlicher Ansatz wird im Plugin-System des Storefrontsystems verfolgt. Für jeden Erweiterungspunkt legt das Kernsystem ein Interface fest, das von einem Plugin implementiert werden muss. Anders als bei Eclipse erfolgt die Registrierung nicht über eine Metadatei, sondern durch die Implementation des Interfaces *IPlugin*, das jedes Plugin realisieren muss. Das Kernsystem nutzt an den verfügbaren Erweiterungspunkten alle Klassen dieses Typs, die zudem das jeweils benötigte Interface implementieren. In den folgenden Abschnitten werden die Einsatzmöglichkeiten des Plugin-Systems durch diese Interfaces detailliert beschrieben. In den UML-Klassendiagrammen wird anhand eines Beispiel aufgezeigt, welche für das erläuterte Konzept relevanten Interfaces das Kernsystem zur Verfügung stellt und wie diese von einem Plugin im Paket *@repo/plugins* konsumiert werden können. Für die Implementierung eines Plugins ist es zudem möglich, mehrere dieser Interfaces zu integrieren. Ein Beispiel hierfür ist das Statistikplugin, das die Interfaces aus den Abschnitten 5.7.3, 5.7.4, 5.7.5 und 5.7.6 nutzt.

5.7.2 Definition eines Plugins

Damit das Kernsystem ein Plugin erkennt und verwendet, muss dieses das Interface *IPlugin* implementieren. Dieses Interface definiert die Metadaten für die Verwaltung im Adminbereich und stellt Lifecycle-Methoden[21] zur Verfügung. Die beiden untersuchten Plugin-Systeme, Eclipse und WordPress, bieten die Möglichkeit, bei einer Änderung des Plugin-Status eigene Logik auszuführen. Diese Funktionalität ist auch für das Storefrontsystem von Vorteil, da sie es einem Plugin-Entwickler ermöglicht, bei der Aktivierung oder Deaktivierung seines Plugins entsprechend zu reagieren.
Durch die Implementierung des in Listing 5.18 beschriebenen Interfaces stellt das Kernsystem sicher, dass diese Methoden bei jeder Änderung im Lebenszyklus des Plugins aufgerufen werden.

```
1  export interface IPlugin {
2    readonly name: string;
3    readonly version: string;
4    readonly creator: string;
5    readonly description: string;
6    readonly priority: number;
7
8    onPluginActivation(): Promise<void>;
9    onPluginDeactivation(): Promise<void>;
10 }
```

Listing 5.18: Basis-Interface Plugin

[21] Lifecycle-Methoden sind spezielle Funktionen in einem Software-System, die zu bestimmten Zeitpunkten im Lebenszyklus einer Komponente, wie einem Plugin, aufgerufen werden. Diese Methoden ermöglichen es Entwicklern, benutzerdefinierte Aktionen durchzuführen, wenn die Komponente initialisiert, aktiviert, oder deaktiviert wird. Sie spielen eine entscheidende Rolle bei der Verwaltung von Ressourcen, und dem ordnungsgemässen Abschliessen von Prozessen.

5.7.3 Seitenelement injizieren

Ein Plugin hat die Möglichkeit, eine eigene Seite in das Storefrontsystem zu integrieren (siehe *UC5 - Seitenelement injizieren* in Abschnitt 3.6.3). Im Unterschied zum Theming-System, das lediglich Presentational React-Komponenten verwendet, umfassen diese Seiten eigene Logik.

Bei der Entwicklung eines Plugins kann sich der Entwickler zwischen zwei Interfaces entscheiden: *IShopPlugin*, das die Anzeige eines Seitenelements im Webshop ermöglicht, und *IAdminPlugin*, das eine Seite im Administrationsbereich einbindet, welche ausschliesslich Administratoren des Storefrontsystems zugänglich ist. Die Definition der Schnittstelle für ein Seitenelement im Adminbereich wird im Listing 5.19 dargestellt.

```
export interface IAdminPlugin extends IPlugin {
  readonly endpoint: string;
  getEntryPoint(): ({ service }: PluginPageProps) => Promise<JSX.Element>;
}
```

Listing 5.19: Interface Plugin im Adminbereich

Es wird deutlich, dass im Gegensatz zur Implementierung des Themes hier ein Promise[22] von einer React-Komponente zurückgegeben wird. Der Grund für diesen Ansatz liegt darin, dass ein Seitenelement eines Plugins stets serverseitig gerendert und daher asynchron verarbeitet wird. Diese Implementierung ermöglicht es dem Pluginentwickler, serverseitige Operationen auf seiner Pluginseite durchzuführen, wie beispielsweise das Aufrufen einer externen API. Die Instanzvariable *endpoint* bestimmt den Slug, unter dem das Plugin zugänglich ist.
Im Kernsystem erfolgt die Implementierung mittels des Konzepts der dynamischen Routen[23] von Next.js. Durch das verzeichnisbasierte Routing kann ein Ordner in eckige Klammern gesetzt und mit einem Namen (dynamisches Segment) versehen werden, was dynamisches Routing ermöglicht. Bei Aufruf einer solchen Route übergibt Next.js das dynamische Segment als Prop-Variable an die aufgerufene Komponente. Üblicherweise wird als dynamisches Segment eine ID verwendet, die dann für einen API-Aufruf genutzt wird, dessen Ergebnis in der Komponente angezeigt wird.

[22]Ein Promise in JavaScript ist ein Objekt, das das Ergebnis einer asynchronen Operation darstellt und Methoden bietet, um auf den Erfolg oder Misserfolg dieser Operation zu reagieren.

[23]Dynamische Routen in Next.js erlauben das Definieren von Routen, die zur Buildzeit noch unbekannt sind und ermöglichen so den Zugriff auf Seiten, deren Pfade und Inhalte erst zur Laufzeit festgelegt werden.[32]

Listing 5.20 demonstriert dieses Konzept anhand einer beispielhaften Todo-Applikation.

```
//app/todo/[id]/page.tsx
export default function Page({ params }: { params: { id: string } }) {
  //API Call: GET /api/todo/{id} and store in response
  return <div>Todo {id}: {response.description}</div>
}
```

Listing 5.20: Beispiel Dynamic Routing

Diese Methode kann genutzt werden, um das Plugin-System effektiv umzusetzen. Dabei wird das dynamische Segment *id* durch *pluginSlug* ersetzt. Mit dieser Variable kann anschliessend über den *PluginService* das entsprechende Plugin ermittelt werden, das sich unter diesem Slug registriert hat. Durch die Implementierung des Interfaces *IAdminPlugin* kann die spezifische Komponente, die angezeigt werden soll, abgerufen werden. Das Listing 5.21 demonstriert die Implementierung, die sowohl im Webshop als auch im Adminbereich identisch eingesetzt ist.

```
//Pfad: app/admin/[pluginSlug]/page.tsx
export async function PluginPage({params}: {params: { pluginSlug: string } }) {
  const pluginService = new PluginService();
  const adminPlugins = await pluginService.getAdminPlugins();
  const matchingPlugin = adminPlugins.find(
    (plugin) => plugin.shopEndpoint === params.pluginSlug,
  );
  if (matchingPlugin == undefined) {
    return notFound();
  }
  const PluginPage = matchingPlugin.getEntryPoint();

  return (
    <main className="container">
      <PluginPage service={new PluginFacade()}/>
    </main>
  );
}
```

Listing 5.21: Implementation Plugin Seitenelement injizieren

Abbildung 5.19 veranschaulicht mittels eines UML-Klassendiagramms, wie Plugins eigene Seiten in das Kernsystem integrieren können. Die *ContainerComponent*, dargestellt im vorherigen Listing 5.21, ist dafür verantwortlich, über den *PluginService* das entsprechende Plugin zu identifizieren und mittels der Methode *getEntryPoint* das Seitenelement des Plugins anzuzeigen.

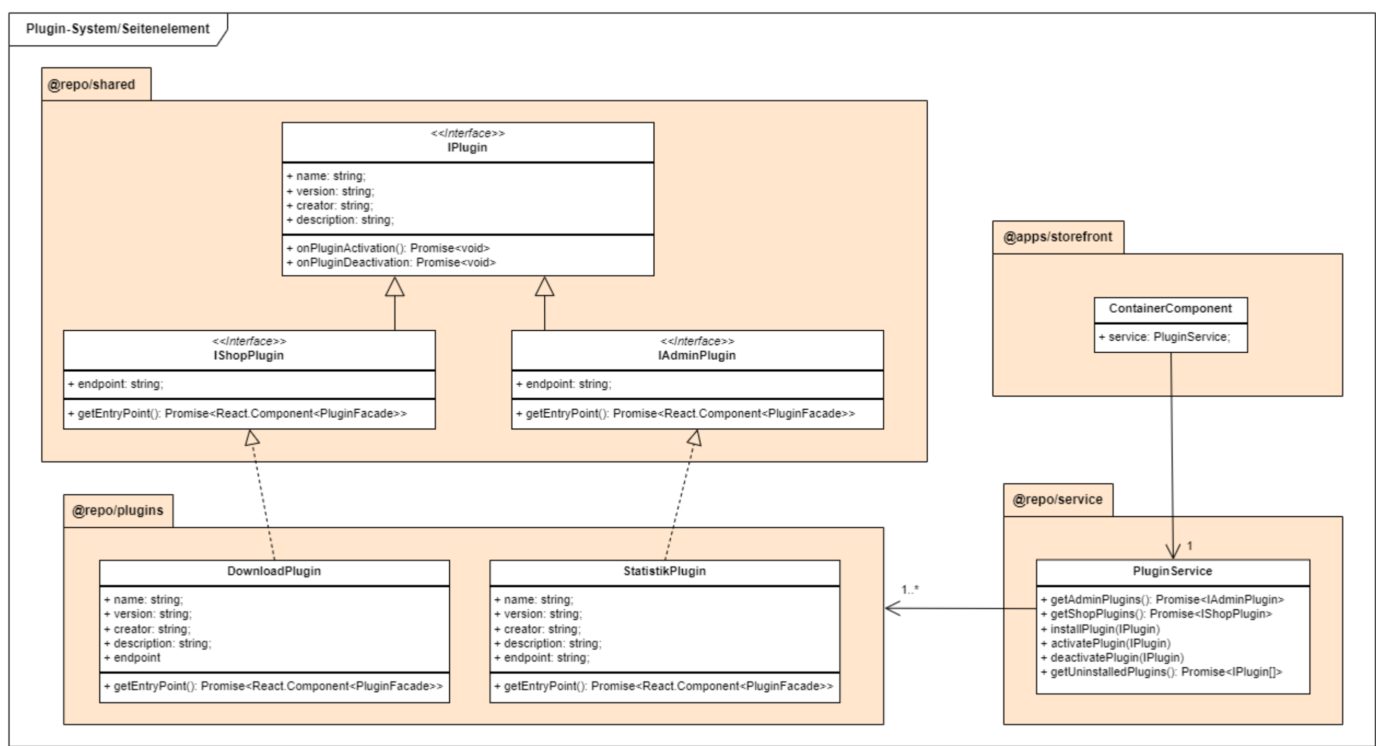


Abbildung 5.19: Plugin: Seitenelement injizieren (UML-Klassendiagramm)

5.7.4 Service konsumieren

Der vorherige Abschnitt erläutert, dass eine Pluginseite komplexe, programmierbare Logik enthalten kann. Oft benötigen Pluginentwickler jedoch eine Interaktion mit den Services des Kernsystems. Beispielsweise erfordert die Implementierung des Promotionsplugins das Hinzufügen eines Artikels zum Warenkorb oder im Downloadcenter das Abfragen von Kundenaufträgen. Allerdings ist es nicht vorgesehen, dass Pluginentwickler eine direkte Abhängigkeit zum Paket *@repo/service* aufbauen, da dieses ein komplexes Subsystem darstellt und viele seiner Services in einem Plugin nicht verwendet werden sollten. Dieses Problem wird durch die Implementierung des Fassaden-Designpatterns gelöst, wie es im Buch *Design Patterns: Elements of Reusable Object-Oriented Software* [15] beschrieben wird. Die Klasse *PluginFacade* bietet Entwicklern eine vereinfachte Schnittstelle zum Service-Paket. Dieses Objekt wird als Prop im Seitenelement des Plugins übergeben (siehe Listing 5.21) und kann dort genutzt werden. Um auch Plugins ohne sichtbares Seitenelement Zugang zu dieser Fassade zu ermöglichen, kann die Plugin-Klasse die abstrakte Klasse *BasePlugin* implementieren, welche die Methode `static getServiceInstance(): IPluginFacade` bereitstellt.

Die Abbildung 5.20 zeigt mittels eines UML-Klassendiagramms, wie das Fassaden-Pattern im Storefrontsystem verwendet wird, um Plugins eine Schnittstelle zu den Services des Kernsystems bereitzustellen. Um die Übersichtlichkeit zu gewährleisten, werden nicht explizit jeder Service und jede Methode modelliert. Die Klasse *PluginFacade* enthält alle erforderlichen Services des Kernsystems als Instanzvariablen und bietet die gemäss dem Interface *IPluginFacade* definierten Methoden durch die Nutzung dieser Services an.

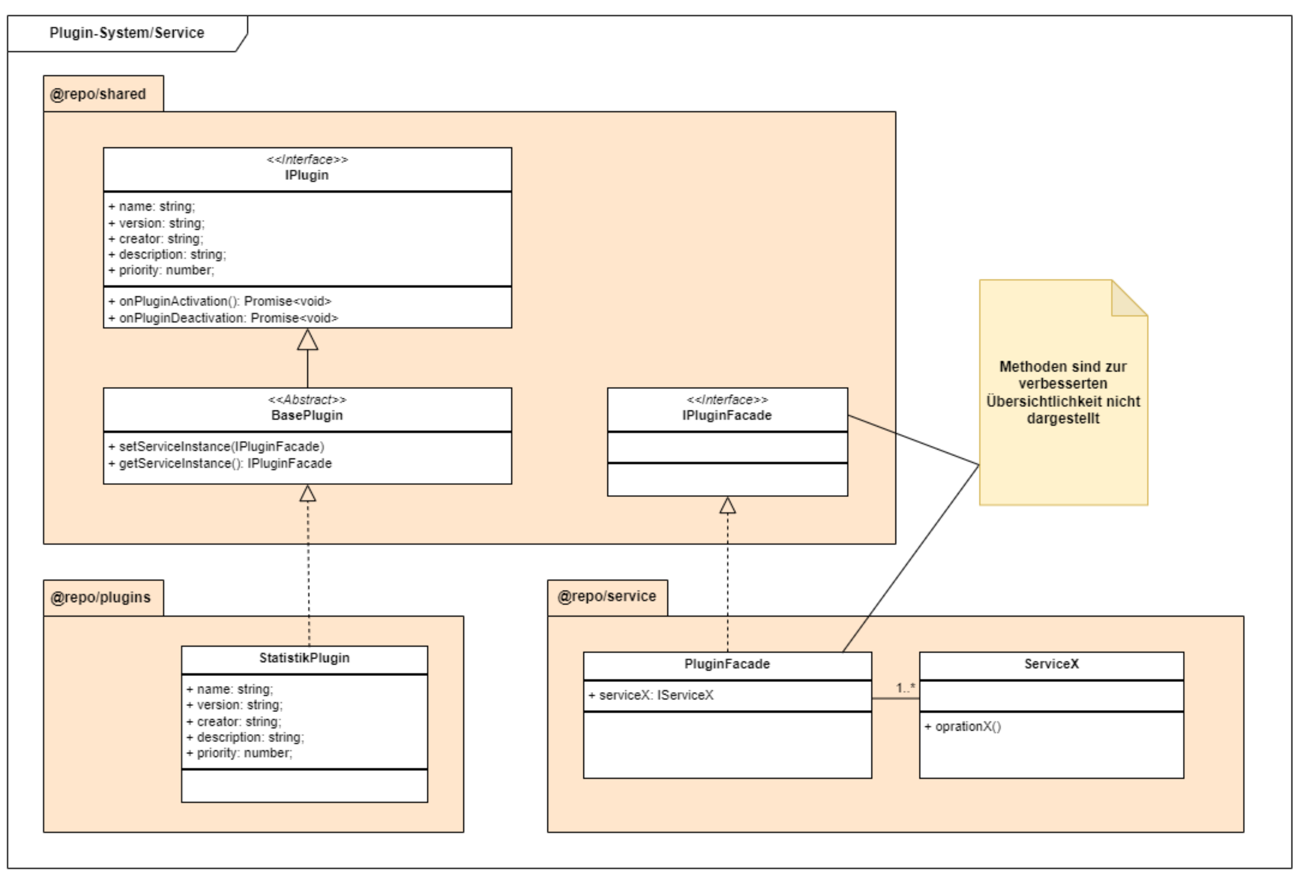


Abbildung 5.20: Plugin: Services konsumieren (UML-Klassendiagramm)

Ein Plugin muss lediglich die abstrakte Klasse *BasePlugin* erweitern, falls die statische Methode zur Abholung der Service-Fassade benötigt wird. Anderenfalls reicht eine direkte Implementation vom Interface *IPlugin*.

5.7.5 Navigationselement einfügen

Ein Plugin bietet die Möglichkeit, eigene Einträge in der Navigation des Storefrontsystems zu erstellen. Dies ist besonders in Kombination mit dem im Abschnitt 5.7.3 dokumentierten Konzept zur Erstellung eigener Seitenelemente von Bedeutung. Durch einen Navigationspunkt wird das Seitenelement des Plugins für den Nutzer leicht auffindbar gemacht. Das Storefrontsystem verfügt über drei verschiedene Navigationsbereiche, in denen ein Plugin eigene Einträge generieren kann:

- **Produktklassierung:** Hauptnavigation im Webshop, basierend auf der im Abacus ERP definierten Produktklassierung.

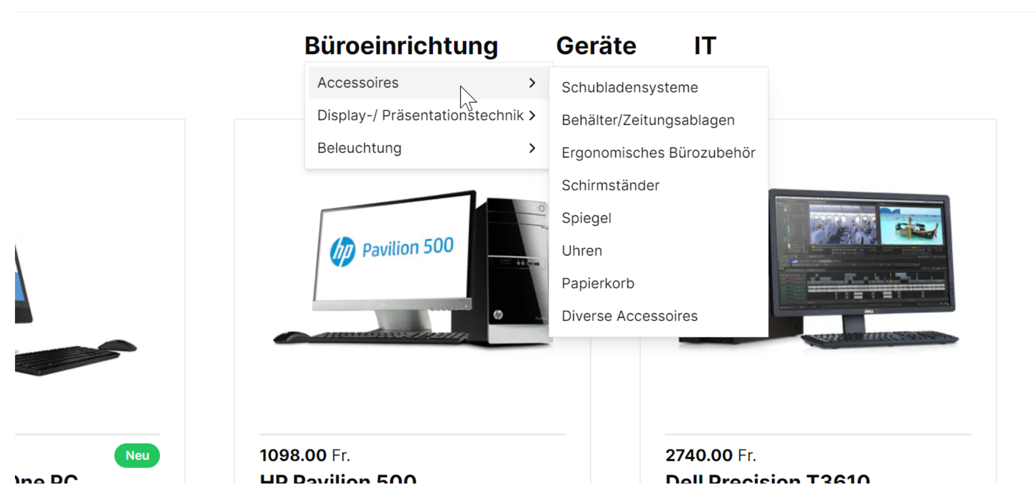


Abbildung 5.21: Navigation der Produktklassierung

- **Kundenprofil:** Navigation im Benutzerprofil des Shoppers.

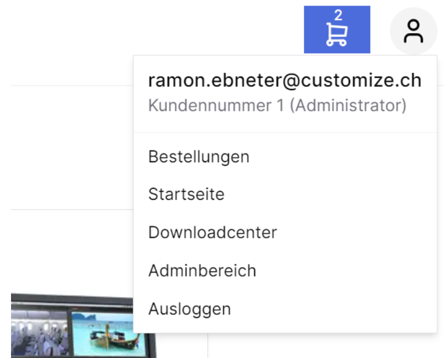


Abbildung 5.22: Navigation im Shopperprofil

- **Administrationsbereich:** Navigation im Adminbereich, welcher ausschliesslich für die Administratoren des Storefrontsystems zugänglich ist.

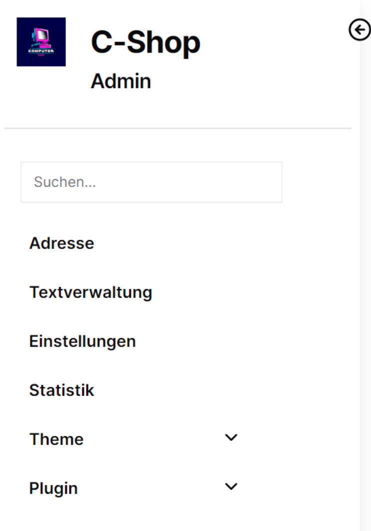


Abbildung 5.23: Navigation im Adminbereich

Für jeden dieser Navigationsbereiche steht ein Interface zur Verfügung, über das ein Plugin entsprechende Navigationspunkte einfügen kann. Ein Beispiel für die Implementierung in der Kundenprofil-Navigation wird in Listing 5.22 dargestellt.

```
export interface IShopProfilePlugin extends IPlugin {
  insertProfileNavigation( navigationTree: ProfileNavigationItem[])
        : ProfileNavigationItem[];
}
```

Listing 5.22: Plugin Interface Kundenprofil-Navigation

Das Kernsystem initiiert bei der Navigationserstellung den Aufruf aller Plugins, die das relevante Interface implementieren. Die jeweilige Methode im Plugin empfängt den Navigationsbaum, fügt das eigene Navigationselement an der gewünschten Stelle ein und gibt den aktualisierten Navigationsbaum zurück. Im Interface *IPlugin* ist die Instanzvariable *priority* festgelegt, welche es einem Plugin erlaubt, seine Priorität zwischen 1 und 100 zu bestimmen. Diese Priorität entscheidet über die Reihenfolge, in der die Plugins vom Kernsystem aufgerufen werden.

In Abbildung 5.24 wird anhand eines UML-Klassendiagramm dargestellt, wie im Storefrontsystem die Navigation am Beispiel des Adminbereichs mit Einbezug der Plugins aufgebaut wird.

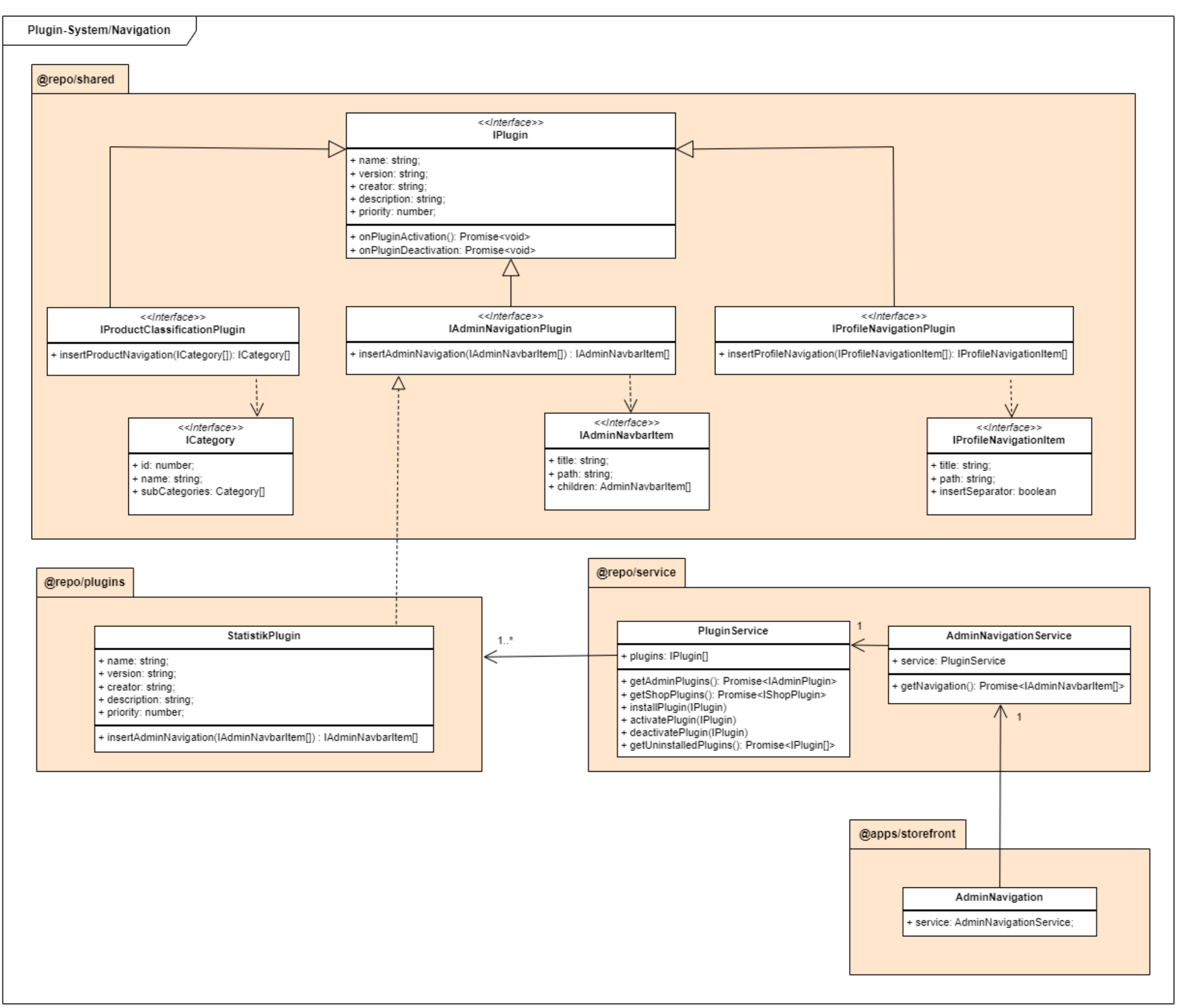


Abbildung 5.24: Plugin: Navigationselement einfügen (UML-Klassendiagramm)

5.7.6 Event-System

Pluginentwickler haben eine Möglichkeit, über spezifische Ereignisse innerhalb des Kernsystems informiert zu werden und darauf reagieren zu können. Zum Beispiel benötigt das Statistikplugin eine Benachrichtigung, sobald ein Kunde die Detailansicht eines Artikels öffnet, damit es die Statistiken aktualisieren kann. Ebenso erfordert das Promotionsplugin eine Information, sobald ein Artikel in den Warenkorb gelegt wird, um entscheiden zu können, ob ein Promotionsartikel zusätzlich in den Warenkorb aufgenommen wird.

Die Implementierung des Event-Systems basiert auf dem Publish-Subscriber-Pattern, das von Hohpe und Woolf vorgestellt wurde [21]. Dieses Pattern ermöglicht eine lose Kopplung zwischen den Komponenten eines Systems und zeichnet sich dadurch aus, dass der Sender (Publisher) der Nachrichten nicht wissen muss, welche oder wie viele Empfänger (Subscriber) es gibt. Diese Entkopplung wird dadurch erreicht, dass der Publisher seine Nachrichten an einen gemeinsamen Nachrichtenkanal sendet, ohne direkt auf die Subscriber einzugehen. Die Subscriber registrieren sich für diesen Kanal und empfangen Nachrichten, die auf ihre spezifischen Ereignisse oder Nachrichtentypen zugeschnitten sind.

Abbildung 5.25 veranschaulicht schematisch die Funktionsweise des Event-Systems im Plugin-System, basierend auf dem Publish-Subscriber-Pattern.

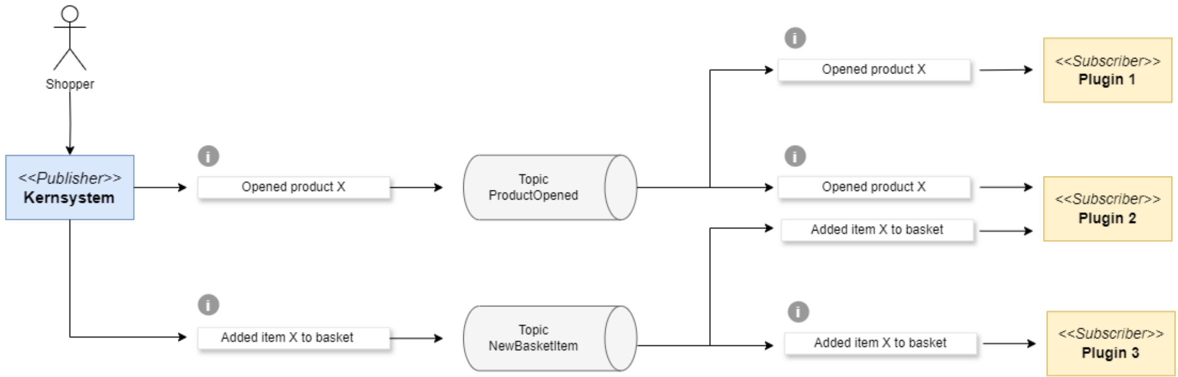


Abbildung 5.25: Event-System mit dem Publish-Subscriber Pattern

Die Plugins registrieren sich nicht selbstständig für die entsprechenden Kanäle, sondern implementieren ein generisches Interface namens *Subscriber*, das in Listing 5.23 gezeigt wird. Es ist nicht vorgesehen, dass ein Plugin selbst als Publisher fungiert, da Events immer vom Kernsystem ausgelöst werden. Da jedoch ein Plugin über die in Abschnitt 5.7.4 beschriebene Fassade auf die Services des Kernsystems zugreift, löst es indirekt auch Events aus.

```
export interface Subscriber<T extends BaseEvent> {
  update: (eventData: T) => void;
}
```

Listing 5.23: Subscriber Interface Event-System

Der *PluginService* registriert die aktivierten Plugins gemäss der implementierten Interfaces auf die entsprechenden Kanäle. Bei jedem Event ruft der Publisher die Methode *update* aller Subscriber auf.

Zum Zeitpunkt dieser Arbeit stehen folgende Events zur Verfügung:

- **NewBasketItem:** Ein Kunde fügt einen Artikel zu seinem Warenkorb hinzu.
- **NewOrder:** Ein Kunde tätigt eine Bestellung.
- **OpenedProduct:** Produkt X wird in der Detailansicht betrachtet.
- **OpenedClassification:** Produktklassifizierung X wird zur Navigation verwendet.

In Abbildung 5.26 ist das UML-Klassendiagramm des implementierten Event-Systems dargestellt. Die Implementierung ist generisch gehalten, sodass jederzeit neue Events hinzugefügt werden können. Voraussetzung ist, dass jeder Event das Basis-Interface *BaseEvent* implementieren muss.

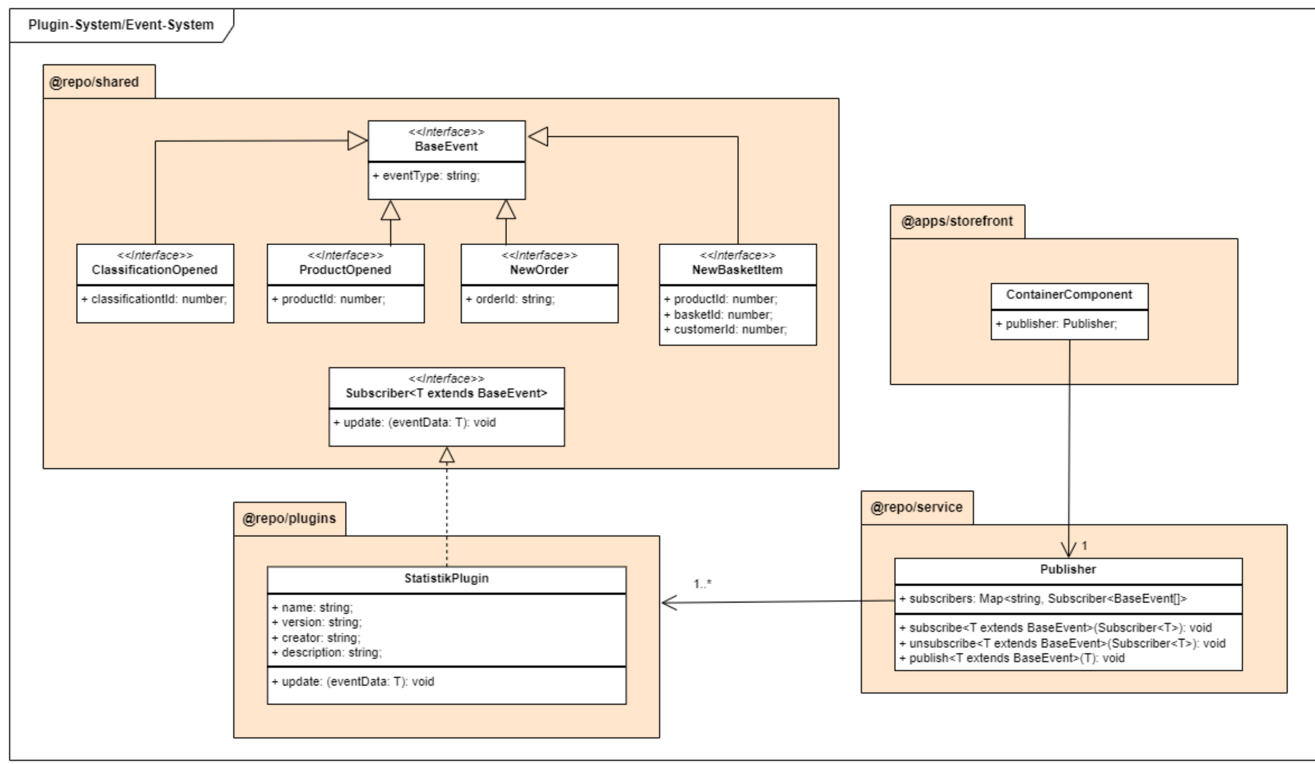


Abbildung 5.26: Plugin: Event-System (UML-Klassendiagramm)

5.7.7 Daten persistieren

Im aufgebauten Monorepo existiert das Paket *@repo/database*, welches dem Service-Paket durch Verwendung eines Prisma-Clients die Interaktion mit der Datenbank ermöglicht. Prisma ist ein ORM (Object-Relational Mapper)[24] in TypeScript, das im gesamten Storefrontsystem eingesetzt wird. Es verwendet eine spezifische Modellierungssprache, die in einer Schema-Datei geschrieben wird und daraus sowohl das Datenbankschema als auch die TypeScript-Typen generiert werden.
Damit Pluginentwickler fortgeschrittene Funktionen implementieren können, benötigen sie eine Möglichkeit, pluginspezifische Daten zu persistieren. Obwohl die Persistierung direkt im Plugin programmiert werden könnte, bietet das Storefrontsystem durch das Paket *@repo/database* bereits die notwendige Infrastruktur, die auch von Plugins genutzt werden kann. Dies verhindert, dass Kunden mehrere Datenbanken verwalten müssen. Beispielsweise persistiert das Statistikplugin jeden Aufruf eines Produkts, um eine entsprechende Auswertung der Produktansichten zu ermöglichen.

Im Service-Layer steht die Klasse *SettingsService* zur Verfügung, die es dem Kernsystem ermöglicht, verschiedene Einstellungen als Key-Value-Paar zu speichern. Eine einfache Möglichkeit für ein Plugin, seine Einstellungen zu persistieren, besteht in der Nutzung dieses Services, wie in Abschnitt 5.7.4 beschrieben. Die Service-Fassade sorgt dabei dafür, dass keine Kernsystemeinstellungen überschrieben werden, indem jede Plugin-Einstellung einen spezifischen Präfix erhält. Dieser Service eignet sich jedoch nur für Einstellungen, da Plugins mit komplexen Daten ganze Datenbanktabellen benötigen.

Zum Zeitpunkt dieser Arbeit fehlen Prisma zwei wichtige Funktionen, um die Erstellung von Datenbanktabellen durch Plugins zu ermöglichen, ohne Änderungen an den Kernpaketen vornehmen zu müssen. Erstens ist es nicht möglich, den Prisma-Client aus mehreren Schema-Dateien zu generieren [41], was nötig wäre, damit jedes Plugin seine eigene Schema-Datei definieren kann. Zum Abschluss dieser Arbeit wurde von Prisma ein Pull-Request gestellt, der diese Funktionalität ermöglichen soll und bald verfügbar sein könnte. Zweitens fehlt die Möglichkeit, Datenbankmigrationen per Code auszuführen [40], was erforderlich ist, um beim Installieren eines Plugins die entsprechenden Tabellen in der Datenbank zu erstellen. Derzeit muss die Migration mit den Commandline-Tools von Prisma durchgeführt werden, was einen neuen Build der Applikation erfordert.

Die Datenpersistierungsfähigkeit in Plugins sollte in der Weiterentwicklung des Storefrontsystems verbessert werden. Die wichtigste Verbesserung würde durch die Möglichkeit der Aufteilung des Prisma-Schemas in mehrere Dateien erreicht, wodurch jedes Plugin sein eigenes Schema definieren und dieses zur Buildzeit[25] vom Kernsystem in die Datenbank migriert werden könnte.

[24]Ein Object-Relational Mapping (ORM) ist eine Technik in der Softwareentwicklung, die es ermöglicht, Daten zwischen objektorientierten Programmiersprachen und relationalen Datenbanken zu konvertieren, indem Datenbanktabellen als Klassen und Datensätze als Objekte dargestellt werden.

[25]Buildzeit im Lebenszyklus von Software bezieht sich auf den Zeitpunkt, an dem der Source-Code eines Projekts kompiliert und zu einer ausführbaren Anwendung zusammengefügt wird.

5.7.8 Export der Plugins

Damit die Plugins für das Paket *@repo/service* zur Verfügung stehen, erfolgt die Bereitstellung nach demselben Prinzip wie im Theming-System, beschrieben in Abschnitt 5.6.6. In der Datei *index.ts* wird ein Array von implementierten Plugins des Typs *IPlugin* exportiert.

5.8 Architektur des Storefrontsystems

Nachdem in den vorangegangenen Abschnitten die beiden zentralen Konzepte, das Theming-System und das Plugin-System, vorgestellt wurden, wird in diesem Abschnitt die gesamte Architektur des in dieser Bachelorarbeit entwickelten Prototyps erläutert. Abbildung 5.27 zeigt alle Komponenten des Storefrontsystems in einem C4-Komponentendiagramm [4].

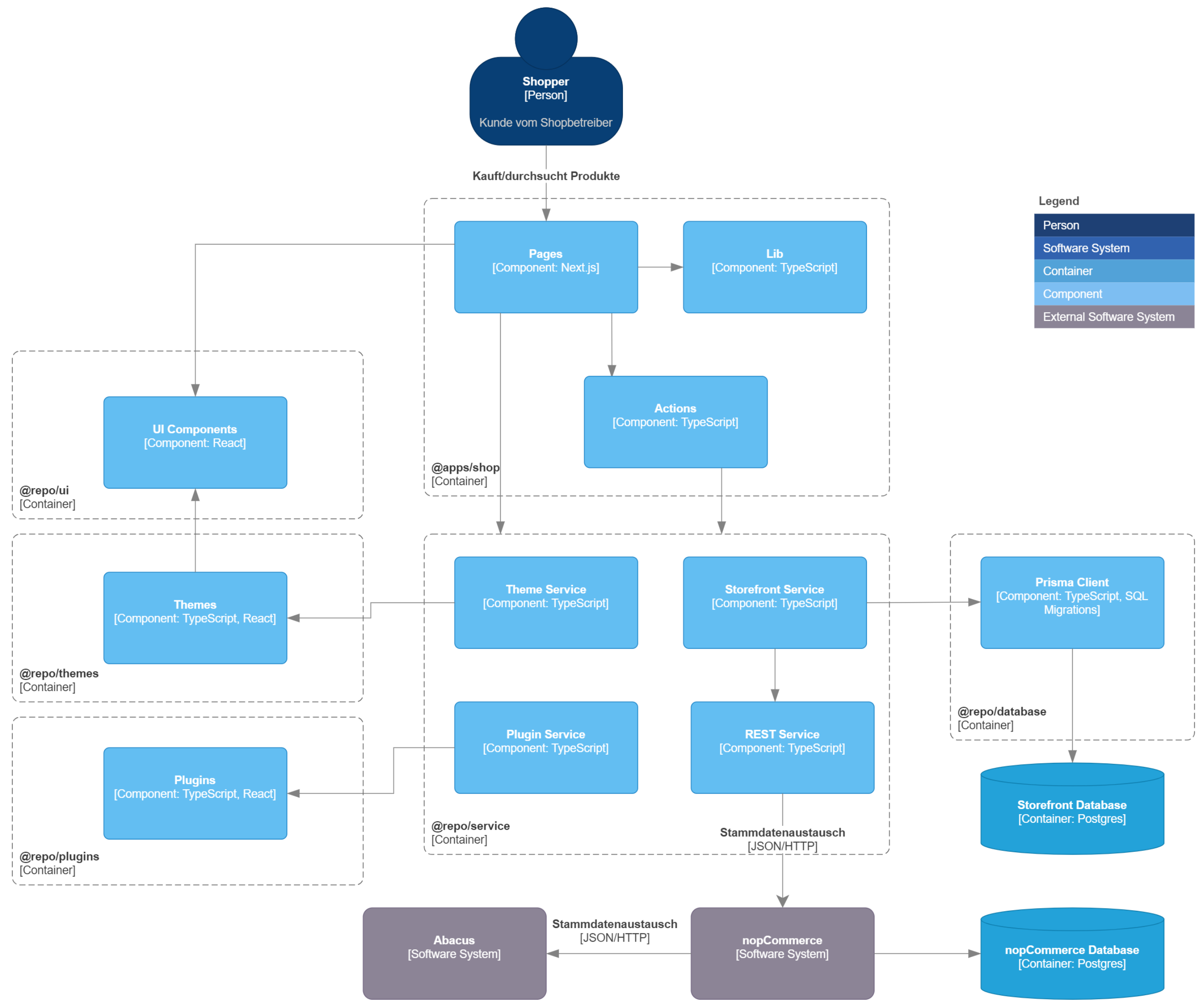


Abbildung 5.27: Komponentendiagramm des Storefrontsystem (C4)

Die Referenz auf die Komponenten des Pakets *@repo/service* im Kernsystem *@apps/shop* ist nicht direkt angegeben, da das Kernsystem alle Komponenten dieses Pakets nutzen kann. Es wird auch nicht explizit modelliert, dass im Paket *@repo/plugins* ein Teil der Komponente *Storefront Service* enthalten ist, da keine direkte Abhängigkeit besteht. Entsprechend der Beschreibung in Abschnitt 5.7.4 übergibt die Komponente *Plugin Service* bei der Instanzierung der Plugins ein Fassadenobjekt mit den Services, die den Plugins zur Verfügung stehen.

5.9 Migration und Entwicklung

Nachdem die vorangegangenen Abschnitte dieses Kapitels die wichtigsten Konzepte der Umsetzung sowie die Architektur des gesamten Systems aufgezeigt haben, beschreibt dieser Abschnitt ein mögliches Migrationsverfahren für Kunden, die von 'AbaShop' auf das neue System umsteigen möchten.

Im Rahmen dieser Bachelorarbeit wurde eine Entwickler-Dokumentation erstellt, die als MDX-basierte[26] Applikation gehostet werden kann. Diese Dokumentations-Applikation, die technische Inhalte wie die Erstellung lokaler Builds, das Deployment sowie Anpassungen im Theming- und Plugin-System behandelt, wird als eigenständige Software bereitgestellt und ist nicht Teil eines spezifischen Kundensystems. Die Inhalte zielen darauf ab, eine Wissensdatenbank für Consultants aufzubauen. Einige der in diesem Abschnitt behandelten Themen werden in der Dokumentation Schritt für Schritt erläutert, weshalb im Anhang B ausgewählte Auszüge der Dokumentation aufgeführt sind.

5.9.1 Abacus Schnittstellen Benutzer

Für den Abacus Connector muss zunächst im Abacus ein Schnittstellenbenutzer erstellt werden, der dann in der Konfiguration des Abacus Connectors hinterlegt wird. Da Abacus die Anfragen prüft und dieser Prozess bis zu zwei Wochen in Anspruch nehmen kann, sollte dies der erste Schritt sein. Der Benutzer kann im Abacus ERP-Programm *Q910 - Abacus API Anbindungen* erstellt werden. Die Anmeldeart ist *benutzerunabhängig (OAuth 2 Grant Type 'Client Credentials')*. Jeder Schnittstellenbenutzer erhält seine Berechtigungen durch definierte Scopes. Für den Abacus Connector werden die folgenden Scopes benötigt:

- CRM: Subjekte lesen und schreiben (abacus.entity.subject.readwrite)
- Auftragsbearbeitung: Lagerbestände lesen (abacus.entity.stock.read)
- Auftragsbearbeitung: Produkt Preise lesen (abacus.entity.productpricing.read)
- Auftragsbearbeitung: Produkte Klassierung lesen (abacus.entity.productclassification.read)
- Auftragsbearbeitung: Produkte lesen (abacus.entity.product.read)
- Auftragsbearbeitung: Varianten Dimensionen lesen (abacus.entity.variantdimension.read)
- E-Business: Inbox schreiben (abacus.entity.ebusinessinbox.write)
- E-Business: Shopper lesen und schreiben (abacus.entity.shopperaccount.readwrite)
- Toolkit: Dossiers lesen und schreiben (abacus.entity.storage.readwrite)
- Toolkit: Wechselkurse lesen (abacus.entity.exchangerate.read)

Basierend auf dieser Konfiguration wird eine JSON-Datei generiert, die an Abacus zur Freigabe der Schnittstelle eingereicht werden muss.

[26] MDX ist eine Erweiterung der Markdown-Syntax, die es ermöglicht, JSX-Komponenten direkt in Markdown-Dateien zu verwenden. Dies erleichtert die Integration von dynamischem React-Code in statische Textinhalte, wodurch Entwickler komplexe Inhalte und interaktive Funktionen nahtlos in Dokumentationen einbetten können.

5.9.2 Kundenumgebung aufsetzen

Für die Verwaltung der verschiedenen Kundeninstanzen wird innerhalb des GitLab des Industriepartners eine Projekt-Gruppe namens *C-Shop* erstellt. In dieser Gruppe befinden sich im Root-Verzeichnis die beiden relevanten Repositories: *nopCommerce*, welches den geforkten Core-Code von nopCommerce sowie die beiden Plugins Abacus Connector und Webapi enthält, und *Storefront*, das entwickelte Storefrontsystem. Für jeden Kunden wird eine Subgruppe angelegt, die alle kundenspezifischen Repositories umfasst. Der Consultant erstellt für den zu migrierenden Kunden eine solche Subgruppe und forkt das Storefront-Repository im benötigten Versions-Branch in die Kunden-Subgruppe. Abbildung 5.28 veranschaulicht die gesamte Projektstruktur im GitLab mit zwei exemplarischen Kunden.

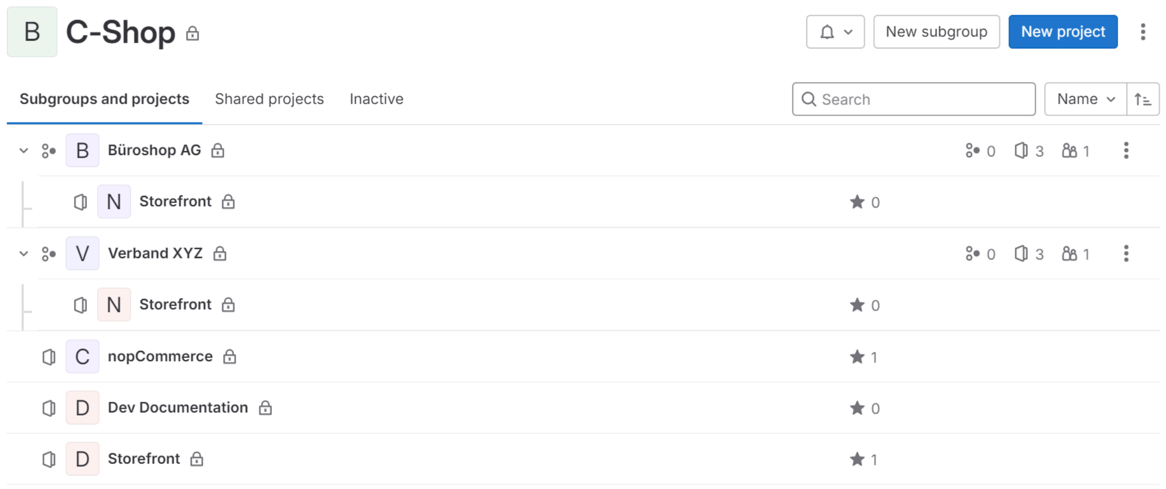


Abbildung 5.28: Projektstruktur GitLab

5.9.3 Deplyoment und Konfiguration

Bei der Implementierung des Deployments besteht die Herausforderung, dass die Kunden des Industriepartners unterschiedliche Anforderungen an das Hosting stellen und einige den Webshop in ihrer eigenen Infrastruktur hosten möchten. In diesen Fällen muss das Deployment projektspezifisch eingerichtet werden. Um diesen Prozess zu vereinfachen, sind alle Komponenten (Storefrontsystem, nopCommerce und Datenbank) als Docker-Container vorbereitet. Für Kunden, die das Hosting an Customize AG auslagern möchten, ist ein standardisiertes Deployment mittels GitLab Actions vorbereitet. Eine detaillierte Anleitung der benötigten Arbeitsschritte ist im Anhang B dokumentiert.

Sobald das Deployment eingerichtet ist und alle Systeme miteinander kommunizieren können, beginnt der umfangreichste Teil der Migrationsarbeiten. Der Consultant kann nun das Storefrontsystem durch die zur Verfügung gestellten Möglichkeiten im Theming- und Plugin-System sowie über Konfigurationen im Adminbereich an die Bedürfnisse des Kunden anpassen. Da die alte Lösung, 'AbaShop', sich technologisch vollständig vom neuen Storefrontsystem unterscheidet, sind keine automatischen Migrationspfade verfügbar.

5.9.4 Logins

In der aktuellen Lösung 'AbaShop' sind alle Passwort-Hashes in einer Datenbank des Softwareherstellers Abacus gespeichert, wobei der verwendete Hashing-Algorithmus unbekannt ist. Wenn die Datenbank zugänglich und der Hashing-Algorithmus bekannt ist, ermöglicht dies eine rollende Migration. Bei diesem Vorgehen loggt sich ein Shopper im neuen Storefrontsystem ein, woraufhin der Hash des Passworts mit dem alten Algorithmus berechnet und mit der Abacus-Datenbank abgeglichen wird. Bei einem erfolgreichen Abgleich legt der Service-Layer von nopCommerce den Benutzer an, der dann bei zukünftigen Logins über nopCommerce verifiziert wird. Nach Absprache mit dem Industriepartner ist derzeit keine Migration der Logins gewünscht. Es ist jedoch möglich, dass ein Kunde spezifische Anforderungen stellt, die dann nach dem beschriebenen Verfahren umgesetzt werden, vorausgesetzt, dass die Datenbank und der Hashing-Algorithmus der alten Lösung verfügbar sind.

Eine Alternative für das Go-Live des neuen Systems ist das Versenden eines Mailings an alle Kunden, um sie zur Erstellung eines neuen Accounts aufzufordern. Dieses Verfahren erfordert lediglich die E-Mail-Adressen aller Shopper, die über ein Reporting aus dem Abacus ERP generiert werden können. Im Webapi Plugin gibt es einen *register* Endpoint, der für diese Aktion verwendet werden kann.

5.9.5 Stammdaten

Die Migration der Stammdaten wird durch den Abacus Connector sichergestellt. Das Plugin bietet einen Synchronisationsmechanismus, der zwischen Voll- und Teilpublikation unterscheidet. Bei der initialen Aktivierung des Plugins erfolgt eine Vollpublikation, wobei der gesamte Datenbestand aus dem Abacus ERP übernommen wird. Die nachfolgenden Intervalle umfassen lediglich eine Teilpublikation, die alle seit der letzten Synchronisation erfolgten Änderungen erfasst.

5.9.6 Entwicklungsprozess

Aufgrund der Erfahrungen mit 'AbaShop' werden vor allem das Theme, aber auch die Plugins, unter ständiger Weiterentwicklung stehen, da die Kunden des Industriepartners regelmässig neue Wünsche und Verbesserungen anbringen. Es ist daher wichtig, klar zu definieren, wie der Entwicklungsprozess in diesem Bereich vorgesehen ist, damit dieser von allen Consultants identisch praktiziert wird.
In Abschnitt 5.9.2 wird aufgezeigt, dass ein Kunde eine eigene Subgruppe innerhalb des gesamten Projekts in GitLab erhält, in der ein Fork des Storefrontsystem-Repositorys erstellt wird. Möchte ein Consultant eine Anpassung des Themes vornehmen, erstellt er einen neuen Feature-Branch[27] im Kundenprojekt. Durch den aufgezeigten Aufbau des Storefrontsystems als Monorepo hat der Consultant eine einfache Möglichkeit, direkt Feedback auf seine Theme-Anpassungen zu bekommen. Das Projekt kann lokal mit `pnpm dev` ausgeführt werden. Durch den integrierten Fast Refresh Mechanismus[28] in Next.js sieht

[27]Eine Feature-Branch ist ein isolierter Entwicklungszweig in einem Versionskontrollsystem, der verwendet wird, um neue Funktionen oder Änderungen unabhängig vom Hauptentwicklungszweig zu entwickeln, bis sie bereit zur Integration sind.

[28]Fast Refresh von Next.js ist ein Entwicklungswerkzeug, das es Entwicklern ermöglicht, Änderungen am Code in Echtzeit zu sehen, ohne den aktuellen Anwendungszustand zu verlieren.

der Consultant seine getätigten Änderungen im lokalen Build direkt. Abhängig von der Komplexität des Kunden wird nach dem gestellten Merge-Request ein Review durch einen anderen Consultant durchgeführt. Nach Durchführung des Merges in den Main-Branch wird durch die Pipeline ein neuer Production-Build angestossen, sodass die getätigten Änderungen am Theme publiziert werden.

Im Rahmen des durchgeführten Benutzertests wurden an diesem Punkt noch Optimierungspotenziale entdeckt, die in Abschnitt 6.4.1 diskutiert werden. Diese betreffen sowohl die Entwicklung von Themes und Plugins als auch das Vorgehen bei der Weiterentwicklung des Kernsystems und dessen Verteilung auf vorhandene Systeme.

Kapitel 6

Ergebnisdiskussion

Das letzte Kapitel fasst die Ergebnisse dieser Bachelorarbeit zusammen und bewertet sie bezüglich der funktionalen und nichtfunktionalen Anforderungen. Weiterhin diskutiert es die aus dieser Arbeit gezogenen Schlussfolgerungen, welche für das weitere Vorgehen im Projekt 'Ablösung 'AbaShop" zusammen mit dem entwickelten Prototypen von Bedeutung sind.

6.1 Kritische Erfolgsfaktoren

Die Aufgabenstellung (siehe Anhang C) definiert die kritischen Erfolgsfaktoren dieser Bachelorarbeit. Diese Faktoren spielten eine entscheidende Rolle bei der Festlegung der funktionalen und nichtfunktionalen Anforderungen. Nachstehend sind diese kritischen Erfolgsfaktoren erneut aufgeführt.

1. Der Prototyp wird auf Basis der Anforderungen eines realen oder fiktiven (aber repräsentativen) Customize-Kunden validiert.
2. Die Erweiterungsmöglichkeiten des Shopsystems werden anhand eines solchen realen 'Beispiel'-Shops demonstriert.
3. Bei der Entwicklung des Prototypen wird auf die Benutzbarkeit geachtet; diese wird durch einen 'User Test' (mit einem Consultant von Customize und/oder einem Endkunden) validiert und notwendige Verbesserungen für die Zukunft dokumentiert.
4. Dem Industriepartner wird eine Benutzerdokumentation zur Verfügung gestellt, mit welcher Mitarbeiter (Consultants) in der Lage sind, einen Shop für einen Kunden aufzusetzen.
5. Ausserdem wird auf die Erweiterbarkeit und Wartbarkeit des Systems geachtet. Bei der Entwicklung werden industrieübliche Prozesse eingesetzt und die an der OST gelernten Software Engineering Hygienefaktoren berücksichtigt (automatisierte Builds/Deployments, Tests, angemessene Versionierung mit Git, etc.).

In Tabelle 6.1 wird die Bewertung der kritischen Erfolgsfaktoren dargelegt.

Nr.	Bewertung
1	Abschnitt 2.3 führt den Musterkunden 'Sportartikel AG' ein. Dieser repräsentiert einen anonymisierten, realen Kunden, der einen umfassenden 'AbaShop' einsetzt. Die Anforderungserfassung basiert auf den Bedürfnissen dieses Kunden. Zusätzlich wurden Anforderungen von fünf weiteren Kunden aufgenommen und aus der Perspektive des Musterkunden dokumentiert. Die Bewertung der im Prototyp des Storefrontsystems implementierten funktionalen Anforderungen wird in Abschnitt 6.2 weitergeführt.
2	Die Möglichkeiten des Theming- und Plugin-System wird durch verschiedene Beispiel-Plugins und -Themes demonstriert. Die drei implementierten Plugins, welche auf Anforderungen von realen Kunden basieren, werden in Abschnitt 5.7.1 beschrieben. Die Bewertung der funktionalen Anforderungen an das Plugin-System findet in Abschnitt 6.2.4 statt, welche die Abdeckung durch die Beispiel-Plugins darlegt. Die entwickelten Themes 'Standardtheme' und 'Moderntheme' demonstrieren umfassend die Möglichkeiten des Theming-Systems.
3	Der durchgeführte Benutzertest ist in Abschnitt A dokumentiert. Die daraus identifizierten Optimierungspotenziale wurden in den Prototyp integriert. Details dieser Optimierungen finden sich in Abschnitt 6.4.
4	Eine umfangreiche Entwicklerdokumentation wurde erstellt, um den Consultants die Einrichtung und Verwaltung des Gesamtsystems, bestehend aus nopCommerce mit den Plugins 'Abacus Connector' und 'Webapi' sowie dem Storefrontsystem, zu erleichtern. Diese Markdown-basierte Dokumentation dient sowohl aktuellen als auch zukünftigen Entwicklern der Plugins und Themes und beinhaltet Anleitungen zur lokalen Einrichtung sowie zu möglichen Azure-Deployment-Szenarien. Sie beschreibt auch ausführlich die Nutzung aller Schnittstellen für die Erstellung von Themes und Plugins. Die Dokumentation findet sich in Anhang B.
5	Das geplante Deployment wird in der Entwicklerdokumentation beschrieben. Mittels der in GitLab integrierten Pipelines werden Tests und Deployments automatisiert durchgeführt. Diese werden dem Industriepartner zusammen mit den abgegebenen Repositories zur Verfügung gestellt.

Tabelle 6.1: Bewertung der kritischen Erfolgsfaktoren

6.2 Bewertung der funktionalen Anforderungen

Die erhobenen funktionalen Anforderungen wurden in vier verschiedene Kategorien unterteilt. Abschnitt 3.3 definiert mithilfe von User Stories Anforderungen, die jeder Kunde als Standardfunktion im Storefrontsystem erwartet. Diese sollen als Teil des Kernsystems allen Kunden identisch zur Verfügung stehen. Abschnitt 3.4 dokumentiert die Anforderungen, die ein Consultant für die spezifische Konfiguration eines Kunden durchführen möchte. In den Abschnitten 3.5 und 3.6 werden die Anforderungen an das Theming- und Plugin-System in Form von generisch gehaltenen Use Cases definiert.

Die funktionalen Anforderungen werden in den nachfolgenden Abschnitten anhand des implementierten Prototypen des Storefrontsystems bewertet. Zudem wird aufgezeigt, welche Anforderungen im Anschluss an diese Arbeit noch umgesetzt werden müssen.

6.2.1 Kernfunktionalitäten

Kernfunktionalitäten sind User Stories, die aus der Sicht des Akteurs Shopper definiert wurden. Sie decken die Standardfunktionalitäten eines Webshops ab und stehen allen Kunden als Teil des Kernsystems identisch zur Verfügung. Tabelle 6.2 gibt einen Überblick über den Status der erfassten Kernfunktionalitäten im Prototypen.

User Story	**Priorisierung**	**Bewertung**
US1 - Produktklassierung durchsuchen	Must-Have	Die Anforderung ist umgesetzt. ✔
US2 - Filter anwenden	Must-Have	Die Anforderung ist umgesetzt. ✔
US3 - Produktdetails einsehen	Must-Have	Die Anforderung ist umgesetzt. ✔
US4 - Produkt in den Warenkorb legen	Must-Have	Die Anforderung ist umgesetzt. ✔ Im Kontext dieser User Story wurde zusätzlich eine umfassende User-Verwaltung implementiert.
US5 - Bestellung tätigen	Should-Have	Die Anforderung ist umgesetzt. ✔ Eine Erweiterung mit zusätzlichen Formularfeldern und Zahlungsoptionen ist erforderlich.
US6 - Sprache im Shop ändern	Should-Have	Die Anforderung ist umgesetzt. ✔

Tabelle 6.2: Bewertung der Kernfunktionalitäten

Der Schwerpunkt dieser Arbeit lag in der Implementierung umfangreicher Konfigurationsmöglichkeiten. Dabei müssen die Kernfunktionalitäten nicht zwingend vollständig erfasst und implementiert sein, sondern dienen dazu, die Konfigurationsmöglichkeiten aufzuzeigen. Dennoch wurden diese Anforderungen umfassend in den Prototypen integriert. Um den Musterkunden 'Sportshop AG' in einem Proof of Concept zu migrieren, ist lediglich eine Erweiterung der User Story *US5 - Bestellung tätigen* erforderlich. Derzeit ermöglicht der Bestellprozess nur die Eingabe der Kundenadresse, zukünftig muss eine möglicherweise abweichende Lieferadresse hinzugefügt werden können. Ausserdem benötigt der Kunde eine Möglichkeit zur Kreditkartenzahlung. Dieser Punkt wurde vom Industriepartner ausgeklammert und wird im Nachgang dieser Arbeit angegangen.

6.2.2 Konfiguration

Die Kategorie Konfiguration umfasst User Stories, die Anpassungen des Storefrontsystems über eine Benutzeroberfläche ermöglichen. Damit können Consultants Anpassungen am Kernsystem vornehmen, ohne den Source-Code zu verändern. Diese Funktionen wurden in einem Administrationsbereich des Prototypen implementiert. Die Bewertung dieser Anforderungen wird in der Tabelle 6.3 aufgeführt.

User Story	**Priorisierung**	**Bewertung**
US7 - Farbgebung editieren	Must-Have	Die Anforderung ist umgesetzt. ✔ Abschnitt 5.6.3 dokumentiert, wie die Farbgebung des gesamten Storefrontsystems über den Adminbereich oder mittels eines Themes angepasst werden kann.
US8 - Logo und Stammdaten editieren	Must-Have	Die Anforderung ist umgesetzt. ✔
US9 - Informationstexte verwalten	Could-Have	Die Anforderung ist umgesetzt. ✔ Die integrierte Textverwaltung ermöglicht die Erfassung beliebig vieler formatierter Texte, die über ein Drag-and-Drop-Interface sortiert werden können.
US10 - Filter aktivieren	Must-Have	Die Anforderung ist umgesetzt. ✔
US11 - Filter pro Klassierungselement definieren	Could-Have	Die Anforderung ist nicht umgesetzt. ✗ Im Prototypen können Filter nur gemäss *Filter aktivieren* global definiert werden.

US12 - Klassierungsnavigation parametrieren	Could-Have	Die Anforderung ist nicht umgesetzt. ✗

Tabelle 6.3: Bewertung der Konfigurationsanforderungen

Die beiden User Stories *US11 - Filter pro Klassierungselement definieren* und *US12 - Klassierungsnavigation parametrieren* wurden aus Zeitgründen nicht in den Prototyp integriert. Der Adminbereich bietet jedoch einen zentralen Dienst zur Verwaltung von Einstellungen, was die Umsetzung dieser Anforderungen ohne architekturrelevante Anpassungen ermöglicht. Durch die Nutzung des Theming-Systems können diese Anforderungen bereits im aktuellen Prototyp entwickelt werden, ohne den Source-Code des Kernsystems zu verändern, vorausgesetzt ein Consultant übernimmt die Entwicklung im Theme.

6.2.3 Theming-System

Die beiden Use Cases *UC1 - Komponenten überschreiben* und *UC3 - Komponentenlayout anpassen* sind in das Theming-System integriert und bieten umfassende Gestaltungsmöglichkeiten für das Storefrontsystem über ein Theme. Im durchgeführten Benutzertest (siehe Anhang A) konnte verifiziert werden, dass ein Consultant mithilfe eines Themes die Kundenanforderungen effektiv umsetzen können. Die Anforderung *UC2 - Theme hochladen* wurde nicht umgesetzt, da sich im Verlauf dieser Arbeit herausstellte, dass die meisten Kunden ein eigenes Theme besitzen werden, welches von einem Consultant kontinuierlich weiterentwickelt wird. Dennoch bleibt es interessant, diese Funktion für die Schaffung eines Theme-Marktplatzes bereitzustellen. Die Priorisierung dieser Umsetzung wird jedoch als niedrig eingestuft. Die Möglichkeiten im Theming-System werden durch die Bereitstellung der beiden Beispiele 'Standardtheme' und 'Moderntheme' aufgezeigt.

6.2.4 Plugin-System

Die für das Plugin-System definierten Use Cases sind, mit Ausnahme der Could-Have-Anforderung *UC9 - Daten persistieren*, alle in einem stabilen Zustand im Prototypen vorhanden. Diese Anforderung konnte nicht wie gewünscht implementiert werden, da zum Zeitpunkt dieser Arbeit gemäss Dokumentation in Abschnitt 5.7.7 zwei benötigte Funktionen im genutzten OR-Mapper *Prisma* noch nicht zur Verfügung stehen. Die Umsetzung des Plugin-Systems ist in Abschnitt 5.7 ausführlich dokumentiert. Es wird aufgezeigt, welche Schnittstellen für die Erfüllung der definierten Use Cases zur Verfügung stehen und wie sie genutzt werden können. Um die Anwendung dieser Schnittstellen zu demonstrieren, wurden im Kontext dieser Arbeit verschiedene Beispiel-Plugins entwickelt.

In Tabelle 6.4 wird eine Übersicht aller definierten Use Cases des Plugin-Systems dargestellt. Es wird aufgezeigt, für welche praktischen Einsatzzwecke diese in den Beispiel-Plugins eingesetzt wurden.

User Case	**Plugin**	**Einsatzzweck**
UC4 - Klassierungselement injizieren	Downloadplugin	Das Downloadplugin injiziert einen Eintrag in die Profilnavigation, sodass ein eingeloggter Shopper das Downloadcenter auffinden kann.
UC5 - Seitenelement injizieren	Downloadplugin	Das Downloadplugin injiziert eine eigene Seite im Adminbereich, um die Konfiguration des Plugins bereitzustellen. Eine weitere Seite wird im Webshop eingefügt, um den Shoppern den Zugang zum Downloadcenter zu ermöglichen.
UC6 - Service konsumieren	Promotionsplugin	Das Promotionsplugin nutzt die Service-Fassade, um Promotionsartikel in den Warenkorb zu legen.
UC7 - Service beeinflussen	Promotionsplugin	Um dem Kunden den Promotionsartikel gratis anzubieten, überschreibt das Promotionsplugin die Preisfindung für diesen Artikel.
UC8 - Auf einen Event reagieren	Statistikplugin	Das Statistikplugin reagiert auf den Event *ProductOpened* und erstellt Statistiken über die angesehenen Artikel.
UC9 - Daten persistieren	-	Dieser Use Case ist nicht implementiert.

Tabelle 6.4: Use Cases des Plugin-Systems

6.3 Bewertung der definierten NFAs

Im Abschnitt 3.8 wurden sechs nichtfunktionale Anforderungen (NFAs) festgelegt, die für die Entwicklung des Prototypen in dieser Bachelorarbeit von Bedeutung sind. Diese NFAs werden in den nachfolgenden Abschnitten im Kontext dieser Arbeit bewertet.

6.3.1 Performance

Dieses NFA spezifiziert eine Performanceobergrenze, welche vom Storefrontsystem beim Laden einer Seite eingehalten werden muss. Hierbei ist festgelegt, dass von dem Moment des Seitenaufrufs bis zur endgültigen Darstellung im Browser eine maximale Ladezeit von zwei Sekunden nicht überschritten werden darf. Inbegriffen ist hierbei der Aufruf des Webapi-Plugins in nopCommerce zur Bereitstellung der Stammdaten. Abbildung 6.1 zeigt die Performanceanalyse durch Google Lighthouse [19].

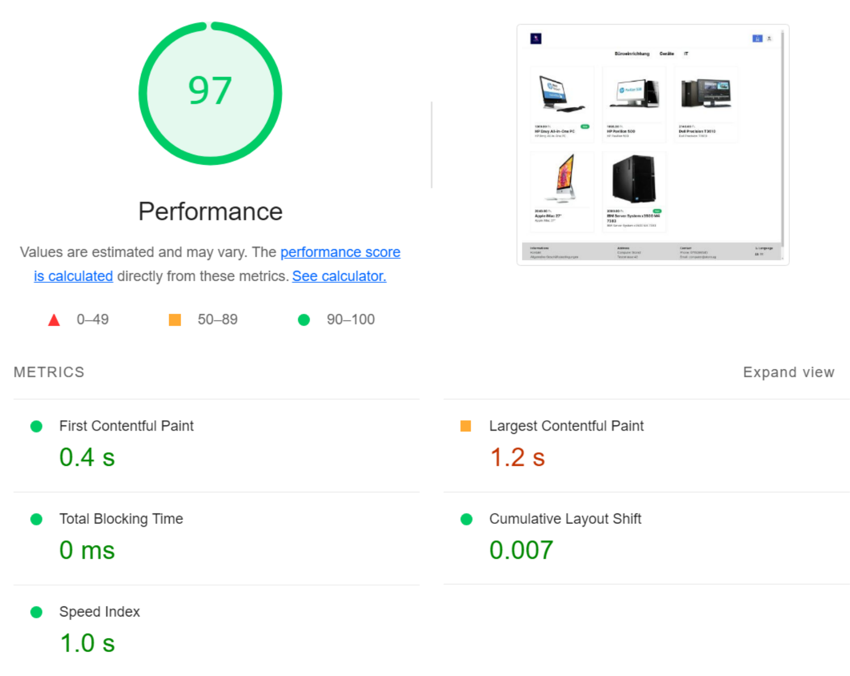


Abbildung 6.1: Performance-Analyse durch Google Lighthouse [19]

Um dieses NFA zu überprüfen, wurde ein Performancetest der Hauptseite des Storefrontsystems durchgeführt. Die Hauptseite wurde ausgewählt, weil sie durch den Abruf der Produktklassierung und Produkte inklusive Bilder die meisten Schnittstellenaufrufe tätigt. Der Performancetest erfolgte mit dem Datenbestand des Abacus-Mustermandanten und umfasste 50 Aufrufe.

Abbildung 6.2 zeigt ein Boxplot [49] des durchgeführten Performancetests.

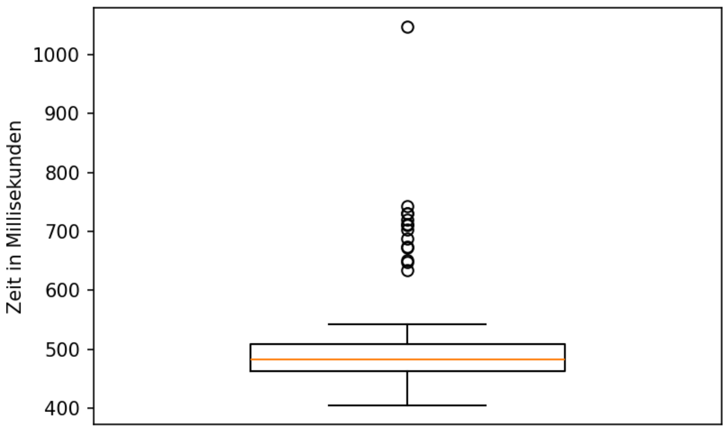


Abbildung 6.2: Boxplot des Performancetests vom Storefrontsystem

Es ist ersichtlich, dass keiner der Aufrufe den festgelegten Schwellenwert von zwei Sekunden überschreitet. Der Ausreisser nach oben entspricht dem ersten Aufruf. Das Storefrontsystem nutzt die serverseitigen Caching-Möglichkeiten, die Next.js bietet, was den nachfolgenden Aufrufen zugutekommt.

✔ Die NFA ist erfüllt.

6.3.2 Zukunftssicherheit

Um die geforderte Lebensdauer von mindestens sechs Jahren zu gewährleisten, wurde dieses NFA in den Designentscheidungen berücksichtigt (siehe Abschnitte 5.1, 5.3 und 5.4). Ebenfalls wurden die Anzahl der Abhängigkeiten (npm) im Kernsystem des Storefronts gering gehalten. Herausforderungen ergaben sich beim Umgang mit npm-Abhängigkeiten in den Themes und Plugins. Da es den Consultants frei steht, npm-Abhängigkeiten in ihren Plugin-Projekten zu verwenden, besteht das Risiko, dass dieses NFA nicht eingehalten wird. Es ist erforderlich, dass zukünftig Richtlinien für den Umgang mit diesen Abhängigkeiten definiert werden. Da diese Richtlinien noch nicht existieren, wird dieses NFA als teilweise erfüllt angesehen.

? Die NFA ist teilweise erfüllt.

6.3.3 Einfachheit eines Themes

Das Theming-System, ein zentrales Werkzeug für Consultants, wird erfahrungsgemäss kontinuierlich für den Kunden weiterentwickelt, weshalb dieses NFA definiert wurde. Die Validierung erfolgte durch einen Benutzertest (siehe Anhang A). Die gesetzte Zeitgrenze für eine einfache Anpassung eines Themes von 10 Minuten wurde eingehalten, obwohl der Consultant das Theming-System an diesem Tag erstmalig verwendete. Das Feedback vom Industriepartner bestätigt zudem, dass das Theming-System sehr intuitiv zu bedienen ist (Zwischenpräsentation vom 16. April 2024).

✔ Die NFA ist erfüllt.

6.3.4 Search Engine Optimierung

Die Erfüllung der Anforderung zur Suchmaschinenoptimierung wird durch eine Analyse mittels Google Lighthouse [19] überprüft. Das Ergebnis wird in Abbildung 6.3 dargestellt.

SEO

These checks ensure that your page is following basic search engine optimization advice. There are many additional factors Lighthouse does not score here that may affect your search ranking, including performance on Core Web Vitals. Learn more about Google Search Essentials.

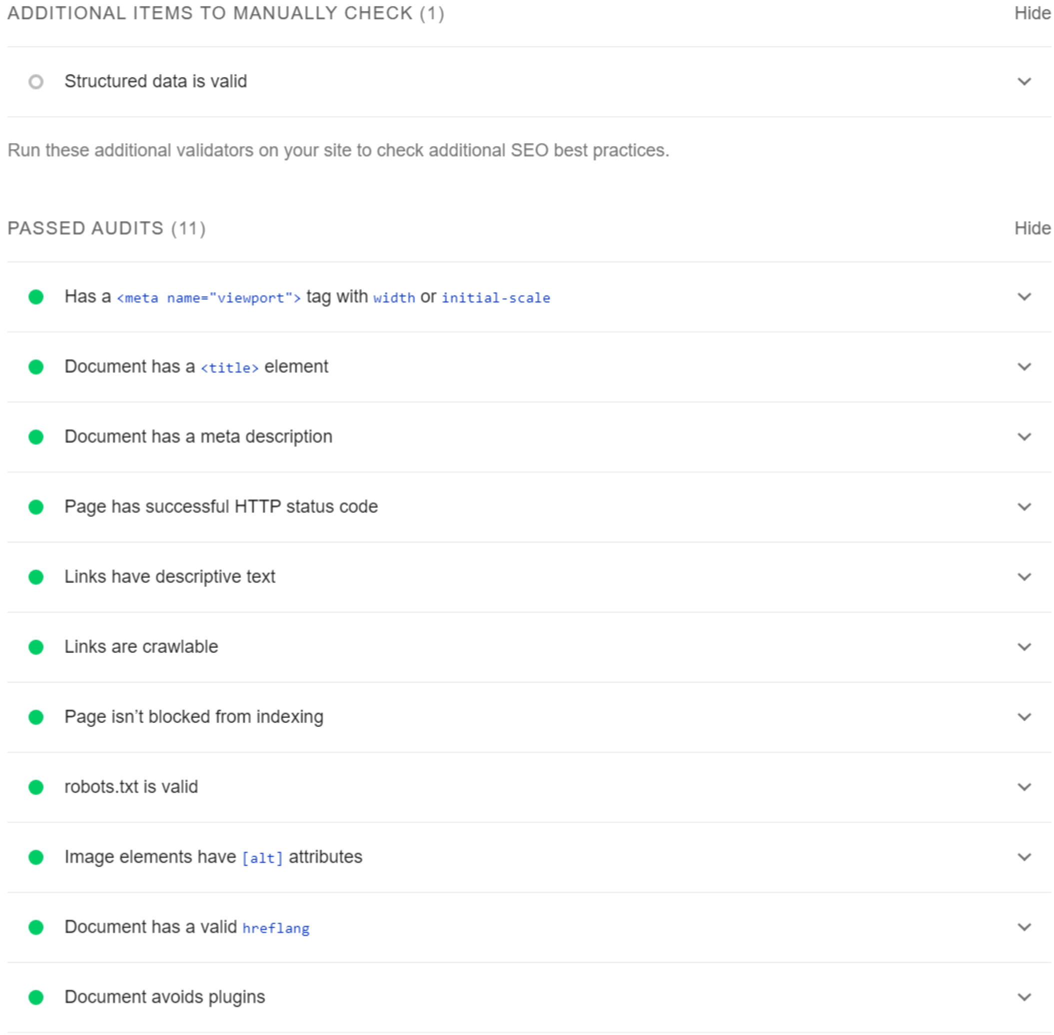


Abbildung 6.3: SEO-Analyse durch Google Lighthouse [19]

Zur Erfüllung dieser NFA wurden mehrere Optimierungen an einem suchmaschinenoptimierten Storefrontsystem durchgeführt:

- **Serverseitiges Rendering:** Alle für Suchmaschinen relevanten Seiten werden serverseitig gerendert.
- **Slug-Optimierung:** Produkte und Klassierungen sind über einen aussagekräftigen Slug anstatt einer numerischen ID aufrufbar (z.B. `https://storefront.ch/products/hp-envy-one-pc` anstelle `https://storefront.ch/products/1`).
- **Meta-Tags:** In nopCommerce können Meta-Tags[1] für Produkte und Klassierungen definiert werden. Diese werden durch das Webapi-Plugin bereitgestellt und vom Storefrontsystem in den entsprechenden HTML-Tags auf allen relevanten Seiten eingebunden.

✔ Die NFA ist erfüllt.

6.3.5 Verfügbarkeit

Durch den Abacus Connector wird sichergestellt, dass das Storefrontsystem auch bei Nichtverfügbarkeit des Abacus ERP funktionsfähig bleibt. Eine Herausforderung waren die kundenspezifischen Preise und Konditionen, da diese nicht in nopCommerce persistiert, sondern bei Bedarf vom Abacus ERP abgefragt werden. Dies hat zur Folge, dass diese Daten nicht verfügbar sind, wenn das Abacus ERP nicht erreichbar ist. Zu diesem Zweck wurde ein 'Preis-Cache' in nopCommerce implementiert, der in Abschnitt 5.2.4 beschrieben ist. Dies ermöglicht dem Storefrontsystem, die kundenspezifischen Preise auch dann abzufragen, wenn das Abacus ERP nicht verfügbar ist.
Die Verifikation dieser NFA erfolgte durch das Abschalten des Abacus ERP und die anschliessende Durchführung manueller Tests im Storefrontsystem.

✔ Die NFA ist erfüllt.

6.3.6 Updatefähigkeit

Die NFA *Updatefähigkeit* beschreibt die Möglichkeit, trotz Umsetzung kundenindividueller Anforderungen ein Kernsystem bereitzustellen, das bei allen Kunden identisch ausgeliefert werden kann. Dies wird durch die Einführung eines Theming- und Plugin-Systems erreicht. Durch die Bereitstellung von zwei Beispiel-Themes und verschiedenen Plugins wird demonstriert, dass die spezifischen Anforderungen ohne Modifikationen am Source-Code des Kernsystems realisiert werden können.

✔ Die NFA ist erfüllt.

[1] Meta-Tags sind HTML-Elemente, die Informationen über eine Webseite bereitstellen, wie etwa deren Inhalt und Schlüsselwörter, und spielen eine wichtige Rolle für die Suchmaschinenoptimierung, indem sie Suchmaschinen dabei helfen, den Inhalt einer Seite besser zu verstehen und zu indexieren.

6.4 Optimierungen

Bei der Implementierung der Beispiel-Plugins sowie im durchgeführten Benutzertest (siehe Anhang A) wurden im Umgang mit dem Theming- und Plugin-System zwei Aspekte identifiziert, die Optimierungspotenzial aufweisen. Diese Punkte betreffen den Entwicklungsprozess der Consultants, die als wichtige Stakeholder intensiv mit dem neuen Storefrontsystem arbeiten werden. Die beiden potenziellen Optimierungen konnten auf einem separaten Branch umgesetzt werden, um dem Industriepartner die vorgeschlagenen Anpassungen zu demonstrieren. In den nachfolgenden zwei Abschnitten werden diese Punkte dokumentiert und die umgesetzten Lösungsvorschläge dargestellt.

6.4.1 Entwicklungsprozess

Im Abschnitt 5.9.6 wird der geplante Entwicklungsprozess für Themes und Plugins beschrieben. Consultants sollen das gesamte Monorepo lokal öffnen, einen neuen Feature-Branch erstellen und darauf im Paket *@package/themes* oder *@package/plugins* die gewünschten Anpassungen vornehmen. Nach einem Merge in den Main-Branch löst die Pipeline ein neues Deployment aus, wodurch die Änderungen live geschaltet werden. Dieser Ansatz weist jedoch zwei bedeutende Nachteile auf, die im Rahmen eines Benutzertests mit dem Consultant diskutiert wurden:

- **Versionierung:** Eine konkrete Versionierung der Themes ist nicht möglich. Der Consultant hat eine Möglichkeit gesucht, dem Theme eine Version zu geben, was im aktuellen Zustand lediglich über die Source-Codeverwaltung von GitLab geschehen könnte. Dabei besteht der Nachteil, dass ein Theme Teil des Monorepos ist und folglich die Versionierung vom Kernsystem mit dem Theme vermischt werden könnte.
- **Schutz des Kernsystems:** Da ein Consultant das gesamte Monorepo auscheckt, kann eine unbeabsichtigte Änderung am Kernsystem nicht ausgeschlossen werden. Daher basiert der Schutz des Kernsystems auf der Eigenverantwortung der Consultants. Änderungen am Kernsystem sind zu vermeiden, um zukünftige Konflikte bei Updates des Storefrontsystems zu verhindern.

Basierend auf diesen Erkenntnissen wurde ein alternativer Entwicklungsprozess diskutiert, der eine Transformation der Pakete *@package/themes* und *@package/plugins* von internen zu auf npm gehosteten Paketen vorsieht, wodurch diese aus dem Monorepo herausgelöst und eigenständig versioniert sowie 'deployed' werden können.

Der Entwicklungsprozess ist für den Industriepartner von grosser Bedeutung. Da die vorgeschlagene Lösung ebenfalls ein Refactoring der Architektur vom Storefrontsystem benötigt, wurde der neue Ansatz auf einem separaten Branch umgesetzt. Die vorgeschlagene Lösung wird im nachfolgenden Abschnitt beschrieben.

Vorschlag alternativer Entwicklungsprozess

Wie zuvor beschrieben, ist es geplant, die Pakete *@package/themes* und *@package/plugins* aus dem Monorepo herauszulösen und auf npm pro Kunde zur Verfügung zustellen. Da diese Pakete von *@package/ui*, das Standardkomponenten bereitstellt, und *@package/shared*, das Typescript-Interfaces für Typendefinitionen enthält, abhängig sind, müssen auch diese auf npm bereitgestellt werden. Im Gegensatz zu den Theme- und Plugin-Paketen werden *@package/ui* und *@package/shared* jeweils nur einmal und nicht für jeden Kunden benötigt. Diese Umstrukturierung führt zu der in Abbildung 6.4 dargestellten neuen GitLab-Struktur mit zwei Beispielkunden. Das Paket *@package/shared* wird dabei als 'Storefront Types' geführt.

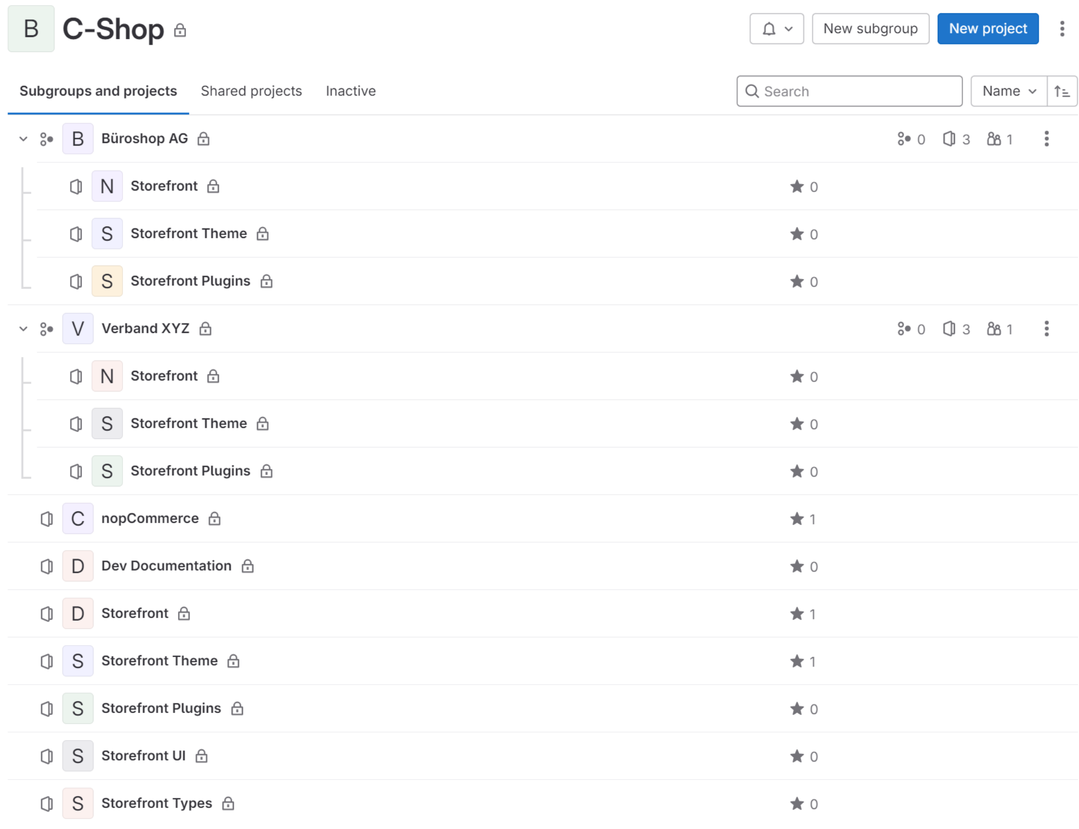


Abbildung 6.4: GitLab-Struktur des alternativen Entwicklungsprozesses

Bei der Einführung eines neuen Kunden werden neben dem Repository *Storefront* auch Forks von *Storefront Theme* und *Storefront Plugins* in die Subgruppe des Kunden erstellt. Diese Repositories werden vom Consultant entsprechend den Kundenanforderungen fortlaufend weiterentwickelt. Im Folgenden wird der vorgeschlagene neue Entwicklungsprozess bei einer Anpassung des Themes dargestellt.

Der Consultant klont das Repository *Storefront Theme* des Kunden, um es lokal zu bearbeiten. Im Gegensatz zum vorherigen Lösungsansatz besteht nun die Herausforderung, dass der Consultant kein direktes visuelles Feedback zu seinen Anpassungen am Theme erhält. Um dies zu gewährleisten, muss eine Instanz von Storefront lokal vorhanden sein. Mithilfe von pnpm-Link [38] wird ermöglicht, dass der Consultant direktes Feedback zu seinen Anpassungen am Theme in der lokalen Storefront-Instanz erhält. Dazu wird im Theme-Repository der Befehl gemäss Listing 6.1 in der Kommandozeile ausgeführt.

```
1 pnpm link --global
```

Listing 6.1: Theme-Paket global zur Verfügung stellen

Im Hintergrund erstellt dieser Vorgang einen Symlink[2] im globalen *node_modules*, der auf das lokale Theme-Repository zeigt.

Im Storefront-Repository wird anschliessend der Befehl gemäss Listing 6.2 in der Kommandozeile ausgeführt.

```
1 pnpm link --global storefront-theme
```

Listing 6.2: Symlink zu globalem npm-Paket erstellen

Dadurch wird im *package.json* die npm-Abhängigkeit vom Theme, welche im produktiven Betrieb auf das npm-Paket des Kunden zeigt, durch den im vorherigen Schritt erstellten Symlink ersetzt. Dies ermöglicht es, dass im Storefront die gewohnten Hot-Reload-Funktionen[3] zur Verfügung stehen und die lokalen Anpassungen am Theme innerhalb von Sekunden sichtbar sind. Nach Abschluss der Arbeit am Theme kann das Theme-Repository 'deployed' werden. Dazu wird in der Datei *package.json* eine neue Version festgelegt. Es wird empfohlen, die Versionierung nach Semantic Versioning [36] durchzuführen. Die Änderungen können auf das GitLab-Repository gepusht werden, und die Pipeline stellt dabei automatisch die neue Version auf npm zur Verfügung. Abschliessend kann im Storefront-Repository des Kunden die neue Version des Theme-Pakets hinterlegt werden.

In Abbildung 6.4 ist ersichtlich, dass in der neuen GitLab-Struktur das Repository *Storefront* pro Kunde geforkt wird, obwohl das Kernsystem bei allen Kunden identisch bleibt. Dies ist notwendig, da jeder Kunde in der *package.json* eine andere Abhängigkeit zu den Theme- und Plugin-Paketen hat, welche auf die npm-Pakete der Repositories *Storefront Theme* und *Storefront Plugins* verweisen. Dieses Repository wird benötigt, damit der Bundler die korrekten Abhängigkeiten in das Bundle integrieren kann, welches

[2]Ein Symlink, oder symbolischer Link, ist eine Art Dateiverknüpfung in Betriebssystemen, die auf eine andere Datei oder ein Verzeichnis verweist und es ermöglicht, darauf zuzugreifen, als wäre es an der Stelle des Links selbst.

[3]Hot-Reload ist eine Entwicklungsfunktion, die es Entwicklern ermöglicht, Änderungen am Code vorzunehmen und diese sofort in der laufenden Anwendung zu sehen, ohne dass ein kompletter Neustart erforderlich ist.

dann über dieses GitLab-Repository 'deployed' wird. Dies kann genutzt werden, um das Kernsystem weiterzuentwickeln und gemäss Anforderung der Consultants auf den Kundenprojekte zu verteilen. Es ist vorgesehen, im globalen *Storefront*-Repository pro Version einen Branch zu führen. Um einen Kunden auf eine neue Version des Kernsystems zu aktualisieren, kann dieser Branch in das Kunden-Repository gemerged werden. Dabei muss lediglich sichergestellt werden, dass die *package.json*-Abhängigkeiten auf die kundenspezifischen npm-Pakete verweisen.

6.4.2 Erkennung von Theme-Komponenten

Wenn ein Consultant an einer spezifischen Stelle im Storefront beispielsweise ein neues Attribut anzeigen lassen möchte, ist es schwierig herauszufinden, welche Komponente dies im Theme ist. Es ist erwünscht, eine visuelle Darstellung der verfügbaren Komponenten im Theme zu haben.
Basierend darauf wurde Storybook [46] evaluiert. Storybook ist ein Tool zur Erstellung interaktiver Dokumentationen von UI-Komponenten. Dies ermöglicht es einem Consultant, ein visuelles Beispiel einer Komponente aus dem Theme anzeigen zu lassen sowie zu betrachten, wie sich eine Komponente aufgrund anderer Argumente verändert. Durch eine prototypische Implementierung wird aufgezeigt, wie eine Integration von Storybook in das Theming-System aussehen könnte. Abbildung 6.5 zeigt die Komponente *ProductDetail* in der Storybook-Dokumentation. Weiter wird dadurch ermöglicht, die zentrale UI-Library für die Consultants zu visualisieren.

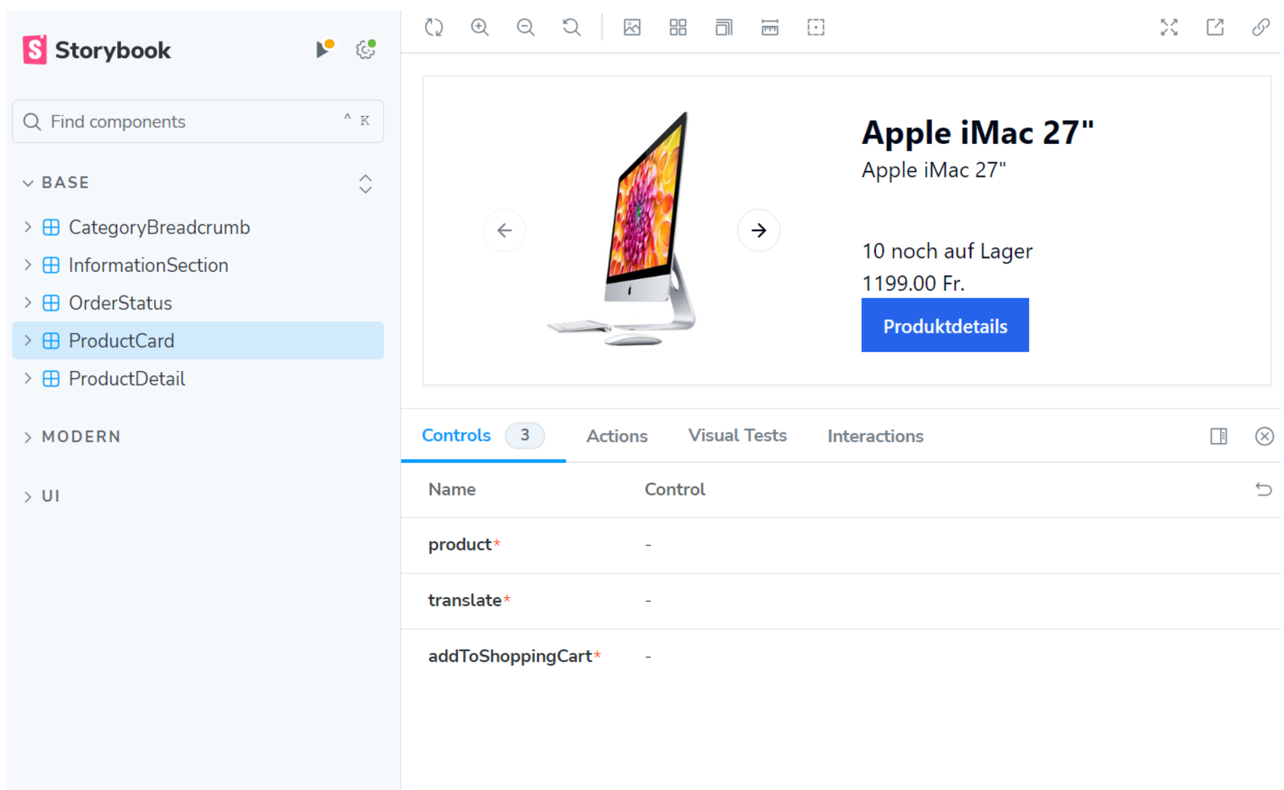


Abbildung 6.5: Theme-Dokumentation mit Storybook

6.5 Abschluss und Ausblick

Diese Bachelorarbeit zeigt mit dem entwickelten Storefrontsystem, dass der Industriepartner seinen Kunden eine geeignete Nachfolgelösung für die bisherige Lösung 'AbaShop' anbieten kann. Die erarbeiteten Themes und Beispiel-Plugins zeigen dem Industriepartner und seinen Consultants, wie sie das Storefrontsystem auf die Bedürfnisse eines individuellen Kunden konfigurieren können. Besonders wichtig ist, dass die Consultants gut geschult sind, um diese Konfigurationsmöglichkeiten effektiv nutzen zu können. Die zur Verfügung gestellte Entwicklerdokumentation bietet hierfür eine solide Grundlage, die kontinuierlich gepflegt werden sollte. Damit die Customize effizient mit vielen verschiedenen Kundeninstanzen des Storefrontsystems umgehen kann, wird vorgeschlagen, das in Abschnitt 5.9.6 dokumentierte Architektur-Refactoring zu übernehmen. Die Auslagerung der Themes und Plugins auf npm eröffnet neue Möglichkeiten für die Integration eines Marktplatzes zum Austausch und Teilen bereits entwickelter Lösungen. Ein weiteres wichtiges Ergebnis dieser Arbeit ist das 'Webapi'-Plugin, das zusammen mit dem 'Abacus Connector'-Plugin eine nahtlose Verbindung zwischen dem E-Commerce-System 'nopCommerce' und dem Abacus ERP ermöglicht und dessen Dienste über eine Schnittstelle anderen Systemen wie z.B. dem Storefrontsystem zur Verfügung stellt. Diese beiden Plugins befinden sich in einem stabilen Zustand und können durch das vorbereitete Versionskonzept kontinuierlich weiterentwickelt werden.

Der in dieser Bachelorarbeit entwickelte Prototyp des Storefrontsystems befindet sich zusammen mit den beiden Plugins für 'nopCommerce' in einem fortgeschrittenen Zustand. Auf Basis dieser Systeme kann Customize den ersten Kunden in einem Proof-of-Concept Projekt migrieren. Dieses Projekt soll genutzt werden, um weiteres Feedback der beteiligten Consultants zu sammeln. Auf dieser Basis kann anschliessend der auf Ende 2024 terminierte 'Make or Buy'-Entscheid im Projekt 'Ablösung 'AbaShop'' gefällt werden.

Anhang

Anhang A

Benutzertest

A.1 Einleitung

In Abschnitt 3.2 wurde der Akteur Consultant eingeführt, ein Mitarbeiter des Industriepartners, der das Storefrontsystem bei den Kunden einführen und betreuen wird. Consultants sind für die Implementierung der Kundenanforderungen zuständig und führen die Migration vom aktuellen AbaShop zum neuen System durch. Als intensive Nutzer des entwickelten Systems ist es entscheidend, deren Fähigkeit zu überprüfen, die Kundenanforderungen effektiv umzusetzen. Die Aufgaben eines Consultants umfassen die Einrichtung, Konfiguration sowie die Entwicklung von Themes und Plugins, beschrieben in den Abschnitten 3.5 und 5.7.

Die Erfahrungen mit AbaShop haben gezeigt, dass künftig intensiv mit dem Theming-System gearbeitet wird. Um sicherzustellen, dass alle Consultants, auch jene mit weniger Programmiererfahrung, das System effektiv nutzen können, wurde in der nichtfunktionalen Anforderung (NFA) 3.8.3 folgendes Kriterium festgelegt:
Ein Consultant, der mit dem entwickelten System nicht vertraut ist, sollte nach einer kurzen Einführung in der Lage sein, innerhalb von zehn Minuten eine einfache Kundenanforderung in einem Theme umzusetzen. Beispiele für einfache Aufgaben sind das Hinzufügen einer neuen Variablen zu einer Komponente oder die visuelle Anpassung eines Textes.

Die Einhaltung dieser Kriterien wurde in einem Benutzertest überprüft, der mit einem Consultant der Customize AG durchgeführt wurde. Dieser Consultant verfügt über sieben Jahre Erfahrung in der Einführung und Betreuung von AbaShop-Kunden und besitzt grundlegende Programmierkenntnisse, jedoch keine Erfahrung mit TypeScript und React. Es ist zu beachten, dass der Benutzertest aufgrund von Zeitbeschränkungen mit nur einer Person durchgeführt wurde und daher nicht repräsentativ sein kann.

A.2 Ausgangslage

Um aussagekräftige Testresultate zu erzielen, sind Grundkenntnisse in TypeScript und React notwendig. Aus diesem Grund erhält die Testperson eine einstündige Einführung in diese Technologien. Zusätzlich wird das Konzept der Themes und Plugins anhand des Komponentendiagramms des Storefrontsystems (siehe Abbildung 5.27) erläutert. Da die Testperson an der Erhebung der funktionalen Anforderungen beteiligt war, sind die Ziele dieser Arbeit ihr gut bekannt.

A.3 Testaufbau und Ablauf

Der Benutzertest wurde am 10.05.2024 in einem Sitzungszimmer der Customize AG in Winterthur durchgeführt. Die Einführung in die Technologien React und TypeScript sowie in das Konzept des Storefrontsystems fand von 08:30 bis 10:00 Uhr statt. Die erste Aufgabe erledigte die Testperson bis zum Mittag um 12:00 Uhr. Die weiteren Aufgaben 2 und 3 wurden in der Zeit von 13:00 bis 15:00 Uhr bearbeitet. Als Hilfsmittel stand der Testperson die Docs-Applikation zur Verfügung, die Nutzung des Internets war jedoch nicht gestattet.

A.4 Aufgaben

A.4.1 Aufgabe 1: Gesamtes System lokal aufsetzen

Aufgabenstellung

Dein Kunde X (Name wird anonymisiert) möchte auf die neue Lösung migrieren. Zunächst möchtest du das gesamte System lokal zum Laufen bringen. Erstelle dazu die definierte Struktur im GitLab-Projekt und richte das System ein. Das Ziel ist es, dass du am Ende das Storefrontsystem sowie nopCommerce eingerichtet hast und alle Systeme, einschliesslich Abacus, miteinander kommunizieren. Unter `https://mavm199.customize.ch` ist eine Installation von Abacus vorbereitet. Du kannst den Mustermandanten verknüpfen.

Beobachtungen

- Die Testperson erkennt schnell, dass in GitLab eine Subgruppe für den Kunden erstellt werden muss.
- Es wird erwähnt, dass die Strukturierung in GitLab als sinnvoll betrachtet wird.
- Beim Klonen des Repositories verwirren die Feature-Branches aus der Entwicklung; die Testperson sucht nach Versionsbranches.
- Die Testperson durchsucht die Docs-Applikation und findet sich darin schnell zurecht.
- Bei der Erstellung des Abacus-Schnittstellenbenutzers ist unklar, welche Scopes vergeben werden müssen.
- Da die Testperson noch keine Erfahrung mit Docker hat, entstehen einige Fragen, die direkt geklärt werden.
- Das Storefrontsystem kann problemlos in Betrieb genommen werden.
- Beim Öffnen des Storefrontsystems wird ein 'unbekannter Fehler' geworfen, der im ersten Moment unklar ist → Da das WebAPI-Plugin in nopCommerce noch nicht installiert wurde, werden 404-Codes retourniert.
- Die Testperson sucht länger, wo die Plugins in nopCommerce installiert werden können.
- Die Verbindung zu Abacus im Abacus Connector kann problemlos erstellt werden.
- Es stellt sich die Frage, was der Preis-Cache ist.
- Die Testperson wünscht sich einen Bereich, wo sie spezifische Logs zum Abacus Connector einsehen kann.

Resultat

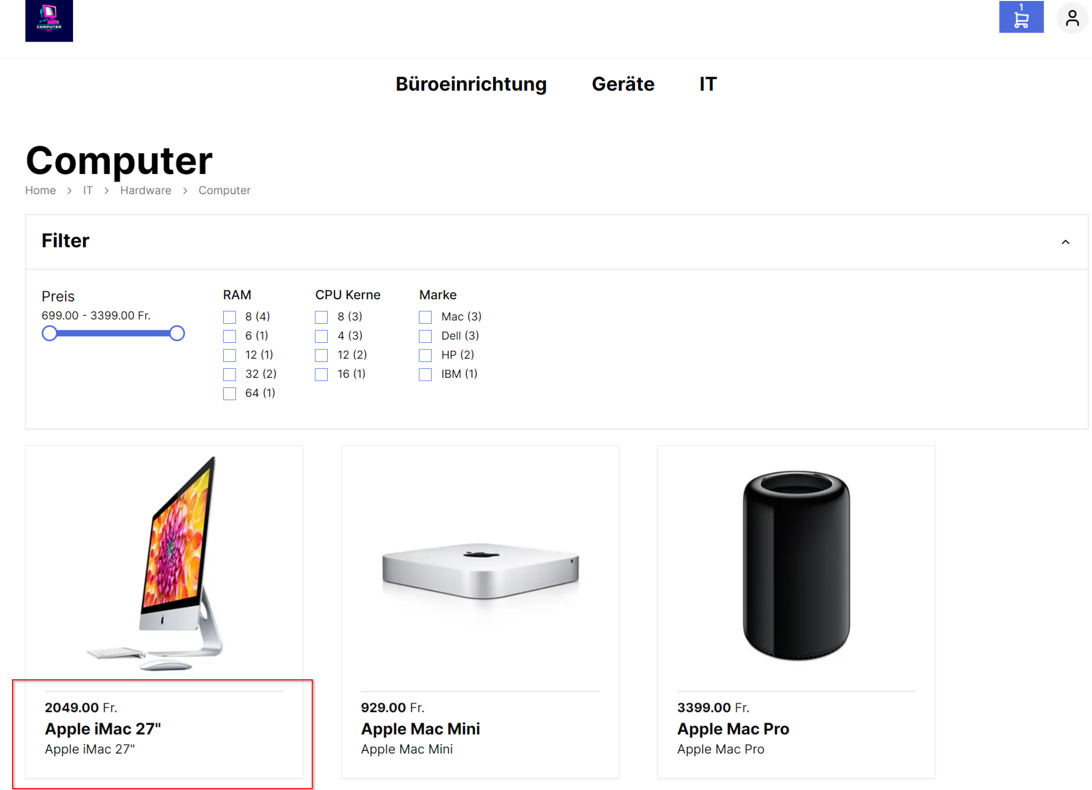
An einigen Punkten musste die Testperson zunächst suchen, bis der gewünschte Ort zur Konfiguration gefunden wurde. Da dies die erste Interaktion mit nopCommerce und dem Storefrontsystem war, kann dies jedoch als normal betrachtet werden. Die Testperson bestätigt, dass die Einrichtung beim nächsten Mal wesentlich schneller erledigt wäre. Sie empfand die Docs-Applikation als extrem hilfreiches Werkzeug, was sich auch darin zeigte, dass diese während der Aufgabe intensiv genutzt wurde und viele Fragen beantwortete.

Längere Diskussionen ergaben sich zum Thema Logging im Abacus Connector. Die Erfahrungen der Testperson aus AbaShop zeigen, dass oft Datendiskrepanzen zwischen dem Abacus ERP und AbaShop bestanden. Es ist daher essenziell, detaillierte Logs zur Verfügung zu haben, um genau nachvollziehen zu können, was im Synchronisationsdienst ablief

A.4.2 Aufgabe 2: Komponente im Theme anpassen

Aufgabenstellung

Nachdem du nun eine laufende Instanz der beiden Systeme hast und diese miteinander kommunizieren, befasst sich die nächste Aufgabe mit der Anpassung des Themes. Aktuell sieht die Produktklassierung für einen Shopper wie folgt aus:



Dein Kunde X, der im B2B-Geschäft (Business-to-Business) tätig ist, benötigt eine Anzeige des Lagerbestandes direkt bei der Produktklassierung (im rot markierten und eingerahmten Bereich), die momentan nur in der Detailansicht des Produktes verfügbar ist. Implementiere diese Anforderung in dem in Aufgabe 1 installierten System. Um die Aufgabe zu vereinfachen, kannst du das bestehende Theme anpassen, anstatt ein neues zu erstellen. Untersuche dabei die Struktur des Themes und erkläre, wie es aufgebaut ist.

Beobachtungen

- Die Testperson benötigt einige Minuten, um sich im Monorepo zurechtzufinden.
- Es gibt Schwierigkeiten herauszufinden, welche Komponente mutiert werden muss, da es eine grosse Anzahl von Komponenten im Theme gibt →Eine Volltextsuche schafft Abhilfe.
- Die Mutation erfolgt dabei problemlos:
  - Die Testperson kann die Mutation sofort durchführen.
  - Es wird erwähnt, dass der Vorgang sehr intuitiv ist und die Erfahrungen aus AbaShop hilfreich sind bzw. die Systematik ähnlich ist.
  - IntelliSense erweist sich als enorm hilfreich (ist bei der aktuellen AbaShop-Lösung nicht vorhanden).
- Die Testperson vermisst eine Möglichkeit, die Anpassung als neue Version zu definieren.

Resultat

Das festgelegte Zeitkriterium von 10 Minuten für eine einfache Mutation am Theme konnte knapp eingehalten werden. Dabei ist zu erwähnen, dass der grösste Teil der Zeit darauf verwendet wurde, dass sich die Testperson im Monorepo zurechtfand. Die eigentliche Mutation wurde in etwa 4 Minuten durchgeführt.

Abbildung A.1 zeigt die vom Consultant angepasste Komponente.

```
export function ProductCard({ product, translate }: ProductCardProps) {
  return (
    <Card>
      <CardContent>
        <section>
          <Carousel className="max-w-[180px]">
            <CarouselContent>
              {product.ImageUrl.map((url, i) => (
                <CarouselItem key={i}>
                  <Image
                    src={url}
                    width={200}
                    height={200}
                    alt={product.ShortDescription}
                  />
                </CarouselItem>
              ))}
            </CarouselContent>
            <CarouselPrevious />
            <CarouselNext />
          </Carousel>
        </section>
        <section className="flex flex-col justify-between">
          <div>
            <h2 className="text-2xl font-bold">{product.Name}</h2>
            <p>{product.ShortDescription}</p>
          </div>
          <div>
            <p> Lagerbestand: {product.StockQuantity}</p>
            <p>{`${product.Price.toFixed(2)} Fr.`}</p>
            <Button>
              <Link href={`/products/${product.SearchEngineProperties.Slug}`}>
                Produktdetails
              </Link>
            </Button>
          </div>
        </section>
      </CardContent>
    </Card>
  );
}
```

Listing A.1: Angepasste Produktcard-Komponente im Benutzertest

Im Anschluss an diese Aufgabe wurden Diskussionen bezüglich der Versionierung geführt. Ursprünglich ist vorgesehen, dass die Anpassungen an einem Theme über einen Branch erfolgen, der anschliessend in der GitLab-Subgruppe des Kunden in den Main-Branch gemerged wird. Dies würde über die Pipeline einen neuen Build auslösen. Die Diskussionen führten zu einer alternativen Lösungsmöglichkeit: Die Versionierung erfolgt über das npm-Paket im Monorepo (*package/theme*). Im Storefrontsystem wird dann in der *package.json* die Version der npm-Abhängigkeit angepasst. Folglich müsste man das Theming aus dem Monorepo herauslösen und auf npm hosten.

A.4.3 Aufgabe 3: Entwicklung einfaches Plugin

Aufgabenstellung

Das integrierte Plugin-System ermöglicht es, kundenindividuelle Anforderungen zu erfüllen, ohne den Source-Code im Storefrontsystem zu ändern. Die Möglichkeiten zur Entwicklung von Plugins für erfahrene Programmierer sind sehr umfangreich. In dieser Aufgabe besteht das Ziel darin, ein sehr einfaches Plugin zu schreiben und produktiv zu schalten. Dein Kunde X gibt an, dass eine Promotion gestartet wird. Jeder Kunde, der den Artikel '0320' in den Warenkorb legt, soll zu Promotionszwecken den Artikel '0520' gratis in den Warenkorb hinzugefügt bekommen. Für diese Aufgabe genügt es, wenn die beiden Artikel fest im Code verankert sind.

Beobachtungen

- Anfangs ist die Testperson unsicher, wie vorzugehen ist, und schaut sich daher die bestehenden Plugins an.
- Die Implementierung des Interfaces *IPlugin* wird daraufhin vorgenommen; benötigte Metadaten werden problemlos ergänzt.
- Die Testperson versucht einige Ansätze, ist sich jedoch unsicher, wie dies angegangen werden könnte →Daraufhin erhält die Testperson den Hinweis, in der Docs-Applikation nachzuschauen:
  - Die Testperson sieht dann relativ schnell, dass mit dem Event-System eine Umsetzung möglich ist.
  - Die Testperson erkennt, dass über BasePlugin die Service-Fassade zur Verfügung steht.
- Die restliche Implementierung wird schnell geschrieben.
- Die Testperson erwartet, dass das Plugin direkt im Adminbereich zur Installation verfügbar ist, sieht dann aber in der Docs-Applikation, dass noch ein Export in der Datei *index.ts* erstellt werden muss.

Resultat

Die Implementierung des Plugins wurde in etwa 30 Minuten erfolgreich abgeschlossen. Auch in dieser Aufgabe erwies sich die Docs-Applikation als äusserst hilfreich. Dabei wurden diverse Punkte identifiziert, die in der Dokumentation noch detaillierter beschrieben werden könnten. In Bezug auf die Versionierung ergibt sich hier dieselbe Thematik, wie bereits in Aufgabe 2 ausgeführt.

Listing A.2 zeigt die Plugin-Implementation der Testperson.

```
import {
  IPlugin,
  NewBasketItem,
  Subscriber,
} from "@repo/shared";
import { BasePlugin } from "../BasePlugin.abstract";

export class PromoPlugin
  implements IPlugin, Subscriber<NewBasketItem>
{

  name = "Promo Plugin";
  version = "1.0";
  creator = "Benutzertest";
  description =
    "Dieses Plugin legt beim Einfuegen von Artikel 0320 zusaetzlich
    den Artikel 0520 gratis in den Warenkorb";
  priority = 1;


  async update(eventData: NewBasketItem) {
    const service = BasePlugin.getServiceInstance();
    if (eventData.productId == "0320") {
      service.addToBasket("0520", 1, 0);
    }
  }
}
```

Listing A.2: Entwickeltes Plugin im Benutzertest

A.5 Bewertung

Das erarbeitete Konzept mit dem Theming-System wurde von der Testperson sehr positiv bewertet. Auch ohne Kenntnisse in React konnte ein Consultant schnell die gewünschten Anpassungen vornehmen, da der Aufbau aus reinen Präsentationskomponenten es ermöglicht, mit bestehenden HTML-Kenntnissen viele Anforderungen umzusetzen. Das Testkriterium der *Einfachheit eines Themes* gemäss NFA wurde somit erfüllt. Insgesamt konnten keine erheblichen Mängel festgestellt werden. Die Durchführung des Benutzertests und die nachfolgenden Diskussionen ergaben einige Punkte, die im nachfolgenden Abschnitt aufgeführt werden.

A.6 Massnahmen

Durch die genaue Beobachtung der Testperson konnten insbesondere in der Docs-Applikation viele Punkte aufgenommen werden, die in die Dokumentation aufgenommen werden sollten oder noch genauer beschrieben werden müssen. Die Verbesserungen in der Dokumentation konnten nach der Durchführung des Benutzertests bereits umgesetzt werden. Neben den Erweiterungen der Dokumentation resultieren zwei Hauptpunkte aus dem Benutzertest:

- **Erkennung von Theme-Komponenten:** Es ist für Consultants schwierig zu erkennen, welche Komponente wo eingesetzt wird und was sie schlussendlich beinhaltet. In Aufgabe 2 musste beispielsweise die Produktcard-Komponente angepasst werden. Durch die grosse Flexibilität des Theming-Systems stehen auch entsprechend viele Komponenten zur Verfügung. Da die Testperson nur wusste, wie die Komponente aussieht, die sie anpassen möchte, fehlte ihr eine Möglichkeit, in den verfügbaren Komponenten direkt zu sehen, was diese beinhalten. Aus diesem Grund wurde im Nachgang des Benutzertests eine prototypische Implementierung von Storybook gemacht. Storybook ist eine interaktive Dokumentation von UI-Komponenten, die es ermöglicht, Komponenten isoliert zu visualisieren und deren Verhalten unter verschiedenen Bedingungen zu simulieren. Storybook dient somit als zentrale Plattform, auf der Consultants schnell und effizient Zugriff auf alle verfügbaren Theme-Komponenten haben. Sie können die Komponenten mit unterschiedlichen Properties konfigurieren und sofort sehen, wie sich die Änderungen auswirken. Storybook ist nicht nur hilfreich, um die Theme-Komponenten darzustellen. Ebenfalls das Paket *package/ui*, welches zentrale UI-Komponenten beinhaltet und von Themes/Plugins genutzt werden kann, kann in Storybook dokumentiert werden.

- **Versionierung der Themes/Plugins:** Grundsätzlich ist vorgesehen, über das Versionierungssystem (Git) die Mutationen an Themes oder Plugins zu versionieren. Dabei wird das Monorepo als Ganzes versioniert. Ein Update am Theme bedeutet schlussendlich ein Update am Monorepo und kann über den Versionsbranch in den Main-Branch produktiv geschaltet werden. Die Diskussion am Benutzertest ergab eine alternative Lösung, die zukünftig geprüft werden soll. Da die Themes in einem eigenen internen npm-Paket *package/themes* leben, könnten die Versionierungsmöglichkeiten von npm genutzt werden. Konsequenz ist, dass die beiden Pakete *package/themes* und *package/plugins* aus dem Monorepo herausgelöst werden müssen und als eigenständiges Projekt auf npm gehostet werden. Möchte der Consultant die 'deployed'e Anpassung am Theme produktiv schalten, muss als zweiter Schritt die Version der npm-Abhängigkeit im Kernsystem angepasst werden.

Anhang B

Developer Dokumentation

Im Rahmen dieser Arbeit wurde eine Docs-Anwendung entwickelt, die dem Industriepartner in einer gehosteten MDX-Anwendung zur Verfügung gestellt wird. MDX ist ein Superset von Markdown, welches zusätzlich das Rendern von React-Komponenten in Markdown ermöglicht.
Zielgruppe dieser Dokumentation sind Consultants, die für einen Kunden ein neues Storefrontsystem aufsetzen wollen und eine technische Dokumentation des Gesamtsystems benötigen, sowie zukünftige Theme- und Plugin-Entwickler.

Abbildung B.1 zeigt einen Ausschnitt aus der entwickelten Docs-Anwendung.

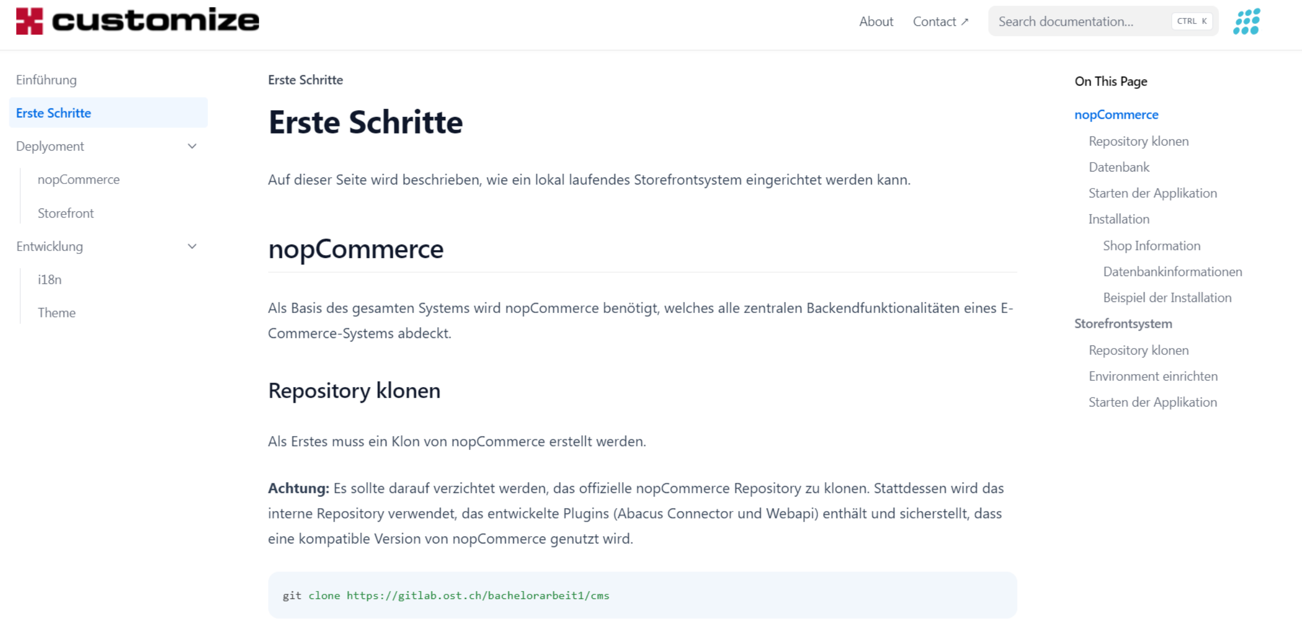


Abbildung B.1: Ausschnitt Dev-Docs

B.1 Erste Schritte

In folgendem Abschnitt wird beschrieben, wie ein lokal laufendes Storefrontsystem eingerichtet werden kann.

B.1.1 nopCommerce

Als Basis des gesamten Systems wird nopCommerce benötigt, welches alle zentralen Backendfunktionalitäten eines E-Commerce-Systems abdeckt.

Repository klonen

Als Erstes muss ein Klon von nopCommerce erstellt werden.

Achtung: Es sollte darauf verzichtet werden, das offizielle nopCommerce Repository zu klonen. Stattdessen wird das interne Repository verwendet, das entwickelte Plugins (Abacus Connector und Webapi) enthält und sicherstellt, dass eine kompatible Version von nopCommerce genutzt wird.

```
git clone https://gitlab.com/c-shop/nopcommerce/
```

Datenbank

Für den Einsatz von nopCommerce wird eine einsatzbereite Datenbank vorausgesetzt. Um den Prozess zu vereinfachen, ist ein Dockerfile vorbereitet worden, das das Hochfahren einer lokalen Postgres-Instanz ermöglicht.

Es wird angenommen, dass Docker lokal installiert ist und Docker Compose für die vereinfachte Erstellung der Container zur Verfügung steht. Sollte dies nicht der Fall sein, können die folgenden Anleitungen genutzt werden, um die Installation durchzuführen:

- Um Docker auf Windows zu installieren, folge diesem Tutorial: Install Docker on Windows
- Um Docker Compose auf zu installieren, folge diesem Tutorial: Install Docker Compose on Windows

Sobald Docker und Docker Compose installiert sind, navigiere in das geklonte Projektverzeichnis und starte die Postgres-Instanz mit dem folgenden Befehl:

```
docker-compose -f postgresql-docker-compose.yml up
```

Starten der Applikation

Nachdem die Datenbank gestartet wurde, kann die Solution in der gewünschten Entwicklungsumgebung ausgeführt werden. Es wird empfohlen, Jetbrains Rider zu nutzen, aber Visual Studio ist ebenfalls eine problemlose Option.

Es ist auch möglich, die Applikation in einem Docker Container zu betreiben. In diesem Fall ist der im Abschnitt Datenbank beschriebene Schritt nicht erforderlich. Für diese Methode steht ein Docker Compose

zur Verfügung, das sowohl die Applikation als auch die Datenbank erstellt.

```
docker-compose up
```

Die Applikation kann nun im Browser der Wahl unter `http://localhost:5000/` aufgerufen werden.

Installation

Beim erstmaligen Starten von nopCommerce wird die Installation durchgeführt. Im geöffneten Fenster werden von nopCommerce die benötigten Informationen abgefragt. Nachdem 'Installieren' gedrückt wird, erstellt nopCommerce die benötigten Tabellen in der Datenbank.

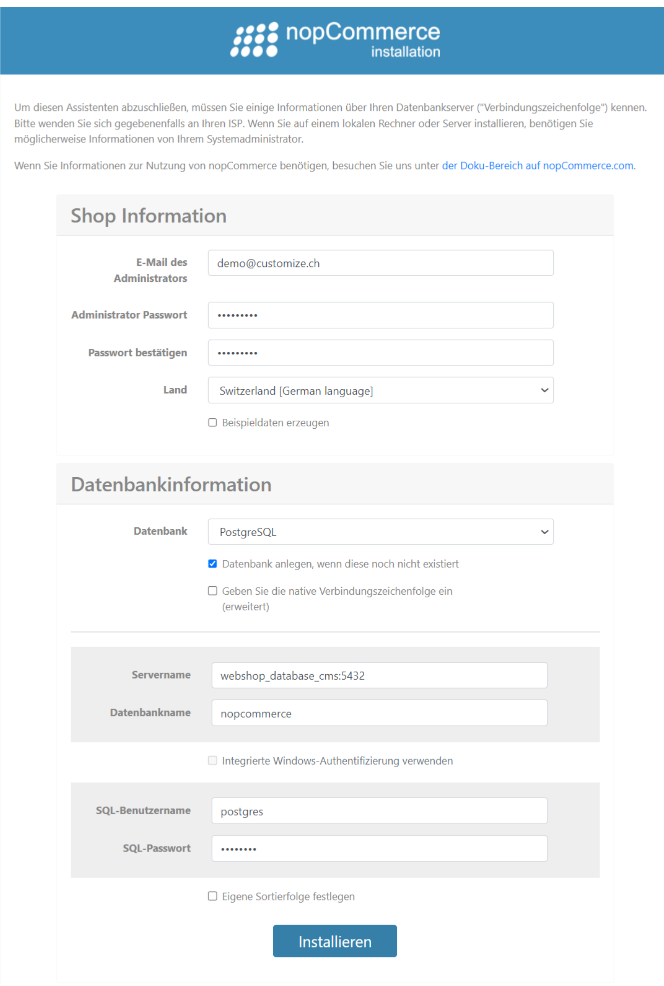
Shop Information Diese Informationen werden später benötigt, um als Administrator auf den Adminbereich zuzugreifen. Es ist empfohlen, bei produktiven Kundenprojekten entsprechend die Informationen vom Kunden zu hinterlegen, sodass die Admininformationen dem Kunden übergeben werden können.

- **E-Mail des Administrators**: E-Mail vom Administrator/Superuser des Kunden
- **Passwort**: Generiere mit dem Passwortmanager der Customize ein sicheres Passwort
- **Land**: Switzerland (German Language)

Datenbankinformationen

- **Datenbank**: PostgreSQL
  - **Hinweis**: Haken bei 'Datenbank anlegen, wenn diese noch nicht existiert' muss bei einer frischen Installation gesetzt werden.
  - **Servername**: server:port

Eigenschaft	Wert
SQL-Benutzername	postgres
SQL-Passwort	postgres



Nach einem Neustart der Applikation ist nopCommerce einsatzbereit.

B.1.2 Storefrontsystem

Das mit Next.js entwickelte Storefrontsystem wird mit dem Package Manager pnpm erstellt. Um pnpm auf Windows zu installieren, folge diesem Tutorial: Install pnpm on Windows

Repository klonen

Als Erstes muss ein Klon des Storefrontsystems erstellt werden.

```
git clone https://gitlab.com/c-shop/storefront/
```

Environment einrichten

In der Datei `.env` können die Umgebungsvariablen für den lokalen Build parametriert werden.

- `POSTGRES_PRISMA_URL`
  - **Wert:** `postgresql://postgres:postgres@localhost:5433/nopcommerce?schema=storefront`
  - **Erklärung:** Dieser Verbindungsstring wird für Prisma zur Verbindung mit der PostgreSQL-Datenbank verwendet.
- `POSTGRES_URL_NON_POOLING`
  - **Wert:** `postgresql://postgres:postgres@localhost:5433/nopcommerce?schema=storefront`
  - **Erklärung:** Ähnlich wie `POSTGRES_PRISMA_URL`, wird dieser String in speziellen Fällen verwendet werden, um Verbindungspooling zu deaktivieren.
- `NOPCOMMERCE_BASE_URL`
  - **Wert:** `http://localhost:5000`
  - **Erklärung:** Die Basis-URL für den Zugriff auf nopCommerce gemäss Einrichtung aus Abschnitt nopCommerce

Starten der Applikation

Das Storefrontsystem kann im root-Verzeichnis mit folgendem Befehl gestartet werden.

```
pnpm dev
```

Die Applikation führt automatisch alle Datenbankmigrationen durch und kann unter `http://localhost:3001` aufgerufen werden.

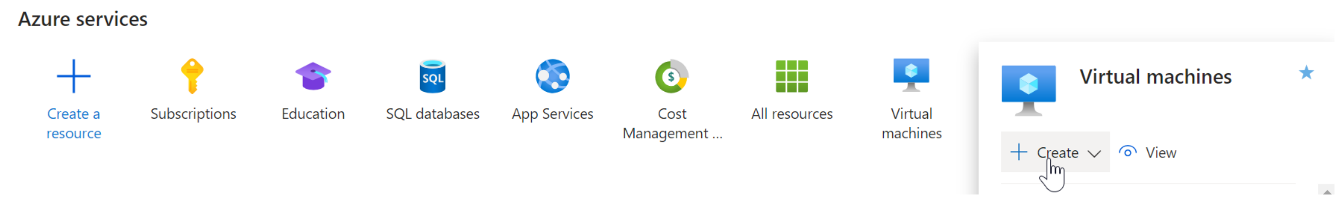
B.2 Deployment nopCommerce

In diesem Abschnitt wird aufgezeigt, wie nopCommerce produktiv 'deployed' werden kann. Es ist vorgesehen, die Applikation auf einem Windows-Server mit IIS zur Verfügung zu stellen. In dieser Dokumentation wird aufgezeigt, wie das Deplyoment mit einer Azure Virtual Machine (VM) durchgeführt werden kann.

B.2.1 VM erstellen

Im Azure Portal wird die VM gemäss benötigten Spezifikationen umgesetzt.

Hinweis: Es ist empfohlen, mindestens 8GB Arbeitspeicher mit 2 virtual CPUs zu nehmen. Der Disk Space ist abhängig von der Grösse des Abacus Mandanten. Beachte, dass alle Klassierungs- und Produktbilder auf dem Server gespeichert werden.



Der Benutzername und Passwort werden im Passwortmanager der Customize abgelegt.

Die restlichen Konfigurationen können gemäss Vorschlag ausgewählt werden.

B.2.2 DNS einrichten

- öffne die erstelle VM im Azure Portal
- navigiere zu `Overview`
- klicke auf `Configure` beim DNS Namen
- vergebe einen global einzigartigen Namen (ein grüner Haken erscheint, sobald er einzigartig ist)

Dies ist die Domain, unter welcher der Server und entsprechend nopCommerce schlussendlich erreichbar sein wird.

B.2.3 Firewall einrichten

Damit die benötigten Ports offen sind, muss die Konfiguration der Firewall vorgenommen werden. Navigiere dazu nach `Networking settings` um neue Firewall Regeln zu erfassen (`Create port rule`).

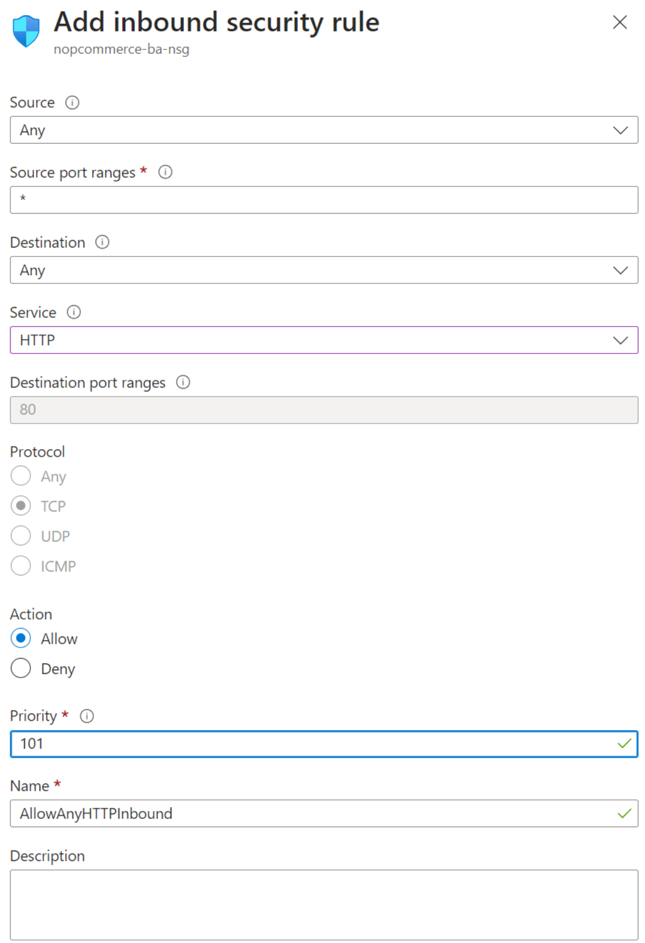
Inbound Rules

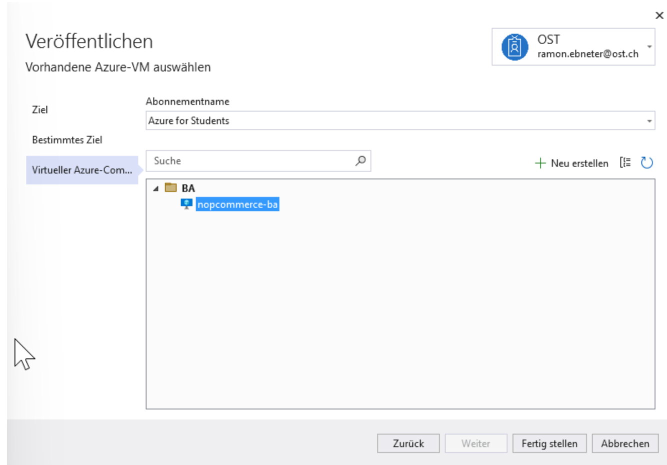
Protokoll	Port	Priorität
HTTP	80	100
WebDeploy	8172	1010
RDP	3389	320

Outbound Rules

Protokoll	Port	Priorität
RDP	3389	100

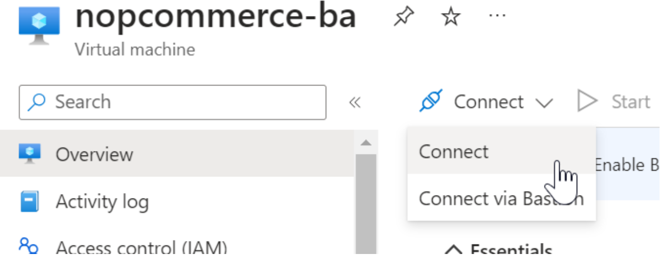
Beispiel



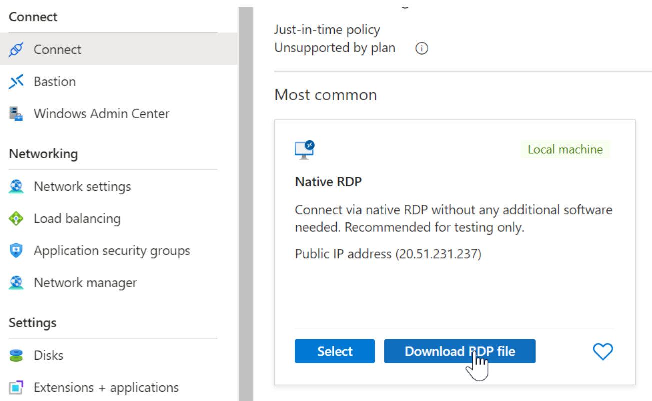


Die nächsten Konfiguration werden innerhalb der VM durchgeführt. Dazu muss diese geöffnet werden. Am einfachsten ist dies möglich, indem im Azure Portal eine RDP Datei heruntergeladen wird.

Navigiere dazu in den Bereich Overview und klicke auf Connect



Selektiere 'Download RDP File'

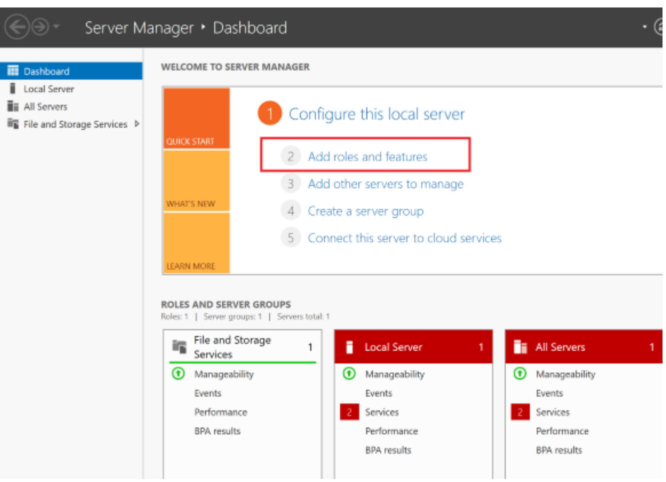


Nach Ausführen der Datei kann mittels Benutzername und Passwort aus dem Abschnitt VM erstellen der Server geöffnet werden.

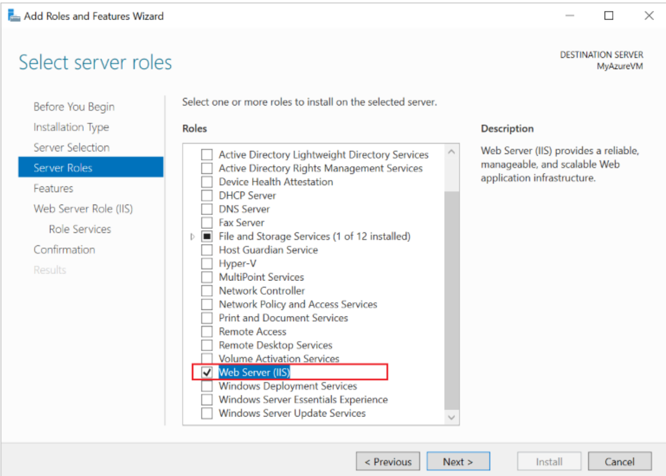
B.2.4 IIS Webserver installieren

Öffne das Server Manager Dashboard (wird beim Start eines Windows Servers automatisch geöffnet).

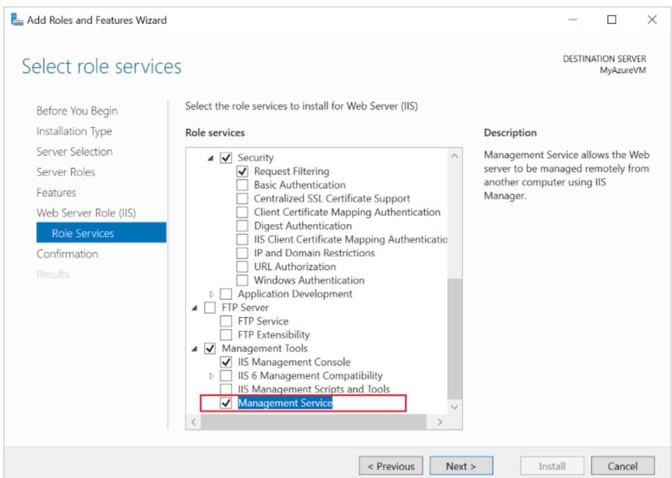
Wähle '2 - Add roles and features' aus.



Akzeptiere die vorgeschlagenen Einstellungen und navigiere bis 'Server Roles'. Wähle 'Web Server (IIS)' an.



Akzeptiere die vorgeschlagenen Einstellungen und navigiere bis 'Role Services'. Wähle 'Management Service' aus. Dies wird benötigt, damit über Port 8172 das Webdeploy zur Verfügung steht.



Die restliche Einstellungen können akzeptiert werden und am Schluss mit 'Install' bestätigt werden. Nach Abschluss dieses Abschnitts wurde folgendes erfolgreich konfiguriert:

- IIS ist installiert und wird ausgeführt, mit einer internen Firewall-Regel erstellt für Port 80.
- Web Management Service ist installiert, mit einer internen Firewall-Regel erstellt für Port 8172.

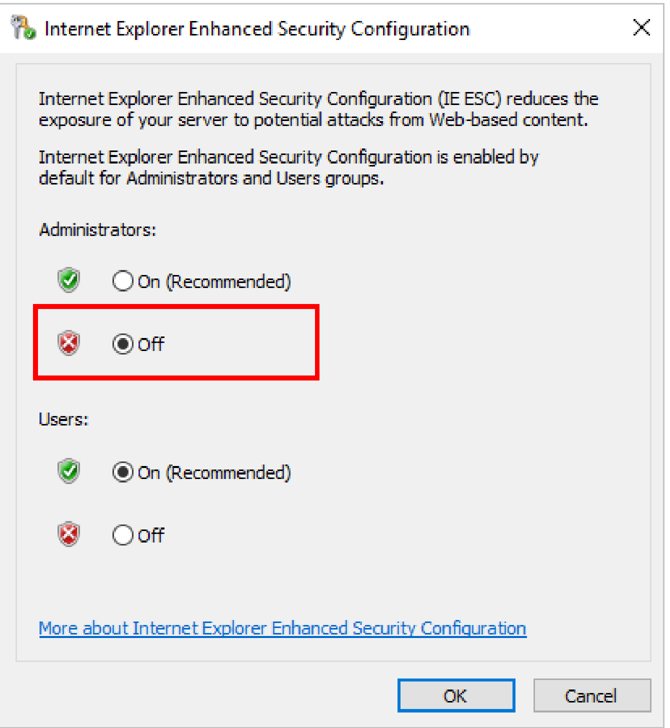
B.2.5 VM Security Einstellungen einrichten

Da in den beiden nachfolgenden Kapiteln .NET installieren und Web Deploy einrichten Executeables aus dem Internet geladen werden müssen, benötigt es Anpassungen an den Sicherheitseinstellungen des Servers, welche im Nachgang wieder rückgängig gemacht werden können. Alternativ können die Executables in den Server kopiert werden, womit dieser Abschnitt optional ist.

1. Öffne den Server Manager und wähle 'Local Server' aus

2. In der Mitte des Panels, neben 'IE Enhanced Security Configuration' wähle 'On' aus.

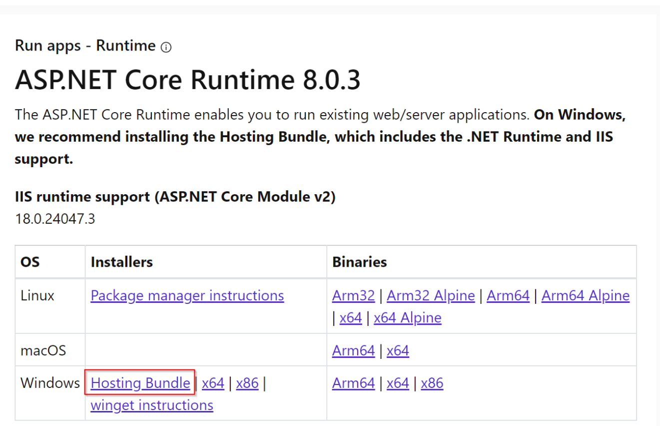
3. Im geöffneten Dialog wähle 'Off' beim Administrator und 'On' beim Umgebungsvariablen



B.2.6 .NET installieren

Da nopCommerce eine .NET Core Applikation ist, wird eine Installation der _.NET Core SDK_ benötigt. Installiere dazu die aktuelle Version.

Ebenfalls wird für das IIS Hosting eine Installation des Pakets .NET Core Windows Server Hosting benötigt.



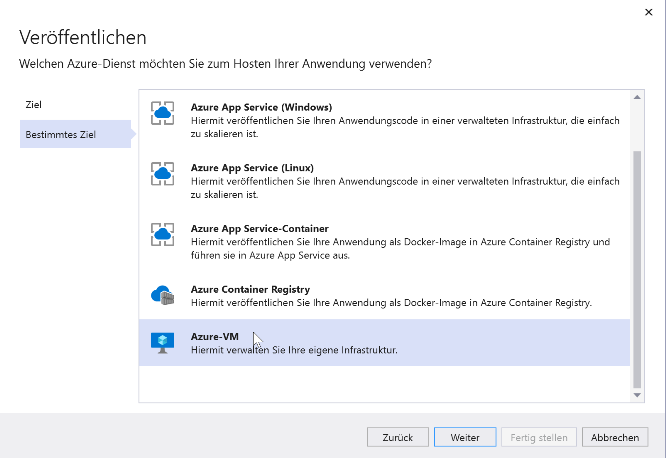
Damit das Paket funktioniert, muss der IIS Webserver neu gestartet werden. Führe dazu auf der CMD folgenden Befehl aus:

```
iisreset
```

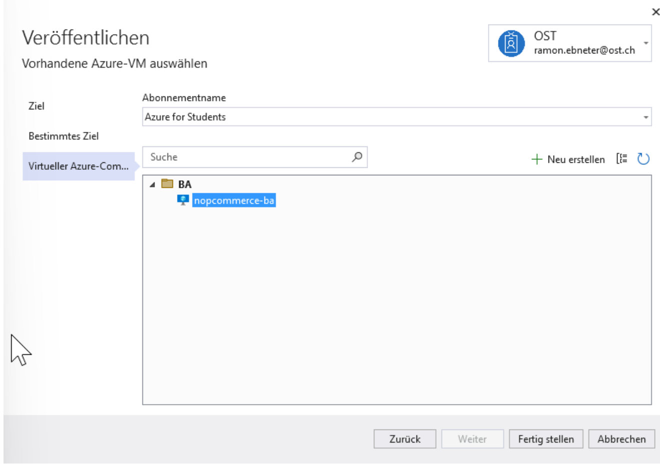
B.2.7 Web Deploy einrichten

Hinweis: Als Referenz kann auch die offizielle Dokumentation von Microsoft beigezogen werden. Die nachfolgende Dokumentation zeigt auf, wie mit Visual Studio das Web Deplyoment durchgeführt werden kann. Falls mit Jetbrains Rider gearbeitet werden, kann die Dokumentation von Jetbrains beigezogen werden.

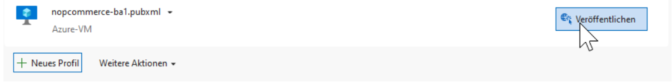
1. Öffne das geklonte nopCommerce Repository aus dem Abschnitt Erste Schritte in Visual Studio.
2. Öffne das Projekt /Presentation/Nop.Web und wähle mit einem Rechtsklick 'Publish aus'
3. Wähle 'Azure aus'
4. Scrolle ganz nach unten und wähle 'Azure-VM'



5. Führe die Authentifizierung mit Azure durch und wähle die angezeigte VM aus.



6. Visual Studio legt nun ein neues Web Deploy Profil an. Auf disem Profil kann die Applikation nun 'deployed' werden.

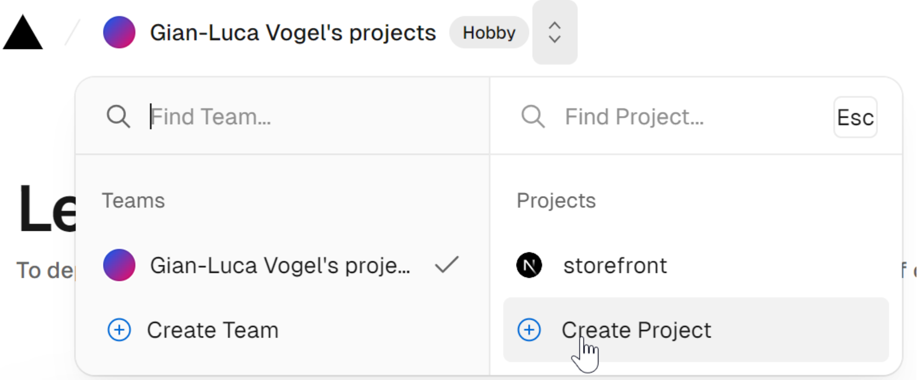


B.3 Deployment Storefrontsystem

In diesem Abschnitt wird aufgezeigt, wie das Storefrontsystem produktiv 'deployed' werden kann. Es ist vorgesehen, die Applikation auf Vercel, dem Hersteller vom Next.js Framework, zu deployen.

B.3.1 Vercel Projekt einrichten

Erstelle auf Vercel ein neues Project.



Öffne das erstellte Projekt und navigiere zu `Settings`. Unter `Build & Development Settings` müssen folgende Einstellungen angepasst werden:

- **Build Command**: `cd ../../ && pnpm build --filter=shop`
- **Install Command**: `pnpm install`

Unter `Root Directory` muss `apps/shop` eingetragen werden.

Umgebungsvariablen einrichten

Navigiere im Vercel Projekt nach `Settings` und dann in `Environment Variables`. Für das produktive Deployment werden die nachfolgenden Variablen benötigt:

Variable	Wert	Beschreibung
`NOPCOMMERCE_BASE_URL`	http://testdeployba.eastus.cloudapp.azure.com	URL der produktiven nopCommerce Instanz
`POSTGRES_URL`	postgres://default:prodpostgresdomain/storefront?sslmode=require	Connection String der produktiven Postgres Instanz mit dem gewünschten Datenbanknamen
`POSTGRES_URL_NON_POOLING`	identisch wie oben	Connection String Postges Datenbank für non Polling Aufrufe

B.3.2 Gitlab Umgebungsvariable anpassen

Damit die vorbereitete Pipeline das Deployment auf Vercel durchgeführt werden kann, wird ein Token benötigt.

1. Erstelle einen Access-Token in den Account Settings unter `Token`
2. Öffne das Storefrontprojekt in Gitlab
3. Navigiere zu `Settings` und dann nach `CI/CD`
4. Unter Variables kann via `Add Variable` eingefügt werden

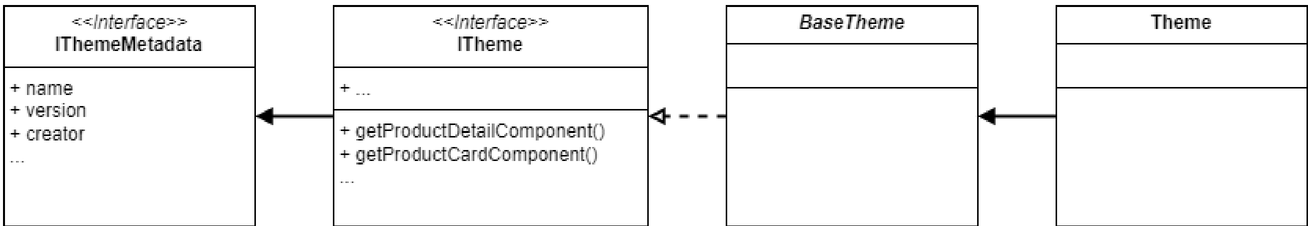
Key	Value
`VERCEL_TOKEN`	Der Token vom vorangegangen Schritt

B.4 Theme Entwicklung

Mit einem Theme lassen sich die einzelnen Komponenten des Webshops sowie der strukturelle Aufbau des gesamten Shops verändern, ohne das Kernsystem anzupassen.

B.4.1 Aufbau

Ein Theme besteht aus einer Klasse, die von der abstrakten Klasse *BaseTheme* abgeleitet ist. Die Klasse *BaseTheme* enthält alle Funktionen, die zum Laden der einzelnen Komponenten benötigt werden. Dadurch müssen bei der Entwicklung eines Themes nur die Komponenten erstellt werden, die geändert werden sollen. *BaseTheme* selbst implementiert das Interface ITheme, das von IThemeMetadata erbt.



B.4.2 Theme Aufbau im Detail

Der folgende Code zeigt das 'Basistheme' mit überschreibbaren Methoden und Instanzvariablen. Die beiden Instanzvariablen *id* und *isActive* werden vom System verwaltet und müssen daher nicht vom Entwickler gesetzt werden.

```
export default abstract class BaseTheme implements ITheme {
    abstract id?: string;
    abstract isActive?: boolean;
    abstract name: string;
    abstract version: string;
    abstract creator: string;
    abstract description: string;

    getProductDetailComponent() {return ProductDetail;}
    getAttributeSelectFilterComponent() {return AttributeSelectFilter;}
    getPriceFilterComponent() {return PriceFilter;}
    getProductCardComponent() {return ProductCard;}
    getCategoryBreadcrumbComponent() {return CategoryBreadcrumb;}
    getLanguageChangerComponent() {return LanguageChanger;}
    getContactSectionComponent() {return ContactSection;}
    getInformationSectionComponent() {return InformationSection;}
    getAddressSectionComponent() {return AddressSection;}
    getThemeColors() {return baseThemeColors;}
    getFooterContainerComponent() {return FooterContainer;}
    getFilterContainerComponent() {return FilterContainer;}
    getShopLayoutContainerComponent() {return ShopLayoutContainer;}
    getNavBarComponent() {return NavBar;}
    getCategoryNavigationComponent() {return CategoryNavigation;}
    getShoppingCartComponent() {return ShoppingCart;}
    getHomeLayoutContainerComponent() {return HomeLayoutContainer;}
    getOrdersContainerComponent() {return OrdersContainer;}
    getSubCategoriesListComponent() {return SubCategoriesList;}
  }
```

B.4.3 Theme Instanzvariablen

In der folgenden Tabelle wird der Nutzen der einzelnen Instanzvariablen erklärt. Zudem wird erläutert, ob diese vom Entwickler initialisiert werden müssen oder nicht.

Name	Funktionen
id	Die ID des Themes muss nicht vom Entwickler initialisiert werden.
isActive	Der Zustand des Themes muss nicht vom Entwickler initialisiert werden.
name	Der Name des Themes muss vom Entwickler initialisiert werden.
version	Die Version des Themes muss vom Entwickler initialisiert werden.
creator	Der Name des Theme-Entwicklers.
description	Kurze Beschreibung des Themes. Muss vom Entwickler initialisiert werden.

B.4.4 Komponenten

Nachfolgend sind alle Komponenten aufgelistet, die im *BaseTheme* enthalten sind. In den jeweiligen Tabellen werden die Eigenschaften der einzelnen Komponenten beschrieben. Bei Komponenten, die *Container* im Namen tragen, handelt es sich um sogenannte Container. Dies sind Komponenten, die nur andere Komponenten beinhalten und dazu da sind, die Anordnung dieser Komponenten zu definieren.

Wichtig: Alle Theme-Komponenten-Eigenschaften implementieren das Interface *ThemeProps*, welches die Funktion für die Übersetzung beinhaltet.

Das Aussehen der einzelnen Komponenten kann man im Storybook des Projektes ansehen. Um Storybook zu starten, navigiere zum Root-Verzeichnis und führe folgenden Befehl aus:

```
pnpm storybook
```

War dies erfolgreich, findest du eine interaktive Dokumentation der einzelnen Komponenten unter der URL `http://localhost:6006`.

ProductDetail

Property	**Beschreibung**
product	Objekt vom Typ Product. Es enthält Details zu einem bestimmten Produkt, wie z.B. den Namen, die Beschreibung, den Preis usw.
addToCart	Diese Funktion akzeptiert drei Parameter: productId (Produkt-ID), quantity (Menge) und price (Preis). Sie gibt ein Promise zurück, das sich auflöst, wenn das Produkt zum Warenkorb hinzugefügt wurde.

AttributeSelectFilter

Property	**Beschreibung**
addFilterValues	addFilterValues ist eine Funktion welche optional ein Array von Strings ('values') als Argument nimmt. Die Funktion wird verwendet, um neue Filterwerte zur aktuellen Filterauswahl hinzuzufügen.

PriceFilter

Property	**Beschreibung**
range	Ist ein Objekt, das zwei numerische Eigenschaften enthält: *min* und *max* ({min: number; max: number}). Diese repräsentieren den minimalen und maximalen Wert des Preisfilter.
addFilterValues	Ist eine Funktion welche ein Objekt als Parameter akzeptiert. Dieses Objekt hat auch zwei numerische Eigenschaften: *min* und *max*. Die Funktion sollte aufgerufen werden wen die Range im Preisfilter zum Filter hinzugefügt werden sollen

ProductCard

Property	**Beschreibung**
product	Objekt vom Typ Product. Es enthält Details zu einem bestimmten Produkt, wie z.B. den Namen, die Beschreibung, den Preis usw.
addToShoppingCart	Diese Funktion akzeptiert zwei Parameter: productId (Produkt-ID) und price (Preis). Sie gibt ein Promise zurück, das sich auflöst, wenn das Produkt zum Warenkorb hinzugefügt wurde.

CategoryBreadcrumb

Property	**Beschreibung**
navigationPath	Das navigationPath Property beinhaltet ein Array aus CategoryNavigationEntry-Objekten, welche benötigt werden um einen Breadcrumb Pfad zu erstellen.

LanguageChanger

Property	**Beschreibung**
languageOptions	Dies ist ein Array von Objekten, wobei jedes Objekt zwei Eigenschaften hat: value und label. Beide sind vom Typ string. Der value beinhaltet den sprachen Code z.B. de für Deutsch. Das Label beinhaltet den Text, welcher im Shop angezeigt wird
currentLanguage	Dies ist ein string, der die aktuell ausgewählte Sprache des Benutzers darstellt.
handleLanguageChange	Dies ist eine Funktion, die einen string als Parameter akzeptiert. Diese Funktion kann aufgerufen werden, wenn der Benutzer eine neue Sprache auswählt, um die aktuelle Sprache zu ändern.

ContactSection

Property	Beschreibung
phoneNumber	Dies ist eine optionale Eigenschaft, die entweder einen string oder null sein kann. Es repräsentiert die Telefonnummer, die in einem Kontaktabschnitt angezeigt werden kann.
email	Dies ist ebenfalls eine optionale Eigenschaft, die entweder einen string oder null sein kann. Es repräsentiert die E-Mail-Adresse, die in einem Kontaktabschnitt angezeigt werden kann.

InformationSection

Property	Beschreibung
richtexts	Dies ist ein Array von RichText Objekten. Das RichText Objekt hat die Variablen `id`, `name`, `value`, `priority`, `active`, `createdAt` und `updatedAt`. Die Variable `name` beinhaltet den Titel des Textes und die Variable `value` beinhaltet den Richtext ansich.

AddressSection

Property	Beschreibung
children	Dies ist ein optionaler Knoten, der die Kinderkomponenten enthält, die in dieser Komponente gerendert werden können.
companyName	Dies ist ein optionaler String, der den Namen des Unternehmens darstellt.
street	Dies ist ein optionaler String, der den Strassennamen darstellt.
streetNumber	Dies ist ein optionaler String, der die Strassennummer darstellt.
zipCode	Dies ist ein optionaler String, der die Postleitzahl darstellt.
city	Dies ist ein optionaler String, der den Stadtnamen darstellt.
lat	Dies ist ein optionaler String, der den Breitengrad darstellt.
long	Dies ist ein optionaler String, der den Längengrad darstellt.

FooterContainer

Property	**Beschreibung**
AddressSection	Dieses Property beinhaltet die AddressSection. Es handelt sich dabei um eine React Komponente.
ContactSection	Dieses Property beinhaltet die ContactSection. Es handelt sich dabei um eine React Komponente.
InformationSection	Dieses Property beinhaltet die InformationSection. Es handelt sich dabei um eine React Komponente.
LanguageChanger	Dieses Property beinhaltet den LanguageChanger. Es handelt sich dabei um eine React Komponente.

FilterContainer

Property	**Beschreibung**
productNodes	Dieses Property beinhaltet eine React Komponente, welche das Aussehen der Produkte Darstellung z.B. (ProductCard) definiert
filterNodes	Dieses Property beinhaltet eine React Komponente, welche das Aussehen der Filter definiert.
subCategoryNodes	Dieses Property beinhaltet eine React Komponente, welche das aussehen der Unterkategorien Liste auf der Seite definiert.

ShopLayoutContainer

Property	**Beschreibung**
NavBar	Dieses Property beinhaltet die NavBar Komponente
CategoryNavigation	Dieses Property beinhaltet die CategoryNavigation Komponente
Page	Dieses Property beinhaltet den Page Inhalt
Footer	Dieses Property beinhaltet die Footer Komponente

NavBar

Property	Beschreibung
authenticatedUserData	Dies ist ein Objekt, welches die Benutzerdaten des authentifizierten Benutzers enthält.
logout	Dies ist eine Funktion, die keine Argumente akzeptiert und nichts zurückgibt. Sie wird verwendet, um den Benutzer auszuloggen.
shopLogoUrl	Dies ist ein String, der die URL zum Shop-Logo enthält.
navigation	Dies ist ein Array von ProfileNavigationItem Objekten. Jedes Objekt repräsentiert einen Navigationspunkt im Profilbereich.
itemAmount	Dies ist eine Zahl, welche die Anzahl der Artikel im Warenkorb darstellt.

CategoryNavigation

Property	Beschreibung
categories	Dies ist ein Array aus Objekten, welche die Daten zu den Kategorien beinhalten. Es bildet zudem den Kategorien Baum ab, welcher zum Navigieren verwendet werden kann.

ShoppingCart

Property	Beschreibung
cartItems	Dies ist ein Array von ShoppingCardUIItem Objekten. Jedes Objekt repräsentiert einen Artikel im Warenkorb.
removeCartItem	Dies ist eine Funktion, die eine ID (Nummer) als Argument akzeptiert und ein Promise zurückgibt, das sich auflöst, wenn der Artikel erfolgreich aus dem Warenkorb entfernt wurde.
updateCartItem	Dies ist eine Funktion, die vier Argumente akzeptiert: die ID des Artikels (Nummer), die Menge (Nummer), den Preis (Nummer) und die Produkt-ID (Nummer). Sie gibt ein Promise zurück, das sich auflöst, wenn der Artikel erfolgreich aktualisiert wurde.
createOrder	Dies ist eine Funktion, die ein OrderRequest Objekt als Argument akzeptiert und ein Promise zurückgibt, das sich auflöst, wenn die Bestellung erfolgreich erstellt wurde.
userData	Dies ist ein optionales Property, welches die Benutzerdaten enthält.

HomeLayoutContainer

Property	Beschreibung
products	Objekt vom Typ Product. Es enthält Details zu einem bestimmten Produkt, wie z.B. den Namen, die Beschreibung, den Preis usw.
ProductCard	Diese Property beinhaltet die ProductCard Komponente
addToCart	Diese Funktion akzeptiert drei Parameter: productId (Produkt-ID), quantity (Menge) und price (Preis). Sie gibt ein Promise zurück, das sich auflöst, wenn das Produkt zum Warenkorb hinzugefügt wurde.

OrdersContainer

Property	Beschreibung
orders	Das Property beinhaltet ein Array von OrderResponseUI Objekten, welche eine Bestellung repräsentieren.

SubCategoriesList

Property	Beschreibung
categories	Dies ist ein Array aus Objekten, welche die Daten zu den Kategorien beinhalten. Es bildet zudem den Kategorien Baum ab, welcher zum Navigieren verwendet werden kann.

Best Practice Entwicklung

Möchte man ein neues Theme entwickeln, kann man wie folgt vorgehen:

1. **Neuen Ordner erstellen:** Erstelle unter `packages/themes/src` ein neues Verzeichnis mit dem Namen deines Themes.
2. **Theme-Klasse erstellen:** Erstelle eine neue TypeScript-Datei, in der du eine Klasse mit dem Namen deines Themes und der Endung 'Theme' erstellst. Diese Klasse soll vom BaseTheme erben, wie oben beschrieben.
3. **Neue Theme-Klasse instanziieren:** Unter `packages/themes/` findest du eine Datei namens `index.ts`, in der ein Array exportiert wird. Instanziiere in diesem Array deine neu erstellte Klasse.
4. **Theme installieren:** Um dein neu erstelltes Theme besser entwickeln zu können, solltest du es nun im Webshop installieren.
5. **Theme aktivieren:** Hat die Installation geklappt, musst du das Theme noch aktivieren. Danach sollte dein neu erstelltes Theme genauso aussehen wie das Standard-Theme, da das Standard-Theme eine reine Implementierung des BaseTheme ist.
6. **Theme-Komponenten anpassen:** Waren alle vorherigen Schritte erfolgreich, kannst du nun beginnen, die einzelnen Theme-Komponenten anzupassen. Dazu kannst du einfach die Methode, welche die von dir angepassten Komponenten zurückgibt, in der von dir erstellten Klasse so überschreiben, dass sie die von dir erstellte/geänderte Komponente zurückgibt.

Tipp: Das Erstellen oder Anpassen eines Themes ist am einfachsten, wenn du die Komponente, die du anpassen möchtest, direkt vom Basistheme kopierst und dann Anpassungen vornimmst. Für eine gute Übersicht solltest du darauf achten, dass die Ordnerstruktur innerhalb deines Theme-Ordners genau gleich ist wie die im Basistheme. Achte auch immer darauf, dass die Importe in deiner Theme-Klasse korrekt sind und du keine Komponenten aus einem anderen Theme importierst.

B.5 Plugin Entwicklung

B.5.1 Aufbau

Damit das Kernsystem ein Plugin erkennt und verwendet, muss dieses das Interface `IPlugin` implementieren. Dieses Interface definiert die Metadaten für die Verwaltung im Adminbereich und stellt Lifecycle-Methoden bereit.

Der untenstehende Codeabschnitt zeigt das `IPlugin`-Interface. Bei der Entwicklung muss der Ersteller des Plugins alle Instanzvariablen selbst instanziieren, mit Ausnahme der Variablen `id` und `isActive`, die vom System instanziiert werden.

```
interface IPlugin {
    id: string;
    name: string;
    version: string;
    description: string;
    isActive: boolean;
    settings: PluginSettings;

    activate(): Promise<void>;
    deactivate(): Promise<void>;
    uninstall(): Promise<void>;
    update(settings: PluginSettings): Promise<void>;
}
```

Die folgende Tabelle erklärt die Funktion der Instanzvariablen

Variable	**Beschreibung**
name	Name des Plugins
creator	Name des Plugin-Erstellers
description	Beschreibung des Plugins
priority	Anzeigereihenfolge
onPluginActivation	Die Funktion wird aufgerufen, wenn das Plugin aktiviert wird
onPluginDeactivation	Die Funktion wird aufgerufen, wenn das Plugin deaktiviert wird

B.5.2 Seite integrieren

Ein Plugin bietet die Möglichkeit, eine eigene Seite in das Storefrontsystem zu integrieren. Im Unterschied zum Themening-System, das lediglich Presentational React-Komponenten verwendet, umfassen diese Seiten eigene Logik. Möchte man diese Funktionalität für sein Plugin nutzen, muss man sich zwischen zwei Interfaces entscheiden.

1. `IAdminPlugin`: Dient dazu, eine Seite im Administrationsbereich einzubinden. Der folgende Codeabschnitt zeigt das `IAdminPlugin`:

```
1  export interface IAdminPlugin extends IPlugin {
2  readonly endpoint: string;
3
4  insertAdminNavigation(navigationTree: AdminNavbarItem[]): AdminNavbarItem[];
5  getEntryPoint(): ({ service }: PluginPageProps) => Promise<JSX.Element>;
6  }
```

2. `IShopPlugin`: Dient dazu, eine Seite im Webshop einzubinden. Der folgende Codeabschnitt zeigt das `IShopPlugin`:

```
1  export interface IShopPlugin extends IPlugin {
2  readonly shopEndpoint: string;
3
4  getEntryPoint(): ({ service }: PluginPageProps) => Promise<JSX.Element>;
5  }
```

Die folgende Tabelle erklärt die Funktion der Instanzvariablen für beide Interfaces:

Variable	**Beschreibung**
endpoint	Slug, mit welchem das Plugin eindeutig über die URL identifiziert werden kann
getEntryPoint	Funktion, die die vom Entwickler erstellte Seite für dieses Plugin zurückgibt
insertAdminNavigation	Funktion, die einen neuen Navigationseintrag in der Administrationsnavigation hinzufügt

Wichtig: Die Seite, die von der `getEntryPoint`-Funktion zurückgegeben wird, muss das Property `services` vom Typ `PluginPageProps` akzeptieren, da sie sonst keine Dienste vom Kernsystem konsumieren kann.

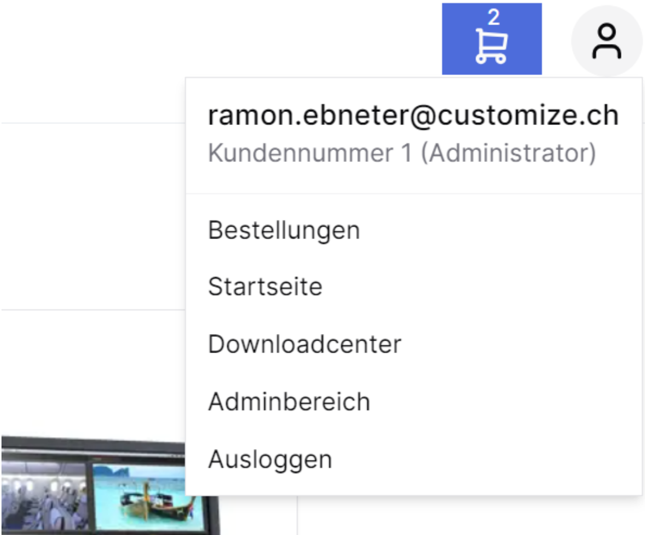
B.5.3 ShopPlugin Navigationeinträge

Im Gegensatz zum Adminplugin kann man im ShopPlugin Navigationseinträge in zwei unterschiedlichen Navigationsstrukturen einfügen.

1. **Navigationseintrag in der Produktklassierung:** Um, wie im untenstehenden Bild, einen Navigationseintrag in der Produktklassierung zu injezieren, muss das Plugin zusätzlich das `IShopNavigationPlugin` implementieren.

```
export interface IShopNavigationPlugin extends IPlugin {
 insertProductNavigation(navigationTree: Category[]): Category[];
}
```

2. **Navigationseintrag in der Kundenprofil Navigation:** Um, wie im untenstehenden Bild, einen Navigationseintrag in der Profilnavigation zu injezieren, muss das Plugin zusätzlich das `IShopProfilePlugin` implementieren.




```
export interface IShopProfilePlugin extends IPlugin {
 insertProfileNavigation(
  navigationTree: ProfileNavigationItem[]
 ): ProfileNavigationItem[];
}
```

B.5.4 Events

Plugins haben die Möglichkeit, sich über die Implementierung des Subscriber-Interfaces für die folgenden vier Ereignisse zu registrieren. Dazu muss der generische Typ des Subscriber-Interfaces mit einem der Eventtypen ersetzt werden:

1. **NewBasketItem:** Plugin wird benachrichtigt, wenn ein Kunde ein neues Produkt in den Einkaufskorb legt.

```
export interface NewBasketItem extends BaseEvent {
 productId: number;
 basketId: number;
 customerId: number;
}
```

2. **NewOrder:** Plugin wird benachrichtigt, wenn ein Kunde eine neue Bestellung tätigt.

```
export interface NewOrder extends BaseEvent {
 orderId: string;
}
```

3. **OpenedProduct:** Plugin wird benachrichtigt, wenn ein Kunde die Produkt-Detailseite öffnet.

```
export interface ProductOpened extends BaseEvent {
 openedProductId: string;
 customerId: number;
}
```

4. **OpenedClassification:** Plugin wird benachrichtigt, wenn ein Kunde eine Klassifizierung öffnet.

```
export interface ClassificationOpened extends BaseEvent {
 classificationId: number;
}
```

B.5.5 Services konsumieren

Ein Plugin konsumiert, wie oben bereits erwähnt, Services in Form eines Objekts des Typs `IPluginFacade` als Property. Die Methoden dieses Objekts können dann in der Plugin-Seite aufgerufen werden. Der folgende Codeausschnitt zeigt das `IPluginFacade` Interface.

```
export interface IPluginFacade {
  getProductById(id: string): Promise<Product>;
  getSetting(settingName: string): Promise<string>;
  createSetting(settingName: string): Promise<void>;
  removeSetting(settingName: string): Promise<void>;
}
```

Info: Bei den Settings handelt es sich um einen einfachen Key-Value-Store, in dem Daten in Form eines Strings gespeichert werden können. Daher ist es wichtig, dass der Name des Keys eindeutig ist, wenn man ein neues Setting erstellt.

Die Tabelle erklärt, wofür die einzelnen Methoden verwendet werden können.

Methode	**Nutzen**
createSetting	Erstellt einen neuen Settings-Eintrag in der Datenbank.
getSetting	Liest einen existierenden Settings-Eintrag aus der Datenbank aus.
removeSetting	Löscht einen existierenden Settings-Eintrag aus der Datenbank.
getProductById	Gibt ein Objekt des Typs Produkt zurück.

Zusätzlich zur `services`-Eigenschaft konsumiert die Plugin-Seite ein weiteres Property namens `updateSettings`. Dabei handelt es sich um eine Methode, die aufgerufen werden kann, um ein Setting zu aktualisieren.

```
updateSetting: (settingName: string, settingValue: string) => Promise<void>;
```

Anhang C

Aufgabenstellung

C.1 Ausgangslage

Abacus, der führende Anbieter von Enterprise Resource Planning (ERP) Software in der Schweiz, bietet eine integrierte Shop-Lösung namens 'Abashop' an, um die E-Commerce-Anforderungen seiner Kunden zu erfüllen. Diese Lösung, realisiert durch statisches HTML und JSP, ermöglichte es Unternehmen, nahtlos in ihre ERP-Systeme zu integrieren. Jedoch hat Abacus kürzlich angekündigt, den Support für 'Abashop' bis Ende 2025 einzustellen.

Diese Ankündigung stellt viele Unternehmen, insbesondere den Auftraggeber dieser Arbeit, die Firma Customize AG, einen Vertriebspartner von Abacus, vor die Herausforderung, geeignete alternative E-Commerce-Lösungen zu finden. In der vorangegangenen Studienarbeit „Proof of Concept: Integrierter Webshop für Abacus" wurde bereits ein Prototyp für eine mögliche Nachfolgelösung des AbaShops für die Kunden von Customize entwickelt, wobei das Open-Source E-Commerce-System 'nopCommerce' evaluiert und über einen entwickelten Connector für den automatisierten Austausch von Stammdaten mit dem ERP-System von Abacus verbunden wurde.

C.2 Ziele der Arbeit und Liefergegenstände

Das von nopCommerce bereitgestellte Frontend entspricht nicht den Anforderungen des Industriepartners. Daher soll das E-Commerce-System ausschliesslich als Headless-CMS verwendet werden. In einem ersten Schritt muss evaluiert werden, wie nopCommerce als Headless-CMS bereitgestellt werden kann, wobei ein kostenpflichtiges Plugin für eine RESTful HTTP-Schnittstelle in Betracht gezogen wird. Aufgrund einer möglichen Abhängigkeit von einem Drittanbieter in Bezug auf Bugfixes sowie Weiterentwicklungen wird jedoch vom Industriepartner eine eigene Schnittstellenlösung bevorzugt, die nach den spezifischen Anforderungen von Customize entwickelt werden kann.

Das Hauptziel dieser Arbeit besteht darin, durch einen Prototyp eine mögliche Zielarchitektur zu demonstrieren, die zeigt, wie auf Grundlage der Semesterarbeit ein eigenständiges Frontend mit einem JavaScript-Framework für den Webshop entwickelt werden kann. Der Industriepartner, welcher als Vertriebspartner von Abacus ERP tätig ist, betreut eine Vielzahl von Kunden, die einen AbaShop im Einsatz haben. Um die vielfältigen Anforderungen an Funktionalität und visuelles Design des Webshops zu erfüllen, muss das Basissystem flexibel erweiterbar sein.

Im Rahmen der Bachelorarbeit sollen die Anforderungen des Industriepartners an die Architektur des Webshops umfassend und aussagekräftig evaluiert werden. Der Schwerpunkt liegt auf der Erweiterbarkeit von Logik und Design im Webshop. Anhand von Beispielen werden konkrete Anforderungen der Kunden für eine Erweiterung dargestellt. Eine Mock-Implementation demonstriert mögliche Umsetzungen.

Die Lösungsarchitektur soll Möglichkeiten aufzeigen, wie Kunden oder Mitarbeiter des Industriepartners Erweiterungen in Funktionalität oder Anpassungen des visuellen Designs des Webshops vornehmen können, ohne den Source-Code des Kernsystems modifizieren zu müssen. Ziel der Bachelorarbeit ist es, einen Prototyp zu entwickeln, der eine mögliche Architektur für ein Webshop-System für Abacus auf der Basis von nopCommerce als CMS präsentiert. Um dieses Hauptziel zu erreichen, werden folgende

Liefergegenstände verfolgt:

- Anforderungen an das System: Die Anforderungen der Kunden des Industriepartners werden repräsentativ und aussagekräftig evaluiert und dargestellt.
- Architekturvorschlag: Auf der Basis der funktionalen und nichtfunktionalen Anforderungen an das Shopsystem soll eine geeignete Architektur, insbesondere für das Frontend, vorgeschlagen werden.
- Prototyp: Es wird ein Prototyp entwickelt, der die wichtigsten Use Cases abdeckt. Basierend auf den ermittelten Anforderungen wird eine geeignete Schnittstelle zwischen dem Webshop (Frontend) und dem CMS (nopCommerce) implementiert.
- Migrationsprozess: Es wird aufgezeigt, welche Schritte notwendig sind, um einen repräsentativen Customize-Kunden auf das neue Shopsystem zu migrieren.

Ergänzend zu den obigen Liefergegenständen, werden folgende kritischen Erfolgsfaktoren für diese Arbeit definiert:

- Der Prototyp wird auf Basis der Anforderungen eines realen oder fiktiven (aber repräsentativen) Customize-Kunden validiert.
- Die Erweiterungsmöglichkeiten des Shopsystems werden anhand eines solchen realen 'Beispiel'-Shops demonstriert.
- Bei der Entwicklung des Prototypen wird auf die Benutzbarkeit geachtet; diese wird durch einen 'User Test' (mit einem Consultant von Customize und/oder einem Endkunden) validiert und notwendige Verbesserungen für die Zukunft dokumentiert.
- Dem Industriepartner wird eine Benutzerdokumentation zur Verfügung gestellt, mit welcher Mitarbeiter (Consultants) in der Lage sind, einen Shop für einen Kunden aufzusetzen.
- Ausserdem wird auf die Erweiterbarkeit und Wartbarkeit des Systems geachtet. Bei der Entwicklung werden industrieübliche Prozesse eingesetzt und die an der OST gelernten Software Engineering Hygienefaktoren berücksichtigt (automatisierte Builds/Deployments, Tests, angemessene Versionierung mit Git, etc.).

Literaturverzeichnis

[1] David J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010. ISBN: 9780984521401.

[2] Angular. *Components*. [Online; Abgerufen am 01.04.2024]. URL: https://angular.dev/essentials/components.

[3] Michele Bertoli. *React Design Patterns and Best Practices*. Packt Publishing, 2017.

[4] C4Model. *C4 Architektur Modell*. [Online; Abgerufen am 01.05.2024]. URL: https://c4model.com/.

[5] Mike Cohn. *User Stories Applied: For Agile Software Development*. Boston: Addison-Wesley Professional, 2004. ISBN: 9780321205681.

[6] Angular Community. *Server-side rendering*. [Online; Abgerufen am 16.03.2024]. URL: https://angular.io/guide/ssr.

[7] Michael Duvigneau. *Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen*. Bd. 4. Logos Verlag Berlin GmbH, 2010.

[8] Ramon Ebneter. "Proof of Concept: Integrierte Webshoplösung für Abacus". Verfügbar unter: https://eprints.ost.ch/id/eprint/1169/. Semester Thesis. OST - Ostschweizer Fachhochschule, 2024.

[9] Eclipse. *Equinox: Mission Statement*. [Online; Abgerufen am 15.04.2024]. URL: https://eclipse.dev/equinox/.

[10] *Eclipse Plugin Development Environment Guide*. Eclipse Foundation. 2023. URL: https://help.eclipse.org/latest/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/plugin.html.

[11] Clemens Eichler. "Entwicklung einer Plug-In-Architektur für dynamische Komponenten". Diss. 2002.

[12] Eric Enge u. a. *The Art of SEO: Mastering Search Engine Optimization*. 3. Aufl. O'Reilly Media, 2015. ISBN: 9781491948965.

[13] Ben Farrell. *Web Components in Action*. Manning Publications, 2019. ISBN: 9781617295775.

[14] James D. Foley u. a. "Computer Graphics: Principles and Practice". In: 2. Aufl. Addison-Wesley Professional, 1996. Kap. Color Models, S. 592–596. ISBN: 9780201848403.

[15] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994. Kap. Structural Patterns, S. 185–194. ISBN: 978-0201633610.

[16] Michael Geers. *Micro Frontends in Action*. Manning Publications, 2020. ISBN: 9781617296871.

[17] Github. *Awesome Vite.js*. [Online; Abgerufen am 16.03.2024]. URL: https://github.com/vitejs/awesome-vite?tab=readme-ov-file\#ssr.

[18] Github. *NSwag: The Swagger/OpenAPI toolchain for .NET, ASP.NET Core and TypeScript.* [Online; Abgerufen am 02.03.2024]. URL: https://github.com/RicoSuter/NSwag.

[19] Google. *Google Lighthouse.* [Online; Abgerufen am 24.02.2024]. URL: https://chrome.google.com/webstore/detail/lighthouse.

[20] Keith J. Grant. "CSS In Depth". In: Chapter 2: Using CSS Variables for Theming. Shelter Island, NY: Manning Publications, 2018. Kap. 2.

[21] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston, MA: Addison-Wesley Professional, 2003. Kap. Messaging Channels, S. 113. ISBN: 978-0321200686.

[22] Dr. Lars Hupel. *What Does a Bundler Actually Do?* [Online; Abgerufen am 14.05.2024]. URL: https://www.innoq.com/en/articles/2021/12/what-does-a-bundler-actually-do/.

[23] Kasun Indrasiri und Danesh Kuruppu. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes.* O'Reilly Media, 2020. ISBN: 978-1492058335.

[24] Jack Jackson. *Github Issue: Module Federation Support.* [Online; Abgerufen am 20.03.2024]. URL: https://github.com/vercel/next.js/discussions/33327.

[25] Zack Jackson. *Module Federation For Next.js.* [Online; Abgerufen am 20.03.2024]. URL: https://www.npmjs.com/package/@module-federation/nextjs-mf.

[26] Martin Kleppmann. "Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems". In: O'Reilly Media, Inc., 2017. Kap. Dataflow Through Services: REST and RPC, S. 135–136.

[27] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Professional, 2000. ISBN: 9780201721638.

[28] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, 2004. ISBN: 9780131489066.

[29] Klaus Marquardt und Markus Völter. "Plugins: Anwendungsspezifische Komponenten". In: *Wirtschaftsinformatik* 2 (2003), S. 20–25.

[30] Microsoft. *gRPC services with ASP.NET Core.* [Online; Abgerufen am 02.03.2024]. URL: https://learn.microsoft.com/en-us/aspnet/core/grpc/aspnetcore.

[31] Next.js. *Documentation.* [Online; Abgerufen am 01.04.2024]. URL: https://nextjs.org/docs.

[32] Next.js. *Dynamic Routes.* [Online; Abgerufen am 15.05.2024]. URL: https://nextjs.org/docs/pages/building-your-application/routing/dynamic-routes.

[33] Next.js. *Lazy Loading.* [Online; Abgerufen am 20.03.2024]. URL: https://nextjs.org/docs/pages/building-your-application/optimizing/lazy-loading.

[34] nopCommerce. *nopCommerce Web API Plugin.* [Online; Abgerufen am 16.03.2024]. URL: https://www.nopcommerce.com/de/web-api.

[35] nopCommerce. *Why .NET developers choose nopCommerce for eCommerce development.* [Online; Abgerufen am 05.03.2024]. URL: https://www.nopcommerce.com/de/why-for-developers.

[36] npm. *Semantic Versioning.* [Online; Abgerufen am 01.06.2024]. URL: https://docs.npmjs.com/about-semantic-versioning.

[37] inc. npm. *About npm.* [Online; Abgerufen am 16.03.2024]. URL: https://docs.npmjs.com/about-npm.

[38] pnpm. *pnpm Link.* [Online; Abgerufen am 01.06.2024]. URL: https://pnpm.io/cli/link.

[39] Rachel Potvin und Josh Levenberg. "Why Google Stores Billions of Lines of Code in a Single Repository". In: *Communications of the ACM* 59.7 (2016), S. 78–87.

[40] Prisma. *Github Issue: Run migrations by code.* [Online; Abgerufen am 15.04.2024]. URL: https://github.com/prisma/prisma/issues/13549.

[41] Prisma. *Github Issue: Support for splitting Prisma schema into multiple files.* [Online; Abgerufen am 15.04.2024]. URL: https://github.com/prisma/prisma/issues/2377.

[42] React. *Start a New React Project.* [Online; Abgerufen am 16.03.2024]. URL: https://react.dev/learn/start-a-new-react-project.

[43] React. *Writing Markup with JSX.* [Online; Abgerufen am 16.03.2024]. URL: https://react.dev/learn/writing-markup-with-jsx.

[44] Seobility. *URL Slug.* [Online; Abgerufen am 17.02.2024]. URL: https://www.seobility.net/de/wiki/URL_Slug.

[45] Shopify. *Headless Commerce: Der neue Trend für Onlineshops.* [Online; Abgerufen am 29.02.2024]. URL: https://www.shopify.com/de/blog/headless-commerce.

[46] Storybook. [Online; Abgerufen am 15.05.2024]. URL: https://storybook.js.org/.

[47] Swagger. *OpenAPI Specification.* [Online; Abgerufen am 02.03.2024]. URL: https://swagger.io/specification/.

[48] Fielding Roy Thomas. "Representational State Transfer (REST)". In: *Architectural Styles and the Design of Network-based Software Architectures* (2000).

[49] Edward R. Tufte. *The Visual Display of Quantitative Information.* Cheshire, Connecticut: Graphics Press, 2001. ISBN: 978-0961392147.

[50] Turbo. *What is a Monorepo?* [Online; Abgerufen am 12.05.2024]. URL: https://turbo.build/repo/docs/handbook/what-is-a-monorepo.

[51] Material UI. *Theming.* [Online; Abgerufen am 15.04.2024]. URL: https://mui.com/material-ui/customization/theming/.

[52] Vite. *Vite: Server-Side Rendering (SSR).* [Online; Abgerufen am 16.03.2024]. URL: https://vitejs.dev/guide/ssr.html.

[53] Vue.js. *Documentation.* [Online; Abgerufen am 01.04.2024]. URL: https://vuejs.org/guide/introduction.html.

[54] webpack. *Code Splitting.* [Online; Abgerufen am 15.04.2024]. URL: https://webpack.js.org/guides/code-splitting/.

[55] Webpack. *Module Federation*. [Online; Abgerufen am 01.05.2024]. URL: https://webpack.js.org/concepts/module-federation/.

[56] WordPress. *Theming*. [Online; Abgerufen am 17.04.2024]. URL: https://developer.wordpress.org/themes/getting-started/what-is-a-theme/.

[57] WordPress Contributors. *Plugin Developer Handbook*. WordPress Foundation. 2024. URL: https://developer.wordpress.org/plugins/.

[58] O. Zimmermann. “Architectural refactoring for the cloud: a decision-centric view on cloud migration”. In: *Computing* 99 (2017), S. 129–145. URL: https://doi.org/10.1007/s00607-016-0520-y.

[59] Olaf Zimmermann u. a. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley Signature Series (Vernon). Addison-Wesley Professional, 2022. ISBN: 9780137670109.

Abkürzungsverzeichnis

API Application Programming Interface. 39, 46, 48, 53, 61, 62, 87, 99, 167

CD Continuous Delivery. 20

CMS Content Management System. 39

CRA Create React App. 61

CRM Customer Relationship Management. 9

CSS Cascading Style Sheets. 5, 25, 41, 60, 62, 63, 73, 75–77, 82, 169

ERP Enterprise Resource Planning. iii, 1, 4–6, 8, 9, 12, 13, 15, 16, 19, 21, 24, 34, 35, 44, 49, 54, 85, 99, 101, 112, 117

gRPC Google Remote Procedure Call. 44–48, 167

HSL Hue, Saturation, Lightness. 75

HTML Hypertext Markup Language. 4, 5, 18, 25, 41, 60–62, 73, 79, 80, 112

HTTP Hypertext Transfer Protocol. iii, 1–4, 34, 44, 46–49, 51–53, 57, 167

IDE Integrated Development Environment. 37, 38

JSON JavaScript Object Notation. 44, 46, 47, 52, 53, 99

JSP Java Server Pages. 4, 73

JSX JavaScript XML. 61, 99

MDX Markdown + JSX. 99

MUI Material-UI. 40, 41

MVP Minimum Viable Product. 14, 33

NFA Non-Functional Requirement. 34, 44, 59, 73, 109, 110, 112

npm Node Package Manager. 59, 67–72, 110, 113–115, 117

OAS OpenAPI Specification. 46, 48

SEO Search Engine Optimization. 35, 60, 111, 168

SPA Single Page Application. 60, 61

SSG Static Site Generation. 61

SSR Server Side Rendering. 60–62

SuD System under Development. 6, 7, 44

UC Use Case. 25, 27, 29–33, 87, 107, 108

UI User Interface. 40, 116

UML Unified Modeling Language. 23, 24, 28, 56, 73, 82, 83, 89, 90, 93, 95, 167, 168

URL Uniform Resource Locator. 29, 46, 57

US User Story. 17–22, 24, 25, 28, 30, 63, 75, 105–107

XML eXtensible Markup Language. 46

Abbildungsverzeichnis

2.1 Kontextdiagramm Studienarbeit (C4) [8] 5
2.2 Kontextdiagramm Zielarchitektur der Bachelorarbeit (C4) 7
2.3 Projektplan Ablösung 'AbaShop' 8
2.4 Übersicht Applikationen Abacus [8] 9
2.5 Domänenmodell Abacus 10
2.6 Beispiel Klassierung in Abacus 11

3.1 Wireframe Filter 18
3.2 Use Cases des Theming-Systems (UML Use Case Diagramm) 24
3.3 Wireframe Darstellung Klassierung oben 26
3.4 Wireframe Darstellung Klassierung links 26
3.5 Use Cases des Plugin Systems (UML Use Case Diagramm) 28
3.6 Wireframe Seitenelement 29

5.1 Kontextdiagramm Gesamtsystem (C4) 43
5.2 Funktionsweise von gRPC 45
5.3 Boxplot Performance gRPC vs. RESTful HTTP 47
5.4 Containerdiagramm Abacus Connector (C4) [8] 49
5.5 Containerdiagramm nopCommerce (C4) 50
5.6 Dokumentation Webapi Plugin (Openapi) 51
5.7 Konfiguration Secret-Key Webapi Plugin 52
5.8 API-Version Auswahl in Swagger 53
5.9 Konfiguration Abacus Connector 55
5.10 Ablauf 'Preis-Cache' (UML-Sequenzdiagramm) 56
5.11 Architektur: Komponentendiagramm nopCommerce (C4) 58
5.12 Wöchentliche Downloadzahlen der evaluierten JavaScript Frameworks (npm) 59
5.13 Konzept von Module Federation 65
5.14 Struktur des Monorepos (C4-Containerdiagramm) 71
5.15 Vergleich Produktdetail 'Standardtheme' vs. 'Moderntheme' 72
5.16 Container/Representational Pattern (UML-Komponentendiagramm) 73
5.17 Anpassen Farben vom Theme im Adminbereich 78
5.18 Aufbau Theming-System (UML-Klassendiagramm) 82
5.19 Plugin: Seitenelement injizieren (UML-Klassendiagramm) 89
5.20 Plugin: Services konsumieren (UML-Klassendiagramm) 90

5.21 Navigation der Produktklassierung 91
5.22 Navigation im Shopperprofil 91
5.23 Navigation im Adminbereich 92
5.24 Plugin: Navigationselement einfügen (UML-Klassendiagramm) 93
5.25 Event-System mit dem Publish-Subscriber Pattern 94
5.26 Plugin: Event-System (UML-Klassendiagramm) 95
5.27 Komponentendiagramm des Storefrontsystem (C4) 98
5.28 Projektstruktur GitLab 100

6.1 Performance-Analyse durch Google Lighthouse [19] 109
6.2 Boxplot des Performancetests vom Storefrontsystem 110
6.3 SEO-Analyse durch Google Lighthouse [19] 111
6.4 GitLab-Struktur des alternativen Entwicklungsprozesses 114
6.5 Theme-Dokumentation mit Storybook 116

B.1 Ausschnitt Dev-Docs 129

Tabellenverzeichnis

3.1 Priorisierungschema Anforderungen . . . 14
3.2 Akteure . . . 16
3.3 Zusätzlicher Akteur Use Cases . . . 23
3.4 Priorisierung Anforderungen Theming-System . . . 27
3.5 Priorisierung Anforderungen Plugin System . . . 33

5.1 Antwortzeiten Webapi Endpoint */products* . . . 54
5.2 Antwortzeiten Webapi Endpoint */products* mit 'Preis-Cache' . . . 57
5.3 Übersicht verfügbare CSS-Variablen im Theming-System . . . 76

6.1 Bewertung der kritischen Erfolgsfaktoren . . . 104
6.2 Bewertung der Kernfunktionalitäten . . . 105
6.3 Bewertung der Konfigurationsanforderungen . . . 107
6.4 Use Cases des Plugin-Systems . . . 108