HTW D Hochschule für Technik und Wirtschaft Dresden University of Applied Sciences

Bachelorarbeit

Konzeption und Realisierung einer komplexen individuellen Erweiterung für einen Shopware-6-Shop

vorgelegt von:	Josy Vinnemeier
Studienbereich:	Medieninformatik
Ort:	HTW Dresden
Matrikelnummer:	50392
Erstgutachter:	Prof. Dr.-Ing. Jörg Vogt
Zweitgutachter:	Dipl.-Kfm. Martin Ritter

Abgabedatum:	13.05.2025

© 2025

Dieses Werk ist lizenziert unter einer Creative Commons „Namensnennung 4.0 International“ Lizenz.



Inhaltsverzeichnis

Abkürzungsverzeichnis **III**

Glossar **IV**

Abbildungsverzeichnis **V**

Tabellenverzeichnis **VI**

1 Einleitung **1**
1.1 Hintergrund und Relevanz . . . 1
1.2 Zielsetzung der Arbeit . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Theoretische Grundlagen **3**
2.1 Shopware 6 . . . 3
2.1.1 Technologie-Stack . . . 5
2.1.2 Erweiterungen . . . 5
2.1.3 Plugin-Lifecycle . . . 7
2.1.4 Aufbau und Verzeichnisstruktur . . . 8
2.2 Industrie 4.0 . . . 9
2.2.1 E-Commerce und Produktindividualisierung . . . 9
2.2.2 Mass Customization . . . 10
2.3 Produktkonfiguratoren . . . 11
2.3.1 Entwicklungsstrategien . . . 12
2.3.2 Analyse existierender Produktkonfiguratoren . . . 13
2.3.3 Meistgenutzten Arten . . . 14

3 Implementierung **16**
3.1 Anforderungsanalyse . . . 16
3.1.1 Stakeholder . . . 16
3.1.2 Funktionale Anforderungen . . . 18
3.1.3 Nicht-funktionale Anforderungen . . . 19
3.1.4 Priorisierung (MoSCoW) . . . 19
3.2 Konzeption . . . 20
3.2.1 Datenmodellierung . . . 20
3.2.2 Systemarchitektur/Architekturkonzept . . . 23
3.2.3 Technische Lösungsansätze . . . 24

3.3 Entwicklungsumgebung 28
3.4 Projektstruktur und dessen Bestandteile 30
3.5 Entwicklung 30
3.5.1 Custom Fields 31
3.5.2 Subscriber 32
3.5.3 Frontend-Implementation 33
3.5.4 Warenkorb-Integration 37
3.5.5 Sicherheitsimplementierung 39
3.5.6 Optimierung und Finalisierung 42
3.6 Testing und Qualitätssicherung 43

4 Ergebnisse und Zukunftsaussichten 45
4.1 Zusammenfassung der Ergebnisse 45
4.1.1 Erreichte Funktionalität 45
4.1.2 Benutzerfreundlichkeit 46
4.1.3 Barrierefreiheit 46
4.1.4 Wartung und Pflege des Plugins 47
4.1.5 Limitationen der aktuellen Implementierung 47
4.1.6 Weiterentwicklung 49
4.2 Relevante Enwticklungen 50

Literatur 51

A Anhang 1

Erklärung

Abkürzungsverzeichnis

ATO	Assemble to Order
CTO	Configure to Order
ETO	Engineer to Order
MTO	Make to Order
DAL	Data Abstraction Layer
CMS	Content-Management-System
SaaS	Software-as-a-Service
SPA	Single-Page-Anwendung
API	Application Programming Interface
REST	Representational State Transfer
CODP	Customer order decoupling point
TTI	Time-to-Interactive
SOA	Serviceorientierte Architektur

Glossar

Breaking Changes

: Beschreibt eine Änderung an einer Software oder ähnlichem, wodurch die Anwendung oder Teile davon, nicht mehr funktionieren, also umgangssprachlich 'kaputt gehen'.. 8, 47

Erlebniswelten

: Beschreibt die Funktionalität über die man in Shopware 6 die Content-Seiten modular gestalten kann. Sie ist zu vergleichen mit gängigen CMS-Anwendungen.. 42

Single Page Application

: Bezeichnnet eine Web-Anwendung, bei der nur eine HTML-Seite zum Start geladen wird. Danach werden die Inhalte nur aktualisiert, aber nicht neu geladen.. 15

Time-to-Interactive

: Gibt an wie lange es dauert bis ein Endnutzer mit einer aufgerufenden Website interagieren kann.. 14

Abbildungsverzeichnis

2.1 Prozentuale Verteilung der im Mai 2025 meistgenutzten E-Commerce-Plattformen auf deutschen Webseiten 3
2.2 Shopware 6 Vergleich möglicher Erweiterungen 6
2.3 Shopware 6 Plugin Lifecycle 7
2.4 Platzierung des CODP für die jeweiligen Konzepte. 11
3.1 Konzeptuelles Datenmodell des Palettenkonfigurators auf Basis von Custom Fields 21
3.2 Line-Item-Hierarchie im Warenkorb mit bis zu acht Nested Line Items 22
3.3 Ablaufdiagramm des Palettenkonfigurators 23
3.4 Custom Field Ansicht bzw. Bearbeitung im Admin Bereich 27
3.5 Chrome-Entwicklertools 29
3.6 Custom Fields Übersicht im Produkt anhand des Konfigurators . . . 32
3.7 Visuelle Darstellung des Palettenkonfigurators in der Storefront . . . 34
3.8 Beispielhafte Optionen im Select 35
3.9 Beispiel für Regelung zur Blockung des „Single-Type“ 37
3.10 Warenkorb Storefront Übersicht 39
3.11 Warenkorb Fehlermeldung 40
3.12 Konfigurator mit ausgewählten Optionen ohne Checkbox 41
3.13 ChromeDev Netzwerkübersicht Route Add 44

Tabellenverzeichnis

1 Vergleich der Hauptentwicklungsstrategien bei Produktkonfiguratoren 13
2 Funktionale Anforderungen 18
3 Nicht-funktionale Anforderungen 19
4 Priorisierung der funktionalen und nicht-funktionalen Anforderungen nach dem MoSCoW-Modell 20

1 Einleitung

1.1 Hintergrund und Relevanz

In der heutigen Zeit, die von der fortschreitenden Digitalisierung geprägt ist, zeigt der E-Commerce-Sektor ein stetiges Wachstum. Die COVID-19-Pandemie hat diesen Trend weiter verstärkt und das Konsumverhalten nachhaltig verändert. Statt in Geschäfte zu gehen, rufen Kunden heutzutage einen Online-Shop auf und bestellen die gewünschten Produkte bequem von der heimischen Couch, was zu einem signifikanten Anstieg der Online-Käufe führte [35]. Die Gründe hierfür sind vielfältig und leicht nachzuvollziehen.

Um den wachsenden Bedarf zu erfüllen, wurden Anwendungen entwickelt, die die Erstellung und Verwaltung von Online-Shops vereinfachen. Im Kern handelt es sich dabei um Content-Management-Systeme (CMS), die den Fokus auf E-Commerce setzen. Diese Systeme ermöglichen es Unternehmen, ihre Online-Shops effizient zu erstellen und zu verwalten.

Gleichzeitig wächst das Bedürfnis der Verbraucher nach individualisierten Produkten und Dienstleistungen. Diese Entwicklungen stellen den Online-Handel vor neue Herausforderungen und erfordern neue Lösungsansätze. Produktkonfiguratoren ermöglichen es den Benutzern, Produkte interaktiv und nach ihren Wünschen anzupassen. Sie fungieren als Schnittstelle zwischen dem Wunsch nach Individualisierung und der technischen Umsetzung im digitalen Raum [39, 1].

1.2 Zielsetzung der Arbeit

Ziel dieser Bachelorarbeit ist die Konzeption und prototypische Realisierung einer komplexen, individuell zugeschnittenen Erweiterung innerhalb des E-Commerce-Systems Shopware 6. Diese Erweiterung wird in Form eines Palettenkonfigurators umgesetzt, der es Endkunden ermöglichen soll, ein Produkt gemäß individueller Anforderungen zu konfigurieren.

Die Umsetzung erfolgt vor dem Hintergrund bestehender Limitationen innerhalb von Shopware 6. Zwar bietet das System grundlegende Mechanismen zur Handhabung von Produktvarianten, sowie ein kommerzielles Plugin zur Erweiterung der Konfigurationsmöglichkeiten, jedoch stoßen beide Ansätze an funktionale Grenzen.

Insbesondere, wenn es um die Kombination mehrerer Produkte zu einem Gesamtprodukt geht. Bestehende Web-Konfiguratoren fokussieren sich typischerweise auf die Modifikation einzelner Produkteigenschaften, wie beispielsweise im Automobilbereich[1], aber weniger auf die Kombination mehrerer Produkte.

1.3 Aufbau der Arbeit

Zur Realisierung der Zielsetzung werden zuerst die theoretischen und technischen Grundlagen analysiert, um eine fundierte Basis zu schaffen. Darauf aufbauend wird eine systematische Anforderungsanalyse der funktionalen und technischen Rahmenbedingungen der geplanten Erweiterung erarbeitet. Korrelierend dazu wird ein Konzept für eine benutzerfreundliche und in Shopware 6 integrierbare Lösung erarbeitet. Auf den vorherigen Schritten basierend erfolgt die Implementierung unter Berücksichtigung der Architektur von Shopware 6. Zum Schluss wird die Lösung evaluiert und ein Blick auf potenzielle Weiterentwicklungen geworfen.

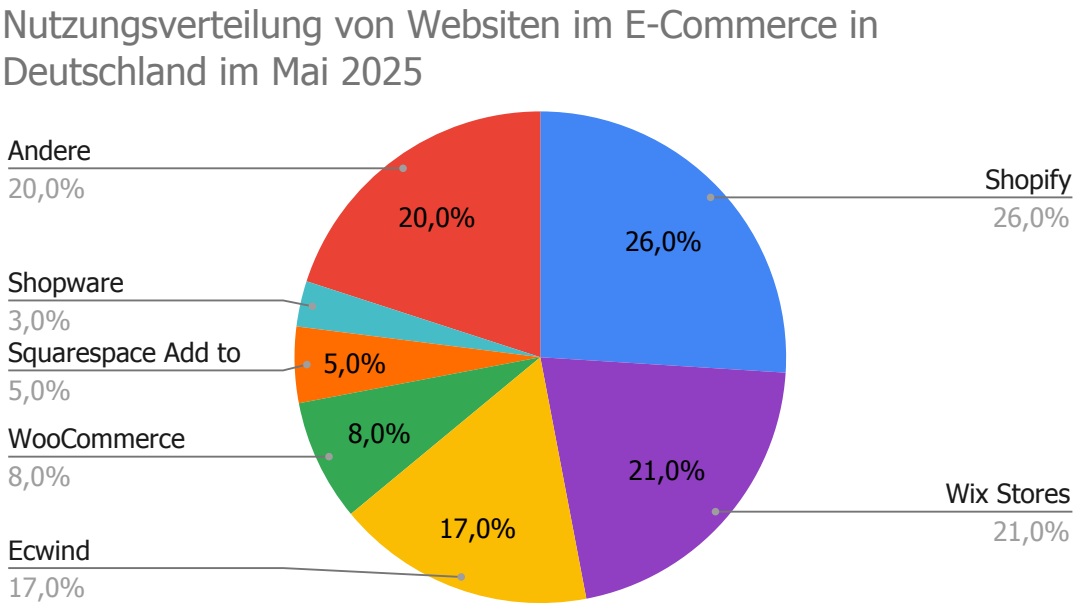
[1] Bsp: `https://configure.bmw.de/de_DE/configure/G80`

2 Theoretische Grundlagen

Im folgenden Abschnitt soll eine Übersicht über die aktuelle Situation und für diese Arbeit relevante Themen gegeben werden. Dabei sollen sowohl die Technologien als auch spezifische zu betrachtende entscheidende Faktoren näher analysiert werden.

2.1 Shopware 6

Die Shopware AG wurde 2008 von den Brüdern Stefan und Sebastian Hamann offiziell eingetragen, nachdem sie den Namen ihres erstellten Produktes zu ihrer Unternehmensidentität machen wollten. Mit der Veröffentlichung einer Early-Access-Version von Shopware 6 im Mai 2019 vollzog das Unternehmen dann einen strategischen Wechsel hin zu einer API-first- und Headless-Commerce-Architektur. [19]



Quelle: https://trends.builtwith.com/shop/country/Germany [abgerufen am 05.05.2025]

Abbildung 2.1: Prozentuale Verteilung der im Mai 2025 meistgenutzten E-Commerce-Plattformen auf deutschen Webseiten
Quelle: https://trends.builtwith.com/shop/country/Germany[4]

Zum Zeitpunkt dieser Arbeit verdeutlicht jedoch die Abbildung 2.1, dass Shopify im deutschen Markt eine dominante Stellung einnimmt, während Shopware, trotz seiner regionalen Verwurzelung und Open-Source-Strategie, einen deutlich kleineren

Marktanteil besitzt. Für diese Situation lassen sich, neben anderen Gründen, im Wesentlichen drei technisch geprägte Ursachen herausarbeiten.

Erstens unterscheidet sich das **Betriebsmodell** beider Plattformen grundlegend: Shopify wird als vollständig verwaltete *Software-as-a-Service(SaaS)* bereitgestellt, sodass Hosting, Wartung und Updates im monatlichen Abonnement enthalten sind. Shopware 6 ist hingegen self-hosted, erfordert also den Betrieb einer eigenen Serverinfrastruktur. Dadurch entstehen zusätzliche Administrationsaufwände und demzufolge damit verbundene Kosten [31]. Dieser Grund verliert jedoch potenziell an Relevanz, da Shopware mittlerweile auch eine SaaS-Lösung anbietet, aber zur Einteilung der bisherigen Entwicklung ist dieser Punkt trotzdem von Bedeutung.

Zweitens profitiert Shopify von einem deutlich **umfassenderen Store für Erweiterungen**: Der offizielle App-Store bietet laut Herstellerangaben mehr als „8000“[17] Anwendungen, während der Shopware Store gegenwärtig über 3000 Plugins listet [33]. Das reduziert den Bedarf an Individualentwicklung bei Shopify merklich.

Drittens bestehen für Bestandskunden signifikante **Migrationshürden**. Ein Wechsel von Shopware 5 auf die technisch neu konzipierte Version 6 erfordert die vollständige Neuerstellung von Themes und Plugins, da die bisherigen Erweiterungen inkompatibel sind. Es existieren zwar Tools, um eine Migration zu erleichtern[1], aber die weiterhin benötigte Re-Implementierung kostet Zeit und Ressourcen, sodass einige Händler auf alternative SaaS-Plattformen ausweichen.

Trotz der beschriebenen Situation ist Shopware 6 weiterhin attraktiv. Vor allem aufgrund der Open-Source-Strategie, die eine kostenfreie Nutzung ohne zwingendes Abonnement ermöglicht. Verfügt ein Händler über die notwendigen technischen Kompetenzen oder arbeitet er mit einer spezialisierten Agentur zusammen, entsteht ein stabiles und weitgehend unabhängiges System mit umfangreichen Anpassungsmöglichkeiten. Ferner bietet die selbst betriebene Serverinfrastruktur spezifische Vorteile für große Händler mit hohem Datenaufkommen oder besonderen Performance-Anforderungen. Diese können ihre Infrastruktur gezielt skalieren, eigene Sicherheitsrichtlinien umsetzen sowie individuelle Anforderungen an Hosting und Deployment berücksichtigen, was bei standardisierten SaaS-Lösungen häufig nur eingeschränkt möglich ist.

[1] Bsp.: `https://store.shopware.com/de/swag257162657297f/migrations-assistent.html`

2.1.1 Technologie-Stack

Im Kern basiert Shopware 6 auf dem Framework Symfony und die interne Struktur folgt dementsprechend dem Model-View-Controller. Daraus folgt, dass ein Entwickler PHP, JavaScript, Twig und SCSS/**sass!** beherrschen sollte, um den Shopware-Code gut zu verstehen [30]. Für die Anpassung oder Erweiterung von Datenbankschemata wird noch Wissen in SQL benötigt. Wie bereits im vorherigen Abschnitt erwähnt, verfolgt die Architektur einen API-first- und Headless-Commerce-Ansatz und ermöglicht damit eine klare Trennung zwischen Backend- und Frontend-Logik.

Für das Administrations-Interface kommt Vue.js 3 zum Einsatz, um reaktive SPA bereitzustellen. Die Storefront hingegen wird serverseitig über Twig gerendert, was eine klare Trennung von Logik und Darstellung ermöglicht. [30, 23]

Im Backend setzt Shopware 6 auf PHP als primäre Programmiersprache. Die Implementierung folgt dabei dem Symfony-Framework, welches bewährte Strukturen für Event-Handling, Dependency Injection und die Modularisierung der Anwendung bereitstellt. Durch den Einsatz von Symfony wird die Entwicklung von Erweiterungen standardisiert und erleichtert.

Für die Datenhaltung verwendet Shopware 6 relationale Datenbanken wie MySQL oder MariaDB [31]. Der Zugriff erfolgt jedoch über das sogenannte Data Abstraction Layer (DAL), mit dem eine konsistente, objektorientierte Schnittstelle für den Zugriff auf Entitäten wie Produkte oder Bestellungen bereitgestellt wird. [26]

Der gesamte Stack ist darauf ausgelegt, Erweiterungen möglichst reibungslos zu integrieren. Die standardisierte Erweiterungsarchitektur stellt klar definierte Schnittstellen, auch APIs (REST-basiert), bereit, sodass neue Funktionalitäten ergänzt oder bestehende Dienste überschrieben werden können, ohne den Software-Code zu modifizieren. Die modulare Architektur erleichtert Drittentwicklern den Einstieg und stärkt die Erweiterbarkeit des Systems. [27]

2.1.2 Erweiterungen

In Shopware 6 stehen drei Erweiterungsarten zur Verfügung (vgl. Abb. 2.2)[27]:

Plugins, die als Symfony-Bundles in PHP implementiert werden und eigene Services, Controller oder Event-Subscriber bereitstellen, um Funktionalitäten im Backend wie im Frontend zu ergänzen oder zu überschreiben.

Themes, die technisch ebenfalls als Plugins oder Apps realisiert sind, jedoch ausschließlich zur Anpassung der visuellen Gestaltung der Storefront dienen und keine PHP-Logik einführen. Sie nutzen SCSS/CSS sowie Twig-Konfigurationen, um Farben, Fonts und Layouts anzupassen.

Apps, die als eigenständige, API-basierte Erweiterungen externe Dienste anbinden. Sie werden nicht als PHP-Bundles im Kern installiert, sondern nutzen das Shopware-App-System mit OAuth-Authentifizierung, um über die Store- und Admin-API auf Shop-Daten zuzugreifen und diese zu erweitern.

Task	Plugin	Theme	App	Remarks
Change Storefront appearance	✅	✅	✅	
Add admin modules	✅	❌	✅	
Execute Webhooks	✅	❌	✅	Apps main functionality is to call Webhooks, but Plugins can be implemented to do that as well.
Add custom entities	✅	❌	✅	
Modify database structure	✅	❌	❌	
Integrate payment providers	✅	❌	✅	
Publish in the Shopware Store	✅	✅	✅	
Install in Shopware 6 Cloud Shops	❌	❌	✅	
Install in Shopware 6 self-hosted Shops	✅	✅	✅	Apps can be installed and used since Shopware 6.4.0.0
Add custom logic/routes/commands	✅	❌	✅	Apps extract functionalities/logic into separate services, so technically, they can add custom logic
Control order of style/template inheritance	❌	✅	✅	

Abbildung 2.2: Shopware 6 Vergleich möglicher Erweiterungen

Quelle: https://developer.shopware.com/docs/guides/plugins/overview.html[27] [Abruf: 09.05.2025]

Diese bilden das Fundament für die Erweiterbarkeit der Shopware-6-Plattform und ermöglichen die flexible Anpassung des Systems an individuelle Geschäftsanforderungen. Dieser Abschnitt fokussiert sich jedoch nur auf Plugins.

Shopware 6 implementiert ein modulares Erweiterungskonzept, welches auf einer serviceorientierten Architektur (SOA) basiert und als eine Erweiterung des zugrunde liegenden Symfony-Frameworks bezeichnet werden kann. Diese Architektur gewährleistet eine lose Kopplung zwischen Kernfunktionalitäten und Erweiterungen, wodurch die Systemstabilität bei gleichzeitiger Flexibilität sichergestellt wird.

2.1.3 Plugin-Lifecycle

Der Plugin-Lebenszyklus in Shopware 6 gliedert sich in vier zentrale Phasen: Installation, Aktivierung, Deaktivierung und Deinstallation (vgl. Abb. 2.3). Jede dieser Phasen wird durch spezifische Methoden repräsentiert, die im Plugin überschrieben und individuell erweitert werden können. Neben typischen Aufgaben wie dem Anlegen oder Entfernen von Datenbankstrukturen bieten diese Methoden auch Möglichkeiten zur Verwaltung von Custom Fields.

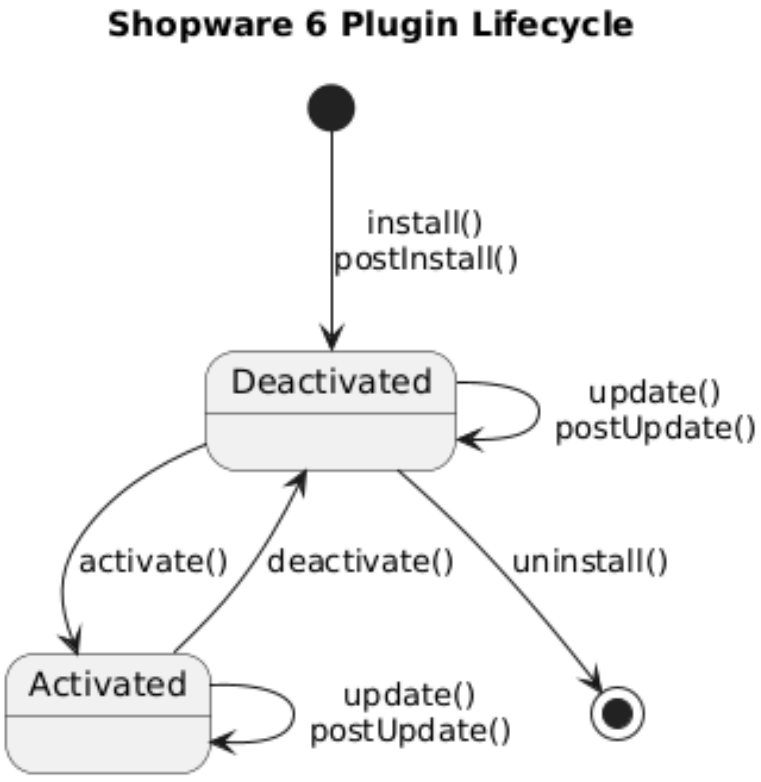


Abbildung 2.3: Shopware 6 Plugin Lifecycle
Quelle: Eigene Darstellung generiert mit https://plantuml.com/de/

Nach dem Hochladen oder Hinzufügen über den Shopware-6-Store erfolgt zunächst die Installation, bei der das Plugin in den Zustand „deaktiviert“ übergeht. Anschließend kann es aktiviert werden, wodurch die vom Plugin bereitgestellten Funktio-

nen im Shop verfügbar werden. Ein Plugin kann in allen Zuständen (bis auf bei der Deinstallation) aktualisiert werden. Zusätzlich zu den Hauptmethoden stellt Shopware 6 weitere Lifecycle-Methoden wie `postInstall()` und `postUpdate()` bereit, mit denen nachgelagerte Operationen ausgeführt werden können. Das Lifecycle-Management gewährleistet somit eine kontrollierte Integration und Entfernung von Plugins. [29]

2.1.4 Aufbau und Verzeichnisstruktur

Die Plugin-Architektur von Shopware 6 basiert auf einem modularen Aufbau, der durch eine einheitliche Verzeichnisstruktur unterstützt wird. Funktionale Komponenten eines Plugins werden typischerweise in Unterverzeichnissen wie Service, Controller und Resources organisiert. Diese klare Trennung der Verantwortlichkeiten verbessert sowohl die Übersichtlichkeit als auch die Wartbarkeit der Codebasis und damit auch die Entwicklung [28].

Plugins werden im Verzeichnis custom/plugins abgelegt, wo sie von Shopware automatisch erkannt und geladen werden. Innerhalb eines Plugins wird die Ordnerstruktur von Shopware 6 weitgehend nachgebildet, um Framework-Funktionen, wie z.B. das Event-System und die Dependency-Injection, nahtlos zu nutzen. Dadurch sinkt der Entwicklungsaufwand, und Dritte können den Quellcode schneller nachvollziehen oder erweitern [28].

Für die langfristige Wartbarkeit und Kompatibilität der Plugins ist ein konsequentes Versionsmanagement von zentraler Bedeutung. Shopware 6 orientiert sich an der Semantischen Versionierung[1], bei der jede Versionsnummer die Art der Änderung (kompatibel, inkrementell oder brechend) reflektiert. Plugins sollten daher exakt angeben, welche Shopware-Versionen unterstützt werden, und auch selbst die Versionierung anwenden. Bei Major-Releases können außerdem Breaking Changes auftreten, die eine Anpassung des Quellcodes notwendig machen.

Insgesamt bietet die Plugin-Architektur von Shopware 6 eine leistungsfähige und zugleich strukturierte Grundlage für die Entwicklung von Erweiterungen. Die Kombination aus klarer Verzeichnisstruktur, definierten Erweiterungspunkten und umfangreicher Dokumentation ermöglicht es, Erweiterungen gezielt in das System zu integrieren, ohne dessen Stabilität zu gefährden. Diese technische Basis bildet die Grundlage für die im Rahmen dieser Arbeit konzipierte Erweiterung.

[1] `https://semver.org/lang/de/`

2.2 Industrie 4.0

Die Industrie 4.0, auch bekannt als vierte industrielle Revolution, ist nicht hundertprozentig klar zu definieren. Auch wenn der Name anderes vermuten lässt, betrifft diese weit mehr als nur die Industrie. Ronald Deckert sieht aber in den Konzepten der Digitalisierung, Vernetzung und künstlichen Intelligenz bedeutsame Anhaltspunkte, um sich der Ausmaße bewusst zu werden [6]. Daraus wird erkennbar, dass die Industrie 4.0 durch technologische Errungenschaften und die damit verbundene rasante Stärkung der Globalisierung geprägt ist. Trotz ihrer allgemeinen Relevanz, hat sie im besonderen einen starken Einfluss auf den E-Commerce Sektor genommen.

2.2.1 E-Commerce und Produktindividualisierung

Die Produktindividualisierung repräsentiert einen signifikanten Entwicklungstrend im modernen E-Commerce, der die Grenzen traditioneller Online-Verkaufsmodelle erweitert. Im Kontext des elektronischen Handels wird unter Produktindividualisierung die Möglichkeit verstanden, standardisierte Produkte nach spezifischen Kundenwünschen anzupassen oder zu konfigurieren. Dieser Ansatz transformiert das klassische E-Commerce-Paradigma von einem statischen Katalogmodell zu einem dynamischen und interaktiven Produktangebot. Während frühe Online-Shops lediglich digitale Abbilder physischer Kataloge darstellten, ermöglichen moderne Plattformen komplexe Interaktionen mit den angebotenen Produkten. Diese Entwicklung reflektiert die zunehmende technologische Reife von Webtechnologien und die gestiegenen Erwartungen der Konsumenten an digitale Einkaufserlebnisse. [1]

Der Mehrwert für Kunden durch Individualisierungsoptionen manifestiert sich auf mehreren Ebenen. Funktional betrachtet ermöglicht die Produktindividualisierung eine präzisere Erfüllung individueller Anforderungen und Bedürfnisse. Psychologisch betrachtet steigert sie das Gefühl von Kontrolle und Einbeziehung im Kaufprozess. Darüber hinaus kann die emotionale Bindung zum Produkt verstärkt werden, da Kunden eine aktive Rolle bei der Produktgestaltung übernehmen. Das führt zu einer höheren Kundenzufriedenheit und Loyalität im Vergleich zu Standardangeboten [14].

Die historische Entwicklung der Produktionssysteme veranschaulicht allgemein den Wechsel von der handwerklichen Einzelfertigung über die standardisierte Massenproduktion bis hin zur Mass Customization. [1]

2.2.2 Mass Customization

Im Besonderen, wenn eine Produktindividualisierung im Kontext von Industrie 4.0 angestrebt wird, sollte der Begriff Mass Customization betrachtet werden. Das Konzept der Mass Customization repräsentiert eine Zusammenführung zwei gegensätzlicher Produktionsweisen: die effiziente Massenproduktion standardisierter Produkte und die kundenspezifische Einzelfertigung. Das betrachtete Konzept zielt hierbei darauf ab, individuelle Kundenwünsche zu erfüllen, ohne die Effizienz und Produktivität der Massenproduktion zu verlieren. [9, 1]

Zentral für die praktische Umsetzung der Mass Customization ist das Prinzip der Modularisierung. Durch die Zerlegung eines Produkts in standardisierte, kombinierbare Module können vielfältige Endprodukte erzeugt werden, ohne dass jede Variante individuell entwickelt werden muss. Dieses Prinzip ermöglicht eine nahezu exponentielle Vervielfältigung der Produktvarianten bei nur linearem Anstieg der Komplexität in der Komponentenfertigung [9]. In der praktischen Umsetzung sieht sich Mass Customization aber mit diversen Herausforderungen konfrontiert. Um solch ein Produkt effektiv ausbalanciert in der Vielfalt und Komplexität zu produzieren und anzubieten, wird ein hohes Maß an Planung benötigt. [9]

Im Kontext dieser Arbeit zeigt sich beim Palettenkonfigurator eine spezifische Ausprägung des Mass-Customization-Ansatzes, die dem „Assemble to Order“ (ATO)-Prinzip entspricht. Bei diesem Ansatz werden standardisierte Komponenten auf Lager gehalten und erst nach Eingang einer Kundenbestellung zu einem Endprodukt auf Basis der Kundenwünsche zusammengesetzt (vgl. Abb. 2.4) [10]. Im Fall des Palettenkonfigurators werden bereits vorhandene Paletten mit verfügbaren Materialien nach Kundenwunsch bestückt. Die elementaren Komponenten liegen also bereits fertig produziert bereit und werden für die Individualisierung nur noch durch die Kombination und Stapelung nach Bestelleingang zusammengestellt.

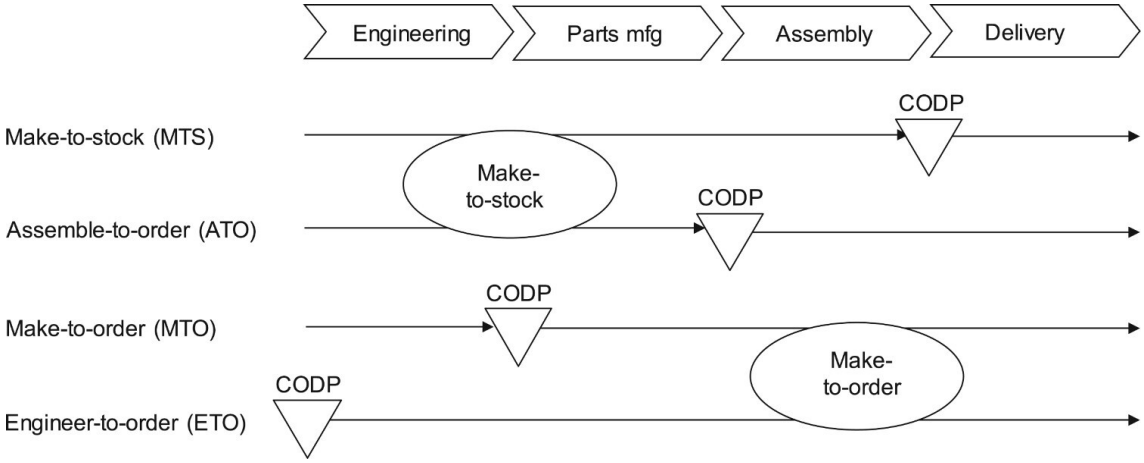


Abbildung 2.4: Platzierung des CODP für die jeweiligen Konzepte.
Quelle: [10] [Abruf: 10.05.2025]

Zur präzisen Einordnung lässt sich die Palettenkonfiguration von anderen Fertigungskonzepten abgrenzen. Beim „Engineer to Order“ (ETO) werden Produkte vollständig kundenspezifisch entwickelt. „Make to Order“ (MTO) hingegen ermöglicht es dem Kunden, Spezifikationen auszuwählen, auf deren Basis dann das Produkt mit den Anpassungen gefertigt wird. Bei „Configure to Order“ (CTO) werden, ähnlich wie bei ATO, vorgefertigte Komponenten nach Kundenvorgabe ausgewählt und zusammengebaut. [10]

Effiziente Konfiguratoren übernehmen hierbei eine Schlüsselrolle: Sie dienen als Schnittstelle zwischen Kundenwunsch und technischer Realisierbarkeit und reduzieren durch strukturierte Prozesse und visuelles Feedback die Komplexität der Auswahl. Sie tragen somit wesentlich zur Reduzierung der „Mass Confusion“ bei, die als negative Begleiterscheinung übermäßiger Auswahlmöglichkeiten auftreten kann. [12, 9]

Zusammenfassend kann festgehalten werden, dass das Konzept der Mass Customization wertvolle theoretische und praktische Grundlagen für diese Arbeit bietet. Der Palettenkonfigurator ermöglicht es, die Effizienz standardisierter Komponenten mit der Flexibilität individueller Kundenanforderungen zu verbinden, und verkörpert damit ein zentrales Prinzip erfolgreicher Mass-Customization-Strategien. [9]

2.3 Produktkonfiguratoren

Im Folgenden sollen kurz Produktkonfiguratoren betrachtet und der aktuelle Stand in der Technik beleuchtet werden. Dies soll eine Referenz für die spätere Entwicklung

schaffen sowie Priorisierungen für den zu implementierenden Produktkonfigurator aufzeigen.

2.3.1 Entwicklungsstrategien

Haug et al. identifizieren sieben unterschiedliche Strategien zur Entwicklung von Produktkonfiguratoren in ETO-Unternehmen, jedoch lassen sich diese prinzipiellen Ansätze auch allgemeiner anwenden. Sie erkannten, dass Projekte von Produktkonfiguratoren häufiger scheitern, und wollten methodische Lösungsansätze bieten, um eine erfolgreichere Planung und Umsetzungsquote zu erreichen. Dabei dominieren drei Strategien (S1–S3) die Praxis, während vier weitere (S4–S7) nur in Wartungsphasen, experimentell oder hypothetisch vorkommen. Die Autoren zeigen, dass die Wahl der Strategie maßgeblich von Produktkomplexität, verfügbaren Kompetenzen und dem gewünschten Dokumentationsgrad abhängt. [11]

Die drei Hauptstrategien bieten Entscheidungsgrundlagen für die Projektplanung, wobei die zu erledigenden Aufgaben sich in 3 Bereiche einteilen lassen: Wissensgewinnung, Wissensrepräsentation und Implementierung. Bei S1 wird der Fokus deutlich stärker auf die Evaluierung sowie Dokumentation gesetzt und unterteilt alle Aufgaben in strikte Rollen. Dem gegenüber steht S3, welches sich nicht auf formale Modelle verlässt, sondern direkt in die Implementierung übergeht. S2 kann als eine Kombination aus S1 und S3 interpretiert werden, denn dieses nutzt Teile beider Herangehensweisen, umgeht jedoch die Nutzung von weiteren Spezialisten und hat damit eine direktere Arbeitsweise als S1. [11]

Im Vergleich sorgt S1 zwar dafür, die Missverständnisse bei hoher Produktkomplexität zu reduzieren. Jedoch werden zusätzliche Ressourcen für die Dokumentation benötigt und es kann durch die Involvierung mehrerer Experten zu Verwirrung zwischen den Gebieten kommen. Zu empfehlen wäre dieser Ansatz hauptsächlich bei sehr komplexen Produktkonfiguratoren. S2 verkürzt dahingegen die benötigte Zeit für Projekte und bietet dabei geringere Verwirrung zwischen den Fachkräften. S3 minimiert die initialen Kosten und die Projektzeit, leidet jedoch bei wachsender Produktkomplexität, was es nur bei einer geringen Komplexität risikoarm macht. [11]

Praxisgerecht ist häufig eine Kombination (z. B. Start mit S1 und Wechsel zu S2 oder S3 innerhalb der Feintuning-Phasen). [11]

	Relative Vorteile (+) und Nachteile (–)	S1	S2	S3
1	Evaluierung der gesammelten Produktinformationen vor der Implementierung	+	+	–
2	Einfache Änderung der implementierten Produktinformationen	+	+	–
3	Einfache Anpassung der Konfigurator-Software	+	+	–
4	Bereitstellen der Kommunikation zwischen Produktexperten und Konfigurator-Software-Experten	+	–	–
5	Minimierung der Nutzung von Ressourcen für Dokumentationsarbeiten	–	–	+
6	Reduzierung von Informationsübergaben (d. h. geringere Wahrscheinlichkeit von Missverständnissen und Fehlern)	–	+	+
7	Schnellere Implementierung der gesammelten Informationen (d. h. schnellere Entwicklung)	–	+	+

Tabelle 1: Vergleich der Hauptentwicklungsstrategien bei Produktkonfiguratoren
Quelle: [11], übersetzt

2.3.2 Analyse existierender Produktkonfiguratoren

Moderne Produktkonfiguratoren zeigen eine Reihe wiederkehrender Gestaltungsprinzipien auf, die unabhängig von Branche und jeweiliger Implementierung beobachtet werden können. In diesem und dem folgenden Abschnitt wurden Konfiguratoren verschiedener Anbieter verglichen, Gemeinsamkeiten festgestellt und Literatur überprüft. [1]

Erstens steht **Interaktivität mit Echtzeit-Feedback** im Vordergrund, was bedeutet, dass jede Benutzereingabe unmittelbar eine Aktualisierung der Visualisierung sowie von Preis- und Verfügbarkeitsangaben auslöst. [3]

Zweitens verfügen die Systeme über integrierte **Validierungs- und Regelmechanismen**, die über hinterlegte Wissensbasen oder Constraint-Solver jede Auswahl mit Restriktionen prüfen und die Korrektheit der Konfiguration sicherstellen [3]. Diese Prüfung sorgt dafür, dass nur umsetzbare Varianten gekauft werden können.

Drittens verlangt die hohe Variantenvielfalt bei häufig vertretenden CTO oder MTO-Konfigurationen eine **attribut- bzw. merkmalbasierte Datenmodellierung**.

[1] Nennenswerte Produktkonfiguatoren:
`https://configure.bmw.de/de_DE/configure/G80`
`https://www.jc-trauringe.de/Konfigurator/`
`https://www.holzplatte-online.com/produkt/elektrisch-hoehenverstellbarer-schreibtisch/`

Bei dieser werden alle Ausprägungen in einem einzigen Produktmodell verwaltet, anstatt für jede Kombination ein separates Produkt anzulegen [3].

Viertens erfolgt die **Preis- und Verfügbarkeitsberechnung**, ohne dass die Benutzeroberfläche neu geladen wird, was für eine **SPA!** spricht. Ergänzend bieten einige Systeme eine **Speichern-und-Fortsetzen-Funktion**, bei der die Konfiguration gesichert werden kann. Mit der Funktionalität wird ermöglicht, den Konfigurationsprozess zu unterbrechen und später fortzusetzen. Ebenfalls kann dabei aber auch der Aspekt des Social Commerce greifen, indem die Möglichkeit zum Teilen der Konfiguration mit anderen gegeben wird.

Es muss ebenfalls zwangsläufig bei umfangreichen Produktkonfiguratoren ein verstärkter Fokus auf **Performance-Optimierung** gelegt werden, um die Time-to-InteractiveTTI möglichst minimal zu halten.

2.3.3 Meistgenutzten Arten

Man kann Konfiguratoren basierend auf ihrer Wissensdarstellung, ihrem Lösungsansatz und der Visualisierung in verschiedene gängige Kategorien unterteilen, die in der Praxis jeweils durch spezifische Technologien gekennzeichnet sind.

Die **regelbasierten Konfiguratoren** bilden den meistverbreiteten Einstieg, weil ihr „Wenn → Dann“-Ansatz eine schnelle Konsistenzprüfung erlaubt. Typische Einsatzfelder sind Zubehör-Bundles oder Farb- bzw. Materialvarianten, bei denen die Komplexität der Abhängigkeiten vergleichsweise überschaubar bleibt. Probleme treten erst bei steigender Regelzahl auf („rule explosion“) oder wenn die Regeln immer komplexer werden. [3]

Constraint-basierte Systeme stellen die Konfigurationen als Constraint-Satisfaction-Problem dar. Jede Variante ist dabei eine Variable und wird durch Restriktionen begrenzt. Ein Constraint-Solver prüft dabei die vom Nutzer getroffenen Entscheidungen, validiert sie und passt die möglichen Optionen entsprechend an. Konfiguratoren, die diesen Ansatz verfolgen, gehören häufig zur Industrie und handeln mit anderen Unternehmen. [2]

In der Praxis etablieren sich häufig **hybride Konfiguratoren**, die einfache Kompatibilitäten regelbasiert prüfen, während komplexere Regeln an einen Constraint-Solver delegiert werden.

Parallel zur Wissensschicht hat sich die **visuelle Ausprägung** der Systeme weiterentwickelt. Für viele Produktkategorien (Automobil[1], Möbel[2], Lifestyle[3]) werden Konfiguratoren zunehmend als 3D- oder AR-Applikationen realisiert[38, 16].

Da ein ständiges Neuladen der Webseite während der Konfiguration äußerst störend wäre, setzen nahezu alle modernen Konfiguratoren auf **Single Page Applications**. Dynamische Aktualisierungen der Preise oder anderer Eigenschaften erfolgen hiermit ganz ohne ein erneutes Laden der Webseite.

[1]Bsp: `https://configure.bmw.de/de_DE/configure/G80`

[2]Bsp: `https://www.holzplatte-online.com/produkt/elektrisch-hoehenverstellbarer-schreibtisch/`

[3]Bsp: `https://www.jc-trauringe.de/Konfigurator/`

3 Implementierung

3.1 Anforderungsanalyse

Die Anforderungsanalyse orientiert sich an etablierten Methoden. Als Grundlage dient dabei die im Kundenauftrag enthaltenen Informationen, insbesondere der ursprüngliche Lösungsvorschlag, exemplarische Visualisierungen sowie Notizen aus der vorherigen Kundenbetreuung. Aus diesen Quellen wurden zunächst die zentralen Anforderungen extrahiert und in strukturierter Form zusammengestellt. Da sich jedoch im Verlauf dieser initialen Analyse zahlreiche Detailfragen ergaben, die nicht aus den vorhandenen Materialien beantwortet werden konnten, wurde direkt mit dem Kunden kommuniziert und es wurden Interviews mit erfahreneren Kollegen geführt.

Darüber hinaus wurden technische Spezifikationen der Shopware 6 Plattform analysiert sowie bereits existierende Konfiguratorlösungen aus Literatur und Praxis herangezogen. Wie bereits in Abschnitt 2.3.1 betrachtet haben Haug et al.[11] 3 Hauptstrategien bei der Entwicklung identifiziert. Innerhalb dieser Arbeit wird eine Kombination aus den Strategien S2 und S3 verfolgt. Die Entwicklung fängt mit der Strategie S2 an, also werden zuerst Informationen gesammelt, die Anforderungsanalyse und Konzeption erstellt und dann erst der Produktkonfigurator implementiert. Für spätere Änderungen oder Anpassungen wird dann jedoch die Strategie S3 verfolgt und eine direkte Implementierung durchgeführt.

Ziel dieses Abschnitts ist die systematische Erfassung, Kategorisierung und Priorisierung der Anforderungen, um eine belastbare Grundlage für Konzeption, Umsetzung und Evaluation der Lösung zu schaffen.

3.1.1 Stakeholder

Im Rahmen der Anforderungsanalyse können drei zentrale Stakeholder identifiziert werden. Die jeweiligen Anforderungen wurden auf Grundlage vorhandener Dokumentationen, dem ursprünglichen Projektauftrag, interner Notizen, sowie direkter Kommunikation mit dem Kunden erarbeitet.

Unternehmen Die Agentur, die das Projekt im Auftrag des Kunden umsetzt, verfolgt primär folgende Ziele:

- Entwicklung einer funktionalen Lösung im Rahmen technischer und wirtschaftlicher Realisierbarkeit
- Einhaltung definierter Budget- und Zeitvorgaben
- Gestaltung eines Plugins mit potenzieller Wiederverwendbarkeit in vergleichbaren Konfigurationsszenarien
- Reibungslose Integration in bestehende Shopware-6-Systeme ohne tiefgreifende Anpassungen

Diese Ziele reflektieren das Bestreben des Unternehmens, eine Balance zwischen Kundenanforderungen und wirtschaftlicher Machbarkeit zu finden. Der Aspekt der Wiederverwendbarkeit unterstreicht zudem den Wunsch nach einem ökonomisch sinnvollen Softwaredesign.

Shopbetreiber Die Ziele des Shopbetreibers basieren auf konkreten funktionalen Anforderungen, die sich aus dem Projektauftrag und den Kundengesprächen ableiten lassen:

- Umsetzung eines Palettenkonfigurators mit acht auswählbaren Ebenen
- Sicherstellung, dass eine bestimmte Produktsorte (Brennholz) nur einmal gewählt werden kann
- Einschränkung des Erwerbs von Optionsprodukten außerhalb einer Konfiguration
- Visuelle Darstellung der gewählten Konfiguration
- Eine übersichtliche Darstellung der konfigurierten Palette im Warenkorb
- Korrekte Berechnung und Anzeige von Preisen und Gewichten

Im Vordergrund steht die zuverlässige Abbildung der gewünschten Konfigurationslogik sowie die korrekte Darstellung relevanter Informationen für den Endnutzer im Kaufprozess. Die Betonung der Benutzerfreundlichkeit und visuellen Rückmeldung reflektiert den Anspruch, einen intuitiven Konfigurationsprozess zu ermöglichen, der Kundeninteresse und Bindung schafft.

Endkunden Basierend auf Erkenntnissen aus Literatur und usability-orientierten Gestaltungsprinzipien lassen sich für die Endnutzer folgende Erwartungen formulieren:

- Intuitive und barrierearme Bedienbarkeit des Konfigurators
- Verlässliche Darstellung der Preise, Mengen und Produktauswahl
- Vermeidung kognitiver Überlastung durch übermäßige Informationsdichte
- Visuelles und interaktives Feedback zur getroffenen Konfiguration

Die empirische Untersuchung von Leclercq et al. zeigt, dass Nutzer insbesondere Wert auf verständliche Auswahlprozesse, konsistente Preisangaben sowie unmittelbares visuelles Feedback legen.[13]

3.1.2 Funktionale Anforderungen

Auf Basis der Stakeholder-Anforderungen und Überprüfung der technischen Machbarkeit wurden folgende funktionale Anforderungen für den Palettenkonfigurator definiert:

ID	Beschreibung
F1	Der Shopbetreiber kann den Konfigurator im Adminbereich definieren und verwalten
F2	Eine Palette kann auf der Produktdetailseite mithilfe von acht Auswahlfeldern konfiguriert werden
F3	Visuelle Repräsentation der gewählten Optionen
F4	Die gewählten Optionen werden im Checkout dargestellt und sind dort nicht mehr veränderbar.
F5	Dynamische Preis- und Gewichtsberechnung auf der Produktseite und im Warenkorb
F6	Persistenz der Konfiguration während des gesamten Kaufprozesses
F7	Die Konfiguration wird im Warenkorb validiert und ungültige Konfigurationen werden automatisch entfernt

Tabelle 2: Funktionale Anforderungen

Die funktionalen Anforderungen beziehen sich sowohl auf den Administrationsbereich – in dem der Shopbetreiber den Konfigurator verwaltet – als auch auf die Storefront, in der Endnutzer mit dem Konfigurator interagieren. Der Fokus liegt auf

der Bereitstellung einer konsistenten, funktionsfähigen und durchgängigen Konfiguration über alle Prozessschritte hinweg.

3.1.3 Nicht-funktionale Anforderungen

Aus der Entwicklerperspektive wurden folgende zu beachtende nicht-funktionale Anforderungen heraus gearbeitet.

ID	Beschreibung
NF1	Kompatibilität mit Shopware Version >= 6.10 und gängigen Browsern
NF2	Minimale Beeinträchtigung der Performance und möglichst kurze Reaktionszeit
NF3	Berücksichtigung von Abhängigkeiten und Drittanbieter-Erweiterungen zur Vermeidung von Konflikten
NF4	Responsives Design mit intuitiver Benutzerführung und visuellem Feedback
NF5	Wartbarkeit durch Einhaltung von Shopware-Konventionen und nutzen bewährter Praktiken

Tabelle 3: Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen stellen sicher, dass der entwickelte Palettenkonfigurator nicht nur die geforderten Funktionen bereitstellt, sondern auch qualitative Aspekte berücksichtigt. Sie bilden – gemeinsam mit den funktionalen Anforderungen – die Grundlage für die Konzeption, Implementierung und spätere Evaluation der entwickelten Lösung.

3.1.4 Priorisierung (MoSCoW)

Das MoSCoW-Modell ist eine etablierte Methode zur Priorisierung von Anforderungen in Softwareentwicklungsprozessen, bei der zwischen Muss-, Soll- und Kann-Kriterien unterschieden wird. Als Grundlage dienen die in den vorangegangenen Abschnitten definierten funktionalen und nicht-funktionalen Anforderungen. Die folgende Tabelle zeigt, welche Anforderungen zwingend erfüllt werden müssen (Muss), welche sinnvoll wären (Soll), und welche lediglich optional berücksichtigt werden können (Kann). Die Einordnung basiert auf der technischen Machbarkeit, dem erwarteten Nutzen sowie der Relevanz für die identifizierten Stakeholder. Eine Untersuchung von Vijayakumar et al. weist jedoch darauf hin, dass die Effektivität der

Methode wesentlich von der Erfahrung der Beteiligten und der Klarheit der formulierten Anforderungen abhängt. Insbesondere können Missverständnisse entstehen, wenn unterschiedliche Stakeholder ungleiche Prioritäten setzen oder Anforderungen nicht eindeutig spezifiziert sind [36].

Priorität	Typ	Anforderung
Muss	F1	Konfiguration und Verwaltung im Adminbereich
	F2	Konfiguration über acht Auswahlfelder
	F5	Dynamische Preis- und Gewichtsberechnung
	F6	Persistente Zuordnung während des Kaufprozesses
	F7	Validierung und automatische Entfernung ungültiger Konfigurationen
	NF1	Kompatibilität mit Shopware Version 6.10 und gängigen Browsern
	NF2	Minimale Beeinträchtigung der Shop-Performance und schnelle Reaktionszeit
	NF5	Wartbarkeit durch Einhaltung von Shopware-Konventionen und nutzen bewährter Praktiken
Soll	F3	Visuelle Repräsentation der gewählten Optionen
	F4	Darstellung im Checkout, keine Bearbeitbarkeit
	NF4	Responsives, benutzerfreundliches Design mit visuellem Feedback
Kann	NF3	Berücksichtigung von Abhängigkeiten und Drittanbieter-Erweiterungen

Tabelle 4: Priorisierung der funktionalen und nicht-funktionalen Anforderungen nach dem MoSCoW-Modell

3.2 Konzeption

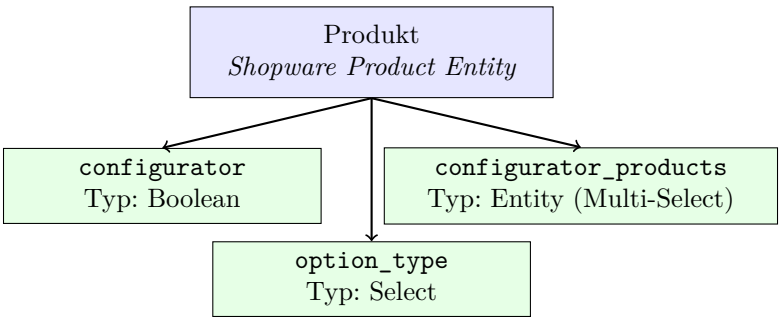
Auf Basis der Anforderungsanalyse erfolgte die anschließende Konzeption der technischen Lösung. Dabei wurden mehrere Implementierungsansätze identifiziert und auf ihre Umsetzbarkeit, Wartbarkeit und Zukunftssicherheit hin evaluiert.

3.2.1 Datenmodellierung

Für die Speicherung der Daten des Palettenkonfigurators wurde ein produktbasierter Ansatz gewählt, der auf den vorhandenen Strukturen von Shopware 6 aufbaut. So-

wohl der Konfigurator selbst als auch die zugehörigen Optionen werden als normale Produkte im Shopware-6-System angelegt. Die Differenzierung erfolgt ausschließlich über benutzerdefinierte Felder (Custom Fields), welche die Kategorisierung und Verknüpfung ermöglichen. Diese Felder sind über das Shopware-Custom-Field-System eingepflegt und in der Produktverwaltung unter einem eigenen Reiter integriert. (vgl. Abb. 3.1)

Der Konfigurator und die dazugehörigen Optionen werden im Shopware-System als reguläre Produkte angelegt. Die funktionale Differenzierung wird lediglich durch individuelle Felder (Custom Fields) erreicht, die die Kategorisierung und Verknüpfung ermöglichen. Die Felder wurden mithilfe des Shopware-Custom-Field-Systems eingefügt und sind in der Produktverwaltung unter einem separaten Tab eingebunden.



Hinweis: Die `Custom Field` Namen wurden für die Übersichtlichkeit vereinfacht.

Abbildung 3.1: Konzeptuelles Datenmodell des Palettenkonfigurators auf Basis von Custom Fields

Quelle: Eigene Darstellung

Das Custom Field isConfigurator (Boolean) kennzeichnet ein Produkt als Konfigurator. Über das Feld configurator_products (Entity) wird festgelegt, welche Optionsprodukte für die jeweilige Konfiguration zur Auswahl stehen. Die Optionsprodukte wiederum erhalten das Feld option_type (Select), welches zur Auswahl multi-type und single-type hat, was bestimmt ob eine Auswahl später mehrfach oder nur einmalig getroffen werden kann.

Durch die Nutzung der bestehenden Produkt- und Custom-Field-Mechanismen von Shopware wird ein hoher Integrationsgrad bei gleichzeitig geringem Entwicklungsaufwand erreicht. Darüber hinaus profitieren Shop-Betreiber von einer vereinfachten Administration über das Standard-Backend und Funktionen von Shopware.

Die im Warenkorb abgelegte Konfiguration folgt einer hierarchischen Struktur: Das Konfigurator-Produkt wird als Parent-Line-Item gespeichert, während die gewählten Optionen als Nested Line Items eingebettet sind (vgl. Abb. 3.2). Diese Struktur wird über die Mechanismen des CartCollectors und CartProcessors erzeugt und verarbeitet.

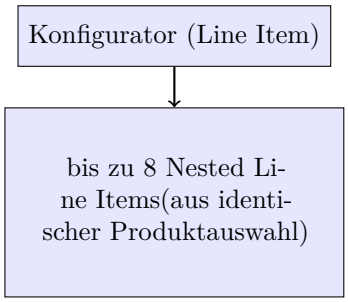


Abbildung 3.2: Line-Item-Hierarchie im Warenkorb mit bis zu acht Nested Line Items
Quelle: Eigene Darstellung

Abbildung 3.3 zeigt den vollständigen Ablauf von der Initialisierung der Konfigurationsansicht bis zur Übertragung der zusammengestellten Palette in das Bestellsystem. Dabei wird mit der zweistufigen Validierung darauf geachtet, das sowohl client- als auch serverseitig eine valide Konfiguration erstellt wird. Im Fall einer negativen serverseitigen Prüfung wird der Prozess abgebrochen und das Produkt automatisch aus dem Warenkorb entfernt.

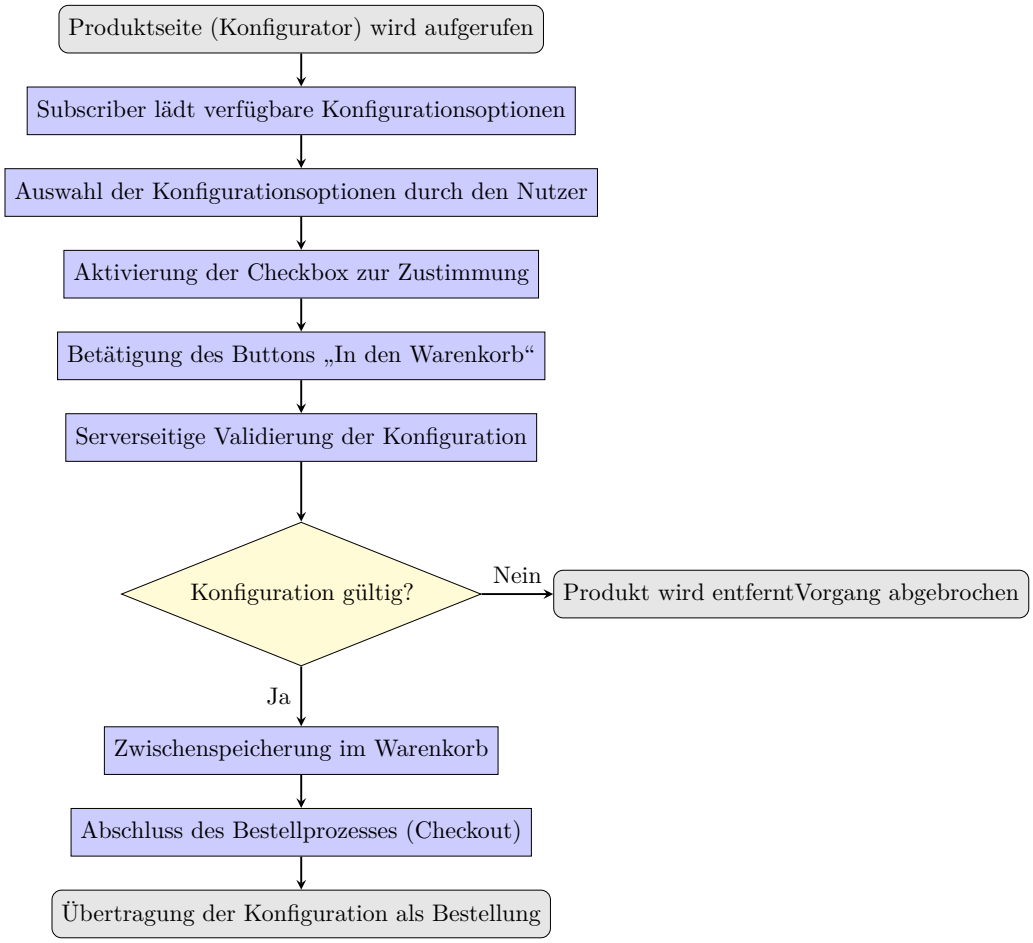


Abbildung 3.3: Ablaufdiagramm des Palettenkonfigurators

Quelle: Eigene Darstellung

3.2.2 Systemarchitektur/Architekturkonzept

Aufbauend auf der Datenmodellierung beschreibt dieser Abschnitt die Systemarchitektur des Palettenkonfigurators. Ziel ist es, die technischen Komponenten sowie deren Integration in die bestehende Plattformstruktur von Shopware 6 darzustellen und ihr Zusammenspiel im Konfigurations- und Bestellprozess zu erläutern. Die Architektur folgt wie Shopware 6 den Prinzipien des Model-View-Controller(MVC)-Modells in angepasster Form und integriert sich direkt in die auf Symfony basierende Struktur von Shopware 6.

Frontend-Integration Im Frontend erfolgt die Einbindung des Konfigurators durch eine Erweiterung der Produktdetailseite mittels Twig-Templates und JavaScript. Die grafische Oberfläche basiert auf SCSS-Stylesheets und ermöglicht die interaktive Auswahl der Konfigurationsoptionen direkt auf der Produktseite.

Backend-Logik und Event-Verarbeitung Das Backend nutzt dedizierte Services und Event-Subscriber zur gezielten Integration in Shopware-6-Prozesse. Der `WbnPalletsConfiguratorSubscriber` greift beim `ProductPageLoadedEvent` auf das Product Repository zu und stellt die relevanten Daten für das Frontend bereit. Für Inhalte in Erlebniswelten übernimmt der `WbnPalletsConfiguratorResolver` eine vergleichbare Funktion im Kontext des CMS-Systems. Die Event-basierte Architektur (z. B. Subscriber) gewährleistet eine lose Kopplung der Komponenten.

Konfigurationsverarbeitung im Warenkorb Die zentrale Logik zur Abbildung der Konfiguration im Warenkorb verteilt sich auf mehrere Komponenten: Der zuständige Controller liest die Benutzereingaben (gespeichert als Payload) aus und erzeugt ein Line Item mit verschachtelten Optionen (Nested Line Items). Der `Collector` prüft, ob ein Line Item einem Konfiguratorprodukt entspricht. Die Verarbeitung erfolgt durch den `WbnPalletsConfiguratorCartProcessor`, der sämtliche Nested Line Items analysiert und auf Basis ihrer Eigenschaften den Preis und das Gewicht des übergeordneten Konfigurators berechnet. Die Werte der enthaltenen Optionen bleiben dabei unverändert. Ergänzend validiert der `WbnPalletsConfiguratorCartValidator` die Konfiguration hinsichtlich der Vollständigkeit, also mindestens eine gewählte Option.

Datenfluss und Persistenz Der Datenfluss erfolgt über bereits existierende Shopware 6 Datenbanken. Die Produktinformationen werden über das Repository aus der Datenbank geladen und bei Bedarf per Subscriber an das Frontend übergeben. Beim Hinzufügen in den Warenkorb wird die Konfiguration temporär als Line-Item-Struktur gespeichert, aber die finale Speicherung erfolgt erst mit Abschluss des Bestellvorgangs in der Order-Datenbank.

Fazit Die gewählte Architektur nutzt konsequent die Erweiterungsmechanismen von Shopware, was eine hohe Kompatibilität, Wiederverwendbarkeit und Wartbarkeit sicherstellt. Die klare Trennung zwischen Präsentations- und Anwendungslogik erleichtert zudem sowohl die Entwicklung als auch das Debugging.

3.2.3 Technische Lösungsansätze

Erweiterung des Shopware-Plugins „Custom Products“[25] oder Entwicklung eines eigenen Plugins Eine erste, naheliegende Option bestand in der Nut-

zung des bestehenden Shopware-Plugins „Custom Products“[1], welches im Rahmen des vom Kunden genutzten Shopware Rise-Plans bereits verfügbar gewesen wäre. Bei näherer Betrachtung erwies sich dieser Ansatz jedoch als problematisch, da er eine signifikante Abhängigkeit von einem Drittanbieter-Plugin geschaffen hätte. Diese Abhängigkeit hätte potenziell zu erheblichen Komplikationen führen können, falls der Kunde seinen Abonnementplan ändern oder das Plugin selbst in zukünftigen Updates inkompatible Änderungen erfahren würde. Im Falle einer Kündigung des Plans hätte dies eine vollständige Neuentwicklung des Palettenkonfigurators erfordert. Aufgrund dieser Risiken wurde dieser Ansatz trotz seiner scheinbaren Direktheit verworfen.

Infolgedessen wurde die Entwicklung eines vollständig eigenständigen Plugins in Betracht gezogen. Auf Grund mangelnder Erfahrung des Autors in diesem Bereich fanden intensive Besprechungen mit unterschiedlichen erfahrenen Teammitgliedern statt. Daraus kristallisierten sich zwei Implementierungsstrategien heraus, die eine eigenständigere und damit zukunftssicherere Lösung versprachen.

Nutzung nativer Shopware-Entitäten oder Datenimport aus externen Quellen Die erste Option sah vor, die Konfigurationsdaten in einer externen Struktur – beispielsweise einer Excel-Tabelle – zu pflegen und diese über einen CSV-Import in das Plugin zu integrieren. Dieser Ansatz hätte es ermöglicht, die Konfigurationsdaten unabhängig vom Shopsystem zu verwalten und theoretisch auch aus Drittsystemen zu übernehmen. Allerdings hätte dieses Vorgehen aus Sicht der späteren Pflege und Erweiterbarkeit erhebliche Nachteile mit sich gebracht. Insbesondere wären Aktualisierungen aufwendig gewesen, da sämtliche Bilder manuell im Mediensystem von Shopware hätten hinterlegt, und ihre Referenzen korrekt in die importierten Datensätze eingetragen werden müssen. Zudem wäre die Bearbeitung für fachlich weniger versierte Shopbetreiber unübersichtlich und fehleranfällig gewesen. Aus diesen Gründen wurde dieser Ansatz als nicht praxistauglich eingestuft.

Die zweite Option bestand in der Nutzung des bestehenden Produktsystems von Shopware. Dabei würden sowohl der Konfigurator selbst als auch die verfügbaren Optionen als reguläre Shopware-Produkte angelegt. Dieser Ansatz versprach eine nahtlosere Integration in die bestehenden Shop-Prozesse und eine intuitivere Verwaltung der Optionen über die vertraute Produktverwaltung. Zwar erforderte dieses Vorgehen das manuelle Anlegen jeder Option als separates Produkt, was einen erhöhten initialen Aufwand mit sich bringt, doch wurde dies durch Vorteile wie die

[1] `https://docs.shopware.com/en/shopware-6-en/extensions/customproducts`

zentrale Integration von Preisen, Abbildungen und Lagerbeständen kompensiert. Die technische Herausforderung bestand dabei vor allem in der korrekten Verarbeitung der Produktdaten innerhalb des Warenkorbs und der Checkout-Logik.

Verwendung eigener Entitäten oder bestehender Produktstrukturen Auf Grundlage dieser Entscheidung stellte sich die Frage nach der konkreten Datenstruktur. Zur Auswahl standen einerseits die bestehenden Produktentitäten von Shopware, die bereits eine umfassende Integration in das System und dessen Geschäftslogik bieten. Andererseits wäre die Definition eigener Entitäten möglich gewesen, was jedoch einen signifikant höheren Entwicklungsaufwand sowie eine separate Verwaltung bedeutet hätte. Das würde aber potenziell für eine größere Anpassbarkeit und Modularität sorgen, mit daraus folgender Wiederverwendbarkeit. Nach genauer Abwägung wurde jedoch gegen eigene Entitäten entschieden, da diese den zeitlichen Rahmen übersteigen würden. Es ist nur ein vergleichsweise simpler Konfigurator gewünscht, was gut mit der bereits existierenden Produktentität machbar ist. Dieser Punkt wäre aber wichtig für komplexere Konfiguratoren oder wenn mehrere Konfiguratoren erstellt werden sollen.

Einsatz von Custom Fields im Vergleich zu einfachen Tags Da die Entscheidung auf einen produktbasierten Ansatz fiel, musste festgelegt werden, wie man diese Produkte identifizieren kann. Zunächst wurde in Erwägung gezogen, dafür Shopware-Tags zu verwenden. Diese sind leicht zu pflegen und flexibel einsetzbar. Allerdings zeigte sich, dass das Risiko einer versehentlichen oder inkonsistenten Verwendung durch den Shopbetreiber vergleichsweise hoch wäre.

Aus diesem Grund fiel die Entscheidung zugunsten sogenannter *Custom Fields*. Diese benutzerdefinierten Felder lassen sich in Shopware direkt an Entitäten, z.B. Produkte, binden und ermöglichen eine strukturierte Datenerweiterung mit vordefinierten Typen wie Boolean, Select oder Entity-Relationen. Sie sind vollständig in die Administrationsoberfläche integriert, können validiert und kontextabhängig gesteuert werden, und bieten dadurch eine deutlich höhere Robustheit im Vergleich zu frei vergebbaren Tags. Damit wird es möglich, die Produkte eindeutig identifizierbar zu machen und ihnen auch weitere Informationen zu übergeben. (vgl. Abb. 3.4)

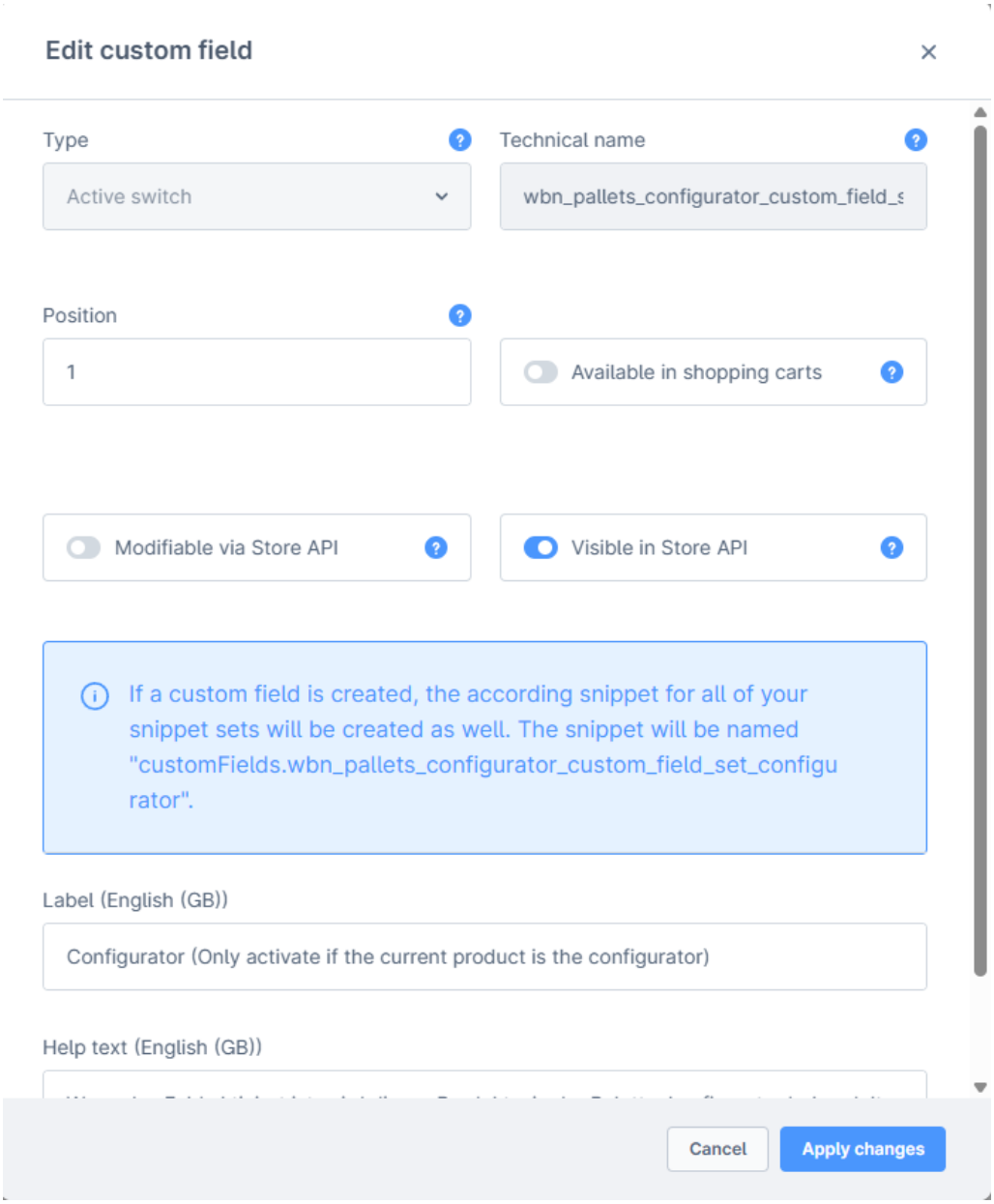


Abbildung 3.4: Custom Field Ansicht bzw. Bearbeitung im Admin Bereich

Quelle: Eigene Darstellung

Integration in bestehende Komponenten oder eigenständige Komponente Ein zentraler Aspekt der Konzeption war die Entscheidung, ob die Erweiterung als eigenständiges CMS-Element oder als Erweiterung eines bestehenden Shopware-Elements umgesetzt werden sollte. Shopware ermöglicht die Entwicklung individueller CMS-Komponenten, die nahtlos in das bestehende System eingebunden werden können. Alternativ besteht die Möglichkeit, vorhandene CMS-Elemente gezielt zu erweitern, was eine tiefere Systemintegration bei gleichzeitig geringerem Entwicklungsaufwand ermöglicht.

Da der Konfigurator selbst als reguläres Shopware-Produkt umgesetzt wurde und somit eine Produktdetailseite besitzt, erschien es naheliegend, ihn in ein bestehendes Element zu integrieren. Die Wahl fiel auf das bereits auf Produktseiten verwendete CMS-Element „Galerie und Buybox“, genauer auf das zugrunde liegende Buybox-Template. Durch die Erweiterung dieses Templates konnte der Konfigurator direkt in die bestehende Produktpräsentation eingebunden werden, ohne dass eine seperate Darstellungsstruktur notwendig wurde und gepflegt werden müsste.

Dieser Ansatz reduziert die Integrationskomplexität und gewährleistet gleichzeitig eine konsistente Benutzererfahrung. Für komplexere Konfigurationsszenarien wäre hingegen die Entwicklung eines eigenständigen CMS-Elements zu bevorzugen.

Implementierungsstrategien für die Warenkorblogik Ein weiteres zentrales Element der Konzeption betraf die technische Umsetzung der Logik zur Übertragung der Konfiguration in den Warenkorb. Shopware 6 bietet hierfür das Konzept von sogenannten Line Items [21], welche durch Collector- und Processor-Klassen erweitert und modifiziert werden können [20]. Es musste entschieden werden, an welcher Stelle die Konfiguration übergeben und validiert wird und wie die verschachtelte Struktur zwischen Hauptprodukt und Optionen im Warenkorb abgebildet werden kann.

Die Shopware 6 Architektur sieht vor, dass ein spezialisierter Controller beim Hinzufügen zum Warenkorb die Konfigurationsdaten aus dem Request ausliest und diese in ein strukturiertes Line Item umwandelt. Anschließend wird diese Struktur durch den `CartCollector` identifiziert und/oder angereichert und durch den `CartProcessor` verarbeitet [20]. Letzterer soll auf Basis der beim behandelnden Plugin notwendigen Kind-Elemente dynamisch Preis und Gewicht für das Hauptprodukt anpassen. Dazu soll ein `CartValidator` die Vollständigkeit der Konfiguration prüfen.

3.3 Entwicklungsumgebung

Das Plugin wurde unter dem Betriebssystem Windows 11 in PHPStorm[1] entwickelt. Die Shopware 6 Installation wurde in der Version 6.6.10.1 vorgenommen und mit einem Docker Container[2] verwendet. Da der Shop für den Shopbetreiber sich während der Erstellung dieser Arbeit noch in der Entwicklung befand, wurde fast ausschließlich der Demostore, also der Zustand bei einer Neuinstallation von Shopware

[1] `https://www.jetbrains.com/de-de/phpstorm/`
[2] `https://www.docker.com/`

6, verwendet. Die Versionskontrolle während der Entwicklung fand mit Git über Bitbucket statt, da es die Unternehmensnorm ist und dort auch für die anderen Mitarbeiter bereitgestellt wird. Da das Plugin selber speziell für ein Kundenprojekt gemacht wurde, wurde es in einem Branch von dem Hauptprojekt angelegt.

Google Chrome wurde für die Browseransicht verwendet, wobei die Chrome- Entwicklertools (engl. Chrome DevTools)[1] intensiv genutzt wurden. Hauptsächlich wurde die Elemente Ansicht verwendet, um sowohl den Code zu untersuchen, aber auch um testweise CSS Anpassungen vorzunehmen und ihre Auswirkungen visuell direkt sehen zu können. (vgl. Abb. 3.5)

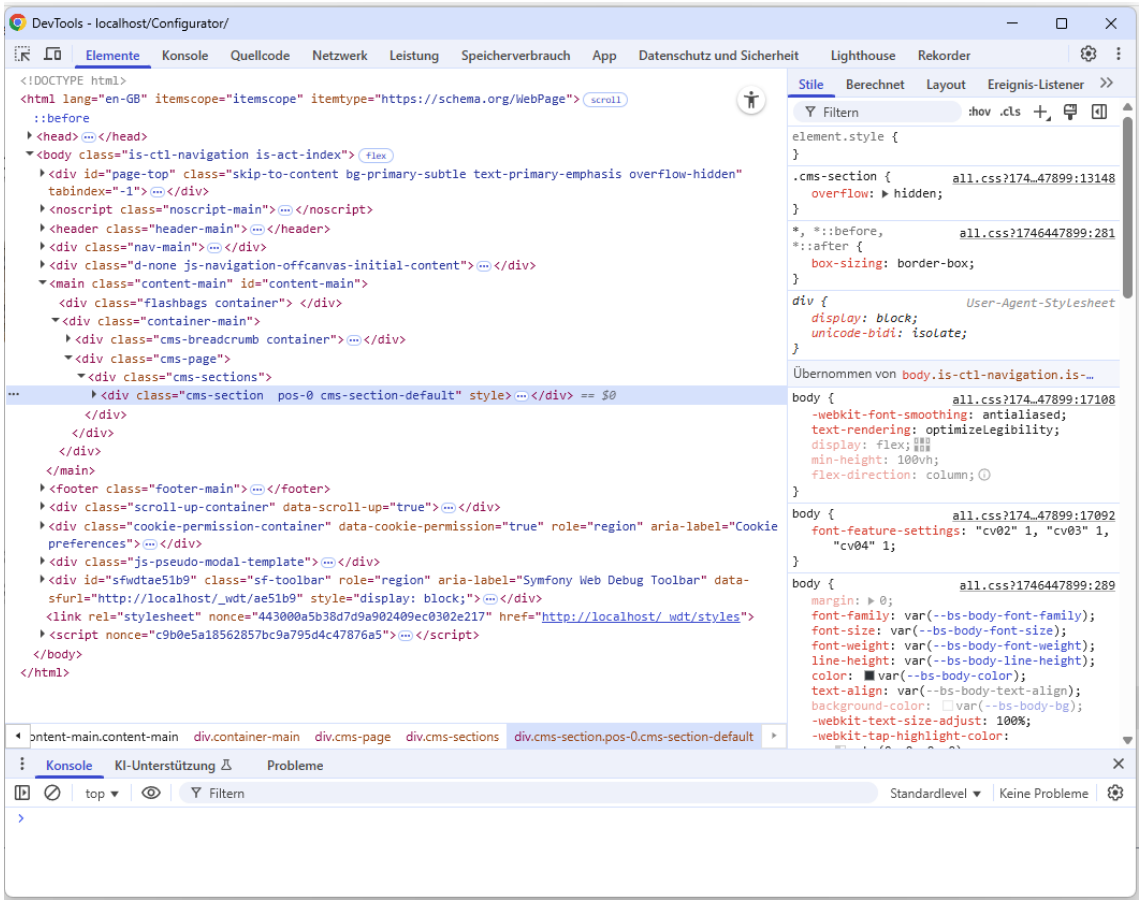


Abbildung 3.5: Chrome-Entwicklertools

Quelle: Eigene Darstellung

Bei einer umfangreichen Software, wie es Showpware 6 ist, ist dies auch sehr hilfreich, da man über das Untersuchen spezielle Teile des HTML-Codes genauer betrachten und darüber die korrelierende Position im Code ausfindig machen kann. Es wurde aber auch die Netzwerkübersicht genutzt zum Debuggen und um im Speziellen zu

[1] `https://developer.chrome.com/docs/devtools?hl=de`

kontrollieren, dass die Formularübermittlung wie gewünscht abläuft. Ebenfalls war die Konsole für das Debugging von Javascript-Code besonders hilfreich. In Kombination mit den Chrome-Entwicklertools wurde außerdem der Symfony Profiler([1]) genutzt, um z. B. einzusehen, welche Events zur Verfügung standen oder welche Templates geladen wurden.

3.4 Projektstruktur und dessen Bestandteile

Die Plugin-Verzeichnisstruktur wurde bereits im Unterunterabschnitt 2.1.4 beschrieben und wird auch in der Projektstruktur verfolgt. Das ist zu empfehlen und auch an vielen Stellen notwendig, da man damit nicht nur die Übersichtlichkeit erhöht, sondern auch sicherstellt, dass alles miteinander verbunden ist und funktioniert. Häufig kann man dadurch auch einen Nutzen von bereits implementierten Funktionen ziehen, die Showpare 6 einem bietet, und erspart somit dem Programmierer extra Aufwand. Dazu basiert Shopware bei der Darstellung auf Templates, die mit Twig erstellt werden. Diese erweitert man parallel im eigenen Projekt, indem man Templates bzw. die Twig-Blöcke überschreibt und erweitert.

3.5 Entwicklung

Die Entwicklung des Palettenkonfigurator-Plugins für Shopware 6 stellte ein komplexes Unterfangen dar, das von der initialen Anforderungsanalyse bis zur finalen Implementierung diverse Herausforderungen beinhaltete. Im Folgenden wird der gesamte Entwicklungsprozess detailliert dargestellt.

Auf Basis der bis hierhin erworbenen Erkenntnisse und unter spezieller Betrachtung der im Unterabschnitt 3.1 analysierten Anforderungen und der daraus entstandenen Konzeption im Unterabschnitt 3.2 begann die Entwicklung. Eine zentrale Herausforderung bei der produktbasierten Implementierung bestand in der Schaffung eines zuverlässigen Systems zur Identifikation der relevanten Produkte. Als technische Lösung für diese Anforderung wurden zuvor Custom Fields identifiziert, die eine präzise Kennzeichnung ermöglichen.

[1] `https://symfony.com/doc/current/profiler.html`

3.5.1 Custom Fields

Die Implementation der Custom Fields erfolgte in den Install- und Uninstall- Methoden (vgl. Unterunterabschnitt 2.1.3) des Plugins in `WbnPalletsConfigurator.php`. Während des Installationsprozesses wird überprüft, ob die benötigten Custom-Field-Sets bereits existieren, um Duplikate zu vermeiden. Falls nicht, werden entsprechende Einträge in der Datenbanktabelle „custom_field_set“ erzeugt, gefolgt von der Definition der einzelnen Custom Fields innerhalb dieses Sets.

```
$customFieldSetRepository->create([[
  'name' => 'wbn_pallets_configurator_custom_field_set',
  'config' => [
    'label' => [
      'de-DE' => 'Wbn Palettenkonfigurator',
      'en-GB' => 'Wbn pallets configurator'
    ]
  ],
  'customFields' => [
    [
      'name' => 'wbn_pallets_configurator_custom_field_set_configurator',
      'type' => CustomFieldTypes::BOOL,
      'config' => [
      'componentName' => 'sw-field',
      'customFieldType' => 'switch',
      'customFieldPosition' => 1,
      'label' => [...],
      'helpText' => [...]
    ]
  ]
],
[...],  % Weitere Custom-Field-Definitionen
'relations' => [[
    'id' => Uuid::randomHex(),
    'entityName' => 'product'
  ]]
]], $$installContext->getContext());
```

Listing 1: Anlegen eines Custom Field Sets und Field in install()

Bei einer Deinstallation des Plugins wurde eine bedingte Löschung implementiert. Die Custom-Field-Daten werden nur dann aus der Datenbank entfernt, wenn der Benutzer explizit die Option gewählt hat, alle Daten zu löschen. Diese Vorgehensweise ermöglicht eine temporäre Deinstallation und Neuinstallation des Plugins bei Problembehandlungen, ohne dass die Datenbankdaten verloren gehen.

```
$customFieldSetRepository = $this->container->get('custom_field_set.repository');
$criteria = new Criteria();
$criteria->addFilter(new EqualsAnyFilter('name', [
'wbn_pallets_configurator_custom_field_set'
]));
$context = $uninstallContext->getContext();
```

```
$ids = $customFieldSetRepository->searchIds($criteria, $context);

if ($ids->getTotal() > 0) {
  $customFieldSetRepository->delete(
  array_values($ids->getData()),
  $context
  );
}
```

Listing 2: Löschen des gesamten Custom Field Sets in uninstall()

Diese angelegten Custom Fields können vom Shopbetreiber im Admin-Bereich auf der Bearbeitungsseite der Produkte eingesehen und ausgewählt werden. (vgl. Abb. 3.6)

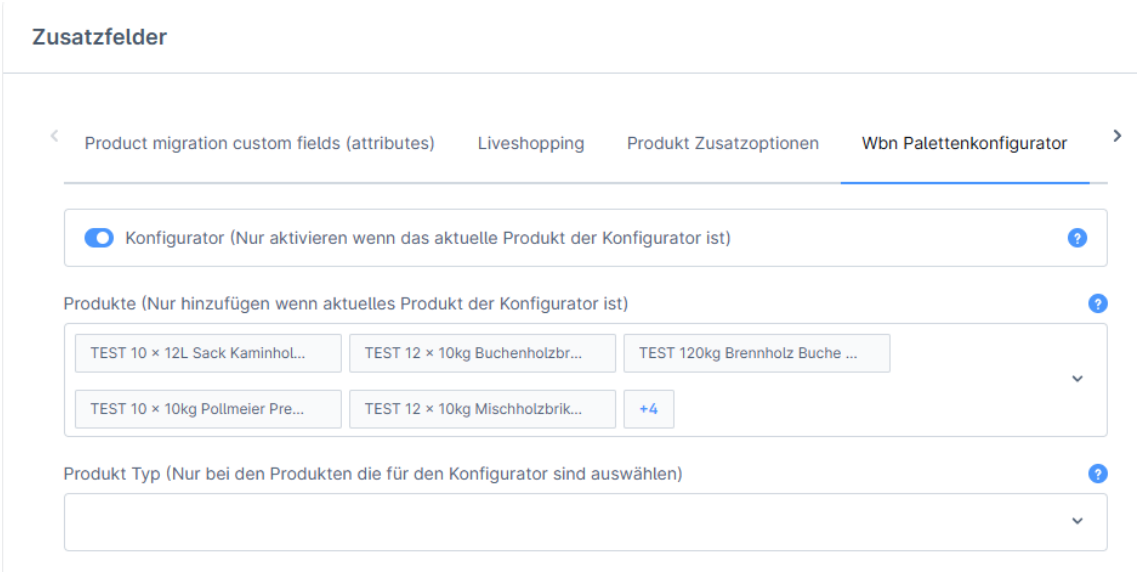


Abbildung 3.6: Custom Fields Übersicht im Produkt anhand des Konfigurators

Quelle: Eigene Darstellung

3.5.2 Subscriber

Subscriber sind Klassen, die, wie der Name vermuten lässt, auf bestimmte vorher festgelegte Events warten und, wenn diese ausgelöst wurden, reagieren. Dadurch können z. B. Validierungen und Interaktionen mit den Datenbanken situationsbedingt ausgeführt und angepasst werden.

In diesem Projekt wurde ein Subscriber (`WbnPalletsConfiguratorProductSubscriber.php`) implementiert, der speziell auf Ereignisse im Kontext der Produktseitenanzeige reagiert. Dieser Subscriber identifiziert das Hauptprodukt des Konfigurators anhand der zuvor definierten Custom Fields und ergänzt die Seitendaten um die

Daten der verfügbaren Konfigurationsoptionen. Technisch geschieht dies durch die Abfrage des Produkt-Repositories. Dabei werden zuerst die im Custom Field des Konfigurators gespeicherten Produkt-IDs (vgl. Abb. 3.6) zwischengespeichert und dann als neuer Filter eingesetzt. Dieser wird zusammen mit dem Kontext der Seite, in diesem Fall der SaleChannelContext, als Suchkriterium genutzt, um mit der Shopware-Funktion `search` die zuvor geladene Produktdatenbank zu durchsuchen. Die erhaltenen Produktdaten werden dann mit `addExtension` zur Produktseite hinzugefügt und dem Frontend zugänglich gemacht.

```
1  if (array_key_exists('wbn_pallets_configurator_custom_field_set_configurator',
       $customFields)
2  && $customFields['wbn_pallets_configurator_custom_field_set_configurator'] === true
3  && $customFields['wbn_pallets_configurator_custom_field_set_products'] !== null) {
4    $linkedProducts = $customFields['wbn_pallets_configurator_custom_field_set_products
         '];
5
6    $context = $event->getSalesChannelContext()->getContext();
7    $criteria = new Criteria();
8    $criteria->addFilter(new EqualsAnyFilter('product.id', $linkedProducts));
9
10   $productConfiguratorRepository = $this->productRepository
11   ->search($criteria, $context);
12
13   $productPage->addExtension('productConfiguratorRepository',
         $productConfiguratorRepository);
14 }
```

Listing 3: Auszug aus dem Subscriber

3.5.3 Frontend-Implementation

Nachdem die zusätzlichen Daten aus dem Backend im Template bereitgestellt wurden (vgl. Unterunterabschnitt 3.5.2), gliedert sich die Frontend-Implementation in zwei aufeinander abgestimmte Teile: erst die visuelle Umsetzung im Twig-Template und anschließend die Implementierung der Funktionalitäten mit dem JavaScript-Plugin.

Visuelle Umsetzung im Twig-Template Die visuelle Umsetzung des Konfigurators wurde durch die Erweiterung des Templates `buy-widget-form.html.twig` angegangen um eine vsl. sichere Position zu haben die nicht mit anderem Code kollidiert. Zuerst wurden existierende Twig-Blöcke der Storefront (beispielsweise `buy_widget_buy_container`) erweitert, um die Auswahlfelder über den Kauf-Button einzufügen (vgl. Listing 4).

```
{% for i in 0..7 %}
<div><select id="select{{ i }}"
name="lineItems[{{ product.id }}][select{{ i }}]"
class="select-form data-wbn-pallets-configurator-plugin">
<option value="" selected data-type="none" data-id="{{ i }}">
{{ "wbnPalletsConfigurator.buyWidgetForm.select.noSelect"|trans|sw_sanitize }}
</option>
{% for item in productConfiguratorRepository.elements if item.active %}
<option value="{{ item.id }}"
data-type="{{ item.customFields.wbn_pallets_configurator_custom_field_set_type
    }}"
data-id="{{ i }}-{{ loop.index }}">
{{ item.name }} -
<span data-price-container="{{ i }}-{{ loop.index0 }}">
{{ palletsConfiguratorProducts[loop.index0].price|currency }}
</span>
</option>
{% endfor %}
</select></div>
<div data-image-container="{{ i }}"…></div>
{% endfor %}
```

Listing 4: Gekürzter Auszug aus dem Twig-Template. Select-Element mit Data-Attributen

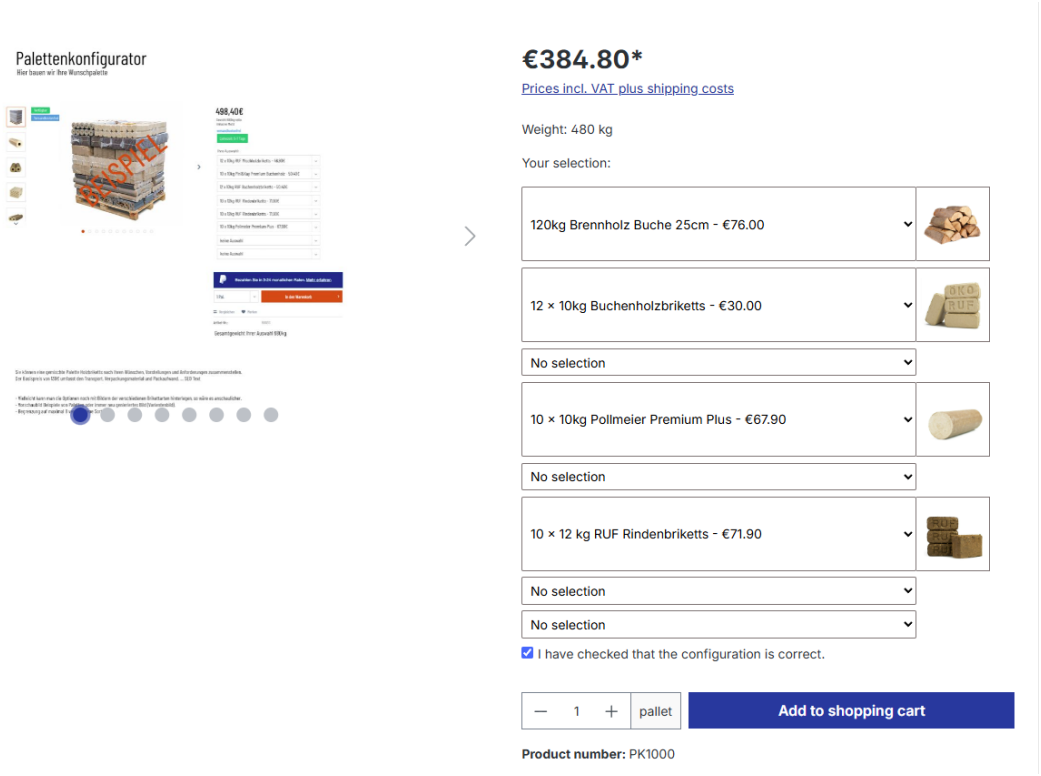


Abbildung 3.7: Visuelle Darstellung des Palettenkonfigurators in der Storefront

Quelle: Eigene Darstellung

Die Produktbilder werden in einem separaten Container neben den Auswahlfeldern unter Verwendung von `sw_thumbnails` angezeigt, wobei nur das zur aktuellen Auswahl gehörende Thumbnail per CSS und JavaScript sichtbar geschaltet wird. Zusätzlich wurden in der `base.scss` Datei weitere CSS-Anpassungen vorgenommen um im Besonderen die Darstellung der Bilder in der angedachten Position gleichmäßigen Anpassen des Selects (vgl. Abb. 3.7) zu ermöglichen.

Die Select Optionen werden wie im Listing 4 zu erkennen ist, mit den Informationen der zuvor geholten Optionsprodukte gefüllt. Dabei wird darauf gechtet die wichtigsten Informationen klar und einheitlich darzustellen (vgl. Abb. 3.8).



Abbildung 3.8: Beispielhafte Optionen im Select

Quelle: Eigene Darstellung

Interaktion zwischen Twig und JavaScript Um auf die Produktdaten, die über den Subscriber im Twig Template verfügbar sind, im Javascript-Plugin zugreifen zu können, werden diese in Twig-Variablen gespeichert. `configuratorProducts` liefert dabei das Array aller auswählbaren Produkte mit ihren Preisen, Gewichten usw., und `configuratorPriceCalc` enthält Basispreis, Gewicht und Währungsinformationen des Konfigurators. Diese werden als JSON über `data`-Attribute im Container-Element an das Template übergeben und als HTML gerendert. Dadurch kann dann das Javascript-Plugin auf die Daten zugreifen. (Listing 5).

```
{% set palletsConfiguratorPriceCalc = {
    'configuratorPrice': product.calculatedPrice.unitPrice
  } %}

{% block wbn_pallets_configurator_buy_widget_div_select %}
<div data-wbn-pallets-configurator-plugin
data-wbn-pallets-configurator-plugin-options='{
  "configuratorPriceCalc": {{ palletsConfiguratorPriceCalc|json_encode|raw }}
}'>
<!-- Gewichtsanzeige -->
<p>{{ "wbnPalletsConfigurator.buyWidgetForm.weight"|trans }}:
<span class="wbn-pallets-configurator-product-weight-number">
{{ product.weight }}
```

```
14    </span> kg
15    </p>
```

Listing 5: Gekürzter Auszug zur Darstellung der Twig-Variablen

Bei der Initialisierung des Plugins liest die JavaScript-Klasse die `data`-Attribute aus und speichert sie in ihrem State. Die zu verändernden HTML-Elemente werden über DOM-Selektoren identifiziert und geholt, um sie bei Benutzeraktionen aktualisieren zu können. Dafür achten EventListener auf ein Ereignis wie die Veränderung eines Selects (vgl. Listing 6).

```
1  const selectElements = this.el.querySelectorAll(".wbn-pallets-configurator-product-
       select .select-form");
2  selectElements.forEach(select => {
3    select.addEventListener("change", this.productSelect.bind(this));
4  });
```

Listing 6: Event Listener der Selects im JavaScript-Plugin

JavaScript-Implementierung Die JavaScript-Komponente `wbn-pallets-configurator.plugin.js` hat mehrere Kernfunktionalitäten und steuert die interaktive Logik. Wie bereits zuvor erklärt achten hier Event-Listener auf Veränderungen und reagieren darauf. Folgende Logik ist für Änderungen gegeben, bzw. entsprechende Anpassungen werden vorgenommen:

- **Preis- und Gewichtsberechnung:** Basierend auf `configuratorPriceCalc` und den im Template angelegten selektierten Produkten wird der Gesamtpreis bzw. das Gesamtgewicht bei jeder Änderung neu berechnet und in den jeweiligen HTML-Containern dynamisch eingefügt.
- **Product-Type-Logik:** Bei Auswahl eines „Single-Type"-Produkts werden alle anderen Optionen dieses Typs deaktiviert (und bei Rücknahme wieder freigegeben), um die Regel „pro Palette nur ein Brennholztyp" durchzusetzen. (vgl. Abb. 3.9)
- **Bildsichtbarkeit:** Die Methode `_updateProductImage()` blendet alle Thumbnails im entsprechenden Container aus und zeigt nur das zur aktuellen Auswahl gehörige Bild an.
- **Aktivierung des Kauf-Buttons:** Die Checkbox muss aktiviert sein und mindestens eine gültige Auswahl getroffen worden sein, damit der Buy-Button aktiviert wird.



Abbildung 3.9: Beispiel für Regelung zur Blockung des „Single-Type“

Quelle: Eigene Darstellung

3.5.4 Warenkorb-Integration

Die Warenkorb-Integration war im allgemeinen eine Herausforderung. Die Integration erforderte ein tiefes Verständnis der Shopware-Cart-Architektur und der zugehörigen Datenflüsse, das durch intensive Studien der Shopware-Dokumentation, insbesondere der Materialien zur Cart Architecture [24], sowie durch Analyse bestehender Implementierungen erworben wurde.

Controller Der erste Schritt zur Implementierung der Warenkorb-Integration bestand in der Entwicklung eines Controllers, der die Übernahme der Konfigurationsdaten und Umwandlung in die `LineItem`-Struktur vornimmt. Dabei kommt Shopwares `RequestBag` zum Einsatz, um die Konfigurationsdaten, welche aus den Formularfeldern übermittelt werden, auszulesen [21].

Aus den extrahierten Daten wird zunächst ein Haupt Line Item für den Konfigurator selbst erzeugt, welcher als Payload die gewählte Konfiguration hinzugefügt bekommt. Anschließend wird für jede gewählte Option mithilfe der Produkt-ID ein eigenes Line Item erstellt und dem Konfigurator-Line-Item als Child hinzugefügt (vgl. Listing 7). Jedes Line Item wird dabei automatisch mit den relevanten Produktdaten wie Name, Beschreibung und Preis aus dem entsprechenden Produkt-Repository angereichert, da sie als Line Item Typ „Produkt“ angelegt wurden. Dieses hierarchische Modell ist von Shopware so geplant, jedoch nicht als Standard implementiert. Für das grundsätzliche Anlegen von Child-Items wurde das Beispiel von Stackoverflow [34] verwendet.

```
1  public function addConfiguratorToCart(Request $request, SalesChannelContext
       $context): Response
```

```
2   {
3     // Konfigurationsdaten auslesen
4     $lineItems = $requestDataBag->get('lineItems');
5     foreach ($lineItems as $lineItemData) {
6
7     // Haupt-LineItem für den Konfigurator
8     $mainItem = new LineItem('wbn-config-' . Uuid::randomHex(), LineItem::
          PRODUCT_LINE_ITEM_TYPE, $lineItemData['productId'], 1);
9     $mainItem->setPayloadValue('configuration', $config);
10
11    // Child-LineItems für jede Option
12    foreach ($lineItemData->get('options') as $optionId) {
13      $child = new LineItem('wbn-config-child-' . Uuid::randomHex(), LineItem::
            PRODUCT_LINE_ITEM_TYPE, $optionId, 1);
14      $mainItem->addChild($child);
15    }
16  }
17    // In den Warenkorb einfügen und speichern
18    $this->cartService->add($cart, $items, $context);
19    $this->cartPersister->save($cart, $context);
20
21    return $this->createActionResponse($request);
22  }
```

Listing 7: Beispielhafter Auszug aus dem Cart-Controller: Erzeugung verschachtelter Line Items

Collector/Processor Eine besondere Herausforderung stellte die korrekte Preisberechnung für die konfigurierte Palette dar. Nach Analyse verschiedener Ansätze, einschließlich der in der Shopware-6-Dokumentation vorgeschlagene Processor-basierte Lösung[20], wurde ein direkterer Ansatz gewählt: Die Preisberechnung erfolgt durch Summierung der Einzelpreise der gewählten Optionen zum Basispreis des Konfigurator-Produkts. Diese Implementierung erwies sich als effektiver und robuster als die komplexeren Processor-basierten Ansätze, da sie die bereits vorhandenen Preisstrukturen der Produkte nutzt und keine zusätzliche Preislogik erfordert. Eine tiefergehende Analyse ergab, dass die LineItems bereits vor der Verarbeitung durch den Processor mit korrekten Preisinformationen ausgestattet waren, weshalb nur noch summiert werden musste.

Warenkorb Darstellung Die Darstellung der konfigurierten Palette im Warenkorb erforderte spezifische Anpassungen, um eine übersichtliche und verständliche Präsentation zu gewährleisten. Zu diesem Zweck wurde ein angepasstes Template für die Warenkorb-Ansicht erstellt, welches die konfigurierten Optionen als ausklappbare Elemente innerhalb des Hauptprodukts darstellt. Diese Darstellung vermittelt

dem Kunden ein klares Bild seiner Konfiguration, ohne die Warenkorb-Ansicht durch zahlreiche separate Einträge zu überladen (vgl. Abb. 3.10).

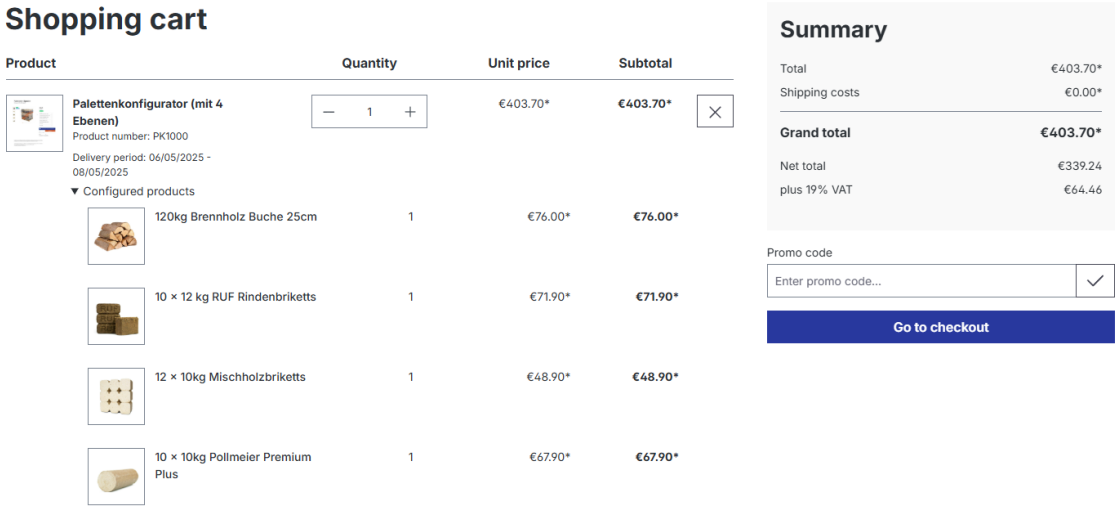


Abbildung 3.10: Warenkorb Storefront Übersicht

Quelle: Eigene Darstellung

Hierbei wurde zwei Mal das zugrunde liegende `product.html.twig` Template erweitert. Das erste ist für den Konfigurator selbst, aber das zweite (`product_child.html.twig`) wird mittels include in das „Parent"-Template einbezogen. Damit werden zwar auch bereits vorhandene Funktionen genutzt, aber in Bezug auf die visuelle Darstellung wirkt es auch einheitlicher. Zusätzlich werden zukünftig auch die Kinderprodukte bei Theme-Anpassungen direkt einbezogen, was ein simples Anpassen und Einbeziehen ermöglicht. Beim Austesten im Kundenshop fiel jedoch auf, dass die Kompatibilität mit einem bereits im Shop installierten Bundle-Plugin erreicht werden musste. Dieses Fremd-Plugin implementiert seine eigene Darstellung von Child-Items, was zu einer Dopplung in der visuellen Präsentation führte. Durch eine Analyse des Bundle-Plugin-Codes konnte eine Lösung implementiert werden, die Konflikte verhindert, indem die Templates bedingt ausgeführt werden – das Bundle-Template wird übersprungen, wenn es sich um ein LineItem des Palettenkonfigurators handelt. Dieser Ansatz ermöglichte die Koexistenz beider Plugins ohne gegenseitige Beeinträchtigung.

3.5.5 Sicherheitsimplementierung

Ein zentraler Aspekt war die Absicherung gegen direkte Bestellungen der Optionsprodukte. Dazu wurde auf Serverseite ein Cart-Validator auf Basis der Shopware-6-

Dokumentation [22] implementiert. Dabei wurde der Code der Fehlermeldung nicht weiter angepasst und nur das Snippet verändert. Beim Hinzufügen von Produkten in den Warenkorb wird geprüft, ob einzelne Line-Items Custom-Fields der Optionsprodukte aufweisen, ohne selbst ein Kind zu sein. Wenn dem so ist, dann entfernt der Cart-Validator diese Line-Items wieder. Anschließend wird eine Fehlermeldung erzeugt, die dem Kunden mitteilt, dass das hinzugefügte Produkt Teil eines Konfigurators ist (vgl. Abb. 3.11).

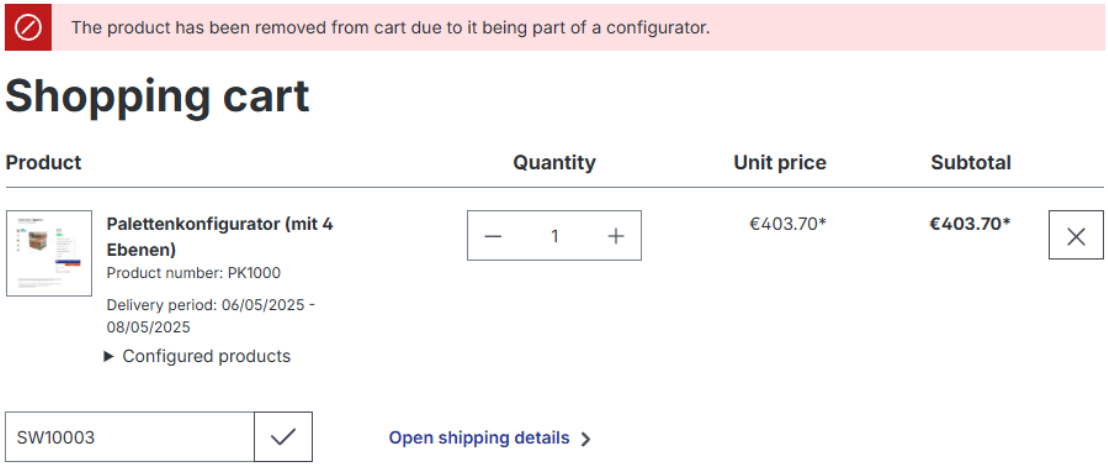


Abbildung 3.11: Warenkorb Fehlermeldung

Quelle: Eigene Darstellung

```php
public function validate(Cart $cart, CartDataCollection $data,
    SalesChannelContext $context): void
{
  foreach ($cart->getLineItems() as $item) {
    // Nur im Konfigurator erzeugte Options-LineItems behalten
    if ($item->getPayloadValue('customFields')['
        wbn_pallets_configurator_custom_field_set_configurator_option'] ??
        false) {
      continue;
    }
    // ungültiges Options-Produkt entfernen und Fehler melden
    if ($item->getType() === LineItem::PRODUCT_LINE_ITEM_TYPE
    && $item->getLabel() === 'Option' ) {
      $cart->remove($item->getId());
      $cart->addErrors([ new CartError(
      'Bitte konfigurieren Sie dieses Produkt über den Konfigurator.'
      ) ]);
    }
  }
}
```

Listing 8: Auszug aus `WbnPalletsConfiguratorCartValidator.php`

Ergänzend wurden im Frontend auf Listing- und Detailseiten der Optionsprodukte die „In den Warenkorb“-Buttons deaktiviert. Auf der Produktdetailseite des Konfigurator-Produkts sorgt zusätzlich eine Checkbox dafür, dass der „In den Warenkorb“-Button nur aktiviert wird, wenn der Kunde mindestens eine Option gewählt und die Zustimmung per Checkbox erteilt hat. Die Extra-Verifizierung verfolgt gleich zwei Absichten. Zum einen kann es Fehlereingaben durch den Kunden verringern, wenn dieser daran erinnert wird, sich die Konfiguration noch einmal anzusehen. Zusätzlich kann sich der Shopbetreiber durch die noch ausstehende Integration eines Widerrufsausschlusses rechtlich absichern. (vgl. Abb. 3.12)

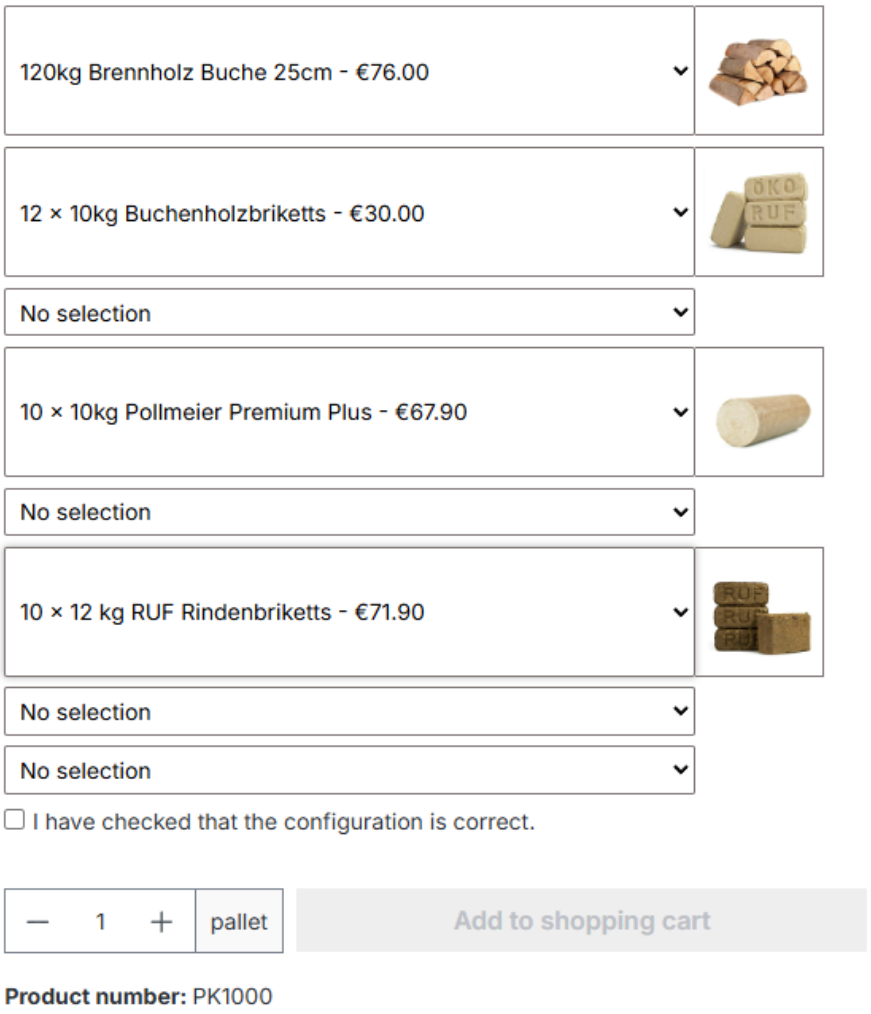


Abbildung 3.12: Konfigurator mit ausgewählten Optionen ohne Checkbox

Quelle: Eigene Darstellung

3.5.6 Optimierung und Finalisierung

Nach der grundlegenden Implementierung aller erforderlichen Funktionalitäten folgte eine Phase der Optimierung und Finalisierung, in der sowohl die technische Qualität als auch die Benutzerfreundlichkeit des Plugins verbessert wurden.

Ein Aspekt der Optimierung betraf die Performance der Bilddarstellung. Die ursprüngliche Implementierung verwendete einfache <img>-tags, die bei jeder Konfigurationsänderung den Link einfügten und luden. Das erstmalige Laden der Bilder bei Optionsauswahl war dabei klar zu erkennen. Deswegen wurde diese Version durch die Integration der Shopware-6-eigenen 'sw_thumbnails' ersetzt. Die Vorteile dieser Version sind zum einen Lazy Loading, aber auch die Bereitstellung der Bilder in verschiedenen Größen, basierend auf den Maßen des Endgeräts.

Ein weiterer Fokus lag auf der Verbesserung der Code-Qualität und Wartbarkeit. Hierzu wurden mehrere Refactoring-Maßnahmen durchgeführt, darunter die Bereinigung ungenutzter Code-Abschnitte, die während der explorativen Entwicklungsphase entstanden waren, sowie die Umbenennung von Dateien und Variablen, um eine konsistente und verständlichere Benennung zu erreichen.

Die Vorbereitung auf Theme-Anpassungen bildete einen weiteren Aspekt der Finalisierung. Hierzu wurden sorgfältig ausgewählte CSS-Klassen und Twig-Blöcke definiert, die eine gezielte Anpassung des Erscheinungsbilds durch Theme-Entwickler ermöglichen. Ebenso wurde versucht W3C Ansätze zu verfolgen, für eine möglichst barrierefreie Webseite. Diese Maßnahmen tragen zur langfristigen Flexibilität und Anpassbarkeit des Plugins bei.

Um das Element, welches in den Erlebniswelten eingesetzt werden kann, allgemein zugänglicher zu machen, wurde überlegt, wie man es an anderer Stelle hinzufügen kann. Dabei ist aufgefallen, dass außerhalb der Produktseite das Element nicht die benötigten Daten bekommen hat. Das war zu erwarten, da der Subscriber sich auf das Laden der Produktseite bezieht. Da das erweiterte Template (vgl. 3.5.3) sich grundsätzlich im gleichen Element befand, war es sinnvoll, die Funktionalität des Subscribers in einen Data Resolver umziehen zu lassen. Ein Data Resolver wird in Shopware 6 speziell für das Verarbeiten von Daten in Verbindung mit einem Element verwendet. Es wurden beide Lösungen im Projekt beibehalten, um den Prozess darzustellen.

Nach Abschluss der Optimierungsphase wurde das Plugin dem Kunden in einem Online-Meeting vorgestellt, um Feedback einzuholen und eventuelle Anpassungs-

wünsche zu identifizieren. Daraus ergab sich ein spezieller Wunsch, der noch angepasst werden wird. Der Shopbetreiber möchte, dass die aktuellen Produkttypen umbenannt und inhaltlich angepasst werden. Der Wunsch ist eine Anpassung des Custom Fields „Produkt-Typ“ (vgl. Abb. 3.6) und der damit verbundenen Funktionalität, wodurch der Shopbetreiber individueller entscheiden kann, wie oft ein Optionsprodukt gewählt werden darf.

Des Weiteren gibt es Ansätze im Code, für die optimalere Lösungen gefunden werden können. So könnte man für die Bereitstellung der Daten vom Twig-Template an das JavaScript-Plugin einen AJAX-Call umsetzen, statt dem aktuellen direkten Einfügen in den HTML-Code über das <data>-Attribut. Dies sind aber Verbesserungen, die nicht mehr in den zeitlichen Rahmen gepasst haben.

3.6 Testing und Qualitätssicherung

Im Rahmen der Qualitätssicherung wurden ausschließlich manuelle Tests durchgeführt. Dabei wurde die Ausgabe der Browser-Konsole mit JavaScript genutzt, um zu verifizieren, dass alle Konfigurationsdaten korrekt übergeben und verarbeitet werden. Auf der Serverseite wurden per `print_r()` in Kombination mit `die()` die empfangenen Parameter validiert und die zugrundeliegenden Datenstrukturen untersucht. Ebenso wurde über die bereits erwähnten ChromeDev-Tools die Netzwerkansicht verwendet, um zu überprüfen, dass das richtige POST-Ereignis ausgelöst wurde und die Daten an die korrekte Route gesendet werden (vgl. Abb. 3.13).

Anschließend wurden verschiedene Konfigurationen erstellt, in den Warenkorb gelegt, bearbeitet und bestellt. Es wurde versucht, neben den typischen Szenarien auch ungewöhnliche gezielt zu testen, um das Verhalten des Konfigurators unter verschiedenen Bedingungen zu prüfen. Abschließend wurden mehrere Testbestellungen ausgelöst und deren Inhalt kontrolliert, um sicherzustellen, dass die ausgewählten Optionen und zugehörigen Produktdaten korrekt in den Warenkorb und in den abschließenden Bestellprozess übernommen werden. Besondere Aufmerksamkeit wurde Szenarien gewidmet, die in früheren Entwicklungsphasen zu Problemen geführt hatten, wie dem Löschen von Konfigurationen aus dem Warenkorb, und kritischen Stellen wie der Preisberechnung.

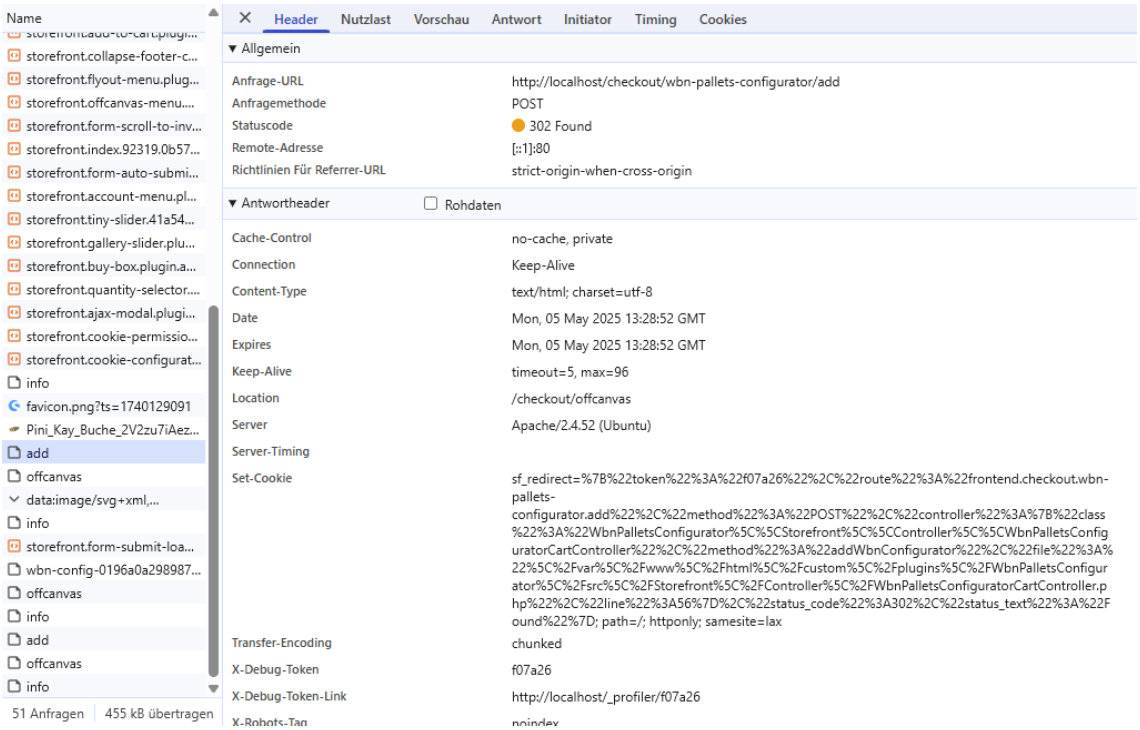


Abbildung 3.13: ChromeDev Netzwerkübersicht Route Add

Quelle: Eigene Darstellung

Ein kritisches Problem, das während der Entwicklung identifiziert und behoben wurde, betraf die Verwaltung der Nested Line Items bei Änderungen des Warenkorbs. Ursprünglich führte das Löschen einer konfigurierten Palette dazu, dass auch die Nested Line Items anderer Konfigurationen im Warenkorb betroffen waren, was zu inkonsistenten Zuständen und inkorrekten Bestellungen führte. Die Lösung bestand in der Einführung einzigartiger IDs für jedes Nested Line Item, wodurch eine präzise Identifikation und Manipulation einzelner Elemente ermöglicht wurde.

4 Ergebnisse und Zukunftsaussichten

Im folgenden werden die erreichten Ergebnisse kritisch und mögliche zukünftige Weiterentwicklungen betrachtet.

4.1 Zusammenfassung der Ergebnisse

4.1.1 Erreichte Funktionalität

Die entwickelte Lösung erfüllt die ursprünglichen Anforderungen der Stakeholder. Dabei wurde insbesondere darauf geachtet, dass die identifizierten funktionalen und nicht-funktionalen (vgl. Tabellen 2, 3) erfüllt werden.

Bei den funktionalen Anforderungen wurden sämtliche als „Muss“-Priorität eingestuften Anforderungen erfolgreich umgesetzt. Dazu zählen unter anderem die Konfigurierbarkeit und Verwaltung des Konfigurators im Adminbereich (F1), die Auswahlmöglichkeit der 8 Ebenen (F2), die dynamische Preis- und Gewichtsberechnung (F5) sowie die Persistenz der Konfiguration während des gesamten Kaufprozesses (F6). Auch die Validierung im Warenkorb mit der automatischen Entfernung ungültiger Konfigurationen (F7) wurde implementiert, kann aber noch erweitert werden mit genaueren Fehlermeldungen.

Darüber hinaus wurden auch die nicht-funktionalen Anforderungen erfüllt. Die Kompatibilität mit Shopware-Versionen ab 6.10 und gängigen Browsern (NF1), keine merkbare Beeinträchtigung bei Nutzerinteraktionen (NF2) sowie die Wartbarkeit durch Einhaltung der Shopware-Konventionen (NF5) sind sichergestellt. Auch die als „Soll“-Anforderungen klassifizierten Aspekte, wie die visuelle Repräsentation der gewählten Optionen (F3) und das responsive Design mit intuitiver Benutzerführung (NF4), wurden weitgehend umgesetzt. Jedoch gibt es in Bezug auf das responsive Design noch Verbesserungsmöglichkeiten. Die Gesamtheit der realisierten Funktionen zeigt aber, dass die Lösung den definierten Anforderungen entspricht und die Erwartungen der Stakeholder erfüllt.

Zusätzlich wurden zuvor nicht definierte Anforderungen, wie die Checkbox oder die Möglichkeit Produkte über den Abverkauf vorübergehend zu deaktivieren, eingefügt.

4.1.2 Benutzerfreundlichkeit

Von Anfang an wurde großer Wert auf eine einfache und übersichtliche Benutzeroberfläche gelegt. Da der Funktionsumfang vergleichsweise gering war, bestand auch nur ein geringes Risiko von „Mass Confusion“ 2.2.2. Dennoch kann selbst ein simpler Konfigurator bei unzureichender Planung und Gestaltung schwer verständlich und somit wenig benutzerfreundlich sein.

Die Interaktionsabläufe sind eindeutig und beschränken sich auf wenige, klar verständliche Schritte. Durch die vergleichsweise geringe Anzahl an Konfigurationsmöglichkeiten ist die kognitive Belastung für die Nutzer von vornherein gering. Dies unterstützt eine gute Orientierung beim Konfigurieren und ermöglicht auch unerfahrenen Endnutzern eine einfache Nutzung. [12]

Bewertet wurde die Benutzerfreundlichkeit durch eine gezielte Analyse der Benutzeroberfläche sowie der Bedienbarkeit. Dabei wurde untersucht, inwieweit die Gestaltung eine intuitive Nutzung unterstützt und wie übersichtlich die Implementation ist. Das Ergebnis bestätigt, dass die gewählte Gestaltung sowohl funktional als auch nutzerzentriert ist.

4.1.3 Barrierefreiheit

Obwohl in der Anforderungsanalyse keine expliziten Anforderungen zur Barrierefreiheit formuliert wurden, ist dieses Thema im Kontext moderner Webanwendungen von wachsender Bedeutung. In der Europäischen Union regelt der „European Accessibility Act“ (EAA) verbindliche Standards zur digitalen Barrierefreiheit [7]. Während bereits seit mehreren Jahren öffentliche Stellen zur Umsetzung barrierefreier Webseiten verpflichtet sind, wird ab dem 28.06.2025 auch der private Sektor, insbesondere auch Online-Shops, zur Einhaltung entsprechender Anforderungen verpflichtet.

Dies betrifft insbesondere Aspekte wie die Bedienbarkeit per Tastatur, die Nutzung von Screenreadern und die semantische Strukturierung von Inhalten. Eine vollständige Umsetzung der WCAG-Richtlinien (Web Content Accessibility Guidelines)[37] ist daher langfristig auch für E-Commerce-Lösungen erforderlich. Shopware hat auf diese Entwicklungen reagiert und verfolgt das Ziel, die eigene Plattform schrittweise an den internationalen Standard der WCAG-Richtlinien anzugleichen. Shopware plant die administrativen Oberflächen sowie die Storefront kontinuierlich auf Barrierefreiheit zu optimieren. [18]

Im Sinne einer zukunftssicheren Gestaltung wurde bei der Implementierung des Konfigurators darauf geachtet, potenzielle Barrieren für Nutzer so gering wie möglich zu halten. Da sich aber zugrunde liegende Themes vorrangig auf visuelle Anpassungen konzentrieren (vgl. Abschnitt 2.1.2), wurden im Plugin lediglich minimale visuelle Erweiterungen vorgenommen. Ziel war es, eine möglichst reibungslose und barrierearme Nutzung des Konfigurators im „Out-of-the-box“-Zustand zu gewährleisten.

Besonderes Augenmerk wurde auf die semantisch korrekte Strukturierung des Formulars durch `<label>`-Elemente gelegt. Die Produktbilder wurden mit `alt`-Attributen versehen, wodurch sie auch von Screenreadern zugänglich gemacht werden. Die dynamisch per JavaScript aktualisierte Gewichtsanzeige ist mit den Attributen `role="status"` und `aria-live="polite"` versehen, um Änderungen dezent, aber wahrnehmbar zu kommunizieren. Schließlich wurde die Checkbox zur Bestätigung am Ende des Formulars mit den Attributen `required` und `aria-required="true"` ausgezeichnet, um ihre Funktion als Pflichtfeld explizit zu kennzeichnen. Diese Anpassungen ermöglichen es assistiven Technologien, eine präzise Interpretation der strukturellen Zusammenhänge vorzunehmen.

4.1.4 Wartung und Pflege des Plugins

Da darauf geachtet wurde, dass das Plugin sich an die Shopware-Architektur hält, ist es gut wartbar. Hauptanpassungen, die zukünftig zu erwarten sind, sind Änderungen, die bei einem Update der Shopware-6-Version aufkommen können. So wurde z. B. in der Version 6.6 Vue von Version 2 auf Version 3 aufgewertet[32], was Breaking Changes eingeführt hat. Ebenso kann es passieren, dass Shopware Komponenten entfernt, verschiebt, umbenennt usw., wodurch betroffener Code überarbeitet werden muss.

Durch den Aufbau des Plugins, eindeutige Variablenbenennung und Kommentare sollte jedoch selbst für fremde Entwickler ein einfaches Anpassen möglich sein. Es wurde auch speziell darauf geachtet, den Storefront-Code in Twig-Blöcke zu setzen und mit Klassen zu versehen, wodurch Frontend-Entwickler einfacher spezifische Template-Anpassungen vornehmen können, wenn gewünscht.

4.1.5 Limitationen der aktuellen Implementierung

Trotz der erfolgreichen Umsetzung weist die aktuelle Implementierung Limitationen auf, die in zukünftigen Anpassungen oder Verwendungen adressiert und weiterent-

wickelt werden könnten.

Begrenzte Visualisierung: Die aktuelle Lösung beschränkt sich auf die Darstellung statischer Produktbilder ohne dynamische Visualisierung der konfigurierten Palette. Eine erweiterte Visualisierung wäre eine signifikante Verbesserung für die Benutzerfreundlichkeit und wird als der Erweiterungspunkt mit der höchsten Priorität eingeschätzt.

Feste Anzahl von Auswahlfeldern: Die Implementierung ist auf acht Auswahlfelder festgelegt. Ein variableres Konzept könnte die Anpassungsfähigkeit an verschiedene Konfigurationsanforderungen ermöglichen. Dieser Punkt lässt sich auf den restlichen Konfigurator ausweiten. So sind auch die Anpassungsmöglichkeiten für das Frontend begrenzt. Die visuelle Gestaltung des Konfigurators ist mit der Auswahl eines Produktbildes für die Option begrenzt. Eine Administration für Darstellungsoptionen wäre denkbar und könnte Anpassungsmöglichkeiten für Shop-Betreiber ohne Entwicklerkenntnisse bieten.

Komplexe Abhängigkeitsregeln: Die aktuelle Lösung unterstützt primär einfache regelbasierte Abhängigkeiten (Sperrung bereits gewählter Optionen und bei Abverkauf). Komplexere Abhängigkeitsregeln mit bedingten Abhängigkeiten würden zusätzliche Implementierungen erfordern.

Barrierefreiheit: Aktuell fehlen noch erläuternde Fehlermeldungen und `ARIA`-Attribute, die beim Ausfüllen und Korrigieren von Formularen unterstützen würden. Zudem ist `aria-hidden` bei den Produktbildern bisher nicht dynamisch umgesetzt. Eine JavaScript-basierte Steuerung ist erforderlich, um die Bilderanzeige aktiver Optionen für Screenreader zugänglich zu machen.

Sicherheitsvalidierung und Error-Handling: Es wurde zwar eine grundlegende Validierung mit entsprechenden Fehlermeldungen implementiert, jedoch gibt es hier noch großes Potenzial, vor allem in Kombination mit anderen Anpassungen.

Diese Limitationen stellen potenzielle Ansatzpunkte für zukünftige Weiterentwicklungen dar, beeinträchtigen jedoch nicht die grundlegende Funktionalität und den Nutzen der aktuellen Lösung. Sie reflektieren den notwendigen Kompromiss zwischen Umfang, Zeit und Ressourcen. Dennoch hat das Plugin noch genügend Weiterentwicklungspotenzial.

4.1.6 Weiterentwicklung

Basierend auf den identifizierten Limitationen und zusätzlichen Überlegungen ergeben sich folgende Weiterentwicklungspotenziale des Konfigurators:

1. **Erweiterte Visualisierung:** Die aktuelle Darstellung der konfigurierten Palette beschränkt sich auf statische Einzelbilder der ausgewählten Optionen. Für eine verbesserte Nutzerführung wäre eine dynamische Visualisierung der Konfiguration wünschenswert. Hierzu zählt beispielsweise die Möglichkeit, überlagerte Bildkompositionen zu generieren, die die gewählten Ebenen visuell zusammenführen und damit die Konfiguration als Gesamtbild darstellen. Auch interaktive Elemente wie schrittweises Ein- und Ausblenden von Ebenen oder konfigurationsabhängige Darstellungen könnten das Benutzererlebnis deutlich verbessern. Technologien wie serverseitige Bildgenerierung oder clientseitige Layer-Kompositionen auf Basis von HTML5-Canvas oder SVG bieten hierfür geeignete Lösungsansätze. Als weiterführende Ausbaustufe käme auch eine 3D-Visualisierung in Betracht.

2. **Flexibilisierung des Konfigurators:** Die aktuell festgelegte Anzahl von acht Auswahlfeldern limitiert die Anpassbarkeit. Eine flexible Konfigurationslogik, die Anzahl und Art der Felder variabel gestaltet und optional auch von vorherigen Nutzereingaben abhängig macht, würde die Einsatzmöglichkeiten deutlich erweitern. Man könnte sogar noch weitergehen und die Konfiguration von mehreren Produktkonfiguratoren durch eine zentrale Administrationsoberfläche erlauben, wie z. B. in „Custom Products“[1] oder der „CMSFactory“[2].

3. **Erweiterte Benutzerinteraktionen:** Perspektivisch könnten Funktionen wie das Speichern, Teilen und Vergleichen von Konfigurationen ergänzt werden. Diese Optionen sind insbesondere für komplexere Produktkonfiguratoren relevant und könnten potenziell die Kundenzufriedenheit und Konversionsraten steigern.

Diese Weiterentwicklungspotenziale bieten Ansatzpunkte für zukünftige Projekte und könnten den Funktionsumfang und Nutzen des implementierten Konfigurators signifikant erweitern.

[1] `https://docs.shopware.com/en/shopware-6-en/extensions/customproducts`

[2] `https://store.shopware.com/de/aku2334640916530m/cms-factory-element-builder-fuer-erlebniswelten.html`

4.2 Relevante Enwticklungen

Die rasante Entwicklung des Webs sowie der weltweite Bedeutungszuwachs des E-Commerce verlangen nach kontinuierlichen technischen Innovationen. In diesem Abschnitt werden ein paar Themen angesprochen, die über die in dieser Arbeit vorgestellten Ansätze und Umsetzungen hinausgehen. Sie können verwendet werden, um weitere Recherchen und Lösungsansätze anzugehen.

Die Nutzung von Augmented-Reality-Technologien (AR) bietet neue Möglichkeiten für die Darstellung von Produkten. Es wird möglich, Objekte direkt in der realen Umgebung darzustellen, was besonders bei großen Produkten wie Möbeln oder auch Paletten hilfreich sein kann. Die Visualisierung in Originalgröße ermöglicht den Kunden, das Produkt besser einzuschätzen, was die Kaufentscheidung beeinflussen kann und die Kundenzufriedenheit erhöht. [1, 16, 38]

Progressive Web Apps (PWAs) gewinnen ebenfalls immer mehr an Bedeutung [8], da sie die Vorteile klassischer Webanwendungen mit den Funktionalitäten nativer Apps verbinden. Sie zeichnen sich durch Plattformunabhängigkeit, Offline-Fähigkeit und eine verbesserte Nutzererfahrung aus. Durch den Einsatz moderner Webtechnologien wie Service Workern und Web-App-Manifests bieten PWAs zudem eine effiziente Lösung und können auch in Bezug auf Visualisierungen und Performanceoptimierung Einsatz finden. [5]

Aber auch der Einsatz von Künstlicher Intelligenz (KI) kann Neuerungen mit sich bringen. Hierbei lassen sich nützliche Konfigurationsvorschläge oder relevante Produktempfehlungen generieren[15]. Ebenso gibt es die Überlegung, Chatbots in die Konfiguration einzubeziehen. Statt also über Auswahlfelder ein Produkt zu erstellen, könnte man dieses potenziell durch Text umschreiben[15]. Die Integration solcher Technologien könnte den Konfigurationsprozess vereinfachen und gleichzeitig stärker personalisieren.

Die beschriebenen Trends in der Entwicklung betonen die vielfältigen Möglichkeiten des E-Commerce und der Konfiguratoren im Online-Handel. Durch die Integration neuer Technologien wie KI, AR und PWA ergeben sich neue Möglichkeiten für kreative Erweiterungen.

Literatur

[1] Aheleroff, Shohin ; Mostashiri, Naser ; Xu, Xun ; Zhong, Ray Y.: Mass Personalisation as a Service in Industry 4.0: A Resilient Response Case Study. In: *Advanced Engineering Informatics* 50 (2021), 101438. http://dx.doi.org/https://doi.org/10.1016/j.aei.2021.101438. – DOI https://doi.org/10.1016/j.aei.2021.101438. – ISSN 1474–0346

[2] Aldanondo, Michel ; Hadj-Hamou, Khaled ; Moynard, Guillaume ; Lamothe, Jacques: Mass customization and configuration: Requirement analysis and constraint based modeling propositions. In: *Integrated computer-aided engineering* 10 (2003), Nr. 2, S. 177–189. – ISSN 1069–2509

[3] Blecker, Thorsten ; Abdelkafi, Nizar ; Kreutler, Gerold ; Friedrich, Gerhard: Product Configuration Systems: State of the Art, Conceptualization and Extensions. (2004), 02, S. 26–36

[4] BuiltWith Pty Ltd: *eCommerce Usage Distribution in Germany*. https://trends.builtwith.com/shop/country/Germany, Abruf: 05.05.2025

[5] Cherukuri, Bangar R.: Progressive Web Apps (PWAs): Enhancing User Experience through Modern Web Development. In: *International Journal of Science and Research (IJSR)* 13 (2024), 10, S. 1550–1560. http://dx.doi.org/10.21275/MS241022095359. – DOI 10.21275/MS241022095359

[6] Deckert, Ronald: *Ausgewählte Grundlagen rund um Industrie 4.0*. Wiesbaden : Springer Fachmedien Wiesbaden, 2019. – 7–19 S. http://dx.doi.org/10.1007/978-3-658-23847-6_2. http://dx.doi.org/10.1007/978-3-658-23847-6_2. – ISBN 978–3–658–23847–6

[7] Europäisches Parlament, Rat der Europäischen U.: Richtlinie (EU) 2019/882 des Europäischen Parlaments und des Rates vom 17. April 2019 über die Barrierefreiheitsanforderungen für Produkte und Dienstleistungen (Text von Bedeutung für den EWR). (2019). https://eur-lex.europa.eu/eli/dir/2019/882/oj

[8] Grand View Research, Inc.: *PWA Market Size & Trends*. https://www.grandviewresearch.com/industry-analysis/progressive-web-apps-pwa-market-report, Abruf: 11.05.2025

[9] Gutai, Andrea ; Havzi, Sara ; Nikolic, Danilo ; Dakic, Dusanka: *Potential of Mass Customization using Product Configurators*

[10] Harfeldt-Berg, Magnus ; and, Jan O.: The customer order decoupling point in empirical operations and supply chain management research: a systematic literature review and framework. In: *International Journal of Production Research* 62 (2024), Nr. 17, 6380–6399. http://dx.doi.org/10.1080/00207543.2024.2314164. – DOI 10.1080/00207543.2024.2314164

[11] Haug, Anders ; Hvam, Lars ; Mortensen, Niels H.: Definition and evaluation of product configurator development strategies. In: *Computers in Industry* 63 (2012), Nr. 5, 471-481. http://dx.doi.org/https://doi.org/10.1016/j.compind.2012.02.001. – DOI https://doi.org/10.1016/j.compind.2012.02.001. – ISSN 0166–3615

[12] Huffman, Cynthia ; Kahn, Barbara E.: Variety for sale: Mass customization or mass confusion? In: *Journal of Retailing* 74 (1998), Nr. 4, 491-513. http://dx.doi.org/https://doi.org/10.1016/S0022-4359(99)80105-5. – DOI https://doi.org/10.1016/S0022–4359(99)80105–5. – ISSN 0022–4359

[13] Leclercq, Tony ; Abbasi, Ebrahim K. ; Dumas, Bruno ; Remiche, Marie-Ange ; Heymans, Patrick: Essential Expectations of Users of Web Configurators: An Empirical Survey. In: *Proc. ACM Hum.-Comput. Interact.* 6 (2022), Juni, Nr. EICS. http://dx.doi.org/10.1145/3534519. – DOI 10.1145/3534519

[14] Merle, Aurélie ; Chandon, Jean-Louis ; Roux, Elyette ; Alizon, Fabrice: Perceived Value of the Mass-Customized Product and Mass Customization Experience for Individual Consumers. In: *Production and operations management* 19 (2010), Nr. 5, S. 503–514. – ISSN 1059–1478

[15] Raji, Mustafa ; Olodo, Hameedat ; Oke, Timothy ; Addy, Wilhelmina ; Oyewole, Adedoyin: E-commerce and consumer behavior: A review of AI-powered personalization and market trends. In: *GSC Advanced Research and Reviews* 18 (2024), 03, S. 066–077. http://dx.doi.org/10.30574/gscarr.2024.18.3.0090. – DOI 10.30574/gscarr.2024.18.3.0090

[16] SELIM, NOHA ; Dinana, Hesham: The Impact of Augmented Reality on Customer Purchasing Intention. In: 16 (2025), 01, 1468-1492. http://dx.doi.org/10.21608/jces.2025.421305. – DOI 10.21608/jces.2025.421305

[17] Shopify Inc.: *Shopify App Store*. https://apps.shopify.com. Version: 2025, Abruf: 09.05.2025. – 'Mit über 8.000 Apps […]'

[18] Shopware AG: *Der nächste Schritt in Richtung Barrierefreiheit: Bereite dich auf den EAA vor und profitiere von Shopwares Neuerungen*. https://www.shopware.com/de/news/barrierefreiheit-im-e-commerce/, Abruf: 10.05.2025

[19] Shopware AG: *Shopware Brand*. https://brand.shopware.com, Abruf: 05.05.2025

[20] Shopware AG: *Shopware Docs Add Cart Collector/Processor*. https://developer.shopware.com/docs/guides/plugins/plugins/checkout/cart/add-cart-processor-collector.html, Abruf: 05.05.2025

[21] Shopware AG: *Shopware Docs Add Cart Items*. https://developer.shopware.com/docs/guides/plugins/plugins/checkout/cart/add-cart-items.html, Abruf: 05.05.2025

[22] Shopware AG: *Shopware Docs Add Cart Validator*. https://developer.shopware.com/docs/guides/plugins/plugins/checkout/cart/add-cart-validator.html, Abruf: 05.05.2025

[23] Shopware AG: *Shopware Docs Administration*. https://developer.shopware.com/docs/concepts/framework/architecture/administration-concept.html, Abruf: 09.05.2025

[24] Shopware AG: *Shopware Docs Cart*. https://developer.shopware.com/docs/concepts/commerce/checkout-concept/cart.html, Abruf: 05.05.2025

[25] Shopware AG: *Shopware Docs Custom Products*. https://docs.shopware.com/en/shopware-6-en/extensions/customproducts, Abruf: 09.05.2025

[26] Shopware AG: *Shopware Docs Data Abstraction Layer*. https://developer.shopware.com/docs/concepts/framework/data-abstraction-layer.html, Abruf: 09.05.2025

[27] Shopware AG: *Shopware Docs Extendability*. https://developer.shopware.com/docs/resources/guidelines/code/core/extendability.html, Abruf: 09.05.2025

[28] Shopware AG: *Shopware Docs Plugin Base Guide*. https://developer.shopware.com/docs/guides/plugins/plugins/plugin-base-guide.html, Abruf: 09.05.2025

[29] Shopware AG: *Shopware Docs Plugin Lifecycle Methods*. https://developer.shopware.com/docs/guides/plugins/plugins/plugin-fundamentals/plugin-lifecycle.html, Abruf: 06.05.2025

[30] Shopware AG: *Shopware Docs Storefront*. https://developer.shopware.com/docs/concepts/framework/architecture/storefront-concept.html, Abruf: 09.05.2025

[31] Shopware AG: *Shopware Docs System requirements*. https://docs.shopware.com/en/shopware-6-en/first-steps/system-requirements?category=shopware-6-en/getting-started, Abruf: 09.05.2025

[32] Shopware AG: *Shopware Docs Vue 3 Upgrade*. https://developer.shopware.com/docs/resources/references/upgrades/administration/vue3.html, Abruf: 05.05.2025

[33] Shopware AG: *Erweiterungen*. https://store.shopware.com/de/erweiterungen/. Version: 2025, Abruf: 09.05.2025. – '3072 Erweiterungen'

[34] Stack Overflow: *Where can I find more information about Nested Line Items?* https://stackoverflow.com/questions/76916796/where-can-i-find-more-information-about-nested-line-items, Abruf: 12.05.2025. – Es handelt sich um die Antwort des Nutzers dneustadt.

[35] United Nations Conference on Trade and Development: COVID-19 boost to e-commerce sustained into 2021, new UNCTAD figures show. In: *UNCTAD* (2021). https://unctad.org/news/covid-19-boost-e-commerce-sustained-2021-new-unctad-figures-show, Abruf: 08.05.2025

[36] Vijayakumar, Suchetha ; K, Krishna P. ; Holla M., Raviraja: Assessing the Effectiveness of MoSCoW Prioritization in Software Development: A Holistic Analysis across Methodologies. In: *EAI Endorsed Transactions on Internet of Things* 10 (2024), Oct. http://dx.doi.org/10.4108/eetiot.6515. – DOI 10.4108/eetiot.6515

[37] World Wide Web Consortium (W3C): *Web Content Accessibility Guidelines (WCAG) 2.1*. https://www.w3.org/TR/WCAG21/, 2019

[38] Yang, Jingyi ; Lin, Zhibin: From screen to reality: How AR drives consumer engagement and purchase intention. In: *Journal of Digital Economy* 3 (2024), 37-46. http://dx.doi.org/https://doi.org/10.1016/j.jdec.2024.07.001. – DOI https://doi.org/10.1016/j.jdec.2024.07.001. – ISSN 2773–0670

[39] Yi, Minzhe ; Huang, Zihao ; Yu, Yuxiang: Creating a Sustainable E-Commerce Environment: The Impact of Product Configurator Interaction Design on Consumer Personalized Customization Experience. In: *Sustainability* 14 (2022), Nr. 23. http://dx.doi.org/10.3390/su142315903. – DOI 10.3390/su142315903. – ISSN 2071–1050

Verzeichnis weiterer Hilfsmittel

Code Quellen

Da es sich bei der Erweiterung um ein Shopware 6 Plugin handelt, ist es sogar gewünscht, sich in vielerlei Hinsicht an die gegebene Struktur zu halten. Dadurch sichert man zwar die Integration, jedoch hat dies auch zur Folge, dass einiger Code sehr ähnlich zu bereits existierendem Shopware 6 Code ist. Es soll mit der folgenden Übersicht ein transparenter Überblick über die Herkunft von kopiertem oder sehr ähnlichem Code erstellt werden.

Datei: custom/plugins/WbnPalletsConfigurator/src/Storefront/Controller/WbnPalletsConfiguratorCartController.php
Betroffen: Methode getLineItem() mit minimalen Anpassungen
Quelle: `https://stackoverflow.com/questions/76916796/where-can-i-find-more-information-about-nested-line-items`

Datei: custom/plugins/WbnPalletsConfigurator/src/Storefront/Controller/WbnPalletsConfiguratorCartController.php
Betroffen: Methode getLineItemArray()
Quelle: Aus Shopware 6 Datei vendor/shopware/storefront/Controller/CartLineItemController.php - Zeile 357

Datei: custom/plugins/WbnPalletsConfigurator/src/Core/Checkout/Cart/-Custom/Error/WbnPalletsConfiguratorCartBlockedError.php
Betroffen: gesamte Datei
Quelle: `https://developer.shopware.com/docs/guides/plugins/plugins/checkout/cart/add-cart-validator.html`

Datei: custom/plugins/WbnPalletsConfigurator/src/Resources/config/routes.xml
Betroffen: Fast komplett, nur Anpassung für implementierten Controller
Quelle: `https://developer.shopware.com/docs/guides/plugins/plugins/storefront/add-custom-controller.html`

Weitere Hilfsmittel

Hilfsmittel	Zweck / Anmerkung
`https://quillbot.com/grammar-check`	Grammatik-Prüfung und alternative Formulierungen
LLM claude sonnet 3.7 thinking	Unterstützende Nutzung z.T. Debugging Hilfe und Ideen-Generierung

A Anhang

Anmerkung: Es gehören digitale Anlagen zu dieser Bachelorarbeit. Diese umfassen die PDF-Version der vorliegenden Arbeit und den Quellcode des entstandenen Plugins (WbnPalletsConfigurator).

Erklärung

Ich versichere, dass ich die Bachelorarbeit „Konzeption und Realisierung einer komplexen individuellen Erweiterung für einen Shopware 6 Shop“ selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe

Datum, Ort

Josy Vinnemeier