BACHELORARBEIT

zum Thema

„Integration einer dynamischen Formularkomponente in das Content Management System Smart-Edit“

vorgelegt an der
Dualen Hochschule Gera-Eisenach

von: [redacted]

Matrikelnummer: [redacted]

Studiengang: Praktische Informatik

Praxispartner:
dotSource GmbH
Digital Success right from the Start
Goethestraße 1
07743 Jena

Gutachter der Dualen Hochschule Gera-Eisenach: [redacted]

Gutachter des Praxispartners (Betreuer i.S.v. § 20 (1) BAPrüfO): [redacted]

Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften – auch in digitaler Form – gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH.

I Inhaltsverzeichnis

II Tabellenverzeichnis

III Abbildungsverzeichnis

IV Abkürzungsverzeichnis

Ajax	Asynchronous JavaScript and XML Asynchrone Datenübertragung von Browser und Server über HTTP-Anfragen, ohne die Internetseite neuzuladen
CMS	Content Management System
CSS	Cascading Style Sheets
CSRF	Cross-site request forgery
CSV	Comma-seperated values
DTO	Data Transfer Object
dS-Addon	dotSource-Addon
HAC	Hybris Administration Console
HTTP	Hypertext Transfer Protocol Protokoll zur Datenübertragung
JSP	JavaServer Pages Programmiersprache für die Erstellung von dynamischem HTML
MiB	Mebibyte 1 MiB $\triangleq 2^{20}$ Byte
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WYSIWYG	What you see is what you get

V Anlagenverzeichnis

1 Einleitung

Die dotSource GmbH ist eine Digitalagentur und besitzt einen Bereich für E-Commerce. Eine große Mehrzahl der Projekte integriert Online-Formulare für verschiedene Anwendungszwecke, beispielsweise für die Kontaktaufnahme mit dem Kundensupport. Die Struktur hängt stark vom Verwendungszweck ab. Unterschiedliche Formulare benötigen verschiedene Felder. Für jeden Verwendungszweck müsste ein eigenes Formular entwickelt werden. Das kostet Zeit, da aufgrund dessen andere Entwicklungen hintenanstehen. Gelöst werden kann dieser Umstand, wenn der Kunde in der Lage ist, das gewünschte Formular selbst zu erstellen. Da die Eingabefelder somit nicht mehr statisch vorgegeben sind, ist die Rede von dynamischen Formularen.

Diese wiederkehrende Anforderung wurde bereits in mehreren SAP Commerce[1] Projekten der dotSource GmbH umgesetzt. Um die Projektrealisierung schneller und kosteneffizienter zu gestalten, soll eine allgemeingültige Lösung geschaffen werden. In neueren Versionen von SAP Commerce wird das alte Content Management Systeme (CMS) WCMS durch das neuere SmartEdit ersetzt. Ein häufiger Kritikpunkt des WCMS ist die Trennung der Bearbeitungsansicht von der Darstellung. Das erfordert den ständigen Wechsel zwischen beiden Oberflächen, um Änderungen zu verifizieren. SmartEdit kombiniert die Ansichten miteinander und lässt den Manager der Seiteninhalte sofort das Ergebnis sehen. Aufgrund dessen soll die Integration für SmartEdit im Fokus stehen.

Ziel dieser Bachelorarbeit soll es sein, eine Möglichkeit zu schaffen, dynamische Formulare mittels SmartEdit zu verwalten. Es sollen dazu Lösungsansätze evaluiert werden und die Umsetzung eines gewählten Ansatzes erfolgen. Damit eine Wiederverwendung in zukünftigen Projekten der dotSource GmbH ermöglicht wird, sollen

[1] Das verwendete Shopsystem (siehe Kapitel 4.14.3)

bestehende Implementierungen zusammengefasst und in einer modularisierten Form zugänglich gemacht werden.

Der erste Teil der Arbeit befasst sich mit den Grundlagen zu Modularität und allgemeinen Lösungen für die Erstellung von Formularen. Daraufhin werden zwei Ansätze für die Integration dynamischer Formulare in SmartEdit vorgestellt und miteinander verglichen. Kapitel 4 stellt die eingesetzten Technologien vor.

Kapitel 5 beschreibt die Erstellung eines SAP Commerce Addon, welches bisherige Implementierungen vereint. Es wird eine Entwicklungs- und Testumgebung aufgesetzt. Anschließend werden die Darstellung und Geschäftslogik der Formularkomponente und die ihrer Eingabekomponenten detailliert erklärt. Darauffolgend wird die Integration in ein SAP Commerce Projekt erläutert. Durch die Demonstration von zwei Anwendungsfällen sollen in Kapitel 6 die Möglichkeiten des entwickelten Formulars aufgezeigt werden.

Das Fazit bewertet die entstandene Lösung kritisch und beleuchtet aufgetretene Probleme. Anschließend werden Möglichkeiten zur Weiterführung der Arbeit diskutiert.

2 Grundlagen

2.1 Modularität und Wiederverwendbarkeit

Bei der Entwicklung von Software kommt es häufig vor, dass sich Funktionen ähneln und Code an mehreren Stellen kopiert und neu verwendet wird. Für den Nutzer einer Software hat das keine spürbaren Auswirkungen. Der Entwickler schafft sich jedoch mehrere Probleme mit dieser Vorgehensweise. Die Wartbarkeit solcher Umsetzungen wird deutlich erschwert. Sollte es zu einem Refactoring oder generellen Änderungen der betroffenen Abschnitte kommen, zum Beispiel aufgrund eines neuen Features, so müssen diese an den jeweiligen Stellen einzeln vorgenommen werden. Daraus resultiert mit zunehmender Projektgröße eine unübersichtliche und komplexe Codebasis.

Ein häufig angewandtes Prinzip, um dem vorzubeugen, ist „Don't repeat yourself" (DRY). Es zielt darauf ab, duplizierten Code zu vermeiden, um die Fehleranfälligkeit zu mindern und die Wartbarkeit zu fördern[2]. Beispielsweise werden ähnliche Codeabschnitte in Methoden ausgelagert, welche um Parameter ergänzt werden können. Es reicht lediglich die Funktion aufzurufen. Da eine Anpassung nur noch an einer einzigen Stelle, nämlich in der Funktion, vorgenommen werden muss, wird das vorherige Szenario des Refactoring entschärft. Die genannten Probleme der Komplexität und des nötigen Aufwands werden auf ein Minimum reduziert und die Wiederverwendbarkeit des Codes wird einfacher und effizienter. Das Konzept, mehrere Module zu einem Gesamtsystem zusammenzusetzen, nennt sich Modularität.[3] Die geschilderten Vorteile kommen erst dann richtig zu Geltung, wenn ein Modul wiederverwendbare Funktionalitäten, an mehr als nur einer Stelle einsetzt. Vorher würden sich eine direkte Implementierung und die Verwendung eines Moduls nur anhand des Methodenaufrufs unterscheiden.

---

[2] [Mar08] vgl. S. 48
[3] [ITD19]

Durch die einfache Wiederverwendbarkeit können Entwickler effizienter arbeiten indem sie eine erneute Implementierung der Lösung umgehen. Selbst wenn es sich dabei auf das Kopieren eines Abschnitts mit minimalen Anpassungen beschränkt. Die geringere Komplexität bei diesem Ansatz wirkt sich ebenso positiv auf diese beiden Faktoren aus und fällt bei zunehmender Projektdauer immer stärker ins Gewicht. Allein aufgrund dieses Zusammenwirkens ist ein modularer Aufbau von jeglicher Software erstrebenswert, insbesondere bei umfangreicheren Projekten.

Auf einfachster Ebene wird Modularität schon durch das Auslagern von Code in Funktionen oder durch die Verzeichnisstruktur eines Projekts realisiert. In anderen Umsetzungen lässt sich ein Modul beispielsweise auch in Form von Programmbibliotheken oder externen Schnittstellen realisieren, welche Teilaufgaben eines Systems übernehmen. Diese lassen sich optional zu einem Projekt hinzufügen oder daraus entfernen. Man spricht oft auch von Erweiterungen oder Extensions. Fügt man eine Extension dem Projekt hinzu, so lassen sich die darin vorhandenen Inhalte nutzen. Eine Eigenimplementierung ist nicht nötig und der Aufwand beschränkt sich auf die Anbindung der Extension.

2.2 Zugrundeliegende Quellen

In vielen bekannten CMS besteht die Möglichkeit, Erweiterungen für die Erstellung von Formularen einzubinden. So gibt es für Joomla![4] zahlreiche Extensions, welche eine solche Funktionalität in verschiedenen Ausprägungen anbieten.[5] Auch WordPress[6] kann das ähnlich umfangreich über Plugins bewerkstelligen.[7] Die zwei beliebtesten Plugins WPForms[8] und Ninja Forms[9] kommen dabei auf über 3 Millionen aktive Nutzer

[4] https://www.joomla.de/
[5] https://extensions.joomla.org/category/contacts-and-feedback/forms/
[6] https://de.wordpress.org/
[7] https://de.wordpress.org/plugins/tags/form-builder/
[8] https://de.wordpress.org/plugins/wpforms-lite/
[9] https://de.wordpress.org/plugins/ninja-forms/

(Stand: Juli 2019). Es scheint demnach eine Vielzahl von Nutzern zu geben, welche über CMS dynamische Formulare erstellen. Dieser Umstand rechtfertig die Auseinandersetzung mit ähnlichen Lösungen im SmartEdit von SAP Commerce.

Drew McLellan, ein Entwickler des Perch Runway CMS[10], definiert drei wesentliche Anforderungen an ein System zur Verarbeitung von Formularen. So soll dieses in der Lage sein das gesamte HTML des Formulars mitsamt seiner Eingabefelder und Beschriftungen selbst zu erstellen.[11] Des Weiteren muss das Absenden des Formulars erkannt und der Inhalt der Felder auf bestimmte Vorgaben validiert werden.[12] Zuletzt müssen die Daten weitergereicht werden, um diese zu verarbeiten.[13] Diese Anforderungen können bei der Integration einer Lösung für dynamische Formulare in SAP Commerce berücksichtigt werden.

[10] https://perchrunway.com/
[11] [McL11]
[12] ebenda
[13] ebenda

3 Evaluierung von Ansätzen

3.1 yForms

yForms ist ein Modul für SAP Commerce, welches aus mehreren Extensions besteht. Es ermöglicht das Erstellen von individuellen Formularen, welche der Storefront hinzugefügt werden können. Über einen umfangreichen Konfigurator kann im Backoffice von SAP Commerce eine sogenannte "Form Definition" erstellt werden. Sie definiert ein Formular zum Beispiel hinsichtlich der Eingabefelder, deren Abhängigkeiten und des Layouts. Diese Definition kann nachfolgend beliebig den Inhaltsseiten in der Storefront über SmartEdit hinzugefügt werden.

Wichtig ist hierbei herauszustellen, dass yForms eine eigene Oberfläche mitbringt und nicht im SmartEdit integriert ist. Das widerspricht dem Kern der Arbeit, nämlich mithilfe einer einfachen Benutzerführung das gesamte Formular im SmartEdit anlegen, bearbeiten und zentral verwalten zu können. Die Verwendung von yForms über das Backoffice und SmartEdit bietet allerdings eine nutzerfreundliche Alternative zum vorherigen Vorgehen: dem Wechsel zwischen Bearbeitungs- und Darstellungsansicht. Bereits im Backoffice beim Erstellen des Formulars ist es visuell dargestellt und erleichtert die Arbeit enorm.

Mit yForms steht somit bereits eine Lösung zur Verfügung, welche je nach projektspezifischen Anforderungen einer dotSource-Implementierung durchaus vorgezogen werden kann. Zum Beispiel dann, wenn Felder abhängig zueinander sein sollen. Durch den Umfang und die vielen Funktionen, die yForms bietet, entstehen jedoch gleichermaßen Vor- und Nachteile. Die Erstellung aufwändiger Formulare mit vielen speziellen Funktionen und Abhängigkeiten der Inputfelder lässt sich bewerkstelligen. Die Funktionen sind allerdings an vielen Stellen unübersichtlich und kaum erklärt. yForms erfordert mehr Einarbeitungszeit als ein simples dotSource-Addon (dS-Addon).

3.2 SAP Commerce Addon

SAP Commerce erlaubt mit Hilfe von Extensions die einfache Erweiterung des Systems. (siehe Kapitel 4.1.1) Da diese einen speziellen Teilbereich an Funktionen abdecken, wäre es vorstellbar, ein Addon[14] zu entwickeln, welches lediglich die Einbindung dynamischer Formulare ermöglicht. Sie könnte dann, ähnlich wie yForms, Projekten optional hinzugefügt werden. Die Integration mit SmartEdit wäre einfach, da es lediglich um einige CMS-Komponenten erweitert werden müsste. Die Administration von Seiteninhalten würde sich auf das SmartEdit beschränken.

3.3 Vergleich beider Ansätze

Eine Übersicht über die Vor- und Nachteile beider Optionen liefert Tabelle 1. Nach dieser Gegenüberstellung schneidet yForms in vielen Punkten besser ab als ein dS-Addon, allein aufgrund der vielen umfangreichen Funktionen. Beide Varianten lassen sich einfach über das Hinzufügen einer Abhängigkeit installieren. Während yForms bereits die Möglichkeit bietet, die Daten eines gesendeten Formulars zu persistieren, müsste dies bei dem dS-Addon zuerst implementiert werden. Generell müsste sich der Entwickler dabei um das Verhalten der Formulardaten selbst kümmern.

Der Funktionsumfang ist bei einer Eigenimplementierung des dynamischen Formulars deutlich geringer. Sie müsste sich auf die wesentlichen Features beschränken. Bei yForms finden sich viele komplexe Funktionen, wodurch die Verwendung, gerade bei umfangreicheren Formularen, erschwert wird.

[14] Eine speziellere Form der Extension, siehe hierzu Kapitel 4.1.2

	yForms	dS-Addon
Vorteile	• viele Funktionen und detaillierte Regelungen • Persistieren möglich • einfache Formulare schnell zu erstellen • einfache Installation	• simple Verwendung für Entwickler und Endnutzer • einfache Integration in SmartEdit • einfache Installation
Nachteile	• Funktionsumfang überfordert • detaillierte Einstellungen schwer verständlich • keine vollständige Integration in SmartEdit • viel Einarbeitung nötig • schlecht dokumentiert	• Verhalten muss selbst implementiert werden • deutlich weniger Funktionen • Persistieren nicht möglich

Tabelle 1: Vor- und Nachteile von yForms und dS-Addon

Ein wichtiges Kriterium ist die Integration in das Content-Management-System SmartEdit. Problematisch ist bei yForms, dass die Oberfläche zum Anlegen von Formularen über das Backoffice verwendet wird und das fertige Formular im SmartEdit nur noch an der richtigen Stelle eingebunden wird. Die Verwendung von yForms würde dem bisherigen Zustand ähneln, bei dem Komponenten im WCMS angelegt werden und erst beim Aufruf der Storefront zu sehen sind. Ein dS-Addon ließe sich sehr einfach mit SmartEdit verwenden. Da die vollständige Integration in SmartEdit, im Kontext der vorliegenden Arbeit, als K.O.-Kriterium zu betrachten ist, wird yForms nicht für die Umsetzung zum Einsatz kommen. Stattdessen wird ein Addon (dsformaddon) implementiert, welche die Funktionen aus bisherigen Projekten der dotSource GmbH zusammenfasst und ergänzt.

Es sei erwähnt, dass die Verwendung von yForms in zukünftigen Projekten nicht ausgeschlossen ist. Die fehlende vollständige SmartEdit-Integration ist der bisherigen Vorgehensweise über das WCMS vorzuziehen. Eine Möglichkeit wäre es sogar, das dS-Addon und yForms parallel zu installieren und zu verwenden. Für spezielle Funktionen, wie zum Beispiel die Abhängigkeit von Eingabefeldern zueinander, kann dann auf yForms zurückgegriffen werden, während einfache Formulare mit dem dS-Addon realisiert werden können. Somit hängt der Einsatz beider Lösungen stark von den Anforderungen und den Wünschen des jeweiligen Kunden ab.

4 Verwendete Technologien

4.1 SAP Commerce

Das eingesetzte Shopsystem SAP Commerce[15] basiert auf Java.
Ein intern häufig eingesetztes Framework für solche Java-Projekte ist das Shopsystem SAP Commerce. Es ermöglicht die deutlich schnellere Umsetzung von E-Commerce-Projekten. Der Grund dafür ist die Bereitstellung eines rudimentären Gerüsts und vielen regelmäßig benötigten Funktionen von Online-Shops. So übernimmt das Framework zum Beispiel das Content Management oder den Bestellprozess zu großen Teilen.[16]

4.1.1 Extension

Anpassungen an einem SAP Commerce System erfolgen modular in der Form von sogenannten Extensions.[17] Wie in Kapitel 2.1 beschrieben, enthält eine Extension Funktionalitäten, die ein bestimmtes Aufgabengebiet abdecken[18]. Sie kapselt unter anderem Features, Datentypen oder Backoffice Konfigurationen. Beispielsweise beinhaltet eine Extension mit dem Namen *projektnamestorefront* Logik für die Benutzeroberfläche des Shops – auch Storefront genannt.

SAP bietet viele Extensions an, welche einige typische Funktionen abdecken. Für kundenspezifische Anpassungen ist es jedoch problemlos möglich, das System um eigene Extensions zu erweitern. Dabei kann deren Verwendung frei und zentral konfiguriert werden. Extensions können auch zueinander abhängig sein und füreinander Logik bereitstellen. Aufgrund der Schichtenarchitektur von SAP Commerce, kommunizieren beispielsweise Controller mit der darunterliegenden Schicht – den Facades. Die

---

[15] https://www.sap.com/products/crm/e-commerce-platforms.html
[16] [SAP19a]
[17] [SAP19b]
[18] [SAP19c]

Extension eines Controllers ist demnach von einer Extension abhängig, die verschiedene Facades bereitstellt.
Für die Erstellung eigener Extensions liefert das SAP Commerce Framework Vorlagen bzw. Extension Templates. Zusammengefasst wird eine Kopie eines solchen Templates erzeugt, welche dann als Ausgangspunkt für weitere Implementierungen dient.[19] Dabei werden Verzeichnisstruktur und wichtige Dateien angelegt. Kapitel 5.1 wird dieses Vorgehen praxisnah beleuchten.

4.1.2 **Addon**

Addons sind eine weitere Möglichkeit, ein SAP Commerce System zu erweitern. Sie besitzen einen geringeren Umfang als Extensions und kümmern sich um einen kleinen speziellen Teil. Im Kontext dieser Arbeit wird durch ein Addon beispielsweise die Möglichkeit geschaffen, ein dynamisches Formular zu erstellen. Addons sind, genauer betrachtet, auch Extensions, die es ermöglichen, Front-End relevante Dateien wie HTML, CSS und JavaScript zu ergänzen, ohne Anpassungen an bestehenden Dateien vornehmen zu müssen.[20] Sie können jedoch auch Geschäftslogik enthalten. Addons erweitern lediglich bestehenden Code. Das Ziel des Konzepts ist die Erweiterung der Storefrontfunktionalität, ohne den Code der Extension anpassen zu müssen.[21]

Ein Addon kann in einer bestehenden Extension installiert und dieser dabei als Abhängigkeit hinzugefügt werden. Die Inhalte des Addon werden dann in die Extension kopiert und können dort problemlos verwendet werden. Ein großer Vorteil ist, dass der bestehende Code der Extension nicht verändert werden muss, um diese Funktionalität bereitzustellen. Dadurch kann das Addon auch jederzeit ohne Auswirkungen entfernt werden. Dies kann beispielsweise hilfreich sein, wenn man bestehenden Code in Form von Extensions nicht anpassen kann oder darf.

---

[19] [SAP19d]
[20] [SAP19e]
[21] ebenda

Bei dem Kopierprozess der Dateien werden die Inhalte des Addon, die sich im Verzeichnis *acceleratoraddon* befinden, in das Verzeichnis *web* der Storefront kopiert, in die es installiert wurde. In der Storefront-Extension werden in speziellen Verzeichnissen die Dateien des Addon abgelegt. Tabelle 2 stellt die Beziehungen zwischen den Ausgangsordnern und den Zielordnern beim Kopieren dar.

Ausgangsordner (im Addon)	**Beschreibung**	**Zielordner (in Storefront)**
<addon_name> /acceleratoraddon/web /webroot/_ui	Beinhaltet statische Ressourcen wie beispielsweise CSS, JavaScript, Bilder	*<storefrontextension>* /web/webroot/_ui/addons /*<addon_name>*
<addon_name> /acceleratoraddon/web/ webroot/WEB-INF/ *<folder/subfolder/resource>*	Beinhaltet TAG-, JSP-, TXT-Dateien …	*<storefrontextension>* /web/webroot/WEB-INF/ *<folder/addons /addon_name/subfolder /resource>*
<addon_name> /acceleratoraddon/web/src	Beinhaltet den Java-Quellcode des Addon	*<storefrontextension>* /web/addonsrc /*<addon_name>*

Tabelle 2: Ausgangs- und Zielordner des Kopierprozesses (vom Autor angepasst)[22]

4.1.3 **ImpEx**

SAP Commerce verwendet für das Importieren verschiedener Daten ImpEx-Dateien. Diese sind wie CSV-Dateien (Comma-seperated values) aufgebaut und ermöglichen die Manipulation von Daten zur Laufzeit oder bei der Initialisierung des

[22] [SAP19f]

SAP Commerce Systems.[23] Nach SAP Commerce Konventionen können in einem Ordner */resources/impex* einer beliebigen Extension ImpEx-Dateien hinterlegt werden, die automatisch eingespielt werden[24]. Mit dem Prefix *essentialdata* werden diese bei jeder Initialisierung des SAP Commerce Systems geladen, während das mit dem Prefix *projectdata* nur auf explizite Anweisung hin geschieht.[25] Das bedeutet, wichtige Daten, welche die Extension für ein fehlerfreies Laufen benötigt, werden mit *essentialdata* gekennzeichnet. Testdaten hingegen können zum Beispiel als *projectdata* markiert werden.

4.2 Spring

SAP Commerce baut zu großen Teilen auf dem Spring Framework auf.[26] Dadurch wird unter anderem die Dependency Injection ermöglicht. So werden die Abhängigkeiten der Instanzen von Klassen zur Laufzeit der Anwendung verwaltet. Das erfolgt in XML-Dateien. Dadurch müssen sich Komponenten in SAP Commerce nicht selbst im Code um die Erzeugung der Objekte kümmern. Das Projekt lässt sich einfacher überblicken und wird dadurch deutlich besser skalierbar, da keine Vielzahl von Konstruktoraufrufen bei Änderungen angepasst werden muss. Des Weiteren verwendet die SAP Commerce Extension *yacceleratorstorefront*, die für die Erzeugung der graphischen Oberfläche eines Shops benutzt wird, Spring MVC. Die Steuerung der Storefront über Controller geschieht somit ebenfalls auf Basis von Spring. Im Rahmen der vorliegenden Bachelorarbeit werden außerdem einige Klassen von Spring Security zum Einsatz kommen.

---

[23] [SAP19g]
[24] [SAP19h]
[25] ebenda
[26] [SAP19i]

4.3 SmartEdit und WCMS

SmartEdit und das WCMS Cockpit sind Content Management Systeme, welche in SAP Commerce integriert sind. Sie ermöglichen es Content Managern die Inhalte des Online-Shops zu verwalten. Dazu zählt beispielsweise das Anlegen von Seiten, das Hinzufügen von Komponenten oder die Bearbeitung von Texten. Das CMS Cockpit gilt als deprecated seit der SAP Commerce Version 6.6[27]. In neueren Versionen wird es unter anderem durch SmartEdit abgelöst. Dieses zeichnet sich durch eine deutlich bessere Benutzerführung aus.

Bei der Verwendung des WCMS müssen Änderungen überprüft werden, indem zwischen Bearbeitungs- und Darstellungsansicht gewechselt wird. SmartEdit löst diesen Umstand durch die Möglichkeit, Anpassungen direkt in der Storefront vorzunehmen. Komponenten können so zum Beispiel per Drag and Drop hinzugefügt werden. Dadurch erhält der Content Manager unmittelbar visuelles Feedback nach dem Prinzip „What you see is what you get“ (WYSIWYG).

[27] [SAP19k]

5 Implementierung

Da sich in den folgenden Kapiteln einige Namensgebungen sehr ähneln, soll eine kurze Übersicht für Klarheit sorgen. In Kapitel 5.4 wird die Darstellung der Formularkomponenten beschrieben. Dabei wird der *DsFormComponentController* erwähnt. Dieser sorgt für die Anzeige der Formularkomponente. Im Gegensatz dazu, kommt für die Ereignisbehandlung des Formulars der *DsFormController* zum Einsatz. Dieser wird in den Kapiteln 5.5 und 5.6.1 genauer erklärt.
Das Persistieren der Komponenten ist kein zentrales Thema der vorliegenden Bachelorarbeit. Sollte nachfolgend die Rede vom Persistieren der Daten oder des Formulars sein, so sind die Inhalte gemeint, die von einem Nutzer des Formulars eingetragen werden.

5.1 Test- und Entwicklungsumgebung

Für die Entwicklung des Addon und um dieses testen zu können, wurde eine virtuelle Maschine zum Einsatz kommen. Diese läuft mit dem Betriebssystem Ubuntu in der Version 16.04.2 LTS. Eine Installation unter Windows Systemen ist möglich.

Nach der Installation von SAP Commerce, ist es über das enthaltenen Tool *modulegen* möglich, schnell und einfach abhängige Extensions generieren zu lassen, welche ein grundlegend funktionales Projekt bilden. In der vorliegenden Arbeit findet es ausschließlich Verwendung, um eine Möglichkeit zum praxisnahen Testen des Addon bereitzustellen. Schließlich ist die Intention, das Addon in produktiven Umgebungen zum Einsatz zu bringen. Der entsprechende Aufruf für *modulegen* sieht für die vorliegende Arbeit wie folgt aus:

```
ant modulegen -Dinput.module=accelerator -Dinput.name=dsform
        -Dinput.package=de.dotsource.dsform -Dinput.template=develop
```

Die automatisch erzeugten Extensions müssen danach in der Datei *localextensions.xml* ergänzt und alle Extensions mit dem Prefix *yaccelerator* entfernt werden. Da somit auch die Extension *yacceleratorstorefront* entfernt wird, werden deren Addons in die neu erstellte *dsformstorefront* installiert:

```
ant reinstall_addons -Dtarget.storefront=dsformstorefront
```

SAP Commerce enthält bereits Daten, um eine Demostorefront darstellen zu können. So kann auf simpler Ebene ein echter Online-Shop simuliert werden und zum Testen dienen. Damit die Oberfläche im Browser aufgerufen werden kann, müssen der Datei *hybris/config/local.properties* die beiden Einträge aus Abbildung 1 hinzugefügt werden.

```
website.electronics.http=http://electronics.local:9001/dsformstorefront
website.electronics.https=https://electronics.local:9002/dsformstorefront
```

Abbildung 1: Einträge in *local.properties* für dsformstorefront

Um die Adressen aufrufen zu können, muss ein entsprechender Host-Eintrag gesetzt werden. Auf Linux-Systemen erfolgt das über den Eintrag aus Abbildung 2 in der Datei */etc/hosts.*

```
127.0.0.1   localhost electronics.local
```

Abbildung 2: Eintrag in der Datei */etc/hosts*

Als nächstes muss der Buildprozess gestartet werden, bei dem Konfigurationen angelegt sowie alle benötigten Java-Klassen in den Extensions kompiliert werden. Hierfür wird im selben Verzeichnis die nachfolgende Kommandozeile abgesetzt:

```
ant clean all
```

Zuletzt muss eine Initialisierung des Systems erfolgen. Gestartet werden kann diese über den Befehl

```
ant initialize
```

Währenddessen werden wichtige Daten importiert und die Datenbankstruktur angelegt. Nach Beendigung kann der SAP Commerce Server wie folgt gestartet werden:

```
./hybrisserver.sh
```

Nachdem der Server vollständig hochgefahren ist, kann die Storefront im Browser über die konfigurierte Adresse aufgerufen werden. Das Addon wird über das Tool *extgen* erstellt. Zuvor muss allerdings in die Datei *hybris/config/localextensions.xml* der Eintrag aus Abbildung 3 übernommen werden.

```
<extension name="yaddon" />
```

Abbildung 3: Ergänzung der Extension *yaddon* in *localextensions.xml*

Das ermöglicht die Verwendung des SAP Commerce Templates, welches als Vorlage für Addons vorgesehen ist. Im Rahmen der Arbeit wird das Addon erstellt mit dem Befehl

```
ant extgen -Dinput.template=yaddon -Dinput.name=dsformaddon
        -Dinput.package=de.dotsource.dsformaddon
```

Um dem SAP Commerce System das generierte Addon bekannt zu machen, erhält die Datei *localextensions.xml* analog zu *yaddon* zwei weitere Einträge für die Extensions *dsformaddon* und *addonsupport*. Das ermöglicht die Installation des Addon in die *dsformstorefront* und erweitert diese um die entsprechenden Funktionalitäten.

Die Installation wird wie folgt ausgeführt:

```
ant addoninstall -Daddonnames="dsformaddon"
        -DaddonStorefront.yacceleratorstorefront="dsformstorefront"
```

Die Datei *localextensions.xml* muss um die folgenden Extensions erweitert werden. (siehe Abbildung 4)

```
<extension name="smartedit" />
<extension name="smarteditaddon" />
<extension name="cmsbackoffice" />
<extension name="cmssmartedit" />
<extension name="cmssmarteditwebservices" />
<extension name="cmswebservices" />
<extension name="permissionswebservices" />
<extension name="previewwebservices" />
<extension name="smarteditwebservices" />
```

Abbildung 4: Einträge in *localextension.xml* für SmartEdit

Anschließend muss, während der Server hochgefahren ist, in der Hybris Administration Console (HAC) unter *Platform – Update* ein Update ausgeführt werden, bei dem die obigen Extensions ausgewählt sind, um Typdefinitionen einzulesen. Nach Beendigung ist SmartEdit unter der Adresse *https://localhost:9002/smartedit/* zu erreichen.

Für das Auflösen von Lokalisierungscodes wird die bestehende firmeninterne Extension *dslocalization* von dotSource zum Einsatz kommen. Das dsformaddon wird diese in ihrer Datei *extensioninfo.xml* als Abhängigkeit definieren und erwartet, dass die Extension vorhanden ist. Dazu wird die *dslocalization*-Extension in dem Unterverzeichnis *hybris/bin/custom/dotsource* abgelegt.

Nach Abschluss dieser Schritte befinden sich unter *hybris/bin/custom* zwei Verzeichnisse, *dotsource* und *dsform,* mit den weiteren Unterverzeichnissen:

- dsformaddon
- dsformbackoffice
- dsformcore
- dsformfacades
- dsformfulfilmentprocess
- dsforminitialdata
- dsformstorefront
- dsformtest

Der Server lässt sich starten und über den jeweiligen Uniform Resource Identifier (URI) lassen sich die Storefront und SmartEdit aufrufen. Das Resultat dieses Setups ist ein Projekt, welches im folgenden Verlauf der Arbeit als Test- und Entwicklungsumgebung dient. Entwicklungen finden lediglich in dem Addon *dsformaddon* statt und sind während der Entwicklung direkt an der Testumgebung nachvollziehbar.

5.2 Zusammenführung von Teilimplementierungen aus Projekten

Das Addon soll die Implementierungen von dynamischen Formularen aus mehreren Projekten der dotSource GmbH zusammenfassen. Es handelt sich um drei verschiedene Umsetzungen, welche sich jedoch an vielen Stellen ähneln. Da sie sich in einigen Details unterscheiden, muss vorher definiert werden, welche Features jeweils in das Addon übernommen werden, um die größtmögliche Funktionalität bereitzustellen. Unterschiede finden sich in den unterstützten Eingabefeldern der Formulare. So stellt eine Implementierung ein Feld für das Hochladen von Bildern bereit. Ein anderes Projekt erlaubt nicht nur Bilder, sondern auch andere Dateien.

Eine große Gemeinsamkeit der drei Umsetzungen ist, dass beim Abschicken der Formulare eine E-Mail an eine vorkonfigurierte E-Mail-Adresse mit festem Betreff

gesendet wird, welche die Werte der Eingabefelder beinhaltet. Dieser Prozess wird über einen Button gestartet, welcher automatisch jedem Formular hinzugefügt wird. Es ist somit momentan nicht möglich die eingetragenen Daten zu persistieren oder in anderer Form weiterzuverarbeiten. Die Funktion zum Versenden der E-Mails soll entfernt werden. Tabelle 3 soll die wesentlichen Unterschiede der einzelnen Projekte zueinander aufzeigen.

	Projekt 1	**Projekt 2**	**Projekt 3**
SAP Commerce Version	Version 6.6	Version 6.6	Version 1811[28]
CMS	WCMS	WCMS	SmartEdit
Komponenten	• Checkbox[29] • Drop-Down[30] • Radio Button[31] • Textfeld[32] • Wrapper • Bild-Upload	• Checkbox • Drop-Down • Radio Button • Textfeld • Wrapper	• Checkbox • Drop-Down • Radio Button • Textfeld • Wrapper • File-Upload • Mehrfachauswahl • Datumsauswahl

Tabelle 3: Darstellung von Unterschieden der Projekte

Zwei der Projekte basieren auf der SAP Commerce Version 6.6 und verwenden als Content Management System das WCMS. Im dritten Projekt kommt SmartEdit zum Einsatz und integriert bereits das dynamische Formular. Projekt 3 dient

[28] Chronologie der SAP Commerce Versionen (Ausschnitt): 6.6, 6.7, 1808, 1811
[29] siehe https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox
[30] siehe https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select
[31] siehe https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/radio
[32] siehe https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/text

aufgrund seine Funktionsumfangs als Basis des Addon. Die Inhalte der beiden anderen werden ergänzt und dann zentral in Form des SAP Commerce Addon bereitgestellt. Da die Unterstützung für WCMS zukünftig abgeschafft wird und die Integration in SmartEdit somit umso wichtiger ist, wird in dieser Arbeit kein Fokus darauf liegen, das Addon für das WCMS kompatibel zu gestalten.

Allen drei Projekten gemein sind jeweils die ersten fünf Komponenten. Die Wrapper-Komponente unterscheidet sich grundlegend von den anderen Komponenten. Sie dient als Container für andere Komponenten. Normalerweise werden die Eingabefelder eines Formulars untereinander dargestellt. Mit der Wrapper-Komponente kann jedoch Einfluss auf das Layout genommen werden. So ist es beispielsweise auch möglich, die Felder nebeneinander anzuordnen.

Zusätzlich besitzt Projekt 1 eine Bild-Upload-Komponente, welche das Formular um die Möglichkeit ergänzt, Bilder zu versenden. Das Dateiformat ist dabei beschränkt auf PNG und JPEG. Ebenso sind Dateigrößen von mehr als 5 MiB unzulässig. Projekt 3 erweitert diese Funktion bereits um die Formate TXT und PDF und wird daher als File-Upload bezeichnet. Die Mehrfachauswahl-Komponente aus Projekt 3 ist dem Drop-Down sehr ähnlich. Aus einer Liste von Elementen können beliebig viele Einträge ausgewählt werden. Zuletzt erlaubt die Datumsauswahl die Wahl eines speziellen Tages über das Öffnen eines Kalenders.

Der Bild-Upload aus Projekt 1 ist aufgrund der umfangreicheren File-Upload-Komponente redundant und wird durch diese ersetzt. Die neue Implementierung soll dem Entwickler mehr Freiheiten bei der Verarbeitung der Formularinhalte lassen. Daher soll der bisher fixe Prozess, nämlich das Versenden der Werte als E-Mail, entkoppelt werden. Es müssen dazu einige, nicht mehr benötigte, Attribute und Typen entfernt werden, um keine redundanten Elemente zu übernehmen und die Übersichtlichkeit zu wahren.

Bei einer vorhandenen Installation von SmartEdit soll das Formular mit allen genannten Komponenten aus Projekt 3 in der Storefront hinzugefügt werden können. Die Darstellung der Komponenten im Formular soll in erster Linie unverändert bleiben und nur wenn nötig angepasst werden. Für jedes Eingabefeld existieren bereits Möglichkeiten zur Validierung der Nutzeingabe. Diese sollen ebenfalls in das Addon übernommen werden. Die entsprechenden Fehlermeldungen bei falscher Bedienung des Formulars werden beibehalten. Die File-Upload-Komponente hängt von einigen, im Projekt implementierten, Java-Klassen ab. Die benötigten Methoden müssen daher im Addon vorhanden sein.

5.3 Anlegen von Formular- und Input-Komponenten

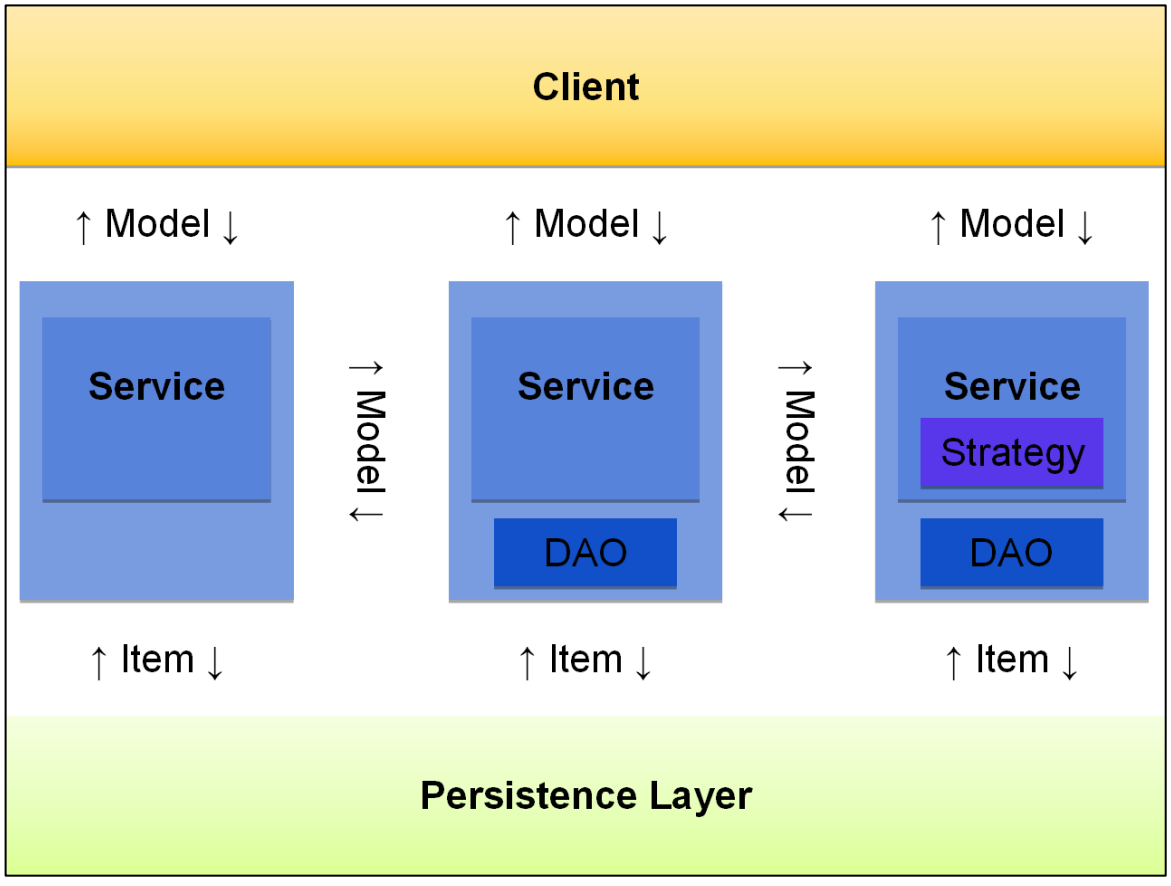


Abbildung 5: Schichtenarchtiektur von SAP Commerce

Um eine eigene Komponente für SAP Commerce zu definieren muss, der Typ in der Datei *dsformaddon-items.xml* des *dsformaddon* ergänzt werden. Dort werden mithilfe von XML die Attribute festgelegt. Beim Erstellprozess, also dem Aufruf von *ant build,* werden aus der Datei Models generiert. Das sind Java-Klassen, welche lediglich die Attribute und deren Getter und Setter implementieren. Modelklassen erlauben in der Geschäftslogik die Verwendung von Objekten um Änderungen an der Datenbank vorzunehmen. In Kombination mit dem Service-Layer lassen sich angelegte Komponenten speichern und laden. Der direkte Umgang mit der Datenbank ist abstrahiert und betrifft den Entwickler nicht. Die Schichtenarchitektur von SAP Commerce ist in Abbildung 5[33] zu sehen.

In Abbildung 6 ist als Beispiel die Definition des Typen *DsFormComponent* dargestellt. Dieser repräsentiert die Basiskomponente des Formulars. Als Elternklasse ist *CMSParagraphComponent* definiert – ein SAP Commerce Standard Typ. Die Eigenschaft *generate* mit dem Wert true sorgt für die Generierung der Modelklasse im Java Code. Durch das Setzen von *autocreate* auf den Wert true, erzeugt SAP Commerce bei einem Update (siehe Kapitel 5.1) einen neuen Subtyp und erweitert die Spalten der Datenbank. *DsFormComponent* besitzt zwei Attribute. Eine Liste mit dem Namen *componentList* deren Elemente vom Typ *SimpleCMSComponent* (SAP Commerce Standard) sind und die Eingabefelder des Formulars beinhaltet. Sowie einen String mit dem Bezeichner *submitButton*, welcher den Text des Buttons zum Absenden des Formulars angibt. Über die Angabe des *persistence*-Tag, wird die Datenbank angewiesen, die Standarddatentypen für das Attribut zu verwenden.

[33] [SAP19l]

```
<itemtype code="DsFormComponent" extends="CMSParagraphComponent"
    autocreate="true" generate="true"
    jaloclass="de.dotsource.dsformaddon.jalo.DsFormComponent">
        <attributes>
            <attribute qualifier="componentList"
             type="SimpleCMSComponentList">
                <persistence type="property"/>
            </attribute>
            <attribute qualifier="submitButton" type="java.lang.String">
                <persistence type="property"/>
            </attribute>
         </attributes>
</itemtype>
```

Abbildung 6: Definition von DsFormComponent in dsformaddon-items.xml

Die Komponenten, welche die Eingabefelder repräsentieren, haben die folgenden gemeinsamen Attribute:

- *mandatory* – Typ Boolean, gibt an, ob das Feld im Formular ausgefüllt sein muss
- *label* – Typ String, gibt an, wie das Feld im Formular beschriftet ist
- *formElementName* – Typ String, gibt den Namen der Komponente an, wird verwendet für die ID des HTML-Elements

Aus diesem Grund wird ein Basistyp mit der Superklasse *SimpleCMSComponent* und dem Namen *FormElementComponent* angelegt, der diese Attribute besitzt. Neue Attribute werden für die Typen *FormMultiSelectElementComponent* sowie *FormSelectElementComponent* – Mehrfachauswahl und Drop Down – ergänzt. Sie besitzen jeweils eine Liste vom Typ *FormOptionElementComponent,* um die verschiedenen Auswahloptionen zu repräsentieren. Außerdem fügt der Typ *FormInputElementComponent* – das Texteingabefeld – Attribute für das eingegebene Format, die Fehlermeldung bei nicht eingehaltenem Format sowie für einen Platzhaltertext hinzu.

Der Typ *WrapperComponent* erbt von *SimpleCMSComponent* (SAP Commerce Standard). Er definiert ebenfalls eine Liste, in welcher sich die hinzugefügten Eingabe-Komponenten befinden. Einige weitere Attribute spezifizieren die Darstellung der

Komponente und ihrer Unterkomponenten. Die Datei *dsformaddon-items.xml* beinhaltet eine Definition für jede Unterkomponente des Fomulars. Damit der Typ *SimpleCMSComponent* problemlos verwendet werden kann, wird die SAP Commerce Extension *cms2* benötigt. Diese muss daher in der Datei *extensioninfo.xml* von *dsformaddon* folgendermaßen ergänzt werden:

```
<requires-extension name="cms2" />
```

5.4 Darstellung der Komponenten

Die Darstellung der Komponenten geschieht in SAP Commerce mithilfe von JSP- und Tag-Dateien. „Eine JSP (JavaServer Pages) […] [ist] eine HTML-Seite mit eingebettetem Java-Code.“[34] Der Dateiname muss äquivalent zum jeweiligen Komponenten-Typ sein. Beispielsweise gehört zum Typ *DsFormComponent* die JSP-Datei *dsformcomponent.jsp*.

Für jede der vorhandenen Eingabefelder sowie für die Wrapper-Komponente ist eine entsprechende JSP-Datei vorhanden. Darin befindet sich die generelle Struktur der Komponenten, die statisch ist und verschiedene ID´s oder Klassen abhängig von den Attributen zuweist. Ergänzt werden die JSP-Dateien der Eingabekomponenten um den Aufruf einer Tag-Library. In diesen wird die Darstellung geregelt. So werden HTML-Elemente nur angezeigt, wenn bestimmte Attribute der Komponente gesetzt sind, welche den *form*-Tags hinzugefügt werden kann. Die *dsformcomponent.jsp* verwendet die Tag-Library *dsFormComponentForm.tag*. Diese bildet den zentralen Ausgangspunkt für die Anzeige des dynamischen Formulars und dessen Unterkomponenten. Zu sehen in Anlage 1. Tabelle 4 erklärt die wichtigsten Attribute.

[34] [Ull10]

Attribut	Erklärung
action	Verweis auf Methode im Controller mit der URL */xhr/dsForm/send*
method	spezifiziert hier, dass es sich um einen Post-Request handelt
data-role	Erleichert im JavaScript den Zugriff auf Formular
enctype	Ermöglicht mit dem Wert *„multipart/form-data“* den Dateiupload

Tabelle 4: Übersicht wichtiger Attribute in *dsFormComponentForm.tag*

Jedes Formular besitzt ein verstecktes Inputfeld, welches als Wert die UID der Formularkomponente besitzt. Kapitel 5.6.2 befasst sich mit der Verwendung der Attribute und diesem Inputfeld im Detail.

Für die Anzeige des Formulars wird in der Tag-Datei über die Liste von Unterkomponenten der Formularkomponente iteriert. Diese werden dann über ihre eigenen JSP-Dateien dargestellt. Eine ähnliche Funktionsweise besitzt die Wrapper-Komponente. Wenn das Formular eine Wrapper-Komponente anzeigen möchte, dann iteriert diese über ihre eigene Liste von Unterkomponenten und zeigt diese an. So werden nacheinander alle im Formular enthaltenen Komponenten in der Storefront dargestellt. Schlussendlich wird darunter ein Button ergänzt, der mit dem konfigurierten Wert aus dem Attribut *submitButton* von *DsFormComponent* beschriftet ist. Sollte dieser Wert leer sein, wird ein Standardwert eingefügt.

Bei der Verwendung der Tag-Libraries ist darauf zu achten, dass diese in die JSP-Dateien eingebunden werden müssen. Das erfolgt über die *taglib*-Direktive zu Beginn der Datei. Dort wird ein Präfix angegeben, unter dem die Tag-Library verwendet werden kann und ein URI, welcher auf den entsprechenden Speicherort verweist.

Um die JSP der *DsFormComponent* anzeigen zu können, muss ein zugehöriger Komponenten-Controller vorhanden sein. Dieser heißt nach Konvention *DsFormComponentController* (siehe Anlage 2) und erweitert die SAP Commerce Klasse *AbstractCMSAddonComponentController*. Er besitzt lediglich zwei private String-Konstanten und überschreibt die Methode *fillModel*. In dieser wird der JSP-Datei ein Objekt vom Typ *DsFormComponentModel* hinzugefügt. Dieses Objekt wird der Methode *emptyForm* aus der Hilfsklasse *DsFormUtil* übergeben und ihr Rückgabewert vom Typ *DsFormComponentForm* wird ebenfalls der JSP-Datei hinzugefügt. Die Methode wird in Kapitel 5.6.3 genauer erklärt. Die Konstanten gelten jeweils als Variablenname der Objekte in der JSP. Bei jedem Aufruf einer Seite, die eine *DsFormComponent* enthält, wird der Controller angefragt und kümmert sich um die Darstellung der Komponente. Abbildung 7 zeigt ein UML-Diagramm entsprechendes UML-Diagramm. Aus Gründen der Übersicht wird in der Klasse *AbstractCMSAddonComponentController* lediglich die Methode *fillModel* dargestellt.

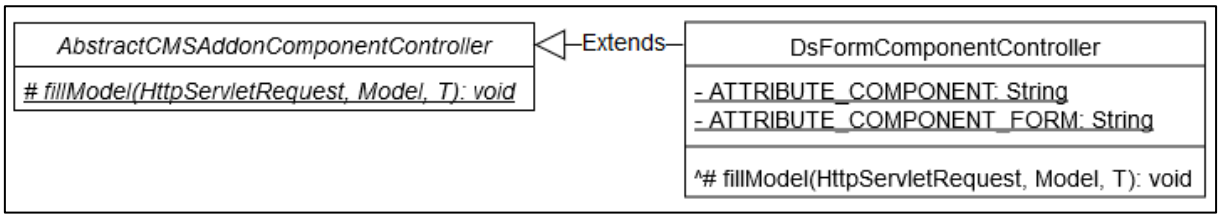


Abbildung 7: UML-Diagramm für *DsFormComponentController*

Für die restlichen Komponenten des Formulars kann ein sogenannter Renderer verwendet werden.[35] In der Datei *dsformaddon-web-spring.xml* wird eine Bean *formElementComponentsRenderer* mit dem Wert *addOnJspInlcueCMSComponentRenderer* für das parent-Attribut definiert. Die Bean wird der Wrapper- sowie jeder Formularkomponente im Attribut *renderer* hinzugefügt. In Abbildung 8 ist diese Zuweisung als Beispiel für die *FormCheckboxElementComponent* dargestellt.

[35] [SAP19m]

```
<bean id="formElementComponentsRenderer"
    parent="addOnJspIncludeCMSComponentRenderer"/>
<bean id="formCheckboxElementComponentRendererMapping"
    parent="addonCmsComponentRendererMapping">
        <property name="typeCode" value="FormCheckboxElementComponent" />
        <property name="renderer" ref="formElementComponentsRenderer" />
</bean>
```

Abbildung 8: Zuweisung eines individuellen Renderer

Damit die Formularkomponente im SmartEdit verfügbar wird und den Inhaltsseiten hinzugefügt werden kann, muss die ImpEx aus Abbildung 9 in das SAP Commerce System eingespielt werden. Damit das bei einer Integration des *dsformaddon* nicht manuell erfolgen muss, erhält die ImpEx-Datei das Prefix *essentialdata* (siehe Kapitel 4.1.3). Sie wird somit bei jeder Initialisierung des Systems geladen. *ComponentTypeGroups2ComponentType* ist eine Relation, die Komponenten zu Komponentengruppen zusammenfasst und so festlegt, welchen Bereichen einer Seite die Komponente hinzugefügt werden kann. Der gesamte Dateiname ist *resources/essentialdata-form-component.impex*.

```
INSERT_UPDATE ComponentTypeGroups2ComponentTyp
      ; source(code)[unique = true]; target(code)[unique = true]
      ; wide                       ; DsFormComponent
      ; wide                       ; WrapperComponent
```

Abbildung 9: Inhalt der ImpEx-Datei essentialdata-form-component.impex

Um die Oberfläche im SmartEdit verständlich zu präsentieren, werden in Lokalisierungsdateien die Bezeichnungen der Komponentenattribute festgelegt. Die definierten Schlüssel werden durch den zugeordneten Text ersetzt. Ein beispielhafter Ausschnitt aus den deutschen Lokalisierungen ist in Abbildung 10 dargestellt.

```
type.DsFormComponent.name = Form Komponente
type.DsFormComponent.componentList.name = Liste der Formularfelder
type.FormRadioElementComponent.name = Radio Buttons
```

Abbildung 10: Ausschnitt aus *dsformaddon-locales_de.properties*

5.5 Konzept zur individuellen Ereignisbehandlung

Den Entwicklern, die das Addon zukünftig verwenden, soll es ermöglicht werden, selbst zu entscheiden, welche Aktion nach Versenden des Formulars erfolgen soll. Da die Anwendungsfälle weitreichend sind, soll keine statische Verarbeitung der Nutzereingaben aufgezwungen werden. Dieser Ansatz erfordert eine Erweiterungsmöglichkeit, die individuelle Implementierungen zulässt. Eine Lösung ist die Bereitstellung eines abstrahierten Controllers. Abbildung 11 veranschaulicht die Idee dahinter.

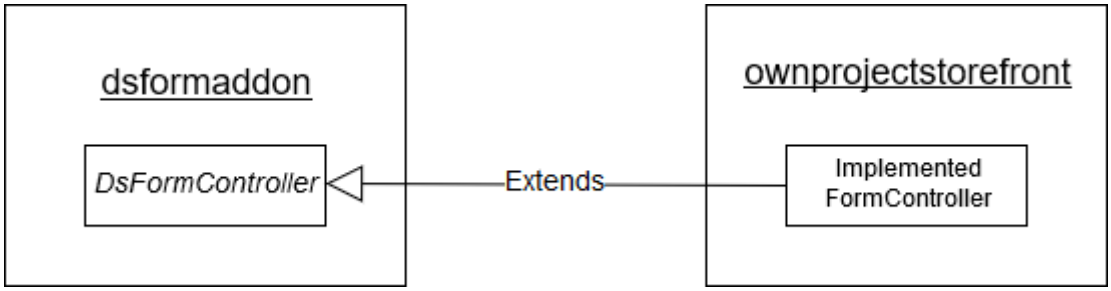


Abbildung 11: DsFormController als abstrakte Basisklasse für Storefront-Controller

Das *dsformaddon* stellt eine Klasse *DsFormController* bereit. Diese wird als abstrakt deklariert und kann somit nicht direkt instanziiert werden. Die Klasse deklariert eine ebenfalls abstrakte Methode *sendForm*, welche auf die Anfrage-URL */xhr/dsform/send* gemappt wird. Wird das Addon einem Projekt hinzugefügt, in dem dynamische Formulare zum Einsatz kommen sollen, so wird ein neuer Controller in der Storefront-Extension dieses Projekts angelegt. Dieser erbt von der abstrakten Klasse *DsFormController*, übernimmt somit die Methode *sendForm* und wird gezwungen, diese zu implementieren. Dort können über die Parameter der Methode, welche im Kapitel 5.6.2 spezifiziert werden, die Inhalte des Formulars ausgelesen und weiterverarbeitet werden.
Nach diesem Prinzip kümmert sich das Addon komplett darum, dass die Anfrage den Controller erreicht. Ein Entwickler muss dann lediglich implementieren, was mit den Daten geschehen soll. Die Verarbeitung der Daten ist entsprechend der Projektanforderungen umzusetzen. Möglich wäre es zwar, Lösungen für häufige Anwendungsfälle im Addon zu integrieren, allerdings ist das nicht mit den zeitlichen Einschränkungen dieser Bachelorarbeit vereinbar.

5.6 Geschäftslogik

5.6.1 Abstrakter Controller

Der abstrakte *DsFormController* erbt von der SAP Commerce Klasse *AbstractController*. Aus diesem Grund muss für das *dsformaddon* eine Abhängigkeit zu der Extension *acceleratorstorefrontcommons* hinzugefügt werden. In der Datei *extensionsinfo.xml* wird daher folgender Eintrag ergänzt:

```
<extension name="acceleratorstorefrontcommons"/>
```

Die abstrakte Methode *sendForm* von *DsFormController* besitzt vier Parameter. Die Signatur ist zu sehen in Abbildung 12. Der Parameter *model* ist notwendig, um Daten in die JSP-Dateien zu überreichen. Die *componentUid* wird in der POST-Anfrage mitgesendet und enthält die UID der *FormComponent,* von der die Anfrage stammt. *DsFormComponentForm* ist eine Klasse, welche lediglich eine Map als Attribut hat, die den Eingabefeldern ihre Werte zuweist und einen Getter sowie Setter für diese Map besitzt. Spring übernimmt das Befüllen dieses Parameters, wenn *modelAttribute* im form-Element der Tag-Datei *dsFormComponentForm.tag* entsprechend dem Parameternamen gesetzt ist. Das *bindingResult* ist verantwortlich für eventuelle Fehler im Formular, wie beispielsweise falsche Nutzereingaben.

```
@RequestMapping(value = "/send", method = RequestMethod.POST,
produces = MediaType.APPLICATION_JSON_VALUE)
public abstract String sendForm(final Model model,
                @RequestParam("componentUid") final String componentUid,
                DsFormComponentForm dsFormComponentForm,
                final BindingResult bindingResult);
```

Abbildung 12: Signatur der Methode *sendForm*

Der Aufruf dieser Controller-Methode erfolgt über einen AJAX-Request (Asynchronous JavaScript and XML). Dazu sind die Attribute des form-Elements in der Datei *dsFormComponentForm.tag*, welche bereits in Kapitel 5.3 beschrieben sind. Einen gekürzten Überblick über dieses Element findet sich in Abbildung 13.

```
<form:form id="formComponent-${component.uid}"
           action="/xhr/dsForm/send"
           method="POST"
           data-role="multipartAjaxForm"
           enctype="multipart/form-data"
           modelAttribute="dsFormComponentForm"
           novalidate="novalidate">
```

Abbildung 13: form-Element in der Datei *dsFormComponentForm.tag*

5.6.2 **AJAX-Request**

Ein AJAX-Request erlaubt die Kommunikation und den Datenaustausch – jeweils asynchron – mit dem Server, ohne dass die Seite im Browser neu geladen wird.[36] Die Formularkomponente unterstützt den Upload von Dateien. Für diese ist die Verwendung von *application/x-www-form-urlencoded* für den Header Content-Type (bzw. den enctype im form-Element) nicht effizient und es wir stattdessen *multipart/form-data* verwendet.[37] Da die Formulare dynamisch sind, kann nicht vorausgesagt werden, ob ein Formular eine Komponente für den Fileupload enthält. Es könnte zwar eine Erkennung dafür implementiert und anschließend zwischen den beiden Werten unterschieden werden, allerdings wird aus Simplizität für alle Formulare der Wert *multipart/form-data* verwendet, unabhängig davon, ob diese eine File-Upload-Komponente besitzen.

Die Logik für das Versenden des AJAX-Request befindet sich in einer JavaScript-Datei *dsformaddon.js*. In Abbildung 14 ist der relevante Ausschnitt zu sehen. Die Datei muss bei jedem Seitenaufruf geladen werden, da jede Seite, solange keine Einschränkungen bestehen, ein dynamisches Formular enthalten kann. Erläutert wird das entsprechende Vorgehen in Kapitel 5.7. Des Weiteren enthält die Datei eine Methode *bindDateChooser*, welche dafür sorgt, dass sich eine Oberfläche zur Datumsauswahl öffnet, sobald ein entsprechendes Feld den Fokus erhält.

---

[36] [Moz19a]
[37] [W3C19]

```
$.ajax({
        method: form.attr('method'),
        url: form.attr('action') + '?CSRFToken=' + ACC.config.CSRFToken,
        dataType: 'json',
        data: data,
        processData: false,
        contentType: false,
        headers: {
                enctype: 'multipart/form-data'
        },
})
```

Abbildung 14: AJAX-Request für dynamische Formulare

Das JavaScript fügt allen HTML-Elementen, welche im Attribut *data-role* den Wert *multipartAjaxForm* enthalten (demnach allen dynamischen Formularen), eine Funktion für das submit-Event zu. Diese unterdrückt zuerst das Standardverhalten des Events, was in diesem Fall dem Absenden des Formulars sowie einem Neuladen der Seite entspricht. In einem FormData-Objekt[38] werden die Felder des Formulars mit ihren Werten gespeichert.

Im Attribut *data* wird das FormData-Objekt angegeben, wodurch der entsprechende Inhalt des Formulars an den Server gesendet wird. Für *method* wird der entsprechende Wert des Attributs am form-Element in *dsFormComponentForm.tag* übernommen. Das Attribut *url* übernimmt die Anfrage-URL des Formulars und fügt dieser ein CSRF-Token als Anfrageparameter hinzu. CSRF steht für Cross-Site Request Forgery.[39] Es beschreibt eine Angriffsart, bei der in einem System eingeloggte Nutzer unwissentlich Anfrage-URLs ausführen.[40] SAP Commerce schützt vor CSRF-Attacken durch die Verwendung von CSRF-Tokens. Diese Tokens sind dem Angreifer unbekannt und verhindern bei Ungültigkeit die Ausführung einer Anfrage.

[38] [Moz19b]
[39] [OWA18]
[40] ebenda

5.6.3 **Hilfsklasse DsFormUtil**

Die Hilfsklasse *DsFormUtil* implementiert zwei Methoden, welche die restliche Logik für das Formular vereinfachen:

- public static DsFormComponentForm emptyForm(DsFormComponentModel component)
- public static List<FormElementComponentModel> getInputComponents(DsFormComponentModel form)

Wie in Kapitel 5.3 erwähnt, findet *emptyForm* Anwendung in der Klasse *DsFormComponentController*. Die Methode gibt ein neues Objekt vom Typ *DsFormComponentForm* zurück. Zusammengefasst wird über die Komponenten innerhalb des Formulars iteriert und eine Fallunterscheidung anhand des Komponententyps durchgeführt. Die ausgewählten Werte in vorhandenen Drop-Downs, Mehrfachauswahllisten sowie Gruppen von Radio-Buttons werden in das Objekt übernommen. Da dieses beim Seitenaufruf der JSP-Datei *dsformcomponent.jsp* als Variable hinzugefügt wird, werden im Formular lediglich die ausgewählten Elemente beibehalten, während die restlichen Eingaben leer sind. Die Methode *getInputComponents* der Hilfsklasse iteriert ebenfalls über die Formular- und Wrapper-Komponenten und fügt sie einer Liste hinzu, welche auf den Typ *FormElementComponentModel* beschränkt ist. Diese wird von der Methode zurückgegeben und enthält letztlich alle Eingabekomponenten des Formulars.

5.6.4 **Validierung**

Die Validierung des Formulars stellt sicher, dass die Eingaben eines Nutzers bestimmten Regeln entsprechen. Sollte dies nicht der Fall sein, kann eine spezielle Behandlung erfolgen. Da sich diese bei Verwendern des *dsformaddon* unterscheiden kann, ist es nicht mögliche, diese direkt zu implementieren. Die Validierung jedoch wird vom Addon bereitgestellt und kann im Storefront-Controller des jeweiligen Projekts verwendet werden.

In der abstrakten Klasse *DsFormController* befindet sich eine Methode *validateForm*, zu sehen in Abbildung 15, mit dem Sichtbarkeitsmodifikator *protected*. Dadurch kann sie von allen erbenden Klassen aufgerufen und überschrieben werden. Das ist wichtig, da nach Kapitel 5.5 eine Erweiterung des Controllers für eine individuelle Ereignisbehandlung vorausgesetzt wird.

```
protected void validateForm(final DsFormComponentModel form,
        final DsFormComponentForm dsFormComponentForm,
        final BindingResult errors) {
        List<FormElementComponentModel> subcomponents
        = DsFormUtil.getInputComponents(form);

        validateMandatoryFields(subcomponents, dsFormComponentForm,
            errors);
         validateInputFields(subcomponents, dsFormComponentForm, errors);
         validateDateChooserFields(subcomponents, dsFormComponentForm,
            errors);
         validateFileUploadFields(subcomponents, dsFormComponentForm,
         errors);
}
```

Abbildung 15: Methode *validateForm* der Klasse *DsFormController*

In der Methode kommt die Hilfsmethode *getInputComponents* zum Einsatz, die in Kapitel 5.6.3 beschrieben ist. Die Validierung der Formularfelder teilt sich in zwei wesentliche Schritte auf. Zuerst wird das Formular darauf geprüft, ob alle Pflichtfelder ausgefüllt wurden. Als Pflichtfelder gelten alle Komponenten, deren Attribut *mandatory* den Wert true besitzt. Die Überprüfung erfolgt in der privaten Methode *validateMandatoryFields*. In dieser wird über alle Pflichtfelder iteriert und abhängig vom Typ der Komponente entschieden ob eine Eingabe vorhanden ist. Bei den Texteingabefeldern und der Datumsauswahl gilt ein leerer String als fehlende Eingabe. Bei File-Upload-Komponenten wird eine fehlende Eingabe über die Methode *isEmpty* des entsprechenden *MultipartFile*-Objekts (Spring[41]) bestimmt.

Im zweiten Teil der Methode *validateForm* werden Validierungen für spezielle Eingabefelder durchgeführt. In *validateInputFields* werden die Texteingabefelder des

[41] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/multipart/MultipartFile.html

Formulars auf den regulären Ausdruck in ihrem Attribut *regex* geprüft. Nur dann, wenn die Nutzereingabe dem definierten Format entspricht, gilt das Feld als gültig. Für die Datumsauswahlfelder wird versucht, aus der Nutzereingabe mit der Standard Java-API ein *Date*-Objekt in einem bestimmten Format zu erzeugen. Sollte dies fehlschlagen, so gilt die Eingabe als ungültig. Die Methode *validateFileUploadFields* ist bisher nicht implementiert. Die Validierung der File-Upload Felder wird in Kapitel 5.6.7 genauer erläutert. Sie wurde dennoch bereits dem Controller hinzugefügt, damit diese von Verwendern des Addon angepasst werden kann. Nutzer des Addon können die Methode *validateForm* in einem eigenen Storefront-Controller verwenden. Alle Methoden des *DsFormController* sind als *protected* gekennzeichnet. Dadurch kann das Verhalten jederzeit überschrieben werden.

5.6.5 **Fehlerbehandlung**

Die Validierung kann feststellen, ob Eingabefehler im Formular vorliegen. Es wird anders auf das Absenden reagiert als bei fehlerfreien Eingaben. Zu häufigen Anwendungsfällen zählen zum Beispiel, den Nutzer auf das richtige Format der Daten hinzuweisen oder ihn auf ein leeres Pflichtfeld aufmerksam zu machen. Damit dieses Verhalten anpassbar ist, muss der Verwender des *dsformaddon* in der Lage sein, aufgetretene Fehler auswerten zu können. Das geschieht über ein Objekt des Typen *BindingResult*. Es wird in der Controllermethode *sendForm* als Parameter bekannt gemacht und von Spring initialisiert. Die Methode *validateForm* erwartet dieses Objekt ebenfalls als Parameter. Diesem wird bei Validierungsfehlern eine neue Instanz der Spring-Klasse *FieldError* hinzugefügt. Der abstrakte Controller besitzt dazu zwei private Methoden. In *createFieldErrorFromMessage* kann ein String als Fehlermeldung gesetzt werden, falls die Nachricht selbst festgelegt werden soll. Im Gegensatz dazu nimmt *createFieldErrorFromCode* einen Code in Form eines Strings entgegen und löst diesen über die Extension *dslocalization* (siehe Kapitel 5.1) auf.

Die Zuordnung dieser Codes zu den entsprechenden Fehlermeldungen ist in Lokalisierungsdateien definiert. Für die Sprachen Deutsch und Englisch sind das die Dateien *base_de.properties* und *base_en.properties.* Beide befinden sich jeweils im Verzeichnis *acceleratoraddon/web/webroot/WEB-INF/messages* des *dsformaddon*. Die Fehlermeldungen für leere Pflichtfelder werden über eine Impex-Datei bekannt gemacht. Diese heißt */resources/essentialdata-ds-form-component-error-message-localization.impex* (siehe Anlage 3). Das Prefix *essentialdata* bewirkt, dass die Datei bei jeder System-Initialisierung eingespielt wird. (siehe Kapitel 4.1.3)

Es gibt für jedes Feld und jeden Fehlerfall einen eigenen Fehlercode. So enthält zum Beispiel *dsform.field.empty.FormFileUploadElementComponentModel* die Fehlermeldung für ein leeres File-Upload Pflichtfeld und *dsform.field.wrong.date* für ein ungültiges Datum bei der Datumsauswahl. Da die Codes bei der Validierung im Falle der Ungültigkeit an das *BindingResult*-Objekt angehangen werden, lässt sich dieses nach der Validierung auswerten. Mit einer Fallunterscheidung können in der Implementierung des Storefront-Controllers die Fehler behandelt werden. In Umsetzungen anderer Projekte wurde der darstellenden JSP-Datei eine Konstante *HAS_VALIDATION_ERROR* mit dem jeweiligen Wert true oder false hinzugefügt. Dort kann dann die Darstellung entsprechend des gesetzten Wertes bestimmt werden. Dieses Vorgehen wird adaptiert und die Konstante in der Klasse *DsformaddonConstants* definiert. Sie kann von Entwicklern, die das *dsformaddon* nutzen, verwendet werden. Die Datei *ds.utils.js* stellt Funktionen bereit, welche Meldungen nach SAP Commerce Standard anzeigen. Sie werden bei Erfolg oder Misserfolg des AJAX-Request verwendet.

5.6.6 **FormDTO**

Da es viele verschiedene Ansätze für die Weiterverarbeitung der Formulardaten gibt, muss sichergestellt werden, dass die Daten an andere Schnittstellen weitergereicht werden können. Das erfolgt über ein Data Transfer Object (DTO). Martin Fowler beschreibt sie als „Objekte, die Daten zwischen Prozessen austauschen, um die Anzahl

von Methodenaufrufen zu reduzieren“.[42] Das geschieht, indem ein einzelnes Objekt Attribute für andere Objekte oder primitive Daten hält. Um die Daten an einen anderen Prozess des Gesamtsystem weiterzureichen, muss lediglich das DTO übertragen werden. Es ist nicht nötig, jeden Wert einzeln über Methodenaufrufe anzufragen.
Das DTO selbst besitzt neben den benötigten Attributen, die es transportieren soll, die zugehörigen Getter- und Setter-Methoden. Abbildung 16 zeigt für das *FormDTO* des dynamischen Formulars ein vereinfachtes Klassendiagramm.

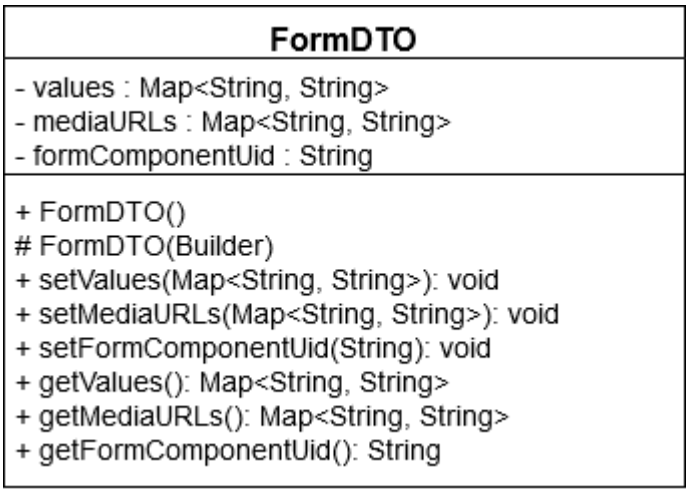


Abbildung 16: Klassendiagramm zu FormDTO

Es fällt auf, dass einer der beiden Konstruktoren einen Parameter vom Typ *Builder* entgegennimmt. Die Erstellung von *FormDTO*-Instanzen erfolgt nach dem Builder-Pattern.[43] Die Klasse enthält eine innere Klasse *Builder*, welche die gleichen Attribute wie *FormDTO* besitzt. Zusätzlich enthält sie Setter-Methoden, deren Namen äquivalent zu den Attributbezeichnern sind und das *Builder*-Objekt selbst zurückliefern. Die Methode *build* liefert ein *FormDTO* zurück, welches mit dem Objekt als Parameter über den entsprechenden Konstruktor instanziiert wird. Der Konstruktor setzt lediglich die Attribute des *FormDTO*-Objekts auf die des *Builder*-Objekts.

---

[42] [Fow19]
[43] https://dzone.com/articles/design-patterns-the-builder-pattern

In der dotSource GmbH wird dieses Pattern projektübergreifend verwendet. Klassen nach dieser Struktur entstehen somit häufig. Damit es nicht nötig ist, diese manuell zu erstellen, werden sie automatisch generiert. Das geschieht über ein Velocity-Template[44]. Darüber kann eine Grundstruktur vorgegeben werden, auf deren Basis es möglich ist, Dateien zu erzeugen.[45] Im vorliegenden Fall handelt es sich dabei um einfache Java-Klassen mit Attributen, Getter- und Setter-Methoden, welche eine innere *Builder*-Klasse enthalten. Das Format dieser Klassen wird in der Datei *beans-with-builders-template.vm* definiert. Sie dient als Vorlage und wird bereits in anderen Projekten der dotSource GmbH verwendet. In der Datei *dsformaddon-beans.xml* werden die Instanzvariablen zu *FormDTO* festgelegt. SAP Commerce legt beim Erstellprozess des Addon den beschriebenen Typen an. Über das Attribut *template* kann ein Velocity-Template angegeben werden, welches die Dateistruktur beschreibt. Abbildung 17 zeigt das für den Fall des *FormDTO*. Die Zeichenkette „<“ maskiert das Zeichen „<“.

```
<beans xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:noNamespaceSchemaLocation="beans.xsd">
        <bean class="de.dotsource.dsform.dtos.form.FormDTO"
        template="resources/beans-with-builders-template.vm">
            <property name="values"
                type="java.util.Map&lt;String, String>" equals="true"/>
            <property name="mediaURLs"
                type="java.util.Map&lt;String, String>" equals="true"/>
            <property name="formComponentUid"
                type="String" equals="true"/>
        </bean>
</beans>
```

Abbildung 17: Typdefinition von *FormDTO* in *dsformaddon-beans.xml*

Durch das Builder-Pattern werden Instanzen an einer zentralen Stelle erzeugt. Das ist vor allem wichtig für die Skalierbarkeit und den Überblick. Würde eine Instanziierung über den new-Operator in Java erfolgen, hätte das zur Folge, dass bei einem wachsenden Projekt die Anzahl der Aufrufe des *FormDTO*-Konstruktors ansteigt. Sollte das DTO zukünftig um Attribute erweitert werden, müssten diese Aufrufe im gesamten

[44] https://velocity.apache.org/
[45] [Apa16]

Code ebenfalls um die neuen Parameter ergänzt werden. Bei Verwendung des Patterns gibt es lediglich eine Stelle, die angepasst werden muss – die Builder-Klasse. Im vorliegenden Projekt orientiert sich diese an der Datei *dsformaddon-beans.xml*, wodurch sich dortige Anpassungen direkt global auswirken.

Die Klasse *DsFormController* enthält eine Methode *getFormDTO*. Diese erstellt anhand der Eingaben, die aus dem Parameter vom Typ *DsFormComponentForm* ausgelesen werden, ein *FormDTO* und gibt dieses zurück. Das ermöglicht es den Entwicklern, die eingegebenen Daten aus dem implementierten Storefront-Controller an andere Teilsysteme weiterzureichen.

5.6.7 **File-Upload**

Die File-Upload-Komponente benötigt im Vergleich zu den übrigen Komponenten deutlich mehr Geschäftslogik. Eines der Attribute von *FormDTO* besitzt den Bezeichner *mediaURLs*. Dateien, die über das dynamische Formular übertragen werden, sollen serverseitig abgespeichert und über einen Uniform Resource Identifier zugänglich gemacht werden. In dem Attribut werden alle URIs zu Dateien gehalten, die das Formular sendet.

Der Einstiegspunkt für diese Logik ist die Methode *getFormDTO*, die bereits in Kapitel 5.6.6 zusammengefasst wurde. Da diese über alle Werte des Formulars iteriert, kann für jeden Wert abgefragt werden, ob dieser vom Typ *MultipartFile* ist. Sollte das der Fall sein, so wird ein Media-Objekt erstellt und nachfolgend dessen URI in die Map *mediaURLs* sowie der Dateiname in die Map *values* gespeichert. Als Schlüssel wird jeweils der Name des Eingabefelds verwendet. Die Erstellung des Media-Objekts geschieht über die Methode *createMediaFromUserUpload* (siehe Anlage 4) aus der Klasse *DefaultDsFormMediaFacade*, welche das Interface *DsFormMediaFacade* implementiert.

In dieser wird zuerst eine temporäre Kopie aus dem *MultipartFile*-Objekt erzeugt, die der Methode *checkForValidFile* der Klasse *DefaultDsFormMediaService* als Parameter übergeben wird. Darin wird geprüft, ob die Entgegennahme der Datei zulässig ist. Die momentan unterstützten Formate sind JPEG, PNG, TXT und PDF. [redacted]

Sollte die temporäre Datei als nicht zulässig gelten, so wird eine *FileUploadException* geworfen. Ansonsten wird mithilfe von SAP Commerce Standard-Funktionen ein Media-Objekt angelegt. In SAP Commerce kann jede Datei als Media repräsentiert werden.[46] Dem Objekt wird ein eindeutig identifizierbarer Code mit dem Prefix „dsform_upload_“ hinzugefügt. Das Verzeichnis, in dem die Datei gespeichert wird, ist vorkonfiguriert auf „/images“ und wird in einer Konstanten *DS_FORM_UPLOAD* in der Datei *DsformaddonConstants.java* festgelegt.

Der Name des Ordners ist an dieser Stelle schlecht gewählt, lässt sich jedoch dadurch rechtfertigen, dass in den drei bestehenden Projekten, welche ein dynamisches Formular implementieren, bisher fast ausschließlich Bilder über das Eingabefeld übermittelt wurden. In Zukunft sollte der Zielordner frei wählbar sein.

Die Daten der temporären Datei werden im Zielpfad und das Media-Objekt, das diese referenziert, in der Datenbank gespeichert. Anschließend wird die temporäre Datei gelöscht und das Media-Objekt von der Methode *createMediaFromUserUpload* zurückgegeben.

Eventuell auftretende *FileUploadExceptions* werden von der Methode *getFormDTO* weitergeworfen. Das bedeutet, dass es mit dieser Implementierung für die Entwickler erst dann möglich ist, die Gültigkeit der gesendeten Dateien zu überprüfen, wenn sie die Methode im Storefront-Controller aufrufen. Dort kann dann die Exception behandelt werden.

---

[46] [SAP19n]

Die Auflösung der Multipart-Dateien übernimmt Spring mit dem Interface *MultipartResolver*. Dieses muss vorher konfiguriert werden, damit Dateien problemlos im Formular versendet werden können[47]. Dazu wird der Ausschnitt aus Abbildung 18 in der Datei *dsformaddon-spring.xml* ergänzt. Dieser teilt Spring mit, wie mit Formularen, die Multipart-Dateien senden, verfahren werden soll. Das Attribut *maxUploadSize* spezifiziert die maximale Dateigröße in Bytes, die zulässig ist, und wird auf 5 MiB gesetzt. Größere Dateien sorgen für eine *MaxUploadSizeExceededException*. Durch das Setzen von *resolveLazily* auf den Wert true wird dieser Fehler erst dann geworfen, wenn per Parameter auf die Datei zugegriffen wird. Das geschieht im Controller.

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
<!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="5242880"/> <!-- 5 MiB -->
    <property name="resolveLazily" value="true"/>
</bean>
```

Abbildung 18: Konfiguration von *MultipartResolver* in *dsformaddon-spring.xml*

Die Ausnahme wird behandelt, indem im *DsFormController* eine Methode mit der Annotation *@ExceptionHandler* und einem Verweis auf die *MaxUploadSizeExceededException*-Klasse hinzugefügt wird. Diese macht ein Attribut in der JSP-Datei bekannt, das es erlaubt, diesen Fehler abzufragen und setzt den HTTP-Statuscode auf *413 Payload too large*. Die Methode kann überschrieben werden, um eine individuelle Fehlerbehandlung zu implementieren.

SAP Commerce verwendet als Webserver Tomcat[48]. Dieser kann ebenfalls Einschränkungen bezüglich der zugelassenen Dateigrößen von Multipart-Files spezifizieren. Es muss darauf geachtet werden, dass der Wert *maxSwallowSize,* der die maximale Anzahl Bytes einer Serveranfrage angibt[49], in der Datei *conf/server.xml* nicht kleiner als der Wert *maxUploadSize* des *MultipartResolver* ist. Ansonsten ignoriert Tomcat die Anfrage und sendet keine Antwort. Abbildung 19 stellt diese Abhängigkeit grafisch dar.

---

[47] [SAP19o]
[48] https://tomcat.apache.org/
[49] [Apa18]

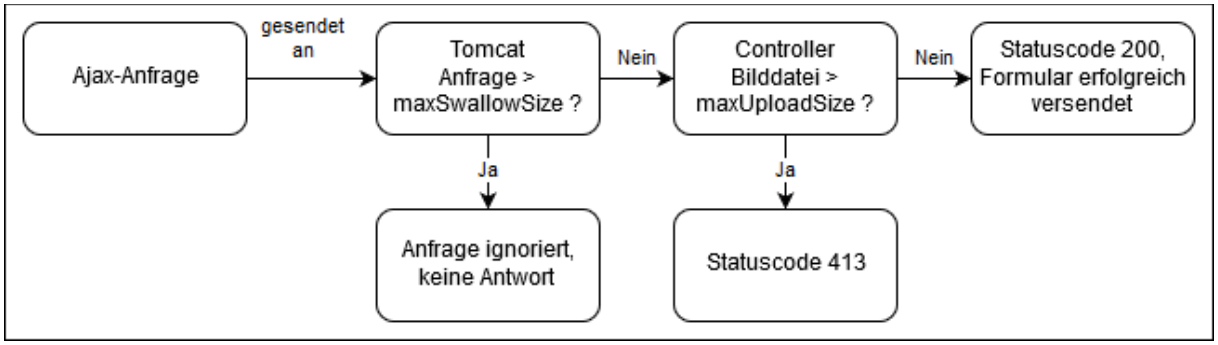


Abbildung 19: Prozess der Größenprüfung von Dateien

5.7 Integrationsmöglichkeiten in Projekte

Damit das dynamische Formular verwendet werden kann, müssen die Schritte bei der Einrichtung beachtet werden. Davon sind einige bereits in Kapitel 5.1 detailliert beschrieben. Zuerst muss der Quellcode dem Projekt hinzugefügt werden. Es gibt dazu zwei Ansätze. Der erste ist das simple Kopieren der Quelldateien des Addon in ein Verzeichnis unterhalb von *hybris/bin*. Ansatz zwei ist die Integration des vorhandenen Git Repository als Git Submodul. Generell wird für die Einbindung firmeninterner Extensions nach Konvention das Verzeichnis *hybris/bin/custom/dotsource* gewählt.

Das Kopieren des Quellcodes ist simpel und mit wenig Aufwand verbunden. Das Addon wird lediglich in das genannte Verzeichnis kopiert und kann dann in den nächsten Schritten dem Projekt als Abhängigkeit hinzugefügt werden. Es ist ohne weiteres möglich, projektspezifische Anpassungen, beispielsweise aufgrund von Kundenwünschen, in dem Addon vorzunehmen. Das kann jedoch problematisch werden, wenn das projektexterne Addon weiterentwickelt wird. Soll der neue Stand ebenfalls im Projekt Verwendung finden, so müsste es erneut kopiert werden, wodurch Anpassungen verloren gehen. Es ist zwar möglich, die hinzugefügte Funktionalität wie anfangs einzeln zu kopieren, allerdings wäre das sehr aufwändig, da jede Änderung einzeln übernommen werden muss und unter Umständen nicht mit dem veränderten Code kompatibel ist. Wenn der Stand des, in dieser Arbeit entwickelten, Addon über einen langen Zeitraum

hinweg unverändert bleibt, so ist dieser Ansatz aufgrund der einfachen Umsetzung zu bevorzugen.

Die Integration über ein Git Submodul ist aufwändiger und weniger trivial. Die Einrichtung erfolgt über den folgenden Befehl:

```
git submodule add https://gitlab.dotsource.de/hybris/dsformaddon
hybris/bin/custom/dotsource/dsformaddon && git submodule update --init
```

Dadurch wird das Submodul initialisiert und der Inhalt des Git-Repository *dsformaddon* übernommen. Da Git nun den Ursprung des Quellcodes kennt, können Änderungen in dem Repository des Addon jederzeit in alle Projekte übernommen werden, die das entsprechende Submodul verwenden. Eigene Anpassungen können zwar nach wie vor im Projektunterverzeichnis vorgenommen werden, kommen jedoch eventuell mit denen des Repository in Konflikt und müssen dann gelöst werden. Vorteilhaft ist, dass neue Versionen von *dsformaddon* schnell, und ohne das gesamte Ersetzen des Addon, verfügbar sind. Um neue Versionen zu übernehmen, muss jedoch jedes Mal

```
git submodule update
```

ausgeführt werden. Es ist demnach notwendig, dass darüber alle betroffenen Projekte informiert werden und innerhalb dieser, jeder Entwickler den Befehl ausführt. Bei guter interner Kommunikation ist dieses Problem zu vernachlässigen. Die Verwendung eines Git Submoduls ist eine gute Alternative, wenn das Repository regelmäßig wichtige Änderungen erfährt, die in den Projekten verwendet werden sollen. Aufgrund des benötigten Verständnisses über das Setup ist aber gleichzeitig die Fehleranfälligkeit deutlich höher.

Die nachfolgenden Schritte sind unabhängig davon, welcher der beiden Ansätze zum Einsatz kommt. In das Verzeichnis *hybris/bin/custom/dotsource* muss ebenfalls die

Extension *dslocalization* ergänzt werden, da das *dsformaddon* von dieser abhängig ist. In der Datei *hybris/config/localextensions.xml* des jeweiligen Projekts sind die beiden Einträge aus Abbildung 20 hinzuzufügen.

```
<extension name="addonsupport" />
<extension name="dsformaddon" />
```

Abbildung 20: Zusätzliche Einträge in *localextensions.xml*

Sollte SmartEdit noch nicht verwendet werden, so ist es außerdem notwendig, die Einträge aus Abbildung 4 zu übernehmen. Das Addon muss anschließend in die Storefront-Extension installiert werden. Beides lässt sich in Kapitel 5.1 genauer nachlesen.

Wie in Kapitel 5.6.2 erwähnt, muss das JavaScript, welches sich um den AJAX-Request des dynamischen Formulars kümmert, auf jeder Seite geladen werden. In SAP Commerce kann das über eine Ergänzung in der Datei *web/webroot/_ui/responsive/common/js/_autoload.js* der Storefront-Extension erfolgen. Siehe hierzu Anlage 5. In der Realität wird dafür jedoch oft eine eigene Lösung in den Projekten umgesetzt. Wichtig ist, dass das JavaScript auf den Seiten geladen wird, die ein dynamisches Formular enthalten. Ansonsten ist dieses nicht funktional. In der Datei *web/webroot/WEB-INF/tags/responsive/template/javaScript.tag* wird außerdem das JavaScript *ds.utils.js* bekannt gemacht, damit dieses global verwendet werden kann. (siehe Abbildung 21)

```
<script src="/dsformstorefront/_ui/addons/dsformaddon/
        responsive/common/js/ds.utils.js"></script>
```

Abbildung 21: Ergänzung des Scripts *ds.utils.js* in *javaScript.tag*

In der Storefront-Extension des Projekts, welches das *dsformaddon* einbindet, muss der *DsFormController* erweitert werden. Als Referenz befindet sich im Verzeichnis *resources* des Addon eine Beispielimplementierung für den Storefront-Controller. (siehe

Anlage 6) Dieser implementiert die Methode *sendForm*. Anschließend muss das Projekt mithilfe von

```
ant build
```

kompiliert und über die HAC ein Update mit ausgewähltem *dsformaddon* ausgeführt werden (siehe Kapitel 5.1).

In der vorliegenden Bachelorarbeit steht die ästhetische Darstellung des Formulars im Hintergrund. Daher werden auch keine Anpassungen bezüglich CSS vorgenommen. Es bleibt dem Entwickler vorbehalten, dieses an die Vorgaben des Projekts anzupassen. Die einzige Ausnahme bildet die Ergänzung aus Abbildung 22 in der Datei *web/webroot/_ui/responsive/common/css/dsformaddon.css*.

```
.radio input[type="radio"],
.radio-inline input[type="radio"],
.checkbox input[type="checkbox"],
.checkbox-inline input[type="checkbox"] {
    margin-left: 0px;
}
```

Abbildung 22: Ergänzung von CSS in der Datei *dsformaddon.css*

SAP Commerce definiert standardmäßig einen Wert von -20 Pixel für die Eigenschaft *margin-left* von Input-Elementen des Typs „radio“ und „checkbox“. Das bedeutet, wenn sich diese am äußersten linken Rand der Seite befinden, sind diese außerhalb des Darstellungsbereich des Browsers und somit nicht sichtbar. Den Wert auf 0 Pixel zu ändern, setzt dieses Verhalten zurück. Sollte weiteres spezifisches CSS notwendig sein, so kann dieses in der Datei hinzugefügt werden.

Nachdem das Projekt kompiliert wurde und ein Update über die HAC erfolgt ist, kann über SmartEdit ein dynamisches Formular mit entsprechenden Formularfeldern angelegt werden.

6 Abbildung von Use-Cases als Demo

6.1 Retoure-Formular

Um die Verwendung des dynamischen Formulars zu demonstrieren, erfolgte die Umsetzung eines realistischen Use-Case. Unter Nutzung der Testumgebung aus Kapitel 5.1 wurde für einen Online-Shop ein Formular angelegt, welches Daten für die Retoure eines Produkts entgegennimmt. Zuerst wird über die Oberfläche des SmartEdit eine Formularkomponente per Drag and Drop in die Seite platziert. Im sich öffnenden Fenster wird der Name des Formulars, die Beschriftung des Abschicken-Buttons sowie ein Einleitungstext über dem Formular ausgefüllt. In der Liste der Formularfelder werden Komponenten ergänzt, welche die Eingabefelder repräsentieren. Abbildung 23 zeigt den schematischen Aufbau.

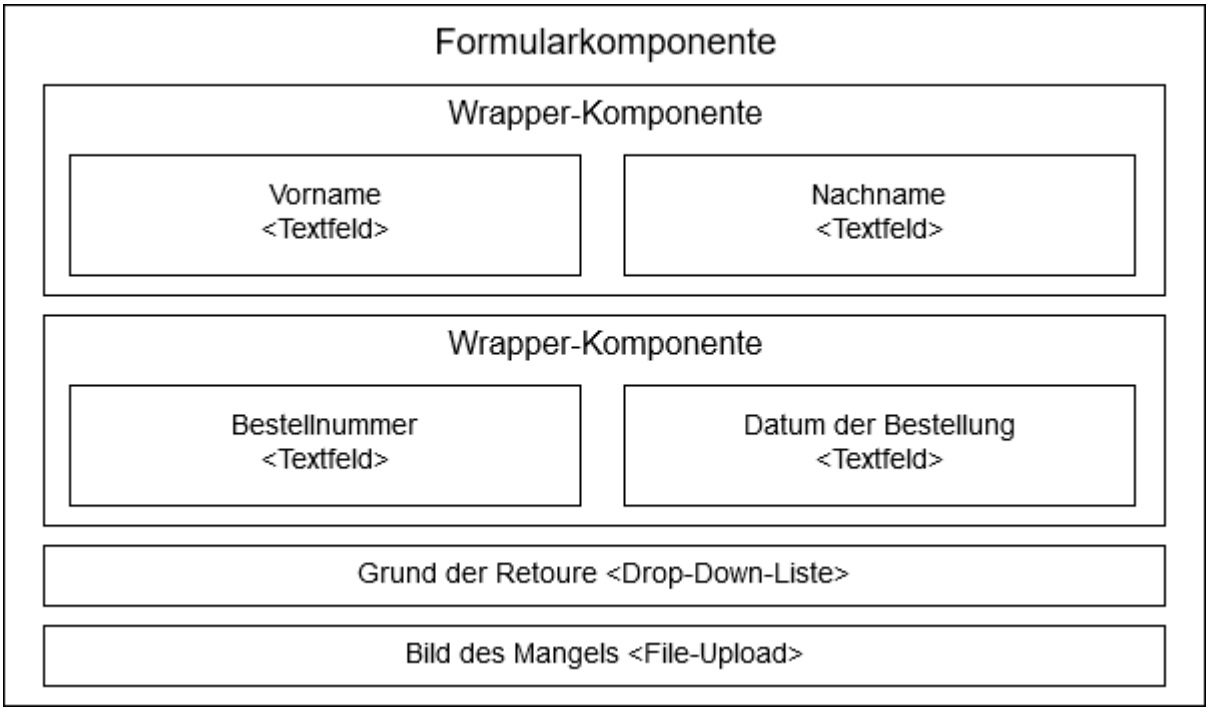


Abbildung 23: Schematischer Aufbau des Retoure-Formulars

Die vollständig ausgefüllte Oberfläche im SmartEdit ist in Anlage 7 zu sehen. Das Formular enthält zwei Wrapper-Komponenten, von denen eine die Felder für Vor- und Nachname und die andere für Bestellnummer und -datum enthält. Darunter befindet sich die Liste für den Grund der Retoure und das optionale File-Upload-Feld. Letzteres ist als einziges kein Pflichtfeld. Wie das erstellte Retoure-Formular in der Storefront aussieht zeigt Abbildung 24.

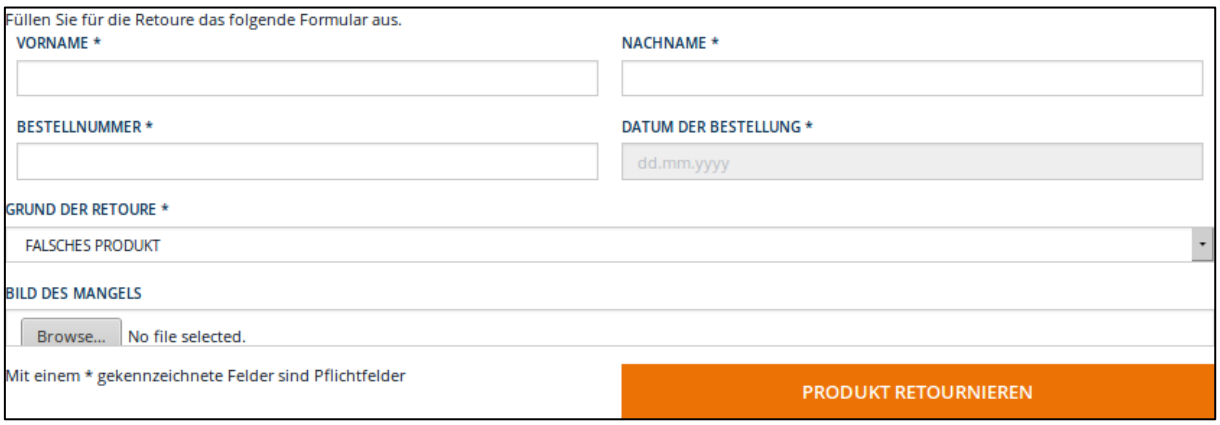


Abbildung 24: Retoure-Formular in der Storefront

6.2 Gewinnspiel-Formular

Ein weiterer Anwendungsfall ist ein Gewinnspiel auf der Onlineseite, um neue Kunden zu werben. Unter Verwendung des *dsformaddon* kann ein entsprechendes Formular erstellt werden. Das geschieht über SmartEdit. In Abbildung 25 ist der Schematische Aufbau des Gewinnspiel-Formulars dargestellt.

Für die Namensfelder kann die Wrapper-Komponente aus Kapitel 6.1 wiederverwendet werden. Eine weitere Wrapper-Komponente beinhaltet Felder für E-Mail und Telefonnummer. Darunter befinden sich die Radio Buttons und die Checkbox. Abbildung 26 zeigt das Gewinnspiel-Formular in der Storefront. Die Übersicht im SmartEdit befindet sich in Anlage 8.

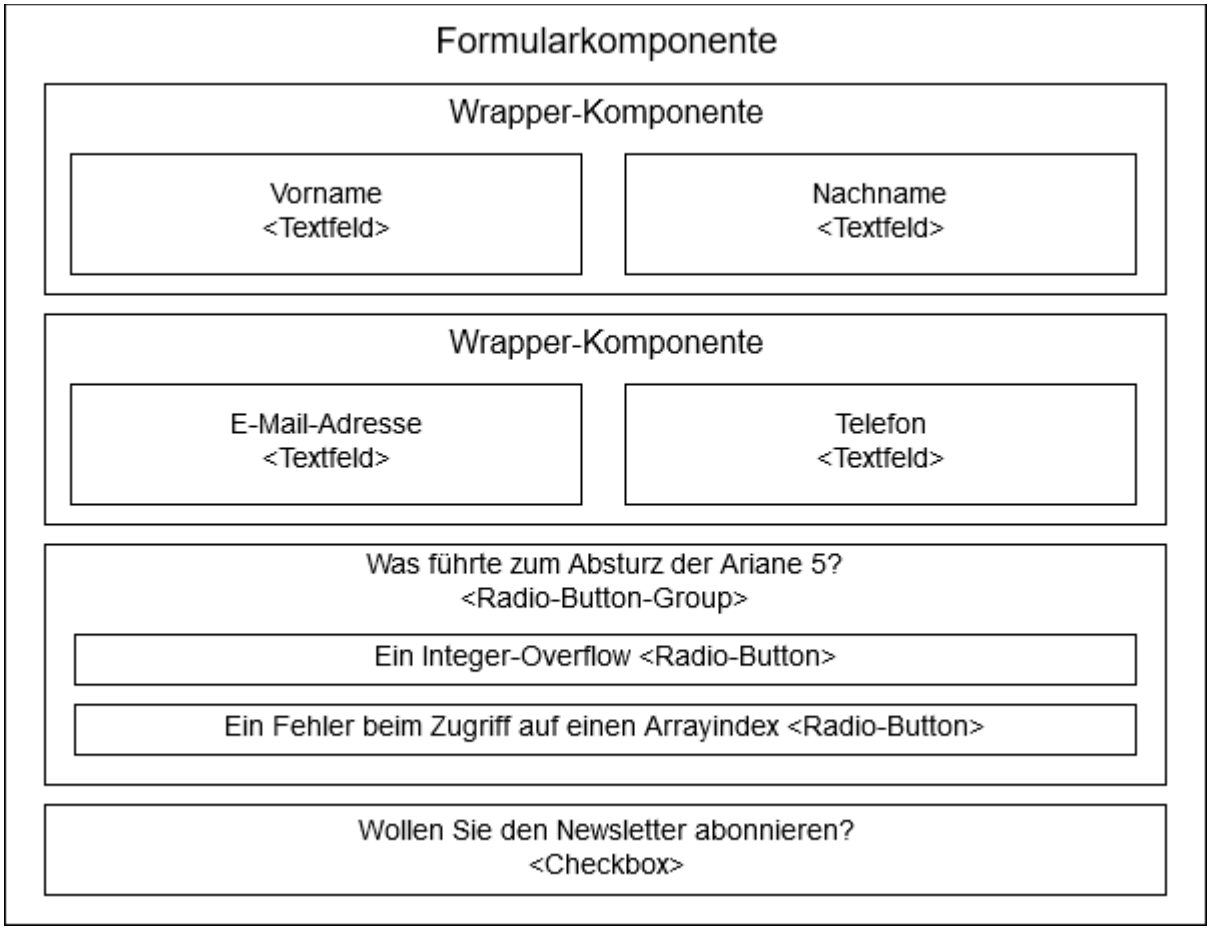


Abbildung 25: Schematischer Aufbau des Gewinnspiel-Formulars

Abbildung 26: Gewinnspiel-Formular in der Storefront

7 Fazit und Ausblick

Das Ziel dieser Bachelorarbeit war es, das Verwalten von dynamischen Formularen im SmartEdit zu ermöglichen. Der Anwender soll die freie Wahl über enthaltene Formularfelder haben und diese beliebig anordnen können. Es sollte die modularisierte Wiederverwendbarkeit in zukünftigen Projekten der dotSource GmbH ermöglicht werden. Als erster möglicher Lösungsansatz wurde die SAP Commerce Extension yForms vorgestellt, die einen großen Funktionsumfang für die Erstellung solcher Formulare besitzt. Die Alternative dazu war die Erstellung eines Addon für SAP Commerce. Nach einem Vergleich der beiden Möglichkeiten, fiel die Wahl auf das Addon. Ausschlaggebend dafür war die fehlende SmartEdit-Untersützung von yForms. Für die Implementierung wurden drei Umsetzungen in Kundenprojekten ausgewertet. Deren Inhalte wurden für das Addon adaptiert und zusammengefasst. Abschließend wurden zur Demonstration ein Retoure- und ein Gewinnspiel-Formular erstellt.

Das Addon lässt sich einfach in andere Projekte integrieren und erweitert diese im SmartEdit um eine Formularkomponente mit entsprechenden Feldern. Die Formularfelder besitzen jeweils eine Validierung, um eingegebene Daten auf ihre Gültigkeit zu überprüfen. Es wurde ein Konzept erstellt, um das Addon in Projekten vielfältig einsetzen zu können. Aus diesem ging die Erstellung eines Controllers hervor, der für die Behandlung des Formulars implementiert werden muss. In einer vorgegebenen Methode kann der Umgang mit den versendeten Formulardaten geregelt werden.

Um die Integration zu erleichtern und den Entwicklern dahingehend mehr Arbeit abzunehmen, könnte das Addon erweitert werden, um häufige Anwendungsfälle abzudecken. Beispielsweise kann der Versand der Formulardaten per E-Mail oder das Persistieren über eine entsprechende Methode bereits ermöglicht werden.

Zukünftig sollte die Logik für das Hochladen von Dateien angepasst werden. Die zugehörige Validierung sollte in den Controller übernommen werden. Momentan ist nur der URI zu den gespeicherten Dateien abrufbar. Sollten die Formulardaten jedoch an

externe Systeme übermittelt werden, muss dies unter Umständen in einer anderen Form geschehen. Es müsste eine Implementierung ergänzt werden, welche die URI eines Formular-Objekts in das vorgegebene Format umwandelt.

Es sollte außerdem möglich sein, die unterstützten Formate, die maximale Dateigröße sowie den Speicherpfad an der File-Upload-Komponente zu bestimmen. Bisher sind die jeweiligen Werte fix gesetzt. Die Komponente muss dafür um die entsprechenden Attribute erweitert werden und die restliche Logik muss diese anstelle der Konstanten berücksichtigen.

Um die Seitenladezeiten nicht unnötig zu belasten, sollte nach einer Alternative gesucht werden, bei der das JavaScript des entwickelten Formulars nicht auf jeder Seite geladen wird. Da ein Formular in den meisten Fällen nur auf einer geringen Anzahl von Seiten vorkommt, kann vor dem Laden eventuell eine Abfrage stattfinden, die das JavaScript nur zugänglich macht, sollte dieses gebraucht werden. Auf diese Weise würden viele Seiten schneller laden und eine bessere Nutzererfahrung bieten. Im Kontext von SAP Commerce, kann das über die Datei *_autoload.js* (siehe Anlage 5) geschehen. In dieser kann eine Bedingung für das Laden eines JavaScripts angegeben werden. Im Falle der vorliegenden Bachelorarbeit wäre diese Bedingung das Vorhandensein einer Formularkomponente.

Bisher hat der Ersteller eines Formulars keinen Einfluss auf die Weiterverarbeitung der Daten. Ein Ansatz wäre es, an der Formularkomponente ein Feld zu ergänzen, dessen Wert bestimmt, welche Verarbeitungsroutine aufgerufen wird. Dem Ersteller eines Formulars würden so noch mehr Freiheiten geboten werden.

Das Addon kann intern für bestehende und zukünftige Projekte verwendet werden. Die einfache Integration kann Zeitersparnisse bei der Entwicklung bewirken. Es ist zu erwarten, dass die durchschnittliche Zeit zur Fertigstellung eines Projekts sinkt. Für Kunden der dotSource GmbH bedeutet das eine frühere Umsatzgenerierung. Die dotSource GmbH kann die eingesparten Aufwände in andere Entwicklungen investieren. Trotz der benannten Potentiale erzielt die entstandene Lösung auf diese Weise bereits für beide Parteien einen Mehrwert. Eine Fortführung dieser Bachelorarbeit könnte sich mit einer Evaluierung von Realisierungszeiträumen befassen und so die Rentabilität des Addon genauer untersuchen.

VI Literaturverzeichnis

[Apa16] Apache Software Foundation: „What is Velocity?”, 2016, http://velocity.apache.org/engine/1.7/user-guide.html#what-is-velocity Abruf: 17.07.2019

[Apa18] Apache Software Foundation: „The HTTP Connector”, 2018, https://tomcat.apache.org/tomcat-8.0-doc/config/http.html Abruf: 17.07.2019

[Fow19] Fowler, M.: „Data Transfer Object”, 2019, https://martinfowler.com/eaaCatalog/dataTransferObject.html Abruf: 17.07.2019

[ITD19] IT Definitions: „Modularity”, 2019, https://www.defit.org/modularity/ Abruf: 24.07.2019

[Mar08] Martin, R. C.: „Clean Code: A Handbook of Agile Software Craftsmanship”, 1. Auflage, Verlag Prentice Hall, Westford Massachusetts, 2008

[McL11] McLellan, D.: „The Best Forms Implementation I’ve Ever Built”, 2011, https://allinthehead.com/retro/358/the-best-forms-implementation-ive-ever-built Abruf: 17.07.2019

[Moz19a] Mozilla Foundation: „What's AJAX?", 2019, https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started
Abruf: 17.07.2019

[Moz19b] Mozilla Foundation: „FormData", 2019, https://developer.mozilla.org/de/docs/Web/API/FormData
Abruf: 17.07.2019

[OWA18] OWASP: „Cross-Site Request Forgery (CSRF)", 2018, https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
Abruf: 17.07.2019

[SAP19a] SAP: „Omnichannel Commerce", 2019, https://www.sap.com/products/crm/e-commerce-platforms/features.html
Abruf: 17.07.2019

[SAP19b] SAP: „Extensions", 2019, https://help.sap.com/viewer/3fb5dcdfe37f40edbac7098ed40442c0/1811/en-US/d5a5a6185c314af09304520716e2065a.html
Abruf: 17.07.2019

[SAP19c] SAP: „Extension Concept", 2019, https://help.sap.com/viewer/b490bb4e85bc42a7aa09d513d0bcb18e/1811/en-US/8bbf0b9d866910149688b8d696c8d47e.html
Abruf: 17.07.2019

[SAP19d] SAP: „Creating a New Extension”, 2019, https://help.sap.com/viewer/b490bb4e85bc42a7aa09d513d0bcb18e/1811/en-US/8b96270b86691014b1c3bbfd7556f0ed.html Abruf: 17.07.2019

[SAP19e] SAP: „AddOn Concept”, 2019, https://help.sap.com/viewer/b490bb4e85bc42a7aa09d513d0bcb18e/1811/en-US/8adc7ca3866910148ddfe860464f0fc4.html Abruf: 17.07.2019

[SAP19f] SAP: „Copying Files Between an AddOn and a Target Storefront”, 2019, https://help.sap.com/viewer/b490bb4e85bc42a7aa09d513d0bcb18e/1811/en-US/8acbff4b866910148e4be4ac7412dc83.html Abruf: 17.07.2019

[SAP19g] SAP: „ImpEx”, 2019, https://help.sap.com/viewer/d0224eca81e249cb821f2cdf45a82ace/1811/en-US/8bee52978669101498548981 87b16c96.html Abruf: 17.07.2019

[SAP19h] SAP: „Essential and Project Data by Convention”, 2019, https://help.sap.com/viewer/3fb5dcdfe37f40ed-bac7098ed40442c0/1811/en-US/9236d781bd6a4330ad12dbc3b8880e77.html Abruf: 17.07.2019

[SAP19i] SAP: „Spring Usage”, 2019, https://help.sap.com/viewer/4c33bf189ab9409e84e589295c36d96e/1811/en-US/8aef69e986691014a9179a7d4ffc1359.html Abruf: 17.07.2019

[SAP19k] SAP: „Deprecation Status”, 2019, https://help.sap.com/viewer/dc198ac31ba24dce96149c8480be955f/6.7.0.0/en-US/8bb15ed586691014a948d1553f4947cf.html Abruf: 17.07.2019

[SAP19l] SAP: „SAP Commerce Architecture”, 2019, https://help.sap.com/viewer/b490bb4e85bc42a7aa09d513d0bcb18e/1811/en-US/8b5588d8866910149d4eb5f99c75b6b4.html Abruf: 24.07.2019

[SAP19m] SAP: „addonsupport Extension”, 2019, https://help.sap.com/viewer/4c33bf189ab9409e84e589295c36d96e/1811/en-US/8abf96c9866910149fbeba7ee11df038.html#addonsupportextension-technicalguide-registeringacustomrendererforaspecificcmscomponenttypecode Abruf: 08.07.2019

[SAP19n] SAP: „Media”, 2019, https://help.sap.com/viewer/d0224eca81e249cb821f2cdf45a82ace/1811/en-US/8c0f64c786691014aa95f9f6b572f248.html Abruf: 08.07.2019

[SAP19o] SAP: „Media API”, 2019, https://help.sap.com/viewer/86dd1373053a4c2da8f9885cc9fbe55d/1811/en-US/e4266bd6987d4203b60f57dd0595dc43.html#loioe4266bd6987d4203b60f57dd0595dc43__section_xql_n5t_3z Abruf: 08.07.2019

[Ull10] Ullenboom, C.: „Java ist auch eine Insel – Das umfassende Handbuch”, 2010, http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_23_001.htm Abruf: 24.07.19

[W3C19] W3C: „Forms”, 2019, https://www.w3.org/TR/html401/interact/forms.html#h-17.13.4 Abruf: 17.07.2019

VII Anlagen

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="cms" uri="http://hybris.com/tld/cmstags" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<c:url var="encodedUrl" value="/xhr/dsForm/send"/>
<c:set var="mandatoryTrue" value="false"/>
<spring:message var="uploadSizeExceededErrorMessage"
code="text.dsform.send.error.uploadSizeExceeded"/>
<spring:message var="errorMessage" code="text.dsform.send.error"/>
<spring:message var="successMessage" code="text.dsform.send.success"/>

<form:form id="formComponent-${component.uid}"
           action="${encodedUrl}"
           method="POST"
           data-role="multipartAjaxForm"
           enctype="multipart/form-data"
           data-successMessage="${successMessage}"
           data-errorMessage="${errorMessage}"
           data-uploadSizeExceededErrorMessage
               ="${uploadSizeExceededErrorMessage}"
           commandName="dsFormComponentForm"
           modelAttribute="dsFormComponentForm"
           novalidate="novalidate">

    <input type="hidden" name="componentUid" value="${component.uid}"/>
    <c:forEach items="${component.componentList}" var="content">
        <div class="tn-mv-10">
            <c:if test="${content.typeCode != 'WrapperComponent'
               && content.mandatory}">
                <c:set var="mandatoryTrue" value="true"/>
            </c:if>
            <cms:component component="${content}"/>
        </div>
    </c:forEach>
    <div class="row">
        <c:if test="${mandatoryTrue}">
            <div class="col-tn-12 col-xs-6">
                <spring:theme code="login.required.message"/>
            </div>
        </c:if>
        <div class="col-tn-12 ${mandatoryTrue ?
               'col-xs-6 tn-pt-20 xs-pt-0' : ''}">
            <button class="btn btn-primary btn-block">
                <c:choose>
                    <c:when test="${not empty component.submitButton}">
                        ${component.submitButton}
                    </c:when>
                    <c:otherwise>
                        <spring:theme code="contact.form.send"/>
                    </c:otherwise>
                </c:choose>
            </button>
        </div>
    </div>
</form:form>
```

Anlage 1: ***dsFormComponentForm.tag***

```
package de.dotsource.dsformaddon.controllers.cms;

import de.dotsource.dsformaddon.model.DsFormComponentModel;
import de.dotsource.dsformaddon.controllers.DsformaddonControllerConstants;
import de.dotsource.dsformaddon.util.DsFormUtil;
import de.hybris.platform.addonsupport.controllers.cms.
        AbstractCMSAddOnComponentController;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;

@Controller("DsFormComponentController")
@RequestMapping(value
        = DsformaddonControllerConstants.Actions.Cms.DsFormComponent)
public class DsFormComponentController
        extends AbstractCMSAddOnComponentController<DsFormComponentModel> {
    private static final String ATTRIBUTE_COMPONENT = "component";
    private static final String ATTRIBUTE_COMPONENT_FORM
        = "dsFormComponentForm";

    @Override
    protected void fillModel(HttpServletRequest request, Model model,
                DsFormComponentModel component) {
        model.addAttribute(ATTRIBUTE_COMPONENT, component);
        model.addAttribute(ATTRIBUTE_COMPONENT_FORM,
        DsFormUtil.emptyForm(component));
    }
}
```

Anlage 2: ***DsFormComponentController.java***

```
INSERT_UPDATE DSLocalizationValue
; key[unique = true]
; value[lang = de]
; value[lang = en]
; dsform.field.empty.FormDateChooserElementComponentModel
; "Bitte wählen Sie ein Datum"
; "Please select a date"
; dsform.field.empty.FormFileUploadElementComponentModel
; "Bitte wählen Sie eine Datei"
; "Please choose a file to upload"
; dsform.field.empty.FormMultiSelectElementComponentModel
; "Bitte wählen Sie mindestens eine Option"
; "Please select at least one option"
; dsform.field.empty.FormRadioGroupElementComponentModel
; "Bitte wählen Sie eine Option"
; "Please select an option"
; dsform.field.empty.FormInputElementComponentModel
; "Bitte füllen Sie das Feld aus"
; "Please fill this field"
; dsform.field.empty.FormSelectElementComponentModel
; "Bitte wählen Sie eine Option"
; "Please select an option"
; dsform.field.empty.FormCheckboxElementComponentModel
; "Bitte wählen Sie diese Option"
; "Please select this option"
```

Anlage 3: ***essentialdata-ds-form-component-error-message-localization.impex***

```
public CatalogUnawareMediaModel createMediaFromUserUpload(
        MultipartFile uploadedFile) throws FileUploadException {
        try {
            final File tempFile = File.createTempFile("temp", "");
            uploadedFile.transferTo(tempFile);
            boolean validFile = getDsMediaService()
                .checkForValidFile(tempFile, uploadedFile);
            if (validFile) {
                final CatalogUnawareMediaModel media
                = getModelService().create(CatalogUnawareMediaModel.class);
                media.setCode("dsform_upload_"
                        + UUID.randomUUID().toString());
                media.setFolder(getDsMediaService()
                        .getFolder(DS_FORM_UPLOAD));
                getModelService().save(media);
                getDsMediaService().setStreamForMedia(media,
                        new FileInputStream(tempFile),
                        uploadedFile.getOriginalFilename(),
                        uploadedFile.getContentType());
                tempFile.delete();
                return media;
            } else {
                tempFile.delete();
                throw new FileUploadException();
            }
        }
        catch (IOException e) {
            LOG.error("Could not create media from user uploaded file ",e);
        }
        return null;
    }
```

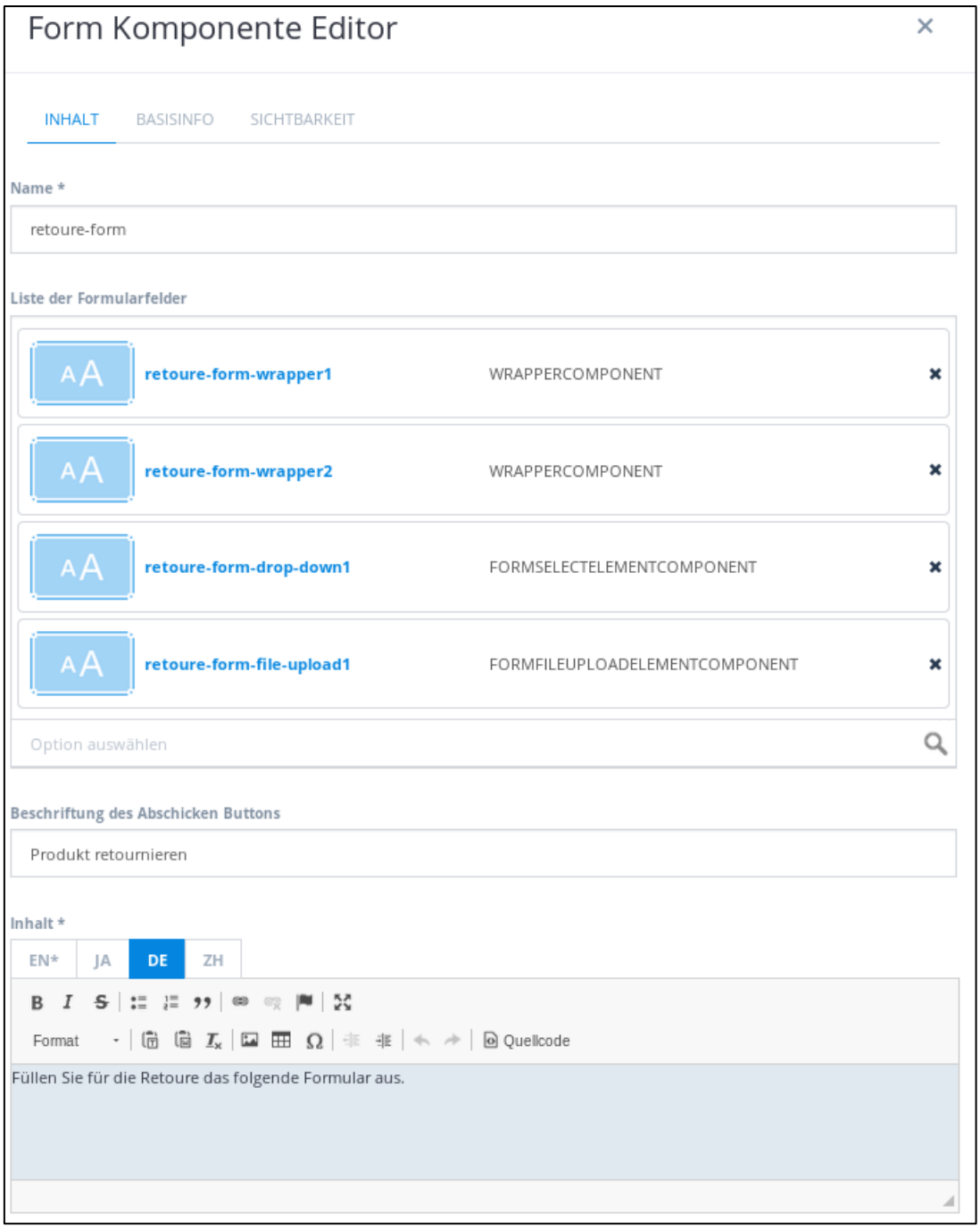
Anlage 4: Die Methode *createMediaFromUserUpload*

```
function _autoload(){
    $.each(ACC,function(section,obj){
        if($.isArray(obj._autoload)){
            $.each(obj._autoload,function(key,value){
                if($.isArray(value)){
                    if(value[1]){
                        ACC[section][value[0]]();
                    }else{
                        if(value[2]){
                            ACC[section][value[2]]()
                        }
                    }
                }else{
                    ACC[section][value]();
                }
            })
        }
    });

    $.each(DS,function(section,obj){
        if($.isArray(obj._autoload)){
            $.each(obj._autoload,function(key,value){
                if($.isArray(value)){
                    if(value[1]){
                        DS[section][value[0]]();
                    }else{
                        if(value[2]){
                            DS[section][value[2]]()
                        }
                    }
                }else{
                    DS[section][value]();
                }
            })
        }
    })
}

$(function(){
    _autoload();
});
```

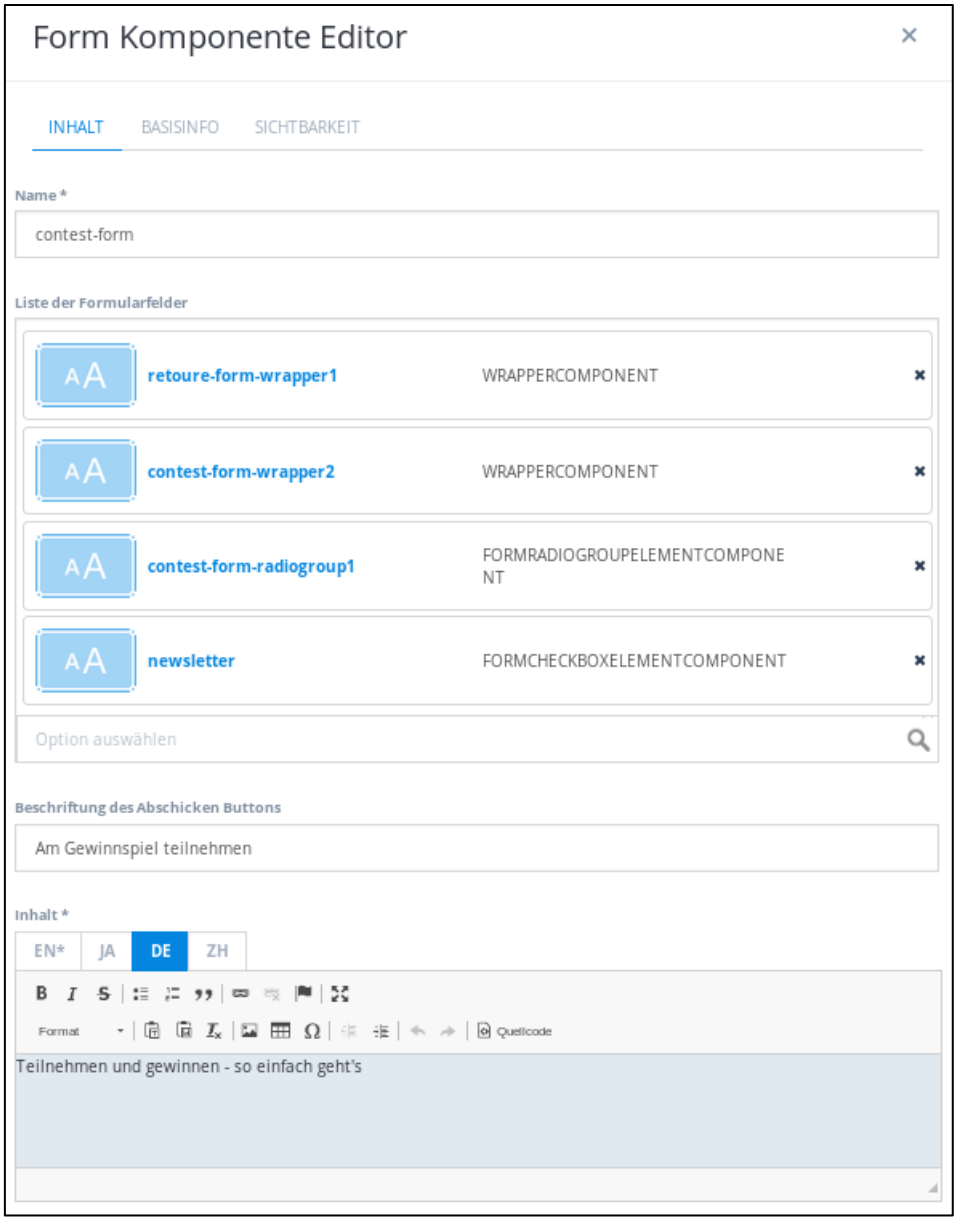
Anlage 5: *_autoload.js*

```java
package de.dotsource.dsform.storefront.controllers.misc;

// Fehlende Imports aufgrund besserer Übersichtlichkeit

@Controller
public class DefaultDsFormController extends DsFormController {

    @Override
    public String sendForm(final Model model,
        @RequestParam("componentUid") final String componentUid,
        DsFormComponentForm dsFormComponentForm,
        final BindingResult bindingResult) {
        try{
            // Hol dir die CMS-Komponente
            DsFormComponentModel component
               = (DsFormComponentModel) getCmsComponentService()
                    .getAbstractCMSComponent(componentUid);

            // Validiere das Formular (z.B. Regex bei Textinputs,
            // Mandatory Felder befüllt...)
            validateForm(component, dsFormComponentForm, bindingResult);

            // Check ob Felder Fehler bei Validierung aufweisen
            if(bindingResult.hasErrors()){
                // Behandlung von Validierungsfehlern
            }

            // DTO, welches die Felder und Werte des Formulars enthält
            // MUSS unbedingt aufgerufen werden, da in getFormDTO auch
            // Files angelegt und gespeichert werden
            // (im Falle von Fileupload)
            FormDTO formDTO = getFormDTO(component, dsFormComponentForm);

            // Formular leer hinzufügen, aber Auswahl beibehalten
            // (Checkboxen, Radiobuttons, Dropdowns)
            model.addAttribute("formComponentForm",
                DsFormUtil.emptyForm(component));
        } catch (CMSItemNotFoundException e) {
            e.printStackTrace();
        } catch (FileUploadException e) {
            e.printStackTrace();
        }

        // Eine View zurückgeben, welche sich in der Storefront unterhalb
        // des Verzeichnis WEB-INF/views/responsive/ befindet
        return "fragments/form/test";
    }
}
```

Anlage 6: Simple Beispielimplementierung des *DsFormController*



Anlage 7: Oberfläche des Retoure-Formulars im SmartEdit



Anlage 8: Oberfläche des Gewinnspiel-Formulars im SmartEdit

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Bachelorarbeit mit dem Thema:

   Integration einer dynamischen Formularkomponente in das Content Management System Smart-Edit

   ohne fremde Hilfe angefertigt habe,
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und
3. dass ich meine Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

_____ _____

Ort, Datum Unterschrift