Erweiterung eines dynamischen Formulargenerators um eine Bildupload-Funktion

Projektarbeit Nr.:	IV
vorgelegt am:	Donnerstag, 7. Februar 2019
von:	[redacted]
Matrikelnummer:	[redacted]
DHGE Campus:	Duale Hochschule Gera-Eisenach Weg der Freundschaft 4a 07546 Gera
Studienbereich:	[redacted]
Studiengang:	[redacted]
Kurs:	[redacted]
Ausbildungsstätte:	dotSource GmbH Goethestraße 1 07743 Jena
Betreuer Praxisbetrieb:	[redacted]
Gutachter Berufsakademie:	[redacted]

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
DTO	Data Transfer Object
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
MVC	Model-View-Controller Ein Model zur Aufteilung von Softwarekomponenten
W3C	World Wide Web Consortium

V Anlagenverzeichnis

1 Einleitung

Formulare finden sich mittlerweile bei vielen Online-Präsenzen. Ihr Anwendungszweck kann sehr verschieden ausfallen. Sie können für das Aktualisieren von Accountinformationen verwendet werden, eine Suchfunktion repräsentieren oder auch zur Kontaktaufnahme dienen. Letzteres findet in Online-Shops häufig Anwendung, um die Kundenbetreuung zu vereinfachen. Es können vordefinierte Formulare mit Eingaben des Nutzers gefüllt werden, wodurch dieser den Grund seiner Anfrage zielstrebiger darlegen kann. So kann beispielsweise an eine zuständige Kontaktperson vermittelt werden.

Beim Pflegen der Inhaltsseiten möchte der Shop-Besitzer selbst Formulare und deren Felder anlegen können. Eine dynamische Formularkomponente ermöglicht genau das. Über das entsprechende Content-Management-System können so entsprechende Formulare auf einfache Weise definiert werden.
In einigen Fällen kann es hilfreich sein, bei der Nutzung eines Online-Formulars eine Bilddatei mitzusenden. Beispielsweise um Produktmängel oder explizite Fragen zum Produkt zu kommunizieren. In einem Kundenprojekt der dotSource GmbH zur Erstellung eines Online-Shops wurde diese Funktion als Anforderung definiert. Mit der Umsetzung dieser beschäftigt sich die vorliegende Projektarbeit.

Das Erstellen von dynamischen Formularen mit verschiedenen Formularfeldern wie Texteingaben und Auswahllisten ist bereits möglich. Beim Absenden dieser wird eine E-Mail aus den Eingaben generiert und an eine voreingestellte E-Mail-Adresse gesendet. Es soll nun zusätzlich ermöglicht werden dem Formular ein Feld zum Mitsenden einer Bilddatei hinzuzufügen. Das Feld soll nicht auf ein Formular beschränkt sein. Daher wird eine Komponente entwickelt, die sich einem Formular optional hinzufügen lässt. Diese soll nur Bilddateien im Format PNG oder JPEG akzeptieren. Der Inhalt und die Vertrauenswürdigkeit der versendeten Bilder in der E-Mail können ohne großen Entwicklungsaufwand nicht sichergestellt werden. Daher

sollen diese nicht als Anhang versendet werden, sondern als Link in der E-Mail, welcher zum jeweiligen Download führt. Die Größe einer Bilddatei soll 5 MiB nicht überschreiten. Jegliches Risiko für das Produktivsystem des laufenden Online-Shops soll bei der Umsetzung möglichst gering ausfallen. Der primäre Fokus der Arbeit liegt auf der Funktionalität. Die visuelle Darstellung ist zweitrangig.

Der erste Teil der Arbeit befasst sich mit den Grundlagen, welche für das weitere Verständnis einiger Zusammenhänge wichtig sind. Es werden dabei die verwendeten Technologien sowie die Ausgangslage des Projekts beschrieben.

Der Hauptteil der Arbeit beschreibt die Implementierung. Erklärt wird, wie die Übermittlung des Formulars mit Dateien realisiert wird, die Verwertung der Anfrage im Controller sowie die Zusicherung des Bildformats und die Persistenz der Bilder. Danach wird die Fehlerbehandlung genauer betrachtet.

Abschließend wird ein Fazit gezogen, in welchem die entstandene Lösung mit den gestellten Anforderungen verglichen wird. Aufgetretene Probleme werden beschrieben und Möglichkeiten für eine Weiterführung der Arbeit diskutiert.

2 Grundlagen

2.1 Beschreibung des bestehenden Projekts

Die nachfolgend genannten Technologien werden im gesamten bestehenden Projekt verwendet. Für die Umsetzung der vorliegenden Arbeit werden diese daher an einigen Stellen zum Einsatz kommen. Es folgen dann jeweils ausführliche Erklärungen. Dabei liegt der Fokus auf hinzugefügter Funktionalität. Falls für das Verständnis nötig, wird es kurze Erklärungen zur bereits vorhandenen Implementierung geben.

Der bestehende Online-Shop verwendet als Shopsystem SAP-Hybris Commerce[1], fortlaufend Hybris genannt. Dieses integriert das verbreitete Spring-Framework[2]. Ein großer Vorteil, der daraus resultiert, ist die Dependency Injection. Sie ermöglicht es, dass die Abhängigkeiten einer Komponente in externen XML-Dateien konfiguriert werden können. Die Komponente muss diese somit nicht selbst verwalten. Durch diese Herangehensweise ist die Skalierbarkeit des Projekts einfacher.[3] Die Typendefinitionen in Hybris erfolgen ebenfalls in XML-Dateien. Das zugehörige Suffix dieser Dateien ist *-items.xml*.
Innerhalb des Projekts kommt das sogenannte MVC-Prinzip (Model-View-Controller) zum Einsatz. Es handelt sich dabei um ein Entwurfsmuster für Software. Im Kontext des bestehenden Webshops repräsentieren JSP-Dateien (JavaServer Pages) die View, also die Ansicht. Als Model kommen Java-Klassen mit Attributen und zugehörigen Getter- und Setter-Methoden zum Einsatz. Anfragen an die Nutzeroberfläche des Online-Shops (Storefront) werden von Spring-Controllern verarbeitet. Den Controller-Methoden der jeweiligen Klasse wird über die Annotation *@RequestMapping* mitgeteilt bei welcher Anfrage-URI sie auszuführen sind.

---

[1] https://www.sap.com/products/crm/e-commerce-platforms.html
[2] https://spring.io/
[3] [SAP19a]

2.2 CMSCockpit

Das CMSCockpit ist das Content-Management-System von Hybris. Demnach lässt sich mit dessen Hilfe der Inhalt der Storefront verwalten. Konkret wird es verwendet, um die Inhaltsseiten des Webshops zu bearbeiten. So können zum Beispiel neue Unterseiten angelegt oder Komponenten hinzugefügt und diese um Bilder oder Texte ergänzt werden. Im Kontext der Projektarbeit findet es Anwendung für das Anlegen der erwähnten Formulare.

Das CMSCockpit besitzt zwei Versionen für den Inhaltskatalog des Online-Shops. In der Staged-Version werden Änderungen am Inhalt vorgenommen. Die Online-Version ist der öffentliche Inhalt in der Storefront. Bei der Synchronisation werden die Änderungen der Staged-Version in die Online-Version übernommen.

2.3 Bestehender Formulargenerator

Um ein neues Formular anzulegen wird im CMSCockpit einer Inhaltsseite des Shops eine generische Formularkomponente zugewiesen. Deren wichtigste Attribute sind in Tabelle 1 dargestellt.

Attributname	**Bedeutung**
E-Mail-Empfänger	E-Mail-Adresse, an die das Formular versendet wird
E-Mail-Betreff	Betreff der versendeten E-Mail
Liste der Formularfelder	Liste mit Komponenten, welche die Formularfelder repräsentieren
Beschriftung des Abschicken-Buttons	Text, welcher auf dem Button zum Senden des Formulars erscheint

Tabelle 1: Attribute der Formularkomponente

Der Liste der Formularfelder können Komponenten hinzugefügt werden, welche die einzelnen Eingabefelder des Formulars repräsentieren. Bisher sind folgende Elemente möglich:

- Texteingabe
- Checkbox
- Listenauswahl
- Radio-Button

Das Formular folgt somit keinem festen Format und kann nach Belieben angepasst werden. Dadurch kann die Erstellung eigener Formulare durch den Anwender übernommen werden. Ein Beispielformular findet sich in Abbildung 1.

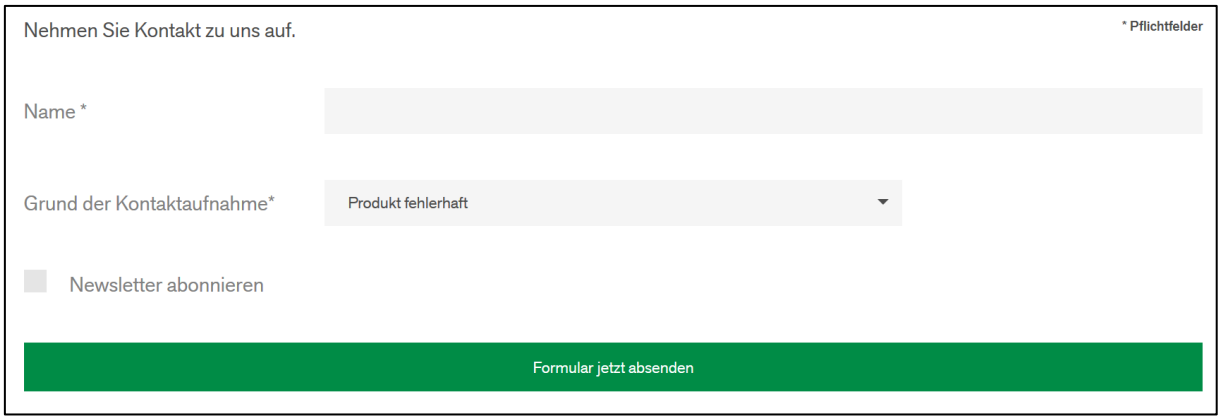


Abbildung 1: Generiertes Demo-Formular

Im CMSCockpit können mit Hilfe einer grafischen Oberfläche die einzelnen Felder angeordnet werden. Für den Upload einer Bilddatei muss es möglich sein der Liste eine entsprechende Komponente hinzuzufügen.

3 Erstellung der Bilduplоad-Komponente

3.1 Erweiterung des Formulargenerators um eine Komponente

Zuerst muss es möglich sein, die Bilduplоad-Komponente über das CMSCockpit einem Formular zuzuweisen. Dafür ist es zunächst nicht notwendig, dass diese funktionsfähig ist. Es wird eine Klasse *FormImageUploadElementComponent* mit der Superklasse *FormElementComponent* angelegt. Dies geschieht über das Hinzufügen eines neuen Typs in der Datei *core-items.xml* wie in Abbildung 2. Direkt vererbt werden die Attribute aus Tabelle 2.

```
<itemtype code="FormImageUploadElementComponent"
        jaloclass="…jalo.components.FormImageUploadElementComponent"
        extends="FormElementComponent" >
        <description>Component for the upload of images</description>
</itemtype>
```

Abbildung 2: Klassendefnition *FormImageUploadElementComponent* **in** *core-items.xml*

Die Angabe einer *„jaloclass“* sorgt dafür, dass Hybris eine weitere (abstrakte) Superklasse anlegt, in welcher Getter- und Setter-Methoden implementiert sind.[4] Die Klasse für die Businesslogik bleibt somit übersichtlicher.

Attributname	**Beschreibung**
mandatory	Boolean - gibt an, ob das Feld ein Pflichtfeld ist
Label	String - Beschriftung des Formularfeldes in der Storefront
formElementName	String – Beschriftung des Formularfeldes in der versendeten E-Mail

Tabelle 2: Übersicht über die Attribute der Klasse *FormElementComponent*

[4] [SAP19b]

Über XML-Dateien werden die Ansichten des CMSCockpit für das Bearbeiten und Anlegen einer Komponente beschrieben. Die entsprechenden Dateien in diesem Fall sind *editorArea_FormImageUploadElementComponent.xml* sowie *wizardConfig_FormImageUploadElementComponent.xml*. Der Inhalt ähnelt der Konfiguration bestehender Formularfeld-Komponenten und unterscheidet sich nur bei den Attributnamen. Da die Bildupload-Komponente nur die Attribute der Superklasse besitzt, ist der genaue Inhalt für das weitere Verständnis nicht relevant. Beide Ansichten bieten lediglich die Möglichkeit die Attribute aus Tabelle 2 zu setzen.
Die Komponente muss beim Synchronisieren im CMSCockpit berücksichtigt werden. Die zuständige Datei heißt *cmscockpit-services.xml.* An dieser wird die Anpassung wie in Abbildung 3 vorgenommen. Bei der nächsten Synchronisation wird die erstellte Komponente übernommen.

```
<bean id="defaultCMSSynchronizationService"
class="de.hybris.platform.cmscockpit.sync.CMSSynchronizationService"
autowire="byName">
<property name="relatedReferencesTypesMap">
        <map>
                <entry key="CMSItem">
                        <list>
                        <!-- Other things to be synchronized -->
                        <value>FormRadioElementComponent</value>
                        <value>FormSelectElementComponent</value>

                        <!-- Hinzufügen der Bildupload-Komponente -->
                        <value>FormImageUploadElementComponent</value>

                        <!-- Other things to be synchronized -->
                        </list>
                </entry>
        </map>
</property>
```

Abbildung 3: Hinzufügen der Bildupload-Komponente zum Synchronisationsservice

Die beiden XML-Dateien des Formulars *editorArea_FormComponent.xml* sowie *wizardConfig_FormComponent.xml* werden um einen Eintrag ergänzt, wodurch die Bildupload-Komponente der Liste der Formularfelder hinzugefügt werden darf. Dazu wird der Name des angelegten Typen der bestehenden Liste erlaubter Komponenten

hinzugefügt. Dies ermöglicht bei Hinzufügen eines Formularfeldes im CMSCockpit die Auswahl der Bildupload-Komponente. Zu sehen ist die Anpassung in Abbildung 4.

```
<property qualifier="BayWaFormComponent.componentList">
        <parameter>
                <name>restrictToCreateTypes</name>
                <value>
                        FormInputElementComponent,
                        FormCheckboxElementComponent,
                        FormRadioGroupElementComponent,
                        FormSelectElementComponent,
                        WrapperComponent,
                        <!— Hinzufügen der Bildupload-Komponente -->
                        FormImageUploadElementComponent
                </value>
        </parameter>
</property>
```

Abbildung 4: Ergänzung der Bildupload-Komponente zu den akzeptierten Formularfeldern

3.2 Darstellung der Komponente in der Storefront

Damit die Komponente in der Storefront angezeigt werden kann, wird über eine JSP-Datei ihre HTML-Darstellung definiert. Deren Inhalt ist in Abbildung 5 zu sehen.

```
<%@ page trimDirectiveWhitespaces="true"%>
<%@ taglib prefix="formElement"
        tagdir="/WEB-INF/tags/responsive/formElement"%>

<div class="form-input-element-component row form-group">
    <div class="${not empty component.label ? 'col-md-9' : 'col-tn-12'}">
        <formElement:formFileSelectBox
                idKey="input-${component.formElementName}"
                path="fields[${component.formElementName}]"
                labelKey="${component.label}"
                mandatory="${component.mandatory}"/>
    </div>
</div>
```

Abbildung 5: Inhalt der Datei *formimageuploadelementcomponent.jsp*

Eine Tag-Library erweitert JSP-Dateien um benutzerdefinierte Tags und Anweisungen und somit Java-Code aus diesen auszulagern.[5] Mit Hilfe der projektinternen Tag-Library *formElement* wird ein, bereits bestehendes, HTML-Eingabefeld für die Dateiauswahl eingebunden. Diesem werden die Attribute der Komponente übergeben, damit jene im HTML gesetzt werden können. Sie finden auch Anwendung für spezielle Fälle der Fehlerbehandlung. Kapitel 4.6 befasst sich ausführlicher mit diesem Thema. In der JSP-Datei des bestehenden Formulars werden alle Formularfelder iterativ über ihre eigenen JSP-Dateien zur Anzeige gebracht.

[5] [Ora19]

4 Implementierung der Funktionalität

Grundlegend lässt sich der Prozess beim Versenden des Formulars auf wenige Teilschritte abstrahieren. Beim Absenden gelangt das Formular per Ajax-Anfrage in den Controller. Dort findet eine Überprüfung der Datei auf ein gültiges Bildformat statt. Bei Gültigkeit wird diese gespeichert und anschließend als Downloadlink zugänglich gemacht. Abbildung 6 stellt den Prozess bei Erfolg grafisch vereinfacht dar.

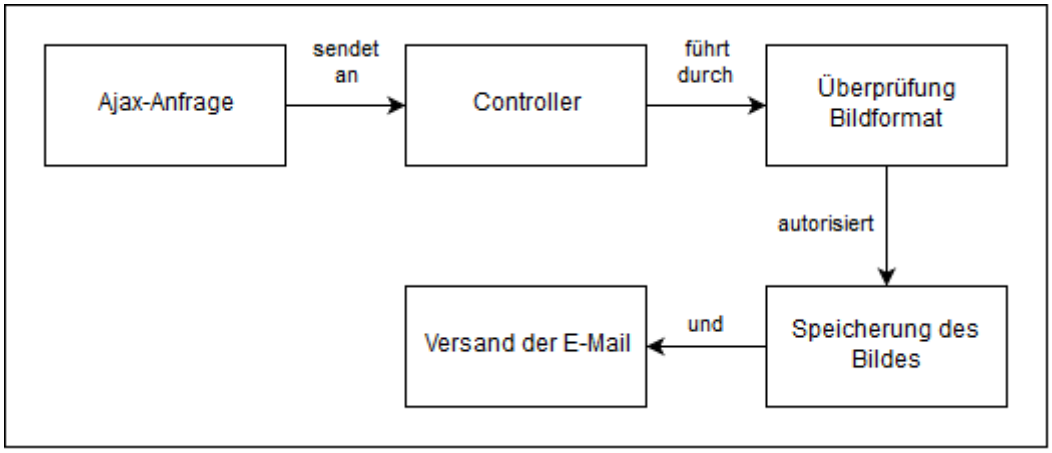


Abbildung 6: Vereinfachte Prozessübersicht bei erfolgreichem Absenden des Formulars

4.1 Anpassung der Formularübermittlung

Das World Wide Web Consortium (W3C) schreibt über das Verhalten eines HTML-Formulars beim Absenden: „When a form is submitted, the data in the form is converted into the structure specified by the enctype, and then sent to the destination specified by the `action` using the given method."[6] Demnach gibt das enctype-Attribut an, in welcher Form die Formulardaten versendet werden. Die möglichen Werte sind[7]:

- text/plain
- application/x-www-form-urlencoded
- multipart/form-data

[6] [W3C17a]
[7] [W3C17b]

Die erste Möglichkeit „text/plain“ lässt sich aufgrund des, für Menschen lesbaren, Formats schwer von Computern interpretieren.[8] Da allerdings eine Weiterverarbeitung der übertragenen Dateien stattfinden soll, entfällt dieser Wert. Wird der Wert auf „application/x-www-form-urlencoded“ gesetzt, ist eine Übertragung von Dateien zwar möglich, allerdings sehr ineffizient.[9] Das W3C empfiehlt explizit die Verwendung von „multipart/form-data“ bei der Übertragung von Dateien über ein Formular.[10] Diese Arbeit richtet sich nach dieser Empfehlung. Das ermöglicht außerdem, die Anfragen durch den MultipartResolver von Spring auflösen zu lassen. Dieser kümmert sich dann um den Zugriff auf eventuell gesendete Dateien.[11]

```
<%@ 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/form/send" />
<spring:message var="errorMessage" code="contact.form.invalid" />
<spring:message var="uploadSizeExceededErrorMessage"
        code="text.form.send.error.uploadSizeExceeded" />

<form:form id="formComponent-${component.uid}"
        action="${encodedUrl}"
        method="POST"
        enctype="multipart/form-data"
        data-role="multipartAjaxForm"
        data-errorMessage="${errorMessage}"
        data-uploadSizeExceededErrorMessage
                ="${uploadSizeExceededErrorMessage}"
        commandName="formComponentForm"
        novalidate="novalidate">
<!-- … Formularfelder, Absenden-Button … -->
</form:form>
```

Abbildung 7: Definition des Formulars in *formComponentForm.tag*

In der bestehenden Datei *formComponentForm.tag* (zu sehen in Abbildung 7) wird das Attribut „enctype“ des Formulars mit dem Wert „multipart/form-data“ ergänzt. Das Attribut „data-role“ wird von „ajaxForm“ auf den Wert „multipartAjaxForm“ geändert um das nachfolgend erläuterte JavaScript anzusteuern. Für das bessere Verständnis wird

[8] [W3C17c]
[9] [W3C19]
[10] [W3C19]
[11] [Spr19a]

an dieser Stelle dem Kapitel 4.6 bezüglich der Behandlung von Fehlern vorgegriffen: Dem Attribut „data-uploadSizeExceededErrorMessage“ wird ein Wert entsprechend des Lokalisierungscodes zugewiesen, der später für die Anzeige einer Fehlermeldung verwendet wird. Die Lokalisierungen sind in Kapitel 4.6 ausführlicher beschrieben.

In der Datei *ds.ajax.js* ist das Verhalten des Formulars beim Absenden definiert. Dazu wird die Funktion „initMultiPartAjaxForm“ implementiert. Diese wird über Aufrufe in anderen JavaScript-Dateien bei jedem Seitenaufruf ausgeführt. Die Implementierung ist, mit leichten Anpassungen zum besseren Verständnis, dargestellt in Abbildung 8.

```
function initMultipartAjaxForm() {
    $(document).on('submit', '[data-role="multipartAjaxForm"]',
    function(event) {
        event.preventDefault();
        var form = $(event.target);
        var data = new FormData(form[0]);
        var uploadSizeExceededErrorMessage =
            form.attr('data-uploadSizeExceededErrorMessage');

        $.ajax({
            method : form.attr('method'),
            url : form.attr('action'),
            dataType : 'json',
            data: data,
            processData: false,
            contentType: false,
        }).done(function (response) {
            if(response.message) {
                DS.Utils.showAlert(response.message, 'success');
            }
            DS.Ajax.formCallback(form, response);
        }).fail(function(xht, textStatus, ex) {
            if(xht.responseText) {
                DS.Utils.showAlert(JSON.parse(xht.responseText), 'danger');
            }
            else if((xht.status == '413' || xht.status == '0')
                && uploadSizeExceededErrorMessage){
                DS.Utils.showAlert(uploadSizeExceededErrorMessage,'danger');
            }
            DS.Ajax.formError(form, xht, textStatus, ex);
        });
    });
}
```

Abbildung 8: Implementierung der Funktion „initMultiPartForm“

Die Funktion weist dem submit-Event eines Elements mit dem Wert „multipartAjaxForm“ für data-role eine Funktion zu, welche das Standardverhalten, also das Abschicken des Formulars, verhindert und die Quelle des Events speichert. Mit der Hilfe der JavaScript Bibliothek JQuery[12], wird anschließend eine Ajax-Anfrage erstellt, welcher das erwähnte Standardverhalten ersetzt. Aus den Attributen, welche in *formComponentForm.tag* gesetzt sind, wird „method“ der Wert „Post“ und „url“ der Wert „/xhr/form/send“ zugeordnet. Beim Absenden des Formulars gelangt die Anfrage daraufhin an die richtige Controller-Methode. Dies ist in Kapitel 4.2 genauer beschrieben.

Weiterhin werden die Daten des Formulars als „data“ der Anfrage gesetzt. Die Einstellungen „processData“ sowie „contentType“ erhalten beide den Wert „false“. Andernfalls würden die Daten nach dem Content-Type „application/x-www-form-urlencoded“ transformiert werden.[13] Wie bereits erwähnt ist das explizit nicht gewollt, um eine performante und problemlose Weiterverarbeitung zu garantieren.

Über „dataType“ wird das Format der erhaltenen Antwort festgelegt – in diesem Fall JSON. Bei erfolgreicher Anfrage wird der done-Zweig der Ajax-Anfrage ausgeführt. Dieser gibt eine Erfolgsmeldung in der Storefront aus oder weist auf Eingabefehler hin. Bei einem Fehlschlag stattdessen, wird im zugehörigen fail-Zweig eine Fehlermeldung ausgegeben. Für den Fall, dass der Statuscode 413 oder 0 ist, wird der Wert von „uploadSizeExceededErrorMessage“ verwendet. Der Hintergrund dieses Spezialfalls wird im Kapitel 5.2 erklärt.

4.2 Anpassung der Controllerlogik

Der Einstieg für die Businesslogik nach dem Absenden eines Formulars findet sich im bereits vorhandenen Controller mit dem Namen *FormController.* Dort werden mit der Spring-Annotation @Resource die benötigten Abhängigkeiten bekannt gemacht (siehe Abbildung 9).

---

[12] https://jquery.com/
[13] [Jqu19]

```
@Resource(name = "cmsComponentService")
private CMSComponentService cmsComponentService;

@Resource(name = "mediaFacade")
private mediaFacade mediaFacade;

@Resource(name = "baseSiteService")
private BaseSiteService baseSiteService;

@Resource(name = "siteBaseUrlResolutionService")
private SiteBaseUrlResolutionService siteBaseUrlResolutionService;
```

Abbildung 9: Abhängigkeiten des *FormController*

Über die URL der Anfrage wird die auszuführende Controller-Methode identifiziert. Diese muss hierzu entsprechend gekennzeichnet sein. Das geschieht mit der der Spring-Annotation @RequestMapping. Der value-Parameter gibt die Anfrage-URL an. Sie stimmt mit dem Wert der Variable „encodedUrl“ aus der Datei *formComponentForm.tag* überein – also „/xhr/form/send“. Für den Parameter „method“ wird ebenso das Äquivalent aus der genannten Datei übernommen und somit die HTTP-Methode auf „POST“ gesetzt. Zusätzlich wird über den Parameter „produces“ angezeigt, dass zu den beiden Kriterien nur Anfragen entgegengenommen werden, welche im Accept-Header JSON als Format auflisten. Die Methodensignatur von „sendForm“ sieht daher aus wie in Abbildung 10.

```
@RequestMapping(value = "/xhr/form/send", method = RequestMethod.POST,
                produces = MediaType.APPLICATION_JSON_VALUE)
public String sendForm(final Model model,
                @RequestParam("componentUid") final String componentUid,
                FormComponentForm formComponentForm,
                final BindingResult bindingResult)
```

Abbildung 10: Methodensignatur von sendForm **und Mapping der Anfrage**

Bei einer HTTP-Anfrage an „/xhr/form/send“ wird die „componentUID“ als Parameter gesendet, um die Ursprungkomponente der Anfrage eindeutig zu identifizieren. Mit der Hilfe von „bindingResult“ werden die Ergebnisse der Validierung der Formularfelder

festgehalten. Es gibt Auskunft über mögliche aufgetretene Fehler. Der Parameter „formComponentForm“ hält das gesendete Formular und den Inhalt aller Felder.

```
@RequestMapping(value = "/xhr/form/send", method = RequestMethod.POST,
                produces = MediaType.APPLICATION_JSON_VALUE)
public String sendForm(final Model model,
                @RequestParam("componentUid") final String componentUid,
                FormComponentForm formComponentForm,
                final BindingResult bindingResult)
{
    try
    {
        FormComponentModel component =
        (FormComponentModel) getCmsComponentService()
            .getAbstractCMSComponent(componentUid);
        model.addAttribute(ATTRIBUTE_COMPONENT, component);

        validateForm(component, formComponentForm, bindingResult);
        if (bindingResult.hasErrors())
        {
           model.addAttribute(ATTRIBUTE_HAS_VALIDATION_ERROR,
               Boolean.TRUE);
           return ControllerConstants.Views.Fragments.Form.FormResult;
        }
        getFormService().sendForm(getFormDTO(component,formComponentForm));

        model.addAttribute(ATTRIBUTE_HAS_ERROR, Boolean.FALSE);
        model.addAttribute(ATTRIBUTE_COMPONENT_FORM,
            FormUtil.emptyForm(component));

    }
    catch (CMSItemNotFoundException e)
    {
        model.addAttribute(ATTRIBUTE_HAS_ERROR, Boolean.TRUE);
    }
    catch (FileUploadException e)
    {
        model.addAttribute(ATTRIBUTE_HAS_INVALID_IMAGE_ERROR,
            Boolean.TRUE);
    }
    return ControllerConstants.Views.Fragments.Form.FormResult;
}
```

Abbildung 11: Implementierung der Methode *sendForm*

Die ursprüngliche Implementierung der Controller-Methode wurde im Rahmen dieser Arbeit kaum geändert. Die einzige Anpassung ist ein ergänzter catch-Zweig für die spätere Fehlerbehandlung im Kapitel 5. Zum Verständnis ist in Abbildung 11 die

gesamte Implementierung der Methode „sendForm“ dargestellt. Zusammengefasst validiert diese die Eingaben des Nutzers und versendet bei Korrektheit eine E-Mail mit den Werten der Felder.

```
private FormDTO getFormDTO(final FormComponentModel component,
    final FormComponentForm form) throws FileUploadException {
        final Map<String, String> fieldValues = new HashMap<>();
        final Map<String, String> imageURLs = new HashMap<>();
        for(Map.Entry entry : form.getFields().entrySet())
        {
            if(!fieldEmpty(entry.getValue()))
            {
                if(entry.getValue() instanceof String)
                {
                    fieldValues.put(entry.getKey().toString(),
                        entry.getValue().toString());
                }
                else if(entry.getValue() instanceof MultipartFile)
                {
                    CatalogUnawareMediaModel media =
                    mediaFacade.createMediaFromUserUpload(
                        (MultipartFile) entry.getValue());
                    if(media != null)
                    {
                        String imageURL =
                        siteBaseUrlResolutionService.getMediaUrlForSite(
                            baseSiteService.getCurrentBaseSite(),
                            true, media.getDownloadURL());
                        fieldValues.put(entry.getKey().toString(),
                            media.getRealFileName());
                        imageURLs.put(entry.getKey().toString(), imageURL);
                    }
                }
            }
        }

    return new FormDTO.Builder()
        .recipient(component.getEmailRecipient())
        .subject(component.getEmailSubject())
        .values(fieldValues)
        .imageURLs(imageURLs)
        .build();
}
```

Abbildung 12: Implementierung der Methode *getFormDTO*

Neu implementiert wird stattdessen die Hilfsmethode *getFormDTO* (zu sehen in Abbildung 12). Sie liest die gesamten Eingaben der Formularfelder ein und erstellt

daraus ein Data Transfer Object (DTO). Dabei handelt es sich um die Instanz einer Klasse, welche die weiter benötigten Felder und zugehörige Getter- und Setter-Methoden besitzt. Das Übergeben dieses Objekts an andere Methoden sichert dieser somit den weiteren Zugriff auf all diese Felder zu.

Die zum Schluss der HTTP-Anfrage versendeten E-Mails sollen die Downloadlinks zu den hochgeladenen Bildern enthalten. Bei der bisherigen Implementierung der Methode *getFormDTO* konnten einfach die Werte der Felder ausgelesen und übernommen werden. Das ist mit einer Bilddatei nicht möglich, da diese nicht in Form eines Strings übertragen wird. Der Link zum Herunterladen der Datei wird des Weiteren erst generiert, wodurch dieser ebenso nicht aus dem Formular gelesen werden kann.

Da das Formular nun eine Bildupload-Komponente enthalten kann, muss dieser Fall behandelt werden. Mit Hilfe einer for-Schleife wird über die Formularfelder iteriert. Eine Abfrage sorgt dafür, dass nur ausgefüllte Formularfelder der E-Mail hinzugefügt werden. Sollte es sich bei dem Wert des Felds um einen String handeln, wird wie bisher der Wert dem zugehörigen Feldnamen zugeteilt. Für den Fall, dass der Wert den Typ MultipartFile[14] besitzt, wird das hochgeladene Bild gespeichert. Kapitel 4.4 befasst sich hiermit noch ausführlicher. Dem Feldnamen wird stattdessen als Wert der Name der Datei zugeordnet.
In einer zweiten Map „imageURLs“ wird außerdem die generierte URL, unter welcher die Datei abrufbar ist, dem Feld zugewiesen. Anschließend wird ein FormDTO aus den Informationen des Formulars von der Methode zurückgegeben. In der Controller-Methode *sendForm* wird dieses DTO an die FormService-Methode *sendForm* als Parameter übergeben. Sie kümmert sich um das Versenden des DTO in einer E-Mail.

[14] Klasse aus dem Springframework, repräsentiert eine formularübertragene Datei

4.3 Verifizierung von Bilddateien

Die bisher beschriebene Implementierung lässt über die Bildupload-Komponente das Hochladen von Dateien jedes Formats zu. Somit könnten auch Dateien mit schädlichem Inhalt hochgeladen werden. Da die Anforderungen das Format eines Bildes eindeutig auf PNG und JPEG beschränken und ein geringes Risiko vorschreiben, muss eine Lösung entstehen, welche diese Einschränkungen realisiert. In der Klasse *DefaultMediaService* entsteht für diesen Zweck die Methode *checkForValidImage**,*** welche als Parameter eine Datei entgegennimmt und als Rückgabe einen Boolean-Wert liefert. Diese Methode soll zusichern, dass es sich bei der Datei um ein Bild in einem der unterstützten Formate handelt.

Viele Implementierungen zur Erkennung des Dateiformats bestimmen dieses anhand der Dateiendung. Das stellt keine zufriedenstellende Lösung dar, da eine mutwillige Manipulation sehr einfach möglich ist und beispielsweise bei einer EXE-Datei ein „.png“ am Ende des Dateinamens ergänzt werden kann. Die finale Umsetzung orientiert sich an einer Implementierung von www.owasp.org.15 Diese befasst sich auch mit dem Überschreiben der Bilder, um möglichen Schadcode zu entfernen. Die Erkennung des Bildformats ist für diese Projektarbeit ausreichend. Das ist mit der Standardschnittstelle von Java für die beiden unterstützen Formate PNG sowie JPEG möglich.[16] Die angepasste Implementierung für die Methode *checkForValidImage* ist in Abbildung 13 dargestellt.

[15] [Rig18]
[16] [Ora18]

```
public boolean checkForValidImage(File f) {
    try {
        if ((f != null) && f.exists() && f.canRead()) {
            String formatName;
            try (ImageInputStream iis = ImageIO.createImageInputStream(f))
            {
                Iterator<ImageReader> imageReaderIterator =
                    ImageIO.getImageReaders(iis);
                if(!imageReaderIterator.hasNext()) {
                    return false;
                }else {
                    ImageReader reader = imageReaderIterator.next();
                    formatName = reader.getFormatName();
                    if(SUPPORTED_IMAGE_FORMATS.contains(formatName))
                    {
                        return true;
                    }
                }
            }
        }

        return false;
    } catch (Exception e) {
        return false;
    }
}
```

Abbildung 13: Implementierung der Methode checkForValidImage

In der ersten if-Abfrage wird geprüft ob auf die Datei problemlos ein Lesezugriff erfolgen kann. Nachfolgend wird geprüft, ob ein ImageReader vorhandenen ist. Sollte das nicht der Fall sein, bedeutet das, dass die Java Schnittstelle das Format der Datei nicht unterstützt. Im Kontext der Prüfung handelt es sich somit um ein invalides Dateiformat. Die Methode gibt „false“ zurück.

Im Fall, dass der Zugriff mit einem ImageReader auf die Datei möglich ist, wird diesem das Dateiformat entnommen und mit einer Liste der unterstützten Bildformate – also PNG und JPEG – verglichen. Sollte das Format in der Liste vorhanden sein, wird der Wert „true“ zurückgegeben. Andernfalls wird ebenfalls „false“ zurückgegeben. Um festzustellen, ob ein Bild in einem der unterstützten Formate hochgeladen wurde, ist dieses Vorgehen hinreichend.

4.4 Persistenz der Bilddateien

Damit die versendeten Bilder des Formulars in der E-Mail als Downloadlink verfügbar sind, müssen sie serverseitig gespeichert werden. Bei der Verwendung von Hybris werden diese vom Typ Media repräsentiert. Ein Media kann eine Datei jedes Formats sein. Wie bereits in Kapitel 4.2 erwähnt, findet das Persistieren der Bilddatei im Controller beim Aufruf von *getFormDTO* statt. Explizit übernimmt diese Aufgabe die Methode *createMediaFromUserUpload,* welche in der Klasse *DefaultMediaFacade* implementiert und in Abbildung 14 veranschaulicht ist.

```
public CatalogUnawareMediaModel createMediaFromUserUpload(
    MultipartFile uploadedFile) throws FileUploadException {
        try
        {
            final File tempFile = File.createTempFile("temp", "");
            uploadedFile.transferTo(tempFile);
            boolean validImage = mediaService.checkForValidImage(tempFile);
            if(validImage)
            {
                final CatalogUnawareMediaModel media =
                    modelService.create(CatalogUnawareMediaModel.class);
                media.setCode("image_upload_" +
                    UUID.randomUUID().toString());
                media.setFolder(getMediaService().getFolder(
                    CoreConstants.Media.Folder.IMAGES));
                getModelService().save(media);
                mediaService.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;
}
```

Abbildung 14: Implementierung der Methode *createMediaFromUserUpload*

Zuerst wird eine temporäre Datei vom Typ File angelegt. Über die Methode *transferTo* wird der Inhalt des MultipartFile in die angelegte Datei übertragen. Das ist notwendig, da die Methode *checkForValidImage* einen Parameter vom Typ File erwartet. In einem Boolean-Wert wird entsprechend Kapitel 4.3 das Ergebnis der Prüfung auf ein valides Format gespeichert. Sollte „validImage“ den Wert „false“ haben, so wird die temporäre Datei gelöscht und eine FileUploadException geworfen. Im Fall einer zulässigen Datei wird mit Hilfe des modelService eine neue Instanz von CatalogUnawareMediaModel erzeugt. Das Attribut „code“ des Models wird mit dem Prefix „image_upload_“ und einer zufälligen Zeichenkette gesetzt. Der Ordner, welchem die Bilder gespeichert werden, erhält als Wert eine Konstante. Mit dem Aufruf der Methode *save* wird das Model in der Datenbank des Hybris-Systems gespeichert. Der Methodenaufruf *setStreamForMedia* von mediaService persistiert die Daten der temporären Datei in dem MediaModel. Hier hätte theoretisch direkt der InputStream des MultiPartFile verwendet werden können, allerdings ist dieser nach dem Aufruf von *transferTo* nicht mehr zugänglich.[17] Dieser ist jedoch für die Prüfung der Datei unumgänglich, weshalb der InputStream der temporären Datei verwendet wird. Zuletzt wird die temporäre Datei gelöscht.

4.5 Darstellung des Bildes in der E-Mail

Damit die Verlinkungen zu den versendeten Bildern in der E-Mail angezeigt werden können, muss die Map „imageURLs“ im E-Mail Template bekannt sein. In Abbildung 15 ist die Anpassung der entsprechenden Klasse FormContext zu sehen. Getter- und Setter-Methoden anderer Attribute werden aus Übersichtlichkeit nicht gezeigt.

[17] [HC03a]

```
public class FormContext extends CustomerEmailContext {
    private String subject;
    private Map<String, String> fieldValues;
    private Map<String, String> imageURLs; // ergänzt

    public void init(final StoreFrontCustomerProcessModel,
        storeFrontCustomerProcessModel,
        final EmailPageModel emailPageModel) {
        super.init(storeFrontCustomerProcessModel, emailPageModel);
        if (storeFrontCustomerProcessModel instanceof FormProcessModel)
        {
            FormProcessModel formProcessModel = (FormProcessModel)
                storeFrontCustomerProcessModel;

            put(EMAIL, formProcessModel.getRecipient());
            setSubject(formProcessModel.getSubject());
            setFieldValues(formProcessModel.getFieldValues());
            setImageURLs(formProcessModel.getImageURLs()); // ergänzt
        }
    }

    // ergänzt
    public Map<String, String> getImageURLs() { return imageURLs; }

    // ergänzt
    public void setImageURLs(Map<String, String> imageURLs) {
        this.imageURLs = imageURLs;
    }

}
```

Abbildung 15: Ergänzungen in der Klasse FormContext

Dem Kontext des E-Mail-Templates wird über die set-Methode ein Wert für „imageURLs“ gesetzt. Dieser wird über das FormProcessModel beschafft, welches zu einem früheren Zeitpunkt mit Hilfe des FormDTO initialisiert wird. Aufgrund der Anpassung in FormContext kann das E-Mail Template *email-FormBody.vm* auf die Map und somit auch auf die Bild-URLs zugreifen. Dabei handelt es sich um ein Velocity-Template, welches über die eigene Velocity Template Language Java-Methoden und -Objekte referenzieren kann.[18] In Abbildung 16 ist der Teil des Templates zu sehen, in welchem die einzelnen Werte der Formularfelder und gegebenenfalls Bild-URLs dargestellt werden.

[18] [Apa19]

```
#foreach($fieldName in $ctx.fieldValues.keySet())
    $fieldName:
    #if($ctx.imageURLs.keySet().contains($fieldName))
        <a href="$ctx.imageURLs[$fieldName]">
            $ctx.fieldValues[$fieldName]
        </a>
        <span style="color:red; font-size:13px;">
            ($ctx.messages.imageLinkWarning)
        </span></br>
    #else
        $ctx.fieldValues[$fieldName] </br>
    #end
#end
```

Abbildung 16: Darstellung der Formularfelder in email-FormBody.vm

Mit einer for-Schleife wird über die Schlüssel der Map „fieldValues“ iteriert und diese zur Anzeige gebracht. Sollte dieser Schlüssel auch in der Map „imageURLs“ vorkommen, so wird ein Link-Element eingefügt. Dessen href-Attribut hat den zugehörigen Wert aus „imageURLs“ und wird mit dem Wert aus „fieldValues“ angezeigt. Hinter dem Link wird in roter Schrift ein lokalisierter Warnhinweis ausgegeben, der auf den fehlenden Virenscan hinweist. In allen anderen Fällen wird ausschließlich der Wert des Schlüssels ausgelesen.

4.6 Hinzufügen der Lokalisierungen

Lokalisierungen werden benötigt, um die verwendeten Anzeigetexte unabhängig von der Implementierung pflegen zu können. Anpassungen müssen somit nicht im Code selbst vorgenommen werden. Dadurch kann der Shopbesitzer die Texte eigenständig aktualisieren, welche sich je nach Anwendungsfall in einer anderen Datei befinden. Lokalisierungsdateien besitzen die Dateiendung *„properties“*. Im Rahmen dieser Projektarbeit war es notwendig Anpassungen an den folgenden drei Dateien vorzunehmen:

- core-locales_de.properties
- email-Form_de.properties
- base_de.properties

Es ist jeweils ein Äquivalent für die englischen Texte vorhanden. Die Dateien bestehen aus Schlüssel-Wert-Paaren. Jedem Lokalisierungsschlüssel wird ein Wert zugeordnet. Die Schlüssel erhalten Namen, die auf ihre Verwendung schließen lassen. Die hinzugefügten Inhalte sind in den Abbildungen Abbildung 17, Abbildung 18 und Abbildung 19 zu sehen. Tabelle 3 zeigt die Maskierung verschiedener Unicode-Zeichen.

```
type.FormImageUploadElementComponent.name = Bild Upload
type.FormImageUploadElementComponent.description
= Mit dieser Komponente k\u00F6nnen Sie ein Bildupload Feld in Ihr Formular laden.
```

Abbildung 17: Lokalisierungen in *core-locales_de.properties*

```
imageLinkWarning = Achtung: Es kann nicht garantiert werden, dass es sich bei der Datei
tats\u00E4chlich um ein Bild handelt. Das Anklicken des Links birgt ein Risiko!
```

Abbildung 18: Lokalisierungen *email-Form_de.properties*

```
form.field.empty.FormImageUploadElementComponentModel
= Bitte w\u00E4hlen Sie ein Bild aus
text.baywaForm.send.error.uploadSizeExceeded
= Die hochgeladenen Bilddateien \u00FCbersteigen die maximale Dateigr\u00F6\u00DFe von 5 MB.
text.baywaForm.send.error.invalidImage
= Es k\u00F6nnen nur Bilder im Format PNG oder JPEG hochgeladen werden.
```

Abbildung 19: Lokalisierungen in *base_de.properties*

Zeichenkette	Unicode-Zeichen
\u00E4	ä
\u00F6	ö
\u00FC	ü
\u00DF	ß

Tabelle 3: Zuordnung von Zeichenketten zum jeweiligen Unicode-Zeichen

5 Betrachtung möglicher Fehlerfälle beim Hochladen von Bildern

Bei der Verwendung der Bildupload-Komponente kann es zu vielen Bedienfehlern des Anwenders kommen. Um das aufgetretene Problem transparent zu machen, ist es wichtig eine Behandlung möglicher Fehler zu implementieren. Im vorliegenden Fall sind das die folgenden drei:

- Die Komponente wurde als Pflichtfeld nicht ausgefüllt
- Die ausgewählte Datei ist zu groß
- Die ausgewählte Datei hat ein nicht unterstütztes Format

Im Folgenden wird die Lösung des jeweiligen Fehlerfalls genauer beleuchten.

5.1 Leeres Pflichtfeld

Die Felder des Formulars können bei der Erstellung als Pflichtfeld definiert werden. Ein Absenden kann dann nur erfolgen, wenn alle Pflichtfelder einen Wert erhalten haben. Im Fall der Bildupload-Komponente muss demnach ein Bild ausgewählt werden. Sollte das nicht geschehen, so muss der Nutzer auf die falsche Bedienung hingewiesen werden.

Wie in Kapitel 4.2 erwähnt, wird beim Absenden des Formulars eine Validierung in der Methode *sendForm* vorgenommen. Eine vorhandene Implementierung prüft bereits, ob ein Feld leer ist oder einer bestimmten Eingabe folgt. Bisher war es allerdings nur notwendig, String-Werte zu prüfen. Für das Versenden eines Bildes muss zusätzlich der Typ MultipartFile behandelt werden.

```
private void validateMandatoryField(
    final FormElementComponentModel inputComponent,
    final FormComponentForm formComponentForm,
    final BindingResult errors) {
        Object fieldValue = formComponentForm.getFields().
            get(inputComponent.getFormElementName());
        if (fieldEmpty(fieldValue))
        {
            errors.rejectValue(getFieldName(inputComponent),
                FIELD_ERROR_PREFIX_EMPTY +
                inputComponent.getClass().getSimpleName());
        }
}

private boolean fieldEmpty(final Object fieldValue) {
    return Objects.isNull(fieldValue)
        || ((fieldValue instanceof String)
            && StringUtils.isEmpty((String) fieldValue))
        || ((fieldValue instanceof MultipartFile)
            && ((MultipartFile)fieldValue).isEmpty());
}
```

Abbildung 20: Implementierung der Prüfung auf leere Pflichtfelder

Der relevante Abschnitt im Quellcode ist in Abbildung 20 dargestellt. Die Methode *validateMandatoryField* beschafft sich den Wert aus dem übergebenen Formularfeld und lehnt diesen mit einem zusammengesetzten Fehlercode ab, sollte das Feld leer sein. Für den Bildupload wäre dieser Fehlercode *form.field.empty.FormImageUploadElementComponentModel*, welcher für die Anzeige der Fehlermeldung sorgt. Angepasst werden muss die Methode *fieldEmpty*, damit auch die Bildupload-Komponente nach dem gleichen Prinzip behandelt werden kann. Ergänzt wird diese um eine Abfrage, ob es sich bei dem Feldwert um ein Objekt vom Typ MultipartFile handelt. Sollte das der Fall sein, so wird über den Aufruf von *isEmpty* die eigentliche Prüfung auf ein leeres Feld durchgeführt. In Anlage 1 ist die Darstellung der Fehlermeldung in der Storefront zu sehen.

5.2 Datei zu groß

Da die Anforderungen das Hochladen von Bildern nicht zu lassen, welche größer als 5 MiB sind, muss der Anwender über das Überschreiten der Maximalgröße informiert

werden. In der Datei *core-spring.xml* wird dazu die Java-Bean „multipartResolver“ wie in Abbildung 21 angepasst, mit der sich Spring intern um die Behandlung von Formularen mit versendeten Dateien kümmert.[19]

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="5242880" /> <!-- 5 MB -->
    <property name="resolveLazily" value="true" />
</bean>
```

Abbildung 21: Konfiguration von multipartResolver in core-spring.xml

Das Attribut „maxUploadSize“ spezifiziert hierbei die maximal erlaubte Größe in Bytes bevor ein Upload abgelehnt wird.[20] Im vorliegenden Fall sind das 5 MiB. Zusätzlich wird das Attribut „resolveLazily“ auf den Wert „true“ gesetzt, wodurch Fehler bei überstiegener Dateigröße erst bei Parameterzugriff auf die Datei (also im Controller) ausgelöst werden.[21] Das erlaubt eine Ergänzung in der Klasse „FormController“, da vor Ausführung des Controllers eine Ausnahme erfolgen würde, die global behandelt werden müsste. Über die Spring-Annotation „@ExceptionHandler“ kann eine Methode definiert werden, welche bei Auftreten bestimmter Ausnahmen ausgeführt wird. Im Falle einer zu großen Datei löst Spring eine „MaxUploadSizeExceededException“ aus. Diese wird nach diesem Prinzip wie in Abbildung 22 behandelt.

```
@ExceptionHandler(MaxUploadSizeExceededException.class)
public ResponseEntity<?> handleMaxUploadSizeExceededException(
    MaxUploadSizeExceededException ex, final Model model){
        model.addAttribute(ATTRIBUTE_HAS_ERROR, Boolean.TRUE);
        return ResponseEntity.status(HttpStatus.PAYLOAD_TOO_LARGE).build();
}
```

Abbildung 22: Behandlung der MaxUploadSizeExceededException

Die Methode *handleMaxUploadSizeExceededException* wird aufgerufen, wenn die entsprechende Ausnahme auftritt und gibt dann lediglich einen HTTP-Statuscode „413

[19] [Spr19b]
[20] [Hoe19a]
[21] [HC03b]

Payload Too Large“ zurück. Entsprechend Kapitel 4.1 wird der Statuscode der Server-Antwort überprüft und dort die entsprechende Fehlermeldung „uploadSizeExceededErrorMessage“ ausgegeben. Zusätzlich wird auf den Statuscode „0“ abgefragt. Diesen gibt es im eigentlichen Sinne nicht. Durch die Verwendung des Webservers Tomcat[22] wird die abgesendete Anfrage allerdings schon anhand der Größe gefiltert, bevor die Businesslogik ausgeführt wird. Sollte die Anfrage die, durch das Attribut „maxSwallowSize“, konfigurierte Größe übersteigen, so wird diese bereits vom Server abgelehnt.[23] Die versendete Ajax-Anfrage wird demnach vom Server keine Antwort erhalten, was als Antwort der Länge 0 zu verstehen ist. Somit ist auch der Statuscode nicht gesetzt und hat den Standardwert 0. Der Fall kann daher genauso wie Status „413“ behandelt werden. Vorher muss allerdings die Größenlimitierung der Anfrage angepasst werden, da der Standardwert 2 MiB beträgt, bei zulässigen Bildern bis zu 5 MiB jedoch zu gering ist. Da bei der Größe der Anfrage alle Werte mit einfließen und nicht nur die Bildgröße, wird das Limit auf 5,5 MiB festgelegt, um auch bei der Übertragung maximal großer Bilder alle Header mitsenden zu können. In der Datei *server.xml* wird, wie in Abbildung 23 zu sehen ist, das Attribut „maxSwallowSize“ gesetzt.

```
<Connector
    <!-- … andere Attribute … -->
    maxSwallowSize="5767168" <!-- 5.5 MB Default: 2097152 -->
/>
```

Abbildung 23: Setzen der maximalen Anfragegröße in der Datei *server.xml*

Abbildung 24 veranschaulicht den Prozess der Größenprüfung. Die Anfrage wird zuerst von Tomcat ausgewertet und im Falle einer Größe unter 5,5 MiB wird im Controller die Datei auf ein Maximum von 5 MiB geprüft. Wenn in beiden Fällen die Größe nicht überstiegen ist, so wird das Formular erfolgreich versendet.

---

[22] http://tomcat.apache.org/
[23] [Apa18]

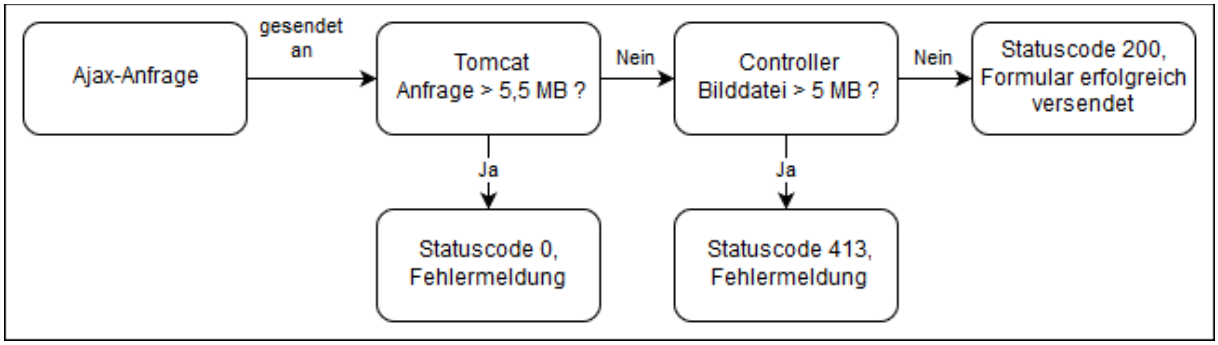


Abbildung 24: Prozess der Größenprüfung

Die Fehlermeldung wird, ähnlich anderer Meldungen im bestehenden Projekt, über eine Nachricht am Bildschirmrand ausgegeben. In Anlage 2 ist die Fehlermeldung in der Storefront zu sehen.

5.3 Falsches Dateiformat

Die Information über eine hochgeladene Datei im falschen Format (weder PNG noch JPEG) wird über die bereits erwähnte FileUploadException realisiert. Sie wird in der Methode *createMediaFromUserUpload* in der Klasse DefaultMediaFacade ausgelöst, wenn die zu verarbeitende Datei als ungültiges Bild erkannt wird. Die Exception wird bis in die Controllermethode weitergegeben und wird dort in einem ergänzten Catch-Block abgefangen. In diesem wird dem Model ein Attribut „hasInvalidImageError“ mit dem Wert „true“ hinzugefügt. In Abbildung 25 vereinfacht dargestellt.

```
try
{
    // ... Validierung ...
    // Potentielle FileUploadException in getFormDTO
    getFormService().sendForm(getFormDTO(component,formComponentForm));
    // ...
}
catch (FileUploadException e)
{
    model.addAttribute(ATTRIBUTE_HAS_INVALID_IMAGE_ERROR, Boolean.TRUE);
}
```

Abbildung 25: Vereinfachter try-catch-Block der Methode *sendForm*

Die Datei javaScriptVariables.tag definiert ein JavaScript. In diesem wird ein Objekt „DS“ angelegt, welches weitere verschachtelte Unterobjekte besitzt. In der jeweils letzten Ebene eines Objektpfads werden Variablen für den Zugriff in JavaScript definiert. Für die Anzeige einer Fehlermeldung bei falschem Bildformat ist die Ergänzung aus Abbildung 26 relevant. Der Variable „invalidImageError“ wird der aufgelöste Wert des Lokalisierungsschlüssels „text.Form.send.error.invalidImage“ zugeteilt. Über den Objektpfad „DS.translations.form.invalidImageError“ lässt sich die Variable im JavaScript aufrufen.

```
<%-- … Einbindung von Taglibs … --%>
<%-- JS configuration --%>
<script type="text/javascript">
    var DS = {
        <%-- … Andere Objektdefinitionen … --%>
        translations: {
            <%-- … Andere Objektdefinitionen … --%>
            form: {
                <%-- … Andere Variablen … --%>
                invalidImageError:
                '<spring:theme code="text.form.send.error.invalidImage" />'
            },
    }
};
</script>
```

Abbildung 26: Definition der Variable „invalidImageError“ in *javaScriptVariables.tag*

Von der Controllermethode wird die Datei *sendForm.jsp* zurückgeliefert und im JSON-Format als Antwort gesendet. Dort wird das Attribut „hasInvalidImageError“, welches im Controller dem Model hinzugefügt wurde, im JSON als Variable mit dem Wert „true“ übernommen. Abbildung 27 zeigt den Inhalt der Datei.

```
<%@ page trimDirectiveWhitespaces="true" contentType="application/json" %>
<%-- … Einbindung Taglibs … --%>
{
    <%-- … weitere JSON-Variablen … --%>
    <%-- Setzen von hasInvalidImageError --%>
    "hasInvalidImageError": ${hasInvalidImageError ? true : false},
}
```

Abbildung 27: Antwort im JSON-Format in der Datei *sendForm.jsp*

Bei einer erfolgreichen Ajax-Anfrage wird die gesendete Antwort der JavaScript-Funktion *formCallback* übergeben, wie in Abbildung 8 zu sehen ist. In dieser wird über eine if-else-Verzweigung abgefragt, ob der Wert von „hasInvalidImageError“ in der empfangenen Antwort „true“ ist. Sollte das der Fall sein, so wird über eine Hilfsfunktion die Fehlermeldung für falsche Bildformate am Bildschirmrand ausgegeben. Abbildung 28 zeigt den relevanten Codeabschnitt diesbezüglich. In Anlage 3 ist die Darstellung der Fehlermeldung in der Storefront zu sehen.

```
if (response.hasError) {
    DS.Utils.showAlert(DS.translations.form.sendFormError, 'danger');
} else if(response.hasValidationError) {
    DS.Utils.showAlert(DS.translations.global.formValidationError,
        'danger');
} else if(response.hasInvalidImageError) { // Abfrage auf valides Bild
    DS.Utils.showAlert(DS.translations.form.invalidImageError, 'danger');
} else {
    DS.Utils.showAlert(DS.translations.form.sendFormConfirmation,
        'success');
}
```

Abbildung 28: Bedingte Fehlermeldung bei ungültigem Bild in *formCallback*

6 Fazit und Ausblick

Das Ziel der vorliegenden Projektarbeit war die Erweiterung eines bestehenden, dynamischen Formulargenerators um die Möglichkeit eines Bild-Uploads, in Form einer eigenen Komponente. Mit dieser sollte es möglich sein, Bilder über ein Formular zu versenden.

Eine Lösung wurde entsprechend der Anforderungen implementiert. Somit kann mit Hilfe der erstellten Komponente einem Formular eine Bilddatei mitgesendet werden. Unterstützte Dateiformate sind hierbei PNG und JPEG. Die Größe aller im Formular enthaltener Bilder muss unterhalb von 5 MiB sein. Bei falscher Bedienung werden entsprechende Fehlermeldungen ausgegeben. Die versendeten Bilder werden gespeichert und in der E-Mail, mit einem Warnhinweis auf mögliche Gefahren, als Link referenziert. Ein Beispielformular ist in Anlage 4 und eine darüber versendete E-Mail in Anlage 5 dargestellt.

Die Maximalgröße von 5 MiB sollte in Zukunft so angepasst werden, dass sie nicht die gesamte Größe der im Formular versendeten Bilder als Kriterium verwendet. Stattdessen sollte bei der Verwendung von mehreren Bild-Upload-Komponenten in einem Formular jedes Bild einzeln ausgewertet werden. Vorerst ist die Lösung ausreichend, da das Eintreten dieses Falls eher selten und dessen Auswirkungen eher gering sind. Es wäre außerdem wünschenswert gewesen, eine Vorschau der ausgewählten Bilder in der Storefront zu ergänzen. Zeitlich war dies jedoch im Rahmen dieser Arbeit nicht möglich.

Hochgeladene Bilder werden in einem festgelegten Verzeichnis gespeichert. Dieses ist bisher nicht änderbar und wird im Quellcode gesetzt. Für eine bessere Bedienung sollte es dem Shopeigentümer möglich sein das Verzeichnis selbst festzulegen und zu ändern. Ebenso hat dieser keinen direkten Einfluss auf die unterstützten Dateiformate, welche von der Bild-Upload-Komponente akzeptiert werden. Es müsste eine einfache

Möglichkeit zur Anpassung geschaffen werden, wenn in der Zukunft weitere Formate ergänzt werden sollen.

Weiterhin kritisch zu betrachten ist die fehlende Prüfung der Dateien auf schädliche Inhalte. Die Implementierung versucht dieses Risiko zu minimieren, indem der Downloadlink einen Warnhinweis erhält und ein Herunterladen der Datei explizit angewiesen werden muss. Die Überprüfung durch einen Virenscanner sollte jedoch hinsichtlich der Anforderung auf minimales Risiko für das Produktivsystem nach wie vor umgesetzt werden.

Zuletzt sollte die Umsetzung der Zusicherung eines validen Bildformats überdacht werden. Für die geforderten Formate PNG und JPEG funktioniert diese nach einigen Tests zwar zuverlässig. Allerdings ist die verwendete Java-Standard-Schnittstelle sehr begrenzt, sollten zukünftig komplexere Bildformate unterstützt werden. In diesem Fall sollte der Nutzen einer entsprechenden externen Bibliothek evaluiert werden, welche diese Lücke füllt.

VI Literaturverzeichnis

[Apa18] Apache: „The HTTP Connector”, 2018, https://tomcat.apache.org/tomcat-9.0-doc/config/http.html Abruf: 31.01.2019

[Apa19] Apache: „The Apache Velocity Project”, 2019, http://velocity.apache.org/engine/1.7/user-guide.html Abruf: 31.01.2019

[Jqu19] jQuery: „jQuery.ajax()”, 2019 http://api.jquery.com/jquery.ajax/#jQuery-ajax-settings Abruf: 31.01.2019

[Ora18] Oracle: „Package javax.imageio”, 2018, https://docs.oracle.com/javase/8/docs/api/javax/imageio/package-summary.html Abruf: 31.01.2019

[Ora19] Oracle: „JSP Tag Libraries”, 2019, https://docs.oracle.com/cd/B10463_01/web.904/b10320/taglibs.htm#1012445 Abruf: 01.02.2019

[Rig18] Righetto, D.: „Protect FileUplod Against Malicious File”, 2018, https://www.owasp.org/index.php/Protect_FileUpload_Against_Malicious_File#Case_n.C2.B03:_Images Abruf: 31.01.2019

[Sap19a] SAP: „Spring Framework in SAP Commerce“, 2019, https://help.hybris.com/6.2.0/hcd/8c63621986691014a7e0a18695d7d410.html Abruf: 31.01.2019

[Sap19b] SAP: „Jalo Layer”, 2019, https://help.hybris.com/6.6.0/hcd/8c00066686691014a5a5d19875a1525b.html Abruf: 31.01.2019

[Spr19a] Spring: „Interface MultipartResolver”, 2019, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/multipart/MultipartResolver.html Abruf: 31.01.2019

[Spr19b] Spring: „Spring’s multipart (fileupload) support”, 2019, https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch16s08.html Abruf: 31.01.2019

[HC03a] Hoeller, J. und Cook, T.: „Interface MultipartFile”, 2003, https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/multipart/MultipartFile.html#transferTo-java.io.File Abruf: 31.01.2019

[HC03b] Hoeller, J. und Cook, T.: „CommonsMultipartResolver”, 2003, https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/multipart/commons/CommonsMultipartResolver.html#setResolveLazily-boolean- Abruf: 31.01.2019

[Hoe19] Hoeller J.: „Class CommonsFileUploadSupport”, 2019, https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/multipart/commons/CommonsFileUploadSupport.html#setMaxUploadSize-long Abruf: 31.01.2019

[W3c17a] W3C: „HTML 5.2”, 2017, https://www.w3.org/TR/html5/sec-forms.html#forms-form-submission Abruf: 31.01.2019

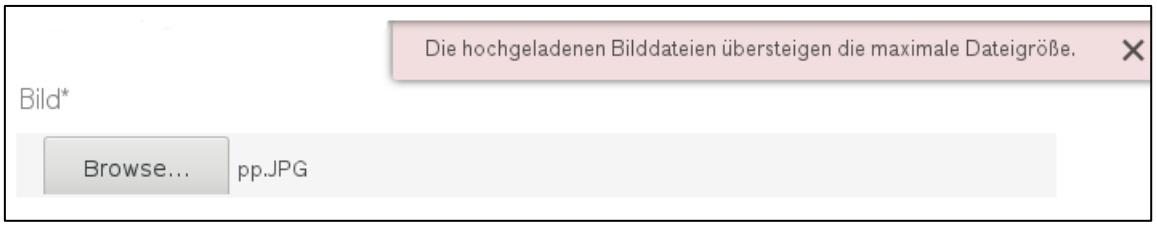
[W3c17b] W3C: „HTML 5.2“, 2017, https://www.w3.org/TR/html5/sec-forms.html#form-control-infrastructure-form-submission Abruf: 31.01.2019

[W3c17c] W3C: „HTML 5.2, 2017, https://www.w3.org/TR/html5/sec-forms.html#plain-text-form-data Abruf: 31.01.2019

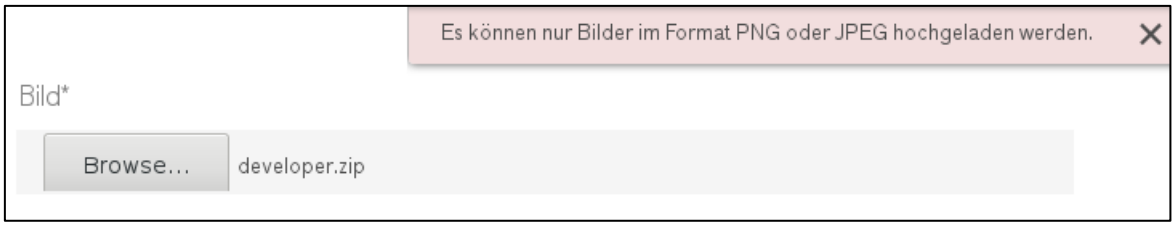
[W3c19] W3C: „Forms”, 2019, https://www.w3.org/TR/html401/interact/forms.html#h-17.13.4 Abruf: 31.01.2019

VII Anlagen

Anlage 1: Fehlermeldung bei leerem Pflichtfeld



Anlage 2: Fehlermeldung bei zu großem Bild



Anlage 3: Fehlermeldung bei falschem Dateiformat

Anlage 4: Demo-Formular mit Bildupload-Komponente

BILD ENTFERNT, PERSONENBEZOGENE DATEN

Anlage 5: Versendete E-Mail beim Abschicken des Formulars aus Anlage 4

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Projektarbeit mit dem Thema:

   Erweiterung eines dynamischen Formulargenerators um eine Bildupload-Funktion

   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 Projektarbeit bei keiner anderen Prüfung vorgelegt habe.

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

_____ _____

Ort, Datum Unterschrift