JAVA Mag inkl. CD

Googles JPA: Bigger Table, lesser Standard? »80

S&S

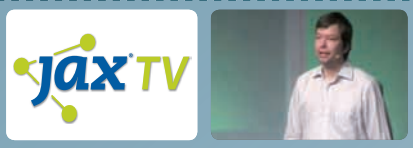
Deutschland €8,50 Österreich €9,80 Schweiz sFr 16,80 12.2011

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

CD-INHALT



Lean Enterprise Architecture

Video von der JAX 2011

HIGHLIGHT

Death of XP

Java Tech Journal Special Edition EXKLUSIV für unsere Leser

WEITERE INHALTE

- Lucene&Solr 3.4
- ActiveMQ 5.5
- CXF 2.4.3

Alle CD-Infos ab Seite 3

Geoinformationssysteme

Orte malen »86

Soft Skills Konkret

Effektiv vermitteln, aber wie »97

W-JAX 11

Brandneues Programm »51

Open Source Integration

Enterprise Service Buses für alle! »18

Best Practices mit Git

Besser Gits nicht! »100

Solr unter Strom

Suchmagie für Applikationsentwickler »48

Apache Cayenne

Mit Remote Objects Welten verbinden »30

4 194586 708507 12

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Ein Ansatz zur Überprüfung über mehrere Felder

Aufs Kreuz gelegt

Die meisten JSF-Entwickler kennen das Problem: Die Eingaben eines Anwenders müssen vor der Weitergabe an das Backend geprüft werden. Dabei ist aber häufig nicht nur der Wert eines einzelnen Feldes entscheidend, vielmehr ergibt sich die Gültigkeit erst aus der Kombination mehrerer Felder. Ziel dieses Artikels ist es, einen Ansatz vorzustellen, mit dem sich schon in der JSF-Validierungsphase eine Überprüfung über mehrere Felder, auch Cross-Validation genannt, realisieren lässt.

von Valentino Pola, Gregor Tudan

Ein wesentliches Argument für Web-Frameworks ist, dass sie dem Entwickler Routinetätigkeiten erleichtern. Eine davon ist das Lesen von Rückgabewerten aus HTML-Seiten und das Überführen der Eingaben in die entsprechenden Java Beans. In JSF ist dies mehrstufig realisiert: Erst werden die Feldwerte einzeln in Java-Typen konvertiert, anschließend validiert und schlussendlich die entsprechende Bean befüllt. Dies hat jedoch den Nachteil, dass in der Validierungsphase noch nicht die gesamte Bean, sondern nur jeweils der einzelne Feldwert vorliegt. Zwar lässt sich die Entity-Klasse nach der eigentlichen Befüllung erneut mittels Bean-Validation überprüfen, dies birgt jedoch Nachteile in puncto Usability: Zuerst werden die einzelnen Felder validiert und eventuelle Fehler dem Nutzer angezeigt. Sind diese behoben und die Seite erneut gesendet, startet die Bean-Validation und der Nutzer wird mit neuen Fehlermeldungen konfrontiert – viele dürften dies unkomfortabel, wenn nicht gar verwirrend finden.

Abgrenzung

Für das vorgestellte Problem, die Prüfung mehrerer abhängiger Eingabefelder, existieren bereits einige Ansätze: Für Validierungen über einzelne Felder hinweg wird zumeist die bereits erwähnte Bean-Validation (JSR 303) verwendet. Hier lassen sich so genannte Class Level Constraints definieren. Um diese nutzen zu können, muss jedoch bereits die komplette Bean durch JSF befüllt worden sein, was erst nach der Validation-Phase der Fall ist. Somit ergibt sich der eingangs erwähnte Effekt, dass Fehler in zwei Phasen angezeigt werden. Ist dies nicht erwünscht, bliebe nur der komplette Verzicht auf JSF-Validatoren zugunsten der Bean-Validation.

Problematisch ist dieser Ansatz auch im Fall von Assistenten, bei denen die Eingaben in mehreren Schritten abgefragt werden: Hierbei wird die Bean erst nach und nach befüllt, wodurch die Validierung erst im letzten Dialogschritt möglich wäre.

Eine weitere Möglichkeit ist die Validierung auf HTML-Ebene mittels JavaScript, wobei Fehler direkt bei der Eingabe, und somit bereits vor dem Absenden der Seite, hervorgehoben werden. Allerdings erscheint dieser Ansatz insbesondere für sensible Daten ungeeignet, findet hier doch die Prüfung ausschließlich clientseitig statt. Um Missbrauch vorzubeugen, müssten die Daten auf Serverseite erneut validiert werden.

Ausgangssituation

Als Beispiel soll ein typisches Zahlungsformular dienen, wie man es in den meisten Webshops vorfindet: Hier werden die Kreditkartendaten, bestehend aus Kartentyp (z. B. Visa, MasterCard...), Nummer und Gültigkeitsdatum abgefragt, letzteres in getrennten Comboboxen für Monat und Jahr (**Abb. 1**). Schon für diesen einfachen Fall ist Cross-Validation notwendig: Da sich das Gültigkeitsdatum erst aus der Kombination der beiden Felder für Monat und Jahr ergibt, können diese nicht unabhängig voneinander geprüft werden.

Weiter soll die Kartennummer auf mögliche Tippfehler überprüft werden, was mittels einer einfachen Prüfsumme möglich ist [1]. Hierbei geht auch der Kartentyp ein, woraus sich ein zweiter Fall für Cross-Validation ergibt.

Bei einer klassischen Implementierung würde man hier zum Beispiel folgendermaßen vorgehen: In der Oberfläche werden für die Einzelfeld-Validierung Validatoren

Abb. 1: Eingabeformular aus dem oben genannten Beispiel vor und nach der Validierung

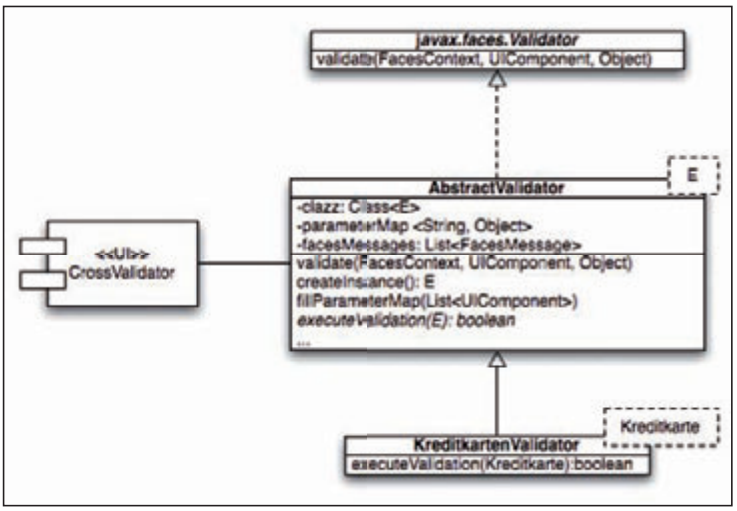


Abb. 2: Klassendiagramm CrossValidator

definiert. Zusätzlich wird in der zuständigen Backing Bean folgender Code für die Verarbeitung des Submits hinterlegt. Zwar ist der Ansatz einfach nachvollziehbar, jedoch birgt er zwei wesentliche Nachteile:

- Wird im ersten Schritt bspw. vergessen, die Kreditkartennummer einzutragen, würde die Fehlermeldung „Kreditkartennummer eintragen“ erscheinen. Nachdem die Nummer ergänzt wurde, erscheint die Fehlermeldung „Kartentyp passt nicht zur Nummer“.

Listing 1

```
<h:form >
  <coi:crossvalidator validator="#{creditCardValidator.validate}">
    <h:outputText value="Karte"/>
    <h:selectOneMenu
    value="#{creditCardBean.creditCard.cardType}">
[...]
    </h:selectOneMenu>
    <h:outputText value="Kartennummer" />
    <h:inputText value="#{creditCardBean.creditCard.cardNumber}"/>
[...]
  </coi:crossvalidator>
</h:form>
```

Listing 2

```
<composite:interface>
  <composite:attribute name="validator" required="true"
    method-signature="void f(Javax.faces.context.FacesContext,
                javax.faces.component.UIComponent, java.lang.Object)" />
</composite:interface>
<composite:implementation>
  <h:messages for="crossvalidator" />
  <ui:fragment><composite:insertChildren /></ui:fragment>
  <h:inputHidden id="crossvalidator"validator="#{cc.attrs.validator}"/>
</composite:implementation>
```

- Das Modell ist bereits mit den fachlich falschen Daten befüllt, was im schlimmsten Fall zu einem inkonsistenten Datenmodell führen kann.

Lösungsansatz

Mit JSF 2 wurde die Implementierung eigener Komponenten enorm vereinfacht, warum also nicht eine Komponente für die feldübergreifende Validierung implementieren? Diese könnte aussehen wie in Listing 1 dargestellt.

Die Komponente *<coi:crossvalidator>* umschließt jene Eingabefelder, die gemeinsam überprüft werden sollen. Als Attribut wird eine Klasse übergeben, welche die eigentliche Validierung realisiert. Wird ein Submit abgesetzt, soll die Komponente innerhalb der Validationsphase eingreifen und dabei die Werte der eingeschlossenen Felder laden, eine Instanz der eigentlichen Bean erzeugen, sie mit den Formularwerten befüllen und im Anschluss die Validierung durchführen. Treten Validierungsfehler auf, können entsprechende Fehlermeldungen entweder direkt an die Eingabefelder angehängt oder in einem Bereich oberhalb des Formulars angezeigt werden.

Um nicht bei jeder Implementierung des Validators den Code zum Lesen von Attributwerten neu zu schreiben, wird eine abstrakte Oberklasse realisiert, von der konkrete Implementierungen erben können. Dort wer-

Listing 3

```
public abstract class AbstractCrossValidator<E> implements Validator {
  private Class<E> clazz;
  private Map<String, Object> parameterMap = new HashMap<>();
  private List<FacesMessage> errorMessages = new ArrayList<>();

  public AbstractCrossValidator() {
    clazz = (Class<E>) ((ParameterizedType) getClass()
    .getGenericSuperclass()).getActualTypeArguments()[0];
  }

  public abstract boolean executeValidation(E instanceToValidate);
}
```

Listing 4

```
private void fillParameterMap(List<UIComponent> components) {
  for (UIComponent child : components) {
    if (child instanceof UIInput) {
      UIInput input = (UIInput) child;
      if (!parameterMap
      .containsKey(getPropertyNameByUIComponent(input))) {
      String propertyName = getPropertyNameByUIComponent(child);
      parameterMap.put(propertyName, input.getValue());
      }
    }
  }
}
```

den die Werte aus den Eingabefeldern gelesen und in eine transiente Instanz der Entität geschrieben, auf deren Basis dann die Validierung durchgeführt wird.

Bausteinsicht

Abbildung 2 zeigt auf abstraktem Niveau die drei Komponenten, die zur Validierung genutzt werden.

Um die Komponente auf der Oberfläche zu nutzen, wird eine JSF-Composite-Komponente (*CrossValidator*) implementiert, in der ein vom Typ *AbstractValidator* abgeleiteter Validator angegeben wird.

Die *AbstractValidator*-Komponente stellt die Funktionalität auf Java-Seite zur Verfügung, um aus den Eingabefeldern die Werte in eine temporäre Entity zu laden, oder im Fehlerfall FacesMessages zu erzeugen und an entsprechender Stelle auszugeben. Diese Komponente ist kontextunabhängig einsetzbar und muss daher in der Regel nicht angepasst werden.

Der *CreditCardValidator* leitet von *javax.faces.Validator* ab und realisiert die fachliche Validierungslogik innerhalb der abstrakten Methode *executeValidation(...)*.

JSF-Komponente CrossValidator

Nach erfolgreicher Konvertierungsphase führt JSF in der Validierungsphase alle Validatoren aus, die an die Felder des Formulars gebunden sind (z. B.: *<h:inputText … validator="#{myValidator.validate}"*). Die Grundidee ist, hier einen feldübergreifenden Validator einzubinden. Listing 2 zeigt die Definition der JSF-Composite-Komponente *CrossValidator*.

Im Attribut *validator* muss ein Method Binding übergeben werden, welches die Prüfung durchführt. Das Metaattribut *method-signature* beschreibt, welche Methodensignatur die anzubindende Methode aufzuweisen hat. Diese haben wir aus *javax.faces.Validator#validate* übernommen, warum wird gleich ersichtlich.

Im *<composite:implementation>*-Teil wird definiert, welche JSF-Komponenten eingefügt werden sollen, wenn die Komponente innerhalb eines Facelets verwendet wird.

<h:messages> wird dazu genutzt, um eventuelle Validierungsfehler anzuzeigen. Mit *<composite:insertChildren/>* wird festgelegt, dass an dieser Stelle innerhalb der Komponente weitere Komponenten eingebettet werden können. Aufgrund eines JSF Bugs [2] ist das umgebende Tag *<ui:fragments>* notwendig, damit nach fehlgeschlagener Validierung die Formularfelder nicht geleert werden.

Der Kern unserer Idee steckt jedoch im *InputHidden*-Feld. Dieses Feld wird zwar auf der generierten HTML-Seite nicht angezeigt, allerdings verhält es sich wie alle anderen *UIInput*-Komponenten: Innerhalb der Validierungsphase wird es mittels des registrierten Validators geprüft. Dies machen wir uns zu Nutze, indem wir das Method Binding des *validator*-Attributs von *CrossValidator* an das Feld durchreichen (*#{cc.attrs.validator}*).

AbstractValidator

Der *AbstractValidator* erfüllt mehrere Funktionen, die im Folgenden erläutert werden. Listing 3 zeigt die Grundstruktur. Um sicherzustellen, dass der Validator überhaupt an ein *UIInput*-Feld gebunden werden kann, implementiert er das *javax.faces.Validator*-Interface. Der Type-Parameter *<E>* gibt an, für welche Bean-Klasse die Validierung durchgeführt wird. Innerhalb des Konstruktors wird der konkrete Typ von *E* bestimmt und als Referenz gespeichert. Dies ist wichtig, da anhand dieser Information später eine neue Instanz der zu validierenden Bean erzeugt wird.

Die Methode *fillParameterMap* (Listing 4) wird genutzt, um aus einer Liste von *UIInput*-Komponenten deren Inhalt auszulesen und im Feld *parameterMap* zu speichern.

Die Methode *createInstance* (Listing 5) erzeugt eine neue Bean, die mit den Werten aus *parameterMap* befüllt wird.

Nun zur eigentlichen *Validate*-Methode (Listing 6): Anstatt selbst eine Validierung durchzuführen, werden die eingebetteten Komponenten innerhalb des *CrossValidators* gesammelt (exklusive dem *InputHidden*-Feld). Aus diesen Informationen wird im nächsten Schritt eine neue Bean-Instanz erzeugt. Die eigentliche Validierung wird an die abstrakte Methode *executeValidation* delegiert.

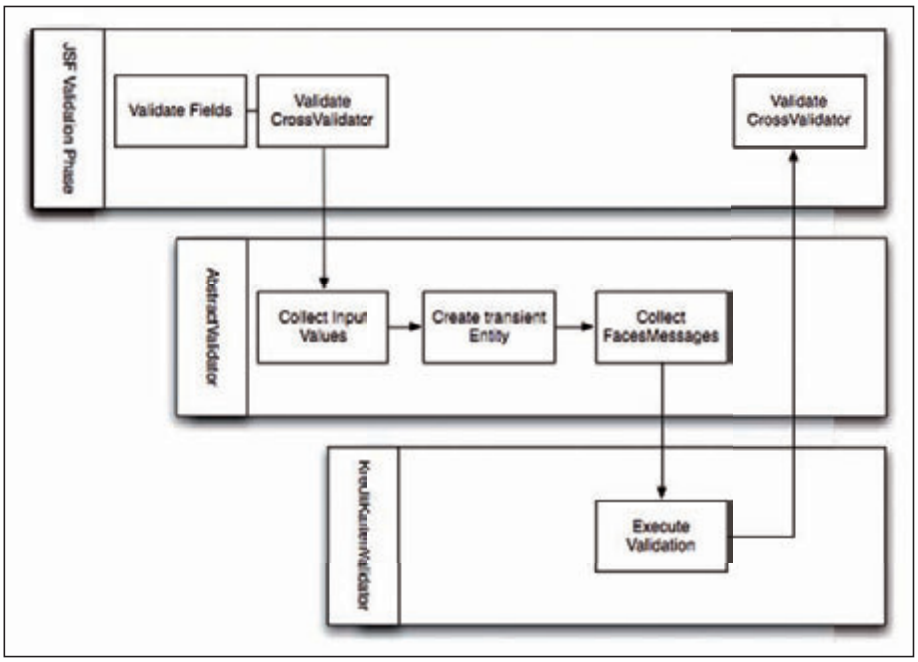


Abb. 3: Übergreifender Ablauf der Validierung innerhalb des CrossValidators

Diese Methode wird im konkreten Validator implementiert und realisiert dort die eigentliche fachliche Validierung. Der Rückgabewert signalisiert, ob die Validierung fehlgeschlagen ist und die *validate*-Methode eine *ValidatorException* werfen soll. Zudem bietet die *AbstractValidator*-Komponente mehrere Convenience-Methoden, um *FacesMessages* zu erzeugen und der Liste von *FaceMessages* hinzuzufügen.

KreditKartenValidator

Für die Realisierung eines Validators für unser konkretes Beispiel wird die Methode *executeValidation* überschrieben (Listing 7). Hier findet die fachliche Prüfung der Eingaben statt, in diesem Beispiel, ob Kartentyp und Kartennummer zueinander passen. Falls die Validierung fehlschlägt, wird mit *addMessage(...)* eine Fehlermeldung in *AbstractCrossValidator#facesMessages* eingefügt, die später auf der Oberfläche ausgegeben werden kann.

Um den Validator einfacher an den *CrossValidator* zu binden, ist es sinnvoll, ihn als *ManagedBean* zu deklarieren, wodurch er via Expression Language referenzierbar wird.

Laufzeitsicht

In **Abbildung 3** wird der Ablauf der Validierung noch einmal zusammengefasst. Nach einem Submit des Formulars werden zunächst alle Felder wie gewohnt zu Java-Objekten konvertiert und validiert. Im letzten Schritt der Validierung wird der *CrossValidator* ausgeführt, wobei zuerst die *validate*-Methode des *AbstractValidators* aufgerufen wird. Diese sammelt die Feldwerte in einer Map, um sie anschließend über *getInstance()* in eine neue Instanz der Bean zu schreiben.

Listing 5

```
private E createInstance() {
  E instance = null;
  try {
    instance = clazz.newInstance();
    for (String propertyName : parameterMap.keySet()) {
      Method m = findMethodFor(propertyName, clazz);
      if (m != null) {
        m.invoke(instance, parameterMap.get(propertyName));
      }
    }
  } catch [...]
  return instance;
}
```

Listing 6

```
public void validate(FacesContext ctx, UIComponent component, Object
                                                              value)
      throws ValidatorException {
  // Alle Kinder des CrossValidators ausser das Hidden Field
  List<UIComponent> children = new
                                  LinkedList<UIComponent>(component
      .getParent().getChildren());
  children.remove(component);
  fillParameterMap(children);
  E instanceToValidate = createInstance();
  executeValidation(instanceToValidate);
  if (!facesMessages.isEmpty()) {
    throw new ValidatorException(facesMessages);
  }
}
```

Listing 7

```
@ManagedBean(name = "creditCardValidator")
public class CreditCardValidator extends
                                  AbstractCrossValidator<CreditCard> {
  private Logger log = Logger.getLogger(CardValidator.class.getName());
  @Override
  public boolean executeValidation(CreditCard creditCard) {
    boolean isValid = validateCardType(creditCard.getCardNumber(),
                                  creditCard.getCardType());
    if(!isValid){
      addMessage(new FacesMessage("Der Kartentyp passt nicht zur
                                  Kartennummer"));
    }
    [...]
    return isValid;
  }
```

Nun wird die abstrakte Methode *executeValidation()* aufgerufen, welche die eigentliche Validierungslogik enthält. Innerhalb des konkreten Validators können eventuelle Fehlermeldungen über *add* an den *AbstractValidator* gemeldet werden. Der Rückgabewert der Methode gibt an, ob die Validierung erfolgreich war, oder fehlschlug.

Zurück in der *validate*-Methode wird der Rückgabewert geprüft und ggf. eine *ValidationException* geworfen, welche die gesammelten Fehlermeldungen beinhaltet. Dies führt zu einem Abbruch des JSF Lifecycles und man gelangt zurück zum Eingabeformular mit den eingeblendeten Fehlermeldungen.

Ausblick

Der vorgestellte Validator ermöglicht es, einfach eigene feldübergreifende Validierungen zu realisieren. Diese müssen jedoch nicht zwangsläufig in Java realisiert sein. Ein interessanter Ansatz wäre beispielsweise eine weitere Abstraktionsschicht, mit welcher sich in dynamischen Sprachen realisierte Validierungsregeln einbinden ließen. Zum Beispiel könnten hier Groovy-Regeln geladen, oder ein komplexeres Business Rules Management System (BRMS) angebunden werden, was viele weitere Möglichkeiten eröffnen würde. So könnten je nach aktueller Maske, Status des Systems, der Entität oder eines Prozesses unterschiedliche Regeln aufgerufen werden, welche sich zur Laufzeit anpassen ließen. Die UI-Komponente behilft sich aktuell mit einem Hidden Field, um einen Validator anstoßen zu können. Vorstellbar ist es, die *CrossValidator*-Komponente selbst von *javax.faces.UIInput* abzuleiten, um so den Validator direkt an die Komponente anhängen zu können. Der Quelltext des vorgestellten Beispiels kann unter [3] heruntergeladen werden.

Valentino Pola ist als Senior Expert Consultant bei der COINOR AG im Bereich der Softwarearchitektur und Entwicklung von Anwendungssystemen zur Abbildung von Bankprozessen tätig. Seine Schwerpunkte liegen bei der flexiblen Skalierbarkeit und dem anwendernahen Design. Technologisch beschäftigt er sich mit Technologien von JBoss Seam über JSF bis hin zur Verarbeitung von Massendaten mit ETL-Werkzeugen. Valentino Pola ist unter valentino.pola@coinor.de erreichbar.

Gregor Tudan ist Consultant bei der COINOR AG mit den Schwerpunkten Serviceorientierte Architekturen und BPM. In seinen aktuellen Projekten beschäftigt er sich mit der Erstellung von Enterprise-Java-Anwendungen und leistungsfähiger grafischer Oberflächen. Gregor Tudan ist erreichbar unter gregor.tudan@coinor.de.

Links & Literatur

[1] http://de.wikipedia.org/wiki/Luhn-Algorithmus

[2] http://java.net/jira/browse/JAVASERVERFACES-1991

[3] http://go.coinor.de/crossvalidator.zip