Tag-Entwicklung mit **Facelets**

Mit dem Schritt von der prozeduralen zur deklarativen Tag-Entwicklung werden anstelle von Tag-Methoden Facelets Templates verwendet, die in jars verpackt werden können. Die Entwicklung kann sich somit auf die Standard-JSF-Tags abstützen und braucht sich nicht in den Fallstricken der Komponentenhierarchie und des Lifecycles zu verfangen.

von Ganesh Jung

Der Artikel „Lego oder Playmobil“ [1] endete mit dem Ausblick auf eine neue Form der deklarativen Tag-Entwicklung mit Facelets, bei der die Schwierigkeiten, die die Entwicklung von JSF-Tags mit sich bringt, vermieden werden können. Mittlerweile wurden die ersten dieser Tags fertig gestellt, und im Oktober werden die ersten zwei Projekte in Produktion gebracht, die sie verwenden. Im Folgenden werden die hinter dieser Form der Tag-Entwicklung stehenden Ideen und der verwendete Code vorgestellt.

Einfache JSF-Oberflächen können mit Standard-JSF-Komponenten entwickelt werden. Auch für komplexe Anforderungen ist es in den meisten Fällen möglich, auf Basis der Pattern, die in dem Artikel „Lego oder Playmobil“ beschrieben wurden, eine Lösung mit Standard-JSF-Tags zu finden. Oft ist dabei die Verwendung von JavaScript nötig. Für Komponenten wie *Combobox*, *Shuttle* oder *Tree* ist es aber wünschenswert, eine fertige Lösung zu verwenden. Solche Lösungen werden üblicherweise als JSF-Tags implementiert und in Tag-Bibliotheken zusammengefasst. Die Entwicklung von JSF-Tags ist eine komplizierte Angelegenheit: Fehler beim Ableiten von der Komponentenhierarchie und bei Verwendung der Interfaces können die Komponentenentwicklung erschweren. Dadurch hat man oft die Wahl zwischen langem und umständlichem Code und der Verwendung von Hilfsklassen einer bestimmten JSF-Implementierung – entweder wird das Tag also schwer zu pflegen oder es wird abhängig von einer bestimmten JSF-Implementierung.

Die Tag-Entwicklung mit JSF ist voller Tücken. Mir sind drei große Anwendungen bekannt, für die seit Ende 2005 JSF-Tag-Bibliotheken entwickelt wurden. Alle sind bei einem Zustand angelangt, in dem nicht mehr darüber nachgedacht wird, wie die Bibliothek zu fixen ist, sondern darüber, wie man sie wieder los wird. Gründe für diese Schwierigkeiten sind in folgenden Bereichen zu finden:

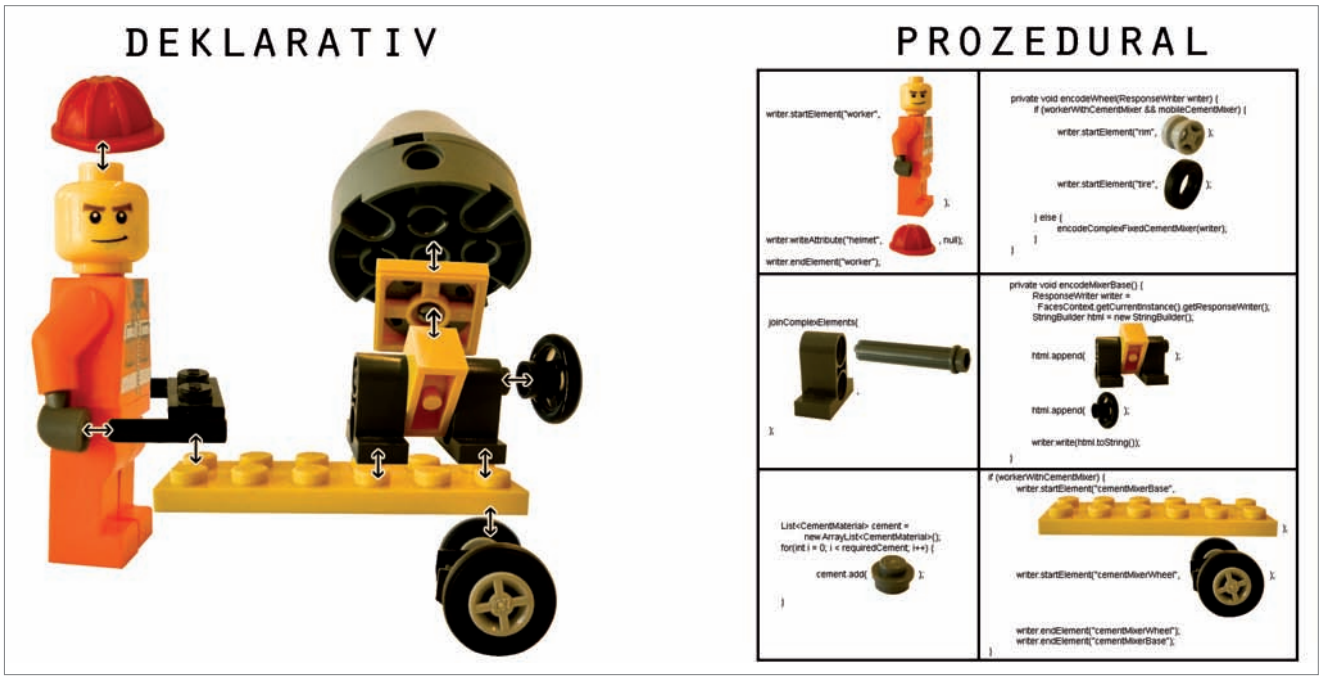
- Abhängigkeiten von einer JSF-Implementierung in eigenen Tag-Klassen (meist MyFaces, das lange die einzig brauchbare Implementierung bot) durch Verwendung von Teilen der Implementierung, die über die reine JSF-API hinausgehen.
- Die Komplexität der Tag-Implementierungen: Projekte/Unternehmen sollten fachliche Anforderungen umsetzen und nicht Standard-GUI-Bausteine entwickeln. Die längerfristige Wartung wird hier oft unterschätzt.
- Probleme mit dem Upgrade der Bibliotheken auf JSF 1.2 und Angst vor den Schwierigkeiten, die durch den Upgrade auf JSF 2.0 entstehen werden. Für Unternehmen rechnet sich meist ein aufwändiger Upgrade nicht, da sich hierfür kein Business Case rechnet.

Diese Schwierigkeiten entstehen nicht mit Template-gestützten Tags, wie sie im Folgenden beschrieben werden. Template Code, der aus JSF-Standard-Tags besteht, die mit JavaScript kombiniert werden, kann mithilfe von Facelets als Tag in ein *jar* verpackt werden. So können alle Fähigkeiten, die in die Standard-Tags hinein implementiert sind, voll genutzt werden. Neue Versionen der JSF-API machen dem Tag keine Schwierigkeiten und der deklarative Tag-Code ist leichter zu lesen und zu pflegen. Darüber hinaus kann der Tag-Code einfach aus dem Tag herausgelöst, individuell erweitert und direkt in die Applikation eingebunden werden, falls zusätzliche Funktionalität benötigt wird. Am Beispiel einer Combobox wird gezeigt, wie elegant ein solches deklaratives Tag entwickelt werden kann.

Beschreibung des Beispiel-Tags

Zunächst werden die Anforderungen an das Combobox Tag definiert. Eine Combobox besteht aus drei Teilen:

Quellcode auf CD



- Ein Eingabefeld, in dem der Benutzer frei editieren kann.
- Eine Tabelle, aus der der Benutzer vorgegebene Werte auswählen kann.
- Ein Knopf zum Öffnen und Schließen der Tabelle, der bei geöffneter Tabelle eine andere Grafik anzeigt, als bei geschlossener Tabelle.

Diese drei Elemente interagieren durch JavaScript:

- Wenn im Eingabefeld eine Taste gedrückt wird, öffnet sich die Tabelle.
- Wenn der Knopf gedrückt wird, öffnet und schließt sich die Tabelle und die Grafik des Knopfes ändert sich.
- Wenn man mit der Maus über die Tabelle fährt, wird die Zeile, über der sich der Mauszeiger befindet, ausgewählt und farblich unterlegt.
- Wenn in der Tabelle eine Zeile geklickt wird, wird der Wert, der zu dieser Zeile gehört, in das Eingabefeld übernommen.
- Wenn im Eingabefeld die Pfeil-unten-Taste gedrückt wird, wandert die ausgewählte und farblich unterlegte Zeile um 1 nach unten.
- Wenn im Eingabefeld die Pfeil-oben-Taste gedrückt wird, wandert die ausgewählte und farblich unterlegte Zeile um 1 nach oben.
- Wenn im Eingabefeld bei geöffneter Tabelle die Enter-Taste gedrückt wird, wird der Wert, der zur ausgewählten und farblich unterlegten Zeile gehört, in das Eingabefeld übernommen und die Tabelle schließt sich.

Was das Tag noch können soll:

- Die CSS Styles aller im Tag verwendeten Elemente sollen von außen zugänglich sein.
- Die Tabelle soll als Liste von JavaBeans übergeben werden.
- Die Tabelle, aus der die Auswahl erfolgt, soll mehrere Spalten darstellen können, deren Inhalte mithilfe der EL als JavaBean Properties angesprochen werden.
- Der Wert, der bei Auswahl einer Tabellenzeile in das Eingabefeld geschrieben wird, soll ebenfalls mithilfe der EL als JavaBean Property angesprochen werden

Die Combobox soll außerdem Ajax-fähig sein:

- Die JavaScript-Events des Eingabefelds sollen Ajax-Requests auslösen können.
- Wenn der Wert einer Tabellenzeile in das Eingabefeld übernommen wird, soll der *onchange*-Event des Eingabefelds ausgelöst werden.
- Die Tabelle soll durch die Ajax-Events, die das Eingabefeld auslöst, neu ren-

Facelets

Das Facelets-Projekt ist eine Open-Source-Software unter der CDDL-Lizenz. Projektleiter sind Jacob Hookom und Roger Kitain, die im JSR 252 an Version 1.2 der JSF-Spezifikation beteiligt waren. Im Rahmen der Entwicklung der JSF-Spezifikation Version 2.0 im JSR 314 wird Facelets voraussichtlich weitestgehend in die Standardspezifikation integriert.

Facelets erweitern JSF an vielen Stellen, die bei der Standard-JSF-Entwicklung besondere Schwierigkeiten machen. So kann z.B. HTML-Code einfach mit JSF-Tags gemischt werden, *EL*-Ausdrücke können ohne *h:outputText* verwendet werden und der Aufruf von Seitenfragmenten kann parametriesiert werden. Von besonderer Bedeutung für diesen Artikel ist die Möglichkeit, ein Seitenfragment als Tag zu verpacken und in einer Bibliothek auszuliefern.

derbar sein, damit die Combobox als Autosuggest Box (wie Google Suggest, z.B. *http://www.google.com/webhp?complete=1&hl=en)*, verwendet werden kann.

Zudem soll mindestens ein Satz CSS Styles zur Verfügung stehen, der als Vorlage für ein eigenes Styling der Combobox verwendet werden kann.

Das Beispiel-Tag

Das Eingabefeld und der Auf- und Zuklappknopf kommen in eine eigene kleine HTML-Table, die es erlaubt, zwischen Feld und Knopf noch einen Rand zu stylen. Die Tabelle mit der Listenauswahl kommt direkt darunter (Listing 1). Das Eingabefeld wird durch die Eingabeparameter *disabled* und *selectOnly* gesperrt. Die Scripts für die Tabellenauswahl werden ebenfalls deaktiviert, wenn *disabled=true* ist. Wenn *selectOnly=true* ist, wird die Combobox hingegen zu einer Selectbox, die einheitlich mit anderen auf einer Page vorkommenden Comboboxen mit CSS Styles versehen werden kann (anders als das *h:selectOneMenu*, das sich aufgrund von HTML/Browserbeschränkungen schlecht stylen lässt). Alle Styles sind über Tag-Parameter zugänglich. Die Übergabe der Werte für das Input-Feld und die Tabellenzeilen erfolgt über die Parameter *inputValue* und *dataList*.

Zur Erzeugung der Tabelle wird aber nicht das *h:dataTable* verwendet, weil dieses keinen Zugang zu den JavaScript-Events der Tabellenzeilen bietet. Stattdessen greifen wir wieder auf die JSTL zurück, iterieren mit *c:forEach* über die Liste und erzeugen die *tr*-Elemente der Tabellenzeilen direkt. Der Inhalt der Tabellenzeilen wird nicht im Template-Code vorgegeben. Stattdessen wird der Facelets-Platzhalter *<ui:insert name="tr" />* verwendet. Wenn das Tag in einer Anwendung verwendet wird, ist in den Body des Tags-Code zu schachteln, der nach dem Muster des folgenden Beispiels aufgebaut ist:

```
<ui:define name="tr">
 <td class="col1">
  <h:outputText value="#{item.name}" />
 </td>
 <td class="col2">
  ... weitere Tabellenspalten
</td>
</ui:define>
```

Dieser Zeileninhalt wird in jede Tabellenzeile übernommen. Die Laufvariable *item* ist dabei vom *c:forEach* durch *var=item* vorgegeben, es kann über die EL auf die Zeilenobjekte zugegriffen werden.

Die Liste wird ein zweites Mal durchlaufen, um für jede Zeile den Schlüssel hinzuzufügen, der beim Klicken der Zeile in das Input-Feld übernommen werden soll. Der Schlüssel wird mithilfe des Facelets-Platzhalters *<ui:insert name="rowKey" />* eingefügt, in der Anwendung steht im Body des Tags *<ui:define name="rowKey">#{item.key}</ui:define>*. Die Zeilennummer als Teil der HTML-ID im Template-Code hält Tabellenzeilen und dazugehörige Keys zusammen.

Die meisten Prinzipien, nach denen das Tag aufgebaut wird, sind hier bereits sichtbar:

- Der Zugriff auf die Parameter, die dem Tag übergeben werden, erfolgt per EL-Ausdruck. Facelets macht die Parameter also über den Request Scope verfügbar.
- JSF-Tags und HTML-Tags können ineinander verschachtelt werden.
- Die Listenauswahl wird mithilfe der JSTL-Tags *c:set* und *c:forEach* als HTML-Table gerendert – hierdurch besteht Zugriff auf die *mouseover-*, *mouseclick-* und *class*-Attribute der *<tr>*-Elemente (der Übersichtlichkeit halber nicht in Listing 1 enthalten).
- Eine JSF Managed Bean mit dem Namen *scriptHelper* erledigt Zugriffe, für die prozeduraler oder parametrisierter Code benötigt wird.
- Die Zeileninhalte der Listenauswahl werden über das Templating-System von Facelets bei Verwendung des Tags mitgegeben.

Bindings mit Script Helper Pattern

Nun kommen wir an die Grenze deklarativer Ausdrucksmöglichkeiten, da JavaScript benötigt wird, um den HTML-Code zu aktivieren. Allerdings muss

Deklarativ oder prozedural?

Bei der deklarativen Softwareentwicklung beschreibt der Quellcode das gewünschte Ergebnis, bei der prozeduralen Entwicklung wird der Weg, auf dem das Ergebnis ermittelt wird, codiert. Die Begriffe „prozedural" und „imperativ" werden oft synonym verwendet, wobei „prozedural" die Abfolge der Programmbefehle hervorhebt, während „imperativ" die Veränderung des Programmzustands im Vordergrund sieht.

Prominentes Beispiel einer deklarativen Sprache ist SQL. SQL beschreibt, welche Spalten welcher Tabellen in welcher Reihenfolge und Kombination ermittelt werden sollen. Die Datenbank ist frei, den besten Weg zur Ermittlung dieses Ergebnisses zu finden.

Ähnlich ist der Schritt von Servlets zu JSPs: An die Stelle von Java-Code, der in Methoden und Klassen den Weg zur Erzeugung einer HTML-Seite beschreibt, traten Seiten, die die Struktur der zu erzeugenden HMTL-Seiten darstellen. Diese Entwicklung wurde schrittweise verfeinert: Zunächst wurden innerhalb der JSPs prozedurale Elemente (Java Scriptlets) verwendet, um Bedingungen und Schleifen darzustellen. Diese wurden durch die JSTL Tags *c:if*, *c:forEach* und weitere abgelöst, die deklarativ aussehen, aber im Kern immer noch prozedurale Strukturen abbilden. Mit JSF wurde hier der vorläufige Höhepunkt erreicht, indem auch für Iterationen und konditionale Darstellungen (durch *h:dataTable* und das *rendered*-Attribut) deklarative Strukturen eingeführt wurden.

Mit Facelets ist es nun möglich, diesen Schritt auch für die Tag-Entwicklung zu gehen und im Tag zu beschreiben, wie das Ergebnis aussehen soll, anstatt in Methoden den Weg zum Ergebnis zu codieren. Weitere Beispiele für deklarative Strukturen sind Java Annotations, XML-Konfigurationsdateien und natürlich HTML. In all diesen Fällen ergänzen sich deklarative und prozedurale Elemente. Im Falle von SQL gibt es z.B. PL/SQL und für HTML gibt es JavaScript als prozedurale Ergänzung einer deklarativen Sprache. Bei Annotations und XML-Konfigurationen ist es umgekehrt: Hier ergänzen deklarative Elemente die prozedurale Sprache.

nicht mehr prozeduraler Tag-Code geschrieben werden, der seinerseits dann wieder unleserlichen JavaScript-Code rendert. Und man erspart sich das leidige Escapen von Sonderzeichen in HTML- bzw. JavaScript-Code, der über einen Java String in einer *Renderer*-Klasse auf den *ResponseWriter* geschrieben wird. Das JavaScript kann direkt in das deklarative Tag hineingeschrieben werden.

Es gibt eine zentrale Problematik bei der Kombination von JSF und JavaScript. Um HTML-Elemente über JavaScript anzusprechen, ist es notwendig, ihre ID zu kennen. Mithilfe der JavaScript-Anweisung *document.getElementById()* kann dann der DOM-Knoten angesprochen werden, um seine Attribute zu verändern. Allerdings werden mit JSF die HTML-IDs vom Framework generiert und ihr Wert ist von der Position des Elements auf der Seite und von der JSF-Implementierung, die verwendet wird, abhängig. Mithilfe des Script Binding Patterns [1] ist es jedoch möglich, die HTML-ID in den JavaScript-Code einzufügen. Die Managed Bean *scriptHelper* referenziert dazu eine Instanz der Klasse *ScriptHelperBean* mit Scope Request. Die *ScriptHelperBean* bietet folgenden

Listing 1

```
<!-- Hier beginnt die kleine HTML Table für
                        Eingabefeld und Klappknopf -->

<h:panelGrid columns="2" styleClass="#{topClass}"
columnClasses="#{fieldCellClass}, #{imageCellClass}">
<h:inputText id="inputField" value="#{inputValue}"
styleClass="#{fieldClass}" readonly="#
                              {disabled || selectOnly}" />
  <h:panelGroup>
   <h:graphicImage id="imgPlus" url="#{imagePlus}"
styleClass="#{imageClass}" />
   <h:graphicImage id="imgMinus" url="#{imageMinus}"
style="display:none;" styleClass="#{imageClass}" />
  </h:panelGroup>
 </h:inputText>
</h:panelGrid>
<!-- Hier kommt die Tabelle mit der Listenauswahl -->
<table style="position:absolute;visibility:hidden"
                                  class="#{bottomClass}"
id="J4Fry_Combo_Table_#{name}">
<tbody>
 <c:set var="J4Fry_table_row" value="#{0}" />
 <c:forEach items="#{dataList}" var="item">
  <tr class="#{scriptHelper.rowClass[rowClasses]
                                  [J4Fry_table_row]}"
id="J4FryTable_For_#{name}_Row_
                                  #{J4Fry_table_row}" >
   <ui:insert name="tr" />
  </tr>
  <c:set var="J4Fry_table_row" value="
                              #{J4Fry_table_row + 1}" />
 </c:forEach>
</tbody>
</table>
<!-- hiddenFields mit rowKeys für jede Zeile
                                  hinzufügen -->
<c:set var="J4Fry_table_row" value="#{0}" />
<c:forEach items="#{dataList}" var="item">
 <span id='J4Fry_Span_For_#{name}_Row_
                                  #{J4Fry_table_row}'
style="display:none;visibility:hidden;">
<ui:insert name="rowKey" />
</span>
 <c:set var="J4Fry_table_row" value="
                              #{J4Fry_table_row + 1}" />
</c:forEach>
```

Listing 2

```
<script type="text/javascript" language="JavaScript">
 var J4Fry_activeKey#{name} = null;
 var J4Fry_activeRow#{name} = 0;
 var J4Fry_selectionType#{name} = null;
 function showJ4FryTable#{name}() {
  var inputField = document.getElementById(
'#{scriptHelper.clientIdForBindings[inputField]}');
  var parentElement = inputField.offsetParent;

  var topPosition = inputField.offsetTop;
  while(parentElement != null) {
   topPosition += parentElement.offsetTop;
   parentElement = parentElement.offsetParent;
  }

  var leftPosition = inputField.offsetLeft;
  parentElement = inputField.offsetParent;
  while(parentElement != null) {
   leftPosition += parentElement.offsetLeft;
   parentElement = parentElement.offsetParent;
  }

  var dTable = document.getElementById
                        ('J4Fry_Combo_Table_#{name}');
  dTable.style.top = topPosition + inputField.
                                    offsetHeight;
  dTable.style.left = leftPosition;
  dTable.style.visibility='visible';
  document.getElementById('#{scriptHelper
.clientIdForBindings[imgPlus]}').style.display='none';
  document.getElementById('#{scriptHelper
.clientIdForBindings[imgMinus]}').style.display='';
 }

 function hideJ4FryTable#{name}() {
  document.getElementById
                        ('J4Fry_Combo_Table_#{name}')
.style.visibility='hidden';
  document.getElementById('#{scriptHelper
.clientIdForBindings[imgPlus]}').style.display='';
  document.getElementById('#{scriptHelper
.clientIdForBindings[imgMinus]}').style.display='none';
 }

 function scrollJ4FryTable#{name}(keyCode) {
  if (keyCode == 38 || keyCode == 40) {

   var newActiveRowNo = J4Fry_activeRow#{name};
   if (J4Fry_activeKey#{name} != null) {
    if (keyCode == 38) {
     if (J4Fry_activeRow#{name} > 0) {
      var newActiveRowNo
= J4Fry_activeRow#{name} - 1;
     }
    } else {
     var newActiveRowNo
= J4Fry_activeRow#{name} + 1;
     }
    }
    highlightJ4FryTableRow#{name}(newActiveRowNo);
    J4Fry_selctionType#{name} = 'key';
   } else if (keyCode == 13) {
    document.getElementById('#{scriptHelper
.clientIdForBindings[inputField]}').value
= J4Fry_activeKey#{name};
    var oldActiveRow = document.getElementById(
'J4FryTable_For_#{name}_Row_'+J4Fry_activeRow
                                    #{name});
    if (oldActiveRow.className == 'rollOverEffect') {
     oldActiveRow.className = oldActiveRow.origActCl;
    }
     J4Fry_activeKey#{name} = null;
    J4Fry_activeRow#{name} = 0;
    document.getElementById('#{scriptHelper
.clientIdForBindings[inputField]}').onchange();
    hideJ4FryTable#{name}();
   } else {
    J4Fry_activeKey#{name}=null;
   }
  }

  function highlightJ4FryTableRow#{name}(rowNo) {
   var oldActiveRow = document.getElementById(
'J4FryTable_For_#{name}_Row_'+J4Fry_activeRow
                                    #{name});
   var newActiveRow = document.getElementById(
'J4FryTable_For_#{name}_Row_'+rowNo);
   if (newActiveRow != null) {
    if (newActiveRow.nodeName != null) {
     if (newActiveRow.nodeName.toLowerCase() == 'tr') {
      J4Fry_activeRow#{name} = rowNo;
      J4Fry_activeKey#{name}
= document.getElementById(
'J4Fry_Span_For_#{name}_Row_' +
rowNo).innerHTML;
      if (oldActiveRow.className=='rollOverEffect') {
       oldActiveRow.className
= oldActiveRow.origActCl;
      }
      newActiveRow.origActCl=newActiveRow.className;
      newActiveRow.className='rollOverEffect';
     }
    }
   }
  }

</script>
```

Mechanismus: Jede Komponente, die per JavaScript angesprochen werden soll, legt eine Referenz auf die zu ihr gehörige UIComponent in eine Map namens *bindings*: *binding="#{scriptHelper.bindings['myComponent']}"*. Der Zugriff auf die *clientId* (das ist die HTML-ID) der UIComponent kann mit *component.getClientId(context)* erfolgen, allerdings ist über die JSF-EL kein parametrisierter Zugriff möglich. Für die notwendige Versorgung mit dem *FacesContext* zur Erzeugung der *clientId* sorgt die *ScriptHelperBean*. Der Zugriff auf *#{scriptHelper.clientIdForBindings['myComponent']}* greift die *ScriptHelperBean* in die *bindings*-Map und gibt zur dort gefundenen UIComponent die *clientId* heraus. Eine der Stärken von Facelets ist, dass die benötigten JavaScripts einfach in das Tag Template aufgenommen und die Ausdrücke zum Zugriff auf die *clientIds* direkt in die Scripts integriert werden können.

Ein Hindernis gilt es noch zu überwinden: Wenn mehrere Combobox Tags auf derselben JSF-Seite verwendet werden, sollen sie unterschiedliche *bindings* verwenden. Um dies zu erreichen, muss der *#{name}*-Parameter des Tags in den Zugriff auf die *bindings*-Map hinein genommen werden. Der direkte Ansatz *#{scriptHelper.bindings['myComponent'name]}* schlägt aber fehl: Der String Key für den Map-Zugriff kann nicht aus zwei Strings konkateniert werden. Stattdessen verwenden wir das JSTL-Tag *c:set*, um Variablennamen zu generieren, die im Laufe des weiteren Renderings den Zugriff auf die UIComponent-*bindings* der *ScriptHelperBean* ermöglichen.

```
<c:set var="inputField" value="inputField#{name}"
/>
<c:set var="imgPlus" value="imgPlus#{name}" />
<c:set var="imgMinus" value="imgMinus#{name}" />
```

Ähnlich splittet der Ausdruck *scriptHelper.rowClass[rowClasses][J4Fry_table_row]}* eine kommaseparierte Liste von Zeilenklassen und liefert Style Class zurück, die durch *J4Fry_table_row* bezeichnet wird. Die Methode *public Map getRowClass(Object rowClasses)* liefert dazu ein Objekt, das *java.util.Map* implementiert und sich den kommaseparierten String *rowClasses* merkt. Wenn die Methode *public Object get(Object rowNum)* dieses Objekts aufgerufen wird, castet sie *rowNum* in eine Integer, splittet *rowClasses* und liefert die durch *rowNum* bezeichnete *rowClass* zurück.

Die JavaScripts

Um das Öffnen und Schließen der Listenauswahl und die Auswahl von Zeilen durch Mausklick und mit Pfeiltasten zu ermöglichen, sind ein paar JavaScripts nötig (Listing 2). Damit einerseits diese Scripts automatisch durch das Tag eingefügt werden können und andererseits jede auf einer Seite vorkommende Combobox unterschiedliche Variablen- und Funktionsnamen verwendet, ist es notwendig, dass jede Combobox über den Tag-Parameter *#{name}* einen eigenen Namen erhält. Dieser Name wird an alle globalen JavaScript-Variablen- und -Funktionsnamen angehängt und erzeugt so die Eindeutigkeit.
In der Variablen *J4Fry_activeKey #{name}* wird der String gespeichert, der in das Eingabefeld übernommen werden soll, falls die Zeile geklickt oder die ENTER-Taste gedrückt wird. Die Variable *J4Fry_activeRow#{name}* speichert die Zeilennummer der Zeile, die gerade ausgewählt und farblich unterlegt ist. Im *J4Fry_selectionType#{name}* ist festgehalten, ob die Zeilenauswahl gerade per Maus (Wert: *mouse*), per Pfeiltasten (Wert: *key*) oder gar nicht (wegen zugeklappter Tabelle, Wert: *null*) erfolgt.

Es sind vier JavaScript-Funktionen definiert: *showJ4FryTable#{name}()* blendet die Tabelle für die Zeilenauswahl ein und *hideJ4FryTable#{name}()* blendet sie wieder aus. Die beiden Funktionen werden natürlich vom Auf-/Zuklappknopf aufgerufen. Außerdem wird das Einblenden aufgerufen, wenn im Textfeld eine Taste gedrückt wird. Wenn die ENTER-Taste gedrückt wird, wird der Wert aus *J4Fry_activeKey#{name}* in das Eingabefeld übernommen und die Funktion zum Ausblenden der Tabelle aufgerufen. Zum Ein- und Ausblenden der Tabelle wird der CSS Style *visibility* (*visible* und *hidden*) und zur abwechselnden Anzeige der Auf- und Zuklappknöpfe der CSS

Listing 3

```
onkeydown="if (!('#{disabled}' == 'true')) {

 showJ4FryTable#{name}();
scrollJ4FryTable#{name}(event.keyCode);
}"
onblur="if (!('#{disabled}' == 'true')) {
if (J4Fry_activeKey#{name} != null) {
  if (J4Fry_selectionType#{name} == 'mouse') {
   this.value = J4Fry_activeKey#{name};
   var oldActiveRow =
document.getElementById(
'J4FryTable_For_#{name}_Row_'+
J4Fry_activeRow#{name});
   if (oldActiveRow.className == 'rollOverEffect') {
    oldActiveRow.className=oldActiveRow.origActCl;
   }
   J4Fry_activeKey#{name} = null;
   J4Fry_activeRow#{name} = 0;
   this.onchange();
  }
 }
 hideJ4FryTable#{name}();
}"
onchange="var J4Fry_Dummy_For_Combo = 0;"
                                        J4Fry_activeKey#{name} = null;

   J4Fry_activeRow#{name} = 0;
   this.onchange();
  }
 }
 hideJ4FryTable#{name}();
}"
onchange="var J4Fry_Dummy_For_Combo = 0;"
```

Listing 4

```
<fry:combo name="combobox1" inputValue="#{comboBean.value1}"
dataList="#{comboBean.veges}" fieldClass="comboField"
imageClass="comboImg"
rowClasses="odd, even" bottomClass="comboTable"
                                                topClass="comboTable"
fieldCellClass="top1" imageCellClass="top2"
imagePlus="J4Fry/Resource/j4fry_plus.jpg"
imageMinus="J4Fry/Resource/j4fry_minus.jpg">
<ui:define name="fieldExtension">
 <fry:ajax event="onchange" reRender="combo2" action=
                                              "#{comboBean.refresh2}"
loadingbarId="#{scriptHelper.clientIdForBindings['loadingBar']}"
alarmThreshold="error" />
</ui:define>
<ui:define name="rowKey">#{item.name}</ui:define>
<ui:define name="tr">
 <td class="col2">
  <h:outputText value="#{item.name}" />
 </td>
 <td class="col1">
  <h:outputText value="#{item.weight}" converter="org.j4fry.Number">
   <f:attribute name="Pattern" value="#,##0.00#" />
   <f:attribute name="Language" value="de" />
   <f:attribute name="Country" value="DE" />
  </h:outputText>
 </td>
</ui:define>
</fry:combo>
```

Style *display* („ “ oder *none*) verwendet. In der Methode zum Einblenden ist auch der Trick versteckt, der das Ganze optisch zu einem zusammengehörigen Element werden lässt: Beim Aufklappen wird die Tabelle relativ zum Input-Feld positioniert. Dies ist möglich, weil die Tabelle zu Beginn den Style *position:absolute* erhalten hat.

highlightJ4FryTableRow#{name}() wird jedes Mal aufgerufen, wenn die ausgewählte und farblich unterlegte Zeile wechselt. Die geschieht beim Überfahren einer Zeile mit dem Mauszeiger und beim Drücken der PFEIL-HOCH- oder PFEIL-RUNTER-Tasten. Beim Tippen einer Taste wird *scrollJ4FryTable#{name}(keyCode)* aufgerufen, um festzustellen, ob auf PFEIL HOCH/PFEIL RUNTER/ENTER reagiert werden muss. Dabei wird auf die *keyCodes 13* (ENTER), *38* (PFEIL HOCH) und *40* (PFEIL RUNTER) reagiert.

Die JavaScript-Events

Nun sollen noch die JavaScripts an die Oberflächenelemente gebunden werden. Das Eingabefeld reagiert auf die Events *onkeydown*, *onblur* und *onchange* (Listing 3). Die Events werden nur ausgeführt, wenn nicht *true* im Tag-Attribut *disabled* übergeben wurde. *onkeydown* blendet die Tabelle ein und ruft *scrollJ4FryTable* auf, damit auf PFEIL- und ENTER-Tasten reagiert wird. Das *onblur*-Event wird aufgerufen, wenn das Eingabefeld den Fokus verliert. Dies geschieht, wenn man mit der Maus außerhalb des Eingabefelds klickt und wenn man im Eingabefeld eine Taste drückt, die den Fokus auf ein anderes Feld überträgt. Wenn gerade eine Zeile durch *mouseover* ausgewählt wurde, soll *onblur* den Wert dieser Zeile übernehmen. Das lässt sich auch mithilfe des *onclick*-Events der Tabellenzeile implementieren, aber weil *onblur* die Tabelle ausblenden soll, wird der *onclick*-Event der Tabelle nicht ausgelöst: Die ist dann schon verschwunden. Das *onchange*-Event enthält nur Dummy-Code, der notwendig ist, weil der *this.onchange()*-Aufruf des *onblur*-Events sonst zu Fehlern führt, solange kein Ajax-Event am *onchange* hängt. Der Aufruf von *this.onchange()* durch den *onblur*-Event ist aber wünschenswert, damit ein Ajax-Event, der am *onchange* des Eingabefelds hängt, auch ausgelöst wird, wenn der Wert durch *onblur* geändert wird.

Der Knopf zum Einblenden blendet beim Klicken die Tabelle ein und setzt den Fokus auf das Eingabefeld:

```
onclick="if (!('#{disabled}' == 'true')) {
 showJ4FryTable#{name}();
 document.getElementById('#{scriptHelper.
                clientIdForBindings[inputField]}')
.focus();
}"
```

Der Knopf zum Ausblenden blendet die Tabelle aus und setzt den ausgewählten Schlüssel und die Zeilennummer zurück:

```
onclick="if (!('#{disabled}' == 'true')) {
 hideJ4FryTable#{name}();
 J4Fry_activeKey#{name} = null;
 J4Fry_activeRow#{name} = 0;
}"
```

An den TR-Elementen der Zeilenauswahltabelle hängen *onmouseover*- und *onmouseout*-Events. Durch *onmouseover* wird *highlightJ4FryTableRow#{name}()* mit der Zeilennummer aufgerufen und der *J4Fry_selectionType#{name}* auf *mouse* gesetzt.

```
onmouseover="highlightJ4FryTableRow#{name}
                                (#{J4Fry_table_row});
J4Fry_selectionType#{name} = 'mouse';"
```

Durch *onmouseout* werden Style, Schlüssel, aktive Nummer und *selectionType* wieder zurückgesetzt:

```
onmouseout="var oldActiveRow = document.getEle-
mentById(
'J4FryTable_For_#{name}_Row_'+J4Fry_
activeRow#{name});
if (oldActiveRow.className == 'rollOverEffect')
oldActiveRow.className=oldActiveRow.origActCl;
 J4Fry_activeKey#{name}=null;
J4Fry_activeRow#{name}=0;
J4Fry_selectionType#{name} = null;"
```

Durch Schachteln von

```
<ui:define name="fieldExtension">
... Code, der hier eingefügt wird, wird in das
h:inputText des Eingabefelds geschachtelt
</ui:define>
```

in das Combo Tag, kann das Eingabefeld um Ajax-Funktionalität erweitert werden. Eine Bibliothek, die Ajax mit JSF 1.1, JSF 1.2, Facelets und JSR 168 Portlets unterstützt, ist J4Fry [2]. In Listing 4 findet sich ein Beispiel, wie das Tag innerhalb einer Anwendung verwendet werden kann, die Livedemo kann online [3] eingesehen werden und der Quellcode steht unter www.javamagazin.de zur Verfügung. Livedemo und Quellcode zeigen, wie sich mit dem Combo Tag zusätzlich zu den hier beschriebenen Features eine *Autosuggest*-Komponente realisieren lässt.

Das Tag mit Facelets verpacken

Das so in XML formulierte Tag lässt sich mit Facelets leicht in einem *jar* ausliefern. Dazu erstellt man im *META-INF*-Folder des *jar*-Files eine Datei, die auf *.taglib.xml* endet. In dieser beschreibt man das Tag, indem man dort auf die XHTML-Datei verweist. Wenn die Datei unter *META-INF/combo/combo.xhtml* liegt und das Tag „combo" heißen soll, sieht die Tag-Beschreibung folgendermaßen aus:

```
<tag>
 <tag-name>combo</tag-name>
 <source>combo/combo.xhtml</source>
</tag>
```

Beim Start einer Facelets-Anwendung durchsucht Facelets automatisch alle auf dem Classpath gelegenen Bibliotheken nach solchen *.taglib.xmls* und stellt diese der Anwendung zur Verfügung.

Fazit: Aufbau einer Tag Library

JSF 2.0 ist auf dem Weg zu uns. Viele Anwendungen, die Komponentenbibliotheken nutzen, werden erhebliche Schwierigkeiten haben, auf JSF 2.0 zu migrieren. Diese Bibliotheken werden einige Zeit brauchen, bis sie auf die neue API umgestellt haben. Tags, die auf der deklarativen Technologie aufsetzen, werden, anders als die prozeduralen Bibliotheken, keine Migrationsprobleme haben. Auch die Umstellung auf weitere zukünftige JSF-Versionen wird für eine deklarative Bibliothek um Vieles leichter und schneller gehen. Und mehr noch: Facelets werden zum großen Teil in den Standard übernommen, was die Erzeugung deklarativer Tags noch eleganter auf den Standard aufsetzen lässt. Unter [4] werden einige Tags beschrieben, die letztendlich in einer eigenen Tab Library zusammengefasst zur Verfügung gestellt werden sollen. Jeder ist dort eingeladen, weitere Tag-Vorschläge unter der Apache-Lizenz 2.0 zu machen oder sich an der Entwicklung der Tag Library zu beteiligen. Dabei werden wir gerne helfen, die Winkel von JSF, JavaScript, HTML und CSS zu durchleuchten.

Ganesh Jung ist freiberuflicher Softwareentwickler, hat die Architekturen von mehreren JSF-Anwendungen für eine deutsche Großbank entworfen und an deren Entwicklung mitgearbeitet. In seiner Freizeit unterstützt er die Open Source Community www.j4fry.org, deren Mitglieder an diesem Artikel unterstützend mitgewirkt haben.

Links & Literatur

[1] Die Pattern zur Entwicklung mit Standard-JSF: www.j4fry.org/resources/jung_JSF_JavaMagazin.pdf

[2] Vielseitiges Ajax Tag: www.j4fry.org/jsjAjax.shtml

[3] Livedemo des Combo Tags: www.j4fry.org/JSFExamples/faces/facelets/combos.xhtml

[4] Fragen zum Artikel: https://sourceforge.net/forum/message.php?msg_id=5164855