Android wird andersARTig: Technische Evolution der Plattform ▸108

S&S

JAVA Mag

8.2014

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

BDD mit JBehave
Verhaltensregeln für Anwendungen ▸101

JSF und Bootstrap
Frischzellenkur für Web-Apps ▸86

MICRO SERVICES

Löst die Silver Bullet unsere Probleme? ▸29

Nie wieder M

JavaFX-Tutorial: Von der Klasse zur EXE ▸54

Java 8 Streams: Erzeugung und Operationen ▸16

Mit OpenStack zur eigenen Cloud: Privatsache ▸95

Moderne Weboberflächen mit JavaServer Faces und Bootstrap

Frischzellenkur

JSF erfreut sich einer starken Verbreitung. Auch wenn die Aufnahme in den Java-EE-Stack ihren Teil dazu beigetragen hat, so war dies sicher nicht der einzige Grund für den Erfolg. Die Stärken sind seit jeher die umfangreiche Komponentenbibliotheken sowie die einfache Anbindung von Geschäftslogik an das Frontend. In der Praxis hat sich jedoch häufig gezeigt, dass sich die Bibliotheken als zu schwergewichtig erwiesen haben und das Design von modernen Weboberflächen aufwändig werden konnte. Die Anpassung gemäß den Vorgaben der Fachabteilungen und der Corporate Identity hat schon vielen Entwicklern einiges an Zeit und Nerven gekostet.

von Christian Laboranowitsch und Philipp Bayer

Im folgenden Artikel wollen wir zeigen, wie mit JSF und Bootstrap moderne Weboberflächen entwickelt werden können, die optisch ansprechend sind und Nutzern eine gelungene „User Experience“ bieten. Auch wenn es in der Zukunft noch zahlreiche Herausforderungen geben wird, so denken wir, dass es mit den Neuerungen von JSF 2.2 möglich ist, modernere und leichtgewichtigere Frameworks für Weboberflächen mit JSF zu verknüpfen.

„Mobile First“: Rette sich wer kann. Kleine Devices zuerst!

Die Entwicklung mit dem Bootstrap-Framework erleichtert die Entwicklung von Webseiten, egal um welches Endgerät es sich handelt. Ursprünglich von Twitter entwickelt, ist es seit August 2011 als Open-Source-Projekt auf GitHub für die Öffentlichkeit verfügbar. Über die Jahre hinweg wurde es aktiv weiterentwickelt und ist mittlerweile eines der beliebtesten Projekte auf GitHub. Neben der Unterstützung der gängigsten Browser war der Fokus von Bootstrap schon immer die Unterstützung von verschiedenen Endgeräten. Während sich schon Bootstrap 2 den Responsive-Design-Ansatz auf die Fahne geschrieben hatte, ist dieser Aspekt in Bootstrap 3 mit seinem „Mobile-First-Ansatz“ stärker in den Vordergrund gerückt. „Mobile First“ bedeutet nichts anderes als das Webseitendesign auf die Erfordernisse der mobilen Endgeräte auszurichten und auf die klassischen Formate (Desktop) zu skalieren. Neben einer Vielzahl spezieller Komponenten bietet Bootstrap auch ein spezifisches Layoutsystem, das es ermöglicht, Layouts zu erstellen, die sich den Fähigkeiten der Endgeräte anpassen. Analog zu Javas „Write once, run anywhere“

verspricht Bootstrap die zielgerichtete Darstellung von Webseiten auf allen Endgeräten.

„Smart Grids“ mal anders: Layout mit Bootstrap

Ein zentraler Bestandteil von Bootstrap ist das Grid-Layout. Es ist vergleichbar mit dem Gridbag-Layout von Swing. Inhalte werden in Zellen platziert, die sich über mehrere Spalten erstrecken können. Über die Anzahl der Spalten wird gesteuert, wie viel Platz Inhalte in Relation zueinander einnehmen. Per Default besteht das Grid aus zwölf Spalten, es ist aber auch möglich, diesen Wert zu ändern.

Die Verwendung des Grid-Layouts folgt ein paar einfachen Regeln. Die grundlegende Idee ist, dass mittels CSS-Klassen beschrieben wird, welche Rolle ein HTML-Element im Grid-Layout hat. Zeilen und Spalten erzeugt man mittels der *.row-* und *.col*-CSS-Klassen. Die Zahl am Ende des Namens der *.col*-Klasse steuert die Anzahl der Spalten, die das Element einnimmt. In unserem ersten Beispiel würde der *div*-Container mit der ID „first“ drei Viertel und der *div*-Container mit der ID „second“ ein Viertel des zur Verfügung stehenden Platzes bekommen. Wichtig dabei ist, dass die Summe der *.col*-Klassen immer zwölf ergibt und somit immer die volle Breite des Grids genutzt wird:

```
<div class="row">
  <div id="first" class="col-md-9">...</div>
  <div id="second" class="col-md-3">...</div>
</div>
```

Zusätzlich wird mit dem mittleren Teil des Klassennamens (*col*-**md**-*9*) die Größe des Zielendgeräts definiert (*xs* steht für Extra Small Devices (< 768 px Breite), *sm* für Small Devcices (< 992 px Breite), *md* für Medium Devices (< 1200 px Breite) und *lg* für Large Devices (≥ 1200 px Breite)). **Abbildung 1** zeigt die Verwendung der Layoutklassen und die resultierende Aufteilung. Falls das Endgerät eine Auflösung größer oder gleich des Zielendgeräts hat, wird das Layout wie gewünscht angezeigt und die Spalten skalieren, um den eventuell noch vorhandenen Platz zu nutzen. Falls das Endgerät eine kleinere Auflösung hat, wird das Layout umgebrochen. In diesem Fall werden die einzelnen Spalten direkt untereinander dargestellt und nehmen die gesamte Bildschirmbreite des Endgeräts ein. Zwischen den einzelnen Zeilen wird ein Abstand eingefügt. **Abbildung 2** zeigt die Darstellung des Beispiels auf einem Endgerät mit einer geringeren Auflösung.

Über die Angabe alternativer Zielendgeräte lässt sich das Layout für verschiedene Auflösungen anpassen. Dies ist immer dann sinnvoll, wenn auf verschiedenen Endgeräten unterschiedliche Platzverteilungen gewünscht sind. Dafür bekommt jedes Spaltenelement eine eigene *.col*-Klasse je Zielendgerät. In Listing 1 wird in der Desktopdarstellung der Platz gleichmäßig auf alle drei Spalten verteilt. Auf einem Mobiltelefon wird die erste Spalte die Hälfte der zur Verfügung stehenden Breite

Abb. 1: Bootstrap Grid

Abb. 2: Bootstrap Grid mit Umbruch

Abb. 3: Navigationsbarbar mit allen Menüpunkten

Abb. 4: Navigationsbar in zusammengeklapptem Zustand

einnehmen und die anderen beiden Spalten jeweils ein Viertel. Solche Anpassungen sind oftmals hilfreich, um trotz des beschränkten Platzes auf kleineren Endgeräten eine ansprechende Darstellung zu erreichen.

Auf der Bootstrap-Seite findet sich zum Grid-Layout eine ausführliche Anleitung mit vielen Beispielen [1].

Leichtgewichtsweltmeister: Bootstrap-Komponenten

Bootstrap bietet neben einem ausgefeilten Layoutmanagement auch eine Vielzahl an Komponenten. Neben optisch ansprechenden Standardkomponenten wie Buttons oder Labels erweitert die Bootstrap-Bibliothek die Werkzeugkiste des Programmierers um hochwertige Komponenten wie z. B. eine Navigationsbar. Diese Komponenten bringen zusätzliche Funktionalitäten mit und passen sich dem Endgerät an. So wird aus der Navigationsbar auf mobilen Endgeräten ein Button mit einem Drop-down-Menü, auf größeren Endgeräten die einzelnen Menüpunkte in der Menüleiste dargestellt (**Abb. 3** und **4**).

Add-ons für Inputelemente sind einfache Erweiterungen, mit denen Eingabefelder optisch aufgewertet werden können. Sie haben eine ähnliche Funktion wie Labels und beschreiben Eingabefelder genauer. Daneben können auch funktionale Elemente, wie Buttons oder

Listing 1: Bootstrap-Grid-Layout

```
<div class="row">
  <div class="col-xs-6 col-md-4">...</div>
  <div class="col-xs-3 col-md-4">...</div>
  <div class="col-xs-3 col-md-4">...</div>
</div>
```

Abb. 5: Ein paar Inputelemente mit verschiedenen Add-ons

Checkboxen in Add-ons eingebettet werden. Den Möglichkeiten sind (fast) keine Grenzen gesetzt. **Abbildung 5** zeigt einige Beispiele zur Verwendung von Add-ons.

Bei der Pagination handelt es sich um eine Komponente zur Erweiterung der Navigation zum Blättern. Neben den Forward- und Back-Buttons ist dies die Seitennummerndarstellung. Mittels der *.active*-Klasse können die aktive Seite markiert und mittels der *disabled*-Klasse einzelne Elemente deaktiviert werden. Anzumerken ist, dass bei der Pagination der Fokus auf der grafischen Darstellung liegt, die eigentliche Funktionalität des Paginierens ist durch den Entwickler zu implementieren. Später folgt ein konkretes Beispiel mit JSF.

Mittels Badges lassen sich eine Vielzahl von Elementen mit Zusatzinformationen dekorieren. Übliche Anwendungen sind das Anzeigen der Anzahl von Einträgen in einer Liste oder das Anzeigen der Reputation eines Diskussionsteilnehmers. Als zusätzliches Feature werden Badges nicht angezeigt, wenn das zugehörige *span*-Element leer ist.

Nicht zuletzt bietet Bootstrap mit den Glyphicons eine Bibliothek mit 200 Icons zur Nutzung in eigenen Anwendungen – eine Dienstleistung, deren kostenfreie Verfügbarkeit andernorts nicht selbstverständlich ist.

LESS: Weniger ist mehr

Bootstrap verwendet LESS, um die Arbeit mit CSS zu vereinfachen. Da Bootstrap komplett mit LESS geschrieben wurde, werden wir einige Vorzüge aufzeigen. LESS ist eine Obersprache von CSS, die in reine CSS-Dateien kompiliert wird. LESS steht dabei für „Leaner CSS“ und genau das ist das. Durch die Verwendung soll das Schreiben von CSS Style Sheets vereinfacht und der benötigte Code verschlankt werden. Dazu bietet LESS eine Reihe von Erweiterungen an. So ist es möglich, mittels des „@“-Symbols Variablen zu definieren. Damit können mehrfach verwendete Werte an zentraler Stelle als Variable definiert werden. Dadurch ist es möglich, diese Werte an einer Stelle zu ändern, ohne das sonst obligatorische „Search & Replace“ zu verwenden, wie es sonst bei CSS Style Sheets erforderlich wäre. Außerdem können die Werte von zuvor definierten Variablen überschrieben werden, wodurch sich das CSS von Bootstrap sehr leicht anpassen lässt. Durch Überschreiben des Werts der *@grid-columns*-Variablen kann man beispielsweise die Anzahl der Spalten im Grid-Layout vom Default-Wert zwölf auf einen anderen Wert ändern. Und mittels der *@brand-primary*-Variablen ist die Anpassung, beispielweise der Primärfarben, an die Corporate Identity möglich.

Daneben bietet LESS die Möglichkeit, so genannte Mixins zu definieren und zu nutzen. Bei Mixins handelt es sich um ein Gebilde ähnlich einer Funktion, die an den aktuellen CSS-Selektor eine Reihe CSS Properties anfügt. Sowohl LESS selbst als auch Bootstrap enthalten vordefinierte Mixins. Das folgende Listing zeigt, wie mittels Mixins einem Selektor eine rechteckige Größe von 15 Pixeln zugewiesen wird. Als Parameter für Mixins können auch definierte Variablen verwendet werden. Beim Aufruf von parameterlosen Mixins sind die Klammern optional. Der Aufruf von *.clearfix();* ist somit identisch zu dem Aufruf von *.clearfix;*:

```
.savebtn{
  .square(15px);
  .clearfix;
  background-color: blue;
}
```

Grundsätzlich ist jeder Klassenselektor auch ein Mixin; entsprechend lässt sich aus mehreren Klassen ein Style zusammenbauen. Im folgenden Listing wird gezeigt, wie eine eigene Style-Klasse aus mehreren Bootstrap-Styles erzeugt werden kann. Dies erspart nicht nur Aufwand, weil man nur noch eine Style-Klasse statt derer vier verwenden muss, sondern erleichtert auch spätere Änderungen:

```
.myAlert{
  .alert;
  .alert-danger;
  .alert-dismissable;
  max-width: 15%;
}
```

Tausend Mal berührt: JSF 2.2 und HTML5

Die fortschreitende Unterstützung von HTML5 durch JSF 2 verdeutlicht den Stellenwert, den es mittlerweile erreicht hat. So werden mit JSF 2.2 Facelets standardmäßig als HTML5-Dokumente gerendert, und zwar unabhän-

Listing 2: Verwendung von Passthrough-Attributen

```
<html ... xmlns:p="http://xmlns.jcp.org/jsf/passthrough">
  ...
<label for="nameInput" class="btn btn-default" p:data-toggle"tooltip"
                    p:data-placement="bottom" p:title="Bitte geben Sie Ihren Namen ein">
  Name:
</label>
  ...
```

Listing 3: Bootstrap-Button mit JSF-Action

```
<html ..... xmlns:jsf="http://xmlns.jcp.org/jsf">
  ...
  <button type="button" class="btn btn-default " jsf:action="#{myBean.save}">
    <span class="glyphicon glyphicon-floppy-save"></span> Speichern
  </button>
  ...
</html>
```

gig davon, welcher Doctype im XHTML definiert wurde. Allerdings genügen diese Änderungen alleine noch nicht, um moderne HTML5-Webseiten mit JSF zu entwickeln.

Ein bedeutendes HTML5-Feature, das auch von Bootstrap verwendet wird, besteht darin, für HTML-Elemente beliebige Attribute zu vergeben. Diese können dann ausgewertet und verarbeitet werden. Dieses Feature wird unter anderem von vielen JavaScript-Frameworks wie jQuery Mobile oder Knockout verwendet. Bislang war es nicht möglich, dieses Feature mit JSF-Komponenten zu verwenden, da der JSF Renderer ihm unbekannte Attribute verwirft. Folglich sind solche Attribute im gerenderten Markup nicht mehr vorhanden. Mit der neuen JSF-Spezifikation gibt es nun eine elegante Lösung für diese Herausforderung. Mittels des „Passthrough Namespace" kann man Attribute als solche kennzeichnen. Passthrough-Attribute werden vom Renderer ins fertige Markup durchgereicht und dabei nicht ausgewertet. Zu beachten ist dabei, dass der JSF Renderer weiterhin ihm unbekannte Attribute verwirft, wenn sie nicht das *p:*-Präfix haben (Listing 2).

Auch an anderer Stelle wurde der Renderer erweitert. So können jetzt HTML-Elemente mit JSF-Komponenten kombiniert werden. Diese so genannten Passthrough-Elemente sind gewöhnliche HTML-Elemente, die mit einer serverseitigen UIComponent verknüpft werden. Dadurch ist es möglich, das gerenderte Markup genau zu bestimmen und mit dem JSF-Komponentenmodell zu verheiraten. Sobald ein HTML-Element ein Attribut aus dem *jsf:* Namespace beinhaltet, wird es von JSF als Passthrough-Element erkannt und mit einer UIComponent verknüpft. Genauere Informationen, welche konkrete UIComponent erzeugt wird, kann man unter [2] finden. Sollte es kein direktes Mapping geben, erzeugt JSF die speziellen Elementkomponenten. Dadurch können sich allerdings die gültigen Werte von Attributen ändern. Beispielsweise hat das *disabled*-Attribut eines HTML-Inputelements als gültigen Wert *disabled* für das Mapping auf eine JSF-Komponente, muss dieser Wert zu *true* geändert werden:

```
<input jsf:id="jsfId" type="text" disabled="true" />      Richtig
<input jsf:id="jsfId" type="text" disabled="disabled" />  Falsch
```

Als Beispiel für die Verwendung von Passthrough-Elementen wird in Listing 3 die Nutzung von Bootstrap-Buttons gezeigt. Mit bisherigen JSF-Mitteln wäre dies nicht möglich gewesen, da der Renderer einen *h:commandButton* immer als *<input type="submit" … />* rendert. Mittels der neuen Passthrough-Elemente kann aber genau festgelegt werden, dass ein HTML-Button gerendert werden soll, um diesen dann mit der entsprechenden JSF-Action zu verknüpfen. Eine weitere Einsatzmöglichkeit von Passthrough-Elementen ist das Reagieren auf DOM-Events eines HTML-Elements mittels *f:ajax*. So könnte man in das Button-Element aus Listing 7 ein *f:ajax* einbetten. Hier ist aber Vorsicht geboten, *f:ajax* kennt nur die Events, die auch eine entsprechende JSF-Komponente besitzen. Eine Reaktion auf Custom-Events, die z. B. aus JavaScript-Bibliotheken stammen, ist auf diesem Wege nicht durchführbar.

Die hohe Schule: Bootstrap und JSF

Mit den Möglichkeiten von JSF 2.2 ist es möglich, Bootstrap mit JSF zu verbinden. So lassen sich beispielsweise die Bootstrap-*.label*-Klassen mittels des *styleClass*-Attributs an ein *h:outputLabel* anfügen. Für andere Komponenten ist dazu allerdings mehr Aufwand notwendig, auch wenn Bootstrap ein solides Grundgerüst an Komponenten mitbringt, enthalten diese aber keine Logik. So besteht beispielsweise bei der Pagination-Komponente die Möglichkeit, die aktuelle Seite zu markieren und die Forward- und Back-Buttons zu disablen, eine entsprechende Applikationslogik, um diese Mechanismen zu verwenden, muss selbstverständlich durch den Entwickler bereitgestellt werden. Hier bieten Frameworks wie PrimeFaces und RichFaces einen Mehrwert, da diese bereits Komponenten mitbringen, die eine solche Funktionalität nahtlos in den JSF-Programmieransatz integrieren.

Der Einsatz von JSF-Komponenten-Frameworks bietet allerdings nicht nur Vorteile. Gefürchtet unter Entwicklern sind Kompatibilitätsprobleme aufgrund unterschiedlicher Versionen der eingesetzten JavaScript-Frameworks. Ein weiterer Nachteil ist die enge Verzahnung mit den Releasezyklen der Komponentenentwickler. Diese sind zwar meist bemüht, mit der JSF-Entwicklung Schritt zu halten, wir mussten aber in der Vergangenheit selbst schmerzliche Wartezeiten in Kauf nehmen, um mit RichFaces den Umstieg auf JSF 2.0 vollziehen zu dürfen. Somit stellt die Implementierung mithilfe der Bootstrap-Komponenten eine echte Alternative dar.

Zu guter Letzt möchten wir die allseits beliebten Style Guides der „Corporate-Welt" nicht vernachlässigen. Unsere häufig gemachte Erfahrung ist, dass Marketingabteilungen die Grenzen der bekannten JSF-Komponentenbibliotheken nicht akzeptieren können. So wurde schon aus der sprichwörtlichen Mücke zum Zeitpunkt der Schätzung ein Elefant in der Umsetzung, sehr zum Leidwesen aller Beteiligter.

Die Integration an Beispielen

Im Folgenden finden sich einige Beispiele, die eine Integration zwischen Bootstrap und JSF zeigen. Eine komplette Beispielanwendung findet sich auf GitHub [3].

Der Start mit Bootstrap gestaltet sich einfach. Es müssen die Bootstrap-CSS- und JavaScript-Dateien sowie jQuery in die Seite eingebunden werden. Um mit LESS zu arbeiten, müssen zusätzlich die Bootstrap-LESS-Sourcen [4] geladen werden, um aus ihnen das CSS selbst zu kompilieren. Dafür muss in der eigenen LESS-Datei die *bootstrap.less*-Datei importiert werden. Mittels Maven lässt sich die LESS-Kompilierung auch in den Eclipse Build einbinden (Listing 4). Bei Änderungen der LESS-Datei wird dann auch gleich das entsprechende CSS neu kompiliert und landet im *Resources*-Ordner

der Webapplikation. Damit Bootstrap die Glyphicons findet, sollte in der LESS-Datei der Pfad angepasst werden: *@icon-font-path: "../resources/bootstrap/fonts/";*.

Ein grundsätzliches Setup des Bootstrap-Projekts ist damit abgeschlossen, und es findet sich eine Projektstruktur wie in **Abbildung 6**. Um nun Bootstrap in die JSF-Seiten einzubinden, sollte im JSF-Template ein Header wie in Listing 5 definiert werden. Zu beachten ist, dass dabei nicht das von Bootstrap mitgelieferte *bootstrap.css* eingebunden wird, sondern die selbst erstellte CSS-Datei.

In Listing 6 wird ein Cancel-Button im typischen Bootstrap-Styling erzeugt. Zu diesem Zweck wird ein gewöhnlicher HTML-Button verwendet, der mittels Bootstrap-Klassen gestylt wird. In diesen Button wird anschließend ein Glyphicon eingebettet. Damit das Drücken des Buttons auch die gewünschte Aktion auslöst, wird durch Verwendung der *jsf:action*- und *jsf:immediate*-Attribute aus dem HTML-Button ein Passthrough-Element erzeugt. Dadurch generiert JSF für diesen HTML-Button eine serverseitige Komponente, die auf Aktionen reagiert.

Das Beispiel zeigt, dass für diese doch einfache Aufgabe recht viele CSS-Klassen notwendig sind. Dieser Umstand lässt sich mittels LESS Mixin verschlanken (Listing 7). Die Lesbarkeit und Übersichtlichkeit des Codes wird dadurch erheblich verbessert (Listing 8). Ein schöner Nebeneffekt ist, dass Änderungen an diesem Cancel-Button nun zentral an einer Stelle erfolgen können. Um also beispielsweise die Größe oder das Glyphicon aller *.cancelbtn*-Buttons zu ändern, reicht es, die entsprechenden Mixins in der LESS-Datei anzupassen.

Listing 9 zeigt die Umsetzung eines typischen Formulareingabefelds mittels Bootstrap. Dabei werden anstelle von JSF-Tags gewöhnliche HTML-Elemente verwendet. Die Anbindung an die JSF Backing Bean erfolgt, indem das Inputelement in ein Passthrough-Element umgewandelt und der Wert mittels *jsf:value* gebunden wird. Durch

Listing 4: LESS kompilieren mittels Maven

```
<build>
  <plugins>
    <plugin>
      <groupId>org.lesscss</groupId>
      <artifactId>lesscss-maven-plugin</artifactId>
      <version>1.3.3</version>
      <configuration>
        <sourceDirectory>${basedir}/src/main/less</sourceDirectory>
        <outputDirectory>${basedir}/src/main/webapp/resources/css
                                                    </outputDirectory>
        <compress>true</compress>
        <includes>
          <include>myCustomBootstrap.less</include>
        </includes>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 5: Bootstrap-JSF-Template

```
<h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <h:outputScript library="js" name="jquery-1.10.2.min.js" />
  <h:outputScript library="bootstrap" name="js/bootstrap.min.js" />
  <h:outputStylesheet library="css" name="myCustomBootstrap.css" />
</h:head>
```

Listing 6: Input mit Bootstrap

```
<button type="button" class="btn btn-default btn-lg"
  jsf:action="#{controller.cancel}" jsf:immediate="true">
  <span class="glyphicon glyphicon-remove"></span> Cancel
</button>
```

Listing 7: Input mit Bootstrap

```
.cancelbtn{
  .btn;
  .btn-default;
  .btn-lg;
  span{
    .glyphicon;
    .glyphicon-remove;
  }
}
```

Listing 8: Input mit Bootstrap

```
<button type="button" class="cancelbtn"
  jsf:action="#{controller.cancel}" jsf:immediate="true">
  <span"></span> Cancel
</button>
```

Listing 9: Input mit Bootstrap

```
<form class="form-horizontal" role="form">
  <div class="form-group">
    <label for="price" class="control-label col-xs-2">Price:</label>
    <div class="input-group col-xs-10">
      <input type="text" class="form-control" id="price"
        p:placeholder="Enter price" jsf:value="#{bb.price}"/>
      <span class="input-group-Add-on">$</span>
    </div>
  </div>
</form>
```

die Verwendung der *.form-group*-Klasse von Bootstrap werden die logisch zusammengehörigen Formelemente gruppiert. Die bei JSF übliche horizontale Ausrichtung der Formularelemente erfolgt mittels *.form-horizontal*-Klasse. Mithilfe der *.col*-Klassen steuert man das Layout für die Label und Inputelemente.

In unserem letzten Beispiel (Listing 10) möchten wir die Verwendung einer Bootstrap-Pagination-Komponente mit JSF zeigen. Die Steuerung, die Aktivierung und Deaktivierung von einzelnen Elementen erfolgt dabei über EL-Ausdrücke, mit denen die entsprechende Style-Klasse ermittelt wird. Mittels der *h:commandLink*-Elemente werden beim Klicken die entsprechenden Aktionen am Server ausgelöst. Das *f:ajax* sorgt dafür, dass nur die Pagination-Komponente und die angezeigte Table neu gerendert werden. Letztlich wird mit *c:forEach* über die Anzahl der vorhandenen Seiten iteriert und für jede der Seiten ein entsprechender Eintrag dargestellt.

Fazit

Die Neuerungen in JSF zeigen ein grundlegendes Umdenken. So wird nicht mehr versucht, zwanghaft eine Abstraktionsschicht über HTML zu bilden, sondern Möglichkeiten geboten, wie HTML zusammen mit JSF genutzt werden kann. Wir zumindest fanden es erfrischend, Benutzeroberflächen einmal unter dem Gesichtspunkt moderner HTML5-, CSS- und JavaScript-Technologien zu betrachten und mit dem JSF-Ansatz zu verheiraten. Generell muss man sich die Frage stellen, ob die Verwendung von JSF-Komponenten für alle Elemente einer Webseite notwendig ist. Oftmals reichen auch HTML-Elemente, wie Labels, deren dynamischer Teil dann mittels eines EL-Ausdrucks ermittelt wird. Ein sich dadurch ergebender Vorteil ist, dass der Komponentenbaum verkleinert wird und somit auch Performanceverbesserungen erwartet werden können. Bei unserer Beispielapplikation hat sich gezeigt, dass es in der Regel ausreicht, nur jene Elemente als JSF-Komponenten zu deklarieren, die in der Geschäftslogik genutzt werden sollen oder solche Komponenten, die im Rahmen von Ajax Requests neu gerendert werden.

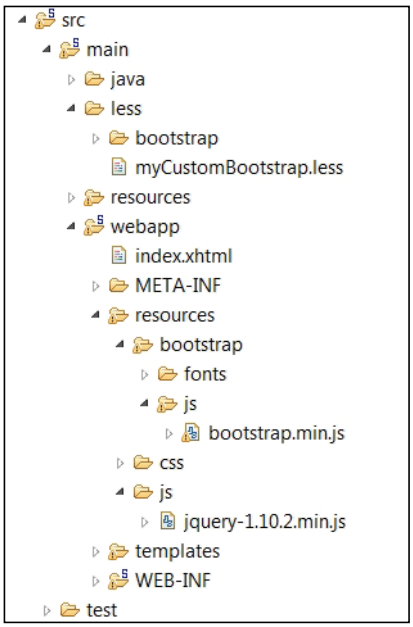

Abb. 6: Projektsetup

„Wire Frames“ sind mit modernen Entwicklungsumgebungen wie WebStorm unter Verwendung von HTML5 und Bootstrap schnell erstellt und erleichtern die Abstimmung mit der Fachseite. Bootstrap-Seiten sehen einfach gut aus und lassen auch den in Designfragen weniger geübten Entwickler schnell zu brauchbaren Ergebnissen kommen. Insbesondere wenn Anforderungen an „Responsive-Design“ im Lastenheft zu finden sind, stellt der Mobile-First-Ansatz von Bootstrap eine Alternative dar, die man ernsthaft in Erwägung ziehen sollte. Die klassische Entwicklung mit JSF muss deshalb nicht über Bord gehen.

Listing 10: Pagination-Beispiel

```
<h:panelGroup id="paginationCC">
  <ul class="pagination">
    <li class="#{controller.currentPage ne 1 ? '' : 'disabled'}">
      <h:commandLink value="&#171;" actionListener="#{controller.prevPage()}"
      disabled="#{controller.currentPage eq 1}">
        <f:ajax render="paginationCC #{cc.attrs.tableID}" />
      </h:commandLink>
    </li>
    <c:forEach begin="1" end="#{controller.pages()}" var="counter">
      <li class="#{controller.currentPage eq (counter) ? 'active' : ''}">
        <h:commandLink href="#" value="#{counter}">
          <f:setPropertyActionListener target="#{controller.currentPage}"
            value="#{counter}" />
          <f:ajax render="paginationCC #{cc.attrs.tableID}"/>
        </h:commandLink>
      </li>
    </c:forEach>
    <li class="#{controller.currentPage eq controller.pages() ? 'disabled' :''}">
      <h:commandLink value="&#187;" actionListener="#{controller.nextPage()}"
        disabled="#{controller.currentPage eq controller.pages()}">
        <f:ajax render="paginationCC #{cc.attrs.tableID}" />
      </h:commandLink>
    </li>
  </ul>
</h:panelGroup>
```

Christian Laboranowitsch ist als Softwareentwickler und Architekt beim IT-Beratungsunternehmen BridgingIT GmbH angestellt und schon seit mehr als fünfzehn Jahren in der Softwareentwicklung mit und ohne Java unterwegs.

Philipp Bayer ist als Softwareentwickler und Consultant beim IT-Beratungsunternehmen BridgingIT GmbH angestellt. Er beschäftigt sich seit acht Jahren mit Fragestellungen rund um Java und Webentwicklung, dabei war er in den unterschiedlichsten fachlichen Umfeldern tätig.

Links und Literatur

[1] https://github.com/twbs/bootstrap

[2] http://docs.oracle.com/javaee/7/tutorial/doc/jsf-facelets009.htm

[3] https://github.com/BitJavaGit/jsf-bootstrap-demo

[4] http://getbootstrap.com/css/#grid