Ganz ohne Ballast

Redundanzfreie Definition von Weboberflächen

Michael Hoppe, Dirk Pessarra, Fabian Schulte

Die Entwicklung großer Webapplikationen mit vielen Dialogen führt ohne ein geplantes Vorgehen zu einer großen Menge an redundanten Definitionen für grafische HTML-Elemente. Die Erfüllung von Anforderungen, wie Konformität und Wartbarkeit, wird dadurch erheblich erschwert. Der Artikel beschreibt, wie mit Templates und SiteMesh eine Kapselung von HTML-Elementen und zugehörigen JavaScript-Blöcken an nur einer Stelle erreicht wird, und zeigt die Vorteile der redundanzfreien Dialogdefinitionen auf.

Praxisbeispiel: Online-Reservierungsplattform

Das Essener Technologie- und Entwicklungs-Centrum (ETEC) bietet seit Mitte 2014 seine Konferenzräume und andere Ressourcen über die Online-Reservierungsplattform *oodra.de* an. Bei der Konzeption der mandantenfähigen Plattform war schnell klar, dass umfangreiche fachliche Prozesse mit einem feingranularen Rollen- und Rechtekonzept realisiert werden müssen. Die Funktionen reichen von der Verfügbarkeitsprüfung der Räume, über die Reservierung bis hin zum automatischen Versand von Buchungsbestätigungen.

Ziel bei der Oberflächenentwicklung

Eine redundante Nutzung von Oberflächenelementen für gleiche Datenstrukturen erschwert die Konformität und Wartbarkeit größerer Webanwendungen erheblich. Wir zeigen auf, welche Umstände bei der Entwicklung von Weboberflächen Redundanz erzeugen und wie sie vermieden wird. Das Ziel ist, für jedes sichtbare Dialogelement nur ein Quellcodekonstrukt zu verwenden, um das Don't-Repeat-Yourself-Prinzip (DRY) strikt einzuhalten. Missverständnisse während der Wartung über die Menge der von einer Oberflächenänderung betroffenen Stellen werden dadurch vermieden. Denn schließlich wird jedes Oberflächenelement nur durch ein Quellcodeelement definiert.

Dialogstrukturen wiederverwenden

Für die Entwicklung komplexer Geschäftsanwendungen bieten sich Webframeworks wie Grails, Rails oder Play an, welche die Anwendungsfälle für das Erstellen (*c*reate), die Anzeige (*r*ead), das Ändern (*u*pdate) und das Löschen (*d*elete) von Objekten aus dem fachlichen Modell heraus erzeugen. Diese werden CRUD-Anwendungsfälle genannt und der Prozess der Erzeugung aus dem Fachmodell wird als Scaffolding (engl. Gerüst) bezeichnet.

Dieser Rapid-Application-Development-Ansatz ermöglicht eine Abstimmung von Dialogstrukturen und Funktionen mit dem Kunden auf Basis der schnell erzeugten ersten Anwendungsversion. Je Geschäftsobjekt wird ein Dialog erzeugt, der nur die Attribute des Objekts selbst, nicht aber Informationen aus assoziierten Objekten enthält. Auftraggeber wünschen oft, dass Informationen aus fachlichen Objekten zusammen dargestellt werden. Die durch das Scaffolding erzeugten Dialoge müssen daher manuell angepasst werden.

Als Beispiel betrachten wir die Raumreservierung in Abbildung 1, die eine Beziehung zwischen den fachlichen Objekten *Kunde* und *Raum* darstellt. Für die Objekte *Kunde*, *Reservierung* und *Raum* gibt es jeweils CRUD-Dialoge. Bei der Ansicht der Reservierung möchte der Anwender im Dialog direkt die wichtigsten Daten des Kunden und des Raums mit angezeigt bekommen (s. Mitte Abb. 1). Die Dialogstrukturen des Kunden und des Raums sollten wieder verwendet werden, damit eine redundante Realisierung der Strukturen vermieden wird.

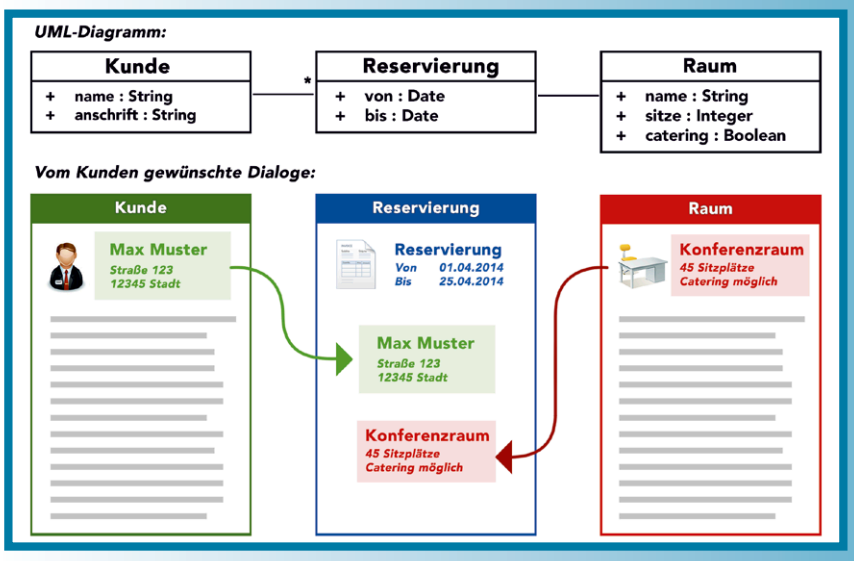

Abb. 1: UML-Diagramm und gewünschte Dialoge

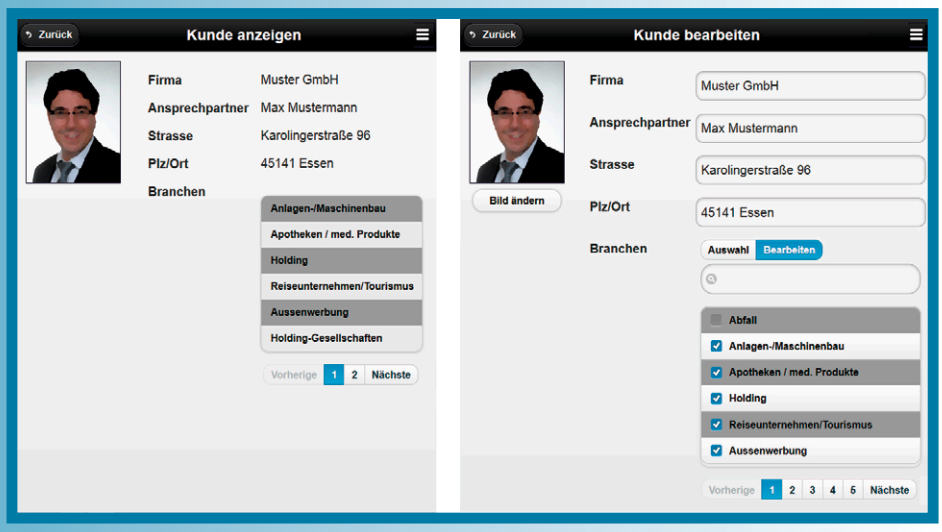

Abb. 2: Read- und Update-Dialog für das gleiche fachliche Objekt Kunde

Struktur und Verhalten

Eine Anwendung lässt sich durch ihre Struktur und ihr Verhalten definieren. Die Dynamik einer Anwendung führt oft zu Konstrukten, die sich nur durch ihr Verhalten unterscheiden. Abhängig von seinen Rollen und den verbundenen Rechten darf zum Beispiel ein Außendienstmitarbeiter die Kundendaten nur ansehen (linker Dialog in Abb. 2), während der Kundenbetreuer die Kundendaten verändern kann (rechter Dialog in Abb. 2). Beide Benutzer sehen die Kundendaten in gleicher Struktur dargestellt.

Bei den üblichen, durch Scaffolding erzeugten CRUD-Dialogen entstehen ein Read-Dialog für die Ansicht und ein Update-Dialog für die Bearbeitung der Daten. Eine Änderung des Objekts zieht die Änderung beider Dialoge nach sich. Es sind redundante Strukturinformationen vorhanden.

Lösungsideen

Für die Wiederverwendung von Dialogstrukturen bietet sich die Verwendung von Templates an. Die modularen Dialogstrukturen werden dabei in einzelne Templates ausgelagert und in verschiedene Dialoge eingewoben.

Weniger offensichtlich sind redundante Strukturinformationen, die durch unterschiedliche ausgeprägte Rechte entstehen. Wodurch unterscheiden sich die Dialoge aus Abbildung 2? Die unterschiedlichen Anwenderrechte erfordern im Detail andere HTML5- und CSS-Elemente sowie ein anderes Verhalten. Die Struktur der Dialoge ist ansonsten gleich. Wird diese Verhaltensänderung in die Tags einer Tag-Bibliothek verlagert, können redundante Dialogstrukturen gänzlich vermieden werden.

Technologien

Bevor wir näher auf die Realisierung dieser Lösungsidee eingehen, erklären wir kurz die wichtigsten, in dem Praxisbeispiel eingesetzten Technologien. Der vorgestellte Ansatz nutzt diese Technologien, ist jedoch nicht auf diese beschränkt.

Grails

Grails ist ein Web Application Framework, das auf der Programmiersprache Groovy basiert. Das Framework folgt dem MVC-Muster (Model, View, Controller) und bietet ausgereifte Konzepte zur Internationalisierung, Scaffolding und Erstellung von Tag-Bibliotheken sowie ein Plug-in-System. Des Weiteren überzeugt Grails durch die Verwendung von bewährten Frameworks wie Spring, Hibernate und SiteMesh. Anwendungen, die mit Grails entwickelt wurden, werden als WAR-Datei exportiert und sind somit in jedem Servlet-Container installierbar.

jQuery Mobile (JQM)

JQM ist ein für Touchscreen-Displays optimiertes JavaScript-Webframework. Es baut auf der populären JavaScript-Bibliothek jQuery auf, die unter anderem ein breites Spektrum an Funktionen zur DOM-Manipulationen und Transformation sowie Animationen und Ajax-Funktionalitäten bereitstellt. JQM stellt dem Entwickler ein umfassendes GUI- und Seitennavigationskonzept zur Verfügung, das die Webseite auch auf Tablets und Smartphones heimisch werden lässt.

Webseiten mit JQM werden in HTML geschrieben und das Verhalten und die Ausprägung der einzelnen Elemente über spezielle Attribute definiert. Diese Attribute werden von JQM automatisch bei Seitenaufruf in validen HTML-Quellcode umgewandelt.

Die Navigation bei einer mit JQM entwickelten Webseite wird mit Ajax realisiert. Bei Klick auf einen Link wird eine Ajax-Anfrage initiiert, die den neuen Seiteninhalt im Hintergrund lädt, dem DOM zufügt und optional mit einer Animation auf diesen neuen Inhalt überblendet.

Templates & SiteMesh

Templates erlauben durch die Kombination von statischen und dynamischen Elementen die Wiederverwendung von einmalig definierten Strukturen. Der dynamische Bestandteil eines Templates wird bei dessen Verwendung durch eine Template-Engine mit dem eigentlichen Inhalt ersetzt. Wird zum Beispiel ein Textbaustein als Formularelement in einem Template gekapselt, so ermöglicht das Template die Trennung von Objekt und Repräsentation. Der visuelle, statische Anteil des Textbausteins wird in einer einzelnen Datei gespeichert. Die dynamischen Anteile werden als Kontextobjekte an die Template-Engine übergeben, die dann den statischen Inhalt anreichert. Dieses Vorgehen ist in fast allen großen Entwicklungs-Frameworks üblich.

Ein weiterer Schritt zur Modularisierung von visuellen Elementen ist die Nutzung eines Decorator-Musters, um die Seitenstruktur zu definieren. Dies wird in Grails durch SiteMesh realisiert. Das SiteMesh-Framework erlaubt, mit einfachen Mitteln übergeordnete Template-Elemente parametrisiert zu definieren und somit auszulagern. So kann beispielsweise ein Menü oder sogar die Basisseite in einer ausgelagerten Template-Datei definiert und je nach Kontext in der anzuzeigenden Seite verwendet werden.

Realisierung der Lösungsideen

Im Folgenden präsentieren wir die Umsetzung der oben beschriebenen Ansätze. In Listing 1 ist die Wiederverwendung von Dialogbereichen für das Beispiel des Reservierungsobjekts aus Abbildung 1 dargestellt.

```
01 <!-- Reservierungsdialog -->
02 <ui:mode mode="READ">
03   <ui:fieldBean bean="${reservierungInstance}">
04     <ui:dateField name="gebuchtVon"></ui:dateField>
05     <ui:dateField name="gebuchtBis"></ui:dateField>
06   </ui:fieldBean>
07   <ui:fieldBean bean="${reservierungInstance.kunde}">
08     <g:render template="kundeForm"/>
09   </ui:fieldBean>
10   <ui:fieldBean bean="${reservierungInstance.raum}">
11     <g:render template="raumForm"/>
12   </ui:fieldBean>
13 </ui:mode>
```

Listing 1: Wiederverwendung von Dialogstrukturen durch den Grails-<g.render>-Tag

Durch den **<g:render>**-Tag werden die Templates **kundeForm** und **raumForm** in der aktuellen Benutzeroberfläche wiederverwendet. Durch SiteMesh wird das umliegende Grundlayout des Dialogs mit Titelleiste und Menü erzeugt. Der Quellcode dieser Elemente ist in separate Templates ausgelagert. Dies ist der einfachere Fall der Redundanzvermeidung.

Am Beispiel von Abbildung 2 wird für die beiden Dialoge mit unterschiedlicher Dynamik der schwierigere Teil betrachtet. Hier kann eine DRY-konforme Definition von Elementen einer Benutzeroberfläche mithilfe von JQM und Grails realisiert werden. Abhängig vom gewünschten Verhalten (Read oder Update) werden die Dialoge durch Listing 2 generiert. Das durch SiteMesh erzeugte umliegende Grundgerüst enthält alle Referenzen auf benötigte JS- und CSS-Dateien, die serverseitig zusammengefasst und komprimiert werden können.

```
01 <!-- Kundendialog -->
02 <ui:mode mode="${action}">   <-- Die Variable „action“ bestimmt
                                   das Verhalten:
                                   „READ“ zur Anzeige oder
                                   „UPDATE“ für die Änderung der Daten
03   <ui:fieldBean bean="${kundeInstance}">
04     <ui:image name="image"/>
05     <ui:textField name="firma" label = "Firma"/>
06     <ui:textField name="ansprechpartner"/>
07     <ui:textField name="strasse"/>
08     <ui:textField name="plz_Ort" label = "Plz/Ort"/>
09     <ui:multiAjaxSelectField name="branchen" controller="kunde"
10       fieldName="branchen" label="Branchen"/>
11   </ui:fieldBean>
12 </ui:mode>
```

Listing 2: Eine kleine Änderung im Quellcode macht aus der Anzeige einen Dialog für die Bearbeitung der Daten

Ein Dialog zum Anzeigen oder Bearbeiten eines Domain-Objekts besteht aus den Repräsentationen der einzelnen Felder des Objekts. Alle Visualisierungen von Domain-Objekt-Feldern werden in Aufrufe zu Tags in einer Bibliothek gekapselt. Dies bietet die Möglichkeit, zusätzliche Logik in den Tag-Methoden zu verwenden und Tag-spezifische Informationen in den Seitenkontext zu legen. Abhängig vom Seitenkontext kann, je nach Verhaltensmodus und Attributkombination, das geeignete Template ausgewählt werden. Dieses wird an das Rendering weitergegeben, um die Seite zu erstellen. Dadurch ist einheitlich definiert, wie eine einzelne Eigenschaft eines Domain-Objekts (z. B. eine Zahl oder eine Beziehung) in der Applikation visuell repräsentiert wird.

Der in Listing 2 gezeigte Tag **<ui:mode>** erstellt keine HTML-Strukturen, sondern legt nur Informationen im Seitenkontext ab, wodurch der aktuelle Modus an jedes Kindelement propagiert wird. Dieser Modus kann explizit angegeben oder durch Rollen und Rechte definiert werden. Durch die vorhandene Modusinformation wird in jedem Tag das anzuwendende Template ermittelt. Ist ein Recht zum Ändern vorhanden, wird die änderbare Variante eines Feldes oder einer Beziehung dargestellt. Ist das Recht nicht vorhanden, werden die Daten nur angezeigt. Diese Modi sind durch die Modularisierung erweiterbar, ohne dass vorhandene Referenzen verändert oder neue Kontrollstrukturen eingeführt werden müssen.

```
01 <div data-role="fieldcontain"
        class="ui-field-contain ui-body ui-br">
02   <label for="strasseTextField"
          class="ui-input-text">Strasse</label>
03   <div class="ui-input-text ui-corner-all ui-btn-shadow ui-body-c">
04     <span>Karolingerstraße 96</span>
05   </div>
06 </div>
```

Listing 3: Generierter HTML-Code des <ui:textField>-Tags im Read-Modus

In den Listings 3 und 4 ist zu erkennen, dass für den Read-Modus ein normales ****-Element und für den Update-Modus ein **<input>**-Element generiert wird. Bei beiden Varianten wird die Darstellung mithilfe des JQM-Elements **fieldcontain** realisiert, welches eine Beschriftung und einen Container um das Feld erstellt, der die Felder je nach Bildschirmbreite umbricht. Die beiden Modi unterscheiden sich für dieses Textfeld nur durch den Austausch des Span-Elements gegen ein Input-Element. In diesem Fall sind die Vorteile der Nutzung des Gesamtkonstrukts bereits erkennbar. Bei Änderungen der Anforderungen an die Repräsentation eines Textfeldes muss nur ein Template angepasst werden.

```
01 <div data-role="fieldcontain" class="ui-field-contain ui-body ui-br">
02   <label for="strasseTextField"
          class="ui-input-text">Strasse</label>
03   <div class="ui-input-text ui-corner-all ui-btn-shadow ui-body-c">
04     <input type="text" name="strasse" id="strasseTextField"
05       value="Karolingerstraße 96" class="ui-input-text ui-body-c">
06   </div>
07 </div>
```

Listing 4: Generierter HTML-Code des <ui:textField>-Tags im Update-Modus

Ohne die Kapselung des HTML-Codes in Templates müssten auch bei kleinen Änderungen, wie dem Hinzufügen einer CSS-Klasse, alle Textfelder aufwendig manuell angepasst werden. Ebenfalls kann in einer frühen Iteration der Entwicklung einer Applikation, beispielsweise ein Datumsfeld als reines Textfeld trivial gestaltet werden. In späteren Ausbaustufen ist es möglich, für alle Datumsfelder ein Kalender-Widget zu verwenden, indem nur an einem Tag aus der Bibliothek gearbeitet wird.

Eine zum Domänenmodell passende Tag-Bibliothek erfordert die einmalige Entwicklung eines Tags für jeden im Modell vorhandenen Attribut- und Assoziationstyp. Am aufwendigsten sind hier Beziehungen zwischen den Domain-Objekten, da diese zur Einhaltung des DRY-Prinzips komplexere Realisierungen erfordern.

Wie in Listing 2 in Zeile 9 dargestellt, kann eine n:m-Beziehung mit einem einzigen Tag abgebildet werden. Der **<ui:multiAjaxSelectField>**-Tag ist in diesem Fall ein generisches Element, das eine n:m-Beziehung zu Branchenobjekten abbildet. Im Modus „Read“ wird eine Liste über die gewählten Branchen (s. Abb. 2, linker Dialog), bei „Update“ ein Dialogelement zur Bearbeitung der Auswahl angezeigt (rechter Dialog).

Das komplexe Dialogelement verfügt über eine Paginierung für die Liste, eine Suchfunktion und eine Gruppe von Checkboxen in Form einer Liste der assoziierten Objekte.

Die Liste wird an dieser Stelle per Ajax aus einer generierten Controller-Methode geladen. Im Update-Modus wird eine generische Checkbox-Liste per Ajax nachgeladen.

Der durch den `<ui:multiAjaxSelectField>`-Tag generierte Code umfasst ein JavaScript-Modul mit 100 Zeilen und mehrere GSP-Templates mit insgesamt 90 Zeilen Code. Durch Nutzung der jQuery-Widget-Factory kann der JavaScript-Code in eigenen JS-Dateien gespeichert werden. Die Initialisierung erfolgt dann zum Zeitpunkt des Seitenaufrufs durch das `data-role`-Attribut an einem Element des `MultiAjaxSelect`-Tags, wie im jQuery-Framework üblich. Die JS-Datei wird über ein Grails-Tag als Abhängigkeit für das Template definiert und bei einem Seitenaufruf automatisch mitgeladen. Aus der Trennung von JavaScript und Groovy resultieren stabilere Skripte und eine bessere Wartbarkeit. Die Kapselung in einen Tag ermöglicht eine Aufteilung des notwendigen Wissens. Der Dialog-Entwickler benötigt Kenntnisse über die Anwendung des Elements, nicht aber über die komplexen Realisierungsdetails. Und der Tag-Entwickler kann die Wartung der Funktionalität an einem zentralen Punkt bearbeiten.

Fazit

Wir haben gezeigt, wie durch die Verwendung von Templates, SiteMesh und Tags redundante Dialogstrukturen gänzlich vermieden werden. Im Ergebnis können an der Oberfläche sichtbare Programmstrukturen durch ein Quellcodeelement realisiert werden, wodurch die Definition der Oberflächenelemente eine Abstraktionsebene angehoben wird. Der Quellcode rückt damit näher an das fachliche Modell heran. Die dadurch erst mögliche stringente Einhaltung des DRY-Prinzips zieht eine Menge positiver Konsequenzen nach sich:

- Es gibt nur eine Stelle, die von einer Datenstrukturänderung im Programm betroffen ist.
- Die Anwendung besitzt einen hohen Grad an Konformität, wodurch die Bedienbarkeit wesentlich einfacher ist.
- Die Oberflächenelemente werden konsequent wiederverwendet.
- Die Wartung von Dialogelementen erfolgt an zentraler Stelle.
- Die Lesbarkeit der Dialogdefinitionen ist optimal, da sie ohne technisch notwendigen komplexen Quellcode auskommt.
- Komplexe, technisch notwendige Programmkonstrukte, wie die Kommunikation über Ajax oder die Verwendung von JavaScript, lassen sich zusammen mit den Oberflächenstrukturen modularisieren.
- Im Scaffolding können deutlich einfachere Transformationen zwischen fachlichem Modell und Dialogen verwendet werden.
- Ein Wechsel des UI-Frameworks (z. B. von JQM auf Kendo UI) kann ohne Änderung der Dialogdefinitionen durchgeführt werden, da die konkreten UI-Elemente dort nicht referenziert werden.
- Die Abhängigkeit von einzelnen Personen im Projekt nimmt ab, da die Quellcodestellen leichter verständlich und das Ergebnis damit übertragbarer wird.
- Komplexe Elemente werden an einer Stelle definiert und konsequent wiederverwendet, wodurch die Entwicklung der Oberflächen im Durchschnitt einfacher ist.
- Die Qualität der erzeugten Oberflächen ist höher, da diese konformer sind und weniger Elemente zur Definition benötigen.
- Das Layout der Elemente und somit das gesamte Design der Anwendung ist leicht zu verändern, da für gleiche Elemente stets die gleichen HTML-Konstrukte verwendet werden.

Dem gegenüber werden folgende Nachteile gesehen:

- Die Entwicklung der modular wiederverwendbaren UI-Elemente ist schwieriger, da die „richtigen“ Parameter für die Verwendung der generischen Varianten vorausgedacht werden müssen.
- Beim Debugging sind die komplexen technischen Strukturen wieder sichtbar und eine Zuordnung zu den erzeugten Programmstrukturen ist aufwendiger.
- Der Ansatz benötigt Rechenleistung während der Programmausführung, wobei das Vorkompilieren der Templates den größten Teil abfängt.
- Für kleinere Anwendungen (z. B. mit weniger als 8 Klassen) lohnt sich die Entwicklung einer UI-Bibliothek mit den genannten Eigenschaften nicht.

Für die Anfangs erwähnte Plattform haben wir im Laufe des Projektes eine entsprechende Bibliothek erstellt. Die Zuordnung der erzeugten Programmstrukturen für das Debugging haben wir durch eine Angabe der Template-Bezeichnung im generierten HTML-Quellcode erreicht, sodass der zweite Nachteil stark abgeschwächt wird. Die Vorteile bei der Verwendung der Tag-Bibliothek waren so groß, dass die recht einfache Umstellung der alten UI-Strukturen auf die modularen Elemente der Bibliothek sukzessive, während der Anpassung von Dialogen aufgrund geänderter Anforderungen, durchgeführt wurde.

Als letzter Punkt sei zu erwähnen, dass eine Umstellung der Oberflächen der Plattform oodra.de auf eine neuere JQM-Version in kurzer Zeit, durch die Anpassung von ca. 20 verschiedenen UI-Elementen, die an über 600 Stellen verwendet wurden, durchgeführt werden konnte. Die Umstellung konnte also mindestens um den Faktor 30 beschleunigt werden, da eine Umstellung von 600 Stellen einen höheren Koordinationsaufwand sowie eine größere Fehleranfälligkeit mit sich gebracht hätte.

Links

[Grails] `https://grails.org`
[Groovy] `http://groovy.codehaus.org/`
[JQM] `http://jquerymobile.com`
[jQuery] `http://jquery.com`
[Scaffolding] Grails-Scaffolding-Plug-in, `http://grails.org/plugin/scaffolding`
[SiteMesh] `http://sitemesh.org`
[Template] `http://en.wikipedia.org/wiki/Web_template`

Michael Hoppe ist Consultant bei der neomatt GmbH. Er ist spezialisiert auf die Konzeption und Entwicklung plattformunabhängiger Weboberflächen, entwickelt aber auch native Anwendungen für iOS- und Android-Geräte.
E-Mail: michael.hoppe@neomatt.de

Dirk Pessarra ist Softwarearchitekt mit 30 Jahren Erfahrung in der Entwicklung. Mit Gründung der neomatt GmbH, deren Geschäftsführer er ist, hat er sich auf die Architektur von mandantenfähigen responsive Webanwendungen spezialisiert.
E-Mail: dirk.pessarra@neomatt.de

Fabian Schulte entwickelt und berät mit dem Schwerpunkt skalierbarer Webanwendungen und einem Fokus auf serverseitige Business-Logik inklusive der effizienten Datenspeicherung über Hibernate und SQL.
E-Mail: fabian.schulte@neomatt.de