Plus CD!

Stellenmarkt ▶ 49

Java-Memory-Modell ▶ 22

5.09

Java Magazin

Deutschland €7,50 Österreich €8,60 Schweiz sFr 15,80

S&S

Java magazin

Java • Architekturen • SOA • Agile

www.javamagazin.de

ALLE INFOS ZUR Jax09 AB SEITE 59

CD-Inhalt

Eberhard Wolff:
Spring und Architekturen

GUI-Renderer
- JAXFront 2.61

Frameworks
- Spring Web Services 1.5.6
- Grails 1.1
- Apache CXF 2.1.4
- Spring Framework 3.0.0 M2

User-Interface-Bibliotheken
- Dojo Toolkit 1.2.3
- jQuery 1.3.2

RFID-Open-Source-Tools
- Rifidi Emulator 1.5.3
- LLRP Toolkit
- Fosstrak Filtering and Collection 0.4.0

Alle CD-Infos ▶ 3

D 45867

4 194586 707500 05

Datenträger enthält Info- und Lehrprogramme gemäß § 14 JuSchG

Java WEB SERVICES

Frameworks: Spring-WS, Apache CXF, Axis2 ▶ 84

SPRING 3.0

Meilenstein 1: Alle Änderungen im Überblick ▶ 46

MyBlog wird erwachsen – Grails sei Dank

Folgende Stichworte sind für den geplagten Webentwickler oft gleichzusetzen mit „Überstunden schieben“: Ajax, Paging, I18N, Layout Templates, Tag Libs, URL Rewriting, Wizards, MVC, Servicekapselung und Security. Der Artikel zeigt, wie Grails den Feierabend retten kann.

von Marc-Oliver Scheele

Im letzten Teil dieses Tutorials haben wir den Grundstein für unsere Blogapplikation gelegt und insbesondere die Persistenz mithilfe von Domain-Klassen, die Definition von Constraints und das Scaffolding kennengelernt [1]. Ziel dieses zweiten Teils ist es, unseren Blog vollständig fertig zu implementieren. Mit dem Ergebnis aus dem letzten Java Magazin konnten bereits Artikel, Kommentare und User angelegt und verwaltet werden. Nun gilt es, das Frontend für den eigentlichen Blogleser zu entwickeln. Hierbei sparen wir nicht an Features, sodass die im Vorwort erwähnten Begriffe uns auf den nächsten Seiten begleiten werden.

Die folgende Implementierung baut direkt auf dem Ergebnis (also Sourcecode) des letzten Artikels auf, der bei Bedarf heruntergeladen werden kann (Kasten „Sourcecode zum Tutorial“).

Das Layout mit SiteMesh

Zunächst gilt es, das Layout für unser Frontend zu definieren. Da wir keine konkrete Designvorlage von unserem imaginären Auftraggeber erhalten haben und leider gerade kein Designer neben uns steht, müssen wir uns selbst behelfen. Wir entscheiden uns für ein klassisches Layout mit Header, Footer, einem Content-Bereich für die Blogeinträge und einer Sidebar auf der rechten Seite für zusätzliche Informationen. Glücklicherweise gibt es im Web einige gute Layoutvorlagen, sodass wir einen Großteil des HTML- und CSS-Codes nicht selbst entwickeln müssen. Die Seite *freecsstemplates.org* liefert uns eine schöne Vorlage mit dem Namen *Commission* [2].

Grails hat eine Template Engine integriert, um die Layouts einer Weboberfläche sauber mithilfe des *Decorator*-Patterns zu kapseln. Hierfür hat es das Framework *SiteMesh* integriert [3]. Wie kann man nun SiteMesh in einer Grails-Applikation nutzen? Der Trick besteht im Wesentlichen aus einem speziellen Verzeichnis unter *grails-app/views/layouts* und der Verwendung eines Meta Tags in den HTML-Dateien, um das Layout anzuwenden. Folgende Dateien legen wir zunächst in unserer Grails-Applikation an, bzw. kopieren sie aus dem heruntergeladenen Zip-File:

- Die Datei *style.css* als *site.css* nach *web-app/css* kopieren
- Das Bild *img01.gif* nach *web-apps/images* kopieren
- Erstellen einer neuen Datei zur Aufnahme des Layouts: *grails-app/views/layout/site.gsp*
- Zum Auslagern des Headers folgende Datei anlegen: *grails-app/views/layout/_header.gsp*
- Die konkrete Startseite, die das Layout anwendet, landet hier: *grails-app/home/index.gsp*

Jetzt besteht die eigentliche Arbeit darin, die Dokumentstruktur aus der *index.html* von *freecsstemplates.org* in die Datei *layouts/site.gsp* zu überführen. Listing 1 zeigt das Ergebnis. An dieser Stelle ein Hinweis auf die GSP-Dateien: Dies sind die Views im

Das Tutorial: Rapid Blog Development

Teil 1: Grails-Einführung – lauffähige Software in 30 Minuten

Teil 2: Frontend, Web 2.0-Features, Plug-ins – die Anwendung wird erwachsen

Teil 3: Automatisiertes Testing, Build-Systeme – die Qualitätssicherung

Quellcode auf CD

Model-Views-Controller-(MVC-)Konzept von Grails. Sie entsprechen den von JEE bekannten JSP-Dateien, glänzen aber durch erweiterte nützliche Eigenschaften. Beispielsweise sind beliebige Groovy-Ausdrücke, Zugriffe auf Model-Werte und das Lesen von Scope-Parametern mittels *${…}*-Notation möglich. Außerdem können leistungsfähige Grails-TagLibs genutzt und auch eigene Tags definiert werden. In unserer *site.gsp* wurden Standard-TagLibs von Grails, die alle mit *g:* starten, als Platzhalter für den eigentlichen Content genutzt.

Zurück zu unserem Layout: Im Listing sind drei vordefinierte Platzhalter für den HTML-Titel, den HTML-Head und den HTML-Body zu sehen. Weitere Bereiche können beliebig mit dem Tag *<g:pageProperty name="page.XXX">* definiert werden. Ein weiteres Grails-(nicht *SiteMesh*-)Feature ist aus dem Tag *<g:render template="" model="">* abzuleiten. Es entspricht einer *Include*-Anweisung, um beliebige HTML-Schnipsel zur besseren Wiederverwendung auszulagern. Die Template-Dateien selbst beginnen per Konvention mit einem Underscore. Listing 2 zeigt unsere *home/index.gsp*-Datei, die das gerade definierte Layout zur Darstellung nutzt. Wichtig ist hier zum einen das Meta Tag, das den Namen und somit die Verwendung unseres Layouts definiert. Zum anderen gilt es, ein Augenmerk auf die Tags *<title>*, *<head>*, *<body>* und *<content tag="sidebar">* zu legen, dessen Inhalt zur Laufzeit in die entsprechenden Platzhalter des Layouts überführt wird. Unser Ergebnis können Sie wie immer gleich ausprobieren. Starten Sie im Verzeichnis *blogapp* die Applikation wie gewohnt mit: *grails run-app*. Abbildung 1 zeigt, was wir beim Aufruf des URL *http://localhost:8080/blogapp/home/index.gsp* sehen.

URL-Mapping

Rufen wir aktuell die Startseite unseres Blogs unter http://localhost:8080/blogapp auf, so sehen wir das im ersten Teil programmierte GUI für den Backoffice-Bereich und nicht den eigentlichen Blog. Um das zu ändern, nutzen wir das so genannte „URL-Mapping" von Grails. Vielen Webentwicklern wird das Thema aufgrund der Anforderungen – „sorge für lesbare und SEO-konforme URLs" – bestens bekannt sein. Meist hilft man sich in dieser Situation mit einer Eigenentwicklung, nutzt eine fremde Library (z. B. *UrlRewriteFilter* [4]) oder inkludiert

Abb. 1: Das Layout der Blogapplikation

gar den Apache-Webserver in seine Architektur, um das *mod_rewrite*-Modul nutzen zu können. Nicht so mit Grails. Hier haben wir eine recht leistungsstarke, einfach zu nutzende und voll integrierte Lösung zum Bauen von schönen URLs verfügbar. Schauen wir uns die Funktionsweise anhand unseres Blogbeispiels an. Ziel ist, die Startseite korrekt zu setzen und unter *…/backoffice*-Zugriff auf unsere Admin-Funktionen zu bekommen. Um die Funktionsweise zu verdeutlichen, bauen

Listing 1: grails-app/views/layout/site.gsp

(Datei gekürzt! Siehe Kasten „Sourcecode und Grails-Version")

```
<html>
  <head>
    <title>
      <g:layoutTitle default="MyBlog" />
    </title>
    <link rel="stylesheet" href="${createLinkTo(dir:'css',file:'site.css')}" />
    <g:layoutHead />
  </head>
  <body>
    <div id="header-wrapper">
      <g:render template="/layouts/header" />
    </div>
    <div id="page">
      <div id="logo">
        <h1><a href="#">MyBlog</a></h1>
      </div>
      <hr />
      <div id="content">
        <g:layoutBody />
      </div>
      <div id="sidebar">
        <ul>
          <g:pageProperty name="page.sidebar"/>
        </ul>
      </div>
    </div>
    <div id="footer">
      <p>Java-Magazin....</p>
    </div>
  </body>
</html>
```

Sourcecode und Grails-Version

Der hier abgedruckte Sourcecode befindet sich mitsamt der kompletten Grails-Projektstruktur auf der Heft-CD. Teilweise wurde er in diesem Artikel zur besseren Übersicht gekürzt. Er ist selbstverständlich vollständig auf CD oder im Web unter folgendem URL herunterzuladen: http://www.moscon.de/grailstutorial. Auch das Sourcecode-Release aus Teil 1 dieses Tutorials ist dort zu finden.

Basis dieses Artikels ist die neue Version 1.1 von Grails. Da zum Zeitpunkt dieses Artikels das endgültige Release 1.1. noch nicht fertiggestellt war, wurde der abgedruckte Sourecode auf Basis des RC-1 (Release Candidate) getestet. Eventuelle Änderungen am MyBlog-Source zum endgültigen Release sind ebenfalls auf http://www.moscon.de/grailstutorial zu finden.

wir noch ein kleines Gimmick ein: Bei Eingabe eines URL wie *…/colorme/<farbe>* wird der Hintergrund in entsprechender Farbe präsentiert. Das URL-Mapping in Grails wird in der Datei *grails-app/conf/UrlMapping.groovy* durchgeführt (es kann auch auf mehrere Dateien mit der Endung *xxUrlMapping* aufgeteilt werden). Listing 3 zeigt unsere Konfiguration.

Es wird, wie bei Grails üblich, eine Groovy-DSL genutzt, um die URL-Schemata zu definieren. Der erste Teil pro Konfiguration gibt den URL-Pfad, der gemappt werden soll, an. Hier sind feststehende Begriffe, Variablen (mit *$* beginnend) und auch Platzhalter wie die von Ant bekannten Wildcards * und ** erlaubt. Nach dem URL-Pfad wird in der Regel das Mapping auf eine Controller-Action oder eine View definiert. Optional kann ein weiterer Block angegeben werden, der die vorangegangenen Definitionen z. B. durch Constraints weiter eingrenzt. Wir sehen in Listing 3 beispielsweise, dass alle URLs, die mit *…/colorme* starten, gefolgt von */red*, */blue*, */yellow* oder */green*, auch auf unsere Startseite verweisen (Index-Action im Home-Controller). Das Besondere ist, dass der Wert *$color* später im Request-Zyklus als *params.color* zugreifbar ist. Um dieses Farbe als Background zu setzen, muss der Body Tag in unserem Layout (Datei *site.gsp*) wie folgt angepasst werden: *<body style="${params.color?'background-color:'+params.color:''}">*. Bei dieser Gelegenheit können wir auch gleich die Links des Menüs richtig setzen (*_header.gsp*) und unsere Backoffice-Übersichtsseite ein wenig aufräumen (*../index.gsp*) (siehe mitgelieferter Sourcecode). Zu guter Letzt müssen wir noch den im Mapping definierten *HomeController* anlegen, um die neuen URLs ausprobieren zu können. Folgendes Grails-Kommando reicht dafür: *grails create-controller de.javamagazin.myblog.Home*. Nun können wir es ausprobieren:

- *http://localhost:8080/blogapp/* führt zur Abbildung 1
- *http://localhost:8080/blogapp/backoffice* zeigt die Startseite unserer Admin-Funktionen
- *http://localhost:8080/blogapp/colorme/yellow* liefert unseren Blog mit gelben Hintergrund
- *http://localhost:8080/blogapp/home/index* führt auch zur Abbildung 1 (Erklärung im nächsten Absatz)

Dem konzentrierten Leser wird in Listing 3 sicher die Definition */$controller/$action?/$id?* aufgefallen sein. Hierbei handelt es sich um ein sehr nützliches Standard-Mapping von Grails, das anzusprechenden Controller inklusive optionaler Action und einem optionalen *params.id*-Parameter direkt aus den Angaben in dem URL ableitet. Dadurch erreicht der Grails-Entwickler durch geschickte Benennung seiner Controller-Klassen und der darin enthalten Actions ohne explizite Konfiguration sehr lesbare URLs. Ein weiteres Schmankerl: Das Grails Tag *<g:link>*, das in den GSP-Seiten genutzt werden kann, um einen Link zu rendern, berücksichtigt die Mapping-Definitionen, sodass automatisch die passenden Links in die Seiten integriert werden (*Reverse URL Mapping*). Probieren Sie es aus: *<g:link controller="home" action="index" params="[color:'green']">Seite färben.</g:link>*

Das URL-Mappping von Grails bietet noch Vieles mehr. Unter anderem Mapping von HTTP-Request-Methoden für REST-Anwendungen, Mapping von HTTP-Fehlercodes und sprachabhängiges URL-Mapping (I18N).

Controller, Views und Paging

Wie in der Architekturdiskussion im ersten Teil bereits erläutert, basiert die Frontend-Entwicklung von Grails auf Springs MVC-Modul. Das V (=View) haben wir in Form der GSP be-

Listing 2: grails-app/views/home/index.gsp

(Datei gekürzt! Siehe Kasten „Sourcecode und Grails-Version“)

```
<html>
  <head>
    <meta name="layout" content="site"> <!-- Anwendung Layout site.gsp -->
    <title>Willkommen bei MyBlog</title>
  </head>

  <body>
    <div class="post">
      <h2 class="title">Überschrift eines Blogeintrages</h2>
      <!-- .... -->
    </div>
    <div class="post">
      <h2 class="title">Mein Bla-Bla Titel</h2>
      <!-- .... -->
    </div>
  </body>

  <content tag="sidebar">
    <li>
      <h2>Dies ist unsere Sidebar</h2>
      <p>Hier stehen später seitenspezfische Informationen....</p>
    </li>
  </content>

</html>
```

Listing 3: grails-app/conf/UrlMapping.groovy

```
class UrlMappings {
  static mappings = {

    // startpage
    "/" (controller:'home', action:'index')

    // backoffice
    "/backoffice" (view:'index')

    // showcase: Use special URLs to render the page in a different color
    "/colorme/$color" (controller:'home', action:'index') {
      constraints {
          color matches: /(red)|(blue)|(yellow)|(green)/
      }
    }

    // standard mapping of Grails
    "/$controller/$action?/$id?" {}

    // error definition
    "500"(view:'/error')
  }
}
```

reits beim Thema Layouts gesehen. Das C (=Controller) haben wir auch schon kurz gestreift, indem wir unseren HomeController mit einer leeren Action namens *index* angelegt haben. Das M (=Model) ist das Bindeglied zwischen dem Controller und der View. Es übergibt die dynamischen Daten in Form einer Map-Datenstruktur an die View. Lassen Sie uns nun den Blog erweitern, um unsere Blogartikel aus der Datenbank zu lesen und entsprechend anzuzeigen. Dabei bauen wir ein Paging ein, sodass maximal 10 Einträge pro Seite angezeigt werden und eine Blätterfunktion verfügbar ist. Neueste Artikel sollen oben in der Liste erscheinen.

Wie schon gewohnt, ist diese Erweiterung mit Grails ein Kinderspiel und somit sehr schnell erledigt. Die Action im Controller erweitern wir um zwei Dynamic-Finder-Methoden, um die Datenbank auszulesen (für Erläuterungen zu „Dynamic Finder" siehe [1]). Wir passen die View *home/index.gsp* an, um die Daten aus der Model Map zu entnehmen, und nutzen das leistungsfähige Grails Tag <*g:paginate*>, um die Blätterfunktion zu realisieren. Noch ein paar kleine Änderungen in der *site.css*, um das Layout für die Paging-Knöpfe anzupassen, und schon sind wir fertig. Um das Ganze vernünftig ausprobieren zu können, wurde die Datengenerierung in der Datei *conf/Bootstrap.groovy* erweitert. Beachten Sie bitte, dass unser alter Blogeintrag nur zu sehen ist, wenn Sie seinen Status im Backoffice-Bereich auf PUBLISHED umstellen. Listing 4 fasst alle Änderungen zusammen. Ein Hinweis noch zum Rückgabewert der *index*-Action: In unserem Fall wird nur ein Model zurückgegeben (*return model*). Woher weiß Grails nun aber, welche View mit den Daten gefüllt und angezeigt werden soll? Ist keine explizite Angabe einer View in der Action gegeben (durch *render* (*view:'xxx'..*) oder *redirect* (*view:'xxx',..*)), wird die View per Konvention aufgelöst. Der Pfad der View leitet sich in diesem Fall aus dem Namen des Controllers und der Action ab. In unserem Fall also: *views/home/index.gsp*.

Blogkommentare mit Ajax

Als Nächstes fügen wir unserem Blog die Kommentarfunktion hinzu. Da die Kommentarlisten in der Praxis sehr lang werden können, sollen diese je Blogartikel erst nach Mausklick nachgeladen werden. Ein perfekter Anwendungsfall, um das Ajax-Handling von Grails kennenzulernen. Abbildung 2 zeigt das erwartete Ergebnis: Bei Klick auf *Kommentar (x)* unter einem Artikel werden zugehörige Kommentare inklusive einem Formular zum Hinzufügen angezeigt.

Auch in Bezug auf Ajax und JavaScript verfolgt Grails das Motto „Don't reinvent the wheel". Anstatt eigene Funktionalität zu entwickeln, werden vorhandene und etablierte JavaScript-Bibliotheken elegant eingebunden. Standardmäßig setzt Grails auf Prototype und Scriptaculous [5]. Durch freie Grails-Plug-ins kann die zugrunde liegende JavaScript-Bibliothek u. a. auch gegen Yahoo-UI, Dojo oder jQuery ausgetauscht werden [6].

Was ist nun der Mehrwert gegenüber direkter Nutzung der JavaScript-Frameworks? Grails bietet eine Reihe von Tags an, mit denen beispielsweise sehr einfach Formulare (<*g:formRemote*>) oder Links (<*g:remoteLink*>) über die Ajax-Schnittstelle des Browsers verarbeitet werden können. Außerdem gibt es Spezialfunktionen wie das Mitlesen von Zeichen bei der Eingabe in Textfeldern (<*g:remoteField*>). Betrachten wir nun die Änderungen an dem Blogsystem. Listing 5 zeigt die wesentlichen Erweiterungen in dem Source. Zusammenfassend sind folgende Schritte durchzuführen:

- Im Layout (*grails-app/views/layout/site.gsp*) wird eine Anweisung hinzugefügt, um Prototype einzubinden: <*g:javascript library="prototype"*/>
- Der View-Bereich zum Anzeigen der Kommentare wird in ein eigenes Template ausgelagert (*grails-app/view/home/index.gsp*)
- Das Template beinhaltet das Layout und die Anweisung, um die Kommentare inklusive Eingabeformular abzubilden (*grails-app/view/home/_comments.gsp*). Kommen-

Listing 4: Blogartikel inklusive Paging dynamisch anzeigen

```
// grails-app\controllers\de\javamagazin\myblog\HomeController.groovy
class HomeController {
  def index = {
    Map model = [:]
    model.blogArticles = BlogArticle.findAllByStatus ( ArticleStatus.PUBLISHED,
                         [max:10,
                          offset:params.offset,
                          sort:'dateCreated',
                          order:'desc'] )
    model.blogCount = BlogArticle.countByStatus ( ArticleStatus.PUBLISHED)
    return model
  }
}

// Änderungen an grails-app\views\home\index.gsp
...
 <body>
  <g:each var="article" in="${blogArticles}">
   <div class="post">
    <h2 class="title">${article.subject}</h2>
    <p class="meta"><em>
     { Veröffentlicht von ${article.author.username} am
      <g:formatDate format="dd.MM.yy" date="${article.dateCreated}"/> }
    </em></p>
     <div class="entry">
      <p>${article.body}</p>
      <p class="links">
       <a href="#" class="comments">
         Kommentare (${article.comments.size()})
       </a>
      </p>
     </div>
   </div>
  </g:each>
  <div class="paginateButtons">
    <g:paginate total="${blogCount}" />
  </div>
 </body>
...

// Für Änderungen im Design (web-app\css\site.css) und
// Erweiterungen der Testdaten (conf\BootStrap.groovy)
// siehe bitte mitgelieferter Sourcecode
```

Abb. 2: Kommentarfunktion mit Ajax

tare werden jedoch nur gerendert, falls im Model der Wert *showComments* gesetzt ist. In dem Template befinden sich ebenfalls die Tags für die Ajax-Aufrufe (*g.remoteField* und *g.formRemote*). Wichtig in den Remote Tags ist das *update*-Attribut: Sämtlicher HTML-Code, den der Ajax-Request zurückliefert, ersetzt den Inhalt des HTML-Tags mit entsprechender ID (in unserem Fall das * * in der *home/index.gsp*)

- Der *HomeController* wurde um zwei weitere Actions erweitert, die für das Bearbeiten der Ajax-Requests zuständig sind. Die Action *ajaxShowComments* antwortet lediglich mit dem HTML-Code aus *_comments.gsp* inkl. der Kommentarliste. *ajaxSaveComment* verarbeitet das Anlegen eines neuen Kommentars.

Internationalisierung (I18N)

Selbstverständlich werden mit Grails auch sehr große Websites gebaut. Von daher darf das Thema Internationalisierung nicht fehlen. Grails bedient sich hierbei im Wesentlichen der von Java bekannten Mechanismen. So werden sprachabhängige Texte standardmäßig in Resource Bundles abgelegt. Sie finden sich in der jeweiligen Property-Datei im Verzeichnis *grails-app/i18n*. Das aktuelle Locale, das bestimmt, welche Sprache und Formatierung verwendet wird, ergibt sich standardmäßig aus dem HTTP-Request des Browsers. Durch Konfiguration kann das Locale aber auch fest verdrahtet oder beispielsweise abhängig von bestimmten URL-Patterns definiert werden.

Listing 5: Kommentarfunktionen mit Ajax

```
//
// Änderungen an grails-app\views\home\index.gsp
//
...
<body>
...
   <div class="entry">
     <p>${article.body}</p>
     <div id="comment_area_${article.id}">
       <g:render template="comments" model="[article:article]" />
     </div>
   </div>
 ...
</body>
...

//
// grails-app\views\home\_comments.gsp
//
<g:set var="comments" value="${article?.comments?.sort{it.dateCreated}}" />
<g:set var="index" value="${article?.id}" />
<p class="links">
  <g:remoteLink action="ajaxShowComments" id="${article?.id}"
        update="comment_area_${index}" class="comments" >
    Kommentare (${comments?.size()})
  </g:remoteLink> </p>
<g:if test="${showComments}">
  <div id="show_comments_${index}" class="commentBox">
   <p class="links" style="margin-bottom:5px">
    <a href="#" onclick="$(,show_comments_${index}').remove();return
                                                  false;">[Kommentare schliessen]</a>
   </p>
   <g:each var="comment" in="${comments}">
     <p>
       Kommentar von ,${comment.user?.username?:'Anonymous'}" am
       <g:formatDate format="dd.MM.yy" date="${comment.dateCreated}"/>: <br/>
       ${comment.message}
     </p>
   </g:each>
   Kommentar hinzufügen:
   <g:eachError bean="${newComment}">
     <div class="error"><g:message error="${it}"/></div>
   </g:eachError>
   <g:formRemote name="newComment" update="comment_area_${index}" url=
                                                  "[action:'ajaxSaveComment']">
       <g:textArea class="input" name="message" value="${newComment?.message}"
                                                  rows="2" cols="40"/>
       <input type="hidden" name="articleId" value="${article?.id}">
       <input type="submit" class="submit" value="Abschicken" />
   </g:formRemote>
  </div>
</g:if>


//
// Änderungen an grails-app\controllers\de\javamagazin\myblog\HomeController.groovy
//
def ajaxShowComments = {
  def ba = BlogArticle.read(params.id)
  render (template:'comments', model:[article:ba,showComments:true])
}


def ajaxSaveComment = {
  Map model = [:]
  BlogArticle article = BlogArticle.read(params.articleId)
  BlogComment comment = new BlogComment()
  comment.message = params.message
  comment.article = article
  if (!comment.save()) {
    model.newComment = comment
    model.showComments=true  // show form again on error
  }
  model.article = article
  render (template:'comments', model:model)
}
```

Abb. 3: Internationalisierte Fehlermeldungen

Wollen wir exemplarisch unseren Linktext *Kommentare (x)* internationalisieren, so definieren wir für die deutsche Umgebung in der *message_de.properties*: *myblog.comments.title=Kommentare ({0})*. Innerhalb der GSP kann über folgendes Tag darauf zugegriffen werden: *<g:message code="myblog.comments.title" args="[comments?.size()]"/>*

Da jedes Tag als Methodenaufruf auch in Controllern verwendet werden kann, würde der Zugriff aus dem Controller wie folgt aussehen: *g.message(code:'myblog.comments.title',args:[comments?.size()])*

Eine Besonderheit ist beim Anzeigen von Fehlermeldungen zu vermerken. In Listing 5 ist zu sehen, dass wir eventuelle Fehlermeldungen beim Anlegen eines Kommentars wie folgt ausgeben:

```
<g:eachError bean="${newComment}">
    g:message error="${it}"/>
</g:eachError>
```

Per Konvention sucht Grails eine Fehlermeldung im Resource Bundle, die den Namen der Domain-Klasse, des Properties und des Constraints beinhaltet. Standardtexte sind hierfür bereits hinterlegt. Wir ändern die Meldungen in der Property-Datei für den Fehler eines leeren Kommentars bzw. falscher Textlänge wie folgt:

```
blogComment.message.blank=Bitte geben Sie einen Kommentar ein.
blogComment.message.size.error=Der Kommentar muss zwischen {3} und {4} Zeichen lang sein.
```

Listing 6: grails-app\services\de\javamagazin\myblog\AdvertisingService.groovy

```
package de.javamagazin.myblog

class AdvertisingService {
  static int counter=0

  boolean transactional = false

  String catchNextAdvertising() {
    def ads = [
      'Das Java Magazin informiert stets über innovative Java…',
      'Bei www.moscon.de gibt es Unterstützung in Sachen Grails…',
      'Kaufen Sie sich jetzt eine Bank. Sie sind so billig wie noch nie'
    ]
    return ads[(counter++) % ads.size()]
  }
}
```

Abbildung 3 zeigt den Unterscheid zwischen der Standard- und der selbstdefinierten Fehlermeldung.

Advertising mit Services

In Zeiten der Weltwirtschaftskrise muss man sich selbstverständlich Gedanken über die Finanzierung des MyBlogs machen. Unser imaginärer Auftraggeber schlägt deshalb vor, ein Anzeigenmodul zu integrieren. Folglich müssen wir in unserer Applikation zunächst die Logik für die Anzeigenschaltung und Anzeigenbeschaffung unterbringen. Hierfür bieten sich in der Grails-Architektur die so genannten Services an. Wie in jeder anderen klassischen Softwarearchitektur werden in der Serviceschicht Funktionen gekapselt, die wiederverwendbar und modularisierbar sind. Insbesondere sind Funktionen für die transaktionale Datenmodifikation und Integrationsaufgaben dort angesiedelt. In Grails können wir wie immer die Servicedatei entweder per Hand anlegen oder vom Grails-Tool anlegen lassen: *grails create-service de.javamagazin.myblog.Advertising*. Normalerweise müssten wir nun fortgeschrittene Funktionen zur Einbindung diverser Anzeigenpartnerfirmen programmieren. Wir vereinfachen aber ein wenig (Listing 6).

Was ist das Besondere an Grails Services? Hier sind im Wesentlichen zwei Eigenschaften zu erwähnen:

- Der Service wird automatisch als Spring Bean eingebunden und kann somit in der ganzen Applikation via Dependency Injection eingebunden werden. In Grails reicht es, hierzu in Controllern, Domain-Objekten, Tag Libs oder anderen Services eine Instanzvariable mit dem Namen des Service zu de-

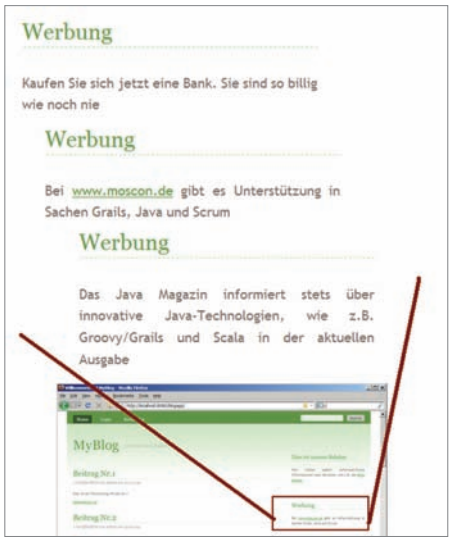

Abb. 4: Werbeanzeigen in MyBlog

klarieren. In unserem Beispiel: *def advertisingService* (siehe nächster Abschnitt).

- Die Methoden des Service können als Transaktionsgrenzen dienen. Das ist immer sinnvoll, wenn schreibende Datenbank- bzw. Ressourcenzugriffe stattfinden. Ist dies nicht gewollt, wie bei unserem Beispiel, so muss Folgendes deklariert werden: *boolean transactional = false*.

Eigene Tags entwickeln

Um unsere Anzeige in dem GUI zu präsentieren, spendieren wir unserer Applikation ein eigenes GSP Tag. Somit sind das Anzeigenlayout und die zugrunde liegenden Funktionen in einer eigenen Komponente gekapselt und können in unseren Views an beliebigen Stellen wiederverwendet werden. Der erfahrene JEE-Entwickler wird an dieser Stelle die Stirn runzeln, da er sich die Custom Tags in JSPs mitsamt der aufwändigen Implementierung in Erinnerung ruft. Keine Angst: auch bei diesem Feature bleibt Grails seinem Motto „keep it simple“ treu. Zunächst legen wir eine neue Groovy-Datei für eine neue Tag Library an. Am einfachsten mit dem Aufruf von: *grails create-tag-lib de.javamagazin.myblog.MyBlog*

Innerhalb der erzeugten Groovy-Klasse *MyBlogTagLib* können nun mittels einer Property-zugewiesenen Closure beliebig viele Tags definiert werden. Listing 7 zeigt unsere Tag Lib mit dem implementierten Tag *<my:showAdvertising>*.

Mittels der Deklaration von *def advertisingService* wird der oben definierte Service in unsere Klasse injectet. Mit *static namespace* kann ein eigener Namespace für die Tags definiert werden (ohne dies wären die Tags mit *<g:…>* zugreifbar). Die Logik unseres Tags besteht lediglich aus dem Aufruf des Service, um den Anzeigentext zu erhalten. Die in Tag Libs verfügbare Variable *out* wird dann verwendet, um das HTML samt Text zur Laufzeit anstelle des *<my:showAdversing>* Tags auszugeben. Die beiden Parameter *attrs* und *body* enthalten Tag-Attribute bzw. den Text zwischen Tag-Start und -Ende. In unserem Fall wird das Tag jedoch ohne Attribut und Body aufgerufen, sodass wir diese ignorieren können. Das reicht, um das Tag zu implementieren. Wir können es nun in unserer View *home/index.gsp* nutzen (Ergebnis in Abb. 4):

```
…
 </li>
 <my:showAdvertising />
</content>
…
```

Wizards und Web Flows

An dieser Stelle in aller Kürze der Hinweis auf ein weiteres, sehr nützliches Feature von Grails zur GUI-Entwicklung. Steht beispielsweise die Anforderung ins Haus, eine Registrierung oder einen Kaufprozess über mehrere Pages hinweg zu implementieren, wird es häufig kompliziert. Datenobjekte müssen über mehrere Requests verwaltet und der Status muss beobachtet werden, um den Nutzer bei seinen Klicks auf dem korrekten Pfad zu wissen. Kennern des Spring Frameworks fällt sicher gleich der Begriff „Spring Web Flow“ ein. Genau dieses Modul nutzt auch Grails. Hiermit lässt sich ein Zustandsautomat für Seitenabläufe

Listing 7: grails-app\taglib\de\javamagazin\myblog\MyBlogTagLib.groovy

```
package de.javamagazin.myblog

class MyBlogTagLib {

  def advertisingService

  static namespace = "my"

  def showAdvertising = { attrs, body ->
    String ad = advertisingService.catchNextAdvertising()
    out << """
      <li>
        <h2>Werbung</h2>
        <p>${ad}</p>
      </li>
    """

  }
}
```

Listing 8: grails-app\conf\SecurityFilters.groovy

```
class SecurityFilters {
  def filters = {
    auth(controller: "*", action: "*") {
      before = {
        // HomeController is always accessible, others need an Admin
        if (controllerName == "home") return true
        accessControl {
          role("ADMIN")
        }
      }
    }
  }
}
```

definieren. In bekannter Grails-Manier wird eine eigene Groovy-DSL angeboten, mit der in den oben kennengelernten Controllern sehr schnell und in Kombination mit normalen Actions die so genannten Web Flows abgebildet werden können [7].

Plug-ins und Module

Zu guter Letzt lassen Sie uns das Thema „Plug-ins" besprechen. Wie jede andere erfolgreiche Entwicklungsplattform bietet auch Grails ein einfach zu nutzendes, aber dennoch sehr leistungsstarkes Plug-in-Konzept (in Version 1.1 von Grails wurde es nochmals stark erweitert). Uns als Applikationsentwickler können Grails-Plug-ins in zweierlei Hinsicht sehr hilfreich sein:

- Nutzung fremder, öffentlicher Plug-ins, um die Funktionalität der eigenen Applikation ohne Implementierungsaufwand zu erweitern. Siehe dazu Tabelle 1 „Nützliche Grails-Plug-ins".
- Nutzung, um die eigene Applikation besser zu strukturieren und in Module aufzuteilen. Dies sollte vor allem bei größeren und anspruchsvolleren Grails-Applikationen als „Architekturwerkzeug" genutzt werden. (Die eigene Entwicklung von Plug-ins wird hier nicht beschrieben. Es sei auf die Onlinedokumentation verwiesen [8])

Im Folgenden wollen wir das JSecurity-Plug-in nutzen, um unsere Applikation mit Authentifizierung und Autorisierung auszustatten. Mit folgendem Befehl erhalten wir zunächst eine Übersicht aller im Grails-Repository veröffentlichten Plug-ins und können uns überzeugen, dass ein JSecurity-Plug-in vorhanden ist:

```
grails list-plugins
....
jsecurity  <0.3> --  Security support via the JSecurity framework.
....
```

Grails-Sicherheit

Da Grails ein Framework zum Bauen von Webanwendungen ist, ist das Thema Sicherheit ein wichtiger Bestandteil. Neben dem verwendeten JSecurity-Plug-in gibt es aktuell alternativ noch drei weitere Plug-ins für diesen Bereich: Spring-Security-Plug-in, Stark Security Plug-in und das Authentication-Plug-in. Selbstverständlich kann auch jedes andere Java-Security-Framework in Grails verwendet werden. Die Plug-ins bieten lediglich eine sehr komfortable Einbindung/Nutzung für den Entwickler.

Authentifizierung und Autorisierung ist natürlich nur ein kleiner Bestandteil, um eine Applikation abzusichern. Folgendes bietet Grails gegen verbreitete Angriffe:

- *XSS-Angriffe:* Automatisches HTML-Encoding von Model Values in Views (global, per Page, per Value) und URL-Encoding
- *SQL Injection:* Verwendung von Parametern für Queries und damit Escaping
- *CSRF-Angriffe:* Neu in Grails 1.1 – Hinterlegung von Token pro Formularabruf. Löst auch das Double-Submit-Problem!

Mit folgendem Kommando lassen sich Detailinformationen zu dem Plug-in ausgeben:

```
grails plugin-info jsecurity
```

Nachdem wir uns nun genügend informiert haben, installieren wir die neueste Version des JSecurity-Plug-ins wie folgt:

```
grails install-plugin jsecurity
```

Auf Ihrer Konsole können Sie verfolgen, wie die Zip-Datei des Plug-ins von *plugins.grails.org* heruntergeladen wird. Nach erfolgreicher Installation wird das Plug-in standardmäßig unter *$USER_HOME/.grails* abgelegt und von dort auch unserer Applikation zur Verfügung gestellt. In unserer Applikation selbst wird lediglich die *application.properties* automatisch angepasst, um auf das Plug-in zu verweisen. Erst in dem Moment, wo wir ein *war*-File zur Installation bauen (*grails war*), werden die JSecurity-Artefakte in unser Projekt übernommen.

Security

An dieser Stelle ist wieder ein wenig Programmierung angesagt, um die Funktionen des JSecurity-Plug-ins im Blog zu verwenden. Das Plug-in stellt das Java-Framework JSecurity, das ein leistungsstarkes, aber dennoch handlich zu nutzendes API für die Authentifizierung/Autorisierung einer Applikation anbietet [9], zur Verfügung. Unter anderem bietet es auch feingranulare Kontrolle über „Permissions". Für unseren Blog reicht rollenbasierte Sicherheit jedoch völlig aus (Kasten

Abb. 5: Login mit Jsecurity

„Grails-Sicherheit"). Rufen wir uns zunächst das Domain-Modell in Erinnerung, das wir im Rahmen des letzten Artikels angelegt hatten, wo wir zwar einen BlogUser definiert hatten, aber noch keine Rollenklasse. Das holen wir jetzt nach:

- Anlegen der Domain-Klasse: *BlogRole* mit einer bidirektionalen *m:n*-Beziehung zu dem *BlogUser*

Nützliche Grail Plug-ins

Aktuell existieren über 150 Plug-ins für verschiedenste Aufgaben [9]. In der Grails-Community wird zurzeit ein Plug-in-Portal entwickelt, um eine bessere Übersicht und Verwaltung der schnell wachsenden Zusatzmodule sicherzustellen (zum Zeitpunkt dieser Veröffentlichung wahrscheinlich schon verlinkt unter http://grails.org)
Folgende Auflistung zeigt exemplarisch ein paar interessante Plug-ins.

GrailsUI	Elegante und einfache Einbindung von komplexen JavaScript- und Ajax-Funktionen auf Basis von Yahoo! UI und der Bubbling Library
RichUI	Alternative zum vorstehenden Plug-in
SyntaxHighlighter	Sourcecode-Darstellung in den Views für den Endnutzer
Google Chart	Einfache Verwendung von Googles Chartgrafiken
Quartz	Ausführung von asynchronen Jobs
Searchable	Einfach Konfiguration von Volltextsuche auf Basis von Compass/Lucene
Datasource	Aufteilung von Domain-Klassen auf mehrere Datasources/ Datenbanken
JMS	JMS-Verwendung auf Basis von Message-driven POJOs
Feeds Plugin	Einfache Generierung von RSS/Atom Feeds
Encryption Plugin	Einfache Verwendung von Verschlüsselung auf Basis von Blowfish und PGP
Mail	Versand von E-Mails inklusive Template Engine
PayPal	Anbindung an den Zahlungsdienst PayPal
Twitter	Lesen und Schreiben von Twitter-Nachrichten
Autobase	Unterstützung von Datenbankmigration während der Entwicklung auf Basis von Liquibase
DynamicJasper	Einfaches Erzeugen von Reports auf Basis von Jasper
PHP Plugin	Etwas exotisch: Erlaubt die Ausführung von PHP-Code in einer Grails-/Java-Anwendung
OpenID	Authentifizierung über den OpenID-Standard
PartyTime	Kein echtes Plug-in, sondern eher eine Basisapplikation für den Quick-Start: Implementierung von häufigen Web-Features (User-Verwaltung, Messaging, Friends, Blogs)
Terracotta	Generiert Konfiguration und Skripte für die Skalierung von Grails-Applikationen über Terracotta

Tabelle 1: Nützliche Grails-Plug-ins

- Erweiterungen im *Bootstrap.groovy*, um der Datenbank die Rollen *ADMIN* und *USER* hinzuzufügen sowie unserem vorhandenen Nutzer *admin* die Admin-Rolle zuzuordnen
- Anlegen eines dynamisch gescaffoldeten Controllers *BlogRoleController*, um die Rollen in unserem Admin-GUI administrierbar zu machen.

Diese Schritte sollten nach Lektüre des ersten Tutorial-Teils selbstständig ausführbar sein, sodass der Source hier nicht abgebildet ist (Kasten „Sourcecode"). Eine Besonderheit gibt es noch beim Passwort des *BlogUsers*. Bisher war es unverschlüsselt. Das korrigieren wir nun, indem wir die JSecurity-Funktion *Sha1Hash* verwenden. Hierzu überschreiben wir die entsprechende Setter-Methode im *BlogUser* wie folgt:

```
void setPassword(String passwd) {
 if (passwd && !(passwd ==~ /[\da-fA-F]{40}/)) {
  this.password = new Sha1Hash(passwd).toHex()
 } else {
  this.password = passwd
 }
}
```

Damit wird jedes Passwort, das nicht nach SHA1 aussieht (Länge von 40 Zeichen mit Hexadezimalwerten), entsprechend kodiert. Außerdem gilt es noch, den Constraint des Passworts in der Domain-Klasse auf 40 Zeichen zu erhöhen. Als Nächstes legen wir den so genannten DB-Realm an. Das ist eine von JSecurity verwendete Klasse, um User, Passwörter, Rollen und Permissions gegen existierende Daten in der Datenbank zu prüfen. Das Plug-in bringt ein Skript mit, das uns diese Klasse generiert: *grails create-db-realm*.

Nach dem Aufruf erhält die Datei *realms/JsecDbRealm.groovy* die Standardimplementierung. Zusätzlich generiert das Plug-in noch einige *Jsec*-Domain-Klassen unter *grails-app/domain*, die wir löschen sollten, da wir ja bereits unsere eigenen Domain-Klassen modelliert haben. Den DB-Realm müssen wir jetzt noch auf unser Domain-Modell anpassen. Dies ist mehr oder weniger eine Ersetzung der *Jsec*-Klassen durch unsere eigenen (siehe mitgelieferter *JsecDbRealm.groovy* Sourcecode). Was nun noch fehlt, ist ein schöner Login-Dialog. Das heißt in MVC-Sprache: Eine View und ein Controller müssen her. Glücklicherweise könnten wir uns diese auch über ein Skript des JSecurity-Plug-ins erstellen lassen (und ggf. anpassen): *grails create-auth-controller*. Die erzeugten Dateien übernehmen wir eins zu eins. Einzig das Meta-Tag-Layout in der *views/auth/login.gsp* sollten wir auf unser Site-Layout umstellen (*<meta name="layout" content="site" />*), sodass der Login-Dialog in unserem Design erscheint (Abb. 5).

Nun können wir die Login/Logout-Funktion mit unserer vorhandenen Oberfläche verlinken. Das JSecurity-Plug-in bringt selber eine Reihe von GSP Tags mit, um den Login-Status zu überprüfen. Folgende Ergänzungen fügen wir in der Datei *views/layout/_header.gsp* hinzu:

```
<jsec:notAuthenticated>
  <g:link controller="auth" action="login">Login</g:link>
```

```
</jsec:notAuthenticated>
<jsec:authenticated>
  <g:link controller="auth" action="signOut">Logout</g:link> <jsec:principal/>
</jsec:authenticated>
```

Sie können sich nun einloggen (z. B. mit User *admin* und Passwort *geheim*), ausloggen und über das Backoffice-GUI beliebig neue Nutzer anlegen, editieren oder löschen. Fast hätten wir es vergessen: Das An- und Abmelden bringt natürlich allein recht wenig. Wir müssen vor allem sicherstellen, dass nur Nutzer mit der Rolle *ADMIN* auf das Backoffice zugreifen können. Hier bietet sich der Security-Filter von JSecurity an. Filter in Grails sind vergleichbar mit den Servlet-Filtern. Sie werden vor bzw. nach jedem Request aufgerufen, sodass hier Querschnittsaufgaben implementiert werden können, die bei jeder Anfrage erledigt werden sollen. Das JSecurity-Plug-in bringt eine erweiterte DSL mit, um sicherzustellen, dass bestimmte URLs nur von bestimmten Rollen gesehen werden dürfen. Listing 8 zeigt den Security-Filter für die MyBlog-Applikation.

Fazit

In diesem Artikel haben wir eine Reihe von Features in die Blogapplikation eingebaut und dabei viele Grails-Features kennengelernt. Hätte man mit einem herkömmlichen Java-Framework entwickelt, wäre sehr viel mehr Sourcecode zu programmieren gewesen. Das sollte die Effektivität von Grails für die Entwicklung von Webapplikationen vor Augen führen. Der Feierabend des Entwicklers wurde tatsächlich gerettet. Im letzten Teil dieses Tutorials werden wir das bisher stark vernachlässigte Thema „Testing“ betrachten. Außerdem wird es um die Frage gehen, wie Grails-Anwendungen gebaut (Build-Systeme) und installiert (deployt) werden können. ■

Marc-Oliver Scheele (mos) ist freiberuflicher IT-Berater und hilft seinen Kunden als Programmierer, Architekt und Scrum-Master bei der Realisierung von Softwareprojekten. Weitere Informationen und Kontakt unter http://www.moscon.de/

Links & Literatur

[1] M.O. Scheele: Rapid Blog Development mit Grails (Teil 1), Java Magazin 4.09

[2] HTML/CSS-Vorlage: http://www.freecsstemplates.org/preview/commission

[3] Layout-/Template-Framework: http://www.opensymphony.com/sitemesh/

[4] Library für URL-Rewrites: http://tuckey.org/urlrewrite/

[5] JS-Bibliotheken: http://www.prototypejs.org/ und http://script.aculo.us/

[6] JS-Grails-Plug-ins: http://www.grails.org/jQuery+Plugin und http://grails.org/Ajax

[7] Grails Web Flow: <http://grails.org/doc/1.1.x/guide/6. The Web Layer.html#6.5 Web Flow>

[8] Entwicklung von Grails-Plug-ins: <http://grails.org/doc/1.1.x/guide/12. Plug-ins.html>

[9] Grails-Plug-ins: http://grails.org/Plugins