Plus CD!

Stellenmarkt ▶ 50

Tool-Radar: JavaRebel ▶ 13

4.09

Java Magazin

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

S&S

Java magazin

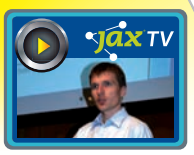
Java • Architekturen • SOA • Agile

www.javamagazin.de

Alle Infos zur Jax 09 ab Seite 63

CD-Inhalt

Web
- Canoo WebTest 2.6
- JBoss Portal 2.7.1
- Resin 3.2.1
- Apache Pluto 1.1.7



Adam Bien:
Java SE 6 u10, JavaFX SDK, Applets 2.0, WebKit – aber wartbar!

Frameworks
- Groovy
- Ajax-Frameworks ZK
- Grails 1.0.4

Datenserialisierung
- JBoss Serialization
- Hessian 3.2.1
- XStream 1.3.1
- Castor 1.3

Alle CD-Infos ▶ 3

Vorhang auf: Scala verändert die Java-Welt ▶ 100

„Es geht uns um Skalierbarkeit“

Im Gespräch: Scala-Mastermind Martin Odersky ▶ 105

Das Tutorial ▶ 34

Grails: Eigener Blog in 30 Minuten

Grails vereinigt moderne Programmieransätze mit der etablierten und leistungsstarken Java-Welt. Das Java-Magazin-Tutorial zeigt diesmal, wie man mit Grails in 30 Minuten einen voll funktionsfähigen Blog entwickeln kann.

Das Java Magazin Tutorial

D45867

04 4 194586 707500

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Blogentwicklung in 30 Minuten – mit Grails

Eine bekannte Szene: Der Ruby-on-Rails-Entwickler zeigt mit Stolz, wie elegant und schnell er seine Webapplikation entwickelt. Der Java-Entwickler steht etwas ungläubig und neidisch daneben. Doch damit ist jetzt Schluss: Grails vereinigt moderne Programmieransätze mit der etablierten und leistungsstarken Java-Welt.

von Marc-Oliver Scheele

Im dreiteiligen Tutorial möchten wir mit Ihnen in Form eines Hands-On-Tutorials die Vorteile von Grails erkunden. Wir werden ein voll funktionsfähiges Blogsystem programmieren und dabei die wichtigsten Funktionen von Grails kennenlernen. Der Autor selbst hat im letzten Jahr sein „Aha-Erlebnis" gehabt: Die Jahre zuvor hatte er ganz traditionell JEE-Enterprise- und Webapplikationen entwickelt, in jüngster Zeit meist auf Basis von Spring und Hibernate. Etwas genervt von dem Zeitaufwand, der aus langatmigen Konfigurationsdateien, Boilerplate-Code oder fehlenden Sprachfeatures entstand, sollte für ein neues Projekt nach technischen Alternativen gefahndet werden.

Die Wahl fiel auf Grails. Um es gleich vorwegzunehmen: Es war die richtige Entscheidung. Dank der Prinzipien DRY (Don't Repeat Yourself) und CoC (Convention over Configuration) sowie dem DDD-(Domain-driven-Development-)Ansatz von Grails konnte die Applikation kontoblick.de [1] in kürzester Zeit entwickelt werden. Auch andere Referenzen [2], z. B. das Portal von sky.com [3] mit über 3 Millionen Hits pro Tag, zeigen, dass Grails reif für den produktiven Einsatz ist.

Das Tutorial: Rapid Blog Development

- **Teil 1: Grails-Einführung – lauffähige Software in 30 Minuten**
- Teil 2: Plug-ins, Web-2.0-Features – die Anwendung wird erwachsen
- Teil 3: Automatisiertes Testing, Build-Systeme – die Qualitätssicherung

In 30 Minuten

Wir wollen in exemplarischen 30 Minuten versuchen, eine nutzbare Webapplikation zu entwickeln. Zum besseren Verständnis vorab ein paar Worte zur Grails-Architektur. Grails-

Applikationen werden in bewährter Manier in drei Schichten gebaut:

- Präsentationsschicht: Model View Controller (MVC) Pattern
- Businessschicht: in der Regel zustandslose und transaktionale Services (optional)
- Persistenzschicht: Object Relation Mapping auf Basis von Domain-Objekten

Technologisch nutzt Grails gängige und ausgereifte Java-Technologien unter der Haube: Für die Persistenzschicht wird auf einen JPA Provider gesetzt. Standardmäßig wird Hibernate mitgeliefert. Das Spring Framework liefert das Fundament von Grails. Es ist quasi auf allen Schichten zu finden. Neben der Dependency Injection (DI), der Steuerung von Transaktionsgrenzen und Vielem mehr wird auch die Präsentationsschicht auf der Basis von Spring MVC und Spring Web Flow gebaut. Die Views selbst können als JSPs oder besser als GSPs – den so genannten Groovy Server Pages – programmiert werden. Als Template Engine wird SiteMesh genutzt. Das ist erstmal nicht viel Neues für den erfahrenen Java-Programmierer. Wo ist das Geheimnis, und was macht Grails so produktiv?

Die Antwort ist: Groovy! Diese dynamische Sprache für die JVM wird konsequent von Grails verwendet. Sie ist syntaktisch und „bytecodemäßig" sehr eng mit dem klassischen Java verbunden. Das bringt wesentliche Vorteile gegenüber anderen JVM-Sprachen mit sich:

1. Der Java-Entwickler muss sich nicht lange einarbeiten, sondern ist sehr schnell in der Lage, Groovy-Code zu verstehen und zu schreiben.
2. Groovy und die Java-Sprache sind austauschbar: In einer Grails-Applikation können Groovy- und Java-Klassen/Libraries beliebig verwendet und ausgewechselt werden.

Durch Groovy wird der Programmiercode sehr ausdrucksstark und kann zur Laufzeit beliebig erweitert werden. Dadurch erhält Grails ungeahnte Fähigkeiten, die mit einem Framework auf Basis der „nackten" Java-Sprache so nicht möglich wären. Weitere Details sind dem Kasten „Groovy für Grails-Entwickler" zu entnehmen. Abbildung 1 fasst nochmals die Grails-Architektur zusammen.

Exploration Phase

Betrachten wir kurz die Anforderungen an unseren Blog. Ziel dieses ersten Tutorial-Teils ist es, den Backoffice-Teil des Blogsystems fertigzustellen. Das heißt, die Verwaltung von Artikeln, Kommentaren und Usern soll von der Datenbank bis zur HTML-Oberfläche implementiert sein. Im Einzelnen:

- Anlegen, Verändern, Löschen und Einsehen der Artikel, Kommentare und User
- Ablage der Daten in einer relationalen Datenbank
- Validierung der Daten auf Datenbank- und GUI-Ebene

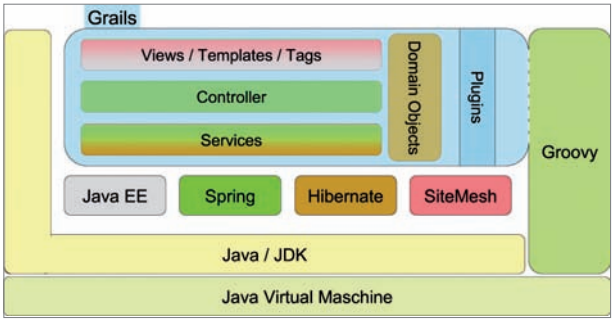


Abb. 1: Grails-Architektur

- Einfache HTML-Benutzeroberfläche für den Blogbesitzer/Administrator

Ab jetzt läuft die Stoppuhr. In 30 Minuten wollen wir Version 1.0 fertig entwickelt haben und nutzen können. Starten wir mit der Installation von Grails.

1. – 5. Minute – Grails installieren

Für dieses Tutorial wird die aktuelle Grails-Version 1.1.x genutzt. (Der Artikel wurde noch auf Basis von Grails 1.1 Beta 3 entwickelt.) Sämtlicher Code in diesem Tutorial sollte sowohl mit der Beta- als auch mit der Finalversion lauffähig sein. Voraussetzung für die Installation von Grails ist ein aktuelles JDK (>=1.5) auf Ihrem Linux-, Windows- oder Mac-OS-System inklusive gesetzter JAVA_HOME-Umgebungsvariable. Die Installation von Grails beschränkt sich auf das Herunterladen und Auspacken eines ZIP- oder TAR/GZ-Files und das Setzen zweier Umgebungsvariablen:

- Download der Binary-Distribution unter *http://www.grails.org/Download*
- Auspacken der ZIP-Datei in ein beliebiges Verzeichnis
- Umgebungsvariable GRAILS_HOME auf das Grails-Verzeichnis setzen
- Umgebungsvariable PATH um „$GRAILS_HOME/bin" erweitern

Freunde von Linux-Package-Managern finden mit etwas Glück auch eine Distribution für ihr System. Sie werden zwar aktuell nicht offiziell vom Grails-Projekt veröffentlicht, jedoch gibt es genug Freiwillige, die zeitnah zu einem Release ein entsprechendes Package bauen. Meist sind diese von der Grails-Download-Site verlinkt.

Ein erster Test für die erfolgreiche Installation ist der Aufruf des *grails*-Kommandos auf der Shell oder DOS-Eingabeaufforderung:

```
$ grails
Welcome to Grails 1.1-beta3 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: d:/dev_soft/grails-1.1-beta3
No script name specified. Use 'grails help' for more info or 'grails interactive' to enter
interactive mode
```

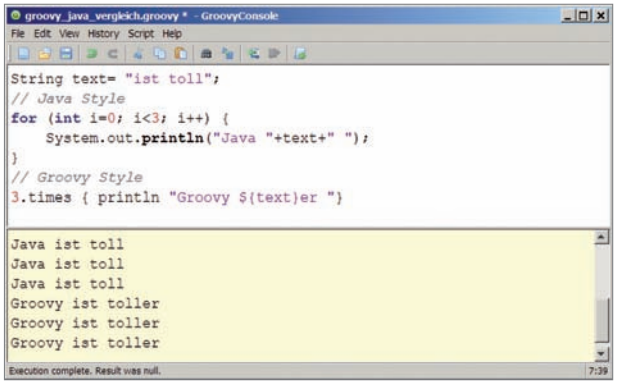


Abb. 2: Groovy Console

Das *grails*-Kommando ist quasi die Steuerzentrale für alle Grails-Aktivitäten. Mit folgendem Befehl erzeugt Grails für uns die Projektstruktur und generiert eine entsprechende Standardkonfiguration: *grails create-app blogapp*. Abbildung 3 zeigt die angelegte Projektstruktur inklusive Erläuterung der wichtigsten Verzeichnisse.

Ohne weiteres Installieren oder Programmieren können wir bereits unsere Applikation starten:

```
cd blogapp
grails run-app
```

Der Server wird durch den *run-app*-Befehl gestartet. In der Shell erscheint abschließend die Zeile *Server running. Browse to http://localhost:8080/blogapp*. Geben Sie den URL in Ihren Browser ein und Sie sehen eine freundliche Begrüßungsseite unserer Applikation.

Erstaunlich! Wieso können wir bereits unsere Webapplikation starten, ohne dass wir einen Servlet-Container wie z. B. Tomcat installiert haben? Die naheliegende Antwort: Grails bringt alles bereits mit. Standardmäßig wird Jetty [6] mitgeliefert, der beim *run-app*-Kommando gestartet wird. Auch eine Datenbank ist bereits vorhanden. Grails nutzt in der Standardkonfiguration als In-Memory-Datenbank das Produkt HSQLDB [7]. Das heißt jedoch keinesfalls, dass man auf Jetty als Applikationsserver oder HSQL als Datenbank festgelegt

Groovy für Grails-Entwickler

Zum Ausprobieren bietet es sich an, die aktuelle Groovy-Version herunterzuladen (http://groovy.codehaus.org/Download) und folgende Beispiele in der Groovy Console einzutippen und auszuführen (Abb. 2):

1. Java ist Groovy: Bis auf sehr wenige Ausnahmen wird ganz normaler Java-Code auch vom Groovy-Compiler verstanden. Dennoch gibt es meist elegantere Ausdrucksformen im „Groovy-Style“ (siehe z. B. Abb. 2)

2. Keine Semikolons/keine Klammern/keine Returns: Solange nicht mehrere Anweisungen in einer Zeile stehen, kann das Semikolon am Ende eines Statements weggelassen werden. Auch Klammern bei Methodenaufrufen werden meist nicht benötigt. Auf das explizite Return-Statement kann ebenfalls verzichtet werden, da der Wert der letzten Zeile einer Methode automatisch zurückgegeben wird.

3. Duck-Typing: Typinformationen können, müssen aber nicht, bei Deklarationen angegeben werden:

```
Integer staticVar = 1
def   dynamicVar = 2              //'def' means untyped
assert staticVar.class == dynamicVar.class
assert dynamicVar instanceof java.lang.Integer
dynamicVar = "Now I'm a String"
//staticVar = "Compile error if I want to be a String"
assert dynamicVar instanceof java.lang.String
```

4. Syntax für Listen und Maps: Für die häufig genutzten Datenstrukturen Listen/Arrays und Maps spendiert Groovy eine eigene Notation:

```
List myList = ['a','b','c','d','e']
assert 'd' == myList[3]
assert ['b','c','d'] == myList[1..3]
assert ['d','e'] == myList[3..-1]
myList << 'f'
assert 6, myList.size()
assert 'f' == myList[-1]
assert ['A','B','C','D','E','F'] == myList*.toUpperCase()
Map myMap = ['myKey':'myValue', 'zwei':2]
assert 2 == myMap['zwei']
assert 2 == myMap.zwei
myMap['drei'] = "New element"
assert myMap.containsKey('drei')
```

5. Erweitertes String Handling und Regex-Unterstützung: Umständliche String-Verkettungen und Escapings sind nicht mehr notwendig:

```
def orgString = 'Classic Java String'
assert 'Java' == orgString[8..11]
def gString =
            "Double-Quotes. Insert something:
${orgString.toUpperCase()}"
assert gString.endsWith(" CLASSIC JAVA STRING")
def multiLine = """ Cool
I can write over
many lines! """
assert 2 == multiLine.count('\n')
def regPattern = /Cool\W.*over/
assert multiLine =~ regPattern
```

6. Groovy-Wahrheit: Listen, Maps, Strings, Zahlen usw. haben Wahrheitswerte. Leere „Dinge“ sind false:

```
assert ! []
assert [6,7]
assert ! [:]
assert 'ok'
assert ! ''
assert 123
assert ! 0
```

7. Closures: Für Java in der Diskussion. In Groovy schon lange Realität:

```
void catchMe (Closure c) {
  try {
    c.call()
  } catch (Throwable t) {
    println "Closure executed. Exception catched!"
  }
}
def someAction = {
  def value = 5/0    // upps, exceptions
}
catchMe someAction
assert "I'm here. No Exception stops execution!"
```

8. Properties: Für Java in der Diskussion. In Groovy auch schon Realität: Es müssen keine Getter- und Setter-Methoden mehr geschrieben werden und es gibt eine einfache Zugriffssyntax für Properties:

ist. Die Applikation selbst kann als normales *war*-File auf jeden beliebigen JEE-konformen Applikationsserver deployt, die Datenbank über einen Konfigurationsparameter leicht ausgetauscht werden.

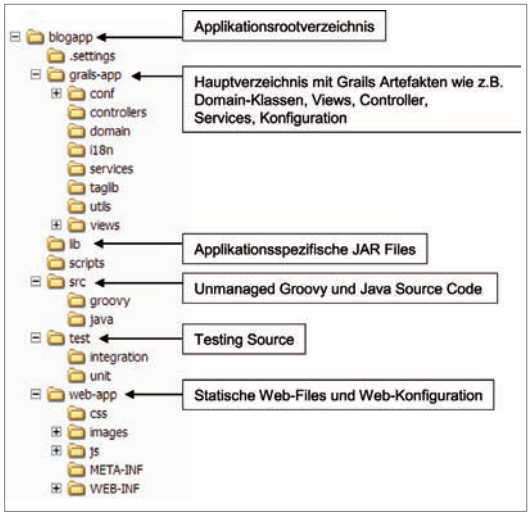


Abb. 3: Projektverzeichnisse

6. – 20. Minute – Das Datenmodell

Im Kern verfolgt Grails den Ansatz des Domain-driven Developments. Das heißt, die Fachlichkeit wird zunächst in Form der Businessobjekte, der so genannten Domain-Klassen, abgebildet. Hiervon ausgehend können dann leicht, und wenn gewünscht, auch durch automatische Generierung, weitere Artefakte wie Controller und


```
class MyBean {
  String name
  Date birthday

  // overwrite a getter
  String getName() {
    return "Your name is: $name"
  }
}

def mb = new MyBean(name:'mos')
mb.birthday = new Date()
mb.setBirthday( mb.birthday -3)  // three days back
assert "Your name is: mos" == mb.name
assert "Your name is: mos" == mb.getName()
assert mb.birthday < new Date()
```

9. Sicheres Dereferenzieren: Keine aufwändigen if … then-Fallunterscheidungen notwendig, um NullPointerExceptions zu vermeiden:

```
def myBean=null
assert null == myBean?.name
```

10. Builder und Parser: Für häufig benötigte Datenstrukturen werden einfach zu nutzende Builder und Parser angeboten:

```
def xml = new groovy.xml.MarkupBuilder()
xml.person (id:3) {
   name 'Karl Heinz'
   birthday '03.04.1974'
}                        // --> prints a nice xml doc to stdout
xml = "<person id='4'><name>mos</name></person>"
def person = new XmlSlurper().parseText(xml)
assert 'mos' == person.name.toString()
```

11. Meta Object Protocol (MOP) und das GDK: Durch so genannte Metaprogrammierung können Eigenschaften von Klassen und Objekten zur Laufzeit verändert werden. Dies macht sich Groovy auch selbst zunutze, indem es vorhandene JDK-Klassen sinnvoll erweitert. Werfen Sie doch einen Blick auf das GDK API [11]:

```
String.metaClass.repeat = { int count ->
 def stringValue = delegate
 return stringValue * count
}
println "fünfmal".repeat(5)      // java.lang.String
with new method repeat
assert "abc abc abc " == "abc ".repeat(3)
```

Groovy leistet noch einiges mehr, jedoch würden weitere Details diesen Kasten sprengen. Vieles ist selbsterklärend und wird sich im Rahmen des Grails Tutorials automatisch verdeutlichen. Wer tiefer einsteigen möchte, dem sei das Standardwerk von Dierk König „Groovy In Action" [4] und das Buch von Scott Davis „Groovy Recipes" [5] empfohlen.

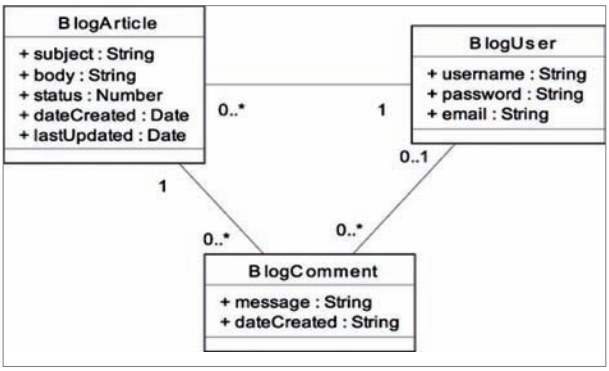


Abb. 4: Domain Model

Views erstellt werden. Kümmern wir uns also zunächst um unser Domain Model. Für den Anfang gilt es, drei Entitäten zu modellieren:

- Ein *BlogArticle* bezeichnet einen Blogeintrag, der eine Überschrift, einen Text und einige Meta-Daten beinhaltet.
- Ein *BlogUser* kennzeichnet eine Person mit Zugriff auf unseren Blog. Zunächst wird lediglich der Administrator, der per Definition Schreibzugriff hat, abgebildet.
- Ein *BlogComment* ist ein Kommentar zu einem BlogArticle. Er kann als anonymer Kommentar genutzt werden oder einem eingeloggten User zugeordnet sein.

Entwicklungsumgebungen für Grails

Im Grunde reicht für die Softwareentwicklung mit Grails ein einfacher Texteditor. Das Kommandozeilentool „grails" bringt viele Funktionen bereits mit, die sonst eine IDE rechtfertigen: Generierung von Code, Starten der Applikation, Starten von Tests, Debug-Möglichkeiten, Builds usw.

Die Leser dieses Grails Tutorials können alle Beispiele und Aufgaben entsprechend mit ihren Lieblingseditoren und dem Kommandozeilentool „grails" umsetzen. Dennoch: Für größere Projekte und verwöhnte Entwickler liefert eine ausgereifte Entwicklungsumgebung selbstverständlich viele weitere Vorteile. Wer nur Syntax-Highlighting für Groovy sucht und Grails-Kommandos über Menüeinträge konfigurieren möchte, wird sicher schon mit einem Tool wie jEdit [8] zufrieden sein.

Unter den großen Drei hat IntelliJ IDEA [9] aktuell die beste Grails-Unterstützung. Von Code Completion, Codeanalyse bis hin zu einem ausgereiften Debugger ist alles für Groovy und Grails vorhanden. Leider ist IntelliJ jedoch nicht kostenlos. NetBeans von Sun hat stark aufgeholt und bietet mittlerweile auch gute Unterstützung für Groovy und Grails [10]. Das Schlusslicht bezüglich Grails-Unterstützung ist aktuell Eclipse. Auch hier gibt es zwar ein entsprechendes Groovy-Plug-in, allerdings fehlt spezifische Grails-Unterstützung und wichtige Features, wie z. B. die Verwendung des Groovy/Java Joint Compilers sind noch nicht vollständig. Da sich die Firma SpringSource seit Herbst letzten Jahres aktiv an der Weiterentwicklung von Groovy/Grails beteiligt, ist insbesondere auch von einer besseren Eclipse-Unterstützung in naher Zukunft auszugehen.

Abbildung 4 zeigt das entsprechende Klassendiagramm unseres Domain Models.

Auch beim Anlegen unserer Domain-Klassen kann uns das *grails*-Kommando behilflich sein. Führen Sie folgende drei Befehle in der Shell aus (immer vorausgesetzt, Sie befinden sich im Verzeichnis *blogapp*!):

```
grails create-domain-class de.javamagazin.myblog.BlogArticle
grails create-domain-class de.javamagazin.myblog.BlogUser
grails create-domain-class de.javamagazin.myblog. BlogComment
```

Grails hat nach diesen Kommandos für uns im Verzeichnis *grails-app\domain* leere Groovy-Klassen in der gewünschten Package-Struktur angelegt. Zusätzlich wurden im Verzeichnis *\test\unit*-Klassen für das Unit Testing vorbereitet. Um diese werden wir uns im dritten Teil dieser Serie kümmern.

An dieser Stelle dürfen wir das erste Mal in die Programmierung einsteigen. Bei aller Zauberei schafft es Grails schließlich noch nicht, unsere Fachlichkeit zu erraten und zu generieren. Öffnen wir zunächst die *BlogArticle.groovy*-Datei. Wer sich nun fragt, welcher Editor bzw. welche IDE am besten genutzt werden sollte, sei auf den Kasten „Entwicklungsumgebungen für Grails" verwiesen.

Der Klasse *BlogArticle* fügen wir nun Properties inklusive Constraints und Beziehungen zu den anderen Klassen hinzu. Die Constraints beschreiben Einschränkungen und Eigenschaften der Properties, die sowohl im Datenbankschema als auch beim Validieren von User-Input Verwendung finden. Entsprechend verfahren wir mit den anderen beiden Klassen. Listing 1 bis Listing 3 zeigen den vollständigen Sourcecode der drei Domain-Klassen.

Betrachten wir die Klasse *BlogArticle*: Die Felder, die direkt den generierten DB-Tabellen-Spalten entsprechen, wurden als ganz normale Groovy Properties deklariert. Das Property *status* ist vom Typ eines selbstdefinierten Enums. Die Enum-Definition wurde direkt im gleichen File angehängt. Wer den Code trennen möchte, was etwas sauberer wäre, sollte das Enum in eine eigene Datei im Verzeichnis *src/groovy* auslagern.

Die Assoziationen zu anderen Klassen wurden mit der Property *author* und der *hasMany*-Deklaration *comments* abgebildet. Das ist auch schon alles, um aus Sicht des BlogArticles die Beziehungen herzustellen: Für eine *to-one*-Beziehung wird einfach ein Property vom entsprechenden Typ definiert. Eine *to-many*-Beziehung wird über die statische *hasMany*-Liste abgebildet. Die Gegenseite definiert die Beziehungen entsprechend, wodurch beliebige *1:1*-, *1:n*- und *n:m*-Beziehungen abgebildet werden können.

Eine Besonderheit findet sich in der Klasse *BlogComment* mit der *belongsTo*-Deklaration. Hierdurch wird der Kommentar zum Bestandteil (Aggregation) eines Blogartikels. Das hat direkte Auswirkung auf das Kaskadieren der Delete-Funktion. Wann immer ein Artikel gelöscht wird, werden automatisch alle zugehörigen Kommentare mitgelöscht.

Das Grails-Modul, das für die Interpretation der Domain-Klassen zuständig ist und daraus entsprechende Datenbank-

Listing 1: blogapp\grails-app\domain\de\javamagazin\myblog\BlogArticle.groovy

```
package de.javamagazin.myblog

class BlogArticle {

 // properties
 String    subject
 String    body
 ArticleStatus status = ArticleStatus.UNPUBLISHED
 Date      dateCreated
 Date      lastUpdated

 // relations
 BlogUser     author
 static hasMany = [comments:BlogComment]

 // properties constraints
 static constraints = {
  subject (blank:false, nullable:false, size:5..200,
unique:true)
  body  (blank:false, nullable:false, size:5..10000)
  status  (nullalbe:false)
  author  (nullable:false)
 }

}


enum ArticleStatus {
 UNPUBLISHED,
 PUBLISHED,
 DISABLED

}
```

Listing 2: blogapp\grails-app\domain\de\javamagazin\myblog\BlogUser.groovy

```
package de.javamagazin.myblog

class BlogUser {

 String username
 String password
 String email

 static hasMany = [comments:BlogComment,
        articles:BlogArticle]

 static constraints = {
  username   (blank:false, nullable:false, size:3..30,
                                          unique:true,
       matches:"[a-zA-Z]+")
  password   (blank:false, nullable:false, size:3..30)
  email    (email:true)
 }

}
```

Listing 3: blogapp\grails-app\domain\de\javamagazin\myblog\BlogComment.groovy

```
package de.javamagazin.myblog

class BlogComment {

 String message
 Date dateCreated

 BlogUser      user
 static belongsTo = [article:BlogArticle]

 static constraints = {
  message   (blank:false, nullable:false, size:5..1000)
  user    (nullable:true)    // anonymous comments are ok
 }

}
```

Listing 4: blogapp\grails-app\conf\BootStrap.groovy

```
import de.javamagazin.myblog.*

class BootStrap {

 def init = { servletContext ->

  BlogUser user = new BlogUser (username:'admin',
                                          password:'geheim',
              email:'grails@moscon.de')
  if ( !user.save() ) {
   log.error "SAVING OF USER FAILED: \n ${user.errors}"
   return
  }

  BlogArticle article = new BlogArticle()
  article.subject='Java Magazin bringt Grails Tutorial'
  article.body='"Es ist soweit:
In der neuesten Ausgabe des Java Magazins findet sich der
                                  erste Teil eines Grails Tutorials.
Insgesamt wird es zwei weitere Teile geben.
Der aufmerksame Leser wird am Ende des Tutorial in der
                          Lage sein, selber professionelle Grails
                                  Applikationen zu schreiben!
  '"
  article.author=user
  article.save()
  if (article.hasErrors()) {
   log.error "SAVING OF ARTICLE FAILED: \n ${article.errors}"
   return
  }

  BlogComment comment = new BlogComment()
  comment.message = "Danke für den Artikel.
              Mein konstruktives Feedback: Dies und das..."
  comment.article = article
  if ( !comment.save() ) {
   log.error "SAVING OF COMMENT FAILED:
                                  \n ${comment.errors}"
  }


 }

 def destroy = { }

}
```

Listing 5: Backoffice-Controller: blogapp\grails-app\controller\.....

```
// File: BlogArticleController.groovy

package de.javamagazin.myblog
class BlogArticleController {
 def scaffold = true
}


// File: BlogCommentController.groovy

package de.javamagazin.myblog
class BlogCommentController {
 def scaffold = true
}

// File: BlogUserController.groovy

package de.javamagazin.myblog
class BlogUserController {
 def scaffold = true
}
```

Listing 6: Backoffice-Controller: blogapp\grails-app\controller\.....

```
// File: BlogArticleController.groovy

package de.javamagazin.myblog
class BlogArticleController {
 def scaffold = true
}


// File: BlogCommentController.groovy

package de.javamagazin.myblog
class BlogCommentController {
 def scaffold = true
}

// File: BlogUserController.groovy

package de.javamagazin.myblog
class BlogUserController {
 def scaffold = true
}
```

definitionen und Datenbankzugriffe generiert, nennt sich GORM (Grails Object Relational Mapping). Es nutzt hierzu Hibernate, abstrahiert allerdings nochmals, indem es vereinfachte Syntax anbietet und diverse Zusatzfunktionen implementiert. Eine direkte Hibernate-Konfiguration via XML-Dateien und Annotation ist bei Grails also nicht notwendig, kann jedoch notfalls im Verzeichnis *grails-app/conf/hibernate* zusätzlich spezifiziert werden. In der zukünftigen Grails-Version ist geplant, dass man Hibernate gegen einen beliebigen JPA-Provider austauschen kann.

Ein weiteres sehr spannendes und leistungsfähiges Feature in den Domain-Klassen sind die Constraints. Wie in den Listings zu sehen ist, werden sie in einer statischen Closure namens *constraints* definiert. Für jedes Property können hier Bedingungen deklariert werden. Ein *nullable:false* definiert beispielsweise, dass die *subject*-Property nicht *null* sein darf. Bei der Datenbankschemagenerierung wird in diesem Fall ein

In-Memory-Datenbank als Datei

Grails verwendet in der Entwicklung standardmäßig die HSQL-Datenbank im In-Memory-Modus, ohne die Daten beim Server-Shutdown zu sichern. Falls die Daten auch nach einem Serverneustart zur Verfügung stehen sollen, muss die Datei *grails-app/conf/DataSource.groovy* bearbeitet werden. Sie beinhaltet Definitionen für die Datenbank-Connection und den Second Level Cache. Dies ist auch die richtige Stelle, um die Verbindung zu einer anderen Datenbank wie MySQL oder PostgreSQL zu konfigurieren.

Was weiter auffällt: Grails kennt die verschiedenen Laufzeitumgebungen *development, test und production*. Für jede Umgebung können eigene Datenbankeinstellungen vorgenommen werden. Starten Sie die Applikation mit dem Befehl *grails run-app*, läuft die Applikation per Default in der development-Umgebung. Ein deploytes WAR-File wiederum läuft in production.

Wie Sie der Datei entnehmen können, ist für ***production*** bereits eine Dateiablage für HSQLDB konfiguriert. Zusätzlich wird Hibernate mittels *dbCreate = "update"* veranlasst, Schemaänderungen automatisch durchzuführen, aber die Nutzdaten unverändert zu lassen. Definieren Sie analog die Einstellung für die Umgebung ***development*** und Ihre Daten bleiben zwischen den *grails run-app*-Aufrufen erhalten:

```
.....
environments {
 development {
  dataSource {
   dbCreate = "update"
   url = "jdbc:hsqldb:file:devDb;shutdown=true"
  }
 }
.....
```

Ein Nebeneffekt dieser Änderung ist, dass wir versuchen, die Datensätze, die wir beim Applikationsstart in der *grails-app/conf/BootStrap.groovy*-Datei instanziieren, wiederholt anzulegen. Das führt natürlich zu Unique-Constraint-Verletzungen. Möchte man sich die Fehlermeldung auf der Konsole sparen, einfach im BootStrapping prüfen, ob der entsprechende Datensatz schon existiert:

```
class BootStrap {
 def init = { servletContext ->

  if (BlogUser.findByUsername('admin')) {
     return
  }
 ...
```

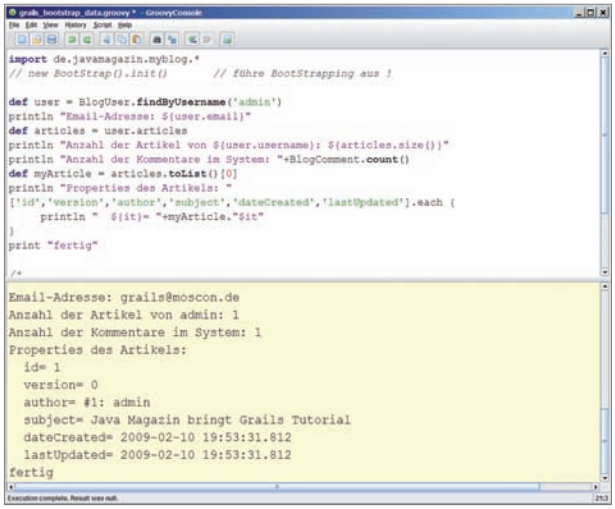


Abb. 5: Grails Console

Abb. 6: Backoffice GUI

not null Constraint auf das entsprechende Datenbankfeld gesetzt. Viele Constaints, wie *null-able*, *size* oder *unique* haben folglich direkte Auswirkung auf das Datenbankschema – andere wiederum wie *email* oder *matches* werden lediglich beim Validieren der User-Eingaben verwendet. In der Klasse *BlogUser* haben wir beispielsweise über eine *RegEx* definiert, dass ein *username* nur aus Buchstaben bestehen darf. Außerdem muss er im System eindeutig und zwischen 3 und 30 Zeichen lang sein.

GORM erlaubt noch weitere Einstellungen, um Lazy/Eager Loading, das Mapping auf vorgegebenen Tabellen bzw. Spaltennamen, Definition von *transient*-Properties, Konfiguration des Second Level Cache und einiges mehr zu definieren. Für den Anfang brauchen wir diese Features jedoch erstmal nicht.

Sourcecode des Tutorials

Der hier abgedruckte Sourcecode befindet sich mitsamt der kompletten Grails-Projektstruktur auf der Heft-CD. Eventuelle Aktualisierungen und Ergänzungen sind unter http://www.moscon.de/grailstutorial zu finden. Da der Artikel noch auf Basis der Betaversion von Grails 1.1 geschrieben wurde, wird unter dem URL auch ein Update der Blogapplikation auf Grails 1.1 (Release) zur Verfügung gestellt (soweit notwendig).

Datenmodell ausprobieren

Unser Datenmodell ist damit implementiert. Lassen Sie uns nun die Domain-Klassen testen. Da es noch kein GUI gibt, behelfen wir uns mit dem Bootstrapping-Feature von Grails. Dabei handelt es sich schlicht um die Datei *BootStrap.groovy* im Verzeichnis *grails-app/conf*. Der Trick: Sämtlicher Code in dieser Datei wird automatisch beim Starten des Servers ausgeführt. Wir nutzten dieses Feature, um unseren Administrator-User anzulegen und exemplarisch einen Blogeintrag inklusive Kommentar zu erzeugen. Listing 4 zeigt, wie in der *BootStrap.groovy* entsprechende Domain-Objekte instanziiert und persistiert werden.

Ein kleiner Hinweis bezüglich des Passworts: Normalerweise würden wir es selbstverständlich SHA1 verschlüsseln. Das sparen wir uns zunächst in V1.0 und werden es im nächsten Teil nachholen, wenn es um Security-Plug-ins von Grails geht.

Sie sehen, dass wir ganz normale Groovy-Objekte erzeugen, die Properties über Named-Parameter im Konstruktor oder als Zuweisung anlegen und anschließend die *save()*-Methode aufrufen. Anhand des Rückgabewerts von *save()* oder durch den nachgelagerten Aufruf *hasErrors()* wird geprüft, ob es Validierungsfehler gab und somit das Objekt nicht gespeichert werden konnte. Sie können es selbst ausprobieren, indem Sie, wie bereits oben erprobt, den Server starten: *grails run-app*. Wenn Sie keinen Fehler gemacht haben, startet der Server wie beim letzten Versuch. Falls ein Problem auftritt, wird durch unsere Log-Ausgabe eine entsprechende Fehlermeldung in der Konsole ausgegeben. Sie können die Fehler provozieren, indem Sie die Constraints verletzten. Probieren Sie zum Beispiel, die E-Mail-Adresse im User-Objekt zu zerstören oder entfernen Sie die Zuweisung *article.author=user*.

Der aufmerksame Leser wird sich bei Betrachtung von Listing 4 gefragt haben, wieso wir mittels *log.error()* Nachrichten ausgeben können, ohne vorher die *log* Property definiert zu haben. Dies ist eine Konvention von Grails. In jede von Grails (bzw. Spring) gemanagte Klasse wird automatisch eine *log* Property injectet. Dabei handelt es sich um einen Log4J *Logger*, der direkt wie im Beispiel genutzt werden kann.

Wie können wir uns die angelegten Objekte nun anschauen? Die schnelle Antwort: Via Datenbank-Query-Tool. Da wir aktuell ein solches Tool aber nicht zur Hand haben und das Ganze sich bei der momentan genutzten In-Memory-Datenbank auch etwas schwierig gestalten dürfte, greifen wir auf ein feines Tool zurück, das Grails mitliefert: Die Grails Console. Sie entspricht der Groovy Console, die wir im Kasten „Groovy

für Grails-Entwickler" kennengelernt haben. Zusätzlich wird jedoch der Spring Application Context inklusive Hibernate-Session hochgefahren, sodass voller Zugriff auf Grails-Features und auch die Datenbank besteht. Somit können wir die *Finder*- und *Query*-Methoden unserer Domain-Objekte verwenden, um die Datenbank auszulesen (oder auch, um Daten zu verändern). Abbildung 5 zeigt die Grails Console in Aktion. Probieren Sie es aus: *grails console*.

Da die Konsole, anders als beim Serverstart, unseren Bootstrapping-Code nicht automatisch ausführt, müssen wir dies im ersten Schritt selbst tun. Tippen Sie Folgendes in die Grails Console und drücken zum Ausführen STRG-R: *new BootStrap().init()*. Löschen Sie nun die Zeile und probieren Sie nach Belieben folgende Zeilen aus:

```
import de.javamagazin.myblog.*
def user = BlogUser.findByUsername('admin')
println "Email-Adresse: ${user.email}"
def articles = user.articles
println "Anzahl der Artikel von ${user.username}: ${articles.size()}"
println "Anzahl der Kommentare im System: "+BlogComment.count()
def myArticle = articles.toList()[0]
println "Properties des Artikels: "
['id','version','author','subject','dateCreated','lastUpdated'].each {
  println "  ${it}= "+myArticle."$it"
}
print "fertig"
```

Gleich das erste Kommando *findByUsername()* ist ein weiteres Hammer-Feature von Grails: Die so genannten „Dynamic Finders". Jede Domain-Klasse versteht Methoden wie *findByProperty1AndProperty2* (*value1*,*value2*). Grails generiert hieraus eine SQL-Abfrage mit entsprechender *where*-Klausel. Neben *findBy…()*, die immer ein Element zurückgibt, existieren noch *findAllBy…()* und *countBy…()*. Für komplizierte Datenbankabfragen können die von Hibernate bekannten HQL-Kommandos oder Criterias verwendet werden. Ein Beispiel für eine Abfrage über zwei Tabellen (*join*) mittels HQL:

```
def list = BlogArticle.executeQuery(
    "select subject from ${BlogArticle.name} a where a.author.username=:name",
    [name:'admin'])
println 'Titel: '+list[0]
```

Die letzten Zeilen des obigen Beispiels beinhalten noch eine kleine Überraschung. Für den gefundenen Artikel werden verschiedene Properties ausgegeben. Das Ergebnis:

```
Properties des Artikels:
 id= 1
 version= 1
 author= de.javamagazin.myblog.BlogUser : 1
 subject= Java Magazin bringt Grails Tutorial
 dateCreated= 2009-02-07 18:33:56.468
 lastUpdated= 2009-02-07 18:38:17.625
```

Was verwundert: Unser Domain-Objekt beinhaltet die Properties *id* und *version*. Beide sind analog auch in der zugehörigen Datenbanktabelle zu finden. Außerdem haben die von uns definierten Properties *dateCreated* und *lastUpdated* einen Zeitstempel, ohne dass wir einen Wert im *BootStrap.groovy* gesetzt hätten.

Die Erklärung liefert das von Ruby on Rails bekannte Prinzip „Convention over Configuration". Per Konvention spendiert Grails jedem Domain-Objekt ein Feld *id*, um dort automatisch den Primärschlüssel jedes Objekts zu setzen. Das Feld *version* wird für das optimistische Locking von Hibernate verwendet. Es wird bei jedem Schreibzugriff hochgezählt und stellt sicher, dass bei gleichzeitigem Schreibzugriff auf ein Objekt Daten nicht stillschweigend überschrieben werden. Die Properties *dateCreated* und *lastUpdated* sind vordefinierte Namen, die Grails erkennt und jeweils automatisch das Erzeugungsdatum respektive das letzte Änderungsdatum setzt. Vermutlich sind diese Konventionen für 95 Prozent aller Projekte sinnvoll – machen Sie sich keine Sorgen bezüglich der restlichen fünf Prozent: Sämtliche Konventionen können meist sehr einfach durch ein Stück Konfiguration überschrieben werden.

21. – 25. Minute – das Backoffice GUI

Gönnen wir uns weitere 5 Minuten, um die erste Version unseres Backoffice GUI zu erstellen. Die Anforderungen sind hier

recht einfach: Artikel, Kommentare und User sollen eingesehen, angelegt, verändert und gelöscht werden können. Mit anderen Worten: Für jedes unserer Domain-Objekte gilt es CRUD- (Create-Read-Update-and-Delete-)Funktionen inklusive HTML-Oberfläche zu implementieren.

Klingt erst einmal nach einer langweiligen und langatmigen Implementierungsaufgabe. Da es sich jedoch um ein Standardproblem vieler Applikationen handelt, liefert Grails glücklicherweise auch hierfür eine Standardlösung. Mittels so genanntem *Scaffolding* generiert uns Grails auf Basis der vorhandenen Domain-Klassen sämtlichen Code, der für unsere Aufgabe notwendig ist. Man unterscheidet zwischen statischem und dynamischem Scaffolding. In der statischen Variante werden die *grails generate-…*-Kommandos genutzt und sichtbare Groovy- bzw. GSP-Dateien werden mit entsprechendem Sourcecode generiert. Das ist immer dann sinnvoll, wenn man den Code inspizieren und verändern möchte. Da wir zunächst aber nur die Standardfunktionen benötigen, sind wir mit dem dynamischen Scaffolding besser bedient. Zunächst legen wir für jede unserer Domain-Klassen einen Standard-Controller an:

> Für den Backoffice-Teil des Blogs müssen wir nur 150 Zeilen Code schreiben.

```
grails create-controller de.javamagazin.myblog.BlogArticle
grails create-controller de.javamagazin.myblog. BlogComment
grails create-controller de.javamagazin.myblog.BlogUser
```

Öffnen Sie die Datei *BlogArticleController*, die im entsprechenden Package unter *grails-app/controller* zu finden ist. Die Zeile mit der *index* Property können wir löschen. Stattdessen definieren wir die Property *scaffold*, um das dynamische Scaffolding zu aktivieren: *def scaffold = true*. Probieren wir das Scaffolding aus. Einfach den Server wieder starten und den Start-URL aufrufen: *grails run-app*.

Im Browser sehen wir die Namen der drei Controller. Klicken wir auf *BlogArticleController*, sehen wir, dass unser im Bootstrapping angelegter Datensatz sichtbar ist und sämtliche CRUD-Funktionen verfügbar sind. Gehen Sie analog für die beiden anderen Controller vor. Sie werden ein weiteres sehr nützliches Feature der Grails-Umgebung zu schätzen lernen: Die Änderungen im Controller, also in diesem Fall das Aktivieren des Scaffoldings, sind direkt im Browser verfügbar, ohne dass der Server neu gestartet werden muss. Listing 5 zeigt nochmals den (wenigen) Sourcecode unserer Backoffice Controller. Abbildung 6 zeigt das Backoffice GUI beim Anlegen eines neuen Artikels (inklusive Fehlermeldung).

Ein kleiner Optimierungstipp für das Backoffice GUI: Referenzen zu anderen Objekten werden mittels der *toString()*-Methode der jeweiligen Objekte angezeigt. Standardmäßig zeigt Grails hier den Klassennamen und den Primary Key an. Wer das ändern möchte, überschreibt in den Domain-Klassen einfach die *toString()*-Methode. (Achtung: Nach Änderung der Domain-Objekte könnte ein Restart des Servers notwendig sein und eventuell zusätzlich das Kommando grails clean.)

26. – 30. Minute – Fazit

Dank Grails haben wir jetzt sogar noch fünf Minuten übrig, um ein Fazit zu ziehen. In diesem ersten Teil haben wir die Grails-Philosophie und die grundlegende Architektur kennengelernt. Sie wissen nun, wie man seine Daten modelliert, persistiert und validiert. Weiter haben wir gesehen, wie schnell man mittels Scaffolding voll funktionale Oberflächen bauen kann. Unser Backoffice-Teil des Blogs ist nun fertig und kann genutzt werden. Dazu haben wir weniger als 150 Zeilen Code geschrieben (und das schon inklusive Leerzeilen und Code für Testdaten). Mit der Applikation können Sie nun erste Blogartikel formulieren und verwalten. Um sie dauerhaft zu speichern, sollten Sie die Datenbank auf Dateibasis umstellen (Kasten: „In-Memory-Datenbank als Datei“).

Sollten Sie die Applikation außerhalb der Grails-Umgebung nutzen wollen, können Sie mittels des Kommandos *grails war* ein *war*-File erzeugen, das Sie in jedem beliebigen Webcontainer (z. B. Tomcat) deployen können. Im nächsten Teil programmieren wir die Blogansicht für den Konsumenten, steigen tiefer in die View-, Controller- und Serviceimplementierung ein und beleuchten u. a. die Themen Grails-Plug-ins, Autorisierung sowie Ajax. ■

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] Grails based: www.kontoblick.de
[2] Grails – Success Stories: grails.org/Success+Stories
[3] Grails based: www.sky.com
[4] Groovy in Action: www.manning.com/koenig/
[5] Groovy Recipes: www.pragprog.com/titles/sdgrvr/groovy-recipes
[6] Jetty-Webcontainer: jetty.mortbay.com/jetty
[7] HSQLDB-Datenbank: hsqldb.org
[8] jEdit-Editor: www.jedit.org
[9] IDE IntelliJIDEA: www.jetbrains.com/idea/
[10] IDE NetBeans: www.netbeans.org/features/groovy/index.html
[11] GDK: groovy.codehaus.org/groovy-jdk/