Plus CD!

Stellenmarkt ▶ 61 Performance-Killern auf der Spur ▶ 64

6.09

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

Java Magazin

Java magazin

Java • Architekturen • SOA • Agile

www.javamagazin.de

CD-Inhalt

- WebCastellum
- Spring Integration 1.0.2
- JSFUnit 1.0.0. GA
- JGAP 3.4.3
- ScalaTest
- Selenium
- HtmlUnit 2.4

Marc Gille:
SOA, SaaS – und ich?

Alle CD-Infos ▶ 3

D 45867

Datenträger enthält Info- und Lehrprogramme gemäß § 14 JuSchG

WEB APPLICATION SECURITY

Was JEE-Architekten beachten müssen ▶ 46

Funktionale Programmierung mit Scala ▶ 23

Das Tutorial ▶ 38

Bauen und Testen mit Grails

Die voll funktionsfähige Blog-Applikation mit Grails ist fertig, jetzt kommt die Kür. Das Tutorial zeigt Testmöglichkeiten sowie die unterstützten Build-Systeme. Dann klappt's auch mit der Qualitätssicherung.

Das Java Magazin Tutorial

Bauen und Testen mit Grails

Die letzten beiden Teile des Tutorials haben sich mit der Implementierung der Blogapplikation beschäftigt. Da dieses Dank Grails im Handumdrehen erledigt war, bleibt uns im letzten Teil nun noch Zeit, einige nichtfunktionale Eigenschaften zu betrachten.

von Marc-Oliver Scheele

In diesem Teil des Tutorials soll es um die Themen der Wartbarkeit, Qualität und Integration in die Laufzeitumgebung gehen [1]. Hierzu werden wir uns im Kern die Testmöglichkeiten einer Grails-Applikation sowie die unterstützten Build-Systeme anschauen.

Testabdeckung

Da Grails die agile Softwareentwicklung propagiert und unser Werkzeug mit Groovy eine dynamische Programmiersprache ist, hätten die Tests eigentlich im Zuge der Implementierung geschrieben werden sollen. Dieser als TDD („Test-driven Development“) bezeichnete Ansatz ist relativ schmerzlos mit Grails zu bewerkstelligen: Aufgrund der entkoppelten Komponenten (dem Spring Framework sei Dank), der dynamischen Sprache Groovy und den vielen von Grails mitgelieferten Mock-Klassen, sind automatisierte Tests meist mit minimalem Aufwand zu schreiben. Versichern wir uns zunächst, wie der Stand der Testabdeckung innerhalb unserer MyBlog-Anwendung aussieht. Hierzu nutzen wir das Plug-in Code Coverage [2]. Es kapselt das Tool Cobertura [3] und ermöglicht eine Aussage, wie viel Prozent unseres Sourcecodes durch automatisierte Tests abgedeckt ist. Das Plug-in installieren wir mit dem bekannten Kommando:

```
mos@~/myblog/blogapp$ grails install-Plug-in
code-coverage
```

Code Coverage hängt sich in das Testkommando von Grails ein. Mit *grails test-app* rufen wir nun sämtliche Tests unserer Grails-Applikation auf. Nach erfolgreichem Durchlauf finden sich innerhalb unserer Projektstruktur unter *test/reports/* sämtliche Testberichte. Unter *html* liegen die JUnit-Reports, die aktuell noch nicht so spannend sind, da wir noch keine Tests geschrieben haben. Aufregender ist das Verzeichnis *cobertura*, das uns eine Aussage über die Testabdeckung gibt (Abb. 1).

Das Tutorial: Rapid Blog Development

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

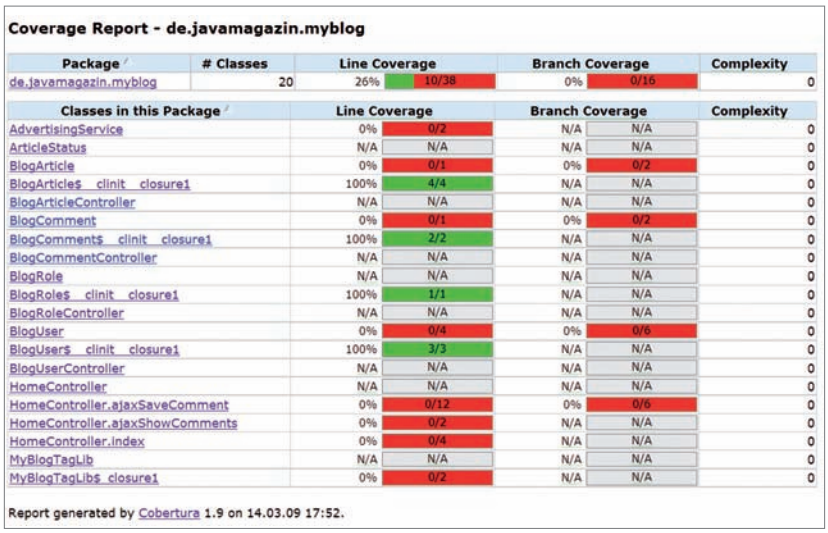
Teil 2: Frontend, Web-2.0-Features, Plug-ins – die Anwendung wird erwachsen

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

Sourcecode und Grails-Version

Der hier abgedruckte Sourcecode befindet sich mitsamt der kompletten Grails-Projektstruktur auf der Heft-CD. Teilweise wurde der Sourcecode in diesem Artikel zur besseren Übersicht gekürzt. Dieser ist selbstverständlich vollständig auf CD oder im Web unter http://www.moscon.de/grailstutorial herunterzuladen.

Wie zu erwarten, ist unsere Testabdeckung, erkennbar an den roten Balken, aktuell sehr schlecht. Lediglich die Constraint-Definitionen werden vom Code-Coverage-Plug-in von Anfang an als getestet angezeigt. Krempeln wir also die Ärmel hoch und schreiben automatisierte Tests, um die Qualität und vor allem die Wartbarkeit unserer Anwendung zu verbessern. Wie bei jeder anderen Software haben wir als Entwickler auch bei Grails die Möglichkeit, Tests auf verschiedenen Abstraktionsebenen zu schreiben. Da wären zunächst die klassischen Unit-Tests. Alle Abhängigkeiten zu anderen Objekten bzw. Schnittstellen zur Datenbank und Fremdsystemen werden durch so genannte Stubs oder Mocks simuliert. Die Unit-Tests prüfen somit immer isolierte Funktionen und sind extrem schnell in der Ausführung. Sie sollten im Sinne von TDD stets parallel zur Implementierung ausgeführt werden und sind in den IDEs meist auf Knopfdruck unmittelbar ausführbar. Die nächste Stufe bilden die Integration-Tests, bei denen der Kontext verfügbar ist. Das heißt bei einer Grails-Anwendung insbesondere der Zugriff auf die Datenbank und über Springs *ApplicationContext*-Zugriff auf andere Komponenten. Diese Tests bieten sich folglich immer dann an, wenn komplexere Datenbankabfragen oder aber Querschnittsfunktionen über mehrere Schichten getestet werden sollen. Zu guter Letzt steht in der Toolbox noch der Funktionstest zur Verfügung. Hierbei wird End-to-End getestet. Das heißt, der Applikationsserver wird hochgefahren, die Requests des Clients werden simuliert und die gesendeten Dokumente der Anwendung werden auf Korrektheit geprüft. Diese Tests automatisieren also das sonst immer zu wiederholende und stupide Durchklicken einer Anwendung im Browser.



Coverage Report - de.javamagazin.myblog

Package	# Classes	Line Coverage	Branch Coverage	Complexity
de.javamagazin.myblog	20	26% 10/38	0% 0/16	0

Classes in this Package	Line Coverage	Branch Coverage	Complexity
AdvertisingService	0% 0/2	N/A N/A	0
ArticleStatus	N/A N/A	N/A N/A	0
BlogArticle	0% 0/1	0% 0/2	0
BlogArticle$ clinit closure1	100% 4/4	N/A N/A	0
BlogArticleController	N/A N/A	N/A N/A	0
BlogComment	0% 0/1	0% 0/2	0
BlogComment$ clinit closure1	100% 2/2	N/A N/A	0
BlogCommentController	N/A N/A	N/A N/A	0
BlogRole	N/A N/A	N/A N/A	0
BlogRole$ clinit closure1	100% 1/1	N/A N/A	0
BlogRoleController	N/A N/A	N/A N/A	0
BlogUser	0% 0/4	0% 0/6	0
BlogUser$ clinit closure1	100% 3/3	N/A N/A	0
BlogUserController	N/A N/A	N/A N/A	0
HomeController	N/A N/A	N/A N/A	0
HomeController.ajaxSaveComment	0% 0/12	0% 0/6	0
HomeController.ajaxShowComments	0% 0/2	N/A N/A	0
HomeController.index	0% 0/4	N/A N/A	0
MyBlogTagLib	N/A N/A	N/A N/A	0
MyBlogTagLib$ closure1	0% 0/2	N/A N/A	0

Report generated by Cobertura 1.9 on 14.03.09 17:52.

Abb. 1: Testabdeckung mit Cobertura

Unit-Tests

Standardmäßig wird von Grails für jedes über *grails create-xxx* angelegte Artefakt (Domain-Klassen, Controller usw.) ein leerer Unit-Test mit angelegt. Die Testklassen liegen im Projektverzeichnis unter *test/unit*. Falls nicht vorhanden, kann die Testklasse für unseren *HomeController* auch wie folgt angelegt werden:

```
grails create-unit-test de.javamagazin.myblog.
                                    HomeController
```

Exemplarisch wollen wir die Action zum Anlegen eines Blogkommentars testen. Sie erhält die Blog-ID als Parameter und legt daraufhin einen neuen Kommentar an. Falls keine Constraints des Kommentars verletzt sind (z. B. Kommentar muss mindestens 5 Zeichen beinhalten), hat der Aufruf der *save()*-Methode Erfolg und ein neuer Kommentar wird gespeichert. Ansonsten enthält das neue Kommentarobjekt einen Fehler, der an die GUI übergeben wird (Listing 1 und letzter Tutorial-Teil). Folgendes wollen wir im ersten Schritt testen:

- Im Erfolgsfall wurde ein neuer Kommentar angelegt, das Model beinhaltet den Blogartikel und eine Prüfung, ob ein Nutzer zu diesem Zeitpunkt authentifiziert war.
- Im Fehlerfall wurde kein Kommentar angelegt, das Model beinhaltet den fehlerhaften Kommentar inklusive Fehlerbeschreibung.
- Das unterschiedliche Verhalten mit eingeloggtem Nutzer versus anonymem Nutzer werden wir nicht testen. Dieses sollte im Anschluss eine leichte Übung für den motivierten Leser sein.

Listing 2 zeigt unsere Testklasse. Die Testklasse selbst sollte von ControllerUnitTest ableiten, sodass out-of-the-box die notwendigen Mocking-Funktionen zur Verfügung stehen. In der *setUp()*-Methode wird die Initialisierung je auszuführenden Test implementiert. An dieser Stelle gilt es folglich, die Abhängigkeiten zu dritten Daten und Funktionen zu „mocken". Die Daten, die in Form von Domain-Objekten gelesen und geschrieben werden, sind mit der Methode *mockDomain()* zu simulieren. Dabei kann eine Liste von Objekten angegeben werden, um vorhandene Datenbankeinträge abzubilden. Für die *BlogArticle* hinterlegen wir so einen Datensatz. Die *BlogComments* sind zunächst „leer". Richtig hübsch wird es bei dem Mocking der JSecurity-Zugriffe. Hierzu simulieren wir zunächst den Aufruf auf die statische Methode *getSubject()* mittels der *mockFor*-Methode. Dieser Ansatz erlaubt, jede Instanz- und/oder Klassenmethode zu mocken und ist nahezu mit der *MockFor*-Klasse von Groovy identisch. Die Rückgabe, die als Closure abgebildet wird, ist wiederum ein simulierter Aufruf der Methode *isAuthenticated()*. Hierbei haben wir das Feature von Groovy genutzt, Map-Datenstrukturen als Klassenimplementierungen zu verwenden. Im Ergebnis wird jeder Aufruf von *SecurityUtils.subject.isAuthenticated()* in der *ajaxSaveComment*-Action den Wert *false* zurückliefern. Die Range (1..2) gibt die Anzahl der erwarteten Aufrufe dieser Methode an. Der Wert muss hier auf 2 stehen, da sich in Grails V1.1 ein kleiner Initialisierungs-Bug eingeschlichen hat [4].

Die folgenden Testmethoden *testAjaxSaveComment…()* prüfen die er-

warteten Ergebnisse im Gut- und im Fehlerfall. Mittels der Variable *mockParams* (eine Map) können die simulierten Request-Parameter gesetzt werden. Nachdem die Action via *controller.ajaxSaveComment()* aufgerufen wurde, können über die Variable *renderArgs* die an die GUI übergebenen Werte ausgelesen und getestet werden. Unsere Liste *comments* wird um das neue Objekt ergänzt, falls die *save()*-Methode Erfolg hatte. Zu guter Letzt können wir über *securityMock.verify()* sicherstellen, dass unsere Authentifizierungsabfrage stattgefunden hat. Starten wir die Tests mit *grails test-app –unit –nocoverage*, sollten wir zwei erfolgreiche Testdurchläufe in der Konsole sehen (die Parameter verhindern das zeitaufwendige Ausführen der Integration-Tests und des obigen Code Coverage):

```
Running tests of type 'unit'
-------------------------------------------------------
Running 2 unit tests...
Running test de.javamagazin.myblog.
                    HomeControllerTests...PASSED
Tests Completed in 1157ms ...
-------------------------------------------------------
Tests passed: 2
Tests failed: 0
```

Falls ein Fehler existiert (oder provoziert wird, indem beispielsweise das „!“ vor dem *comment.save()* in der Action entfernt wird), sollte man sich den Fehlerreport unter */test/reports/html/index.html* zu Gemüte führen (Abb. 2).

Ein weiterer Tipp: Lassen Sie das Code-Coverage-Plug-in erneut laufen (*grails test-app*). Betrachten Sie die Klasse *HomeController*, werden Sie feststellen, dass wir die Testabdeckung ordentlich erhöht haben. Der Fleiß hat sich gelohnt.

Integrationstests

Motiviert von der erreichten Testabdeckung, stürzen wir uns direkt auf den nächsten Test, um unsere Testquote noch weiter zu erhöhen. Die Action *index* im *HomeController* liefert veröffentlichte *BlogArticle* aus. Sicherzustellen ist hier, dass die neuesten Artikel oben in der Liste

Listing 1: Action ajaxSaveComment im HomeController

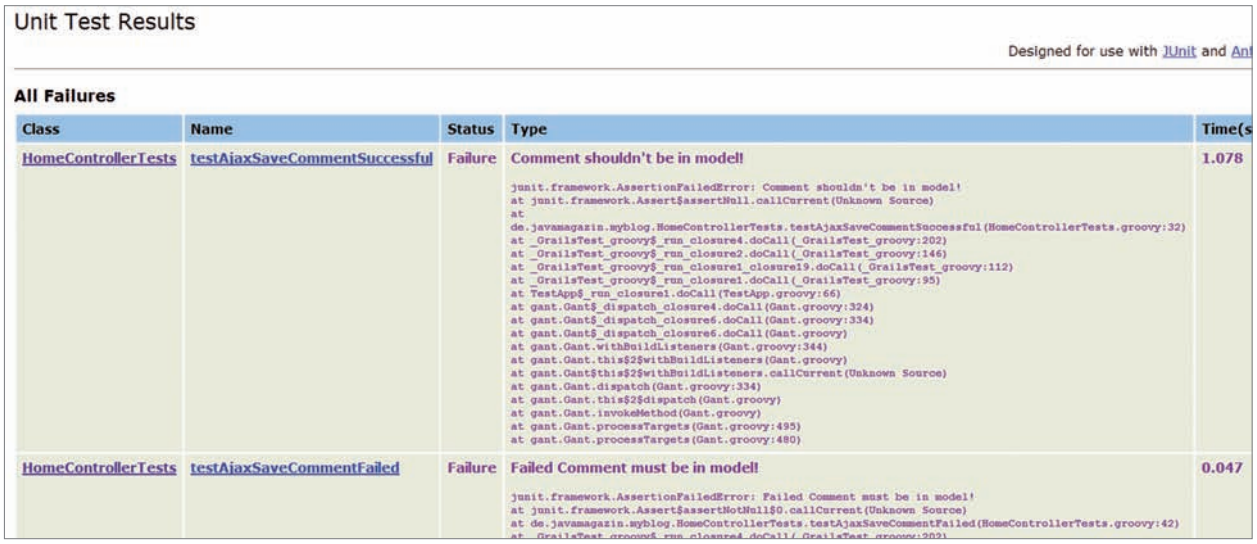
```
def ajaxSaveComment = {
  Map model = [:]
  BlogArticle article = BlogArticle.read(params.articleId)
  BlogComment comment = new BlogComment()
  comment.message = params.message
  comment.article = article
  if (SecurityUtils.subject.isAuthenticated()) {
    comment.user = BlogUser.findByUsername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

Listing 2: Unit-Test: HomeControllerTests

```
package de.javamagazin.myblog

import grails.test.*
import org.jsecurity.*
import org.jsecurity.subject.*

class HomeControllerTests extends ControllerUnitTestCase {

  HomeController controller
  def securityMock
  def comments = []

  protected void setUp() {
    super.setUp()
    // mock article domain objects
    def blogArticle = [ new BlogArticle(id:11, subject:'Wir testen...') ]
    mockDomain (BlogArticle, blogArticle)
    // mock comment domain objects
    mockDomain (BlogComment,comments)
    // mock security object
    securityMock = mockFor(SecurityUtils)
    securityMock.demand.static.getSubject(1..2)
          {-> [isAuthenticated:{false}] as Subject}
    // init object under test
    controller = new HomeController()
  }

  void testAjaxSaveCommentSuccessful() {
    mockParams.articleId = 11
    mockParams.message = 'Mein Kommentar zum Blog'
    controller.ajaxSaveComment()
    assertNotNull renderArgs.model.article
    assertNull "Comment shouldn't be in model!",
          renderArgs.model.newComment
    assertEquals 1, comments.size()
    assertEquals 'Mein Kommentar zum Blog', comments[0].message
    securityMock.verify()
  }

  void testAjaxSaveCommentFailed() {
    mockParams.articleId = 11
    mockParams.message = 'x'
    controller.ajaxSaveComment()
    assertNotNull "Failed Comment must be in model!",
          renderArgs.model.newComment
    assertTrue renderArgs.model.showComments
    assertTrue renderArgs.model.newComment.hasErrors()
    assertEquals 0, comments.size()
  }

}
```

Listing 3: Action index im 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
  }
```



Unit Test Results

Designed for use with JUnit and Ant

All Failures

Class	Name	Status	Type	Time(s
HomeControllerTests	testAjaxSaveCommentSuccessful	Failure	Comment shouldn't be in model! junit.framework.AssertionFailedError: Comment shouldn't be in model! at junit.framework.Assert$assertNull.callCurrent(Unknown Source) at de.javamagazin.myblog.HomeControllerTests.testAjaxSaveCommentSuccessful(HomeControllerTests.groovy:32) at _GrailsTest_groovy$_run_closure4.doCall(_GrailsTest_groovy:202) at _GrailsTest_groovy$_run_closure2.doCall(_GrailsTest_groovy:146) at _GrailsTest_groovy$_run_closure1_closure19.doCall(_GrailsTest_groovy:112) at _GrailsTest_groovy$_run_closure1.doCall(_GrailsTest_groovy:95) at TestApp$_run_closure1.doCall(TestApp.groovy:66) at gant.Gant$_dispatch_closure4.doCall(Gant.groovy:324) at gant.Gant$_dispatch_closure6.doCall(Gant.groovy:334) at gant.Gant$_dispatch_closure6.doCall(Gant.groovy) at gant.Gant.withBuildListeners(Gant.groovy:344) at gant.Gant.this2withBuildListeners(Gant.groovy) at gant.Gant$this$2$withBuildListeners.callCurrent(Unknown Source) at gant.Gant.dispatch(Gant.groovy:334) at gant.Gant.this2dispatch(Gant.groovy) at gant.Gant.invokeMethod(Gant.groovy) at gant.Gant.processTargets(Gant.groovy:495) at gant.Gant.processTargets(Gant.groovy:480)	1.078
HomeControllerTests	testAjaxSaveCommentFailed	Failure	Failed Comment must be in model! junit.framework.AssertionFailedError: Failed Comment must be in model! at junit.framework.Assert$assertNotNull$0.callCurrent(Unknown Source) at de.javamagazin.myblog.HomeControllerTests.testAjaxSaveCommentFailed(HomeControllerTests.groovy:42)	0.047

Abb. 2: Testreport von Grails

erscheinen und dass nur Artikel angezeigt werden, die den Status PUBLISHED haben. Betrachten wir die Action (Listing 3), so ist recht wenig Sourcecode zu erkennen. Zu testen ist hier insbesondere, ob unsere Datenbankabfragen in Form des Dynamic Finders korrekt sind. Hier muss ein Integrationstest unter Verwendung der echten Datenbank herhalten. Zunächst legen wir unsere Testklasse an (falls noch nicht vorhanden):

```
grails create-integration-test de.javamagazin.
                    myblog.HomeControllerIntegration
```

Vor jedem Integrationstest stellt Grails sicher, dass der Test eine saubere Datenbank vorfindet. Das heißt, die Datenbank wird geleert und die Bootstrap-Datei wird ausgeführt. In unserem Fall kann also jeder Test sicher sein, dass die in *Bootstrap.groovy* angelegten Testdaten verfügbar sind. Als Entwickler sollte man die relevanten Preconditions überprüfen, um Seiteneffekte bei Änderung der Bootstrap-Testdaten ausschließen zu können. In unserem Fall prüfen wir, dass mehr als 20 Artikel vorhanden sind, wobei mindestens zehn den Status PUBLISHED haben müssen und mindestens ein UNPUBLISHED-Artikel existiert (*setUp()* in Listing 4).

Der Test *testIndexSorting()* prüft die Sortierung nach den jüngsten Artikeln. Hierzu werden zunächst die Testdaten modifiziert, um sicherzustellen, dass eine nach *dateCreated* stark unsortierte Liste vorliegt. Der Aufruf von *flush()* garantiert, dass die Änderungen unmittelbar in die Datenbank geschrieben werden und somit für nachfolgende DB-Operationen vorhanden sind. Als Nächstes werden in gewohnter Manier die *Controller*-Action aufgerufen und die Werte im Model geprüft. Ein Assertion-Error wird geschmissen, wenn ein höherer Eintrag in der Liste ein jüngeres Datum haben sollte. Es folgt ein zweiter Aufruf mit einem Offset von 10, um die zweite Seite der Artikel zu testen.

Die Methode *testIndexPublished()* setzt alle Artikel auf einen UNPUBLISHED-Status und verifiziert, dass kein Artikel beim Aufruf der Action zurückgegeben wird. Listing 4 zeigt den kompletten Sourcecode.

Funktionstests

In Sachen Testing fehlt uns nun noch der Funktionstest, der auch als Black-box Test oder End-to-end Test bezeichnet wird. Anders als bei obigen Testvarianten bietet Grails hierfür keine unmittelbare Unterstützung an. Stattdessen haben wir die Qual der Wahl und können eines von aktuell drei vorhandenen Plug-ins für die Funktionstests nutzen.

- *Selenium* – Plus: Nutzt direkt die Browser-Engine. Minus: Schwergewichtig und aufwändigere Build-Integration [5].
- *Webtest* – Plus: Ausgreift und das erste Plug-in in diesem Bereich für Grails. Minus: Etwas aufwändiger in der Nutzung [6].
- *Functional Test* – Plus: Einfach zu verwenden und analog der anderen Tests einzubinden. Minus: In Sachen Features noch nicht voll ausgereift [7].

Unsere Entscheidung fällt auf das *Functional-Test-Plug-in*, da es gute Chancen hat, in einer zukünftigen Grails-Version als Core-Plug-in mit ausgeliefert zu werden. Das Ziel dieser Übung ist es sicherzustellen, dass das Paging in der Blogansicht funktioniert und dass der Backoffice-Bereich nur nach einem Login zugreifbar ist. Los geht es mit der Installation des Plug-ins (wie immer aus dem *blogapp*-Verzeichnis heraus):

Listing 4: Integrationstest: HomeControllerIntegrationTests

```
class HomeControllerTests extends ControllerUnitTestCase {

  def controller
  def sessionFactory

  protected void setUp() {
    super.setUp()
    // check pre-condition  (bootstrap-data)
    assertTrue BlogArticle.count()>20
    def countPublished = BlogArticle.countByStatus ( ArticleStatus.PUBLISHED)
    assertTrue countPublished > 10
    assertTrue countPublished < BlogArticle.count()
    // init object under test (OUT)
    controller = new HomeController()

  }

  void testIndexSorting() {
    // prepare test-data: shuffle the creationDate
    def dayDelta = 2.0
    BlogArticle.list().each {
      dayDelta = -dayDelta * 1.2
      it.dateCreated = new Date() + (int)dayDelta
      it.save()
    }
    sessionFactory.currentSession.flush()
    // call first page
    Map model = controller.index()
    List articles = model.blogArticles
    assertEquals 10, articles?.size()
    (1..articles.size()-1).each {
      assertTrue articles[it-1].dateCreated >= articles[it].dateCreated
    }
    // call second page
    mockParams.offset = 10
    model = controller.index()
    List nextArticles = model.blogArticles
    assertTrue nextArticles[0].dateCreated <= articles[-1].dateCreated
    (1..nextArticles.size()-1).each {
      assertTrue nextArticles[it-1].dateCreated >= nextArticles[it].dateCreated
    }
  }

  void testIndexPublished() {
    // prepare test-data: set all entries to unpublished
    BlogArticle.list().eachWithIndex { a, i ->
      a.status = i%2 ? ArticleStatus.DISABLED : ArticleStatus.UNPUBLISHED
      a.save()
    }
    sessionFactory.currentSession.flush()
    // check result
    Map model = controller.index()
    List articles = model.blogArticles
    assertEquals 0, model.blogArticles.size()
    assertEquals 0, model.blogCount
  }
}
```

Listing 5: Funktionstest: BlogFunctionalTests

```
class BlogFunctionalTests extends functionaltestPlug-in.FunctionalTestCase {

  void testPaging() {
    // check the first page
    get('/')
    assertStatus 200
    assertContentContains '<h1>MyBlog</h1>'
    def nodesFirst = page.getByXPath("//div[@class='post']")
    assertEquals 10, nodesFirst?.size()
    // check the second page
    click ('Naechste')
    assertStatus 200
    def nodesSecond = page.getByXPath("//div[@class='post']")
    assertTrue nodesSecond?.size() > 3
    assertContentContains 'class="prevLink"'
    // articles from first page shouldn't be on the second page
    def titleSecond = nodesSecond*.firstChild*.asText()
    for (node in nodesFirst) {
      def title = node.firstChild.asText()
      assertTrue ("${title} shouldn't be in ${titleSecond}",
          !titleSecond.contains(title))
    }
  }

  void testBackofficeLogin() {
    get ('/backoffice')
    assertContentContains '<form action="/blogapp/auth/signIn"'
    form ('login') {
      username = 'admin'
      password = 'geheim'
      click 'submit'
    }
    assertContentContains ,Backoffice von <a href="/blogapp/">MyBlog</a>'
  }

}
```

Listing 6: Logging in Config.groovy

```
log4j = {
  error 'org.codehaus.groovy.grails'      // show grails errors
  warn  'org.mortbay.log'             // show jetty warning
  debug 'grails.app'
     'de.javamagazin.myblog'
}

loggingProduction = {
  def myAppender = new org.apache.log4j.DailyRollingFileAppender(
          name:'file',
          file:'/log/myblog.log',
          datePattern:"'.'yyyy-MM-dd",
          layout: pattern(conversionPattern: ,[%d{HH:mm:ss.SSS}]
                                                          [%5p] [%t] [%c] %m%n')
        )
  appenders {
    appender myAppender
  }
  root {
    error 'file'
  }
  info 'grails.app'
     'de.javamagazin.myblog'
}

environments {
  development {
    // default logging
  }
  test {
    // default logging
  }
  production {
    log4j = loggingProduction
  }
}
```

```
grails install-Plug-in functional-test
```

Nach der erfolgreichen Installation steht uns ein neues Kommando zur Verfügung, um eine neue Datei für den Funktionstest anzulegen: *grails create-functional-test*-Blog. Unter *test\functional* befindet sich nun eine neue Testklasse, die wir, wie bereits bei den vorangegangenen Tests gesehen haben, mit Testmethoden ausfüllen können. Das Besondere: Die Testklasse leitet von *functionaltestPlug-in.FunctionalTestCase* ab und bietet dadurch eine DSL und erweitere Assertions, um bequem die Web-Requests zu deklarieren und die Ergebnisse zu prüfen. Listing 4 zeigt das Ergebnis. Mit der Methode *testPaging()* prüfen wir, ob das grundsätzliche Paging über mehrere Seiten funktioniert. Hierzu rufen wir mittels *get* ('/') die Startseite auf und prüfen, ob der http-Request einen korrekten Statuscode inklusive erwartetem Text und zehn Div-Elementen mit den Blogeinträgen zurückliefert. Interessant ist die Verwendung der *page*-Variablen. Das Functional-Test-Plug-in nutzt unter der Haube das Framework HtmlUnit [5]. Die *page*-Variable hält die vollständig zurückgelieferte Antwort des Servers und kann über das API von HtmlUnit bequem ausgelesen und verarbeitet werden. In unserem Fall werden über einen XPath-Ausdruck alle Knoten mit Blogeinträgen herausgefiltert.

Mit dem Befehl *click* (*Naechste*) betätigen wir den entsprechenden Link auf der Seite und gelangen, wenn alles gut geht, auf die nächste Seite. Dieses prüfen wir wieder mittels des Statuscodes und anhand der Existenz von mindestens vier Blogeinträgen und einem *Previous*-Link auf der Seite. Abschließend vergleichen wir die Überschriften der Artikel von Seite 1 mit der aktuellen Seite und verifizieren, dass die Mengen disjunkt sind.

Unser zweiter Test *testBackofficeLogin()* ruft direkt den Admin-Bereich unseres Blogs auf. Wir prüfen zunächst, dass anstatt der Backoffice-GUI die Login-Page erscheint. Alles andere wäre ein Sicherheitsloch in unserer Applikation. Im nächsten Schritt füllen wir die Felder im Formular mit dem Namen *login* aus und schicken es ab. Die letzte Assertion prüft die Überschrift unserer Backoffice-Startseite.

Das war es auch schon. Den Rest erledigen Grails und das Plug-in für uns, indem der Funktionstest wie folgt gestartet wird: *grails functional-tests*. Ein frisches War-File wird gebaut, der Jetty-Applikationsserver wird gestartet und unser Test setzt die entsprechenden Web-Requests ab. Sollte sich ein Fehler eingeschlichen haben, wird dies angezeigt und ausführlich im oben kennengelernten JUnit-Html-Fehlerreport vermerkt. Ist alles gut gegangen, sehen wir auf der Konsole:

```
-------------------------------------------------------
Running 2 Functional Tests...
Running functional test BlogFunctionalTests...
  testPaging... ## Passed!
  testBackofficeLogin... #### Passed!
Functional Tests Completed in 6672ms ...
-------------------------------------------------------
```

Konfiguration/Logging

Auch wenn Grails stark auf Konventionen setzt, können im Bedarfsfall eine Vielzahl von Einstellungen konfiguriert werden. Schließlich heißt es ja „Conventions over Configuration" und nicht „Conventions instead of Configuration". Grails kennt standardmäßig drei Dateien für die Konfiguration:

- *BuildConfig.groovy*: Parameter, die den Build-Prozess und die Entwicklungsumgebung betreffen.
- *Config.groovy*: Allgemeine Konfigurationsparameter, die zur Laufzeit auf die Applikation wirken.
- *DataSource.groovy*: Konfigurationsparameter für die Anbindung von Datenbanken (Kasten: „In-Memory Datenbank als Datei", Teil 1 [1]).

An den Endungen ist bereits zu erkennen, dass es sich nicht um normale Java-Property-Dateien handelt, sondern um Groovy-Code. Die Dateien werden von Groovys *ConfigSlurper* eingelesen, der im Kern die gleichen Funktionen wie Javas Properties-Klasse abbildet. Der nicht zu unterschätzende Vorteil ist jedoch, dass beliebiger Groovy-Code in der Konfiguration verwendet werden kann. Dadurch können beispielsweise, abhängig von verschiedenen Umgebungen, verschiedene Werte gesetzt werden. Oder es können DSLs verwendet werden, um spezielle Konfigurationsaspekte übersichtlicher zu gestalten.

Ein Beispiel für einen Eintrag in der *BuildConfig.groovy*-Datei wäre z. B. *grails.project.work.dir = „work"*. Hierdurch wird das Worker-Verzeichnis von *$User_Home/.grails/1.1/projects* nach *work/* innerhalb der Projektstruktur verschoben. Dieses ist insbesondere dann zu empfehlen, wenn mehrere Projekte mit dem gleichen Namen auf einem Rechner existieren, da Grails sonst mit der Verwaltung durcheinander kommt (Achtung! Verzeichnis auf *ignore* innerhalb der Versionsverwaltung setzen.).

Um die Features der Grails-Konfiguration kennen zu lernen, werden wir nun exemplarisch die Logging-Einstellungen anpassen. Grails nutzt das verbreitete *Log4J* zur Protokollierung von Applikationsnachrichten. Standardmäßig ist ein Root-Logger mit einem Log-Level *error* für Stdout definiert. Unser Ziel ist es, für die Entwicklungsumgebung unsere Klassen auf den Log-Level *debug* zu setzen. In der Produktionsumgebung jedoch soll die Ausgabe in eine täglich rotierende Log-Datei mit dem Log-Level *info* geschehen. Wir nutzen hierzu die Grails eigene DSL zur Konfiguration von Log4J. Listing 5 zeigt den entsprechend angepassten Ausschnitt aus der *Config.groovy*.

Das Ganze mag auf den ersten Blick ein wenig verwirrend aussehen, zeigt aber bei genauerer Betrachtung durchaus seine Eleganz. Über die Variable *log4J*, der eine Closure mit DSL-Befehlen zugewiesen wird, wird die Konfiguration vorgenommen. Dieses nutzen wir, um per Default u. a. unsere Grails-Artefakte (*grails.app*) und alle Klassen in unserem Package (*de.javamagazin.myblog*) auf den Debug-Level zu setzen. Die *loggingProduction* zugewiesene Closure definiert die *Log4J*-Einstellung für einen *DailyRollingFileAppender*, wobei unsere Klassen und Artefakte auf den Log-Level *Info* gesetzt werden. Zuletzt kann die *environments*-Methode verwendet werden, um das Logging für Production zu überschreiben. Probieren Sie es aus, indem Sie z. B. Grails im Produktionsmodus starten (*grails prod run-app*) oder

das WAR-File bauen (*grails war*) und deployen.

Builds/Continuous-Integration/Deployment

In der Version 1.1 unterstützt Grails eine Vielzahl von Build-Systemen, sodass jedes Team seine gewohnte Java-Build-Umgebung beibehalten kann. Ant inklusive Ivy für das Abhängigkeitsmanagement wird genauso unterstützt wie Maven 2. Intern setzt Grails auf das Build-System Gant [9]. Es nutzt und erweitert die klassischen Ant-Tasks, wobei die Konfiguration nicht über XML, sondern über Groovy geschieht. Das hat den Vorteil, genauso wie beim Thema Konfiguration oben gesehen, dass beliebige Logik in die Ant-Skripte eingebaut werden kann. Dieses ist sehr leistungsfähig und erzeugt les- und wartbare Build-Skripte. Auch für das Bauen eigener Grails-Skripte ist Gant die erste Wahl. Das sind wiederkehrende Aufgaben, die der Grails-Entwickler in das Verzeichnis *blogapp/scripts* ablegen kann. Diese Skripte sind dann via *grails skript-name* direkt aufrufbar oder können über eine Namenskonvention auch automatisch bei bestimmten Events aufgerufen werden. Zum Beispiel könnte ein Skript am Event „War-File bauen" lauschen und sämtliche JavaScript- und CSS-Dateien komprimieren, bevor sie „verpackt" werden. Nicht zu vergessen: Die Unterstützung von Gradle [10], dem neuen, vielversprechenden Build-System und potenziellen Maven-Nachfolger, ist mit Grails 1.1 ebenfalls verbessert worden.

Bezüglich Continuous Integration und Deployment gibt es glücklicherweise nicht viel Überraschendes zu berichten. Eine gebaute Anwendung steckt in einem War-File und ist eine ganz normale JEE-Anwendung. Für sie gelten die gleichen Regeln und Vorzüge, wie für jede andere Webapplikation auf Java-Basis. Sprich: Egal ob Tomcat, WebSphere oder Oracle, die Installation kann auf jedem standardkonformen Applikationsserver durchgeführt werden. Einzig der Speicherverbrauch könnte durch den Einsatz von Groovy ein wenig höher sein, sodass Sie genügend Memory zuweisen sollten. Nutzen Sie hiefür die entsprechenden JVM-Parameter:

```
-XX:MaxPermSize=256m
-Xmx512M
```

Dadurch, dass eine Grails-Applikation auch mit einer klassischen *build.xml*-Datei (Ant) ausgeliefert wird, kann sie mit jedem beliebigen Continuous-Integration-Server getestet und gebaut werden. Besonders zu empfehlen ist hier der Einsatz von Hudson [11], da es zum einen ein sehr gutes System ist und zum anderen über ein Plug-in direkte Unterstützung von Grails anbietet. Somit kann auf ein Build-Skript sogar komplett verzichtet werden.

Abschluss

An dieser Stelle beenden wir unsere dreiteilige Reise durch die Welt der neuen Programmierplattform Grails. Wir haben gesehen, wie schnell und einfach sich eine Webapplikation auf dieser Basis bauen lässt. Auch die professionellen Ansprüche in Bezug auf Robustheit, Wartbarkeit und Skalierbarkeit werden nicht zuletzt aufgrund der zugrunde liegenden Java-Plattform zufrieden gestellt. Durch den Mix der soliden Java-Basis und Übernahme von Konzepten aus innovativen Frameworks wie Ruby on Rails ist mit Grails ein sehr leistungsfähiges Tool entstanden. Dieses sollte sich jeder Architekt/Programmierer im Webumfeld unbedingt näher anschauen. Insbesondere bei komplett neu zu bauenden Webapplikationen ist Grails im Java-Umfeld sicherlich zur ersten Wahl aufgestiegen. Ein paar Kinderkrankheiten, die es hier und da noch gibt, können durch Workarounds und eine sehr gute Unterstützung durch die Community abgefangen werden. Dadurch, dass Groovy/Grails Ende letzten Jahres offiziell unter die Fittiche von SpringSource genommen wurde, ist auch langfristig ein kommerzieller Support gesichert. Sollten noch Zweifel bezüglich der Einsatzfähigkeit bestehen, lohnt sich ein Blick auf die ständig wachsenden Referenzen [12].

Dem Leser, der tiefer in die Grails-Programmierung einsteigen möchte, sei zuerst das Buch „The Definitive Guide To Grails" empfohlen (siehe Rezension S. 11). Auch sehr nützlich, insbesondere zum Nachschlagen, ist der gute Online-User-Guide von Grails [13]. Wertvolle Informationen und die Beantwortung von spezifischen Fragen erhält der Grails-Entwickler auf der stetig wachsenden User-Mailing-Liste [14]. ■

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 & 2), Java Magazin 4.09 und 05.09
[2] Plug-in: Code Coverage: http://www.grails.org/Test+Code+Coverage+Plugin
[3] Cobertura: http://cobertura.sourceforge.net/
[4] http://jira.codehaus.org/browse/GRAILS-4271
[5] http://www.grails.org/plugin/selenium
[6] http://www.grails.org/plugin/webtest
[7] http://www.grails.org/plugin/functional-test
[8] HtmlUnit: http://htmlunit.sourceforge.net/
[9] Gant: http://gant.codehaus.org/
[10] Gradle: http://www.gradle.org/
[11] Hudson: https://hudson.dev.java.net/
[12] http://www.grails.org/Success+Stories
[13] Grails Reference Documentation: http://grails.org/doc/1.1.x/
[14] Grails-Mailing-Listen: http://www.grails.org/Mailing+lists