Groovy, Grails und eine Twitter-Anwendung

Grooviges Gezwitscher

Grails ist ein Open-Source-Framework zur Entwicklung moderner Webapplikationen auf Basis der dynamisch typisierten Programmiersprache Groovy und bewährten Technologien wie dem Spring Framework, Hibernate und SiteMesh. Eine Vielzahl von Plug-ins macht es zudem möglich, wiederkehrende Problemstellungen mit bewährten Lösungen umzusetzen.

von Christian Metzler und Stefan Glase

Basierend auf der dynamischen Programmiersprache Groovy lassen sich mit Grails Anwendungen entwickeln, die von einer extrem ausdrucksstarken und auch für Java-Entwickler nach kurzer Einarbeitungszeit beherrschbaren Sprache profitieren. Durch die nahtlose Integration von Groovy in das Java-Ökosystem kann man auf bestehende Klassen und Bibliotheken zurückgreifen. Das macht den Umstieg besonders dann interessant, wenn man Investitionen in die Java-Plattform nicht so einfach über Bord werfen kann oder möchte. In diesem Artikel zeigen wir anhand einer kleinen Applikation die Vorzüge von Groovy und Grails. Als Beispiel haben wir uns das Web-2.0-Phänomen Twitter zum Vorbild genommen und werden nun den Grundstein legen, um in unserer Anwendung ein wenig zu „zwitschern".

Entwicklungsumgebung

Grundvoraussetzung für die Entwicklung mit Grails ist ein installiertes Java JDK 1.6 oder höher, das über die Variable *JAVA_HOME* bekannt gemacht wurde und den *PATH* um den eigenen *bin*-Ordner erweitert. Nach dem Herunterladen und Entpacken der aktuellen Grails-Distribution (derzeit 2.0.x) von der offiziellen Grails-Webseite www.grails.org muss die Umgebungsvariable *GRAILS_HOME* gesetzt und auch hier der *PATH* um den *bin*-Ordner erweitert werden. Im Anschluss kann auf einer Konsole mit dem Befehl *grails –version* die korrekte Installation überprüft werden. Als Entwicklungsumgebung stehen je nach eigener Präferenz diverse Werkzeuge zur Verfügung, die bei der Entwicklung einer Grails-Anwendung unterstützen. Die beiden wohl am weitesten verbreiteten Entwicklungsumgebungen für Grails sind die kostenlos verfügbare SpringSource Tool Suite und das kommerzielle IntelliJ IDEA in der Ultimate Edition.

Applikation erstellen

Grails bietet für die Erzeugung einer neuen Applikation und aller dafür erforderlichen Artefakte einen einfachen Konsolenbefehl an. Mit dem Kommando *grails create-app gezwitscher* erzeugen wir eine neue Applikation mit dem Namen „gezwitscher" in dem Verzeichnis, in dem wir das Kommando aufgerufen haben. Inspiriert von Ruby on Rails, legt auch Grails viel Wert auf die Berücksichtigung der Paradigmen „Convention over Configuration" und „Don't repeat yourself". So gibt Grails bereits in der Verzeichnisstruktur einen einheitlichen Rahmen vor, der es Entwicklern einfach macht, sich in andere Grails-Projekte einzuarbeiten.

Eine Grails-Applikation besteht aus so genannten *Grails-Artefakten*, die je nach Typ in einem eigens dafür vorgesehenen Pfad abgelegt werden. Grails unterscheidet hierbei zwischen

- den Fach- oder Domänenklassen (*grails-app/domain*)
- der Serviceschicht (*grails-app/services*)
- der Controller-Schicht (*grails-app/controller*)
- den Message Bundles (*grails-app/i18n*)
- den Views (*grails-app/views*)
- den Tag-Bibliotheken (*grails-app/taglib*)
- den Utility-Klassen (*grails-app/utils*) und
- den Konfigurationsdateien (*grails-app/conf*)

Listing 1: Modellierung der User-Klasse

```
package gezwitscher

class User {

    String username
    String password
    String email

    static hasMany = [followers: Follower, statuses: Status]

    static constraints = {
        username(blank: false, unique: true)
        password(blank: false, password: true)
        email(blank: false, unique: true, email: true)
    }

}
```

Sehr starkes Augenmerk legt Grails auch auf das automatisierte Testen der Anwendung, und so werden von Haus aus bereits Integrationstests (*test/integration*) und Unit Tests (*test/unit*) unterstützt. Auch funktionale Tests sind über Erweiterungen einfach zu automatisieren.

An dieser Stelle haben wir bereits eine lauffähige Grails-Anwendung vorliegen, die uns mit diversen Erleichterungen bei der weiteren Entwicklung unterstützt. So können beispielsweise Abhängigkeiten zu anderen Bibliotheken über Maven Repositories aufgelöst, das bereits vorkonfigurierte Logging angepasst, das Deployment der Anwendung in einem Apache Tomcat durchgeführt oder die Internationalisierung von Anwendungstexten über bereitgestellte Message Bundles vorgenommen werden. Der aktuelle Stand der Anwendung kann nun mit dem Kommando *grails run-app* im Browser betrachtet werden.

Domänenobjekte modellieren

Für unsere Applikation benötigen wir vorerst drei Domänenobjekte – die Klasse *User*, die Daten über einen Benutzer enthält, die Klasse *Status*, die eine Statusnachricht eines Benutzers darstellt, sowie die Klasse *Follower*, die die Beziehung zwischen verschiedenen Benutzern herstellt.

Zuerst legen wir die Klasse *User* an. Das erfolgt über die Konsole mittels des Kommandos *grails create-domain-class User*. In der von Grails erzeugten Klasse können nun die fachlichen Eigenschaften modelliert werden. Wenn die Sichtbarkeit nicht explizit angegeben wird, werden automatisch Getter und Setter für die entsprechenden Attribute und Beziehungen angelegt (Listing 1).

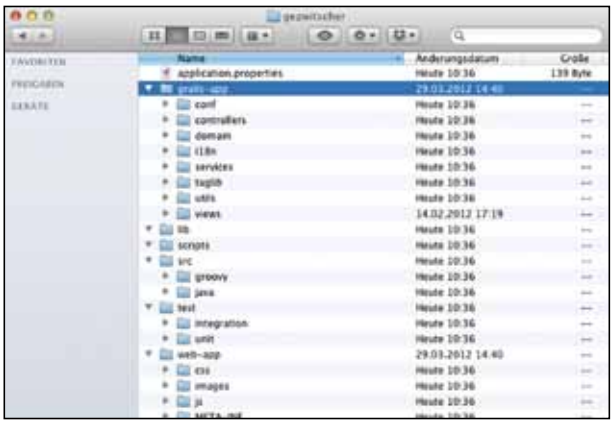

Abb. 1: Verzeichnisstruktur eines Projekts

Grails nutzt für die Abbildung von Domänenklassen auf relationale Datenbanken das bekannte Framework Hibernate. Dabei erweitert Grails das objektrelationale Mapping von Hibernate mit eigenen Features (Stichwort GORM – Grails' Object Relational Mapping).

Eine domänenspezifische Sprache (DSL) ermöglicht es zum Beispiel, die Beziehungen von verschiedenen Do-

mänenklassen zu definieren. So kann man in unserem Beispiel sehen, dass ein *User* mehrere *Follower* und *Status* besitzen kann. Hier wird durch das Schlüsselwort *hasMany* eine 1-zu-N-Beziehung definiert. Außerdem können mittels dieser DSL auch Validierungsregeln (Constraints) für die Klasse erstellt werden. Während das Schlüsselwort *unique: true* bewirkt, dass im Datenbankschema ein entsprechendes Unique Constraint erstellt wird, hat beispielsweise *email: true* zur Folge, dass eine applikationsseitige Validierung auf eine syntaktisch gültige E-Mail-Adresse durchgeführt wird.

Wenn gewünscht, können aber auch einfach eigene Validatoren implementiert werden, um beispielweise Abhängigkeiten zwischen Attributen einer Klasse zu definieren. In der gleichen Weise erzeugen wir die Klassen für den *Status* und den *Follower* (Listings 2 und 3).

Eine von Grails vorgegebene Konvention möchten wir uns in der Klasse *Status* zunutze machen: Wenn in einer Klasse ein Attribut *dateCreated* angelegt wird, wird es automatisch von Grails mit dem aktuellen Datum bei der Anlage gefüllt.

Listing 2: Modellierung der Status-Klasse

```
package gezwitscher

class Status {

    String message
    Date dateCreated

    static belongsTo = [user: User]

    static constraints = {
        message(blank: false, maxSize: 140)
        dateCreated()
        user()
    }

}
```

Listing 3: Modellierung der Follower-Klasse

```
package gezwitscher

class Follower {

    User follows

    static belongsTo = [user: User]

    static constraints = {
        user()
        follows()
    }

}
```

Das Mapping zur Datenbank kann ebenfalls über die DSL in der Klasse beeinflusst werden. Dazu gehören unter anderem das Benennen des Tabellennamens und einzelner Spalten oder das Erzeugen von Indizes. An dieser Stelle wollen wir allerdings nicht näher darauf eingehen und stattdessen auf die sehr umfangreiche Dokumentation verweisen.

Prototyping der Anwendung

Um das Domänenmodell nun erstmals zu testen, bietet Grails die Möglichkeit des *Scaffolding* an. Dazu muss für jede Domänenklasse ein Controller erstellt werden, der generisch alle CRUD-Operationen (create, read, update und delete) zur Verfügung stellt. Solch ein Controller kann über das Kommando *grails create-scaffold-controller gezwitscher.Follower* angelegt werden (Listing 4). Die erforderlichen Masken werden zur Laufzeit generiert und helfen, das zuvor erstellte Datenmodell auf seine Korrektheit und Vollständigkeit hin zu überprüfen.

Nach Erstellung der Controller kann nun erneut die Applikation mittels *grails run-app* gestartet werden. Insbesondere in Zeiten agiler Softwareentwicklung ist es sehr wertvoll, in kürzester Zeit einen Prototyp zu erstellen, es weiterzuentwickeln und an neue Anforderungen anzupassen.

Das Gezwitscher kann beginnen …

Wir wollen angemeldeten Benutzern unserer Applikation jetzt die Möglichkeit geben, Statusnachrichten zu verfassen und zu speichern. Dazu implementieren wir

Listing 4: Mittels Scaffolding angelegter Controller

```
package gezwitscher

class FollowerController {
    static scaffold = true
}
```

Listing 5: Abgeleitete Klasse User

```
package gezwitscher

import gezwitscher.security.SecUser

class User extends SecUser {

    String email

    static hasMany = [followers: Follower, statuses: Status]

    static constraints = {
        email(blank: false, unique: true, email: true)
    }

}
```

entwickler.press

Tobias Bosch, Stefan Scheidt, Torsten Winterberg

Mobile Web-Apps mit JavaScript

Leitfaden für die professionelle Entwicklung

218 Seiten, August 2012

PRINT	ISBN: 978-3-86802-085-4	34,90 € / 35,90 € (A)
PDF	ISBN: 978-3-86802-273-5	21,99 €
EPUB	ISBN: 978-3-86802-610-8	21,99 €

Es herrscht ein regelrechter Hype um mobile Lösungen, erst recht seitdem Android-basierte Smartphones massenhaft auf den Markt drängen. Einen ähnlichen Aufschwung erlebt JavaScript, denn mit der enormen Ausbreitungsgeschwindigkeit von HTML5 wird diese Sprache immer interessanter.
Dieses Buch liefert eine durchgehende Anleitung, wie man professionell auf hohem Niveau Webanwendungen für mobile Endgeräte schreibt. Der Leser erhält so einen hervorragenden Einstieg in die Programmierung mobiler Web-Apps, in die testgetriebene Entwicklung für JavaScript und das Schreiben von Clean Code.

Mehr Infos zum Buch finden Sie unter: www.entwickler-press.de/mobile_web_apps

einen Authentifizierungsmechanismus und die benötigte Geschäfts- und Darstellungslogik.

Absicherung der Anwendung

Zur Absicherung der Anwendung wollen wir uns die benötigte Funktionalität über den von Grails bereitgestellten Plug-in-Mechanismus in die Applikation holen. Mittlerweile existieren weit über 700 Plug-ins, die die modulbasierte Grails-Plattform ergänzen oder erweitern. Sie können über das Kommando *grails install-plugin <Name des Plug-In>* der Applikation hinzugefügt werden. Die benötigten Ressourcen werden automatisch über die verfügbaren Repositories bezogen.

In unserem Fall wählen wir das Plug-in *spring-security-core*. Es enthält genau die Funktionalität, die wir benötigen. Plug-ins haben die Möglichkeit, dem Framework neue Skripte hinzuzufügen, die dann als Grails-Kommando ausgeführt werden können. Das Security-Plug-in fügt zum Beispiel ein neues Kommando *s2-quickstart* hinzu, mit dessen Hilfe die Applikation sehr schnell abgesichert werden kann. Hierzu rufen wir das Kommando mit drei Parametern auf, die das Package und die Namen für die Klassen *Benutzer* und *Rolle* angeben: *grails s2quickstart gezwitscher.security SecUser SecRole*. Mit diesem Befehl werden die Klassen *SecUser* und *SecRole* angelegt. Damit wir unseren bereits angelegten *User* weiterbenutzen können, leiten wir ihn einfach von *SecUser ab* und entfernen die doppelten Attribute.

Nun sind wir in der Lage, den *StatusController* mithilfe einer einzigen Annotation abzusichern (Listing 6).

Diese Annotation bewirkt, dass ausschließlich angemeldete Benutzer, die die Rolle *ROLE_USER* besitzen, die Aktionen dieses Controllers nutzen können. Der Loginmechanismus, die dazugehörigen Views sowie das komplette Session-Handling werden vom Plug-in bereitgestellt. Damit wir die Absicherung testen können, muss den Benutzern nun die entsprechende Rolle zugewiesen werden.

Bereitstellung von Testdaten

Damit man für das Testen nicht jedes Mal mehrere Benutzer mit der entsprechenden Rolle anlegen muss, bietet Grails die Option, dies automatisch beim Start der Applikation zu tun. Hier kann während des Starts der Anwendung beliebiger Sourcecode ausgeführt werden (Listing 7).

Listing 6: Abgesicherter StatusController

```
package gezwitscher

import grails.plugins.springsecurity.Secured

@Secured(['ROLE_USER'])
class StatusController {
    static scaffold = true
}
```

Dieser Vorgang kann in Grails für unterschiedliche Umgebungen auch differenziert implementiert werden. Grails bietet hier standardmäßig drei vordefinierte Profile an: *development*, *test* und *production*.

Controller und Views für Dateneingabe

Ein Controller definiert über seine sichtbaren Methoden im Zusammenspiel mit der Konfigurationsdatei *UrlMappings.groovy* die von der Anwendung angebotenen URLs. Wir hätten gerne unter dem Pfad */status/create* eine Maske zur Erfassung einer neuen Statusnachricht und implementieren dazu entsprechend der Grails-Konventionen im *StatusController.groovy* die Methode *create*, die im Model gemäß MVC ein leeres Statusobjekt zur weiteren Bearbeitung bereitstellt (Listings 8 und 9).

Listing 7: Bootstrapping der Applikation

```
import gezwitscher.User
import gezwitscher.security.SecRole
import gezwitscher.security.SecUserSecRole
import gezwitscher.Status

class BootStrap {

    def init = { servletContext ->
        // Rolle anlegen
  def user = new SecRole(authority: 'ROLE_USER')
      .save(failOnError: true)

        // Testnutzer anlegen
        def fred = new User(
            username: 'fred', password: 'geheim',
            email: 'fred@feuerstein.de', enabled: true)
            .save(failOnError: true)
        new SecUserSecRole(secUser: fred, secRole: user)
            .save(failOnError: true)

        def wilma = new User(
            username: 'wilma', password: 'geheim',
            email: 'wilma@feuerstein.de', enabled: true)
            .save(failOnError: true)
        new SecUserSecRole(secUser: wilma, secRole: user)
            .save(failOnError: true)

        // Gezwitscher hinzufügen
        def messages = [
            "Wow, ist das einfach!",
            "Super, es funktioniert!",
            "Grails ist toll!"]
        messages.each {
            new Status(message: it, user: wilma)
                .save(failOnError: true)
        }
    }

    def destroy = {}

}
```

Damit mit dem Absenden eines Formulars aus der Maske heraus der neue Status in der Datenbank gespeichert werden kann, implementieren wir im gleichen Controller die Methode *save*. Die reichert das Statusobjekt um den aktuellen Benutzer an und versucht dann, das Statusobjekt zu speichern. Schlägt die Speicherung fehl, wird erneut die View *create* angezeigt, und Fehlermeldungen (beispielsweise Validierungsfehler) können dargestellt werden. Ist die Speicherung erfolgreich, legen wir eine Nachricht in den *flash*-Scope der Anwendung und leiten weiter an die Methode *index* des *TimelineController.groovy*.

Implementierung der Datenausgabe

Für die Anzeige der Timeline gilt es, alle Statusnachrichten zu identifizieren, die von Usern verfasst wurden, die den angemeldeten Benutzer zu ihren Followern zählen. An dieser Stelle können wir mithilfe von GORM die Suche nach diesen Objekten mit sehr wenig Code ausdrücken. Dazu haben wir mit *grails create-service gezwitscher.Timeline* einen Service erstellt (Listing 10).

Mittels der durch das Spring Framework in die Applikation gebrachten Dependency Injection können wir uns im *TimelineService* den *UserService* geben lassen und auf diese Weise unkompliziert an den aktuell angemeldeten Benutzer gelangen. Im Anschluss erstellen wir mittels *grails create-taglib gezwitscher* noch eine Tag-

Listing 8

```
package gezwitscher

import grails.plugins.springsecurity.Secured

@Secured(['ROLE_USER'])
class StatusController {

    def userService

    def create() {
        [status: new Status()]
    }

    def save(Status status) {
        status.user = userService.currentUser()

        if (!status.save()) {
            render view: "create", model: [status: status]
        } else {
            flash.message = "Status successfully posted!"
            redirect controller: "timeline", action: "index"
        }
    }

}
```

Bibliothek zur Ausgabe von Statusnachrichten, da wir erkannt haben, dass eine solche Ausgabe uns nicht nur auf einer einzelnen View begegnet (Listing 11).

An jeder Stelle, an der wir fortan ein Statusobjekt zur Anzeige bringen wollen, kann das nun mit dem Tag *<gezwitscher:displayStatus status="${status}"/>* erledigt werden, und auch eine Änderung in der Darstellung unserer Statusnachrichten muss nur noch an einer Stelle im Code vorgenommen werden.

Listing 9

```
<html>
<head>
  <title>Create Status</title>
  <meta name="layout" content="main"/>
</head>

<body>

<div class="content">
<g:form action="save">
  <g:hasErrors>
    <g:renderErrors bean="${status}" as="list"/>
  </g:hasErrors>

  <fieldset class="form">
    <div class="fieldcontain">
      <label for="message">New Status:</label>
      <g:textArea id="message" name="message"
                          value="${status.message}" rows="3" cols="60"/>
    </div>
  </fieldset>

  <fieldset class="buttons">
    <g:submitButton name="submit" value="Post Status"/>
  </fieldset>
</g:form>
</div>

</body>
</html>
```

Listing 10

```
package gezwitscher

class TimelineService {

    def userService

    def timeline() {
        def query = Status.where {
            user in followedUsers() + userService.currentUser()
        }
        query.list(sort: 'dateCreated', order: 'desc')
    }

    private def followedUsers() {
        Follower.findAllByUser(userService.currentUser()).follows
    }

}
```

Fazit

Die vielversprechende Kombination der dynamischen Programmiersprache Groovy und der Webanwendungsplattform Grails, die auf etablierten und erprobten Open-Source-Frameworks basiert, zeigt nicht nur in kleinen Beispielanwendungen ihre Stärken: Sie hat sich für uns auch bereits in großen Enterprise-Anwendungen bewährt. Durch die ausdrucksstarke Sprache entsteht gut lesbarer und in der Folge wartbarer Quellcode. Die Vielzahl stabiler und funktionsreicher Plug-ins bedingt es, dass so mancher Frameworkcode nicht mehr selbst geschrieben werden muss. Und dank der strukturellen Konventionen innerhalb von Grails-Anwendungen fällt Entwicklern die Einarbeitung in neue Projekte um ein Vielfaches leichter. All diese Faktoren helfen dabei, die Fachlichkeit einer Anwendung wieder mehr in den Vordergrund zu rücken. Unsere Kunden werden es uns danken, wenn Anforderungen in kürzerer Zeit umgesetzt und an veränderte Bedingungen angepasst werden können.

Stefan Glase (stefan.glase@opitz-consulting.com) ist Senior Consultant im Bereich Application Engineering bei der OPITZ CONSULTING Gummersbach GmbH. Er beschäftigt sich seit mehreren Jahren mit der Architektur und Implementierung von Enterprise-Applikationen. Seine Schwerpunkte sind Enterprise Java mit dem Spring Framework sowie Groovy & Grails. Er ist Autor von Fachartikeln und spricht regelmäßig auf Fachkonferenzen.

Christian Metzler (christian.metzler@opitz-consulting.com) ist Consultant im Bereich Application Engineering bei der OPITZ CONSULTING Gummersbach GmbH. Er beschäftigt sich seit einigen Jahren insbesondere mit der Entwicklung von Webanwendungen. Seine Schwerpunkte liegen insbesondere auf der Sprache Groovy und dem leichtgewichtigen Framework Grails.

Listing 11

```
package gezwitscher

class GezwitscherTagLib {

    static namespace = "gezwitscher"

    def displayStatus = { params, body ->
        Status status = params.remove('status')

        out << "<div class='status'>"
        out << "<span class='meta'> status.user.username} posted on
                        ${g.formatDate(date: status.dateCreated)}: </span>"
        out << "${status.message}"
        out << "</div>"
    }

}
```