Grails Web Application Framework

Weiterführende Themen zu Internet- und WWW-Technologien
Sommersemester 2011

Matthias Springer

Betreuer:
Matthias Bauer, Christian Willems
Prof. Dr. Christoph Meinel

21. Juli 2011

Inhaltsverzeichnis

1 Grails als Web Application Framework

Mit einem Web Application Framework ist ein Programmiergerüst [17] gemeint, das bei Entwicklung von dynamischen Web-Seiten, Web-Anwendungen und Web-Diensten [19] verwendet werden kann. Das Grails Application Framework bündelt bereits vorhandene Softwarekomponenten und ermöglicht eine schnelle Softwareentwicklung ohne aufwendige Installation und Konfiguration der benötigten Bestandteile.

1.1 Entwicklungsprozess und Deployment

Grails basiert auf der Java Servlet API und unterstützt als Programmiersprachen Java sowie Groovy, eine objektorientierte dynamische Programmiersprache, die ebenfalls zu Bytecode für die Java Virtual Machine (JVM) kompiliert wird. Vorhandene Java-Bibliotheken und die Java Standard-Bibliothek (Java EE) können also problemlos verwendet werden. Java-Bytecode-Dateien, Server Pages, externe Bibliotheken, statische Ressourcen und alle anderen benötigten Dateien werden zu einem Web Application Archive (WAR) [18] verpackt. Dieses Dateiformat entspricht dem bekannten Java Archive (JAR) Dateiformat[1] und wird für das Deployment auf einem Java Application Server benötigt. Beim weit verbreiteten Apache Tomcat geschieht das Deployment der Web-Anwendung dann beispielsweise durch das Kopieren der WAR-Datei in das `webapps`-Verzeichnis. Danach kann die Anwendung verwendet werden.

1.2 Bestandteile von Grails

Im Folgenden werden die wichtigsten Softwarekomponenten, aus denen Grails besteht, kurz vorgestellt [20]. Dieses Paper bezieht sich auf die zur Zeit aktuelle Grails-Version 1.3.7.

- **Apache Tomcat** ist der Java Application Server, der beim Debuggen verwendet wird[2].
- **Groovy** ist eine dynamisch typisierte Programmiersprache für die Java Virtual Machine.
- Die **HSQLDB** (Hyperthreaded Structured Query Language Database) ist eine In-Memory-Datenbank, die standardmäßig in der Entwicklungsphase verwendet wird.
- **Hibernate** ist für die objektrelationale Abbildung von Java- bzw. Groovy-Klassen auf eine relationale Datenbank zuständig und abstrahiert damit vom zu Grunde liegenden Datenbanksystem.
- Von der **Java Servlet API** wird an vielen Stellen abstrahiert, sie kann aber dennoch direkt angesprochen werden. Ebenso können **JavaServer Pages** (JSP) bzw. **Groovy Server Pages** (GSP) verwendet werden.

[1] Es handelt sich ebenfalls um ein ZIP-Archiv.
[2] Ältere Versionen von Grails verwenden Jetty.

- **SiteMesh** ist ein Layout- und Decoration-Framework [16], das ein konsistentes Layout und Design großer Web-Anwendungen sicherstellt.
- Das **Spring Framework** ist für Dependency Injection zuständig und ermöglicht die einfache Verwendung von Grails Services[3].

1.3 Geschichte von Grails

Die Entwicklung von Grails begann im Juli 2005. Der damalige Name des Projekts war noch *Groovy on Rails*, erst später wurde es in Grails umbeannt. Die erste Version 0.1 erschien am 29. März 2006 und seitdem wurde Grails ständig weiterentwickelt, sodass im Februar 2011 Version 1.3.7 veröffentlicht werden konnte [3]. Grails steht unter der freien Apache-Lizenz, jedoch sind auch Unternehmen wie *VMWare*[4] an der Entwicklung beteiligt. Da Grails auf Java basiert, ist es plattformunabhängig und kann auf jedem modernem Betriebssystem verwendet werden. Es wird lediglich das Java SDK 1.5 (oder neuer) benötigt.

2 Die Programmiersprache Groovy

Wie bereits erwähnt basiert Groovy auf Java und wird zu Java-Bytecode kompiliert. Groovy führt neue Konzepte wie zum Beispiel Closures ein. Bestehende Java-Bibliotheken und -Quelltextdateien können dabei weiterverwendet werden. Es ist sogar möglich, ganz normalen Java Quelltext zu schreiben.

2.1 Überblick über Groovy

Groovy ist eine dynamisch typisierte Programmiersprache. Der Typ einer mit dem `def`-Schlüsselwort dynamisch deklarierten Variablen steht also erst zur Laufzeit fest. Es besteht jedoch auch die Möglichkeit, Variablen statisch zu typisieren. Dann kann bereits der Compiler Verstöße gegen das Typsystem erkennen.

```
1 String x = "Hello World"
2 def y = "Hello World"
3 assert x == y
```

Listing 1: Statische und dynamische Typisierung

Listing 1 zeigt die statische und dynamische Deklaration einer String-Variablen. Außerdem zeigt es, dass der ==-Operator bei Strings auf Wertegleichheit prüft und abschließende Semikolons weggelassen werden können, wenn die Semantik des Quelltextes dann immer noch eindeutig ist[5].

[3] Grails Services werden in diesem Paper nicht behandelt, daher sei auf [12] verwiesen.
[4] Das Unternehmen SpringSource wurde 2009 von VMWare aufgekauft.
[5] Das ist zum Beispiel nicht der Fall, wenn mehrere Anweisungen in eine einzige Zeile geschrieben werden.

Die folgende Auflistung zeigt weitere leicht verständliche Unterschiede zu Java.

- Impliziter Import von `java.io.*`, `java.lang.*`, `java.net.*`, `java.util.*`.
- `println` als Kurzschreibweise für `System.out.println`.
- Semikolon und runde Klammern optional, wenn der Quelltext dann immer noch eindeutig interpretiert werden kann.
- `return`-Schlüsselwort optional.
- `public` als Standard-Sichtbarkeit für Felder und Methoden.
- Abfangen von Exceptions optional.

2.2 Closures

Bei Closures handelt es sich um anonyme Codeblöcke, die den aus anderen Programmiersprachen bekannten Lambda-Ausdrücken entsprechen. Der in einer Closure gespeicherte Code kann wie jedes andere Objekt hin- und hergeschoben werden. Eine Closure kann Argumente entgegennehmen und einen Wert zurückgeben. Listing 2 zeigt zwei äquivalente Closures, die die Fläche eines Kreises berechnen.

```
1 def area1 = {def radius -> return 3.14 * radius ** 2}
2 def area2 = {3.14 * it ** 2}
3 assert area1(42) == area2(42)
```

Listing 2: Verwendung von Closures

Closures werden in geschweifte Klammern gesetzt und Parameter können vor dem Pfeil (`->`) wie ganz normale Variablen definiert werden. Sie können wahlweise dynamisch oder statisch typisiert werden. Wird nur ein einziger Parameter erwartet, kann auf dessen explizite Definition verzichtet werden, er ist dann unter dem Bezeichner `it` verfügbar. Werte können wie in Methoden mit dem `return`-Schlüsselwort zurückgegeben werden[6].

2.3 Collections

Listen und Wörterbücher können in Groovy besonders einfach mit einer Python-ähnlichen Syntax verwendet werden.

Listing 3 verdeutlicht die Verwendung von Listen. Der Plus-Operator ist überladen, sodass er zur Konkatenation zweier Listen verwendet werden kann. Listen stellen außerdem die `collect`-Methode bereit, die eine Closure als Parameter erwartet, die auf jedes Element der Liste einzeln angewendet wird. Die Methode gibt dann eine Liste der von der Closure zurückgegebenen Werte zurück.

[6] Wie bereits erwähnt ist das `return`-Schlüsselwort optional.

```
1 def list = [1, 2, 3]
2 list += [4, 5]
3 assert list.collect {it * it} == [1, 4, 9, 16, 25]
```

Listing 3: Verwendung von Listen

Listing 4 zeigt die Verwendung von Wörterbüchern. Neue Tupel können wahlweise mit der Punkt-Syntax oder der Klammern-Syntax eingefügt werden. Die `each`-Methode iteriert über alle Elemente und führt die als Parameter übergebene Closure mit dem aktuellen Element als Parameter aus.

```
1 def map = ['FR':'Paris', 'GER':'Berlin']
2 map.'UK' = 'London'
3 map['USA'] = 'Washington, D.C.'
4
5 map.each {it.value += ' (Capital)'}
6 assert map['UK'] == 'London (Capital)'
```

Listing 4: Verwendung von Wörterbüchern

2.4 Groovy Strings

Groovy kennt zwei verschiedene Arten von String. Herkömmliche Java-Strings werden in einfache Anführungszeichen gesetzt. Werden doppelte Anführungszeichen verwendet, handelt es sich um Groovy-Strings, die beliebige Closures enthalten können, deren Rückgabewert dann in den Text eingefügt wird. Listing 5 verdeutlicht die Verwendung von Groovy-Strings.

```
1 assert "The answer is ${21 * 2}" == 'The answer is 42'
2 assert "This ${"is ${"crazy!"}"}" == 'This is crazy!'
```

Listing 5: Groovy-Strings und normale Strings

3 Grails

3.1 Grails Commands

Der Kommandozeilenbefehl `grails` kann zum Erstellen, Verwalten und Debuggen von Grails-Anwendungen verwendet werden. Die folgende Auflistung gibt einen Überblick über die wichtigsten Grails Kommandos.

- `grails clean` löscht alle temporär erzeugten und kompilierten Dateien.
- `grails create-app <name>` erstellt eine neue Grails-Anwendung mit dem Namen <*name*> in einem Unterverzeichnis mit dem gleichen Namen.

- `grails create-controller <name>` erstellt einen neuen Controller. Ähnliche Kommandos gibt es für Domain-Klassen, Plugins und Services.
- `grails help` zeigt weitere Kommandos an.
- `grails war` erstellt ein Web Application Archive für das Deployment auf einem Java Application Server.

3.2 Das MVC-Konzept

Das MVC-Konzept findet bei den meisten Web Application Frameworks Anwendung und strukturiert die Anwendung in drei Einheiten [15].

- Die **Steuerung** (engl. *controller*) nimmt Anfragen von Benutzern entgegen und vermittelt zwischen dem Modell und der Präsentation.
- Die **Präsentation** (engl. *view*) ist für die Aufbereitung der von der Steuerung erzeugten Daten, z.B. zu HTML, zuständig. Dazu können Groovy Server Pages erstellt werden.
- Das **Modell** (engl. *model*) enthält die Datenstruktur sowie die Geschäftslogik der Anwendung. Es wird in Domain-Klassen abgelegt und automatisch von *Hibernate* in ein relationales Datenbankschema überführt.

3.3 Controller

Anfragen, die Benutzer oder ein anderes System über die externe Schnittstelle (z.B. HTTP) an die Anwendung schicken, landen bei einem Controller. Das *URL Mapping*[7] legt fest, welche URLs bei welchem Controller landen und kann vom Entwickler verändert werden[8].

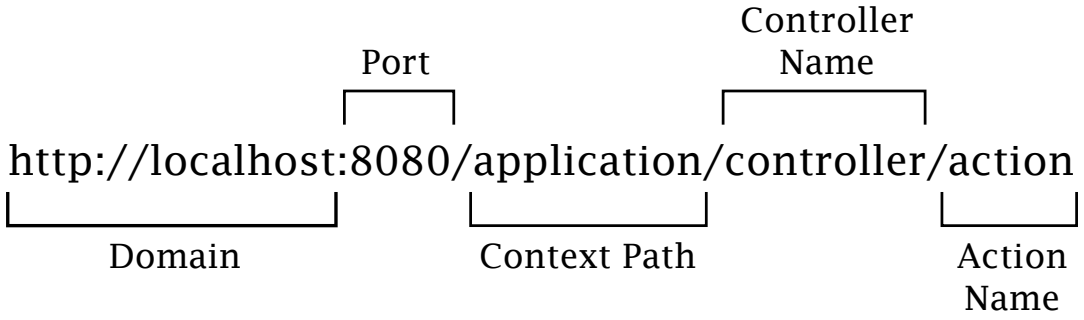

Abbildung 1: Standardmäßiges URL Mapping

[7] Abbildung nach [21].
[8] Das URL Mapping wird in der Datei `grails-app/conf/UrlMappings.groovy` definiert.

Der *Context Path* ist normalerweise der Name der Anwendung[9] und ist notwendig, damit mehrere Anwendungen auf dem gleichen Server ausgeführt werden können. Für jeden Controller wird eine Groovy-Klasse erstellt und im Verzeichnis `grails-app/controllers` gespeichert[10]. Für jede *Action* wird eine Closure erstellt. Deren Rückgabewert wird normalerweise an eine View weitergeleitet. Es ist aber auch möglich, mit dem `redirect`-Schlüsselwort die Anfrage an eine andere Action (auch in einem anderen Controller) zu deligieren. Mit dem `render`-Schlüsselwort kann die Action selbst die Antwort auf die Anfrage generieren und somit die View-Schicht überspringen.

```
1 package MyFirstApplication
2
3 class HelloController {
4   def sayHello = { redirect(action:"englishHello") }
5   def germanHello = { render "Hallo Welt!" }
6   def englishHello = {render "Hello World!" }
7 }
```

Listing 6: Aufbau eines Controllers

Wie aus Listing 6 ersichtlich ist, enden Controller immer mit dem Suffix *Controller*. Der Dateiname muss immer der Name der enthaltenen Klasse sein. Bei diesem Beispiel würde ein Browser bei Aufruf der Action `sayHello`[11] über den HTTP Status Code *302 Moved Temporarily* an die entsprechende URL der Action `englishHello` umgeleitet werden, worauf der Schriftzug *Hello World!* erscheint.

Soll eine Anfrage nicht nur an eine andere Action, sondern auch an einen anderen Controller übergeben werden, kann zusätzlich der Parameter `controller` verwendet werden. Ebenso ist es möglich, mit dem Parameter `uri` eine programminterne[12] URI oder mit dem Parameter `url` eine absolute URL zu spezifizieren.

3.3.1 Parameter

Innerhalb von Actions ist das Wörterbuch `params` verfügbar, mit dem auf übergebene Parameter zugegriffen werden kann. Dabei spielt es keine Rolle, ob die Parameter mittels *HTTP GET*, *HTTP POST* oder irgendeiner anderen REST-Methode übergeben wurden. Um bei einem `redirect` Parameter zu übergeben, kann mit dem Parameter `params` ein Wörterbuch übergeben werden.

Die in Listing 7 definierte Action zeigt bei Aufruf eine Nachricht mit dem als Parameter übergebenen Namen[13].

[9] Nach dem Deployment in der Regel der Name der WAR-Datei.

[10] Dabei muss natürlich jede Klasse - wie auch in Java - im entsprechenden Unterverzeichnis mit dem Paketnamen liegen.

[11] `http://localhost:8080/MyFirstApplication/hello/sayHello`

[12] *Intern* bedeutet, dass sich die referenzierte Ressource innerhalb der aktuellen Grails-Anwendung befinden muss, z.B. `/hello/sayHello`.

[13] `http://localhost:8080/MyFirstApplication/welcome/sayHello?name=Adam` würde

```
1 package MyFirstApplication
2
3 class WelcomeController {
4   def sayHello = { render "Welcome ${params.name}, have a nice day!" }
5 }
```

Listing 7: Zugriff auf Parameter

3.3.2 Java HttpServlet API

Grails baut auf die Java Servlet API auf und stellt selbst viele grundlegende Methoden, wie beispielsweise die `render`-Methode, bereit, die dem Entwickler die Arbeit erleichern. Trotzdem ist es oft notwendig, direkt auf die Servlet API zuzugreifen. Im Folgenden werden einige hilfreiche Operationen der Servlet API vorgestellt. Erwartet eine Methode keinen Parameter, kann das `get`-Präfix weggelassen und auf die Methode wie auf ein Feld zugegriffen werden.

- `request` implementiert das Interface `HttpServletRequest` und liefert Informationen zur aktuellen Anfrage [8].
  - `getCookies()` gibt ein Array aller übertragenen Cookies zurück.
  - `getHeader(String name)` ermittelt den Wert eines Feldes im Anfrage-Header[14].
  - `getMethod()` gibt die Art der Anfrage, also z.B. `GET`, `PUT` oder `POST` zurück[15].
- `response` implementiert das Interface `HttpServletResponse` und kann zum Senden einer Antwort benutzt werden [9].
  - `setStatus(int sc)` legt einen HTTP Status Code fest.
  - `getOutputStream()` gibt eine Referenz auf den Ausgabe-Stream zurück, der z.B. zum Senden von Binärdaten verwendet werden kann[16].
  - `addCookie(Cookie cookie)` übergibt ein Cookie.
- `session` implementiert das Interface `HttpSession`, dient zur Speicherung von clientspezifischen Session-Daten und kann wie ein Wörterbuch verwendet werden[17].

den Text *Welcome Adam, have a nice day!* erzeugen.

[14] Damit kann beispielsweise der Browser (User-Agent) des Benutzers ermittelt werden.

[15] Wenn REST-Methoden unterschiedlich behandelt werden sollen, sollte ein entsprechendes URL Mapping verwendet werden. Siehe dazu [14].

[16] Dafür wurde der « Operator überladen, sodass `response.outputStream « byteArray` geschrieben werden kann.

[17] Weitere Informationen dazu in der Grails Quick Reference [11]

3.4 View

Eine von einem Controller generierte Antwort auf eine Anfrage kann an eine View weitergeleitet werden, um die Daten beispielsweise grafisch aufzubereiten. Wird keine `render-`, `redirect-` oder ähnliche Anweisung am Ende einer Action ausgeführt, wird die View in der Datei `grails-app/views/controller/action.gsp` verwendet, wobei *controller* und *action* für die gerade ausgeführte Action im aktuellen Controller stehen[18]. Der Rückgabewert einer Action sollte ein Wörterbuch sein. Aus der View kann dann auf jedes Element des Wörterbuchs zugegriffen werden.

Mit der `render`-Methode ist es möglich, gar keine oder eine andere View zu verwenden. Mit dem Parameter `model` wird dann das Wörterbuch übergeben[19].

Views sind Groovy Server Pages (GSP) und enthalten normalerweise HTML-Code[20]. In den HTML-Code können Closures eingefügt werden, die später durch ihren Rückgabewert ersetzt werden. Die Syntax entspricht dabei der Syntax von Groovy-Strings. Außerdem existieren Grails-spezifische XML-Tags, die mit dem Präfix `g:` beginnen. Die folgende Auflistung gibt einen Überblick über die wichtigsten Tags.

- `<g:link action="action" controller="contr">Text</g:link>` erzeugt einen Hyperlink, der auf eine bestimmte Action verweist.
- `<g:each in="${parameter}"> ... </g:each>` erzeugt eine for-each-Schleife, die für jedes Element in der Collection `parameter` den GSP-Code innerhalb des Tags auflistet. Auf die Laufvariable `it` kann über Closures zugegriffen werden. Die Variable `parameter` muss zuvor natürlich mit dem Wörterbuch übergeben worden sein.
- `<g:if test="${bedingung}"> ... </g:if>` testet auf die Bedingung und listet den GSP-Code genau dann auf, wenn die Bedingung *wahr* ist.
- `<g:form controller="contr" action="action"> ... </g:form>` erzeugt eine HTML-Form, die mittels *HTTP POST* abgeschickt wird [5].

Listing 8 zeigt den Controller `UserController`, der bei Aufruf der Action `list`[21] eine Liste mit den Namen zweier Personen generiert und an die entsprechende View weiterleitet. Für jede Person wird dafür ein Wörterbuch erstellt.

```
1 package MyFirstApplication
2
3 class UserController {
4   def list = {
5     def users = []
```

[18] Existiert diese Datei nicht, wird nach Aufruf der Action eine Grails-Fehlermeldung angezeigt.

[19] Siehe dazu [7].

[20] Es wäre auch möglich JavaScript-Dateien oder andere Dateien auf diese Art und Weise dynamisch zu erzeugen, wofür aber Views nicht verwendet werden sollten.

[21] `http://localhost:8080/MyFirstApplication/user/list`

```
6       users.add([first:"George", last:"Lukas", city:"Naboo"])
7       users.add([first:"Douglas", last:"Adams", city:"Magrathea"])
8       return [userlist:users]
9     }
10  }
```

Listing 8: Übergabe von Parametern an eine View

Die zugehörige View befindet sich in der Datei `grails-app/views/user/list.gsp`. Dort ist die Liste der Personen unter dem Namen `userlist` bekannt. Die View gibt die beiden Personen in einer für Menschen lesbaren Form aus.

```
1 <html><body>
2 <h1>Registered users</h1>
3 <g:each in="${userlist}">
4   <p>${it.first} ${it.last} living in ${it.city}</p>
5 </g:each>
6 </body></html>
```

Listing 9: Zugriff auf Parameter in Views

Momentan sind die Personen noch fest einprogrammiert. Das soll sich nun ändern.

3.5 Model

Ein Model definiert die Datenstruktur, die zur Speicherung von Informationen in der Anwendung verwendet wird. Es handelt sich um eine Groovy-Klasse, die öffentliche Felder enthält. Hibernate wertet jedes Model einzeln aus und erstellt daraus je eine Tabelle in einer relationalen Datenbank. Listing 10 zeigt ein Model für die Speicherung von Personendaten und befindet sich in der Datei `grails-app/domain/MyFirstApplication/Person.groovy`.

```
1  package MyFirstApplication
2
3  class Person {
4      static constraints = {
5        first(size:3..50)
6        last(size:3..50, nullable:false)
7      }
8
9    String first
10   String last
11   String city
12 }
```

Listing 10: Aufbau eines Models

Das Feld `constraints` legt zusätzliche Anforderungen an die Daten fest. In diesem Beispiel wurde festgelegt, dass die Anzahl der Zeichen bei Vor- und Nachnache zwischen 3 und 50 liegen muss, sowie dass auf jeden Fall ein Nachname angegeben werden muss.

Der Controller `UserController` wurde nun etwas erweitert. In der Action `list` werden die Daten nun aus der Datenbank geladen. `getAll()` gibt eine Liste mit allen Personen zurück und für jede Person wird ein Wörterbuch erstellt, das dann der Liste für das View hinzugefügt wird.

```
1  package MyFirstApplication
2
3  class UserController {
4    def list = {
5      def users = []
6      Person.getAll().each { users += [first:it.first, last:it.last, city:it
           .city] }
7      return [userlist:users]
8    }
9
10   def create = { }
11
12   def submit = {
13     def user = new Person(first:params.first, last:params.last, city:
           params.city);
14     user.save()
15     redirect(action:"list")
16   }
17 }
```

Listing 11: Zugriff auf Models

Wird die Action `create` aufgerufen, wird sofort die folgende View[22] angezeigt. Sendet der Benutzer das Formular per HTTP POST ab, landet der Inhalt in der Action `submit`, wo der Datensatz gespeichert und der Benutzer wieder zur Übersichtsseite umgeleitet wird.

```
1 <html><body>
2 <h1>Add new user</h1>
3 <g:form action="submit">
4   First name: <g:textField name="first" /><br />
5   Last name: <g:textField name="last" /><br />
6   City: <g:textField name="city" /><br />
7   <g:submitButton name="submit" value="Submit" />
8 </g:form>
9 </body></html>
```

Listing 12: HTML-Form als GSP

3.5.1 Model Constraints

Die folgende Auflistung gibt einen Überblick über die am häufigsten benötigten Beschränkungen an Models. Wird gegen eine Beschränkung verstoßen, tritt eine Exception auf und der

[22] `grails-app/views/user/create.gsp`

Datensatz wird nicht gespeichert.

- `inList:('Potsdam', 'Berlin', 'Munich')` legt fest, dass nur bestimmte Werte zulässig sind.
- `matches:/http:\/\/.+/` stellt sicher, dass der Wert einen regulären Ausdruck erfüllt. In diesem Beispiel muss er mit `http://` beginnenn.
- `max:100` legt einen Maximalwert (z.B. bei Zahlen) fest.
- `unique:true` stellt sicher, dass jeder Wert in diesem Feld nur einmal verwendet wird.
- Mit `validator:{it.startsWith('http://')}` können eigene Beschränkungen erstellt werden [13], z.B. dass der Wert mit `http://` beginnen muss.

3.5.2 Daten aus der Datenbank holen

Die in Listing 11 gezeigte Operation `getAll()` ermittelt alle Datensätze. Um spezielle Datensätze abzufragen, können sog. *dynamic finders* verwendet werden. Dabei handelt es sich um Methoden, die die in Listing 13 spezifizierte Form haben[23].

```
1 <findall_expression> ::= "findAllBy" <statement>
2 <statement> ::= <statement> <bool> <statement> || <attribute> <comparator>
      || <attribute>
3 <bool> ::= "and" || "or"
```

Listing 13: Dynamic finders in erweiterter Backus-Naur-Form.

Die folgende Auflistung zeigt eine Reihe an möglichen Komparatoren (`Comparator`) [2].

- `LessThan` sucht Datensätze, deren angegebenes Feld kleiner als der übergebene Parameter ist.
  Beispiel: `findAllByAgeLessThan(18)` sucht alle minderjährige Personen.
- `LessThanEquals`, `GreaterThan`, `GreaterThanEquals` analog.
- `Like` sucht Datensätze, deren angegebenes Feld *wie* der Parameter ist. Es handelt sich um den aus SQL bekannten *Like* Operator.
  Beispiel: `findAllByFirstLike('Ma%')` sucht alle Personen, deren Vorname mit *Ma* beginnt.
- `NotEqual` sucht Datensätze, deren angegebenes Feld ungleich dem Parameter ist.
  Beispiel: `findAllByFirstNotEqual('Adam')` sucht alle Personen, deren Vorname nicht *Adam* ist.

[23] Darstellung nach [2].

- Wird kein Komparator angegeben, handelt es sich um Gleichheit.
  Beispiel: `findAllByFirst('Adam')` sucht alle Personen, deren Vorname *Adam* ist.

Wie aus der Backus-Naur-Form ersichtlich wird, können auch mehrere Ausdrücke mit boolscher Logik verknüpft und sehr komplexe Anfragen formuliert werden. Es kann aber auch die *Hibernate Query Language* verwendet werden, die eine SQL-artige Syntax aufweist.

3.5.3 Verknüpfung von Models

Um Assoziationen zwischen Instanzen von Models herzustellen, bietet Grails eine spezielle Schreibweise an. Listing 14 zeigt, wie festgelegt werden kann, dass eine Person mehrere Freunde haben kann.

```
1 package MyFirstApplication
2
3 class Person {
4   static hasMany = [friends:Person]
5
6   String first
7   String last
8   String city
9 }
```

Listing 14: Assozitionen zwischen Models

Das `hasMany`-Feld definiert eine 1:n-Assoziation, die auch mit dem Ausdruck `java.util.Set<Person> friends` umschreiben werden könnte [6]. Die `hasMany`-Variante stellt jedoch referentielle Integrität über eine entsprechende Fremdschlüssel-Beziehung im darunterliegenden Datenbanksystem sicher. Es können auch mehrere `hasMany`-Beziehungen definiert werden, das statische Feld kann wie ein Wörterbuch verwendet werden.

Analog können auch 1:1-Assoziationen mit dem `hasOne`-Feld definiert werden. Über das `belongsTo`-Feld kann ein *Besitzer* festgelegt werden [4]. Damit wird festgelegt, wie die referentielle Integrität sichergestellt wird, also ob Änderungen kaskadiert oder abgelehnt werden.

3.6 Debugging

3.6.1 Testen

Grails Anwendungen werden mit dem Kommando `grails run-app` gestartet. Soll der Application Server auf einem bestimmten Port gestartet werden, kann der Parameter `-Dserver.port=9090` verwendet werden [10]. Läuft der Server bereits und werden dann Änderungen an Quelltext-Dateien vorgenommen, muss er nicht neu gestartet werden. Grails überprüft in regelmäßigen Zeitabständen[24] den Zeitstempel aller Java- und Groovy-Dateien und kompiliert diese ggf. automatisch neu.

[24] Standardeinstellung ist 3 Sekunden.

3.6.2 Scaffolding

Scaffolding ermöglicht es, die Daten eines Models zu verwalten. Es wird einfach ein neuer Controller erzeugt, der außer der Festlegung des zu untersuchenden Models keinerlei Quelltext enthält. Listing 15 verdeutlicht die Verwendung von Scaffolding.

```
1 package MyFirstApplication
2
3 class UserDebugController {
4   static scaffold = Person
5 }
```

Listing 15: Verwendung von Scaffolding

Wird dieser Controller dann ohne Action aufgerufen, werden alle Instanzen des Models in einer schlichten Weboberfläche angezeigt. Außerdem können neue Instanzen erstellt oder bestehende Instanzen gelöscht werden. Scaffolding kann sowohl zum Debugging, als auch als Administrationsoberfläche verwendet werden.

4 Fazit

Grails richtet sich vor allem an Java-Entwickler. Während das Erstellen von Java EE Anwendungen oft sehr zeitaufwendig ist, folgt Grails dem *Convention over Configuration* [1] Paradigma und ist deshalb auch für Anfänger schnell erlernbar. Auch Entwickler, die bereits mit anderen Web Application Frameworks gearbeitet haben, werden sich schnell zurechtfinden, da viele bekannte Konzepte wie MVC oder REST auch in Grails voll unterstützt werden.

Literatur

[1] Grails Documentation: Getting Started, 2011. [Online; Stand 21. Juli 2011] http://grails.org/doc/1.0.x/guide/2.%20Getting%20Started.html.

[2] Grails Documentation: Object Relational Mapping (GORM), 2011. [Online; Stand 18. Juli 2011] http://grails.org/doc/1.0.x/guide/5.%20Object%20Relational%20Mapping%20%28GORM%29.html.

[3] Grails (framework) — wikipedia, the free encyclopedia, 2011. [Online; Stand 18. Juli 2011] http://en.wikipedia.org/w/index.php?title=Grails_(framework)&oldid=439901967.

[4] Grails Quick Reference: belongsTo, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Domain%20Classes/belongsTo.html.

[5] Grails Quick Reference: form, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Tags/form.html.

[6] Grails Quick Reference: hasMany, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Domain%20Classes/hasMany.html.

[7] Grails Quick Reference: render, 2011. [Online; Stand 21. Juli 2011] http://www.grails.org/doc/latest/ref/Controllers/render.html.

[8] Grails Quick Reference: request, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Servlet%20API/request.html.

[9] Grails Quick Reference: response, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Servlet%20API/response.html.

[10] Grails Quick Reference: run-app, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Command%20Line/run-app.html.

[11] Grails Quick Reference: session, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Servlet%20API/session.html.

[12] Grails Quick Reference: The Service Layer, 2011. [Online; Stand 21. Juli 2011] http://www.grails.org/doc/latest/guide/8.%20The%20Service%20Layer.html.

[13] Grails Quick Reference: validator, 2011. [Online; Stand 18. Juli 2011] http://www.grails.org/doc/latest/ref/Constraints/validator.html.

[14] Grails Quick Reference: Web Services, 2011. [Online; Stand 21. Juli 2011] http://www.grails.org/doc/latest/guide/13.%20Web%20Services.html.

[15] Model-view-controller — wikipedia, the free encyclopedia, 2011. [Online; Stand 15. Juli 2011]

http://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=439691864.

[16] Sitemesh3 overview, 2011. [Online; Stand 22. Juni 2011] http://www.sitemesh.org/overview.html.

[17] Software framework — wikipedia, the free encyclopedia, 2011. [Online; Stand 21. Juni 2011] http://en.wikipedia.org/w/index.php?title=Software_framework&oldid=434460967.

[18] War file format (sun) — wikipedia, the free encyclopedia, 2011. [Online; Stand 22. Juni 2011] http://en.wikipedia.org/w/index.php?title=WAR_file_format_(Sun)&oldid=421372260.

[19] Web application framework — wikipedia, the free encyclopedia, 2011. [Online; Stand 21. Juni 2011] http://en.wikipedia.org/w/index.php?title=Web_application_framework&oldid=433179050.

[20] B. Jawad. *Groovy and Grails Recipes*. Recipes: a Problem-solution Approach. Apress, 2008.

[21] Graeme Rocher and Jeff Brown. *The Definitive Guide to Grails*. Apress, second edition, 2009.