Seminararbeit
Entwicklung einer Webanwendung mit Groovy und Grails

Betreuer: Stefan Seedorf

Von
Markus Hartmann
27.11.2009

Inhaltsverzeichnis

1 Einleitung....1
2 Groovy & Grails....2
2.1 Groovy....2
2.1.1 Zielsetzung....2
2.1.2 Syntax....3
2.1.3 Groovy in der Grails-Umgebung....5
2.2 Grails....6
2.2.1 Zielsetzung....6
2.2.1.1 Scaffolding....7
2.2.1.2 Konvention über Konfiguration....9
2.2.1.3 Erweiterbarkeit....10
2.2.2 Komponenten....11
2.2.2.1 Übersicht....11
2.2.2.2 GORM / Hibernate....12
2.2.2.3 Spring....14
2.2.2.4 Datenbank....15
2.2.2.5 Jetty....15
2.2.2.6 SiteMesh....16
2.2.3 Weitere Funktionen....16
2.2.3.1 Umgebungen....17
2.2.3.2 Tests....18
2.2.3.3 Tag-Bibliotheken....18
3 Projekt....19
3.1 Aufbau / Domänenmodell....19
3.2 Funktionen & Beispiele....20
4 Analyse....23
4.1 Allgemein....23
4.2 Vergleich mit J2EE....24
4.3 Vergleich mit Ruby on Rails....25
4.4 Performance-Analyse....25
5 Fazit....28
6 Literaturverzeichnis....29
7 Abbildungsverzeichnis....31

Abkürzungsverzeichnis

API	Application Programming Interface
CoC	Convention over Configuration
CRM	Customer Relationship Management
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DRY	Don't Repeat Yourself
ERP	Enterprise Resource Planning
GORM	Grails objektrelationales Mapping
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HCA	Hibernate Criteria API
HQL	Hibernate Query Language
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
IoC	Inversion of Control
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JCP	Java Community Process
JSF	Java Server Faces
JSP	Java Server Pages
JVM	Java virtuelle Maschine
ORM	Objektrelationales Mapping
POGO	Plain Old Groovy Object
RIA	Rich Internet Application
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1 Einleitung

In den vergangenen Jahren hat die Nutzung des Internets, besonders im Bereich Web 2.0, auf Grund von neuen Technologien wie *Asynchronous JavaScript and XML(Ajax)* und Breitband-Internetanschlüssen, stark zugenommen. Laut aktuellen Marktforschungen ist die Zahl der Privatanwender im Jahr 2009 auf über eine Milliarde angewachsen. (comScore 2009) Vor allem junge Anwender nutzen die Möglichkeit zur Kommunikation und Informationsbeschaffung über Weblogs, Social Networking oder einfachen Suchmaschinen, wie Google, die laut comScore von 77% aller Internetanwendern regelmäßig genutzt wird. Doch auch bei mittelständischen und großen Unternehmen ist ein deutlicher Trend erkennbar, dass sie mit ihren Enterprise-Applikationen, wie *ERP*- oder *CRM*-Systemen, über eine webbasierte, *grafische Benutzerschnittstelle (GUI)* arbeiten wollen. Dieser Trend erfordert nicht nur eine Weiterentwicklung der Performance von aufwändigen Webseiten und Applikationen, sondern auch den Ausbau und die Erweiterung der Webtechnologie, einhergehend mit einer größeren Benutzer-, bzw. Entwicklerfreundlichkeit. So entstehen auch immer mehr Programmiergerüste, sogenannte Frameworks, um die Entwickler beim Erstellen von einfachen Webseiten bis hin zu *Rich Internet Applications (RIA)* zu unterstützen, indem Entwurfsmuster oder auch Klassenbibliotheken zur Verfügung gestellt werden. Eine wichtige Eigenschaft von Frameworks ist, dass möglichst unkompliziert performante und stabile Applikationen, meist gemäß einer Trennung in Schichten, entwickelt werden können. Das bedeutet unter anderem die Berücksichtigung der Hürde der Client/Server basierten Kommunikation zu vereinfachen und den Konfigurations-, bzw. Entwurfsaufwand zu minimieren.

Diese Seminararbeit befasst sich mit dem noch relativ jungen Framework *Grails* und der im Zusammenhang mit Grails hauptsächlich verwendeten Programmiersprache Groovy. Der erste Teil beschreibt die grundsätzlichen Funktionsweisen und Konzepte der Sprache Groovy und gibt einen Überblick über die anderen in der Grails Umgebung verwendeten Komponenten. Im

zweiten Teil folgt ein praktisches Beispiel anhand einer Applikation. Eine Analyse nach den Kriterien Performance, Lernkurve und Anwendbarkeit erfolgt im dritten Teil im Vergleich mit der klassischen *Java 2 Enterprise Edition(J2EE)* Technologie und dem Framework Ruby on Rails.

2 Groovy & Grails

Grails bietet als Framework viele klassische Funktionen wie *Inversion of Control (IoC)* oder *Dependency Injection*, aber auch einige Klassenbibliotheken die den Programmierprozess mit Standardfunktionalitäten vereinfachen sollen. Dabei baut Grails zu einem Großteil auf der Programmiersprache Groovy auf, die hier im Detail betrachtet wird.

2.1 Groovy

Groovy ist eine kompilier- und interpretierbare Programmiersprache. Das heißt, sie kann während der Entwicklung interpretiert ausgeführt, aber auch vor der Ausführung vollständig in Bytecode übersetzt werden, um eine bessere Performance zu erzielen. Inspiriert wurde Groovy hauptsächlich von Sprachen wie Java, Ruby, Python, Perl oder Smalltalk. (Judd, Nusairat, Shingler 2008) Mit welcher Intention Groovy entwickelt wurde und welche Basisfunktionalitäten enthalten sind, wird im folgenden beschrieben.

2.1.1 Zielsetzung

Groovy wird von dem *Java Community Process (JCP)* betreut und wird daher auch vor allem durch die Programmiersprache Java und deren Entwicklung beeinflusst. Bereits im Laufe des Entwicklungsprozesses wurde Groovy für die *Java Virtuelle Maschine (JVM)* spezifiziert. Sie ist vollständig darauf lauffähig und ist daher einerseits Plattform unabhängig, aber auch zu 100% kompatibel zu allen auf Java basierenden Anwendungen, indem sie die komplette *Java API* (*Application Programming Interface*) implementiert. (Rocher & Brown 2009) Mit dieser stark ausgeprägten Kompatibilität steht Groovy in Kontrast zu vielen anderen Sprachen, die dagegen einfach nur auf der JVM lauffähig sind.

Groovy verfügt über einen eigenen Compiler, der Groovy- und Java-Quellcode in reinen Java-Bytecode übersetzt. Es ist aber auch, wie bereits erwähnt, möglich auf das vorherige Kompilieren zu verzichten. Die Implementierung der *Java API* ermöglicht eine enge Verzahnung zwischen Java und Groovy. So können Klassen beider Sprachen in einem Programm verwendet werden. Groovy ist damit besonders ansprechend für Entwickler aus der Java Umgebung, da es eine rein ergänzende Funktionalität bietet. Es können sämtliche Java-Bibliotheken und Komponenten problemlos in Groovy eingebunden und auch vollständig in der gewohnten Java Syntax gearbeitet werden. So funktioniert unter anderem auch herkömmliches JUnit, ergänzt mit einigen spezifischen Groovy-Befehlen.

Das Ziel von Groovy ist es, sich vollständig in bereits bestehende Java-Systeme zu integrieren und die Komplexität, die im Bereich der Java Entwicklung und speziell im Bereich der *Java 2 Enterprise Edition (J2EE)* besteht, deutlich zu reduzieren. Komplexität kann z.B. in Form von Mappings via XML, Overhead im Quellcode durch strenge Schichtenarchitektur, oder ähnlichem bestehen. Groovy bietet, wie Java, objektorientierte Funktionalitäten, aber auch viele Skriptsprachen-Elemente wie z.B. dynamische Klassenzugehörigkeit. (Tilly 2004)

2.1.2 Syntax

Wie bereits erwähnt, ist es, im Bereich der Groovy Entwicklung, möglich nativen Java Code zu verwenden, der dann mit dem Groovy Compiler in Java Bytecode übersetzt wird. Groovy erweitert aber darüber hinaus die normale Java-Syntax mit vielen eigenen Funktionalitäten, vereinfacht dadurch den Entwicklungsprozess und gestaltet den Quellcode deutlich kürzer und lesbarer. So ist Groovy auch als Skriptsprache lauffähig, was bedeutet, dass auf eine Main-Methode wie in Java z.B. üblich, vollständig verzichtet werden kann. (Judd et al. 2008)

Als weitere Vereinfachung besteht in Groovy die Möglichkeit auf eine dynamische Typisierung (*duck-typing*) zurückzugreifen. (Tilly 2004) Dabei müssen Variablen keine Datentypen zugewiesen werden. Diese werden automatisch, je nach Anforderung, vom Compiler bei der Übersetzung erkannt und vergeben. Dadurch kann der Quellcode deutlich reduziert werden, allerdings entfernt sich Groovy damit weiter von „ordentlicher“ Programmierung, da z.B. die Überprüfung von Tippfehlern in Variablennamen deutlich erschwert wird. Groovy kommt zur weiteren Übersicht auch ohne Strichpunkte am Ende von Statements aus, erlaubt die Übergabe von Parametern anhand des Namens, was das Aufrufen von Methoden erleichtert und kann anstelle von Return-Anweisungen die letzte Zeile einer Funktion, bzw. Methode, verwenden. (König & Rocher 2006)

Eine Besonderheit der Programmiersprache Groovy ist die Verwendung von Closures. Eine Closure ähnelt auf den ersten Blick einer Funktion, bzw. Methode. Closures werden jedoch wie Objekte behandelt. Sie können anderen Funktionen als Parameter übergeben werden. Darüber hinaus haben sie Zugriff auf alle Variablen, die in dem selben *Scope* wie die Definition der Closure stehen. (www.javabeat.net o. J.)

Beispiel Closure:

```
01 def Liste = [„Markus“, „Dominik“, „Christoph“]
02 def sagHallo = { println „Hallo ${it} }
03 liste.each(sagHallo)
```

Mittels eines integrierten Templatesystems kann mit Groovy innerhalb von Webseiten vergleichbar mit PHP oder *Java Server Pages(JSP)* entwickelt werden.

Trotz der Ausführbarkeit von Groovy als Skript, wird der Quellcode in Templates nicht zur Laufzeit als abstrakter Syntax-Baum ausgeführt, sondern vorher in Java-Bytecode übersetzt. (Judd et al. 2008)

Die Groovy-Syntax berücksichtigt speziell auch die Verarbeitung von regulären Ausdrücken. Im Gegensatz zu Java müssen nicht extra Patterns und Matcher definiert werden. Im Quelltext kann der find-Ope*rato*r (=~), für das Auffinden von Zeichenfolgen innerhalb eines Strings, bzw. der match-Operator (==~), für exakte Übereinstimmungen, verwendet werden. (König & Rocher 2006) Wahlweise kann ein String auch mit der *~string-Anweisung* kompiliert werden um, wie in Java, die Laufzeit bei häufigen Vergleichen zu verkürzen.

Eine weitere Funktion von Groovy sind überladene Operatoren. Diese werden z.B. von der nativen Syntax für Listen verwendet. Ein Beispiel ist die Methode *add* um Elemente hinzuzufügen.

Beispiel für überladene Operatioren:

```
01 def liste = []
02 liste.add „Markus“
03 liste << „Markus“
```

In diesem Beispiel sind Zeile zwei und drei äquivalent. Operatoren stellen Methoden dar, die auf den Objekten ausgeführt werden. So steht z.B. „<<“ für die Methode „add“ oder „++“ für „next“. Diese Methoden können bei eigenen Klassen implementiert werden. (Judd et al. 2008)

Groovy bietet noch eine Vielzahl weiterer Funktionen, auf die jedoch in dieser Arbeit nicht näher eingegangen wird. Unter anderem unterstützt Groovy eine native Darstellung für bigdecimal und biginteger, bietet eine eigene Syntax für Collections, Maps und Listen und den Groovy-spezifischen GString-Datentyp. (König & Rocher 2006)

2.1.3 Groovy in der Grails-Umgebung

Groovy ist als eine Ergänzung von Java, die viele Prozesse in der Entwicklung vereinfacht und schnellere Entwicklungszeiten verspricht, ideal für den Einsatz in der Grails Plattform. Hierbei wird Groovy für Domainklassen zum Darstellen von Persistenzobjekten, als Skriptsprache zum Ausführen von serverseitigem

Code und die Template Engine von Groovy zur Darstellung der Elemente der Präsentationsschicht verwendet. (Laforge 2007)

Alle Konfigurationen sind innerhalb des Grails-Frameworks mit Groovy Skripten realisiert. Eine Konfiguration über XML Dateien ist so fast vollständig obsolet. (Rocher & Brown 2009)

2.2 Grails

Grails ist ein MVC aktionsbasiertes Web Framework, zur agilen Softwareentwicklung, dass von *SpringSource* betrieben und von *codehouse.org* gehostet wird. In diesem Kapitel werden die allgemeinen Funktionalitäten, Komponenten und Grundideen, welche bei der Entwicklung zugrunde gelegt wurden, erläutert.

2.2.1 Zielsetzung

Wie viele andere Frameworks setzt Grails auf eine strenge Trennung der Applikationskomponenten in die Bereiche *Model*, *View* und *Controller*. Der View Bereich besteht aus *Groovy Server Pages* (GSP), die *SiteMesh* als Templatesystem implementieren und wird um Grails eigene Tags ergänzt. Aktionen vom Anwender werden von der Präsentationsschicht direkt an die entsprechenden Controller weitergeleitet. (Laforge 2007) Die Controller leiten den Anwender auf andere Ansichten, zeigen Daten aus dem Datenmodell an oder modifizieren diese. Das Datenmodell und die Controller sind vollständig in Groovy, in sogenannten *Plain Old Groovy Objects (POGOs),* implementiert. Das Auffinden, bzw. die Kommunikation der Komponenten untereinander, wird von Grails durch Konvention und das Spring Framework vorgenommen. (Judd et al. 2008)

Im Vergleich zu anderen Web-Frameworks bietet Grails eine vollständige Entwicklungsplattform, die nicht nur für einfache Web-Anwendungen, sondern auch für *WebServices* und *Batch*-Verarbeitung geeignet ist, an. Diese Plattform bietet eine Vielzahl von Funktionen, wie z.B. eine integrierte Datenbank oder einen Applikations-Server, so wird eine Einrichtung dieser Komponenten zur

Entwicklung obsolet. Grails implementiert darüber hinaus die *Java EE Plattform,* ein Projekt kann also als *Web Archive* auf Java Servern eingesetzt werden. (Judd et al. 2008)

Frameworks werden im Allgemeinen an der Ausführung von spezifischen Konzepten gemessen. So sind unter anderem Modifizierbarkeit, Erweiterbarkeit, Skalierbarkeit und Portierbarkeit wichtige Kriterien zur Bewertung. Darüber hinaus sind natürlich die Einfachheit und Geschwindigkeit beim Erstellen von Applikationen und eine möglichst einfache Lernkurve entscheidende Kriterien für Web-Frameworks. Im Folgenden werden Bestandteile der Grails-Plattform, die ausschlaggebend für die Bewertung nach diesen Kriterien sind, näher erläutert.

2.2.1.1 Scaffolding

Grails soll, genauso wie Groovy, der Intention nach die Entwicklung von Applikationen möglichst vereinfachen, indem unter anderem der Konfigurationsaufwand minimiert wird. So kann eine Grails Anwendung, mit vollständiger *Create, Read, Update* und *Delete (CRUD)* Funktionalität, mit nur einer Domainklasse erstellt werden. In der Grails Umgebung nennt sich dieser Prozess *Scaffolding*. Bei diesem Vorgang wird eine vollständige Applikation auf Basis von Attributen und gegebenenfalls Constraints (z.B. nullable, maxLength...) automatisch generiert. Unterschieden wird dabei in *dynamisches*- und *statisches Scaffolding*. (Laforge 2007)

Bei *dynamischen Scaffolding* genügt es einen leeren Controller, mit einem *Scaffolding*-Verweis auf die zugehörige Domainklasse, zu erstellen. Bei jeder Änderung der Domainklasse wird der Controller und die Präsentationsschicht automatisch neu generiert.

Das *statische Scaffolding* über die Anweisung *generate-all* erstellt sämtliche Controller und Dateien der Präsentationsschicht zu dem Zeitpunkt der Eingabe, diese können dann manuell verändert, bzw. durch erneutes Generieren überschrieben werden.

Es werden beim *Scaffolding*-Vorgang immer mehrere Ansichten erstellt, die die Elemente in einer Tabelle, sowie das Aktualisieren und Erstellen von neuen Elementen unter Berücksichtigung der jeweiligen Constraints ermöglichen. (Rocher & Brown 2009)

Folgende Grafik zeigt ein einfaches Beispiel für die drei Teile einer mit Scaffolding erstellte Anwendung, für die nur eine Domainklasse *User* geschrieben wurde.

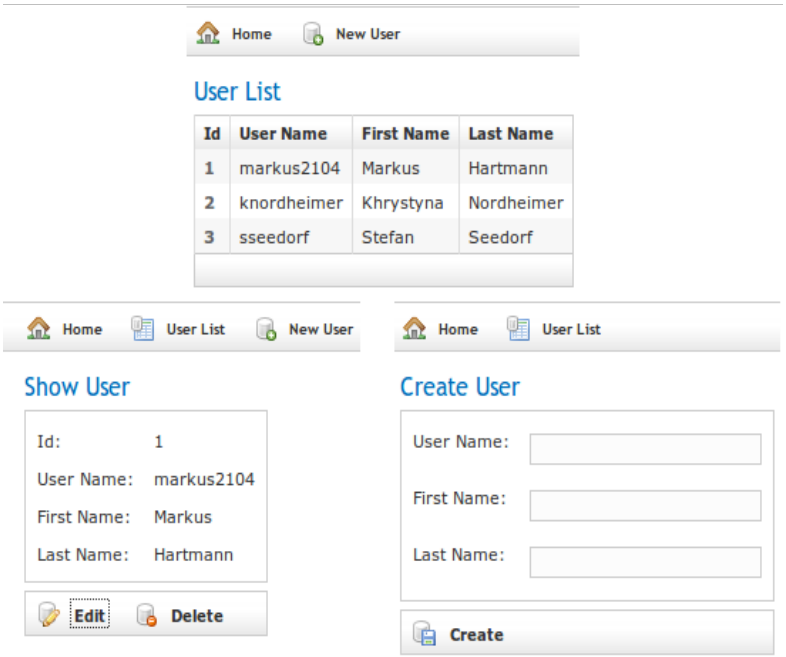


Abbildung 1: Scaffolding

Eines der wichtigsten Prinzipien von Grails ist *Don't Repeat Yourself (DRY)*. (Laforge 2007) Dieses Prinzip sagt aus, dass möglichst wenig redundanter Code während der Entwicklung entstehen soll. Dies wird unter anderem durch das dynamische Scaffolding unterstützt, da beim Ändern der Attribute der zugrunde liegenden Domainklasse keine weiteren Anpassungen notwendig sind. Die Controller und die Präsentationsschicht werden automatisch angeglichen. So wird die Entwicklung und im späteren Verlauf auch die Wartung stark vereinfacht.

2.2.1.2 Konvention über Konfiguration

Ein anderes wichtiges Prinzip von Grails ist *Convention over Configuration (CoC).* (Laforge 2007) Dies bedeutet, dass bevor, meist relativ aufwändige, Konfigurationen über z.B. *XML* oder *YAML* notwendig sind, generelle Konventionen diese Arbeit obsolet werden lassen. Im Fall von Grails, wird dieses Prinzip schon beim Erstellen einer Anwendung über die *create-app* Anweisung anschaulich, da eine komplexe, konventionelle Ordnerhierarchie angelegt wird.

Als Grails-Entwickler ist es dadurch möglich, wenn keine stark spezifischen Anforderungen existieren, fast vollständig ohne aufwändige Konfigurationen zu arbeiten. Grails verwendet viele quelloffene Technologien und jederzeit können, sollte es nötig sein, individuelle Detailänderungen manuell vorgenommen werden. Dies ist im Regelfall jedoch unnötig.

Grails verwendet eine strenge Namenskonvention, so dass die entsprechenden Teile aus der Domain oder die Controllermethoden aus der Präsentationsschicht über diese Namensgebung gefunden und aufgerufen werden können. Dadurch müssen keine Mappings definiert werden. Unter Anderem ist es beispielsweise möglich individuelle Texte zur Internationalisierung im entsprechenden Verzeichnis (i18n) zu hinterlegen. Beim Einhalten der Namenskonvention, verwendet Grails automatisch, gemäß der regionalen Einstellung des Klienten, die entsprechenden Texte. (Rocher & Brown 2009)

Werden Klassen über die korrekten Kommandozeilen-Befehle (*create-domain-class, create-controller...*) erstellt, so werden diese Konventionen automatisch eingehalten. Dabei wird auch der Case-Sensitive berücksichtigt und die Benutzereingabe gegebenenfalls angepasst. Die MVC Trennung erfolgt z.B. über die Ordner *domain*, für Domainklassen, *controllers*, für die dazugehörigen Controller und *views*, der die Templates der Präsentationsschicht enthält. (Rocher & Brown 2009) Abbildung 2 stellt die Ordnerhierarchie dar.

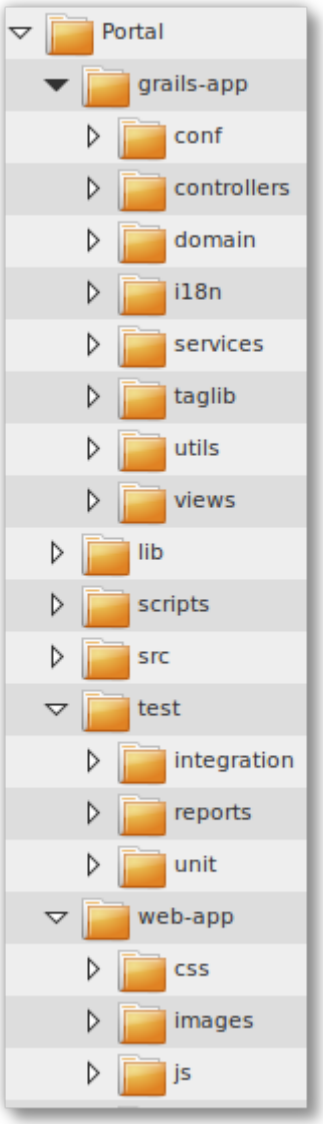


Abbildung 2: Grails Verzeichnisbaum

2.2.1.3 Erweiterbarkeit

Sollte die Basisfunktionalität von Grails nicht ausreichen, so kann der Funktionsumfang, für spezielle Anforderungen, leicht über eine Plugin-Schnittstelle erweitert werden. Alle registrierten Plugins können über den Befehl *grails install-plugin* installiert und unmittelbar eingesetzt werden. (Judd et al. 2008)

Eine Vielzahl Plugins kommen aus dem Ajax Bereich. So z.B. das „*searchable*“ Plugin, dass durch einfache Referenz in der Domainklasse die Ansicht im Darstellungsbereich des Scaffolding automatisch um eine optisch ansprechende Suche mit Vorschlagsbox erweitert.

Grails implementiert sämtliche großen Ajax-Bibliotheken, wie z.B. *scriptaculous, prototype* und *Yahoo UI.* Um Eine oder Mehrere zu verwenden, muss das entsprechende Tag, mit dem Verweis auf die Bibliothek, in dem Template gesetzt werden. Daraufhin stehen dann eine Vielzahl von grails-tags, wie z.B. zum Aufruf von Remote-Funktionen oder das übermitteln eines Formulars über AJAX, zur Verfügung. Es können aber auch alle Funktionen der Bibliotheken über eigene JavaScript-Codings verwendet werden. (Judd et al. 2008)

2.2.2 Komponenten

Ein weiteres wichtiges Prinzip von Grails lautet: *Don't reinvent the wheel*. Grails bietet nur wenig an neuen, ausgefeilten Technologien, dagegen wird auf bereits bewährte und bekannte Komponenten aufgebaut. Die einzelnen Bestandteile der Grails Plattform werden im Folgenden beschrieben.

2.2.2.1 Übersicht

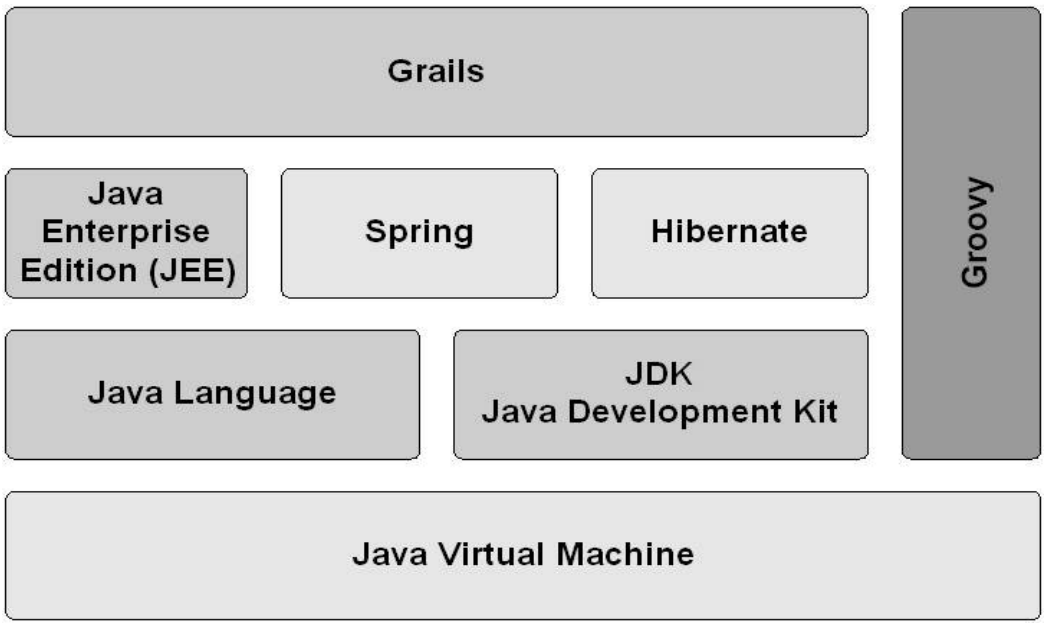


Abbildung 3: Grails Struktur (Rocher & Brown 2009)

In Abbildung drei sind die wichtigsten Komponenten und die Struktur der Grails Plattform veranschaulicht. Da viele bekannte Technologien verwendet werden, sollte der Einstieg erfahrenen Web-Entwicklern relativ leicht fallen. Andererseits ist Grails durch die vielen umfangreichen Bestandteile eine komplette Entwicklungsplattform und es werden im Normalfall keine weiteren Third-Party Werkzeuge benötigt. Darüber hinaus werden alle Komponenten von verschiedenen Firmen, bzw. Gemeinschaften, weiterentwickelt, so dass die Grails-Entwicklung relativ schnell voranschreitet.

Grails bietet also im Vergleich zu vielen anderen Frameworks nicht nur ergänzende Funktionalitäten, die die technischen Möglichkeiten der Web Entwicklung erweitern, bzw. deren Komplexität verringern, sondern bietet durch die verwendeten Technologien die Möglichkeit ohne Einrichtung eines Servers oder Datenbank auf hohem Niveau, z.B. durch Einführung verschiedener Laufzeitumgebungen und Testfällen, ohne großen vorhergehenden Aufwand, zu entwickeln.

Obwohl so viele verschiedene, unabhängig voneinander entwickelte Komponenten in Grails zusammenarbeiten, ist es während eines Entwicklungsprozesses nicht nötig mit XML oder komplexen Annotationen zu arbeiten. Grails ersetzt aufwändige Konfiguration durch Konvention. Es ist aber auch jederzeit möglich darauf zurückgreifen, wenn spezielle Anforderungen, wie z.B. individuelle Tabellennamen in der Datenbank, referentielle Integrität oder einfach nur besondere URL-Mappings nötig sind. (Rocher & Brown 2009)

2.2.2.2 GORM / Hibernate

GORM steht für „*Grails Object Relational Mapping*“. Im Allgemeinen bildet das objektrelationale Mapping eine Technik Objekte, bzw. Domainklassen einer objektorientierten Programmiersprache persistent relational auf einer Datenbank abzubilden. (Richardson 2008) Mit der Verwendung von Hibernate ist es unnötig sich als Entwickler um Details wie Tabellenstrukturen und -schlüssel oder SQL zu kümmern. Grails verwendet dafür, wie die *Java Persistence API (JPA)*, die in der *Java EE* Entwicklung verbreitet Verwendung findet, *Hibernate* als Grundlage.

Hibernate übernimmt vollkommen automatisiert das Abbilden der Domainklassen in Grails. Verwendet wird für den Datenbankzugriff die Java Database Connectivity (*JDBC)*. Alle Beziehungen zwischen Relationen(1:1,1:N,N:M) werden übernommen. (Rocher & Brown 2009)

Auch Vererbung von Klassen wird bei der Verwendung von *GORM* mit abgebildet. Dazu gibt es entweder die Möglichkeit die Objekte mit allen Attributen der Ober- und Unterklasse in einer Tabelle, oder, wenn dies zu viel Overhead in der Tabelle verursacht, eine Tabelle für die Oberklasse und eine für die Unterklasse anzulegen. Die Beziehung wird dann auch bei Abfragen berücksichtigt. Die Möglichkeit Unterklassen auf zwei verknüpfte Tabellen auszudehnen, wird momentan noch nicht unterstützt. (Judd et al. 2008)

Eine besondere Funktionalität, die *GORM* im Vergleich zu anderen *ORM* Frameworks bietet, ist die Berücksichtigung von referentieller Integrität. Obwohl GORM theoretisch ohne Konfiguration direkt lauffähig ist, ist es trotzdem auch möglich sämtliche Einstellungen, wie beim direkten Arbeiten mit der Datenbank, manuell vorzunehmen. So unter anderem auch Einstellungen für referentielle Integrität, die Verwendung anderer Primärschlüssel als ausschließlich generierte Identifikationsnummern, oder individuelle Bezeichnungen für Tabellen und Felder. Hierfür wird keine YAML- oder XML Konfigurationsdatei verwendet, sondern natives Groovy. (Crause 2009)

Zusätzlich können auch Constraints und Validierungen direkt in der Domainklasse in einer Closure hinterlegt werden.

Bsp. Für einen Constraint:

```
01 static constraints = { name(blank:false) }
```

GORM bildet aber nicht nur Objekte relational ab, sondern übernimmt auch alle SQL Operationen die während der Laufzeit beim Zugriff auf Objekte nötig werden. Werden also beispielsweise Objektattribute geändert, so wird die Tabelle automatisch mit den neuen Informationen aktualisiert, bzw. erweitert. *GORM* verwendet hierfür den SQL Dialekt der jeweiligen zugrunde liegenden

Datenbank. (Judd et al. 2008) Somit ist die Applikation unabhängig von der darunter liegenden Persistenzschicht, solange diese in der Konfiguration definiert wurde.

Darüber hinaus können aber auch komplexere Abfragen an die Datenbank mittels der *Hibernate Criteria-API* oder der *Hibernate Query Language*, SQL ähnlichen Sprachen, die sich auf die Hibernate-Objekte beziehen, durchgeführt werden. (Rocher & Brown 2009) So stehen eine Reihe dynamischer Methoden für Hibernate Objekte zur Verfügung um komplexere Abfragen möglichst zu vermeiden. Zum Beispiel kann zum Auffinden einer Liste von Objekten, oder eines einzelnen Objekts die Methode *findByAttribute()* verwendet werden. „Attribute“ kann durch jedes beliebige Attribut des Objektes ersetzt werden. Als Parameter wird der entsprechende Suchwert übergeben. Weitere Möglichkeiten sind *findWhere(), findAllWhere()* etc. (Judd et al. 2008)

Grundsätzlich kann konfiguriert werden, ob beim Laden einer Seite Daten in *eager-*, bzw. *lazy-loading* geladen werden. Standardmäßig werden alle normalen Tabellen direkt vollständig in den Hauptspeicher geladen (*eager loading*), Tabellen, die über Schlüsselattribute mit anderen Tabellen verknüpft sind, werden erst bei Bedarf nachgeladen (*lazy-loading)*. (Laforge 2007; Richardson 2008)

2.2.2.3 Spring

Spring ist als *Inversion of Control(IoC)* und *Wrapper Framework* dafür zuständig den Zusammenhalt und die Kommunikation zwischen allen Elementen des Grails Frameworks zu ermöglichen und zu sichern. Beispielsweise benötigt Hibernate, um Objekte zu persistieren, Zugriff auf die Datenbank, oder die *GSPs* müssen Zugriff auf den entsprechenden Controller, und dieser auf das zugrunde liegende Datenmodell haben. Spring übernimmt einen Großteil der Konfiguration durch Konvention, so dass es als Grails-Entwickler nur bei wirklich speziellen Anforderungen nötig wird sich um das Mapping, bzw. die Konfiguration von Schnittstellenabhängigkeiten zu kümmern.

Das *Inversion of Control* Prinzip sagt aus, dass die Kontrolle verschoben wird. Dadurch ist es in Grails nicht mehr zwingend notwendig eine Datenbank zu erstellen, die dann von Hibernate interpretiert wird, sondern Hibernate erstellt die Datenbank nach den vorhandenen Domainklassen. So wird die Kontrolle also auf die Applikationsebene verlegt. (Spring Source o. J.)

2.2.2.4 Datenbank

Grails unterstützt fast alle auf dem Markt verfügbaren Datenbanken. In der Standardkonfiguration von Grails kommt jedoch zur Entwicklung eine HSQL Datenbank zum Einsatz. Ziel hiervon ist es, ohne viel Aufwand, direkt mit der Entwicklung beginnen zu können. So muss keine extra Datenbank angelegt werden, sondern sie wird direkt bei jedem Start der Anwendung neu erstellt und in den Hauptspeicher geladen. D.h. bei jedem Neustart beginnt die Applikation immer mit einer neuen Datenbank mit leeren Tabellen, die auf die Domainklassen abgestimmt sind. Über die entsprechende Konfigurationsdatei (*Bootstrap.groovy*) ist es aber auch möglich neue Objekte zu erzeugen, die dann gleich zu Beginn persistiert werden, um mit einem gewissen Testdatenbestand arbeiten zu können.

Bestehen spezielle Anforderungen und soll daher eine andere Datenbank angebunden werden, so kann die entsprechende Konfiguration in der *DataSource.groovy* vorgenommen werden. (Judd et al. 2008)

2.2.2.5 Jetty

Jetty ist ein HTTP Server, HTTP Client, *javax.servlet-container* und ein Applikationsserver. (Eclipse Foundation o. J.) Damit handelt es sich bei Jetty um keine abgegrenzte Komponente des Grails Frameworks, sondern umfasst als Container und Server die ganze Plattform.

Durch Jetty wird der Grails Entwicklungsprozess deutlich vereinfacht. Der Serverdienst wird bei der Installation von Grails mit eingerichtet. Mit dem Befehl *grails run-app* wird die Applikation automatisch in einen Container geladen und

der Server lokal gestartet. So ist es unnötig sich vor der Entwicklung um die Einrichtung eines Servers zu kümmern. Auch eine Datenbankverbindung wird durch die integrierte *HSQLDB* unnötig. Der Vorteil dabei ist, dass Plattform-übergreifend, nur mit der Installation von Grails, Applikationen direkt, wie bei einer nicht web-basierten Entwicklungsumgebung, entwickelt und ausgeführt werden kann.

2.2.2.6 SiteMesh

SiteMesh ist ein weiteres Java basiertes Web-Framework, dass in die Grails Plattform integriert wurde. Es übernimmt hauptsächlich Komponenten für Webseiten-Layout und Dekoration um ein konsistentes Layout zu integrieren. Es unterbricht HTTP Anfragen, die vom Webserver verarbeitet werden um die Seite zu parsen und ggf. dynamische Informationen gemäß dem *Decorator Pattern* einzufügen. Darüber hinaus kann es, ähnlich der überholten HTML-Frameset Technologie, *GSPs*, bzw. HTML-Dateien wie durch *Server-Side-Includes* als Panels in Seiten einbinden. Es ist also möglich Templates zu definieren, die dann ähnlich wie in unterschiedlichen *Frames* gerendert werden, ohne dabei die bekannten Nachteile von *Frames* zu haben. (Judd et al. 2008; Rocher & Brown 2009)

2.2.3 Weitere Funktionen

Grails bietet eine große Anzahl an Funktionen und Erweiterungen. In diesem Teil werden die folgenden drei näher erläutert: Die verschiedenen Systemlandschaften, die Grails zur Unterstützung des Entwicklungsprozesses integriert, die Möglichkeit Testfälle zu erstellen und die Einrichtung von Tag-Bibliotheken.

2.2.3.1 Umgebungen

Immer mehr Firmen sind dazu gezwungen 5-System Landschaften einzurichten um die S*ox-Compliance* zu erfüllen, da sie an der US Börse gelistet sind. So geht das allgemeine Bewusstsein, auch im Small- und Medium-Enterprises Bereich, immer mehr in Richtung von sauberen Trennungen für verschiedene Umgebungen. Unabhängig davon bringt die Aufteilung in getrennten Entwicklungs-, Debugging- und Produktiv-Landschaften viele Vorteile mit sich. Sie ermöglichen einen deutlich effektiveren und zeitsparenderen Entwicklungsprozess, dessen Endprodukt einfacher zu warten ist.
Grails unterscheidet drei verschiedene Systemlandschaften: (Judd et al. 2008)

1. *Development*
2. *Debug*
3. *Productive*

Die verschiedenen Umgebungen haben eine Vielzahl verschiedener Konfigurationsmöglichkeiten. So z.B. das Anzeigen von Dumps, die von Fehlern in der Applikation verursacht werden, oder die Einstellungen für die zu verwendende Datenbank. Unter anderem wird ermöglicht, dass alle drei Landschaften auf unterschiedliche Datenbanken, bzw. Tabellen zugreifen. Auch dass technische Fehlerbeschreibungen in einer Produktivumgebung, die nicht von Entwicklern genutzt wird, fehlplatziert sind, aber in der Entwicklungs-, bzw. Debug-Umgebung unverzichtbar sind, ist klar und kann deshalb separat für jede Landschaft konfiguriert werden.

Wie bereits erwähnt, verwendet Grails zu Entwicklungszwecken eine in den Arbeitsspeicher geladene HSQL Datenbank. In einer Produktivumgebung z.B. enthält eine Datenbank Persistenzdaten und sollte keinesfalls nach Beenden der Applikation gelöscht werden.

Die Entwicklungsumgebung unterstützt *hot-reloading*. (Rocher & Brown 2009) Das bedeutet, dass Änderungen am Quellcode, während die Applikation läuft, sofort übernommen werden. Beispielsweise veranlassen Änderungen an dem Datenmodell sogar einen direkten Neustart der Applikation. Dies könnte in

einer Debug-Umgebung unerwünschte, bzw. in einer produktiven Umgebung sogar katastrophale Folgen nach sich ziehen.

2.2.3.2 Tests

Werden die Domain- und Controllerklassen mit den dafür vorgesehenen Konsolenbefehlen erstellt, so werden automatisch Testklassen dazu erzeugt. Diese erfüllen, zusammen mit den eigentlichen Klassen, auch die Grails Namenskonvention. Dadurch können ohne viel Aufwand und unter Verwendung der Groovy- und *JUnit*, bzw. *GUnit*-, Syntax sowohl Unit- als auch Integration-Tests erstellt werden. Unit-Tests haben dabei keinen Zugriff auf applikationsweite Informationen (z.B. Datenbanken oder andere Klassen). Aus diesem Grund bietet es sich für Domainklassen an nur Integration Tests anzufertigen, da sie direkt mit der Datenbankschicht verzahnt sind. (Judd et al. 2008)

Tests werden mit dem *grails test-app* Befehl durchgeführt. Dabei wird ein detaillierter, HTML-basierter Report erstellt, durch den der Zustand der Anwendung und Fehlerquellen einfach identifizieren werden können.

2.2.3.3 Tag-Bibliotheken

Wie bereits erläutert, können in der Präsentationsschicht zusätzlich zu dem Server-seitigem Code der *Groovy-Template-Engine*, Grails eigene Tags verwendet werden. Dies wird durch die entsprechende Tag-Bibliotheken, sogenannten *Taglibs,* unterstützt. Das grundsätzliche Verwenden der *Taglib*-Technologie, ermöglicht das einfache Definieren eigener Tags, bzw. das Einbinden von Third-Party-*Taglibs*. Diese können einfach im entsprechenden Verzeichnis der Ordnerhierarchie angelegt und darauf im Template verwiesen werden. (Rocher 2006)

Eigene, nur einmal benötigte Tags können auch über eine Closure innerhalb eines Controllers definiert werden. (Judd et al. 2008)

Bsp. Für ein individuelles Tag auf Basis einer Closure:

```
01 def meinTagName = { attrs -> rückgabewert }
02 <g:meinTagName meinParam="wert" />
```

3 Projekt

Im folgenden soll die Funktionalität von Grails anhand einer Beispielanwendung näher veranschaulicht werden. Dieses Projekt entspricht einem einfach gehaltenen Studentenportal. Das Portal enthält ein Vorlesungsverzeichnis, eine Anmeldungsmaske für Studenten und die Möglichkeit Vorlesungen, für die sich bereits Studenten angemeldet haben, einzusehen.

Für die spätere Evaluation und Gegenüberstellung mit anderen Anwendungen wird kein Session-Handler und keine Benutzerverwaltung verwendet. Der Benutzer muss so seine Stammdaten jedes mal bei der Anmeldung für eine Prüfung erneut eingeben.

3.1 Aufbau / Domänenmodell

Die Applikation stellt, gemäß der Spezifikation, drei verschiedene Ansichten für die spezifizierten Funktionalitäten bereit:

1. Übersicht über alle Vorlesungen
2. Anmeldung für Vorlesungen
3. Übersicht über bereits angemeldete Vorlesungen

In der Anwendung wird vollständig auf die Scaffolding Funktionalität von Grails verzichtet. Die Intention dahinter ist die Entwicklung in der Präsentationsschicht von Grund auf mit Groovy-Templates als auch unter dem Einsatz von Grails Tags durchzuführen und nicht nur bereits automatisch generierten Templates anzupassen, um aus dem kleinen Projekt möglichst viel Erfahrung in der praktischen Anwendung von Grails zu gewinnen.

Für dieses Projekt werden drei Domänen verwendet, die in drei Datenbanktabellen persistiert sind.

Es existiert eine 1:n Beziehung von Fachbereich zu Vorlesung und eine n:m Beziehung von Vorlesung zu Student. Da alle Daten bei der Anmeldung immer erneut eingegeben werden müssen, existiert keine separate Tabelle für die Stammdaten der Studenten. Ohne eine dazugehörige Vorlesung kann ein Student nicht existieren. Es werden 3 Tabellen für die Applikation verwendet: *Vorlesung*, *Angemeldet* und *Fachrichtung*.

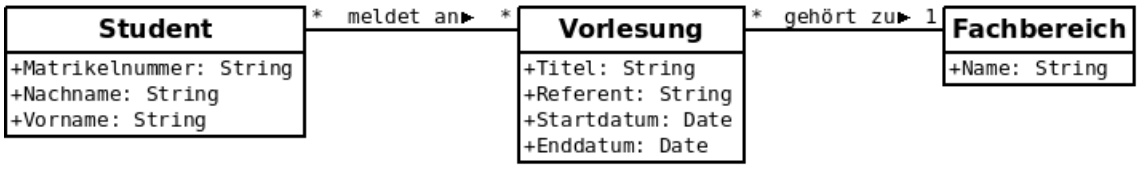


Abbildung 4: Domänenmodell

3.2 Funktionen & Beispiele

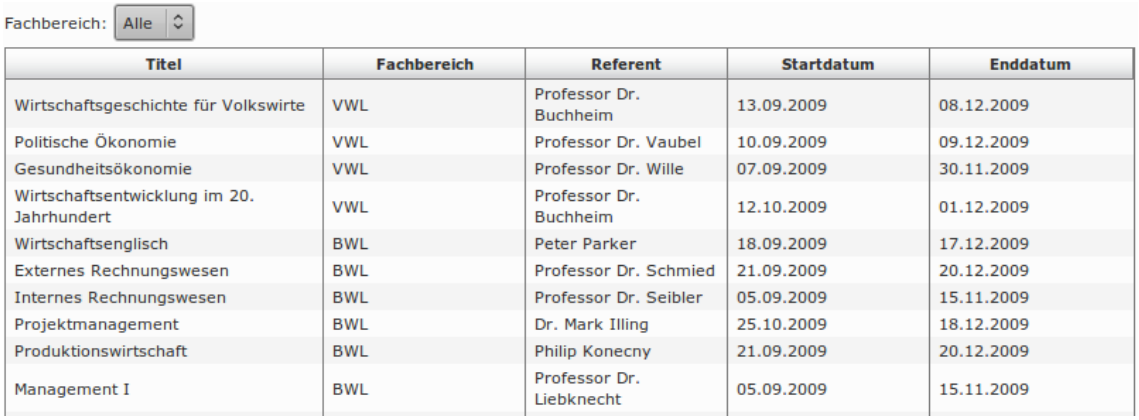
Um bei der Anmeldung oder bei der Suche möglichst ohne eine zusätzliche HTTP-Anfrage auszukommen, damit die Performance zu erhöhen, aber trotzdem die Dropdowns, bzw. Tabellen dynamisch zu generieren, werden in der Applikation Ajax Aufrufe, unter Verwendung des *criptaculous*-Frameworks gestartet.

Bsp. Für einen Ajax Aufruf:

```
01 onchange="${remoteFunction(action:'getVorlesungen',
update:'updateMe',params:'\'fachbereich=\'+ this.value')}"
```

In diesem Aufruf wird der Inhalt der Closure im Controller *Vorlesungen* aufgerufen, und wenn darin nicht umgeleitet wird, die *getVorlesungen.gsp* gerendert, und in dem div *updateMe*, unter Verwendung des ausgewählten Fachbereichs als Parameter angezeigt. Nach dem selben Prinzip funktioniert im Bereich „angemeldete Vorlesungen“ die Suche nach Matrikelnummer und die Selektion der anzuzeigenden Vorlesungen im Bereich „Anmeldung“ nach

Auswahl des Fachbereichs. Nach jeder Eingabe wird die Tabelle unter Berücksichtigung der Eingabe als Parameter asynchron nachgeladen.


Fachbereich: Alle

Titel	Fachbereich	Referent	Startdatum	Enddatum
Wirtschaftsgeschichte für Volkswirte	VWL	Professor Dr. Buchheim	13.09.2009	08.12.2009
Politische Ökonomie	VWL	Professor Dr. Vaubel	10.09.2009	09.12.2009
Gesundheitsökonomie	VWL	Professor Dr. Wille	07.09.2009	30.11.2009
Wirtschaftsentwicklung im 20. Jahrhundert	VWL	Professor Dr. Buchheim	12.10.2009	01.12.2009
Wirtschaftsenglisch	BWL	Peter Parker	18.09.2009	17.12.2009
Externes Rechnungswesen	BWL	Professor Dr. Schmied	21.09.2009	20.12.2009
Internes Rechnungswesen	BWL	Professor Dr. Seibler	05.09.2009	15.11.2009
Projektmanagement	BWL	Dr. Mark Illing	25.10.2009	18.12.2009
Produktionswirtschaft	BWL	Philip Konecny	21.09.2009	20.12.2009
Management I	BWL	Professor Dr. Liebknecht	05.09.2009	15.11.2009

Abbildung 5: Vorlesungsverzeichnis

Im Datenmodell haben die beiden Objekte *Vorlesung* und *Angemeldet* eine 1:n Beziehung. Dies wird in der Klasse *Angemeldet* über eine einfache Objektvariable des Typs Vorlesung, die den Verweis auf die entsprechende Vorlesung enthält, realisiert. Die Klasse *Vorlesung* enthält einen speziellen Constraint, der auf die Beziehung verweist. In dem folgenden Beispiel ist ein Auszug aus der Klasse *Angemeldet*, aus der die Objektvariable und verschiedene Arten von Constraints erkenntlich werden.

Auszug aus der Domainklasse *Angemeldet*

```
01 Vorlesung vorlesung
02 static constraints = {
03   matrikelnummer(blank:false,matches:/\d\d\d\d\d\d\d/)
04   nachname(blank:false, minSize:2, validator:{val, obj->
isInteger(val)})
05   vorname(blank:false, minSize:2, validator:{val, obj->
isInteger(val)})
06   vorlesung(blank:false)
07 }
```

Abbildung 6: Anmelden für eine Vorlesung

Wie bereits erläutert, verweist die Objektvariable auf die Verlinkung der beiden Tabellen in der Persistenzebene. Der erste Constraint basiert auf einem regulären Ausdruck und garantiert, dass die Matrikelnummer eine 7-stellige Zahl ist. Für die Attribute Nachname und Vorname existiert einerseits ein individueller Validator, der sicherstellt, dass keine numerischen Werte eingegeben wurden und andererseits die Überprüfung, dass mehr als zwei Zeichen eingegeben werden und genauso wie bei der Vorlesung, dass diese Attribute nicht leer sind. Bevor die Daten persistiert werden, wird in dem Controller ein Objekt erzeugt und validiert. Ist dieser Vorgang nicht erfolgreich so kann es zu einer ähnlichen Ausgabe wie in Abbildung 7 kommen.

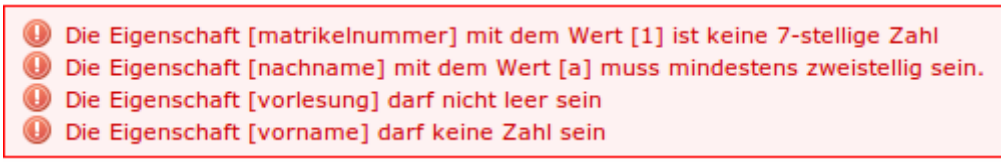


Abbildung 7: Feedback nach fehlerhafter Eingabe

Das folgende Beispiel enthält einen Auszug aus der Validierung für die Anmeldung im entsprechenden Controller:

Auszug aus der *Validierung für Anmeldungen*

```
01 angemeldet = new Angemeldet()
02 angemeldet.properties = params
03 if(angemeldet.validate()){
04   Vorlesung.findByTitel(it).addToAngemeldete(angemeldet)
05   angemeldet.save()
06 }
```

4 Analyse

In den vorhergehenden Kapiteln wurden die Grundkonzepte und Funktionsweisen von Groovy und Grails benannt und erläutert. In diesem Kapitel folgt nun, auf Grund von Benchmark Ergebnissen, eine Analyse und Vergleich mit anderen Frameworks.

4.1 Allgemein

Grails soll, der Intention nach, ein einfach zu erlernendes aber trotzdem überaus mächtiges und umfangreiches Web-Framework mit vollständiger Entwicklungsplattform zu sein. Damit ist die Plattform nicht nur für Privatanwender, sondern auch für Firmen im Enterprise Bereich interessant. So hat Grails auf jeden Fall, wie bereits erläutert, einige positive Aspekte im Bereich der Erlernbarkeit, Modifizierbarkeit und Vollständigkeit.

Im Kontrast dazu stehen die noch relativ dünn gestreuten Ergebnisse, bei der Suche nach spezifischen Problemlösungen. Grails ist eine noch junge Technologie, die zwar sehr gut dokumentiert ist, jedoch noch nicht so weite Verbreitung findet wie vergleichbare Frameworks.

Wie Grails im Hinblick auf Performance, bzw. Verwendbarkeit im Vergleich zu seinen Konkurrenten dasteht, wird im folgenden, beim Vergleich mit zwei

Platzhirschen unter den Web Frameworks, *J2EE* und *Ruby on Rails,* näher ausgeführt.. Kriterien sind dabei sowohl Performance und Funktionsumfang, als auch Konfigurations- und Entwicklungstechniken, die erst bei einer Detailanalyse offenkundig werden.

4.2 Vergleich mit J2EE

Bei der Arbeit mit dem bewährten, schwergewichtigen *Web-Framework* der *J2EE* ist im Vergleich zu der Grails-Plattform eine Vielzahl von Unterschieden erkennbar. So ist J2EE bereits als robustes Enterprise-fähiges Framework bestätigt, hat aber seine Nachteile darin, nicht Suchmaschinen-freundlich zu sein, da viele Seiten nur über JavaScript erreichbar sind.

Das ORM erfolgt in J2EE mittels JPA, welches keine andere Technologie implementiert, als das auch in Grails verwendete Hibernate. Im Gegensatz zu Grails verzichtet J2EE auf die automatische Konfiguration durch Konventionen. Zu dem Entwicklungsprozess gehört also auch die Verwendung von Annotationen und XML. Es gibt eine Vielzahl an Schichten und Konfigurationsdateien, die während des Entwicklungsprozesses viel Zeit in der Konfiguration, anstatt in der Entwicklung, fordern. Dies führt dazu, dass einfache Anwendungen mit *CRUD* Funktionalität bei J2EE, mangels *Scaffolding*, um ein Vielfaches aufwändiger zu erstellen sind als mit Grails. Dabei muss auch bei Änderungen im Datenmodell oft an vielen Stellen nachgearbeitet und angepasst werden. Dies widerspricht im Grails dem strengen *DRY* Prinzip.

Auch wenn sich Grails als Plattform zur agilen Softwareentwicklung, versteht so muss hier auch ein Kontrapunkt erwähnt werden. Controller- und View-Elemente lassen sich einfach über die *Scaffolding* Funktionalität erstellen. Im Gegensatz zu dem langwierigen Entwicklungsprozess von *J2EE* ist dies ein Vorteil. Allerdings, sollten die generierten Elemente vom Entwickler für besondere Spezifikationen manuell angepasst worden sein (*statisches Scaffolding*), so ist, bei Änderung des Datenmodells, eine neue Generierung

unmöglich. Die folgende Anpassung der Applikation befindet sich damit auf dem selbem Niveau wie das der *J2EE* Technologie.

4.3 Vergleich mit Ruby on Rails

Ruby on Rails baut im Gegensatz zu *J2EE* und *Grails* nicht direkt auf der *JVM* auf. Zwar bietet es mittels *JRuby* die Möglichkeit, auf Java Servern eingerichtet zu werden, hat dabei aber einen hohen Performance-Verlust. (Rocher 2007) Wie das Framework im Vergleich zu Grails und *J2EE* besteht, zeigt der nachfolgende Benchmark.

Für das Mapping und Persistieren von objektrelationalen Daten in Datenbanktabellen verwendet Rails *ActiveRecord*. Die Funktionalität ist ähnlich zu der von *GORM*, allerdings fehlt hier das *IoC*. *ActiveRecord* generiert zu einer bestehenden Datenbankdefinition die entsprechende Benutzerschnittstelle, Getter-, Setter- und Validierungs-Methoden. Dabei existieren allerdings erhebliche Probleme mit bereits erstellten und gefüllten Tabellen. Darüber hinaus sollte nach der Generierung durch *ActiveRecord* keine Änderung des zugrunde liegenden Datenmodells mehr erfolgen. Hinzu kommt, dass die Rails Technologie nur mit *GUID*s als Primärschlüssel umgehen kann. Daher ist Ruby on Rails unter Verwendung von *ActiveRecord* wohl ungeeignet für Enterprise Applikationen. Die Konfiguration erfolgt über *YAML*. (Crause 2009)

4.4 Performance-Analyse

Die Werte dieser Performance-Analyse basieren auf einer Präsentation von *SUN Microsystems* auf der Java One 2007.

Diese Ergebnisse sind mit Vorsicht zu genießen, da SUN Microsystems verständlicherweise auf ihr Produkt, die *J2EE* Technologie, fokussiert ist.

Ferner ist zu beachten, dass die Analyseergebnisse auf 2007 datieren. Mittlerweile sind die verwendeten Versionen von Ruby, Java und Grails veraltet. Allerdings ist dies das aktuellste, veröffentlichte Benchmark.

Die Analyse wurde mit der Benchmark Technologie *FABAN* durchgeführt. Alle drei Frameworks setzen dabei auf einem *Glassfish*-Server auf. Sowohl *FABAN* als auch *Glassfish* sind *SUN* eigene Technologien.

Für diesen Benchmark wurden bei einer einheitlichen Applikation die Durchlaufzeiten bei *CRUD*-Operationen in Relation zur Anzahl der Benutzer gesetzt. Abbildung 4 veranschaulicht die Reaktionszeiten der Operationen, in Abhängigkeit der Benutzeranzahl. Abbildung 5 zeigt die dazugehörige Prozessorauslastung des Applikationsservers.

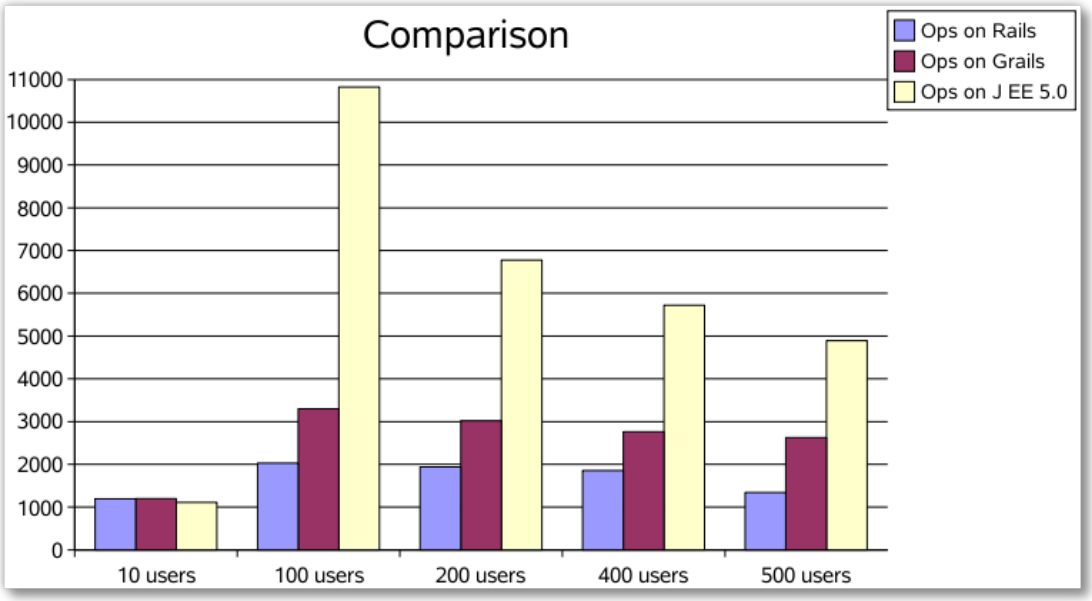


Abbildung 8: Java One Benchmark Operationen per Nutzer (Daly & Cooke 2007)

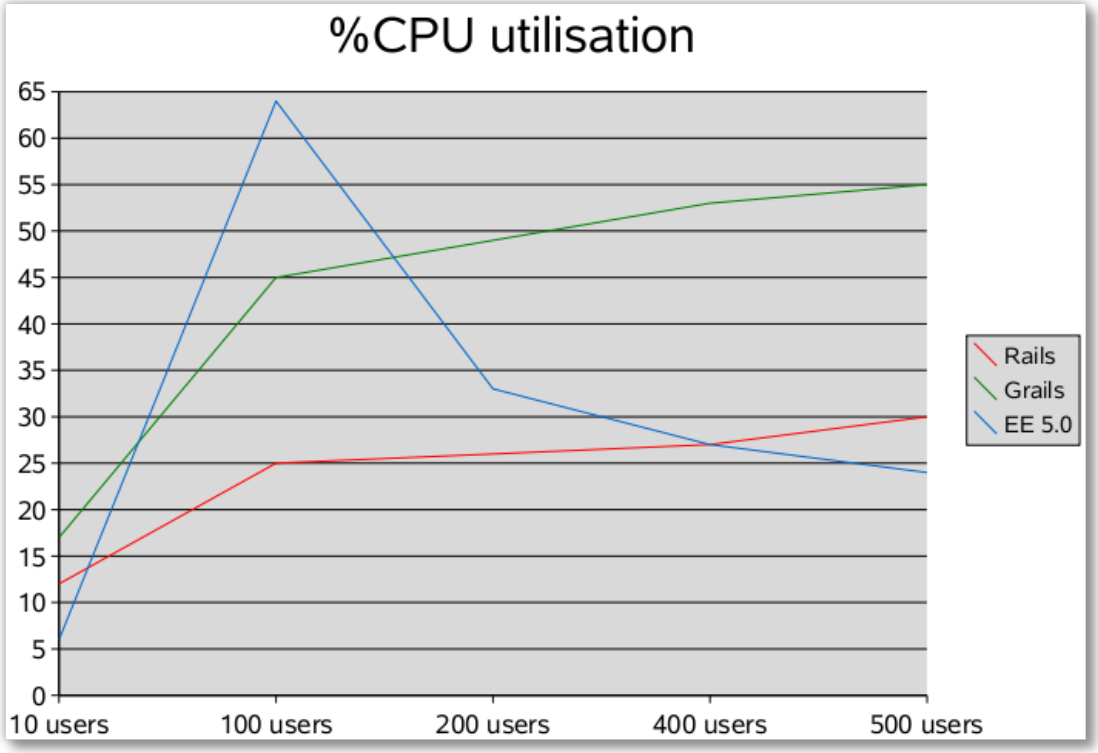


Abbildung 9: Java One Benchmark CPU Auslastung (Daly & Cooke 2007)

Aus den Ergebnissen wird deutlich, dass *J2EE* vor allem bei durchschnittlicher, bzw. großer Nutzerzahl deutlich höhere Durchlaufzeiten hat als Rails und Grails. Im Gegensatz dazu verläuft die CPU-Auslastung bis 100 Nutzer exponentiell steigend, bei einer größer werden Anzahl jedoch sinkend. Grails hat im Vergleich zu Rails eine etwas höhere Durchlaufzeit und CPU-Auslastung. (Daly & Cooke 2007)

Graeme Rocher, Leiter des Grails Projektes, hat selbst einen Benchmark zwischen Grails und Rails durchgeführt. Diese Ergebnisse beziehen sich noch auf Grails in der Version 0.4.2, als dieses noch auf einem Tomcat anstelle von Jetty lief. Auch diese Ergebnisse sind aus neutraler Position vorsichtig zu bewerten, da Graeme nach eigenen Angaben weder ein Rails noch ein Benchmark Experte ist. Diese Analyseergebnisse stehen im Kontrast zu dem Performance-Test von Daly und Cooke: Grails ist in allen Bereichen hinsichtlich

der Durchlaufzeit und CPU-Auslastung deutlich performanter als Rails. (Rocher 2007)

5 Fazit

Grails ist ein umfangreiches Web-Framework, dass aus einer Vielzahl unterschiedlicher Komponenten besteht und auf vielen verschiedenen Technologien aufbaut. Dies ermöglicht Grails einerseits eine hohe Skalierbarkeit, aber erschwert auch Entwicklern über alle Teilbereiche und Komponenten hinweg das volle Potential von Grails ausschöpfen zu können. Jedoch ist die Lernkurve für Neueinsteiger, die keine Enterprise-Applikationen mit hohem Customizing Aufwand erstellen wollen, besonders aus dem Java-Bereich sehr gering. Darüber hinaus ist es bereits mit nur wenigen Kenntnissen über Groovy oder Hibernate möglich, mächtige und performante Anwendungen erstellen.

Somit ist Grails im Vergleich zu *J2EE* sowohl ein sehr leichtgewichtiges, aber auch umfangreiches und skalierbares Framework. Ob es dauerhaft bei großen Anwendungen gegen das schwergewichtige und stark Schichten-orientierte *J2EE* bestehen kann, bleibt offen. Es ist auf jeden Fall noch eine sehr junge Technologie, von der wohl noch einige Bewegung seitens der Entwickler zu erwarten ist. Schon jetzt gibt die, in Kürze zu veröffentlichende, Version 1.2 einen Einblick in die Zukunft von Grails. Auch in diesem Release wird wieder an Komplexität abgebaut, aber gleichzeitig der Funktionsumfang erweitert, indem die Plattform auf *Spring 3* und einem *Apache Tomcat* aufgebaut wird, individuelle URL-Mappings noch einfacher zu erstellen sind und dynamische *FindBy*-Methoden für *Boolean*-Werte eingeführt werden. (Ghani 2009)

6 Literaturverzeichnis

comScore, I., 2009. Global Internet Audience Surpasses 1 Billion Visitors, According to comScore - comScore, Inc. Erreichbar unter: http://www.comscore.com/Press_Events/Press_Releases/2009/1/Global_Internet_Audience_1_Billion [Zugegriffen November 18, 2009].

Crause, N., 2009. Grails First Impressions – GORM vs. ActiveRecord - Nathan Crause. Erreichbar unter: http://nathan.crause.name/entries/programming/grails-first-impressions-%E2%80%93-gorm-vs-activerecord [Zugegriffen November 2, 2009].

Daly, T. & Cooke, D., 2007. Comparing the Developer Experience of JavaTM EE 5.0, Ruby on Rails, and Grails.

Eclipse Foundation, About the Jetty Project. Erreichbar unter: http://www.eclipse.org/jetty/about.php [Zugegriffen November 2, 2009].

Ighani, M., 2009. Grails 1.2: Spring 3, Uri Rewriting und mehr. Erreichbar unter: http://it-republik.de/zonen/portale/psecom,id,101,online,2587.html [Zugegriffen November 23, 2009].

Judd, C., Nusairat, J. & Shingler, J., 2008. *Beginning Groovy and Grails*,

König, D. & Rocher, G., 2006. Groovy and Grails Seminar.

Laforge, G., 2007. Grails: Spring+Hibernate development reinvented.

Richardson, C., 2008. ORM in Dynamic Languages.

Rocher, G., 2006. Graeme Rocher's Blog: Grails: Tag Libraries & The Power of Closures. Erreichbar unter: http://graemerocher.blogspot.com/2006/04/grails-tag-libraries-power-of-closures.html [Zugegriffen Oktober 29, 2009].

Rocher, G., 2007. Grails vs Rails Benchmark - Grails - Codehaus. Erreichbar unter:

http://docs.codehaus.org/display/GRAILS/Grails+vs+Rails+Benchmark [Zugegriffen November 5, 2009].

Rocher, G. & Brown, J., 2009. *The definitive Guide to Grails* 2. Aufl., Apress.

Spring Source, Enterprise Java Development Tools | SpringSource. Erreichbar unter: http://www.springsource.com/products/enterprise [Zugegriffen Oktober 28, 2009].

Tilly, M., 2004. Die Skriptsprache Groovy.

www.javabeat.net, Closures in Groovy. Erreichbar unter: http://www.javabeat.net/articles/62-closures-in-groovy-1.html [Zugegriffen Oktober 26, 2009].

7 Abbildungsverzeichnis

Abbildung 1: Scaffolding....8
Abbildung 2: Grails Verzeichnisbaum....10
Abbildung 3: Grails Struktur....11
Abbildung 4: Domänenmodell....20
Abbildung 5: Vorlesungsverzeichnis....21
Abbildung 6: Anmelden für eine Vorlesung....22
Abbildung 7: Feedback nach fehlerhafter Eingabe....22
Abbildung 8: Java One Benchmark Operationen per Nutzer....26
Abbildung 9: Java One Benchmark CPU Auslastung....27