



Seminar Webframeworks 2007
Hasso-Plattner-Institut

Betreuer: Christian Willems, Katrin Wolf

Von
Juliane Kummerlöwe & Manuel Blechschmidt
18.07.07

Inhaltsverzeichnis

Einleitung.....	3
Teil I GRAILS - web application development made easy.....	4
Konzepte sind wichtig.....	4
Make it as easy as possible but not easier.....	4
Don't repeat yourself.....	4
Create Update Delete.....	5
Model View Controller.....	5
Code by Convention.....	5
Dynamic rocks.....	6
Dependency Injection alias Inversion of Control.....	6
Don't reinvent the wheel.....	7
Teil II GRAILS - Evaluation.....	8
Architektur und Komponenten.....	8
Schichten einer Grails-Applikation.....	8
Komponenten.....	8
Java VM.....	8
Groovy.....	9
Spring Inversion of Control.....	9
Hibernate als Persistence Layer, GORM.....	9
Hypersonic SQL.....	9
Jetty, Servlets und JSP.....	10
Sitemesh fürs Layout.....	10
Installation, Konfiguration und Deployment.....	10
Installation.....	10
Eclipse-Integration.....	10
Alternative Datenbank einbinden.....	10
Plugins.....	11
Deployment.....	11
Benutzte Programmier- bzw. Scriptsprache.....	12
Dynamische Methoden.....	12
Template Engine und TagLibs.....	13
Integrierte TagLibrary.....	13
Eigene Tags.....	14
User handling/ Authorisation.....	14
Support/ Dokumentation.....	14
AJAX.....	14
Validation und Errorhandling.....	15
Validation von Eingaben.....	15
Fehlerbehandlung.....	15
Unit-Testen.....	15
Funktionales Testen.....	15
Skalierbarkeit und Performance.....	16
Anwendungsmodi dev, prod, test.....	18
Teil III GRAILS - Beispiel.....	19
Quellen.....	30

Einleitung

Im Seminar "Webframeworks" haben wir uns mit dem ziemlich jungen Webframework Grails beschäftigt, welches wir in dieser Ausarbeitung theoretisch und praktisch vorstellen. Im ersten Teil führen wir die grundlegenden Konzepte ein. Im zweiten Teil gehen wir genauer auf die zu evaluierenden Kriterien ein. Die praktische Umsetzung soll in Teil 3 in Form eines Tutorials gezeigt werden. Dort wird schrittweise eine lauffähige Beispielanwendung anhand vieler Screenshots und Listings erklärt.

Teil I GRAILS - web application development made easy

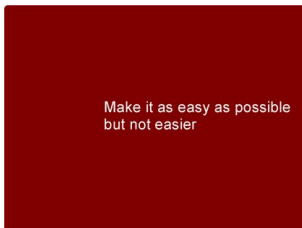
Grails ist eine Erfindung der Groovy-Liebhaber Guillaume LaForge, Graeme Rocher und Steven Devijver. Die Schwierigkeiten mit Java-Webframeworks im Hinterkopf wurden sie von *Ruby on Rails* inspiriert, ein eigenes Webframework-Projekt zu beginnen. Der erste Name des Frameworks war *Groovy on Rails*. Die wichtigste von Rails übernommene Idee ist *Code by Convention*. Mitwirkende Programmierer sind neben den Gründern: Marc Palmer, Dierk Koenig, Sven Haiges und viele weitere. Die aktuelle Version ist die Version 0.5. Die erste Version 0.1 erschien im März 2006. Die Grails-Roadmap sieht eine Veröffentlichung der Version 1.0 für Oktober 2007 vor.



Konzepte sind wichtig

Webframeworks sollen einen flexiblen und mächtigen Rahmen für die Entwicklung von Webapplikationen bieten. Die Effizienz solcher Frameworks steht und fällt mit der Ausgestaltung der Konzepte, die schnelle Modifizierbarkeit, Erweiterbarkeit, Skalierbarkeit, Portierbarkeit usw. ermöglichen sollen. Grails verfolgt dabei ähnliche Ansätze wie andere Webframeworks, z.B. *Coding by Convention* und

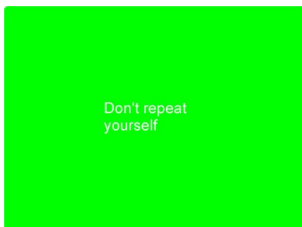
Don't repeat yourself, unterstützt aber auch neue Herangehensweisen, z.B. die dynamische Groovy-Konsole, die den Programmierer bei seiner Arbeit unterstützt. Im Folgenden werden die grundlegenden Konzepte und Ideen vorgestellt, die bei der Entwicklung von Grails im Vordergrund stehen.



Make it as easy as possible but not easier

Die Installation von Grails ist denkbar einfach. Das Installationspaket von Grails bündelt bereits die wichtigsten Komponenten. Hauptvoraussetzung ist eine JDK-Installation. Jede Java-IDE kann zur Programmierung benutzt werden oder einfach ein Texteditor, bei Bedarf mit Groovy-Syntaxhighlighting. In der Standardkonfiguration

benutzt Grails eine integrierte HSQL-Datenbank. Nach dem Entpacken muss nur eine Umgebungsvariable gesetzt werden. Die Installation und das Anlegen einer Applikation dauern insgesamt nur wenige Sekunden. Die mitgelieferten Komponenten (Server, Datenbank usw.) brauchen im einfachsten Fall nicht konfiguriert werden. Eine mit `grails create-app` neu erstellte Webapplikation ist sofort lauffähig. Die Installation von Plugins ist ebenso einfach.



Don't repeat yourself

Das MVC-Pattern verlangt, dass Applikationen in die Komponenten Model, View und Controller zerlegt werden. Das DRY-Prinzip allerdings verlangt, keinen redundanten Code zu produzieren, jedenfalls nicht eigenhändig. Grails nimmt uns die Produktion der zu einer Klasse gehörenden Komponenten ab. Dieses wichtige Feature von Webframeworks nennt man *Scaffolding*. Hat man eine Domänenklasse

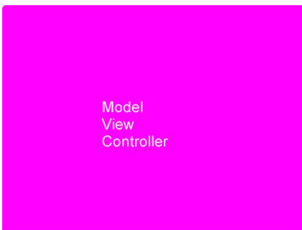
mit den gewünschten Eigenschaften angelegt, ruft man `grails generate-all` auf und die entsprechenden Controller und Views werden generiert. Eine leere Domänenklasse erstellt Grails über den Befehl `grails create-domain-class`. Instanzen von Domänenklassen werden automatisch persistiert.



Create Update Delete

Eine Grundfunktionalität einer datenbankbasierten Applikation ist die Verwaltung von Entitäten, also das Erstellen, Modifizieren und Löschen von Objekten, die als Datenbankeinträge vorliegen. Diese Methoden werden unter *CRUD* - Create Update Delete zusammengefasst und von Grails beim Scaffolding jedem Controller automatisch hinzugefügt. Die CRUD-Methoden verwirklichen ohne

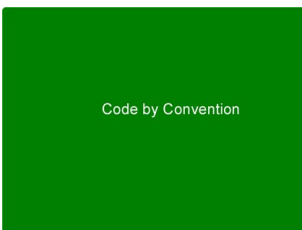
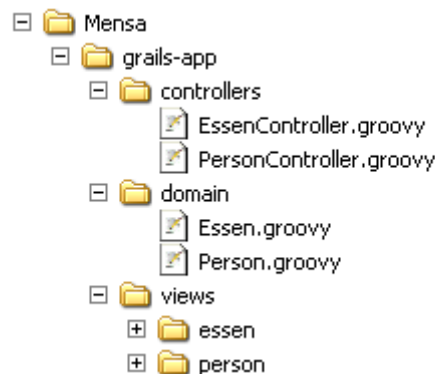
weitere Angaben die Persistierung der Daten bezogen auf die spezifizierte Datenbank. Grails verfolgt einen domänenzentrierten Ansatz. Es werden also die Domänenklassen in der Datenbank abgebildet und nicht umgekehrt.



Model View Controller

MVC ist ein schon lange bekanntes Pattern, welches auf Webanwendungen angewendet bedeutet, dass die Ausgabe von HTML-Code (View) von der Verarbeitung der Serveranfragen (Controller) und den Eigenschaften und Relationen der Domänenklassen (Model) getrennt behandelt wird. Grails verwirklicht dies durch die zu jeder Domänenklasse durch Namenskonvention zugeordneten

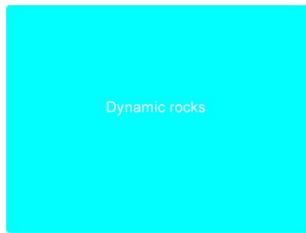
Controllerklassen und gsp-Views. Die Abbildung zeigt zwei Domänenklassen eines Beispielprojektes mit den dazugehörigen Controllern und Views.



Code by Convention

Die in der Abbildung gezeigte Ordnerstruktur ist obligatorisch. Auch die Namensvergabe ist vorgeschrieben. Die in der Unterordnern von *views* gelegenen *.gsp*-Dateien tragen immer den Namen der Funktion, von der sie aufgerufen werden. Jeder Controller, der eine Domänenklasse verwaltet, enthält beispielsweise eine Methode *list*, welche die Instanzen auflistet. Die Anzeigeform der *list*-Methode

wird in *list.gsp* beschrieben. Die Datei muss dann im Ordner der Klasse im Ordner *views* abgelegt werden. Diese Herangehensweise heißt *Konvention statt Konfiguration* oder *Code by Convention*. Das erklärte Ziel der Grails-Entwickler ist es, das Schreiben von XML-Konfigurationsdateien überflüssig zu machen, da es eine häufige Fehlerquelle und daher ein Hindernis für effiziente Programmierung darstellt.



Dynamic rocks

Zum Grailspaket gehört neben den Kommandozeilenbefehlen ein weiteres Feature in zwei Geschmacksrichtungen: die *Groovy-Console* und die *Groovy-Shell*. Sie werden mit `grails console` bzw. `grails shell` gestartet. Die Konsole besitzt eine graphische Oberfläche, während die Shell textbasiert ist. In beiden kann ein Block aus Abfragen und Befehlen eingegeben und sofort ausgeführt werden. Auf diese Weise kann man schnell und einfach Objekte abfragen (z.B. mit den eingebauten Finder-Methoden), Eigenschaften der Anwendung prüfen oder Groovycode ausprobieren, ohne dafür Controllerklassen erstellen zu müssen.

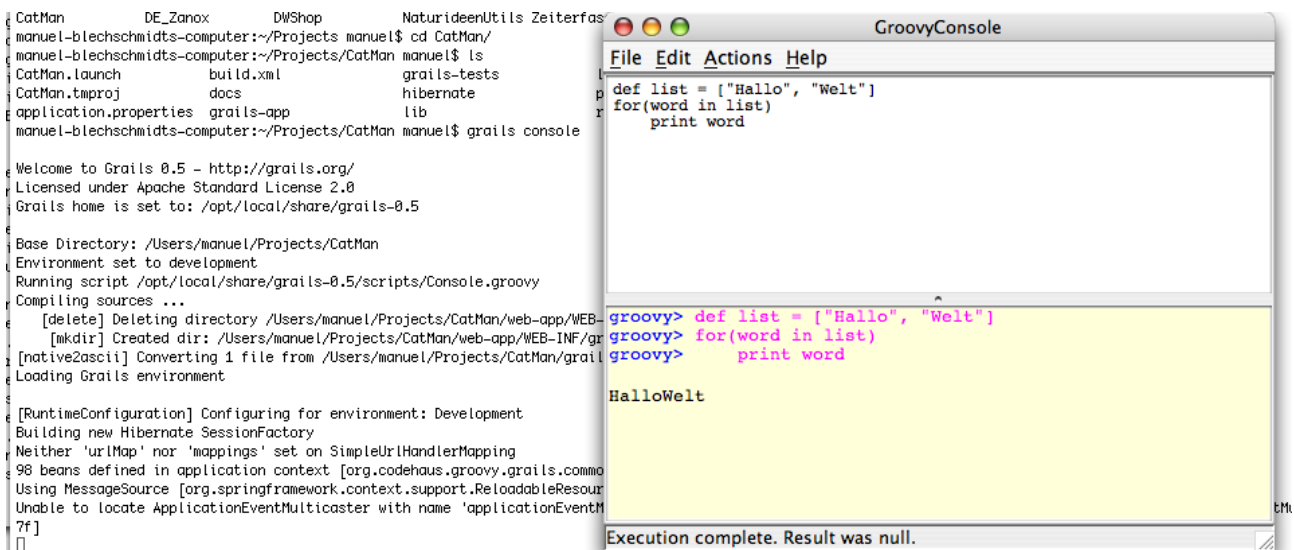
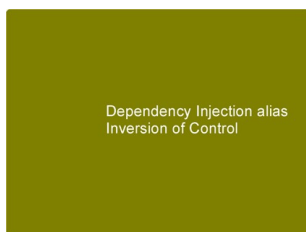


Abbildung 1: Groovy Console in Action

Ein weiteres Feature sind die eingebauten *dynamischen Methoden*, die in keiner Klasse explizit eingebunden werden müssen, sondern einfach zur Verfügung stehen. Sie werden zur Compile-Zeit hinzugefügt. Am eindrucksvollsten sind vielleicht die bereits erwähnten Finder-Methoden, z.B. `findBy*()` oder `findWhere()`.



Dependency Injection alias Inversion of Control

Um die Businesslogik zu implementieren werden *Services* verwendet, die von den Controllerklassen aufgerufen werden. Die *Dependency Injection* geschieht einfach über die Deklaration eines Services in einer Controllerklasse. Der Aufruf von Service-Methoden wird in den Methoden des Controllers gekapselt. Hierbei müssen wieder Namenskonventionen eingehalten werden. Über die Namensvergabe kann das Framework die Zuordnung des Services (also der Service-Klasse) und die Zuordnung der für die Ausgabe vorgesehenen View selbstständig vornehmen.



Don't reinvent the wheel

Den Entwicklern von Grails war es wichtig, bestehende gute Techniken weiterzuverwenden und so basiert Grails auf bekannten und in der Praxis bewährten Java-Komponenten, z.B. Hibernate und Spring. Der Aufbau auf bestehende Frameworks bzw. Technologien stellt nicht nur eine grundlegende Stabilität des Frameworks sicher sondern ermöglicht es zudem, problemlos fertige Programmteile, z.B. in Form von Java-

Bibliotheken, einzubinden und zu nutzen. Der dritte Vorteil ist der zumindest für Java-Programmierer vereinfachte Einstieg in Grails.

Teil II GRAILS - Evaluation

Architektur und Komponenten Schichten einer Grails-Applikation

In der 3-Schichten-Architektur aus Präsentations-, Anwendungs- und Persistenzschicht sind die Artefakte einer Grails-Applikation wie folgt verteilt:

- Webpräsentations-Schicht
 - Views: `views/**/*.gsp`
 - Controller: `controllers/*.Controller.groovy`
- Business-Logik-Schicht
 - Domänenklassen: `domain/*.groovy`
 - Services: `services/*.Service.groovy`
- Persistenz-Schicht
 - CRUD-Methoden
 - `*DataSource.groovy`

Komponenten

Grails konnte nur durch seine leistungsfähigen Komponenten, die schon seit Jahren entwickelt werden, in so kurzer Zeit ein so stabiles und mächtiges Framework werden. Einige dieser Komponenten und Technologien werden nun vorgestellt

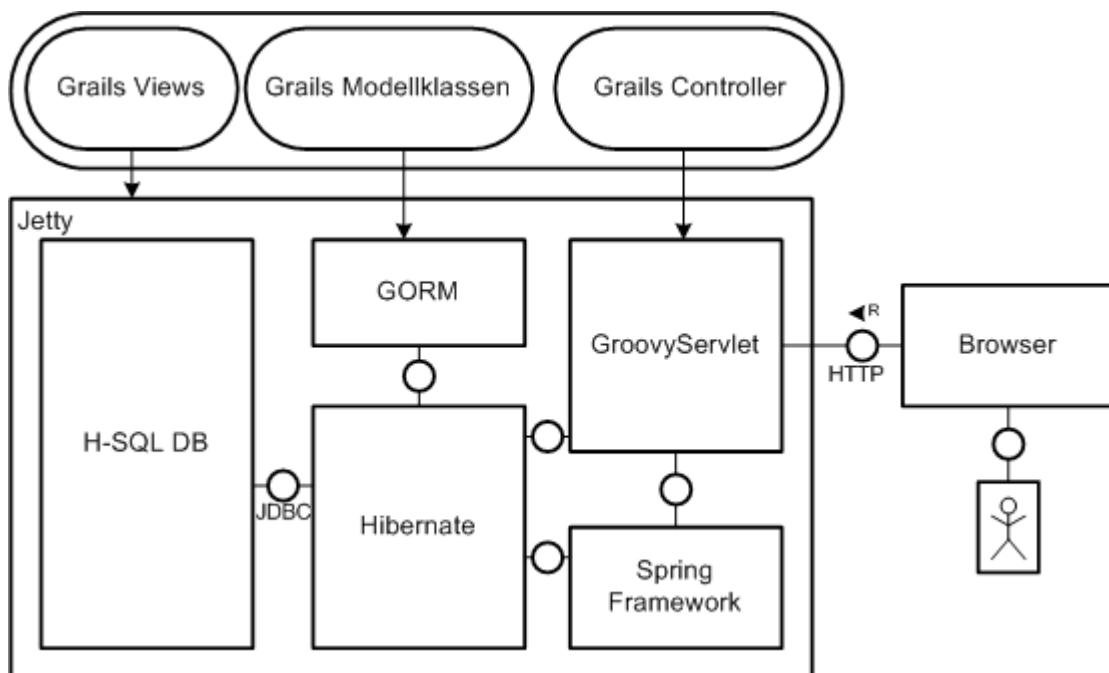


Abbildung 2: Strukturaufbau einer laufenden Grailsanwendung

Java VM

Die Java Virtual Machine ist eine von der Firma Sun entwickelte Plattform zum systemunabhängigen Betreiben von Anwendungen. Die momentan aktuellste Version ist Java 6. Die Technologie zeichnet sich durch qualitativ hochwertige APIs, eine saubere Programmierung und professionelles Vorgehen aus.

Auf dieser soliden Plattform bauen alle Grails Komponenten auf.

Groovy

Groovy ist eine dynamische Programmiersprache vergleichbar mit Python oder Ruby für die Java Plattform. Grails beinhaltet die aktuelle Version BETA 1.1.

Mit Groovy hat man gegenüber zu kompilierenden Programmiersprachen wie Java oder C++ den Vorteil nutzen, dass man seine Klassen zur Laufzeit verändern kann und somit Methoden oder Attribute hinzufügen kann. Da man Groovy auch in einem Scripting- oder Consolenmodus betreiben kann, ist es sogar möglich, das lästige Kompilieren während der Entwicklungszeit zu sparen.

Ein großer Teil der Mächtigkeit von Grails entspringt den Sprachfeatures von Groovy. So werden zum Beispiel Konzepte aus der funktionalen Programmierung mit anonymen Funktionen, den sogenannten Closures, umgesetzt. Es existiert mit GPath eine native Navigation für XML-Dokumente, die an XPath erinnert.

Die moderne Syntax erleichtert die Entwicklung an vielen Stellen, in dem man zum Beispiel mit einem „?“ an einem Ausdruck eine NullPointerException unterdrücken kann oder dass der letzte Ausdruck einer Funktion automatisch als Rückgabewert dient.

Diese großen Vorteile werden allerdings mit kleinen Performanzeinbußen bezahlt.

Spring Inversion of Control

Das Spring Framework, welches von Interface 21 entwickelt wird, ist zu Zeiten von J2EE 1.4 entstanden. Es implementiert den MVC Ansatz für Webframeworks. Im Gegensatz zu der Enterprise Plattform basierte es auf einem leichtgewichtigen Inversion of Control Konzept und wird aus Komponenten aufgebaut. Diese Komponenten werden durch eine XML Datei konfiguriert und können somit unabhängig voneinander entwickelt und eingesetzt werden. Jeder benutzt nur den Teil des Frameworkes, den er wirklich benötigt.

Die Abbildung der Klassen auf die richtigen SpringBeans übernimmt Grails für den Entwickler.

Hibernate als Persistence Layer, GORM

Hibernate ist ein O/R Mapper für Java und wird von Gavin King, der mittlerweile Angestellter der Firma JBoss ist, als OpenSource-Konkurrenz zu anderen Produkten wie Oracle TopLink entwickelt.

Das Projekt ist bereits seit mehreren Jahren aktiv und hatte einen großen Einfluss auf die kommerzielle Software-Entwicklung. Zum Beispiel hat sich die neue Java Persistenz-API unter anderem von diesem Tool inspirieren lassen.

Die aktuelle Version ist 3.0 und auch diese wird mit Grails geliefert. Das Mapping, welches normalerweise über XML Dateien und Annotationen am Quellcode geschehen muss, macht Grails ganz automatisch.

Hypersonic SQL

Hypersonic SQL ist eine in 100% pure Java implementierte OpenSource-Datenbank, welche bei Grails als Testdatenbank mitgeliefert wird. Sie wird über einen JDBC-Driver angesprochen und läuft während der Entwicklung nur im Hauptspeicher.

Jetty, Servlets und JSP

Als Implementierung der Servlet- und JSP-API wird der Servlet Container Jetty, welcher von der Firma Mortbay unter einer freien Lizenz zur Verfügung gestellt wird, mitgeliefert.

Dieser kleine Entwicklungsserver kann sofort nach dem Erzeugen der Anwendung gestartet werden und garantiert somit einen schnellen Erfolg.

Sitemesh fürs Layout

Sitemesh ist die Implementierung des Decorator Patterns für HTML Websites der OpenSymphony Plattform. Mit ihr ist es möglich, einheitliche Layouts auf Basis eines Maintemplates zu erstellen.

Dieses System bildet die Basis für alle Grails Views.

Installation, Konfiguration und Deployment

Installation

Da Grails auf Java basiert, muss Java SDK in einer Version ab 1.4 installiert sein. Unter <http://grails.codehaus.org/Download> gibt's die aktuellen Grails-Downloads. Nach dem Entpacken in ein beliebiges Verzeichnis muss nur noch eine Umgebungsvariable gesetzt werden. Grails ist mit diesen wenigen Schritten installiert und nutzbar. Zum Test kann auf der Kommandozeile `grails help` eingegeben werden. Es sollte eine Liste mit Grails-Kommandos erscheinen. Nun kann mit `grails create-app` eine neue Webapplikation angelegt werden. Die Installation und das Anlegen einer Applikation dauern insgesamt nur wenige Sekunden - dabei werden alle Komponenten (Server, Datenbank usw.) mitgeliefert und müssen im einfachsten Fall nicht konfiguriert werden. Auch die neu erstellte Anwendung ist sofort lauffähig.

Eclipse-Integration

Grails generiert automatisch .projekt-Files und .classpath-Files für Eclipse. Um ein bestehendes Projekt zu importieren, muss `Import -> Existing projekt into Workspace` gewählt werden. Die Eclipse-Classpath-Variable `GRAILS_HOME` (zu erreichen über `Windows -> Preferences... -> Java -> Build path -> Classpath Variables -> New...`) muss auf den Grails-Installationspfad gesetzt werden. Die Run- und Debug-Umgebung von Eclipse kann genutzt werden, die Anwendung läuft dann eingebettet in Eclipse. Um die Grails-Kommandozeilen-Befehle aufzurufen, muss Grails als *External Tool* definiert werden. Dafür `Run > External Tools > External Tools...` öffnen und Grails als Name und `grails.bat` als Ziel eingeben. Folgende Werte setzen: `Working directory` auf `${project_loc}`, `Arguments` auf `${string_prompt}`, im Register `Refresh` die Option `project containing the selected resource` auswählen und im Register `Common Grails` im Feld `Display in favorites menu` aktivieren. Nun kann die Grails-Kommandozeile als externes Tool gestartet werden. Die einzelnen Befehle werden als Parameter übergeben.

Das Groovy-Plugin (Syntax-Highlighting usw.) hat seine Eclipse-Update-Site unter <http://dist.codehaus.org/groovy/distributions/update>.

Alternative Datenbank einbinden

Es können alle Datenbanken verwendet werden, für die ein JDBC-Driver vorliegt.

In der Standardeinstellung von Grails wird Hypersonic als in-memory-Datenbank eingesetzt, welche über die Datei `grails-app/conf/DevelopmentDataSource.groovy` konfiguriert werden kann. In dieser Datei findet sich nach der Standardinstallation folgende Klasse:

```
// conf/DevelopmentDataSource.groovy
class DevelopmentDataSource {
    boolean pooling = true
    String dbCreate = "create-drop" // one of 'create', 'create-drop','update'
    String url = "jdbc:hsqldb:mem:devDB"
    String driverClassName = "org.hsqldb.jdbcDriver"
    String username = "sa"
    String password = ""
}
```

Diese Klasse wird zum Beispiel wie folgt modifiziert, um eine MySQL-Datenbank mit dem Namen `grails_db` einzubinden.

```
// conf/DevelopmentDataSource.groovy
class DevelopmentDataSource {
    boolean pooling = true
    String dbCreate = "update" // one of 'create', 'create-drop','update'
    String url = "jdbc:mysql://localhost:3306/grails_db?autoReconnect=true"
    String driverClassName = "com.mysql.jdbc.Driver"
    String username = "root"
    String password = "password"
}
```

Für den Produktion- und Testmodus werden die Datenbankeinstellungen analog in den Dateien `ProductionDataSource.groovy` und `TestDataSource.groovy` vorgenommen. Der JDBC-Driver muss im Verzeichnis `lib` im Projektordner vorhanden sein. Für die MySQL-DB kann z.B. aus dem Ordner `samples/simple-cms/lib` im Grails-Installationsverzeichnis die Bibliothek `mysql-connector-java-3.1.10-bin.jar` kopiert werden.

Plugins

Die Installation von Plugins ist ebenso einfach wie die Installation von Grails selbst. Dazu wird der Befehl `grails install-plugin` mit dem Pluginnamen als Parameter eingegeben. Es kann alleinstehend der Pluginname (+ ggf. Versionsnummer) angegeben werden. Dann verbindet sich Grails mit dem Grails-Plugin-Repository. Als Parameter kann auch ein Pfad angegeben werden, wenn man das Plugin lokal gespeichert hat.

Auch die Entwicklung und Distribution eigener Plugins wird von Grails unterstützt. Grails stellt Befehle zum Anlegen von Plugin-Projekten, zum Release selbiger usw. bereit. Derzeit stehen im Grails-Repository knapp 20 Plugins für verschiedene Anwendungen zur Verfügung.

Deployment

Mit dem Befehl `grails war` packt man die ganze Applikation in eine `.war`-Datei, welche man dann auf jeden Java-Server exportieren kann.

Benutzte Programmier- bzw. Scriptsprache

Die in Grails verwendete Scriptsprache ist Groovy. Aus Groovy-Programmen wird der gleiche Bytecode kompiliert, wie aus Java-Programmen. Groovy wird dementsprechend auch auf einer Java-VM ausgeführt. Daher ist es auch problemlos möglich in Grails-Applikationen Java-Bibliotheken zu verwenden.

Ein herausragendes Sprachkonzept von Groovy ist die *Closure*. Eine Closure ist eine anonyme Methode bzw. Funktion in geschweiften Klammern geschrieben. Zum Beispiel `def closureC = {x -> x*x}` kann einfach benutzt werden durch `def quadrat = closureC(y)`. In einer Closure kann ein `return`-Wert definiert sein, der dann der Rückgabewert der Closure wird. Falls kein `return`-Wert definiert wird, wird der Wert der letzten Anweisung der Closure zurückgegeben.

Dynamische Methoden

Ein sehr praktisches Feature von Grails sind die eingebauten dynamischen Methoden und Eigenschaften, die in keiner Klasse explizit eingebunden werden müssen, sondern einfach zur Verfügung stehen. Sie werden zur Compile-Zeit hinzugefügt. Die Grails-Entwickler beschreiben den Vorteil dieses Feature mit den Worten *"having far less 'code noise' in your applications"*. Neben den bereits erwähnten Finder-Methoden gibt es unter anderen folgende Methoden und Eigenschaften. Es gibt sie speziell für Domänen- oder Controllerklassen oder GSP-Tags, einige können auch in allen Klassen/Views verwendet werden.

Eigenschaften: `log`, `flash`, `grailsApplication`, `grailsAttributes`, `params`, `request`, `response`, `servletContext`, `session`, `actionUri`, `actionName`, `controllerUri`, `controllerName`, `out`, `constraints`, `properties`

Methoden: `encodeAs*`, `decode*`, `bindData`, `chain`, `render`, `redirect`, `getViewUri`, `getTemplateUri`, `add`, `add*`, `delete`, `discard`, `hasErrors`, `ident`, `merge`, `refresh`, `save`, `validate`, `count`, `countBy*`, `createCriteria`, `executeQuery`, `executeUpdate`, `exists`, `find`, `findAll`, `findBy*`, `findAllBy*`, `findWhere`, `findAllWhere`, `get`, `getAll`, `list`, `listOrderBy*`, `withCriteria`, `withTransaction`

Einige Methoden sind mit einem Sternchen versehen. Diese Methoden bilden die Sahnehäubchen unter den dynamischen Methoden, denn das Sternchen steht für domänenspezifische Varianten der Methode. Diese sind intuitiv zu bilden und zu nutzen, was einen erheblichen Flow beim Programmieren erwirkt. Die Familie der find-Methoden soll dies hier beispielhaft illustrieren.

Die Finder-Methoden haben gemeinsam, dass sie die vorhandenen Instanzen abfragen und dabei die Ergebnismenge eingrenzen. Es werden hier nur einige Parameterkombinationen gegeben, die Methoden sind für weitere Kombinationen überladen:

1 <http://www.grails.org/Dynamic+Methods+Reference>

```

class something{
    String name
    Integer nummer
}
// find-Methoden ohne Sternchen
something.find(query) // kann eine HQL-Query auswerten,
                    // gibt eine passende Instanz zurück
something.findAll() // gibt alle passenden Instanzen zurück
something.findAll(new something(name:"Heinz"))
                    // Suchen anhand eines Beispiels,
                    // gibt alle Heinze zurück
something.findWhere(name:"Horst", nummer:66)
                    // Anfrage mit Attributen ausgedrückt

// find-Methoden mit Sternchen: domänenbezogene Methodennamen
something.findByName("Heinz") // Suche mit Domänenklassenattributen, der
                             // erste heinz wird zurückgegeben
something.findAllByName("Heinz") // ebenso, alle Heinze
something.findAllByNummerBetween(17,20)
                             // alle Instanzen mit nummern
                             // zwischen 17 und 20
something.findByNummerGreaterThanEquals(18)
                             // die erste Instanz mit nummer ≥ 18
something.findAllByNameNotEqual("Horst") // alle außer Horst
something.findAllByNameLike("Schmidt")
                             // alle, die Schmidt oder ähnlich heißen
something.findAllByNameAndNummer("Manuel",21)
                             // alle, die mit dem Namen Manuel
                             // und der Nummer 20
something.findAllByNameOrNummer("Manuel",1000)
                             // alle, die mit dem Namen Manuel
                             // oder der Nummer 1000

```

Template Engine und TagLibs

Die Templates für die Seitendarstellung (Views) werden bei Grails mit `.gsp`-Dateien (*Groovy Server Pages*) erstellt. Diese sind nach dem Vorbild der Java Server Pages definiert, mit dem Unterschied, dass sie nicht in Java, sondern in Groovy programmiert werden und dass die Tags anders eingefasst werden. In Grails ist eine breite Palette an GSP-Tag eingebaut. Programmierer können aber nicht nur GSP-Tags verwenden, sondern auch JSP-Tags und ihre Views als `.jsp`-Dateien abspeichern. Die TagLibrary bietet ebenso eine JSP-Sektion an, welche aber weniger umfangreich ist, als der GSP-Bereich.

Integrierte TagLibrary

Die integrierte TagLibrary von Grails enthält, wie bereits erwähnt, zwei Sektionen: eine GSP- und eine JSP-Bibliothek. Wir gehen hier nur auf die umfangreichere GSP-Bibliothek ein. Zur Verfügung stehen unter anderem logische Tags, iterative Tags, Rendering-Tags, Formular-Tags usw.

Tags werden durch den Aufruf von `<g:tagName parameter=wert>...</g:tagName>` bzw. bei alleinstehenden Tags über `<g:tagName parameter=wert />` eingebunden. Als Wert für die Parameter können Ausdrücke der Form `"${a==b}"` verwendet werden.

Ausschnitt aus der GSP-Referenz:

- Logische Tags: `if`, `else`, `elseif`
- Iterative Tags: `while`, `each`, `collect`, `findAll`, `grep`

- Zuweisungstags: `def`, `set`
- Link-Tags: `link`, `createLink`, `createLinkTo`
- Ajax-Tags: `remoteField`, `remoteFunction`, `remoteLink`, `formRemote`, `submitToRemote`
- Formular-Tags: `actionSubmit`, `actionSubmitImage`, `checkBox`, `currencySelect`, `form`, `hiddenField`, `datePicker`, `select`, `localeSelect`, `textField`, `textArea`, `timeZoneSelect`
- UI-Tags: `richTextEditor`
- Ausgabe- und Layout-Tags: `render`, `renderErrors`, `layoutHead`, `layoutBody`, `layoutTitle`, `pageProperty`, `paginate`, `sortableColumn`
- Validierungstags: `eachError`, `hasErrors`, `message`

Eigene Tags

Neben der integrierten Tag-Bibliothek ist es möglich, eine eigene TagLibrary zu erstellen, welche in `grails-app/taglib/*TagLib.groovy` gespeichert wird. Man kann Tags auch während der Laufzeit hinzufügen und sofort benutzen. Tags werden in Form einer Closure mit `def meinTagName = { attrs -> rückgabewert }` in der TagLib definiert und mit `<g:meinTagName meinParam="wert" />` in der `.gsp`-Datei eingebunden.

Die Grails-Community stellt ebenfalls neue Tags auf der Tag-Contribution-Site zur freien Verfügung.

User handling/ Authorisation

Eine Benutzerverwaltung ist in der Standardinstallation nicht automatisch vorhanden. Das heißt, sie muss selbst gebaut werden, ggf. unter Verwendung vorhandener Plugins oder Java-Bibliotheken. Unter den Tutorials der Grails-Community befinden sich mehrere Anleitungen, dies zu tun. Ein Plugin, welches in Frage kommt ist z.B. das AcegiSecurity-Plugin.

Support/ Dokumentation

Es gibt einige englischsprachige Bücher zu Grails und einige mehr zu Groovy. Ein erstes deutschsprachiges Buch ist für Herbst 2007 geplant.

Sehr umfangreich ist dagegen das Informationsangebot auf der Grails-Webseite. Neben den einführenden Artikeln im *User Guide* gibt es zahlreiche Tutorials zu speziellen Beispielen (Plugins). Die Referenzen (TagLib, Validatoren) sind nicht ganz vollständig und etwas kurz gefasst.

Es gibt auch einen Grails-Podcast von Sven Haiges, einem Mitentwickler, der etwa 40 Beiträge umfasst. Überhaupt ist die Community sehr aktiv und erstellt fleißig neue Tutorials, Präsentationen, Plugins, Tags usw.

Die Groovy-Dokumentation ist sehr gut ausgebaut und befindet sich auf den Seiten von `codehaus.org`.

AJAX

In der TagLibrary gibt es einige Befehle für grundlegende Funktionalitäten auf AJAX-Basis. Um diese benutzen zu können, muss in den GSP-Seiten im Header ein unterstütztes AJAX-Framework angegeben werden mit `<g:javascript library="ajaxBibliothek" />`. Unterstützte Bibliotheken sind derzeit Prototype, Dojo und Yahoo UI. Die Grails-Tags werden dann auf die AJAX-Bibliothek gemappt.

Eine weitere Möglichkeit zur Ajax-Integration gibt das Grails-Plugin DWR (Direct Web Remoting). Dieses generiert eine `AjaxService`-Klasse, welche per Javascript über einen `EventHandler` in die Webseite eingebunden wird. Über diese `JavaScript`-Klasse können eventgesteuerte Javaprogramme auf dem Server gestartet werden. Das DWR-Plugin stellt ein Set von nützlichen `JavaScript`-Methoden zur Verfügung, die dann über einen `Callback` mit der Antwort des Java-Programms die Ajax-Funktionalität erreichen. Das Plugin ist derzeit (noch) nicht verfügbar bzw. nicht vollständig funktionabel.

Validation und Errorhandling

Validation von Eingaben

Zur Laufzeit werden vom Benutzer Eingaben gemacht, wenn dieser Objekte, also Instanzen von Domänenklassen, anlegt oder modifiziert. In den Domänenklassen kann mit einer *Constraints-Closure* spezifiziert werden, welche Bedingungen für die Gültigkeit der Nutzereingaben gelten sollen. Die Überprüfung wird explizit mit `validate()` oder implizit mit der Methode `save()`, welche `validate()` benutzt, angestoßen. Die allgemeine Form für die Closure, die `constraints` heißen muss, ist: `def constraints = { meinDomänenKlassenEigenschaft(kriterium:wert) }`

Als Prüfkriterien stehen derzeit ca. 25 vordefinierte Eigenschaften zur Verfügung: `blank`, `creditCard`, `email`, `inList`, `length`, `matches`, `max`, `maxLength`, `maxSize`, `min`, `minLength`, `minSize`, `notEqual`, `nullable`, `range`, `scale`, `size`, `unique`, `url`, `validator`.

Fehlerbehandlung

Mit den in der `TagLib` vorhandenen Tags zur Fehlerbehandlung können Fehler abgefragt und Fehlermeldungen ausgegeben werden: `eachError`, `hasError`, `message`

Unit-Testen

Eine Testumgebung ist in Grails bereits enthalten. Diese basiert auf `JUnit` und funktioniert auch so ähnlich. Man schreibt einen Testfall, d.h. eine Klasse `*Tests.groovy`, die von `GroovyTestCase` erbt und definiert dort die entsprechenden `assert`-Statements. Über die Kommandozeile mit `grails test-app` werden die Tests gestartet.

```
// Beispiel:
class meineKlasseTests extends GroovyTestCase {
    void testSomething() {
        def mK = new meineKlasse()
        def x = mK.methodeY()
        assert mK != null
        assert x > 1
    }
}
```

Funktionales Testen

Hierfür wird *Canoo Web Test* benutzt. Grails unterstützt durch Kommandozeilenbefehle die Installation und Benutzung. Zur Installation wird der Befehl `grails create-webtest` verwendet. Daraufhin wird die Webtestumgebung runtergeladen und installiert. Mit `grails generate-webtest` wird ein Gerüst für das Testen einer Klasse angelegt. Auf Nachfrage muss hier der Name der Domänenklasse angegeben werden.

WebTest kann auch gestartet werden (`grails run-webtest`), wenn die Grails-Applikation gerade *nicht* läuft, bzw. wenn noch gar keine Änderungen an den generierten Klassen vorgenommen wurden. Die Webtests schlagen dann zwar fehl, aber anhand der erstellten Reports kann ein testgetriebener Programmieransatz unterstützt werden.

Die Tests werden im Ordner `webtest` im Projektordner erstellt und konfiguriert. Der Unterordner `tests` enthält vorerst die für eine Klasse automatisch generierte Testklasse (`meineDomainKlasseTest.groovy`). Jede Testklasse muss von `grails.util.WebTest` abgeleitet sein und eine Methode `suite()` enthalten, in welcher die Testfälle aufgerufen werden.

Skalierbarkeit und Performance

Da Grails noch ein junges Framework ist existieren noch keine fundierten Benchmarks und Langzeiterfahrungen. Um jedoch trotzdem auf diesen Punkt einzugehen werden Performance- und Skalierungseffekte von den verschiedenen Komponenten betrachtet.

Weiterhin existiert ein Benchmark, welcher Rails mit Grails vergleicht. Das Fazit dieses Benchmarks ist, dass Grails in fast allen Teilen eine bessere Performance liefert.

<http://grails.org/Grails+vs+Rails+Benchmark>

Jetty wird als Server verwendet, da an dieser Stelle jeder beliebige Server verwendet werden kann, ist hier eine hohe Skalierbarkeit gewährleistet. Es existieren Implementierungen mit Clusterfähigkeiten wie WebSphere von IBM weiterhin ist die Qualität der Javaserver unbestritten. Da die Threads gepoolt werden kann man einen gemeinsamen Cache nutzen. Groovy ist bisher noch nicht auf paralleles Ausführen von Code optimiert. Dies funktioniert zwar und es gibt eine Unterstützung für Critical Sections, allerdings kann man hier noch eine höhere Geschwindigkeit erzielen.

Die Datenbank wird über JDBC angebunden damit können schnelle Datenbanken wie MySQL oder Oracle genutzt werden. Der O/R-Mapper Hibernate kann durch Features wie Lazyloading oder Connectionpooling ein hohes Maß an Performance und ein geringes Maß an Overhead garantieren.

Weiterhin verwaltet Grails einen Cache für statische Seiten. Dieser wird jedoch während der Entwicklung deaktiviert.

Das Starten des Server und das generieren der verschiedenen Dateien dauert bei Grails am längsten. Hier wird leistungsfähige Hardware empfohlen.

Noch ein kleiner Benchmark für die später noch vorgestellte Mensaanwendung:

System:

MacBook 2.0 GHz Intel Core Duo

2 GByte Ram

\$ java -version

java version "1.5.0_07"

Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-164)

Java HotSpot(TM) Client VM (build 1.5.0_07-87, mixed mode, sharing)

Es werden 100 Anfragen jeweils 10 parallel an die Listview des PersonControllers geschickt. Es existieren 2 Personen in der HSQL in Memory Database.

```
$ ab -c 10 -n 100 http://localhost:8080/Mensa/Person/list
Benchmarking localhost (be patient).....done
Server Software:      Jetty/5.1.5rc2
Server Hostname:     localhost
Server Port:        8080
Document Path:      /Mensa/Person/list
Document Length:    1230 bytes
Concurrency Level:   10
Time taken for tests: 6.416 seconds
Complete requests:   100
Failed requests:
Broken pipe errors:
Non-2xx responses:   100
Total transferred:  150970 bytes
HTML transferred:   123000 bytes
Requests per second: 15.59 [#/sec] (mean)
Time per request:    641.60 [ms] (mean)
Time per request:    64.16 [ms] (mean, across all concurrent requests)
Transfer rate:       23.53 [Kbytes/sec] received

Connnection Times (ms)
      min mean[+/-sd] median max
Connect:    0  0  0.2  0  3
Processing: 375 629 59.8 638 763
Waiting:    374 629 59.9 638 763
Total:      375 629 59.9 638 763

Percentage of the requests served within a certain time (ms)
50%  638
66%  655
75%  667
80%  673
90%  684
95%  701
98%  759
99%  763
100% 763 (last request)
```

Die nicht optimierte Grailsanwendung kann knapp 16 Request pro Sekunde beantworten. Damit sollten auf einem Server ohne größere Probleme 100 gleichzeitige User an einer Anwendung arbeiten.

Anwendungsmodi dev, prod, test

Grails stellt drei vorkonfigurierte Entwicklungsszenarien zur Verfügung: *Development*, *Production* und *Test*. Alle drei Konfigurationen sind mit passenden Default-Werten für die Datenbankanbindung und das Caching-Verhalten besetzt und können über die Dateien im Ordner `grails-app/conf` angepasst werden. In der *Development*-Einstellung ist zum Beispiel der Schreibmodus auf `create-drop` gesetzt. So werden Daten und Struktur nach Beendigung der Ausführung wieder entfernt. Dies ist vor allem sinnvoll, wenn man während der Entwicklung die Domänen-Attribute ändert und dementsprechende Änderungen an Datenbankschema und Datenbestand vornehmen müsste. Bei Verwendung der Einstellung `create` werden nur die Daten gelöscht, bei `update` bleiben Daten und Strukturen erhalten.

In jedem Modus macht eine Änderung am Model einen Neustart der Applikation nötig. Im *dev*-Modus werden Änderungen an Controller und View zur Laufzeit automatisch neu geladen, so dass eine flexible, schnelle Anpassung möglich ist, z.B. für Echtzeit-Prototyping. Da im *prod*-Modus die Controllers und Views statisch geladen werden, also nicht zur Laufzeit modifizierbar sind, hat die Anwendung dann aber eine höhere Performance.

Mit dem Voranstellen von `prod`, `dev` bzw. `test` vor die Grails-Befehle z.B. `run-app` wird eine Konfiguration ausgewählt. Wird der Parameter weggelassen, wird `dev` verwendet.

Beispiele (analog: war-Befehl)

```
grails dev run-app
grails prod run-app
grails test run-app
grails run-app           // wie Zeile 1
```

Teil III GRAILS - Beispiel

In folgender Anwendung werden die Konzepte von Grails und ihre Umsetzung anhand eines praktischen Beispiels beleuchtet.

Java SDK 1.4 oder höher haben wir bereits installiert. Auf der Grails-Webseite laden wir die gewünschte Grails-Version herunter und entpacken sie in das Verzeichnis `C:/Programme/Grails`. Dann setzen wir zwei Umgebungsvariablen: `GRAILS_HOME` wird auf `C:/Programme/Grails` gesetzt und `JAVA_HOME` auf den Java-Installationspfad. Zur Umgebungsvariable `PATH` wird der Pfad `%GRAILS_HOME%\bin` hinzugefügt. Unter Windows wird das jeweils unter `Systemsteuerung -> System -> Erweitert -> Umgebungsvariablen -> Systemvariablen`. Unter Linux mit `export`.

Wir legen einen Ordner `meineGrailsProjekt` an und wechseln per Kommandozeile in den Ordner. Wir rufen `grails create-app` auf. Auf Nachfrage geben wir den Namen `Mensa` ein. Im entstandenen Ordner `Mensa` starten mit `grails run-app`. Die Webapplikation ist fertig und läuft unter `http://localhost:8080/Mensa/`.

Wir legen 2 Domänenklassen (domain classes) mit `grails create-domain-class`, aufgerufen im Ordner `Mensa` an und geben den Klassen die Namen `Person` und `Essen`. Grails legt die leeren Klassen im Ordner `grails-app/domain` an.

Die Domänenklassen brauchen natürlich noch ein paar Attribute. `Essen` bekommt die Eigenschaften `String name` und `Integer preis`. `Person` erhält die Eigenschaften `String name` und `Integer alter`.

Nun lassen wir Grails wieder für uns arbeiten: `grails generate-all` (startet selbstständig `generate-views` und `generate-controller`) generiert die Controller in `grails-app/controllers` und die Views in `grails-app/views`. Jeweils für `Essen` und `Person` ausführen. Danach muss die Applikation neu gestartet werden.

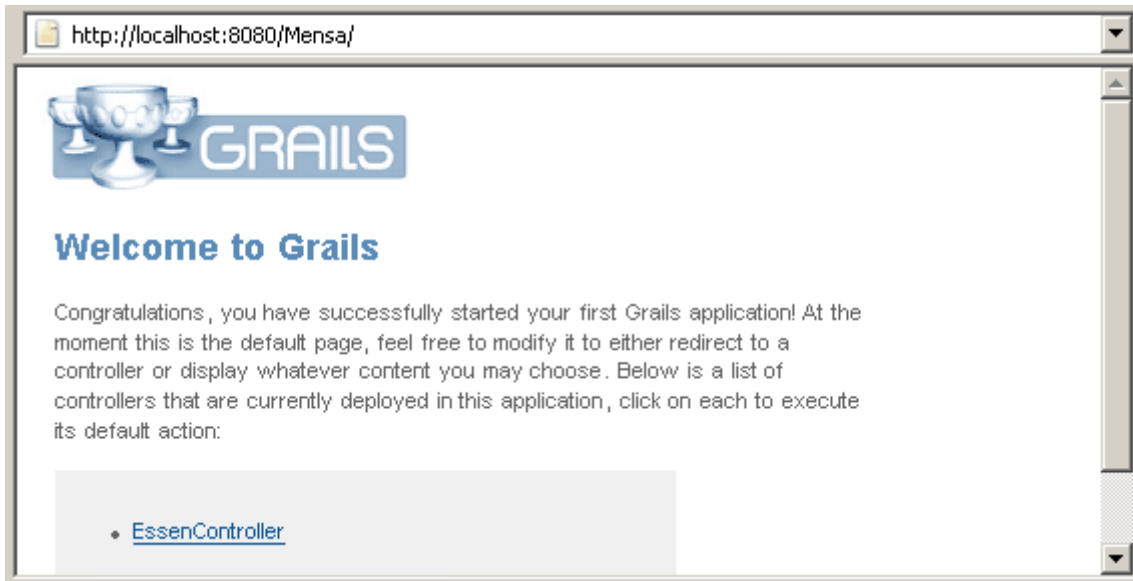


Abbildung 3: Groovy Mensa Demoapplikation. Startseite.

```
// M o d e l
// Domänenklasse Person
// Mensa/grails-app/domain/Person.groovy
class Person {
    String name
    Integer alter
}
```

```
// C o n t r o l l e r für Person
// Mensa/grails-app/controllers/PersonController.groovy
class PersonController {
    def index = { redirect(action:list,params:params) }
    def list = { if(!params.max)params.max = 10
                [ personList: Person.list( params ) ]    }
    def show = { [ person : Person.get( params.id ) ]    }

    // weitere CRUD-Methoden, die von Grails erzeugt wurden:
    // delete, edit, update, create, save
}
```

Die Methode `index()` enthält eine Weiterleitung (`redirect()`) zu einer anderen Methode. Dafür werden als Parameter `action` die neue Zielmethode und als Parameter `params` die aktuellen Parameter übergeben. Man könnte auch weitere Parameter anhängen.

Die Methode `show()` zeigt, wie man eine Map aus den Instanzen einer Domänenklasse (hier bezogen aus das Attribut `id`) generiert.

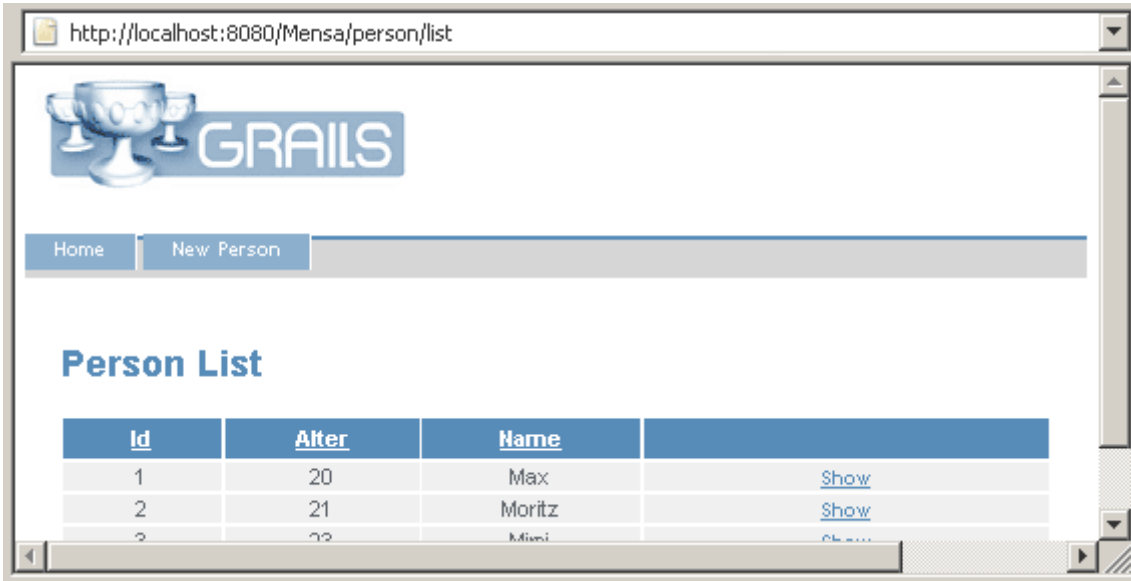


Abbildung 4: Groovy Mensa Demoapplikation. CRUD-Methoden von Grails erzeugt.

```
// V i e w
// List-View für Person, analog werden Edit-, Show- und Create-Views generiert
// Ausschnitt aus Mensa/grails-app/views/person/list.gsp
<html>
<head>
  <meta name="layout" content="main" />
</head>
<body>
<span class="menuButton"><a href="${createLinkTo(dir: '')}">Home</a></span>
<span class="menuButton"><g:link action="create">New Person</g:link></span>
<div class="body">
  <table>
    <tr> <g:sortableColumn property="id" title="Id" />
      <g:sortableColumn property="alter" title="Alter" />
      <g:sortableColumn property="name" title="Name" />
      <th></th>
    </tr>
    <g:each in="${personList}">
    <tr> <td>${it.id?.encodeAsHTML()}</td>
      <td>${it.alter?.encodeAsHTML()}</td>
      <td>${it.name?.encodeAsHTML()}</td>
      <td><g:link action="show" id="${it.id}">Show</g:link></td>
    </tr>
    </g:each>
  </table>
</div>
</body>
</html>
```

Die oben gezeigte `list`-View enthält einige Beispiele für GSP-Tags aus der integrierten `TagLibrary`: `<g:link>`, `<g:sortableColumn>` und `<g:each>`. Während das Schleifenkonstrukt `each` aus Start- und Endtag besteht, die den zu wiederholenden Bereich eingrenzen, sind `link` und `sortableColumn` alleinstehende Tags. Der Zugriff auf die Attribute der aktuell verarbeiteten Instanz geschieht über das eingebaute Schlüsselwort `it`. Auszuwertende Ausdrücke werden mit `${...}` gekennzeichnet.

Da per Default-Einstellung im `dev`-Modus auf einer in-memory-Datenbank gearbeitet wird, sind die Datenbank und die eingefügten Instanzen bei jedem Neustart der Applikation verschwunden. Damit wir zum Entwickeln nicht immer wieder Testdaten einfügen müssen, können wir den Bootstrap-Mechanismus von Grails nutzen und in der Klasse `ApplicationBootstrap` ein paar Essen und Personen einfügen.

```
// grails-app/conf/ApplicationBootstrap.groovy
class ApplicationBootstrap{
    def init = { servletContext ->
        def e = new Essen(name:"Bockwurst", preis:2)
        def p = new Person(name:"Mimi", alter:19)
        e.save()
        p.save()
    }
}
```

Mit `grails create-controller` legen wir einen neuen Controller an. Es sollen alle Personen und jedes Essen angezeigt werden, also nennen wir den Controller `Gesamtanzeige`. Jetzt können wir die generierte `GesamtanzeigeController.groovy` öffnen, eine `list`-Methode einfügen und `index` auf `list` umleiten:

```
//Ausschnitt aus Mensa/grails-app/Controller/GesamtanzeigeController.groovy
def index = {redirect(action:list,params:params)}
def list = {
    params['max'] = 10
    return [ personList: Person.list( params ),
            essenList: Essen.list( params ) ]
}
```

Für diesen Controller wird eine eigene View erstellt. Die Datei `list.gsp`, die im Ordner `../views/gesamtanzeige` ablegt wird, kopieren wir uns am besten aus anderen `list.gsp`-Dateien zusammen. Sie sollte mindestens folgenden Inhalt haben.

```
// Ausschnitt aus grails-app/views/gesamtanzeige/list.gsp
<html><head>
<title>Anzeige als Gesamtübersicht</title>
<link rel="stylesheet" href="\${createLinkTo(dir:'css',file:'main.css')}}" />
</head>
<body>
<div class="body">
<h1>Gesamtanzeige</h1>
<h2>Personen</h2>
<table>
  <tr> <td>Name</td>      <td>Alter</td>          </tr>
  <g:each in="\${personList}">
    <tr> <td>\${it.alter?.encodeAsHTML()}</td>
      <td>\${it.name?.encodeAsHTML()}</td>          </tr>
  </g:each>
</table>
<h2>Essen</h2>
<table>
  <tr> <td>Name</td>      <td>Preis</td>          </tr>
  <g:each in="\${essenList}">
    <tr> <td>\${it.name?.encodeAsHTML()}</td>
      <td>\${it.preis?.encodeAsHTML()}</td>          </tr>
  </g:each>
</table>
</div>
</body></html>
```

Zur Implementierung der Businessfunktionalität werden bei Grails Services verwendet. Arbeitet ein Service auf persistenten Daten oder wird aus sonstigen Gründen Transaktionalität gefordert, kann dies einfach durch das Setzen der vordefinierten Variable `transactional` auf `true` sichergestellt werden.

Wir geben `grails create-service` ein und auf Nachfrage den Namen `Mensa`. Es werden zwei Dateien generiert: `grails-app/services/MensaService.groovy` und `grails-tests/MensaTest.groovy`. In `MensaService.groovy` ist schon eine leere `serviceMethod()` vorgefertigt, in die wir etwas einfüllen könnten. In unserem Beispiel verwenden wir aber eine eigene Methode `tuWas()`. In den Controller `Gesamtanzeige` fügen wir die Closure `serviceAufrufen` ein. Aufrufen können wir den Service unter `http://localhost:8080/Mensa/gesamtanzeige/serviceAufrufen`.

```
// Ausschnitt Mensa/grails-app/services/MensaService.groovy
class MensaService {
  def tuWas() {
    return "Hallo! Ich bin deine Businessfunktionalität."
  }
}
```

```
// Ausschnitt Mensa/grails-app/controller/GesamtanzeigeController.groovy
def MensaService mensaService
...
def serviceAufrufen = {
  render(mensaService.tuWas())
}
```

Die hier verwendete Methode `render()` ist eine der eingebauten dynamischen Methoden. Sie ist sehr vielfältig einsetzbar, wenn man vom Controller aus irgend eine Textausgabe in der View erzeugen möchte. Man kann mit `render "Hallo Welt"` einfachen Text rendern oder eine Schleife z.B. `render { for (p in PersonList) { p.name } }` abarbeiten. In unserer Beispielanwendung ruft `render()` einen Service auf. Es gibt viele weitere Anwendungsmöglichkeiten dieser Methode.

Die `.gsp`-Dateien, die für die Methoden existieren, die eine Anzeige generieren, kann man einzeln an die gewünschte Darstellungsform anpassen. Ein Basislayout für die gesamte Anwendung wird in `views/layouts` definiert. Dort liegt nach der Installation die vordefinierte `main.gsp`. Lässt man Grails eine View generieren, wird automatisch eine Referenz auf diese `main.gsp` eingefügt.

```
// Ausschnitt aus einer beliebigen View z.B. grails-app/views/person/list.gsp
<head>
<meta name="layout" content="main" />
</head>
```

```
// Ausschnitt aus grails-app/views/layouts/main.gsp
<html>
<head>
<title></title>
<link rel="stylesheet" href="${createLinkTo(dir:'css',file:'main.css')}" />
<g:layoutHead />
</head>
<body>
<g:link>
    
</g:link>
<g:layoutBody />
</body>
</html>
```

Die `main.gsp` könnte einfacherweise wie oben dargestellt aussehen. Mit der Methode `createLinkTo()` wird im Header eine `.css`-Datei referenziert und im Body ein Bild.

Um unser Layout weiter zu verschönern, wollen wir eine eigene Tag-Bibliothek anlegen und ein Tag zum Einfügen einer Trennlinie aus Interpunktionszeichen programmieren. Unsere TagLib nennen wir `MensaTagLib.groovy` und speichern sie unter `grails-app/taglib/`. Das erste Tag nennen wir `vieleZeichen` und definieren es als alleinstehendes GSP-Tag. Das zweite Tag wird ein iteratives Tag, bei welchem der Quelltext, der zwischen dem Start- und Ende-Tag steht, wiederholt wird.

```
class MensaTagLib{
    def vieleZeichen = {    attrs ->
        r = attrs['cssStil']
        out << "<span class='${attrs['cssStil']}'>...; *°00o, _..</span>"
    }
}
```

```
def wiederholung = {  attrs, body ->
    def i = Integer.valueOf(attrs['anzahl'])
    def counter = 0
    i.times { out << body(++counter) }
}
}
```

In unseren Views (z.B. in der `main.gsp`) können wir die neuen Tags nun aufrufen:

```
<g:wiederholung anzahl="10">
    <g:vieleZeichen cssStil="trennlinie" />
</g:wiederholung>
```

Als nächstes werden wir in die Domänenklassen Constraints einfügen, um die Nutzereingaben zu validieren. Die Klasse `Person` hat die Eigenschaften `name` und `alter`. Damit keine Personen mit einem unmöglichen Alter geschaffen werden können, fügen wir die Closure `constraints` in die Klasse ein. Die Prüfkriterien `min` und `max` sind bereits von Grails zur Verfügung gestellt.

```
// Domänenklasse Person
// Mensa/grails-app/domain/Person.groovy
class Person {
    String name
    Integer alter
    def constraints = {
        alter( min:0, max:100)
    }
}
```

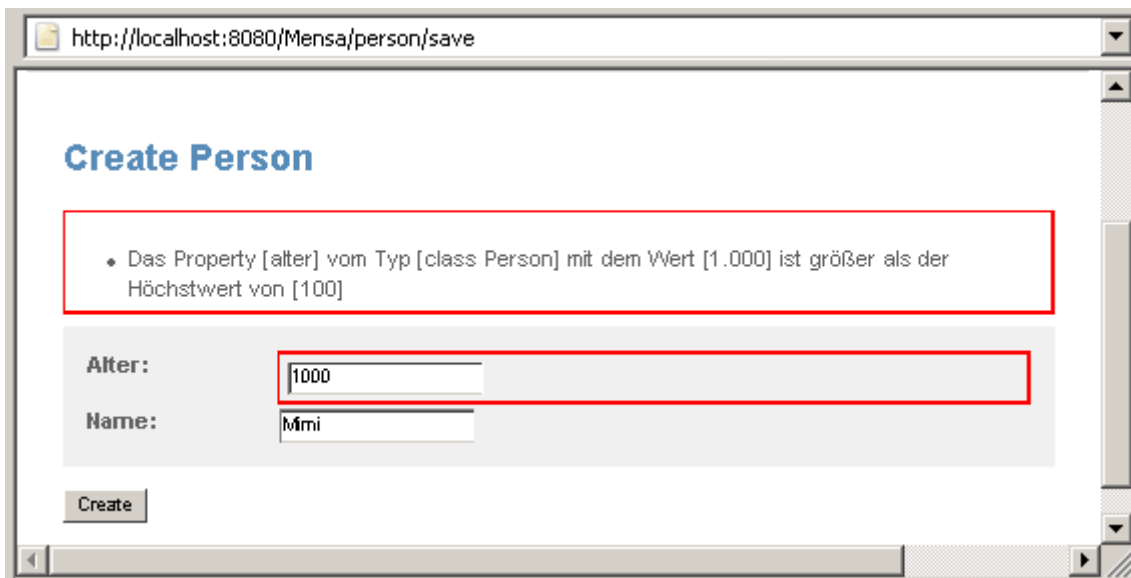


Abbildung 5: Groovy Mensa Demoapplikation. Gültigkeits-Check, Fehlermeldung von Grails eingefügt

Um zu demonstrieren, wie eine 1:N -Beziehung mit Grails verwirklicht wird, lassen wir uns von Grails eine weitere Domänenklasse mit dem Namen `Mensa` generieren und fügen die Attribute `String standort` und `hasMany` ein. In die Klasse `Essen` fügen wir `def belongsTo` ein. `belongsTo` und `hasMany` beziehen sich, wie im Listing gezeigt, auf die jeweils andere Klasse. Damit ist die 1:N-Beziehung ausreichend beschrieben. Jedes Essen kann nun einer Mensa zugewiesen werden (mit der dynamischen Methode `add*()`, zum Beispiel: `meineMensa.addEssen(meinEssen)`). Wird diese Mensa gelöscht, werden die dazugehörenden Essen ebenfalls gelöscht.

```
// Domänenklasse Mensa
// Mensa/grails-app/domain/Mensa.groovy
class Mensa {
    String standort
    def hasMany = [essenAngebot:Essen]
}

// Domänenklasse Essen
// Mensa/grails-app/domain/Essen.groovy
class Essen {
    String name
    Integer preis
    def belongsTo = Mensa
}
```

Bevor wir die Anwendung neu starten, lassen wir uns noch die Controller und Views von `Mensa` generieren. Grails schafft uns dabei ein Gerüst zum Hinzufügen von Essen zur Mensa.

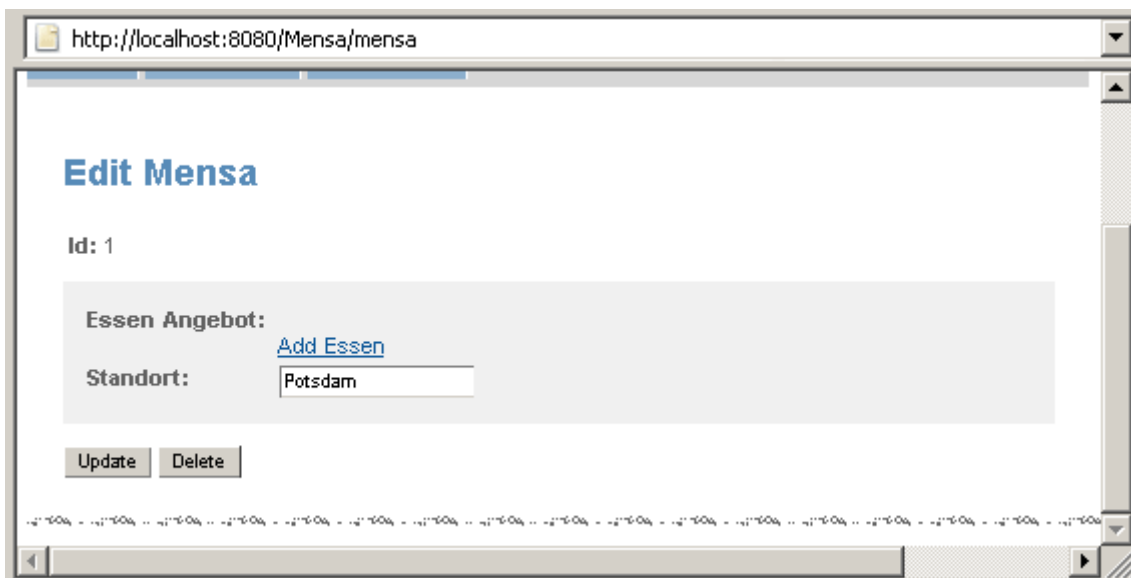


Abbildung 6: Groovy Mensa Demoapplikation. Essen in Mensa einfügen, automatisch von Grails erstellt.

Wir müssen allerdings noch einige Details einfügen. In unserem Beispiel wollen wir beim Editieren einer Mensa neue Essen kreieren und anfügen können. Den Link zur `create`-Methode des EssenControllers und den Parameter `mensa.id` hat uns Grails schon generiert. Wir fügen nun der `save`-Methode des EssenControllers hinzu:

```
// Ausschnitt aus grails-app/controllers/EsSENController.groovy
def save(){
    def essen = new Essen()
    essen.properties = params
    if(params['mensa.id']){
        def m = Mensa.get(Integer.valueOf(params['mensa.id']))
        m.addEssen(essen)
    }
    ...
}
```

Während die Übergabe der Mensa-ID von der `edit`-Seite der Mensa zur `create`-Seite des neuen Essens schon von Grails eingebaut wurde, müssen wir in unserem Beispiel die `create`-Seite noch dahingehend modifizieren, dass die Mensa-ID an die `save`-Methode des EssenControllers weitergereicht wird. Dazu fügst du im `<g:form>`-Tag der `create`-Seite folgendes ein:

```
// Ausschnitt aus grails-app/views/essen/create.gsp
...
<g:form action="save?mensa.id=${params['mensa.id']}" method="post" >
...

```

Die 1:N-Beziehung funktioniert nach diesen wenigen Schritten. Wenn eine Mensa gelöscht wird, werden die dazugehörigen Essen ebenfalls gelöscht.

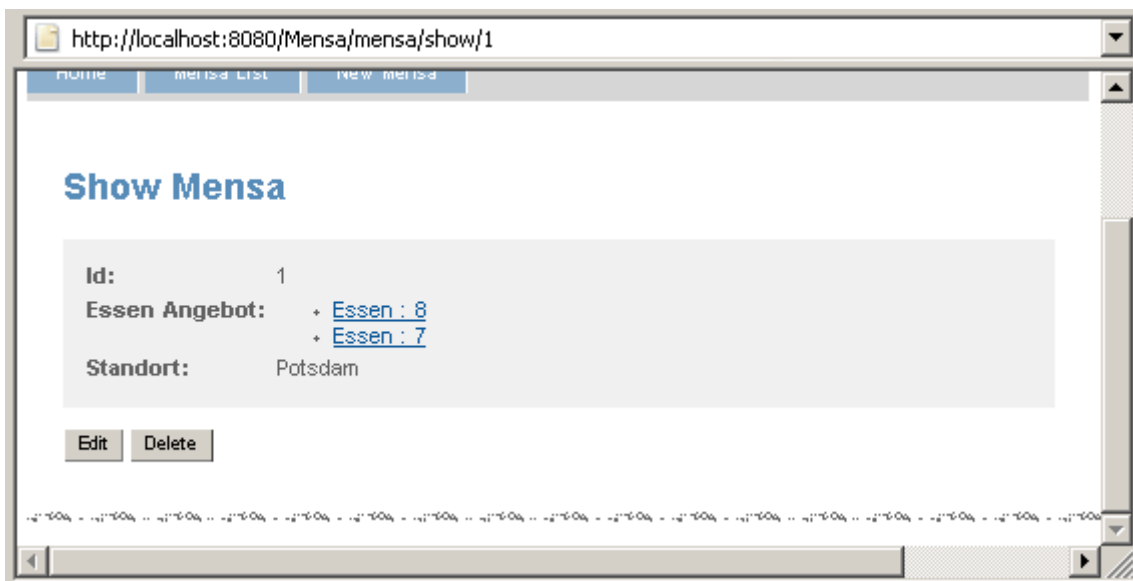


Abbildung 7: Groovy Mensa Demoapplikation. Wird Mensa 1 gelöscht, werden auch Essen 7 und 8 gelöscht.

Abschließend noch ein kleines AJAX-Beispiel. Für die Show-Ansicht von Essen wollen wir den Delete-Button durch einen AJAX-Remotelink ersetzen. Ohne die Seite neu zu laden, soll beim Anklicken das Essen gelöscht werden und die Knopfleiste durch eine Meldung ersetzt werden. Dark Grails sind hierfür absolut keine AJAX-Kenntnisse erforderlich. Einzig den Namen einer der zur Verfügung stehenden AJAX-Bibliotheken müssen wir kennen. Wir nehmen beispielsweise Prototype.

```
// Ausschnitt aus grail-ap/views/essen/show.gsp
<head> ...
<g:javascript library="prototype" />
... </head>
<body> ...
<div class="buttons">
<div id="message">
  <g:form controller="essen">
    <input type="hidden" name="id" value="${essen?.id}" />
    <span class="button"><g:actionSubmit value="Edit" /></span>
    <g:remoteLink action="delete" id="${essen?.id}" update="message">
      Neuer Löschknopf
    </g:remoteLink>
  </g:form>
</div>
</div>
... </body>
```

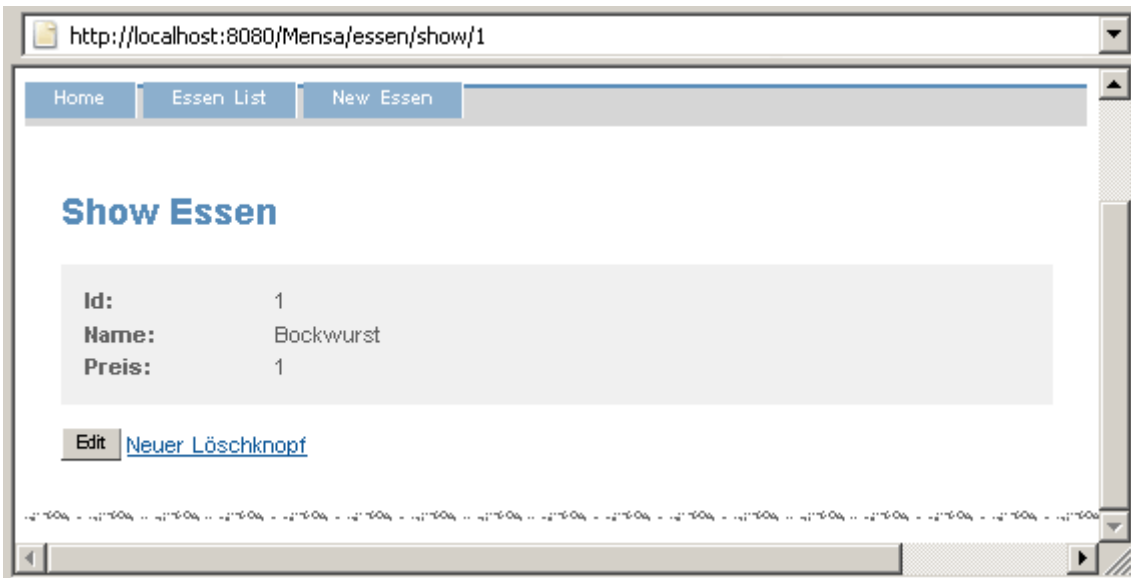


Abbildung 8: Groovy Mensa Demoapplikation. Delete-Button durch AJAX-Link ersetzt

```
// Ausschnitt aus grails-app/controller/EssenController.groovy
def delete() {
  def essen = Essen.get(params.id)
  if(essen) {
    essen.delete()
    render "Essen <em>${essen.name}</em> wurde gelöscht.
          Diese Mitteilung erscheint per AJAX."
    ...
  }
}
```

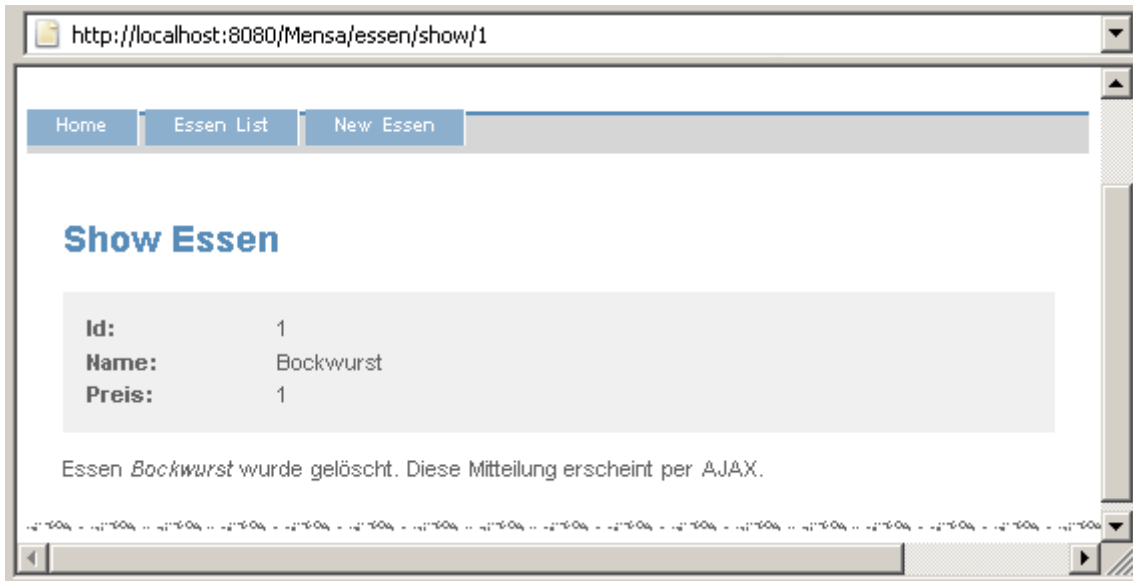


Abbildung 9: Groovy Mensa Demoapplikation. Bockwurst gelöscht und Mitteilung per AJAX erschienen.

Quellen

- <http://www.grails.org>
- <http://groovy.codehause.org>
- <http://www.spring-framework.org>
- <http://www.opensymphony.com>
- <http://www.hibernate.org>
- <http://stud.fh-wedel.de/~minf2622/grails/grails5.htm>
- <http://dev2dev.bea.com/pub/a/2006/10/introduction-groovy-grails.html?page=3>
- <http://skillsmatter.com/downloads/Grails%20Dynamic%20Tag%20Libraries.pdf>
- <http://getahead.org/dwr/>
- <http://www.k-oo.de/blog/2006/12/08/grails-konfiguration-der-datenbankverbindung/>
- <http://grails.org/Grails+vs+Rails+Benchmark>