Dokumentation

Open Laboratory Framework

Version 0.4
30.08.2010

Osthus GmbH
Wilhelmstr. 79
52070 Aachen
Tel. +49 241 94314 0
Fax. +49 241 94314 19
www.osthus.de
info@osthus.de

Dokumentenkontrolle

Titel	Dokumentation OpenLaboratoryFramework
Dokumenten Typ	Dokumentation
Version	0.4
Autor	Markus List
Ausgabedatum	30/08/2010
Gesamtanzahl Seiten	23

Dokumentenhistorie

Version	Ausgabedatum	Bemerkung
0.1	09.07.2010	Ersterstellung
0.4	30.08.2010	Fertigstellung

Dokumentenverweise

Titel	Autor	Version	Datum

OSTHUS
success with R&D

Inhaltsverzeichnis

1 Einleitung 4
1.1 Projektverlauf 4
1.2 Ziele 4
2 GRAILS 5
2.1 Groovy 6
2.1.1 Dynamisches Groovy 6
2.1.2 Collections 7
2.2 GORM und DomainClasses 8
2.3 Controller haben die Macht 9
2.4 Business-Logik durch Service-Klassen 9
2.5 Die View wird generiert durch GSP und TagLibs 10
2.6 TagLibs 11
2.7 Dynamic Scaffolding 11
2.8 Erweiterbarkeit durch Plugins 12
3 OpenLaboratoryFramework 12
3.1 Grundlegender Aufbau (Plugins) 12
3.2 Erweiterung des Inhalts durch Modules 13
3.2.1 TabModules 14
3.2.2 MenuModules 15
3.2.3 AddinModules 16
3.2.4 OperationsModules 17
3.3 Erstellung eines Dokumenten-Plugins 18
3.3.1 Erstellung des leeren Plugin 18
3.3.2 Erstellung des Addins 19
4 Öffentliche Plugins, die verwendet werden 21
4.1 Spring-Security Plugin 21
4.2 GRAILS-UI und RICH-UI 21
4.3 Searchable 22
4.4 Export 22
4.5 Excel-Import 22
4.6 Filter-Plugin 22
4.7 Modalbox-Plugin 22
5 SettingsService 23
6 Weiterführende Literatur 23

1 EINLEITUNG

Das OpenLaboratoryFramework (OLF) wurde im Rahmen einer 6-monatigen Praktikumsarbeit bei der Firma osthus GmbH entwickelt. Auftraggeber und Abnehmer der Software ist die Universität Süd-Dänemark. Konzeptionelle Basis der Arbeit war die sich zuvor im Einsatz befindliche Software „genetracker“ (gt).

1.1 Projektverlauf

Zielsetzung des Projektes war die Entwicklung eines Frameworks, welches die bisherige Funktionalität von gt mit der Flexibilität und Erweiterbarkeit eines modernen Frameworks verbindet. Dazu wurde zunächst ein ausführlicher Vergleich von Technologien auf Java-Basis durchgeführt. Nach Auswahl von GRAILS als Schlüsseltechnologie begann eine Einarbeitungsphase, während derer die Möglichkeiten ausgelotet und praktische Erfahrung gesammelt wurde. Im nächsten Schritt wurden zunächst zentrale Anforderungen auf Umsetzbarkeit geprüft. In den Fällen, in denen eine Umsetzung nicht ohne Weiteres möglich war, wurden Abstriche und Kompromisse gemacht, um der kurzen Projektdauer Rechnung zu tragen. Nach Festlegung der Eckdaten begann dann die Implementierung.

1.2 Ziele

Die folgenden Eigenschaften waren für die Entwicklung des OLF vorgegeben:

- Abbildung der bisherigen Funktionalität des gene-trackers (gt).
- Migration der Altdaten auf das neue System.
- Flexibilität in Bezug auf die zu wählende Datenbank.
- Erweiterbarkeit sowohl um komplett neue Features, als auch Erweiterung bestehender Features ohne große Eingriffe im System.
- Modularisierung, die eine Erweiterung ermöglicht ohne alle Komponenten des Frameworks kennen zu müssen. Ferner soll eine Erweiterung möglichst einfach zu installieren sein.
- Sekundärinformation (anwendungsrelevante Informationen) wie Applikations- und Benutzereinstellungen sollen in einer zweiten lokalen Java-DB gespeichert werden.

- Saubere Trennung nach dem Model-View-Controller (MVC) Pattern. Die Erstellung neuer View-Schichten soll möglichst einfach von der Hand gehen, z.B. um später angepasste Ansichten für andere Systeme (Barcodescanner) zu erstellen.

- Zugriff nicht nur auf die lokale Instanz, sondern auch auf Daten anderer Instanzen über einen Webservice.

- Barcodes sollen gedruckt und gelesen / gespeichert werden können.

- Implementierung einer Sicherheitslösung, welche zunächst ein einfaches Nutzer- /Gruppenkonzept mit Authentifizierung umsetzt und später um ACL-Funktionalität erweitert werden soll.

2 GRAILS

Zunächst sollten moderne Webentwicklungstechniken auf Java-Basis auf ihre Eignung geprüft werden. Die zentrale Forderung war dabei, dass möglichst viel Arbeit bereits durch das Framework abgenommen wird und es trotzdem möglich bleibt eine beliebig komplexe Anwendung zu generieren. Außerdem sollte dabei keine Nischentechnologie zum Einsatz kommen, welche nach kurzer Zeit nicht mehr verfügbar wäre.

Die Entscheidung fiel deshalb letztlich auf GRAILS, einem Java-Framework, welches ganz im Zeichen des „rapid web development“ steht und dies durch konsequenten Einsatz zentraler Dogmen wie „convention over configuration“ erreicht. Seine Stärke erreicht GRAILS vor allem dadurch, dass bestehende - für sich genommen bereits sehr mächtige - Technologien , wie beispielsweise das Spring Framework, Hibernate, Sitemesh und einige andere, in einem einzigen Framework zusammen gefasst werden. Diese Technologien werden dann im Sinne des Facade-Patterns mit einer darüber liegenden Schicht versehen, welche es durch vorgegebene Konfiguration erlaubt sehr schnell eine grobe Webapplikation zu erstellen.

Dabei wurde viel Gewicht darauf gelegt, dass die Mächtigkeit der zugrunde liegenden Technologien trotzdem noch genutzt werden kann, sollte dies nötig sein. So erlaubt es GRAILS auf geschickte Weise, dass Anfänger der Webtechnologien komplexe Seiten erstellen können, ohne sich beispielsweise über Datenbankpersistierung, CSS-Layouting, JavaScript, oder Spring-Bean-Konfiguration jemals Gedanken machen zu müssen. Besser noch: Man muss nicht einmal wissen, dass diese Technologien existieren. Auf der anderen Seite können Profis an den geeigneten Stellen auf die verborgenen Mechanismen jederzeit und problemlos zugreifen. Fortgeschrittene Entwickler können schnell starten und sich anschließend an gegebener Stelle tiefer einarbeiten ohne von der Gesamtkomplexität erschlagen zu werden.

Im folgenden wird auf einzelne Kernaspekte von GRAILS kurz eingegangen. Das Wissen über die grundsätzliche Funktion des Frameworks erlaubt es Entwicklern die bestehende Applikation in ihrer jetzigen Form schneller zu verstehen und damit schneller produktiv zu werden.

2.1 Groovy

Die Namensähnlichkeit mit Rails kommt nicht von ungefähr. Rails dient als Vorbild für „rapid web development“ gilt. Da es jedoch seine Dynamik durch Einsatz der Sprache Ruby gewinnt, kommt es für viele Unternehmen nicht in Frage, die das Umlernen auf eine möglicherweise nur kurzzeitig existente Sprache scheuen. Auch GRAILS erreicht seine Flexibilität durch eine dynamische Skriptsprache namens Groovy. Groovy leitet sich jedoch von Java ab und kann von Java-Entwicklern sehr schnell erlernt werden. Quasi die komplette Syntax, die in Java funktioniert, funktioniert auch in Groovy. Hinzu kommen viele Methoden und Konzepte, welche das Leben erleichtern. Dadurch könnte man von Groovy fast als Supermenge von Java sprechen.

2.1.1 Dynamisches Groovy

Was bedeutet nun „dynamisch“ im Zusammenhang mit einer Programmiersprache? Bei Java ist zur Compilezeit beispielsweise bereits klar, welches Objekt welchen Typ hat. In Groovy werden üblicherweise ähnlich wie in JavaScript gar nicht erst Typen deklariert. Man schreibt nicht String s oder int n, sondern kennzeichnet mit def s oder def n einen dynamischen Typ. Dieser wird durch den Compiler während der Laufzeit bestimmt.

2.1.2 Collections

Die Typunabhängigkeit erstreckt sich durch die ganze Sprache, so spielt es z.B. auch bei Collections keine Rolle, welcher Typ hinzugefügt wird. Man kann auch verschiedene Objekte hinzufügen. Der Zugriff ist dabei sehr einfach gehalten, so kann man mit

```
def map = [:]
```
eine Hashmap deklarieren. Mit

```
map.einbeliebigerkey = einbeliebigerwert
```
hat man einen Eintrag hinzugefügt.

```
[key1:wert1, key2: wert2]
```
funktioniert genau so. Es gibt zahlreiche weitere Beispiele wie Groovy das Leben des Programmierers vereinfacht, doch dies würde den Rahmen dieses Dokuments sprengen.

Der wohl wichtigste Aspekt von Groovy soll jedoch noch genannt werden. Anstelle von Methoden werden in Groovy fast immer sogenannte Closures verwendet. Die Definition unterscheidet sich nur leicht:

```
Methode: def methode(def param1, def param2){ }
Closure: def closure = { attrs ->}
```
Man erhält Zugriff auf die Parameter über die Hashmap attrs. Also

```
attrs.param1, attrs.param2, …
```
An dieser Stelle wird man sich fragen, was dadurch gewonnen ist. Der Vorteil liegt darin, dass es sich bei Closures um Methoden-Objekte handelt, welche als Objekte auch weitergereicht werden können. Eine Closure teil dabei den Scope des umgebenden Codes und kann dadurch flexibler an verschiedenen Stellen eingesetzt werden, ohne dass man explizit eine Vielzahl von Parametern übergeben müsste.

2.2 GORM und DomainClasses

GRAILS benützt Hibernate um eine Datenbankpersistierung über JDBC zu ermöglichen. Kenner von Hibernate, die komplexe Annotation von Klassen oder das Schreiben von XML-Mappings fürchten dürfen aufatmen. Durch Grails Object Relational Mapping (GORM) wird die Benutzung von Hibernate ein Kinderspiel. Man definiert in GRAILS lediglich die benötigten Java-Klassen als DomainClasses und GRAILS erledigt den Rest. Beispiel: Wir legen eine Klasse Gen an und legen diese im passenden Ordner GRAILS-APP/Domain ab. Der Ort ist wichtig, denn hier kommt „Convention-over-Configuraion“ zum Einsatz, welches uns manuelle Konfiguration erspart. Die Klasse Gen könnte so aussehen:

```
class Gene{
String sequence
String accessionNumber
Date createdDate
User author
hasMany [vectors: Vector, primers: Primer]
}
```

Im einfachsten Fall werden nur die gewünschten Properties deklariert. Man baut sich ein sehr einfaches POGO (Plain Old Good Object) und ist bereits fertig. Für 1:n oder n:n Verknüpfungen fügt man eine hasMany Hashmap hinzu. Man kann auch hier durch Konfiguration alles an spezifische Wünsche anpassen, so kann z.B. vorgegeben werden, dass Sequence als Text persistiert werden soll, dass der Name der Relation anders lauten soll, welche Einschränkungen (constraints) für die einzelnen Felder gelten sollen und unendlich viel mehr. Doch auch hier sprengen wir mit den Details den Rahmen des Dokuments.

Worauf wir jedoch noch kurz eingehen wollen sind die sogenannten „Dynamic Finders“, die von GORM zur Laufzeit generiert werden. Solche Methoden sind aus den Properties der DomainClass zusammen gesetzt und erlauben das effektive Suchen nach Daten. So kann z.B. ein Gen mit einer bestimmten AccessionNumber, welches einem bestimmten Projekt angehört so gefunden werden:

```
def gene = Gene.findByAccessionNumberAndProject(accessionNumber, project)
```

Gleichermaßen können auch mehrere Treffer bestimmt werden:

```
def gene = Gene.findAllByName(name)
```

2.3 Controller haben die Macht

Im letzten Kapitel wurden DomainClasses vorgestellt. Dabei handelt es sich um eins von vielen Artefakten. GRAILS versteht unter einem Artefakt Klassen, die bestimmte Bedingungen erfüllen müssen, um eine bestimmte Behandlung zu erfahren. So werden alle Klassen in GRAILS-APP/Domain als DomainClasses behandelt. Alle Klassen in GRAILS-APP/Controller, die auf Controller enden (z.B. GeneController) werden automatisch als Controller erfasst und gehalten. Später werden noch weitere Artefakte vorgestellt.

Was ist nun ein Controller? Bei einem Controller handelt es sich um die Schaltzentralen der Applikation. Ganz dem Model-View-Controller Pattern entsprechend steuern die Controller den Programmablauf, während z.B. DomainClasses die Model-Ebene darstellen. Die View-Ebene wird durch Views vertreten, welche später erläutert werden.

Die Closures, welche in einem Controller definiert werden, werden auch Actions genannt. Üblicherweise wird eine Action Datenbankaktion veranlassen (Objektmanipulation) und am Ende ein Modell erstellen, welches an eine bestimme View zur Darstellung weiter gegeben wird. Die typischen Actions sind hierbei zunächst jene, die die CRUD (create – read – update – delete) Funktionalität herstellen. Genannt werden sie create, save, list, update, show und edit.

Wichtig ist noch das params Objekt. Auch hierbei handelt es sich um eine Hashmap, welche wie andere (z.B. session oder flash) automatisch zur Verfügung steht. Das params Objekt enthält alle Parameter, die der URL zu der Action übergeben wurden, z.B. /site/controller/action?id=5 und man erhält die ID über params.id. Die anderen genannten Objekte dienen zur Haltung von Daten in unterschiedlichem Scope. Session gilt die gesamte Session lang, während flash für diese und den nächsten Request gilt.

2.4 Business-Logik durch Service-Klassen

Auch wenn man oft dazu neigen wird Business-Logik direkt im Controller zu definieren, so ist dies doch schlechter Stil. Besser ist es solchen Code auszulagern. Dies wird einem in GRAILS sehr einfach gemacht. Das Artefakt Service besteht aus Klassen welche auf Service enden und in GRAILS-APP/Service liegen. Diese Methoden werden von GRAILS als Singleton instanziiert (liegen also nur ein einziges mal vor) und sind dank des Spring-Features Dependency Injection überall verfügbar. Man muss in einer Methode lediglich einen Service deklarieren, um die benötigte Instanz verfügbar zu haben. Beispiel:

```
def springSecurityService
def someClosure = {
        springSecurityService.principal() // das funktioniert ohne weiteres
}
```

Damit kann man Code auslagern ohne sich später sorgen zu müssen, wie man an einer bestimmten Stelle wieder Zugriff darauf erlangt.

2.5 Die View wird generiert durch GSP und TagLibs

Von den drei Ebenen des MVC-Patterns fehlt noch die View-Ebene. Diese wird durch HTML Seiten gestellt, welche durch Servercode ausstaffiert werden. Man nennt sie Groovy Server Pages (GSP) und sie lehnen sich vom Konzept her eng an Java Server Pages (JSP) und noch enger an verwandte Techniken wie z.B. icefaces an. Die Idee ist normalen HTML Code mit Platzhaltern zu verknüpfen, welche vom Server zu gegebener Zeit gefüllt werden. Dies geschieht einerseits dadurch, dass direkt Servercode ausgeführt werden kann:

```
<% println „Hallo Welt“ %>
```

Andererseits durch spezielle Tags, z.B.

```
<g:createLink controller=“project“ action=“list“/>
```

Solche Tags können ganz leicht selbst definiert werden, so dass man Code in beliebig vielen GSPs wiederverwenden kann und diese auch kurz und übersichtlich hält. Dazu mehr im nächsten Kapitel.

In den GSPs stehen einem auch Daten zu Verfügung. Im Kapitel über Controller wurde beschrieben, wie eine Action eine View generiert und ihr dabei ein Modell zur Verfügung stellt. Dies geschieht in Form einer Hashmap. Alles was in dieser Hashmap gespeichert wird, steht später auf View-Seite zur Verfügung. So erhält z.B. eine show.gsp (nehmen wir als Beispiel ein einfaches Gen) die darzustellende Geninstanz [geneInstance: Gene.get(id)]. In der View greifen wir darauf zu über den ${} Operator oder direkt in der <% %> Notation. So z.B.

```
<p>Sequence: ${geneInstance.sequence} </p>
```

Das Layout der GSPs wird durch Sitemesh gesteuert. Es gibt in GRAILS-APP/WEB-APP/layouts/ eine main.gsp, welche das zentrale Layout bereitstellt. Alles was hier definiert wird steht später allen davon abgeleiteten Seiten offen. So bietet es sich hier an ein Layout festzulegen und Ressourcen zu importieren.

Um auch GSP-Fragmente wiederverwendbar zu machen und um an Übersichtlichkeit zu gewinnen, können Templates erstellt werden. Diese werden auch unter /GRAILS-APP/WEB-APP/layouts gespeichert und müssen mit einem '_' beginnen (z.B. _header.gsp). Diese können dann in anderen GSPs jederzeit integriert werden.

2.6 TagLibs

GRAILS gibt für alle wichtigen Funktionen einen reichen Schatz an TagLibs vor. Da es sich bei einer TagLib um ein weiteres Artefakt handelt, können einfach eigene TagLibs hinzugefügt werden. Eine TagLib ist eine Klasse, die auf den Namen TagLib endet und in GRAILS-APP/TagLib gespeichert wird. In ihr kann eine beliebige Anzahl von Tags in Form von Closures generiert werden. Sofern nicht anders vorgegeben gehören diese dann dem Default-Namespace (g:) an.

2.7 Dynamic Scaffolding

Speziell zu Beginn der Entwicklung gibt einem ein bestimmtes Feature von GRAILS ein Hochgefühl. Durch dynamic Scaffolding wird Standard-Code automatisch zur Laufzeit erzeugt. Dies gilt sowohl für Controller als auch für GSPs. Hat man beispielsweise seine DomainClass Gene erzeugt und möchte diese nun im Browser bearbeiten, so muss man lediglich eine beliebige Controllerklasse anlegen, welche so aussieht:

```
class GeneController
{
        def scaffold = Gene
}
```

Das ist auch schon alles. Startet man die GRAILS-Applikation, so stehen einem alle CRUD-Operationen zur Verfügung. Man kann es selbst im Browser ausprobieren: http://applicationName/gene

Durch das Kommando „generate-templates“ kann man seinem Projekt auch die Templates hinzufügen, welche für das Scaffolding verwendet werden. Um die Flexibilität des OpenLaboratoryFrameworks zu erhalten, wurden außer einigen eigenen Templates keine GSP-Seiten erzeugt. Statt dessen wurde der Scaffolding Code so erweitert, dass alle gewünschte Funktionalität beliebigen DomainClass Objekten offen steht.

2.8 Erweiterbarkeit durch Plugins

Ein weiteres zentrales Feature von GRAILS ist die Erweiterbarkeit. Das Framework lebt durch die Community, die es unterstützt. Es gibt Plugins für quasi jeden denkbare Aufgabe. Bevor man also selbst los programmiert, sollte man sich nach einem passenden Plugin umschauen. Dadurch, dass diese oft relativ simpel gebaut sind, kann es auch schon Arbeit sparen ein Plugin für die eigenen Zwecke anzupassen. Einziges Manko ist, dass viele Plugins lange nicht gewartet wurden und es deshalb einiger Recherche benötigt, es zum Laufen zu bekommen.

Die Vielfalt erscheint dabei grenzenlos: Es gibt Plugins, die JavaScript-Bibliotheken wie YUI oder andere kapseln, Paypal-Support, ein Plugin welches Test-Daten erzeugt, eines das bei der Datenbankmigration hilft, eines für Import / Export und viele mehr. Das nützlichste aller Plugins ist jedoch das spring-security Plugin, welches GRAILS das sehr gute, aber üblicherweise auch sehr komplizierte Spring-Security-Framework leicht zugänglich macht und eine Applikation in wenigen Minuten mit einem sehr zuverlässigen Sicherheitsrahmen und einem Cookie-basierten Benutzermanagement ausstattet.

Es ist auch durchaus eine Überlegung wert, selbst Plugins zu erstellen, um der eigenen Applikation durch geschickte Modularisierung noch mehr Wiederverwendbarkeit und Übersichtlichkeit hinzuzufügen. Der Plugingedanke war eine der zentralen Ideen hinter dem OpenLaboratoryFramework, welcher durch GRAILS Rechnung getragen werden konnte.

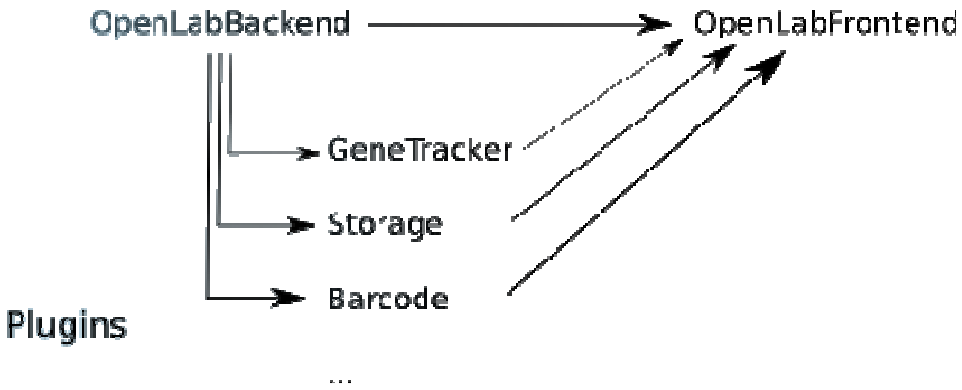
3 OPENLABORATORYFRAMEWORK

Im folgenden Kapitel sollen einige zentrale Gedanken hinter dem OLF erläutert werden. Vor allem soll gezeigt werden, inwiefern Kernaspekte von GRAILS gezielt zum Einsatz gebracht werden, um die Ziele, die zu Beginn des Projekts erarbeitet wurden, zu erreichen.

3.1 Grundlegender Aufbau (Plugins)

Das OLF macht sich den Pluginmechanismus von GRAILS zunutze, um einerseits eine Trennung von Backend und Frontend zu erzielen, und um andererseits die modulare Erweiterbarkeit zu gewährleisten. Die einzelnen Projekte sind dabei folgendermaßen organisiert:

Die Business-Logik und das Modell sind im OpenLabBackend zusammengefasst. Die verschiedenen Plugins, die dem Framework zusätzliche Funktionalität verschaffen benutzen das Backend und vor allem dessen Interfaces als Grundlage. Das OpenLabFrontend schließlich vereint BackEnd und Plugins und fügt dem ganzen eine Grundoberfläche hinzu. Auch die einzelnen Plugins können dabei Frontendelemente beisteuern, die für deren Funktion von Bedeutung sind.



3.2 Erweiterung des Inhalts durch Modules

Da das OLF beliebig erweiterbar sein sollte war es nötig einen Mechanismus zu finden, der dies gewährleistet. Dabei muss folgendes berücksichtig werden:

Einerseits darf bestehender Applikationscode nichts von zukünftigen Erweiterungen wissen, d.h. es muss möglich sein auch unvorhergesehen Erweiterungen schreiben zu können.

Ferner muss es einem Plugin möglich sein auf benötigte Daten des Modells ohne weiteres zugreifen zu können. Dieser Punkt löst sich jedoch durch den Einsatz von GORM und den dynamischen Methoden von selbst auf, da alle DomainClasses, die in der Gesamtapplikation vorhanden sind auch überall zur Verfügung stehen.

An dritter Stelle steht hier der schwierigste Punkt: Es muss möglich sein neue Inhalte ins Frontend einzufügen ohne dass die bestehenden GSP Seiten dafür angepasst werden müssen. Um dieses Problem zu lösen wird das Interceptor-Pattern umgesetzt. Demnach werden verschiedene Stellen im Scaffolding-Code mit TagLibs besetzt, deren Aufgabe es ist die entsprechenden Inhalte zur Laufzeit zu bestimmen und einzusetzen.

Gleichzeitig müssen diese TagLibs dann in der Lage sein zu ermitteln, welche Zusatzinhalte an welcher Stelle anzubringen sind und woher diese stammen sollen. Darüber hinaus muss den dynamischen Inhalten auch das richtige Modell angeboten werden, um die korrekte Darstellung zu ermöglichen.

Um diesen Problemen zu begegnen wurde folgender Mechanismus implementiert:

Es wurde ein neues Artefakt erstellt, welches Klassen erkennt, die auf den Namen Module enden und im Ordner GRAILS-APP/module liegen. Zusätzlich wurde eine Service-Klasse namens ModuleHandlerService erstellt, welche den Zugriff auf diese Klassen ermöglicht. Es wurden nun verschiedene Module-Arten als Interface bereitgestellt. Die Klasse bestimmt das Einsatzgebiet des Moduls. So gibt es im Moment ein Interface für Klassen, die Templates bereitstellen, eines für Klassen, die dem Menü zusätzliche Punkte bereitstellen und eines für das Bereitstellen von Addins. Im Folgenden werden die Module im einzelnen beschrieben.

3.2.1 TabModules

TabModules liefern Inhalte für die Tabs, welche in show.gsp angezeigt werden. Die Idee hinter den Tabs ist es zu der angezeigten Domain Instanz entweder zusätzliche Daten anzuzeigen oder zusätzliche Funktionalität anzubieten. Als Beispiele sei die Erstellung von Cell Viability Daten für Gene oder die Darstellung eines NCBI-Sequence Browsers genannt. Module, die diese Aufgabe erfüllen wollen, müssen das Interface „org.openlab.module.Module“ implementieren. TabModules bekommen eine DomainClass übergeben und müssen melden, ob sie dazu Inhalten beitragen wollen (Inversion of Control). Ist dem so, so müssen Sie auch ein Template und ein Model zum Rendern des Tabs beitragen. Dies wird durch das Interface realisiert, wie folgendes Beispiel zeigt:

```
package org.openlab.module.tab

import org.openlab.module.*;

class CellLineDataModule implements Module{

	def getPluginName() {
		"gene-tracker"
	}

	def getTemplateForDomainClass(def domainClass)
	{
		if(domainClass == "gene") return "cellLineDataTab"

		else return null
```

```
	}

	def isInterestedIn(def domainClass, def type)
	{
		if((type == "tab") && (domainClass == "gene")) return true
		return false
	}

	def getModelForDomainClass(def domainClass, def id)
	{
		if(domainClass == "gene")
		{
			return []
		}
	}
}
```

3.2.2 MenuModules

Zum Header der Applikation gehört neben dem Logo auch ein Menü. Dieses kann durch beliebige Inhalte erweitert werden. Dieses Modul muss das Interace „org.openlab.module.MenuModule“ implementieren. Der Groovy XML Builder wird benutzt, um das eigentliche Menü zu erstellen. Eine weitere Methode gibt die Priorität wieder. Dadurch kann gesteuert werden, ob der Menüeintrag eher links oder eher rechts erscheint. Die Implementierung lässt sich dabei an folgendem Beispiel ablesen:

```
package org.openlab.module.menu

import org.openlab.module.*;
import groovy.xml.MarkupBuilder

class CellLineMenuModule implements MenuModule{

	def getPriority()
	{
		8
	}

	def getMenu()
	{
		def writer = new StringWriter()
		def xml = new MarkupBuilder(writer)
		def controller = "cellLineData"

		xml.root
		{
			submenu(label: 'Cell Line Data')
			{
				menuitem(controller: controller, action: 'create',
label: 'Create Cell Line Data')
```

```
                        menuitem(controller: controller, action: 'list',
label: 'List Cell Line Data')
                    }
            }

            return writer.toString()
      }
    }
```

3.2.3 AddinModules

Die rechte Leiste der Oberfläche ist Erweiterungen in der Art von Portlets vorbehalten. Bei Portlets kann es sich um beliebigen Code handeln, der auch in GSPs eingesetzt wird. Deshalb ist hier ein Template anzugeben. Es muss dabei darauf geachtet werden, dass die Maße für die Addins nicht überschritten werden. Das folgende Beispiel zeigt die Modulimplementierung einer einfachen Legende:

```
package org.openlab.module.addin

import org.openlab.module.*;

class GeneLegendAddinModule implements AddinModule{

      def getName()
      {
            "gene legend"
      }

      def getTemplate()
      {
            "geneLegendAddin"
      }

      def getPluginName()
      {
            "gene-tracker"
      }

    }
```

3.2.4 OperationsModules

Jede show.gsp verfügt auf der rechten Seite über eine Anzahl gerenderter Boxen. Diese beinhalten Informationen über die History oder die Zugriffsrechte. Eine Box – Operations genannt – bietet Zusatzhandlungen an, die zu der angezeigten DomainClass Instanz ausgeführt werden können. So wird standardmäßig die Option der Bookmarkerstellung angeboten. Spezielle Optionen können hinzugefügt werden, so z.B. das erzeugen eines mutierten Gens oder eines Silencing Konstrukts. Auch hierzu wird ein Template benötigt, welche den entsprechenden HTML Code rendert. Beispiel:

```
package org.openlab.module.operations

import org.openlab.module.*;
import org.openlab.genetracker.*;

class GeneOperationsModule implements Module{

	def getPluginName() {
		"gene-tracker"
	}

	def getTemplateForDomainClass(def domainClass)
	{
		if(domainClass == "gene") return "geneOperations"
	}

	def isInterestedIn(def domainClass, def type)
	{
		if((type == "operations") && (domainClass == "gene")) return
true
		return false
	}

	def getModelForDomainClass(def domainClass, def id)
	{
		if(domainClass == "gene")
		{
			def gene = Gene.get(id)
			[geneInstance: gene]
		}
	}
}
```

3.3 Erstellung eines Dokumenten-Plugins

Hier soll eine Schritt-für-Schritt-Anleitung an die Hand gegeben werden, die einem Entwickler zeigt, wie das bestehende Framework mit einem neuen Plugin um Inhalte erweitert werden kann. Als Beispiel dient hier die Erstellung eines Rahmen-Plugins „OpenLabDocuments“, welches es in seiner fertigen Version erlauben soll, an DataObjects Dokumente anzuhängen.

Dazu empfiehlt sich der Einsatz einer dokumentenorientierten Datenbank wie CouchDB. Dieses Kapitel soll jedoch nur die Schnittstelle einer solchen Logik- und Persistierungsimplementierung und dem Framework beschreiben.

Ziel ist die Programmierung eines Addins, welches bereits angehängte Dokumente für das gezeigte DataObject zeigt und das Hinzufügen neuer Dokumente per File-Upload erlaubt.

3.3.1 Erstellung des leeren Plugin

In SpringSource STS kann ein neues Grails-Plugin-Projekt angelegt werden. In diesem Fall nennen wir es „OpenLabDocuments“. Was noch fehlt ist der Ordner, der später die Modules enthalten soll. Deshalb legt man den Ordner „grails-app/modules“ an und fügt diesen als Source-Folder hinzu. Um ein neues Plugin mit dem Framework zu verknüpfen, gibt es zwei Möglichkeiten:

1. Installation als reguläres Plugin. Dafür muss im bestehenden Plugin „package-plugin“ und im OpenLabFrontend „install-plugin“ ausgeführt werden.

2. Erstellung eines Inplace-Plugins. Der Vorteil liegt darin, dass stets die aktuelle Version des Plugins benutzt wird und dem Entwickler viel Zeit gespart werden kann. Hierbei muss das Plugin jedoch von Hand registriert werden. Dazu öffnet man die BuildConfig.groovy im OpenLabFrontend und fügt eine neue Zeile ein:

   ```
   grails.plugin.location.documents = „../OpenLabDocuments“
   ```

3. Entsprechend muss auf das Backend verwiesen werden und zwar in der BuildConfig.groovy des Plugins:

   ```
   grails.plugin.location.backend = „../OpenLabBackend“
   ```

3.3.2 Erstellung des Addins

Wir legen ein neues Package „org.openlab.module.addin“ in grails-app/modules an. Dann fügen wir eine neue Groovy-Klasse „DocumentsAddinModule.groovy“ hinzu. Diese Klasse sollte etwa so aussehen und das Interface AddinModule.groovy implementieren:

```
package org.openlab.module.addin

import org.openlab.module.*;

class DocumentsAddinModule implements AddinModule {

	def getName()
	{
		"documents"
	}

	def getTemplate()
	{
		"attachedDocuments"
	}

	def getPluginName()
	{
		"open-lab-documents"
	}
}
```

Die Modulimplementierung nennt dem ModuleHandlerService den Namen des eigenen Plugins, sowie den Namen eines Templates. Diese Informationen genügen, um das korrekte Template zu lokalisieren und zu rendern. Bei Addins ist die Übergabe eines Models z.Zt. nicht vorgesehen. Es besteht jedoch ein Javascript-Event-Mechanismus, für welchen sich ein Addin registrieren kann, um herauszufinden, was gerade in der Hauptansicht gezeigt wird. Zunächst folgt die Implementierung des Templates, welches sich unter „grails-app/views/addins/_attachedDocuments.gsp“ findet.

```
<div id="documentsAddin">
<h1>Attached Documents</h1>

<p style="padding: 10px;">This addin is supposed to show attached
documents.</p>

</div>

<script type="text/javascript">
	var updateDocumentsAddin = function(type, args, me)
	{
```

```
            ${remoteFunction(params:
"\'name=\'+args[0]+\'&className=\'+args[1]+\'&id=\'+args[2]",
controller:"documentsAddin", action:"renderDocumentsAddin",
update:[success: "documentsAddin", failure: "documentsAddin"])}
      };
      var subscribers = olfEvHandler.bodyContentChangedEvent.subscribers;
      var contains = false;
      for(var i=0; i < subscribers.length; i++)
      {
            if(subscribers[i] == this) contains = true;
      }
      if(!contains)
      {

olfEvHandler.bodyContentChangedEvent.subscribe(updateDocumentsAddin, this);
             contains = false;
      }
    </script>
```

Die GSP enthält nicht viel mehr als ein Tag. Interessant ist an dieser Stelle der javascript Teil. Hier wird – falls nicht bereits geschehen – beim OlfEvHandler (Der Event-Kontroll-Instanz) eine Listenermethode für das Event „bodyContentChanged“ registriert. Dieses Event wird immer dann getriggert, wenn die Hauptansicht ein neues DataObject zeigt. Mit diesem Event werden Klassenname und ID übergeben.

Die registrierte Funktion „updateDocumentsAddin“ löst nun einen AJAX-Aufruf des DocumentsAddinController aus. Dort bekommt die Methode „renderDocumentsAddin“ die Parameter Klassenname und ID übergeben und kann den neuen Inhalt des Addins rendern, da als Ziel das zuvor erzeugte Tag gegeben ist. In der Beispielimplementierung ist der Controller noch leer und gibt nur die beiden Parameter über Konsole aus:

```
package org.openlab.documents

class DocumentsAddinController {

      def renderDocumentsAddin = {
            println "Documents:" + params.className
            println "Documents:" + params.id


            render "No document support so far."
      }
```

4 ÖFFENTLICHE PLUGINS, DIE VERWENDET WERDEN

Zur Erstellung des OLF wurde eine ganze Reihe öffentlicher Plugins verwendet, welche verschiedenes vereinfacht haben. Im Folgenden sind die wichtigsten aufgeführt. Seit Grails 1.3.x ist es möglich Pluginklassen zu überschreiben. Die überschriebenen Klassen sind dann im Projekt selbst aufgeführt. Erkennbar sind diese daran, dass der Eclipse-Compiler vor Code-Duplikation warnt (an dieser Stelle hilft es, die nicht genutzten Pluginklassen vom BuildPath auszuschließen).

4.1 Spring-Security Plugin

Beim Spring Security Plugin handelt es sich um eine Weiterentwicklung des Acegi-Plugins. Das recht schwer zu handhabende Security-Framework wird hier auf einfach zu nutzende Art gekapselt. Bei der Installation des Plugins wird ein rudimentäres Sicherheitssystem vorgegeben. Dieses lässt sich über die Config.groovy konfigurieren.

In der gegenwärtigen Version werden RequestMaps verwendet, um den Zugriff zu regulieren. Dazu werden für eine Tabelle RequestMap Einträge angelegt, welche den Zugriff auf einzelne URLs gestatten oder verbieten. Die RequestMaps werden in der Bootstrap.groovy generiert.

Die Unterstützung für Gruppen ist vorgesehen, aber im Framework noch nicht genutzt. Ebenso die Unterstützung von ACLs.

4.2 GRAILS-UI und RICH-UI

Diese Plugins liefern YUI2-Komponenten als TagLibs verpackt und fügen damit einfach nutzbare Javascript und Ajaxkomponenten hinzu. Beide Plugins werden öfters eingesetzt, u.a. für die Generierung von Bäumen, Menüs und dynamischen Tabellen. In beiden Plugins wurden Klassen überschrieben.

4.3 Searchable

Dieses Plugin kapselt Apache Lucene und fügt damit Volltextsuche hinzu. Um festzulegen, welche Domain-Klassen in der Suche auftauchen sollen, muss der jeweiligen Domain-Klasse ein Attribut „static searchable“ hinzugefügt werden. Hier können über die Eigenschaften „component“ oder „reference“ auch Beziehungen angegeben werden, so dass z.B. bei der Suche nach einem Projekt auch die jeweiligen Gene aufgeführt werden.

4.4 Export

Das Export Plugin bietet die Möglichkeit Listen in verschiedenen Formaten zu exportieren. Teile des Plugins werden auch genutzt, um beispielsweise den Export einzelner Storage-Boxen zu ermöglichen.

4.5 Excel-Import

Dieses Plugin wurde für die Datenmigration eingesetzt. Es dient dazu auf sehr einfach Art und Weise Excel-Sheets in Groovy-Listen zu übersetzen, mit Hilfe derer z.B. Domain-Objekte erstellt werden können.

4.6 Filter-Plugin

Das Filter-Plugin ermöglicht es die in einer Liste angezeigten Treffer nach nicht-assoziativen Kriterien zu filtern. So kann beispielsweise das Datum (LastUpdate) eingeschränkt werden.

4.7 Modalbox-Plugin

Dieses Plugin erlaubt wie der Name schon sagt die Erstellunge von Modal Boxes, d.h. von Dialogen, die über die Website gelegt werden und die Aufmerksamkeit des Nutzers fordern. Hierin können beliebige Inhalte dargestellt werden. Ein Beispiel für den Einsatz liefert die Zuweisung von Projekten zu DataObjects wie z.B. Genen.

5 SettingsService

Die Konfiguration der Applikation ist separat in einer HSQLDB Datenbank-Instanz hinterlegt. Um den Zugriff zu dieser separaten Datenquelle zu ermöglichen, wurde ein Service namens SettingsService eingeführt. Dieser bietet die Möglichkeit

- allgemeine Einstellungen zu speichern (settingsService.getSetting(key: „key“, settingsService.setSetting(key: „key“, value: „value“)
- nutzerspezifische Einstellungen zu speichern (settingsService.getUserSetting(key: „key“), set analog)
- Standard-Einstellungen zu speichern (settingsService.getDefaultSetting(key: „key“), set analog)
- Sprachlabels abzurufen (settingsService.getLabel(key: „key“, settingsService.getLabelInLanguage(key: „key“, language: „DE“))

6 Weiterführende Literatur

- David Klein: Grails – A Quick-Start Guide
- Glen Smith, Peter Ledbrook: Grails In Action
- Robert Fischer: Grails Persistence with GORM and GSQL
- Christopher M. Judd u.a.: Beginning Groovy and Grails (From Novice to Professional)
- Graeme Rocher and Jeff Brown: The Definite Guide To Grails

Das letztgenannte Buch ist hierbei Pflichtlektüre, da es einen guten Einstieg bietet und vom Chef-Entwickler von Grails selbst stammt. Für einen Schnelleinstieg ist tatsächlich „A Quick-Start Guide“ geeignet, in dem einem auch Groovy näher gebracht wird.