Tutorial und Dokumentation

1 Entwicklungsumgebung einrichten

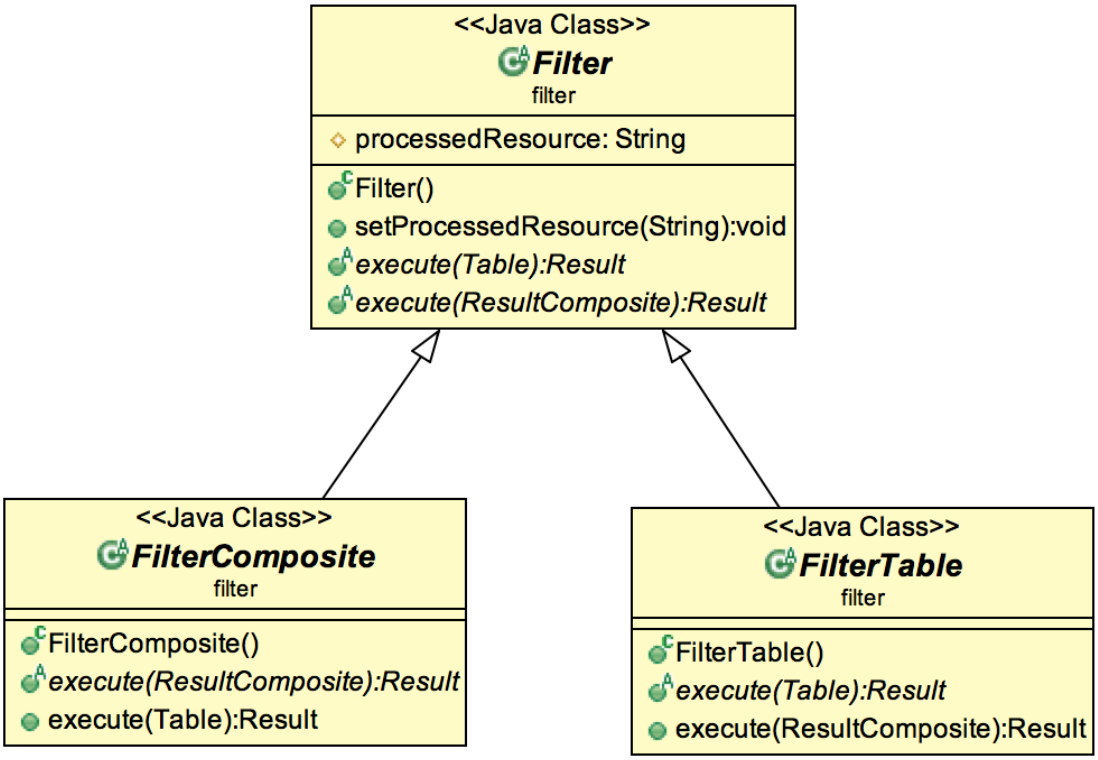
Um Filter entwickeln zu können, müssen zunächst eine entsprechende Entwicklungsumgebung eingerichtet und das Projekt importiert werden.
Wir verwenden Springsource, eine Eclipse-Erweiterung für Spring/Grails, und würden empfehlen dies ebenfalls zu tun, wobei es prinzipiell auch möglich ist, andere Umgebungen zu benutzen. Die Benutzung ist damit sehr ähnlich zu Eclipse und wir gehen davon aus, dass Eclipse bekannt ist.
Springsource ist hier erhältlich: http://www.springsource.org/downloads/sts (Anmeldung ist nicht notwendig, einfach unten auf „I'd rather not fill in the form. Just take me to the download page“) klicken.
Sobald Springsource entpackt und gestartet ist sollten noch im Dashboard (falls nicht sichtbar: Help -> Dashboard) unter Extension „Grails“, „Grails Support“ und „Groovy Eclipse“ installiert werden.
Jetzt das Projekt (LDB) mit dem SVN-Programm der Wahl (Subversion in unserem Fall) auschecken.
Die Filter liegen bisher in src/java im Package filters. Konfigurationsdateien liegen im Config-Ordner in webapp. Die gewünschten Filter/Renderer-Konfigurations-XML-Dateien und andere Einstellungen werden in der appconfig.xml ausgewählt. Alle bisherigen Filter-XML liegen hierbei im Ordner FilterFileFolder, die Renderer-XML im RendererFileFolder. Mittels run-app (z. B. Rechtsklick auf das Projekt -> run as -> 2 Grails Command (run-app) lässt sich das Ergebnis im Browser überprüfen.

2 Filter

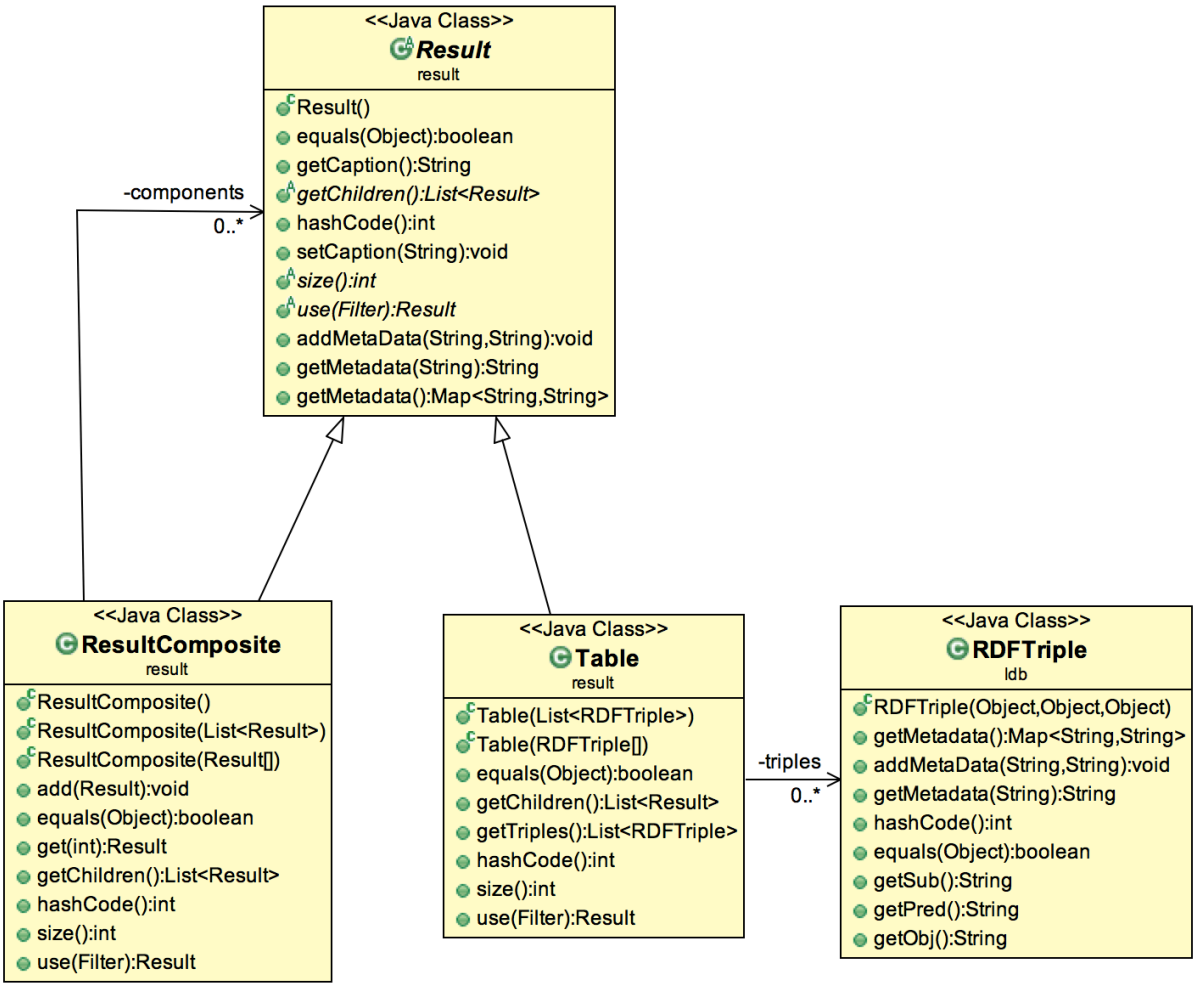
2.1 Filter schreiben

2.1.1 Einleitung

Filter sind ein integraler Bestandteil des Frameworks. Es gibt zwei Typen von Filtern: Tabellenfilter und Baumfilter. Tabellenfilter ermöglichen es, Listen von RDF-Tripeln beliebig zu manipulieren und stellen damit die wichtigste Klasse dar. Baumfilter arbeiten auf Clustern von Ergebnissen. Konkret handelt es sich bei Filtern um Implementierungen der abstrakten Klasse Filter, also insbesondere der „Result execute(Table table)“ bzw „Result execute(ResultComposite resultComposite)“-Methode:



Daneben besteht Filter nur noch aus einem Setter für die aktuell bearbeitete Ressource.
Jetzt zunächst einige Erklärungen zu den beteiligten Klassen:
Die interne Datenstruktur implementiert ein Composite-Pattern mit Result als Komponente, ResultComposite als Kompositum und Table als Blatt.
Bei Table handelt es sich um eine Liste von RDFTriple. RDF-Triple wiederrum ist eine einfache Klasse, die drei Strings, Subjekt, Prädikat und Objekt, hält. ResultComposite, auch Cluster genannt, können dann beliebig viele Tabellen oder weitere ResultComposites beinhalten.
Dazu ein Klassendiagramm:



Die wichtigste Methode in Result ist use(Filter). Falls Filter ein FilterComposite ist, wird er direkt auf das Result angewendet, falls es ein FilterTable ist, werden alle Tabellen in dem Teilbaum damit gefiltert.
Per getChildren kommt man an die direkten Kinder, also die Tabellen und weitere ResultComposites, die dieses Result beinhaltet und per size kann man deren Zahl abfragen.
Result und RDFTriple implementieren das Metadata-Interface mit dem sich Metadaten in Form von String-String-Paaren hinzufügen lassen.
Result implementiert dazu noch Taggable mit dem sich Results taggen lassen und get/setCaption, um Überschriften hinzuzufügen.
Das Tripel in RDFTripel und die Liste der Tripel einer Table sind immutable, die Metadaten(„Metadata“, Tags und Captions) allerdings nicht. Daher ist es beim Verarbeiten von Tabellen mit Filtern wichtig zu beachten, dass man neue Tabellen erstellt, falls man mit Metadata/Tags/Captions arbeitet, um nicht versehentlich die Metadaten der als Argument erhaltenen Tabellen zu überschreiben.

2.1.2 Erstes Beispiel – Table zurückgeben

Betrachten wir nun ein einfaches Beispiel für eine komplette Filter-Klasse:
`DeduplicateFilter.java:`

```java
package filters;

import java.util.ArrayList;
import java.util.List;

import ldb.RDFTriple;
import result.Result;
import result.Table;
import filter.FilterTable;

/**
 * removes duplicates from the table
 *
 */
public class DeduplicateFilter extendss FilterTable {

	@Override
	public Result execute(Table table) {
		List<RDFTriple> deduplicated = new ArrayList<RDFTriple>();
		for (RDFTriple triple : table.getTriples()) {
			if (!deduplicated.contains(triple))
				deduplicated.add(triple);
		}
		return new Table(deduplicated);
	}
}
```

Der Filter bekommt eine Table, erstellt eine neue Liste von Tripeln, fügt dort jedes Tripel aus der Table genau einmal hinzu und gibt dann eine neue Table mit der Liste zurück.

2.1.3 Zweites Beispiel – ResultComposite zurückgeben und Tags setzen

Betrachten wir nun ein Beispiel (jetzt nur noch die execute-Methode), bei dem ein ResultComposite zurückgegeben wird:

HalfFilter.java:

```java
public Result execute(Table table) {
		int halfSize = table.size() / 2;
		List<RDFTriple> firstHalf = new ArrayList<RDFTriple>(halfSize);
		List<RDFTriple> secondHalf = new ArrayList<RDFTriple>(halfSize);

		// split table into two lists
		for (int i = 0; i < halfSize; i++) {
			firstHalf.add(table.getTriples().get(i));
			secondHalf.add(table.getTriples().get(i + halfSize));
		}
		// if the table has an odd number of entries
		if (table.size() % 2 != 0)
			// add last entry to second half
			secondHalf.add(table.getTriples().get(table.size() - 1));

		// create ResultComposite from the two tables
		ResultComposite ResultComposite = new ResultComposite();
		ResultComposite.add(new Table(firstHalf));
		ResultComposite.add(new Table(secondHalf));

		// add Tags
```

```
		ResultComposite.get(0).addTag("first");
		ResultComposite.get(1).addTag("second");

		return ResultComposite;
	}
```

Der Filter spaltet eine Tabelle in zwei Hälften auf und gibt sie als ResultComposite zurück.
Das sind die zwei möglichen Rückgabeklassen: eine elementare Tabelle oder ein ResultComposite.
Außerdem zu sehen ist hier das Setzen von Tags, auf die sich dann beim Anwenden der Filter prüfen lässt, siehe „Filter anwenden“.

2.1.4 Drittes Beispiel – Boolean-Attribut

Es ist möglich, das Verhalten der Filter durch Attribute zu steuern. Ein Beispiel dazu:
SortFilter.java:

```
private boolean ascending = true;

	public void setAscending(boolean ascending) {
		this.ascending = ascending;
	}

	@Override
	public Result execute(Table table) {

		List<RDFTriple> solutions = Utils.cloneTriples(table);
		Collections.sort(solutions, new Comparator<RDFTriple>() {
			@Override
			// implementation of Comparator on RDFTriple
			public int compare(RDFTriple o1, RDFTriple o2) {
				int res;
				// sort first by object
				res = o1.getObj().compareTo(o2.getObj());
				if (res == 0) // both objects identical
					// then by subject
					res = o1.getSub().compareTo(o2.getSub());
				if (res == 0)	// both subjects identical
					// then by predicate
					res = o1.getPred().compareTo(o2.getPred());
				// if not ascending reverse order
				return ascending ? res : res * -1;
			}
		});
		return new Table(solutions);
	}
```

Der Filter sortiert eine Tabelle.
Das Attribut ist hier „boolean ascending“, um zu steuern, ob auf- oder absteigend sortiert werden soll. Die Attribute werden in der XML-Konfiguration des Filters (mehr dazu im nächsten Abschnitt) spezifiziert und können intern dann wie gewohnt verwendet werden. Allgemein sind als Attribute sämtliche primitiven Typen sowie String möglich.

2.1.5 Viertes Beispiel – String-Attribut

Ein anderes Beispiel wäre:

LanguageFilter.java:

```
private String lang = "";

	public void setLang(String lang) {
		this.lang = lang;
	}

	@Override
	public Result execute(Table table) {
		List<RDFTriple> solutions = table.getTriples();
		ArrayList<RDFTriple> filtered = new ArrayList<RDFTriple>();
		for (RDFTriple qs : solutions)
			if (qs.getObj().endsWith(lang)) // check for suffix
				filtered.add(qs);

		return new Table(filtered);
	}
```

Der Filter selektiert Tripel, deren Objekt ein bestimmtes Suffix haben, um nach Sprachen filtern zu können. Das Suffix bzw. die Sprache werden durch den „String lang“ angegeben.

2.1.6 Baumfilter

Baumfilter verwenden die FilterComposite-Klasse und können, wie der Name schon andeutet, auf ganzen Bäumen arbeiten. Damit sind sie insbesondere geeignet, um Teilbäume abzuschneiden. Ein Beispiel dazu:

```
package filters;

import java.util.ArrayList;
import java.util.List;

import result.Result;
import result.ResultComposite;
import filter.FilterComposite;

/**
 * Removes all clusters with the given tag from the tree.
 *
 */
public class RemoveClusterFilter extends FilterComposite {
	/**
	 * tag of clusters to remove
	 */
	protected String removeTag;

	public void setRemoveTag(String tag) {
		removeTag = tag;
	}

	@Override
	public Result execute(ResultComposite composite) {
		if (composite.hasTag(removeTag))
			return null;

		removeResult(composite);
```

```
            return composite;
        }

        private void removeResult(Result result) {

            List<Result> childrenToRemove = new ArrayList<Result>();

            for (Result child : result.getChildren())
                if (child.hasTag(removeTag))
                    childrenToRemove.add(child);
                else
                    removeResult(child);
            result.getChildren().removeAll(childrenToRemove);
        }
}
```

Die Funktionsweise ist im Prinzip die selbe wie beim Tabellenfilter, bloß dass als Argument ein Teilbaum in Form eines Kompositum zur Verfügung gestellt wird. In diesem Fall wird dann rekursiv durch den Baum gelaufen und alle Komponenten, die den spezifizierten Tag haben, entfernt.

2.2 XML-Konfiguration der Filter

Um die Filter in den Browser einzubinden benutzt man eine XML-Konfigurationsdatei.

2.2.1 Grundgerüst

Das Root-Element dieser Datei heißt filters. In diesem filters-Element werden nun die Filter in gewünschter Reihenfolge mittels Filter-Elementen angegeben. Ein Filter wird hierbei eindeutig identifiziert durch einen vollqualifizierten Klassennamen, der im filter-Attribut „classname“ angegeben wird. Insgesamt sieht das Gerüst also so aus:

```
<filters name="beliebiger Name">
        <filter classname="Pfad zu diesem Filter"/>
        <filter classname="Pfad zu diesem Filter"/>
        …
</filters>
```

2.2.2 Überschriften vergeben

Zusätzlich kann man den Filtern, also den filter-Elementen noch einige zusätzliche Anforderungen mitgeben. Eine simple ist, der/den durch den Filter entstehenden Tabelle/n eine Überschrift für die Anzeige zu geben. Das geschieht durch ein optionales filter-Attribut „caption“. Ein filter-Element sieht dann so aus:

```
        <filter classname="Pfad zu diesem Filter" caption="beliebige Überschrift/>
```

2.2.3 Parameter setzen

Weiterhin muss man, falls die Filter Parameter haben, wie beispielsweise der LanguageFilter aus Abschnitt 2.1.5, diese in einem parameters-Element als Kind des jeweiligen filters-Element mitgeben. Hierbei gibt man dem parameters-Element jeweils als Kind mit dem Namen „parameter“ den jeweiligen Parameter mittels des Atributs „name“ und des Atributs „value“ an. Für den LanguageFilter, angenommen er läge im Projekt im Package filters, würde, falls dieser die englischen Einträge herausfiltern soll, die Parametervergabe also folgendermaßen aussehen:

```
<filter classname=“filters.LanguageFilter“>
        <parameters>
                <parameter name=“lang“ value=“en“/>
        </parameters>
</filter>
```

Hierbei ist speziell darauf zu achten, dass der Name des zu setzenden Parameters (hier „lang“) genau dem Namen des zu setztenden Parameters in der Filter-Java-Klasse entspricht also hier gegeben durch die Zeile „String lang =““;“ und zudem der Datentyp passt (also hier „en“ auch wirklich ein String ist).

2.2.4 Mehrere Parameter setzen

Nehmen wir an der LanguageFilter benötigte noch einen weiteren Parameter, wie zum Bespiel eine weitere Sprache, wobei Einträge der Ausgangstabelle auch in dieser Sprache in die gefilterte Tabelle übernommen werden sollen, so müsste man den LanguageFilter aus dem vorherigen Abschnitt eben um diesen Parameter und um den Setter für diesen Parameter erweitern und in das parameters-Element ein weiteres „parameter“-Kind mit Name des weiteren Parameters als Attribut „name“ und dessen Wert als Attribut „value“ aufnehmen.
Dies könnte dann so aussehen:

Erweiterung LanguageFilter.java:

```
private String lang2 = "";
public void setLang2(String lang2) {
            this.lang2 = lang2;
}
```

Ändern des parameters-Element (z. B. auch spanische Einträge aufnehmen):

```
<parameters>
        <parameter name=“lang“ value=“en“/>
        <parameter name=“lang2“ value=“es“/>
</parameters>
```

2.2.5 Tags

Aus dem vorherigen Abschnitt wissen wir, dass Filter Tags setzen und auf Tags prüfen können. Mit hasTag(String tag) kann man überprüfen, ob ein Result ein Tag hat und mit hasTags(List<String> tags) lässt sich überprüfen, ob ein Result alle Tags aus der Liste tags hat.

2.2.6 Filter nur auf Results mit bestimmten Tags ausführen

Eine bequeme und umfassendere Möglichkeit den Filter im jeweiligen filter-Element nur auf Results , die bestimmte Tags haben oder auch nicht haben, auszuführen, ist es dem filter-Element ein optionales tags-Element mitzugeben.
Ohne die Angabe des tags-Elements wird der Filter weiterhin auf allen Tabellen, die Blätter der vorliegenden Baumstruktur sind, ausgeführt. Gibt man ein tags-Element an, wird der filter auf allen unter einem Result, das die entsprechenden, im tags-Element definierten Anforderung an Tags erfüllt, ausgeführt:

```
<filter classname="filters.LanguageFilter">
        <parameters>
                        <parameter name=“lang“ value=“en“/>
        </parameters>
         <tags op="and">
                <tag>Wirtschaft</tag>
                 <not>
                         <tag>Politik</tag>
                 </not>
                 <tags op="or">
                         <tag>Deutschland</tag>
                         <tag>USA</tag>
                </tags>
        </tags>
</filter>
```

Das besprochene tags-Element erhält unbedingt einen boolschen Operator op als Attribut, der „and“ oder „or“ sein kann. Innerhalb dieses tags-Elements können dann wiederum tags-Elemente liegen, oder ein bestimmter Tag mittels eines tag-Elements angegeben werden. Außerdem können Elemente auch Kind eines not-Elements sein, was bedeutet, dass das Gegenteil der Struktur innerhalb des not-Elements gefordert ist.
In obigem Beispiel würde der LanguageFilter also alle englischen Einträge in Results herausfiltern, die die Tags Wirtschaft, nicht den Tag Politik und den Tag Deutschland oder den Tag USA haben.

2.2.7 Tags vergeben

Zuletzt kann man in einem Filter, der Tabellen aufteilt, also ein Cluster erstellt wie der HalfFilter aus dem Abschnitt 2.1.3, auch den Kindern des entstehenden ResultComposites Tags mittels eines addTags-Element geben. Dies ist insbesondere zum Testen gedacht.
Im addTags-Element werden Tags als Kind mit dem Namen „addTag“ über dessen Attribute „name“ und „index“ gesetzt.

Will man beispielsweise der ersten Hälfte, also dem ersten Kind des durch den HalfFilter entstehenden ResultComposites den Tag „ersteHälfte“ und der zweiten Hälfte, also dem zweiten Kind, den Tag „zweiteHälfte“ geben, so geschieht dies folgendermaßen:

```
<filter classname=“HalfFilter“>
        <addTags>
                <addTag name=“ersteHälfte“ index=“0“/>
                <addTag name=“zweiteHälfte“ index=“1“/>
        <addTags>
</filter>
```

3 Renderer

3.1 Renderer schreiben

3.1.1 Einleitung

Renderer sind der zweite modulare Teil des Frameworks. Sie dienen dazu, die gefilterten Ergebnisse darzustellen. Renderer bestehen aus Groovy-Server-Pages (GSP), einer Technologie analog zu den bekannteren Java-Server-Pages, mit Groovy als Skriptsprache. Damit lassen sich gewöhnliche HTML-Seiten schreiben und deren Verhalten dann durch Groovy-Code anpassen. Prinzipiell sind praktisch alle Java-Statements auch valide Groovy-Statements, von daher ist es nicht notwendig extra Groovy zu erlernen. Außerdem lässt sich aus den GSP auf Java-Klassen zugreifen, um die Funktionalität zu erweitern.
Das Anwenden der Renderer lässt sich dann ähnlich wie bei Filtern durch XML-Konfiguration mit Tags steuern. Letztlich werden die gerenderten Ergebnisse der Reihe nach im Ausgabefenster dargestellt. Wie bei den Filtern gibt es zwei Typen von Renderern, die sich im Ziel bei der Anwendung unterscheiden: der Table-Renderer arbeitet auf einzelnen Tabellen, der ResultRenderer arbeitet auf einem gesamten Baum. Vom Render-Algorithmus werden dann die Teilbäume mit passenden Renderern verknüpft und schließlich vom Framework in die Hauptseite eingefügt.

3.1.2 Renderer Beispiele

Der Code der Renderer wird direkt in die Divs auf der Hauptseite eingefügt und kommt daher ohne <html>, <head> und <body> aus.
Einige wichtige Groovy-Elemente sind hierbei:
Mit ${Groovy/Java-Statement} lassen sich beliebige Groovy- bzw. Javastatements ausführen z.B.
${System.out.println("Hello World!")}
<%@page import="*result.Table*" %>
bindet die Klasse result.Table ein, um sie in der GSP benutzen zu können z.B.
${Table.someFunction()}
Außerdem nützlich sind die Groovy-Tags, insbesondere <g:if>, um dynamisch HTML-Code einzufügen. Zum Beispiel

```
<g:if test="${table.size > 0}">
      [some html to display the table]
```

```
      ...
</g:if>
<g:else>
      Table is empty!
</g:else>
```

Dabei wird geprüft ob das Statement bei test wahr ist und falls ja wird der Code danach ins Dokument eingefügt, ansonsten der Teil aus dem Else-Block.
Mit <g:var> lassen sich lokale Variablen setzen, z.B.

```
<g:set var="i" value="${0}" />
```

und mit <g:while> dann Schleifen bauen:

```
<g:while test="${i < table.size()}">
```

Das waren nur die wichtigsten Beispiele; es gibt noch viele weitere Tags. Für zusätzliche Informationen siehe: http://grails.org/doc/2.1.0/ref/Tags/if.html
Das Result, für das der Renderer zuständig ist, ist per ${result1} verfügbar.

Ein komplettes Beispiel für einen Tabellenrenderer könnte dann so aussehen (_DefaultTableRenderer.gsp):

```
<!--  same as DefaultCompositeRenderer but with targettype table and hence more
concise -->
<%@page import="utils.Utils" %>
<%@page import="result.Result" %>
<%@page import="result.ResultComposite" %>
<%@page import="result.Table" %>


<!-- open table -->
<table border="1">
      <tr>
            <th>Subject</th>
            <th>Predicate</th>
            <th>Object</th>
      </tr>
      <!--  set local variables - one for each column -->
      <g:each in="${result.getTriples()}" var="triple">
            <g:set var="sub" value="${Utils.utf8toUnicode(triple.sub)}" />
            <g:set var="pred" value="${Utils.utf8toUnicode(triple.pred)}" />
            <g:set var="obj" value="${Utils.utf8toUnicode(triple.obj)}" />
<!-- insert columns -->
            <tr>
                  <td><g:if test="${sub.startsWith("http://")}">

                        <a href="${sub}" onclick="javascript:return
                              checkForRDF(this);">${sub}</a>
                  </g:if> <g:else>
                        ${sub}
                  </g:else></td>
            <td><g:if test="${pred.startsWith("http://")}">

                        <a href="${pred}" onclick="javascript:return
                              checkForRDF(this);">${pred}</a>
                  </g:if> <g:else>
```

```
                        ${pred }
                    </g:else></td>
                <td><g:if test="${obj.startsWith("http://")}">

                        <a href="${obj}" onclick="javascript:return
                            checkForRDF(this);">${obj}</a>
                    </g:if> <g:else>
                        ${obj }
                    </g:else></td>
                </tr>
        </g:each>
</table>
```

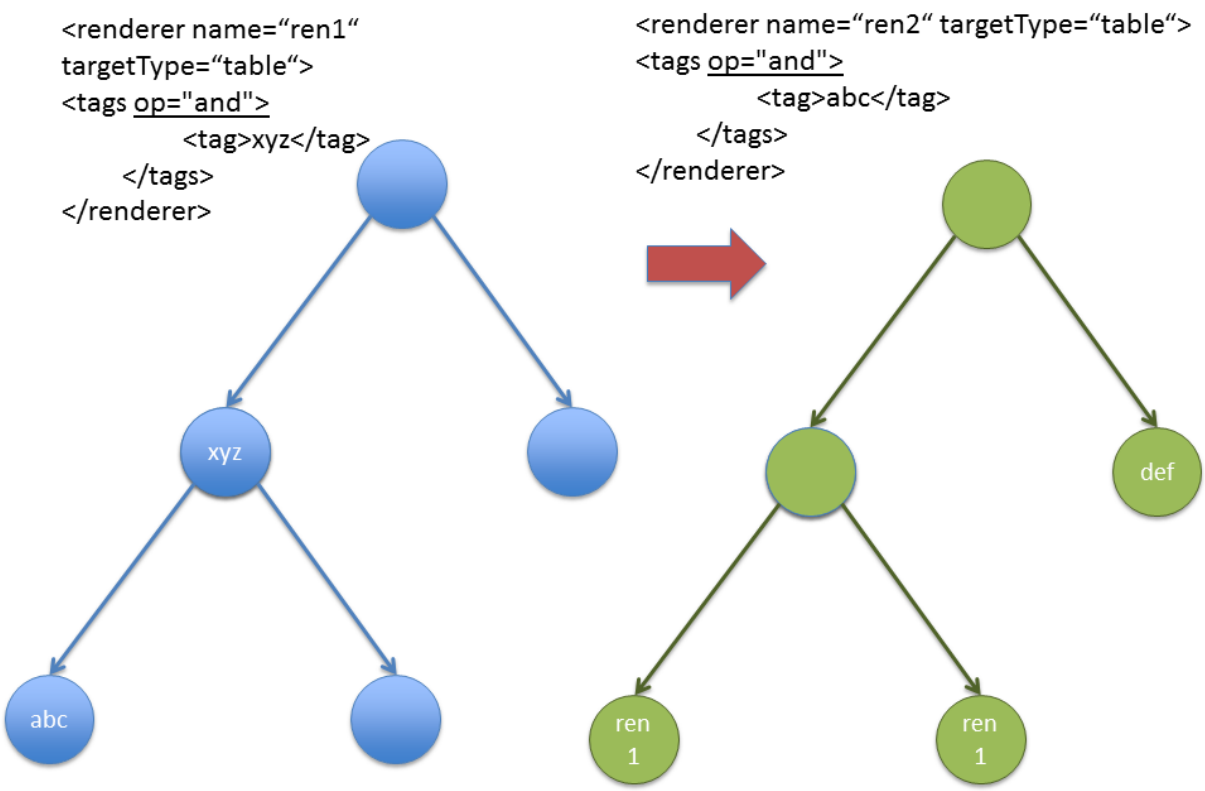
Dabei werden zunächst die benötigten Klassen importiert. Dann wird in gewöhnlichem HTML die Tabelle eröffnet. In dem <g:each>-Tag werden die Tripel der Tabelle durchlaufen und dazu immer einer lokalen Variable zugewiesen. Dabei wird auch eine statische Java-Funktion zum Konvertieren der Zeichensätze aufgerufen. Mit dem <g:if>-Tag wird jeweils geprüft, ob es sich um einen Link handelt und falls ja, auf den Controller verwiesen.

3.2 Renderer anwenden

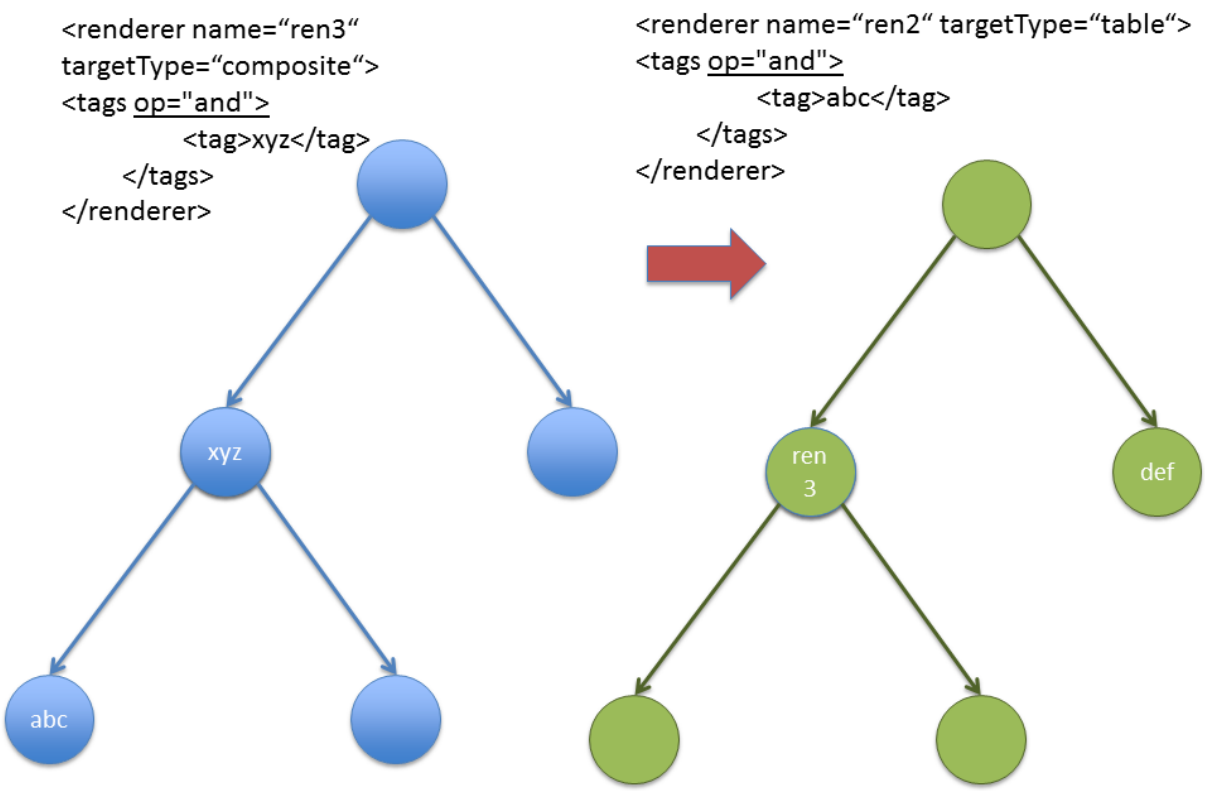
Um einen Renderer zu testen muss die entsprechende GSP-Datei in den Ordner zur index.gsp in views/query gelegt werden. Bei der Benennung ist wichtig, dass der Dateiname mit einem Unterstrich („_“) beginnen muss und keine Leer- oder Sonderzeichen enthält. In der appconfig.xml wird dann eine XML-Datei angegeben, die die genaue Anwendung steuert.

3.2.1 Render-Algorithmus-Dokumentation

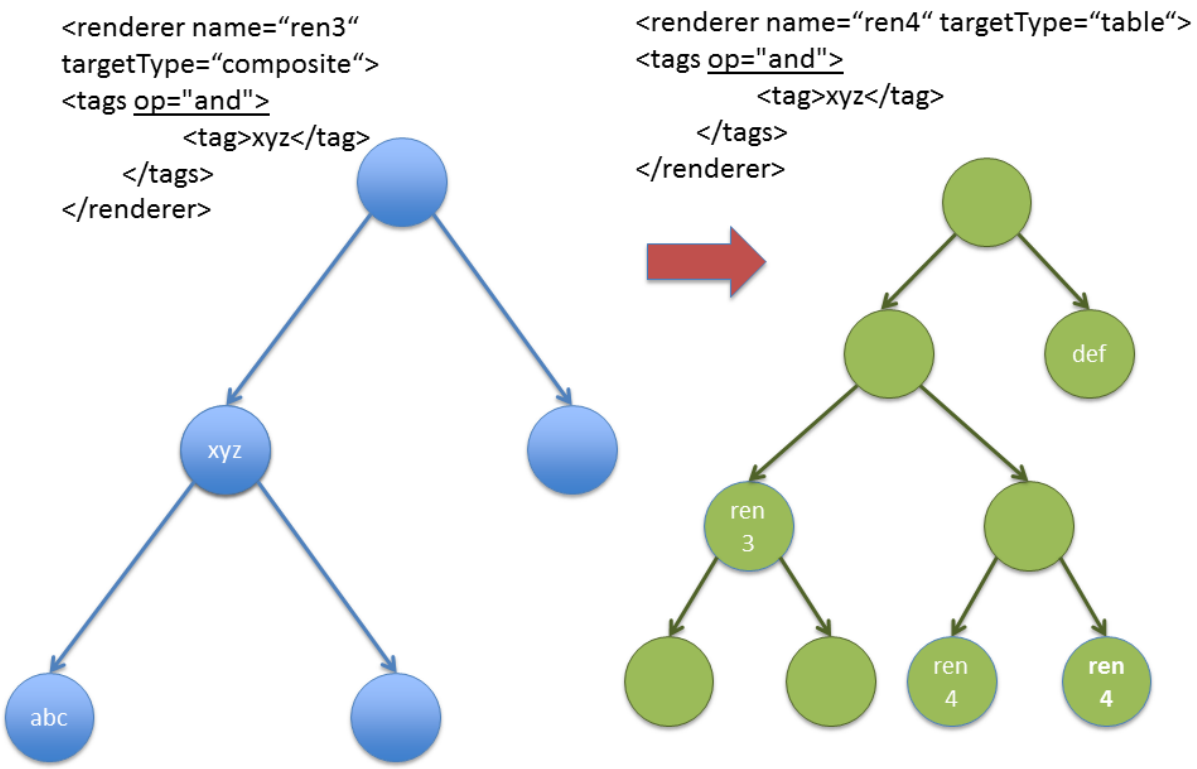
Der Render-Algorithmus dient dazu, aus dem Result-Baum und der Menge von Renderern mit bestimmten Tags eine Liste aus Renderer-Result-Paaren zu konstruieren. Dabei wird der Baum rekursiv abgelaufen und für die Knoten jeweils anhand der Tags nach passenden Renderern gesucht. Wurde kein Renderer gefunden, so wird bei den Kindern weitergesucht und ist man bei Tabellen (die ja keine Kinder haben) angekommen, wird auf den „DefaultRenderer“ zurückgefallen, der die Tabelle in der gewohnten Dreispaltigen Textdarstellung ausgibt. Das heißt, falls kein Renderer angegeben wird, ist die Darstellung unverändert wie zuvor.
Möchte man einen gesamten Baum mit einem anderen Renderer gerendert haben, so gibt man einfach keine Tags als Bedingung an. Möchte man einen Teilbaum mit dem Tag „xyz“ per ABC-Renderer gerendert haben, so gibt man den ABC-Renderer mit dem Tag „xyz“ an und der Rest des Baums wird dann weiterhin per Defaultrenderer dargestellt. Außerdem lässt sich durch den TargetType (Composite/Table) angeben, ob der Filter auf Clustern (=Kompositum) arbeiten soll und damit direkt angewandt wird, oder auf Tabellen und damit erst in den Blättern angewandt wird. Wichtig ist, dass ein Teilbaum jeweils mit dem Renderer des zur Wurzel nächsten Knotens gerendert wird. Dazu folgendes Beispiel:



Der Knoten links unten ist mit „abc“ getaggt, der Elternknoten mit „xyz“. Die Renderer seien wie in den XML-Schnipseln (näheres siehe nächstes Kapitel) oben definidert: Ren1 als Tabellenrenderer auf „xyz“ und ren2 als Tabellenrenderer auf “abc”. „def“ stehe für den DefaultRenderer. Da „xyz“ am nächsten zur Wurzel ist, kommt ren1 zur Anwendung und die beiden unteren Knoten werden von ren1 gerendert. Der Knoten ganz rechts hatte keinen speziellen Renderer angegeben und wird daher vom Defaultrenderer dargestellt.
Würde es sich um einen Composite-Renderer handeln, ergäbe sich folgendes Bild:



Es wird ren3 gewählt, da ren2 mit „abc“ unter „xyz“ liegt und diesmal wird direkt der Knoten, also das Composite bei „xyz“ an den ren3 übergeben.
Werden mehrere Renderer dem selben Knoten zugeordnet, so teilt sich dieser Knoten und die Daten werden jeweils einmal mit den einzelnen passenden Renderern dargestellt.
Hierzu noch einmal ein Beispiel:



Insgesamt geht der Algorithmus also von der Wurzel aus rekursiv durch den Baum und sucht für jeden Knoten nach einem Renderer, der zu dem/n Tag/s des Knotens passt. Wird er fündig, so werden dem Knoten der/die gefundene/n Renderer zugeordnet und der entsprechende Teilbaum ist erledigt. Ansonsten macht er bei den Kindern weiter. Ist er schließlich bei einer Tabelle angekommen, ohne das ein passender Renderer gefunden wurde, so wird der Tabelle der Defaultrenderer zugeordnet. Auf der Hauptseite wird dann die Liste der Renderer-Result-Paare durchgegangen und die Renderer mit den jeweiligen Results aufgerufen.

3.3 XML-Konfiguration der Renderer

Die Konfigration der Renderer funktioniert sehr ähnlich zu der von Filtern. Konkret sieht das bspw. so aus:

```
<renderers>
<renderer name="CoordinateRenderer" targetType="table">
	<tags op="and">
		<tag>Coordinate</tag>
	</tags>
</renderer>
</renderers>
```

Ist eine vollständige Konfiguration für einen Renderer.

```
<renderers>
...
</renderers>
```

Stellt das Wurzelelement dar mit einem beliebig wählbaren Namen.

Jeder einzelne Renderer wird jetzt als Kind der Wurzel in der Form

```
<renderer name="..." targetType="[table/composite]">
	<tags op="...">
		<tag>...</tag>
	</tags>
</renderer>
```

angegeben. Name gibt dabei den Namen der entsprechenden GSP-Datei an in der form _name.gsp. Die zu diesem Renderer gehörige Datei müsste also _CoordinateRenderer.gsp heißen. TargetType gibt an, ob der Renderer auf Tabellen oder Clustern arbeiten soll und muss entsprechend „table“ oder „composite“ sein. Die Tags lassen sich logisch genau so verknüpfen wie bei Filtern. Hier wird einfach nach dem Tag „Coordinate“ gesucht.

Es ist auch möglich den DefaultRenderer durch einen eigenen zu ersetzen. Dazu wird er in der Form
<defaultRenderer name=“MyDefaultRenderer/>
angegeben und wird dann vom Algorithmus automatisch angewandt, wenn kein anderer Renderer zu einer Tabelle gepasst hat. Es muss sich also immer um einen Table-Renderer handeln.

4 Deployment

4.1 Anwendung Deployen

Zunächst wird in Springsource mit dem grails>war Befehl die war-Datei erstellt. Der Pfad zur Anwendung wird von Tomcat standardmäßig durch den Namen dieser war-Datei bestimmt.Grails benötigt alleine etwa 500mb Speicher, es sollten also mindestens 600mb Im Tomcat zur Verfügung stehen („Initial/Maximum Memory Pool“ entsprechend anpassen). Dann kann die war-Datei wie gewohnt deployt werden.

Die Konfigurationsdateien liegen im webapp/Config-Verzeichnis. In der appconfig.xml werden die jeweiligen Filter- und Renderer-XML-Dateien ausgewählt und in den FilterFileFolder/RendererFileFolder die entsprechenden Dateien abgelegt.

Bibliotheken werden als jar in WEB-INF/lib gelegt und sind nach einem Neustart des Tomcat verfügbar.

4.2 Dynamisch laden

4.2.1 Filter dynmisch laden

Um Filter dynamisch geladen zu bekommen, müssen die jeweiligen .class-Dateien in den „[Tomcat]/webapps/LDB/WEB-INF/classes/filters“-Ordner kopiert werden. Bei Springsource finden sich die .class Dateien im target-Ordner.

4.2.2 Renderer dynamisch laden

Um Renderer dynamisch zu laden, muss die entsprechende GSP in den „[Tomcat]/webapps/LDB/WEB-INF/grails-app/views/query“-Ordner gelegt werden.

5 AppConfig

Die appconfig.xml im webapp/Config-Ordner dient dazu, einige Eigenschaften der Anwendung zu steuern. Für den Anwender des Frameworks wichtig sind vor allem die ersten beiden Optionen: „filterFile“ und „rendererFile“, die die zu verwendenden Filter- bzw. Renderer-XML-Dateien spezifizieren. Die restlichen Optionen dienen zum Teil internen Entwicklungszwecken, werden aber dennoch kurz erläutert.
Alle Optionen im Einzelnen:

5.1 filterFile

Gibt den Pfad zur Filter-XML-Datei relativ zum Config-Verzeichnis an.

5.2 rendererFile

Gibt den Pfad zur Renderer-XML-Datei relativ zum Config-Verzeichnis an.

5.3 debugMode

Im Debugmodus erfolgt eine erweiterte Fehlerausgabe. Für den Produktiveinsatz sollte er daher abgeschaltet werden.

5.4 localMode

Im LocalMode werden die RDF-Daten zu einer URI nicht im Internet, sondern im Projektordner in einer RDF-Datei, deren Name der UTF-8-enkodierten URI entspricht, gesucht. Zum Beispiel zu http://dbpedia.org/resource/Darmstadt würde die Datei http%253A%252F%252Fdbpedia.org%252Fresource%252FDarmstadt.rdf geladen.

5.5 saveQuery

Deprecated. Serialisiert das Result einer echten, ungefilterten Anfrage.

5.6 lookupEnabled

Aktiviert den DBPedia-Lookup-Service. Da dieser des Öfteren langsam/unerreichbar ist, bietet es sich an, ihn in solchen Fällen zu deaktivieren.

5.7 cancelByStop

Standardmäßig wird bei CancelAllQueries QueryController.requestCancelled auf true gesetzt und Filter können dann diese Variable prüfen, um ihre Ausführung sauber zu beenden. Ist cancelByStop true, so wird per Thread.stop die Ausführung ungefragt beendet.
CancelQuery, das nur die Anfrage eines einzelnen Nutzers beendet, benutzt immer Thread.stop

5.8 enableCancelButton

Bestimmt, ob der „Cancel Query“-Button angezeigt wird.

5.9 enableCancelAllButton

Bestimmt, ob der „Cancell All Queries“-Button angezeigt wird.

5.10 displayDelays

Aktiviert Timer-Ausgaben während der Ausführung einer Anfrage.

6 Anhang

6.1 Wichtige (Code breaking) Änderungen

6.1.1 Version 2

6.1.1.1 Filter

Zwischen der ersten und der zweiten Version haben sich nach außen hin vor allem die Definition der Filter geändert. Filter ist nun die abstrakte Oberklasse für FilterTable und FilterComposite, entsprechend müssen die alten TabellenFilter von „extends Filter“ bzw. „implements Filter“ zu „extends FilterTable“ umbenannt werden. Ansonsten sollte die Funktionalität nicht beeinflusst sein.

6.1.1.2 XML-Format

Das Format von Parametern und Addtags bei Filter-XML hat sich geändert. z. B. <entriesPerPage>5</entriesPerPage> ist jetzt <parameter name="entriesPerPage" value="5"/> und <coordinates>3</coordinates> ist jetzt <addTag name="coordinates" index="2/>.

6.1.2 Version 3

6.1.2.1 XML-Konfigurationsdateien

Die XML-Konfigurationsdateien sind in den „Dynamic-Ordner“ gewandert. Die Filter und Renderer, die zuvor in config.xml eingestellt wurden, werden jetzt in appconfig.xml eingegeben. Ansonsten sollte alles wie gehabt sein.

6.1.3 Version 4

6.1.3.1 XML-Konfigurationsdateien

XML-Konfigurationsdateien sind nochmals gewandet und befinden sich nun in webapp/Config.