Tutorial and Documentation

1 IDE Setup 2

2 Filter 2

2.1 Writing Filters 2

2.1.1 Introduction 2

2.1.2 First Example – Returning a Table 4

2.1.3 Second Example – Returning ResultComposite zurückgeben and Setting Tags 5

2.1.4 Third Example – Boolean Attribute 6

2.1.5 Fourth Example – String Attribute 6

2.1.6 Tree Filters 7

2.2 XML Configuration for Filters 8

2.2.1 Basic Structure 8

2.2.2 Setting Captions 8

2.2.3 Setting Parameters 8

2.2.4 Setting Multiple Parameters 9

2.2.5 Tags 9

2.2.6 Selecting Results to Be Filtered With Tags 9

2.2.7 Setting Tags 10

3 Renderer 10

3.1 Writing Renderers 11

3.1.1 Introduction 11

3.1.2 Renderer Examples 11

3.2 Applying Renderers 13

3.2.1 Render Algorithm Documentation 13

3.3 XML Configuration of the Renderers 16

4 Deployment 17

4.1 Application Deployment 17

4.2 Dynamic Loading 17

4.2.1 Dynamic Filter Loading 17

4.2.2 Dynamically Renderer Loading 17

5 AppConfig 18

5.1 filterFile 18

5.2 rendererFile 18

5.3 debugMode 18
5.4 localMode 18
5.5 saveQuery 18
5.6 lookupEnabled 18
5.7 cancelByStop 18
5.8 enableCancelButton 18
5.9 enableCancelAllButton 18
5.10 displayDelays 18
6 Important (Code breaking) Changes 19
6.1.1 Version 2 19
6.1.2 Version 3 19
6.1.3 Version 4 19

1 IDE Setup

In order to be able to develop your own filters you first need to setup an IDE and import the project. We are using Sprinsource, an Eclipse extension for Spring/Grails, and would recommend you do so too, although it is possible to use other IDEs. Usage is quite similar to Eclipse, which we assume to be known. Springsource is available at http://www.springsource.org/downloads/sts (registration is not necessary, just click „I'd rather not fill in the form. Just take me to the download page“ at the bottom).

Once Springsource is unpacked and started you need to install “Grails”, “Grails Support” and “Groovy Eclipse” from the Dashboard under “Extension”. If the Dashboard is not visible go to Help->Dashboard. Now you can check out the project (“LDB”) with the SVN tools of your choice (Subversion in our case).

Our own filters are in the filters package in src/java. Renderers are in view/query. In the webapp/Config directory you will find configuration files. In appconfig.xml you can set your filter/renderer configuration files and other options. Filter XML files are in FilterFileFolder and renderer XML files in RendererFileFolder.

You can start the program with run-app (right-click project ->run as -> Grails Command(run app) to verify results in your browser.

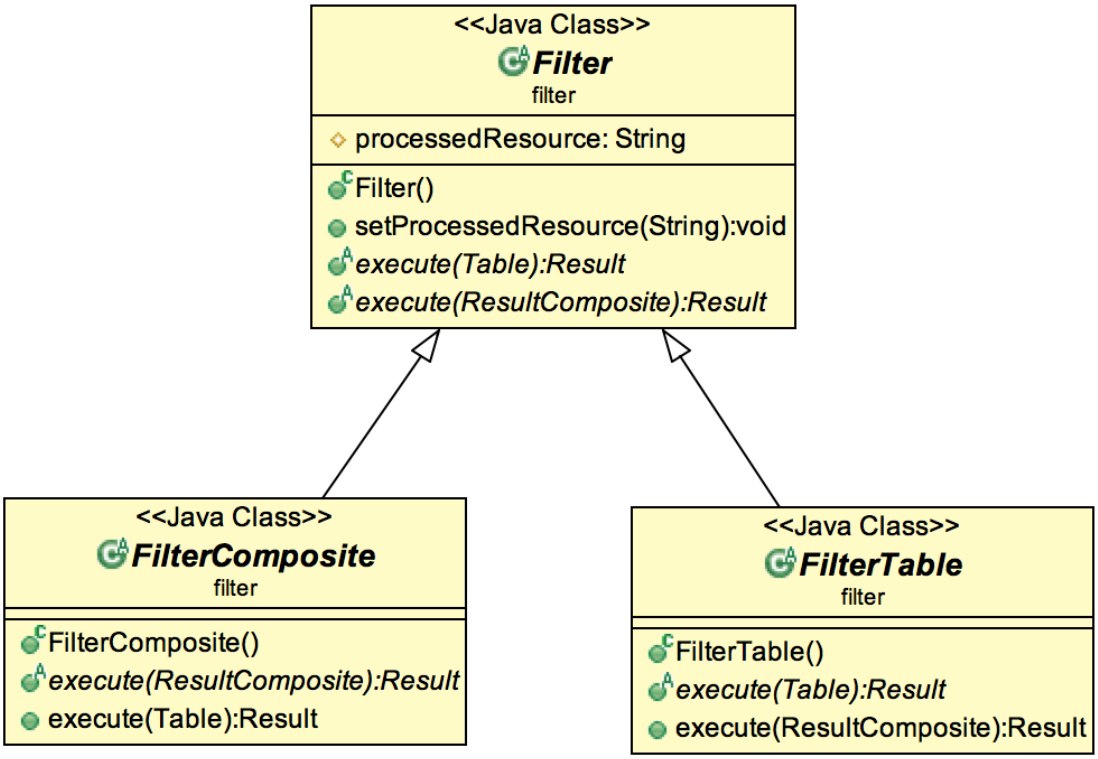
2 Filter

2.1 Writing Filters

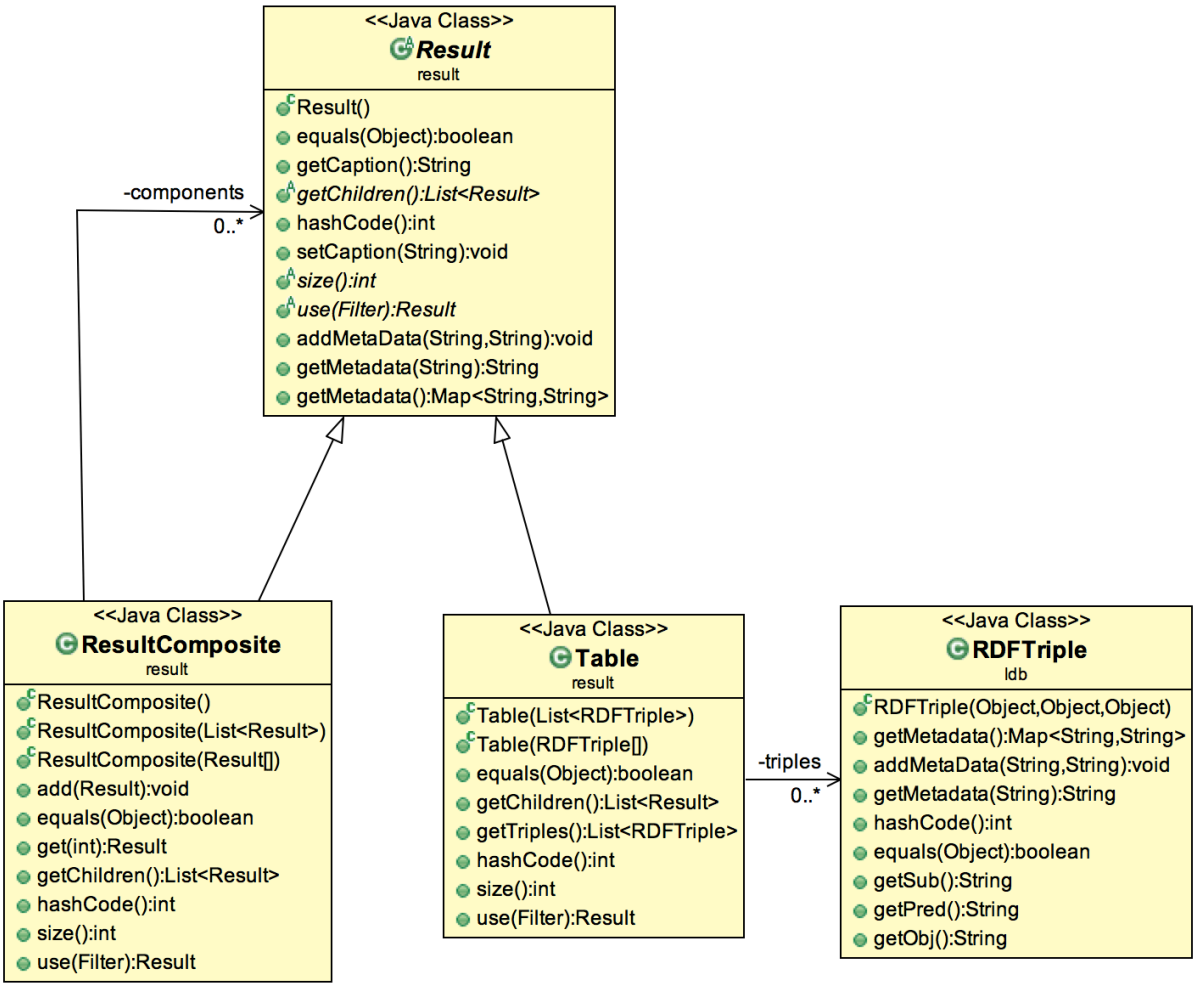
2.1.1 Introduction

Filters are an integral element of the framework. There are two types of filters: table filters and tree filters. Table filters allow you to manipulate lists of RDF triples und represent the most important

class of filters. Tree filters work on clusters of results. Specifically filters implement the abstract class "Filter" and thus the "Result execute(Table table)" resp. "Result execute(ResultComposite resultComposite)" methods:



Other than that filters only contain a setter for the currently processed resource.
Now some explanations for the classes you will be working with:
The internal data structure implements a composite pattern with Result as component, ResultComposite as composite and Table as leaf.
Table contains a list of RDFTriple. RDFTriple is a simple class for holding three strings: subject, predicate and object. ResultComposite, also called "Cluster", can contain an arbitrary number of Tables or other ResultComposites. A class diagram for this:



The most important method in Result is use(Filter). If Filter is an instance of FilterComposite it will be used on this very Result, if it is an instance of FilterTable it will be used on all tables in this composite.

Via getChildren you have access to this result's direct children, i.e. the tables and other ResultComposites it contains and via size you can query their number.

Result and RDFTriple implement the Metadata interface, which allows you to add metadata in form of string-string pairs.

Result furthermore implements Taggable so you can add tags to your results and get/SetCaption for adding captions.

The triple in RDFTriple and the list of Triples in table is immutable, metadata ("Metadata", tags and captions) however are not. It is therefore important, when manipulating metadata with filters, to create new tables so you don't accidentally override the metadata of the tables that were passed in as an argument to Filter.execute().

2.1.2 First Example – Returning a Table

Let's consider a simple example of a complete filter class:

`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);
	}
}
```

This filter receives a Table, creates a list of triples, adds each unique triple once and then returns a new table with those unique triples.

2.1.3 Second Example – Returning ResultComposite zurückgeben and Setting Tags

Now let's have a look at an example (from now on only the execute method) that will return a ResultComposite:

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;
	}
```

This filter splits a table in two halves and returns them as a ResultComposite.
Those are the two possible cases of return types: one basic Table or a ResultComposite.
Also note how you can set tags for which other filters could test, for further information see "Applying Filters".

2.1.4 Third Example – Boolean Attribute

It is possible to control a filter's behavior via custom attributes:
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);
	}
```

This filter sorts a table.
The attribute in this case is "boolean ascending" to choose whether to sort up or down. Attributes are set in the filter's XML configuration (more about that in the next section) and can be accessed in the filters as usual variables. Possible attribute types are all primitive types and String.

2.1.5 Fourth Example – String Attribute

Another example:

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);
	}
```

This filter looks at a triple's object and selects those that have a certain suffix in order to filter by language. The suffix or language is specified via the attribute "String lang".

2.1.6 Tree Filters

Tree filters use the FilterComposite class and as the name implies they can work on whole trees of results. This makes them useful for cutting of sub-trees:

```
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
	 */
	private 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);
        }
}
```

Tree filters are almost identical to table filters except that they receive a whole tree in form of a ResultComposite.
This filter will recursively walk the tree and cut off all components with the specified tag.

2.2 XML Configuration for Filters

In order to integrate filters into the browser you use XML configuration files.

2.2.1 Basic Structure

The XML file's root element is named "filters". Below this "filters" element you can specify your own filters as "filter" elements. They will be applied in the specified order. A filter is identified by its fully qualified class name, which you can enter as "classname".
The basic structure looks like this:

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

2.2.2 Setting Captions

Additionally you can have filters set a caption to be displayed by the renderer for its results. To do this use the optional attribute "caption" like this:

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

2.2.3 Setting Parameters

In case your filter has parameters, like LanguageFilter does, you need to specify their values by using the “parameters” element. Parameters must have a “parameter” element for each parameter you need to set. The “name” attribute of parameter has the parameter’s name and the “value” attribute its value. If you want to set the language of our LanguageFilter to English the filter element would look like this:

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

Make sure the parameters’ names and data types match exactly.

2.2.4 Setting Multiple Parameters

Now assume we want to change the LanguageFilter’s behavior to accept a second language as well by adding another parameter, “String lang2”:

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

Then the corresponding XML file could look like this.

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

2.2.5 Tags

In the last section you have seen how filters can set tags and check for them:
hasTag(String tag) checks if a Result has a certain tag, hasTags(List<String> tags), checks if it has all those tags in the list.

2.2.6 Selecting Results to Be Filtered With Tags

A convenient and comprehensive way to make a filter only work on Results with a certain tag, without a certain tag, or with some combination of tags, is to give the filter element an additional “tags” element.
If you do not specify a “tags” element the filter will be run on the whole (sub)-tree. If you do specify one it will only be run on those (sub-)trees that fulfill certain requirements.
An example:

```
<filter classname="filters.LanguageFilter">
```

```
        <parameters>
                        <parameter name=“lang“ value=“en“/>
        </parameters>
         <tags op="and">
                <tag>Economy</tag>
                 <not>
                        <tag>Politics</tag>
                 </not>
                 <tags op="or">
                        <tag>Germany</tag>
                        <tag>USA</tag>
                </tags>
        </tags>
</filter>
```

The tags element has a mandatory “op” attribute to indicate whether the result has to fulfill all (“and”) or at least one (“or”) of the requirements specified by its child elements. Those child elements can again be “tags” elements or the basic “tag” element, which only contains the string of a single tag. Also you can use the not tag if you want to use a filter on all elements that do not fulfill the requirements within it.
So in the above example the LanguageFilter would be run on all English entries that have the “Economy” tag and either the “Germany” or “USA” tag but do not have the “Politics” tag.

2.2.7 Setting Tags

You can also have filters, which return clusters of results, set tags to those results like the HalfFilter in our example in the last section does. This is mainly intended for testing.
To do so you use the “addTags“ element and its children the “addTag” elements:

```
<filter classname=“HalfFilter“>
        <addTags>
                <addTag name=“firstHalf“ index=“0“/>
                <addTag name=“secondHalf“ index=“1“/>
        <addTags>
</filter>
```

In the addTag element the “index” attribute specifies the index of the result and “name” the name of the tag it will be given.
In this example the first result of HalfFilter, i.e. the first half, will receive the tag “firstHalf” and the second child “secondHalf”.

3 Renderer

3.1 Writing Renderers

3.1.1 Introduction

Renderers are the second modular part of the framework. Their purpose is to convert the filtered results into an HTML output. Renderers consist of Groovy Server Pages (GSP), a technology like the better-known Java Server Pages, with Groovy as its scripting language. GSP enable you to write regular HTML pages and then dynamically change their structure with inline Groovy code. Practically all Java statements are valid Groovy statements too so learning Groovy is not necessary. It is also possible to access Java classes from GSP pages to extend their functionality.

As with filters you can control the renderers' behavior with XML configuration files and tags.
Also analogous to filters there are two kinds of renderers: renderers for tables and renderers for composites. The rendering algorithm, in more detail later, controls the application of renderers to certain results.

3.1.2 Renderer Examples

The code that the renderer produces will be directly inserted into the <divs> on the main page and hence does not need <html>, <head> or <body> tags.

Now some important Groovy elements:

${Groovy/Java-Statement} allows you to execute arbitrary Groovy or Java statements, e.g.:
${System.out.println("Hello World!")}

<%@page import="*result.Table*" %>
imports the result.Table classs so you can use it in your GSP e.g.: ${Table.someFunction()}
Also useful are Groovy Tags, especially <g:if>, in order to dynamically insert HTML code.
For example:

```
<g:if test="${table.size > 0}">
      [some html to display the table]
      ...
</g:if>
<g:else>
      Table is empty!
</g:else>
```

This checks if the statement after test is true and if so the code right after it will be inserted otherwise the code in the <else> tag.

<g:var> allows you to set local variables, e.g.:

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

with <g:while> you can build loops:

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

Those are only the most important examples; there are many more tags. For further information see: http://grails.org/doc/2.1.0/ref/Tags/if.html

The result for which the current renderer is responsible is available via ${result}.

A complete example of a table renderer could look like this:
(_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>
```

First we see the imports for all the Java classes we need. Then we declare a regular HTML table. The <g:each> tag walks all triples in the table and assigns them to local variables. While doing so we call a

regular Java function to do typeset conversion. Then the <g:if> tag is used to check if we are dealing with a link and if so the controller is called.

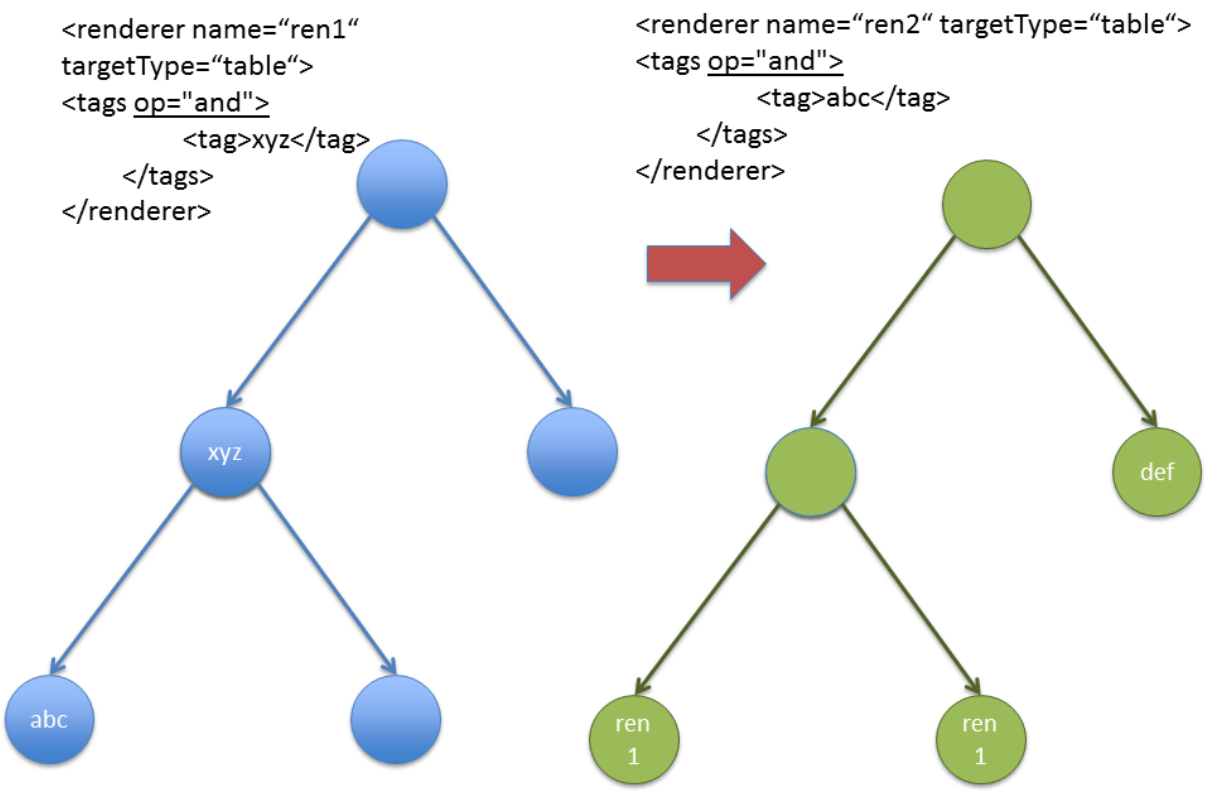
3.2 Applying Renderers

In order to test your renderers they need to reside in the same folder as the index.gsp (regularly views/query). Its name must begin with an underscore ("_") and it must not contain any spaces or special characters. In the appconfig.xml you will then specify an XML file that controls the behavior of your renderers.

3.2.1 Render Algorithm Documentation

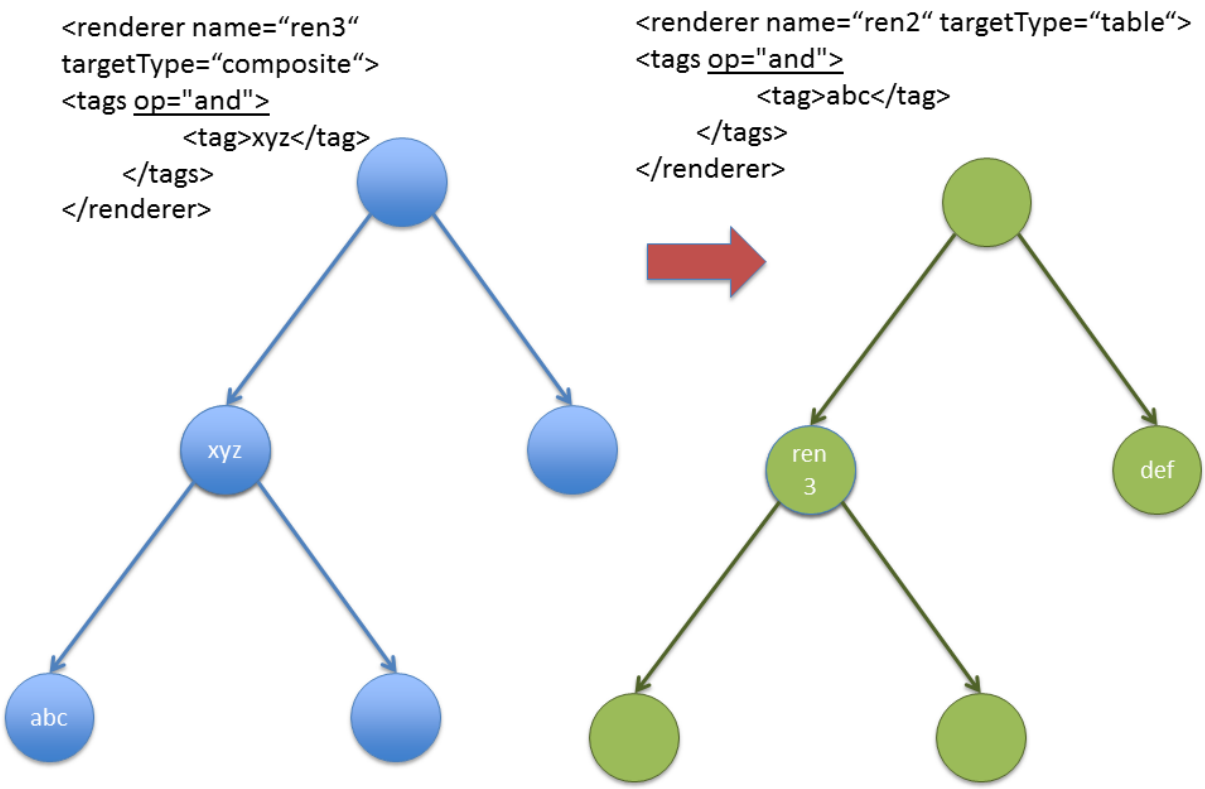
The Render Algorithm's purpose is to match the sub-trees of the result tree to their corresponding renderers by creating a list of renderer-result pairs. To do this it walk the tree recursively and for each node tries to find a renderer that fits its tags. If no renderer matched it continues looking for each of the node's children. If finally a table is reached it falls back to the "DefaultRenderer", which outputs the table in the basic three-column text style. So in case no renderer was specified everything will look the same as before.

In order to apply a renderer to the whole tree you simply leave the tags requirements empty. If you want to apply the "ABC renderer" to a sub-tree with the tag "xyz", you specify the "ABC renderer" with the "xyz" tag. The rest of the tree will still be renderer by the default renderer. Also you can use the "targetType" attribute to specify whether the renderer is supposed to work on clusters, then it will be applied to the first result that fits it, or on tables so it will only be applied to the leaves of the tree.
It is important to note that a tree is rendered from the root down to the leaves. An example of this:



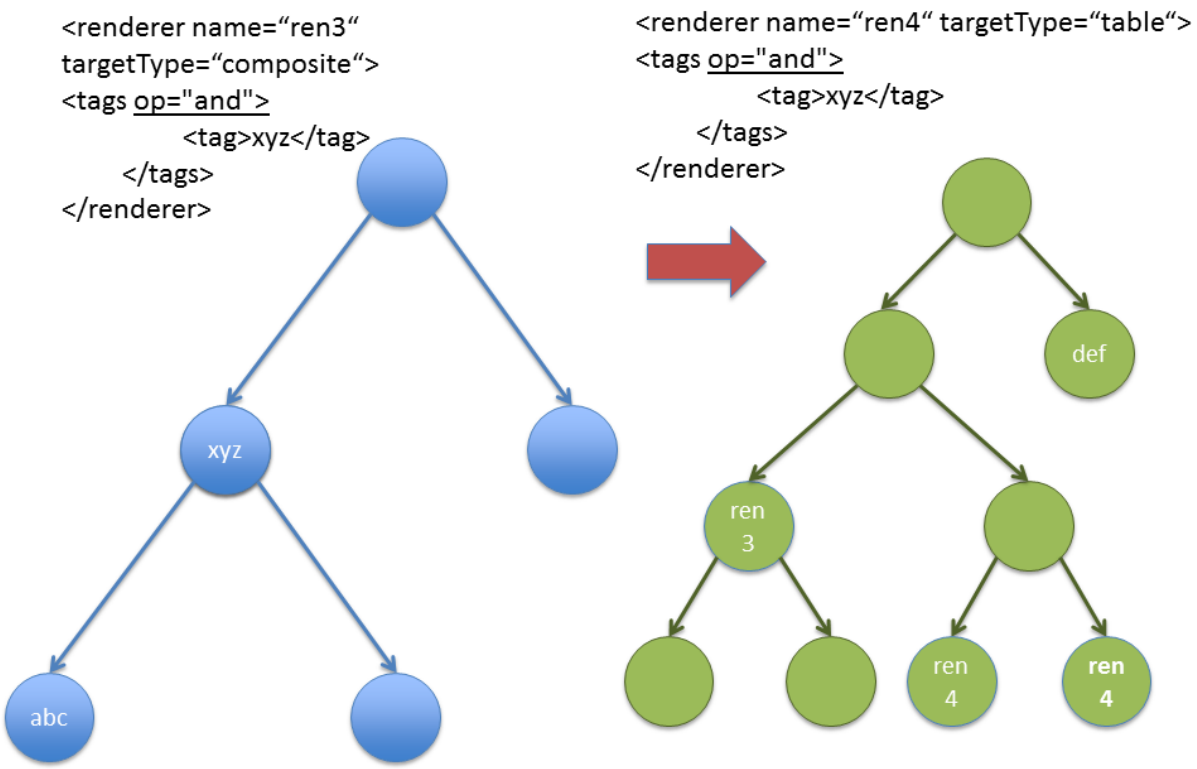
The lower left node of the first tree is tagged with “abc”, its parent with “xyz”. The renderers are defined as seen in the XML snippets (further information in the next section): ren1 as a table renderer for the “xyz” tag and ren2 as a table renderer for the “abc” tag. def represents the default renderer. Since “xyz” is closer to the root than “abc” ren1 will be used for the whole sub-tree and so the two tables in the leaves are rendered by ren1. The node to the right of the root had no specific tag so neither ren1 nor ren2 match and it will be rendered by the default renderer.

In a scenario where the first renderer has targetType composite we would see the following result:



Ren3 is used since the node with its tag, "xyz", is still closest to the root. But since ren1's target type is composite this node is immediately passed to ren3.

In case more than one renderer fits the tags of a given node each of them will receive this node to be rendered.
An example:



So conclusively the algorithm walks the tree recursively from the root towards the leaves and for each node it passes it looks for a renderer that fits this node's tag(s). In case of success the renderer(s) will be paired with that node and the given (sub-) tree is done. Otherwise it continues with that node's children. Finally, if a table is reached without finding a fitting renderer, the default renderer will render that table. Once everything is done, on the main page the list of renderer-result pairs will be traversed and the renderers will be called with their corresponding results.

3.3 XML Configuration of the Renderers

The configuration of renderers is quite similar to that of filters.

It looks like this:

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

This is a complete XML configuration for a renderer.

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

Represents the root element.

Every single renderer is declared as a child of the root element like this:

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

The “name” attribute represents the name of the corresponding GSP file in the form “_name.gsp”. So the GSP file for this renderer would be called “_CoordinateRenderer.gsp”. TargetType indicates whether the renderer is to work on tables (=”table”) or clusters(=”composite”). Tags can be combined with Boolean logic, just as with filter configurations. In this case we are simply looking for the “Coordinate” tag.

It is possible to replace the DefaultRenderer with a custom one. To do so you specify your own default renderer like this: <defaultRenderer name=“MyDefaultRenderer/>
It will then automatically be applied to all tables for which no other renderer could be found so this always has to be a table renderer.

4 Deployment

4.1 Application Deployment

First you need to create the war file in Springsource via grails>war. By default Tomcat determines the path to your application by that war file’s name. Grails needs about 500mb of RAM by itself so you should make available at least 600 mb for your browser (adjust “Initial/Maximum Memory Pool”). Then you can deploy the war file as usually.

Configurations files are placed in the webapp/Config directory. In appconfig.xml you specify your filter and renderer configuration XML files and put the corresponding files in FilterFileFolder/RendererFileFolder.

Libraries are placed in the lib folder in WEB-INF as jars and are available after a restart of the Tomcat.

4.2 Dynamic Loading

4.2.1 Dynamic Filter Loading

In order to dynamically load filters you put the class files into the “webapps/[INSTANCE NAME]/WEB-INF/classes/filters“ folder. In Springsource you can find the class file in the target directory.

4.2.2 Dynamically Renderer Loading

In order to dynamicall load renderers you put the GSP files into “webapps/[INSTANCE NAME]/WEB-INF/grails-app/views/query“.

5 AppConfig

The appconfig.xml file in webapp/Config serves to modify certain properties of the browser. The most important ones for the user of the framework are the first two options: “filterFile” and “rendererFile” as they specify the filter and renderer XML files which the browser uses. Other options are mainly intended for internal development purposes but will still be explained here.

The options are:

5.1 filterFile

Specifies the path to the filter XML file relative to the Config directory.

5.2 rendererFile

Specifies the path to the renderer XML file relative to the Config directory.

5.3 debugMode

DebugMode enables extended error output. Hence for production purposes it should be disabled.

5.4 localMode

In LocalMode a URI’s RDF data is not queried from the Internet but instead loaded from an RDF file in the project folder. That RDF file’s name is the URI encoded as UTF-8 so for example querying http://dbpedia.org/resource/Darmstadt would try to load http%253A%252F%252Fdbpedia.org%252Fresource%252FDarmstadt.rdf.

5.5 saveQuery

Deprecated. Was used to serialize real, unfiltered queries.

5.6 lookupEnabled

Enables the DBpedia Lookup Service. Since that service tends to be laggy or unavailable it can be useful to completely disable it.

5.7 cancelByStop

By default cancelAllQueries sets QueryController.requestCancelled to true so filters can check for this variable and gracefully terminate. If cancelByStop is true, Thread.stop will be used to cancel queries. CancelQuery, which only cancels a single user’s query, always uses Thread.stop

5.8 enableCancelButton

Enables the “Cancel Query” button

5.9 enableCancelAllButton

Enables the “Cancel All Queries” button.

5.10 displayDelays

Display timer output during queries.

6 Important (Code breaking) Changes

6.1.1 Version 2

6.1.1.1 Filter

The most important change between the first and the second version is the definition of Filter. Filter is now the abstract super class for FilterTable and FilterComposite and accordingly the class definition of all old table filters needs to be changed to “extends FilterTable”. Other than that filter functionality should not be affected.

6.1.1.2 XML-Format

The format of parameters and addtags in filter XML files has changed, e.g. <entriesPerPage>5</entriesPerPage> is now <parameter name="entriesPerPage" value="5"/> and <coordinates>3</coordinates> is now <addTag name="coordinates" index="2/>.

6.1.2 Version 3

6.1.2.1 XML Configuration Files

XML configuration files have moved to the “Dynamic” folder. Filters and renderers that were previously configured in config.xml are now configured in appconfig.xml.

6.1.3 Version 4

6.1.3.1 XML Configuration Files

XML configuration files have moved again and now reside in webapp/Config.