XML Searching (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Introduction

This document describes indexing, and searching XML documents in Apache Cocoon.

Indexing is the process of fetching XML documents from an Apache Cocoon instance, and building an index file. Searching is the process of querying the once built index.

See also Wiki: LuceneIndexTransformer

2. Decomposition of XMLSearching

The indexing process is split up into crawling, fetching URL resource, and generating the index.

The searching process is split up into searching, and feeding search result into the Apache Cocoon pipeline.

2.1. Crawling

The crawling process is specified by

1. Base URL to start crawling from
2. Included, and excluded URLs
3. Cocoon view to use for requesting links from an XML resource

Specifying the base URL determines the protocol for fetching XML resources. The implementation offers to specify http: URLs, crawling an Apache Cocoon instance deployed in a servlet-engine. Alternatively you may specify an URI, e.g.: /documents/index.html, offering to crawl the local Apache Cocoon instance only, either servlet-deployed, or in commandline-mode.

2.2. Fetching URL resource

This processing step fetches the URL resource from Apache Cocoon.

Apache Cocoon offers the feature of views. This feature is used to fetch the 'bare' content of an URL.

The crawling component described above is used by the this processing step to retrieve a link of an XML document. The link name is augmented by a cocoon view name for fetching the XML resource.

The Avalon component CocoonCrawler defines the interface of a crawler.

The Avalon component SimpleCocoonCrawlerImpl is the implementation. It can be configured to use a specific view, or default to the 'content' view.

2.3. Generating index

A xml resource is fed into a indexing engine. Generating an index specifies which elements of an XML resources should get indexed, how the elements are stored in the index. Moreover the physical file location of the index is specified by this processing step.

The current implementation splits up an XML resource the following way:

- Use an Lucene Analyzer for splitting up text

- Each XML element is indexed using its name as Lucene field name.
- Each XML attribute is indexed using its element name and the attribute name as field name. An attribute has following field name {element-name}@{attribute-name}.
- XML elements that match the names you configured in cocoon.xconf are added as stored fields.

The Avalon component LuceneCocoonIndexer defines the interface of an indexer.

The Avalon component LuceneXMLIndexer defines an interface for building an lucene index from an XML document. It uses an SAX content handler for parsing an XML document, and generating Lucene fields, the current index layout is implemented by SimpleLuceneXMLIndexerImpl, and LuceneIndexContentHandler.

2.4. Searching

This process uses a search engine for querying the index. The input of this process is a search query string, the result is the search result of the search engine.

The Avalon component LuceneCocoonSearcher defines an interface for searching a Lucene index.

2.5. Feeding Search Results

This is the final step for presenting information stored in the index. The result of search engine is feed into the Cocoon processing pipeline.

A GUI for the searching process may be developed using any java enabled script language, like JSP, or XSP. Moreover a sitemap generator component SearchGenerator is provided which transforms the search result to XML, and feeds it into the Cocoon processing pipeline.

3. Interdependencies

As both Avalon components LuceneXMLIndexer, and LuceneCocoonSearcher may use the same Lucene index, you must take care of the Lucene index structure in both components.

The current implementation uses following Lucene index layout

- Lucene field body indexed field of the pure text of an XML document. The body field is the default field name for searching. Thus the query-string foo, and body:foo is equivalent.
- Each XML element generates a Lucene field having the same name as the XML element name. For example searching for occurences of Cocoon inside of an XML abstract element, use query-string abstact:Cocoon.
- Each XML attribute generates a Lucene field having the name {element-name}@{attribute-name}. For example searching for occurrences of Cocoon inside of an XML title attribute of s1 element, use query-string s1@title:Cocoon.
- The Lucene field url stores the URI of the indexed document. As all fields described above are only indexed information, and no XML document is stored inside the Lucene index, this field is the only reference to the XML document resource.
- The Lucene field uid stores an unique id for implementing updating the index. This field is used for checking if the XML resource is newer than the information stored in the Lucene index.
- Further Stored fields can be added, depending on your configuration. Stored fields are returned in the hits found by the engine.

4. Configuration

Configuring the indexing, and searching Avalon components is specified in the cocoon.xconf file.

4.1. example

This would set up the crawler to crawl all of your site, except pages in the 'search' section, also we are telling the crawler to use a non-standard cocoon-view for getting the links in documents, called my-search-links.

<cocoon-crawler logger="core.search.crawler"> <exclude>.*/search/.*</exclude> <link-view-query>cocoon-view=my-search-links</link-view-query> </cocoon-crawler>

This tells the indexer to use the non-standard 'my-search-content' view to retrieve the content for indexing. Also it tells the indexer that we would like to have any title or subtitle XML elements in the document added to the index as stored fields, so they can be retrieved and displayed to the user with any hits they get.

<lucene-xml-indexer logger="core.search.lucene"> <store-fields>title, subtitle</store-fields> <content-view-query>cocoon-view=my-search-content</content-view-query> </lucene-xml-indexer>

Setting up the sitemap component SearchGenerator takes place in the sitemap.xmap file.

4.2. example

This would generate a document from a search, getting the query and other information from request parameters.

<map:generate type="search"/>

This would generate a document from a search, getting the query from the sitemap parameter '1' and other information from request parameters.

<map:generate type="search"> <map:parameter name="query" value="{1}"/> </map:generate>

5. Implementation notes

The package org.apache.cocoon.components.search holds all searching relevant components. The current implementation uses Jakarta Lucene as its indexing, and searching engine.

SearchGenerator is sitemap generator and is available in the package org.apache.cocoon.generation.

The package org.apache.cocoon.components.crawler holds all crawling relevant sources.

6. WebApp Sample usage

The Cocoon sample webapplication has a link for generating, an index of the Cocoon documentation, and searching the Cocoon documentation.

The following list describes step by step how to make use of webapp sample page:

1. Go to the page "Search the docs".
2. Create an index, follow the link "create". Creating an index may take some time, as the implementation accesses the XML resources via http: protocol.
3. Next you may query the index, by following the link "XSP", or "Cocoon Generators". Typing in a query will result in the table of hits orderer by relevance.

As a result of the creation step, there should exist an Lucene index in the directory index below the

temporary working directory of the servlet engine.

The "XSP" link for searching shows an XSP implementation of invoking the Avalon component CocoonSearch. Using this approach gives fine grained control over the searching process.

The "Cocoon Generator" links defines in the sitemap using the SearchGenerator, and transforming the XML search result to HTML. This approach tries to minimize your effort of using searching, as you need to adapt the XSLT transformation step only to your needs.

7. Extending the Sample

It is easy to extend the search sample to display more information about the search hit than just the url of the resource.

In order to show, for example, the title and summary of a document, these first need to be added to the search index as 'Stored Fields'. Then when the documents are found during a search, that information is available to display, from the search engine itself.

First, decide which fields you want to store.

Decide where is the best place in your pipeline for content to be extracted for indexing, it might not always be the default view 'content'.

Next, decide if you need an XSLT transformation on your documents, to make them more suitable for indexing. This may include deciding on one of several titles in your document, what part of your document gets added to the summary etc. You might want to strip certain tags out because you don't want their content searched. You might be able to raise hit scores on documents by re-arranging content, or keeping larger amounts of content in fewer tags.

Now you tell the search engine (in cocoon.xconf) which tags you'd like storing.

```
<lucene-xml-indexer logger="core.search.lucene"> <store-fields>title, summary</store-fields>
<content-view-query>cocoon-view=search-content</content-view-query>
</lucene-xml-indexer>
```

This example tells the indexer to store any tags called 'title' or 'summary' it finds in your documents. It also tells the indexer to get it's content from the view called 'search-content'.

```
<map:view from-label="search" name="search"> <map:transform src="search-filter.xsl"/>
<map:serialize type="xml"/> </map:view>
```

This is how you might setup that custom view in your sitemap. You would then add a label attribute label="search" to the appropriate place in your pipelines. See the section on views for more information.

After you have re-indexed the site, when you do searches, the new fields will be available in the XML output by Lucene, in the form of a search:field tag, you will need to modify your XSLT that displays the hits to show this.

```
<xsl:template match="search:hit"> <tr> <td> <xsl:value-of select="format-number(
@search:score, '### %' )"/> </td> <td> <xsl:value-of select="@search:rank"/> </td> <td> <a
target="_blank" href="{@search:uri}"> <xsl:attribute name="title"> <xsl:value-of
select="search:field[@search:name='summary']"/> </xsl:attribute> <xsl:value-of
select="search:field[@search:name='title']"/> </a> </td> </tr> </xsl:template>
```

This is how the search sample's xslt might be changed. All the fields you made for each document are available to you as search:field elements in the search:hit elements. The code above assumes you only had one 'title' and one 'summary' per document.

8. Summary

This document gives an overview of the components for using an indexing, and searching engine in Cocoon. It described the component decomposition of the Cocoon XMLSearch subsystem.

1. Comments

add your comments