WebDAVCMS

How to build a poor man's CMS

This tutorial shows you how to build a complete CMS without writing a single line of Java code. It builds on the samples of the WebDAV block. Just go to http://localhost:8888/samples/webdav/

You see that the samples consist of 5 steps where each step builds on the previous one. Each step contains a sitemap with a global variable named <staging>. To WebDAV enable the samples you first need to setup a WebDAV server of your choice. The Apache Web server with the mod_dav module is very easy to setup. Just follow WebDAVmod_dav instructions. Alternatively, Tomcat offers WebDAV (http://localhost:8080/webdav/index.html) and is simpler to setup.

After setting up your WebDAV server just copy the complete "webdav" directory from Cocoon's "sample" directory to your WebDAV server. To do this you can use your favorite WebDAV explorer such as Windows Explorer or Nautilus. Then adjust the <staging> variable of each sitemap to point to the corresponding folder on your WebDAV server. From this point on the content on your WebDAV server is used by the samples.

Alternatively you can mount your sitemaps on the WebDAV server by just changing the parent sitemap.

```
<map:match pattern="*/**">
  <map:mount check-reload="yes" src="webdav://localhost/webdav/{1}/" uri-prefix="{1}"/>
</map:match>
```

Nothing you want to do on a production system, but it's an easy way to version your sitemap changes on a dev system, given you have a DeltaV enabled WebDAV server.

The way it works is by the means of Cocoon's source resolving. The source resolving mechanism is used throughout Cocoon's architecture. Any component that makes use of the source resolving is able to resolve any of Cocoon's pseudo protocols. These pseudo protocols are defined in the "cocoon.xconf" configuration file. Within this file every pseudo protocol (a.k.a. source implementation) is mapped to a particular URI scheme (the URI scheme is the part before the colon of any URI). If Cocoon's source resolver sees any of these URI schemes it automatically uses the mapped source implementation to resolve the URI and get the content (with the file protocol being the default one).

The source resolving mechanism is extensible in that you can build your own pseudo protocol by implementing the "org.apache.excalibur.source.Source" interface and map the implementation to an URI scheme of your choice in "cocoon.xconf".

What does all this has to do with WebDAV? You might ask. Well, nothing and this actually is the best thing. That way you stay open. If you now decide to go with WebDAV you are not tied to WebDAV forever, but you can easily swap it by another repository implementation later on. In addition, if you base your repository on the "Source" interface you can easily integrate any Cocoon component that uses the source resolving.

The "step1" sample just demonstrates the "TraversableGenerator". Its functionality is identical to that of "DirectoryGenerator". But "TraversableGenerator" not only operates on the file system but on any of Cocoon's pseudo protocols that implements the "org.apache.excalibur.source.TraversableSource" interface in addition to the "org.apache.excalibur.source.Source" interface.

"TraversableGenerator" just generates a XML representation of the folder's contents.

```
<map:generate type="traverse" src="repo"/>
```

generates (simplified):

```
<collection:collection name="repo">
  <collection:resource name="contentA.xml"/>
  <collection:collection name="dir1"/>
  <collection:collection name="dir2"/>
</collection:collection>
```

The "step2" applies a simple stylesheet that generates links out of this XML. The generated links point to:

```
<map:match pattern="repo/**/">
  <map:generate type="traverse" src="{global:staging}repo/{1}/"/>
  <map:transform src="{global:staging}styles/dir2html.xsl"/>
  <map:serialize type="html"/>
</map:match>
```

for directories (with trailing slashes) and to:

```
<map:match pattern="repo/**">
  <map:generate src="{global:staging}repo/{1}"/>
  <map:transform src="{global:staging}styles/file2html.xsl">
    <map:parameter name="file" value="repo/{1}"/>
  </map:transform>
  <map:serialize type="html"/>
</map:match>
```

for files. So that the first pipeline matches to any "repo" subdirectory and the second pipeline matches to any file contained within repo or one of its subdirectories. The second pipeline uses another stylesheet "file2html.xsl" that converts the XML files to HTML.

The "step3" sample changes this stylesheet so that the elements of the XML files are converted to to input fields of a HTML form. The "action" attribute of this form points to another pipeline which is the interesting part of this sample.

```
<map:match pattern="write/**">
  <map:generate type="request" label="content"/>
  <map:transform src="styles/request2doc.xsl"/>
  <map:transform src="styles/doc2write.xsl">
    <map:parameter name="file" value="{1}"/>
  </map:transform>
  <map:transform type="write-source"/>
  <map:serialize type="xml"/>
</map:match>
```

The pipeline start with the "RequestGenerator" which feeds an XML representation of all request parameters into the pipeline. This XML representation already contains all parts of the to be written XML document. The "request2doc.xsl" stylesheet simply converts the XML to a format similar to the target document. Then the next stylesheet, "doc2write.xsl", wraps this XML with writing instructions for the "SourceWritingTransformer".

As its name suggests, the "SourceWritingTransformer" allows you to write to a source (as strange as it may sound), given the particular source implementation implements the "org.apache.excalibur.source.ModifiableSource" interface.

```
<source:write>
  <source:source>repo/contentA.xml</source:source>
  <source:fragment>
    <page>
      <title>Page 1</title>
      <content>
        <para>Paragraph 1</para>
        <para>Paragraph 2</para>
      </content>
    </page>
  </source:fragment>
</source:write>
```

A writing instruction for the "SourceWritingTransformer" contains the URI of the source to write to. Similar to other components using Cocoon's source resolving the "SourceWritingTransformer" does not care which pseudo protocol it writes to, as long as it implements the "org.apache.excalibur.source.ModifiableSource" interface.

The "step4" sample extends the "step3" sample in that it creates additional input fields for specifying some meta data. In the writing pipeline these meta data get separated from the document itself. The "doc2write.xsl" stylesheet generates writing instructions for 2 separate files, one for the document and one for the meta data.

Instead of writing the meta data to a separate file you can store them in other places as well, such as a SQL database. To store the meta data in Cocoon's embeded HSQL database add the following lines to "<cocooncontext>/WEB-INF/db/cocoondb.script" and restart Cocoon.

```
CREATE TABLE DOCS(PATH VARCHAR NOT NULL PRIMARY KEY,AUTHOR VARCHAR,CATEGORY VARCHAR,STATE VARCHAR)
INSERT INTO DOCS VALUES('repo/contentA.xml','me','catA','edited')
INSERT INTO DOCS VALUES('repo/dir1/contentB.xml','me','catB','edited')
INSERT INTO DOCS VALUES('repo/dir1/contentC.xml','me','Other Category','edited')
INSERT INTO DOCS VALUES('repo/dir2/contentA.xml','me','catA','edited')
INSERT INTO DOCS VALUES('repo/dir2/contentB.xml','you','catB','edited')
INSERT INTO DOCS VALUES('repo/dir2/contentC.xml','me','other','edited')
INSERT INTO DOCS VALUES('repo/dir2/subdir1/contentA.xml','you','docs','new')
```

Now change the "doc2write.xsl" stylesheet to look like this:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:req="http://apache.org/cocoon/request/2.0"
                xmlns:source="http://apache.org/cocoon/source/1.0">
<xsl:param name="file"></xsl:param>
<xsl:template match="request/parameters">
<page>
  <source:write create="true">
    <source:source><xsl:value-of select="$file"/></source:source>
    <source:path>page</source:path>
    <source:fragment>
      <title><xsl:value-of select="title"/></title>
      <content>
        <xsl:for-each select="content/para">
        <para>
        <xsl:value-of select="normalize-space(.)"/>
        </para>
        </xsl:for-each>
      </content>
    </source:fragment>
  </source:write>

  <sql:execute-query xmlns:sql="http://apache.org/cocoon/SQL/2.0">
    <sql:use-connection>personnel</sql:use-connection>
    <sql:query>
      update docs set
      author = '<xsl:value-of select="author"/>',
      category = '<xsl:value-of select="category"/>',
      state = '<xsl:value-of select="state"/>'
      where path = '<xsl:value-of select="$file"/>'
    </sql:query>
  </sql:execute-query>
</page>
</xsl:template>
</xsl:stylesheet>
```

and add the SQLTransformer to the writing pipeline:

```
<map:match pattern="write/**">
  <map:generate type="request" label="content"/>
  <map:transform src="styles/request2doc.xsl"/>
  <map:transform src="styles/doc2write.xsl">
    <map:parameter name="file" value="{1}"/>
  </map:transform>
  <map:transform type="write-source"/>
  <map:transform type="sql"/>
  <map:serialize type="xml"/>
</map:match>
```

Change the "metapage" pipeline:

```
<map:match pattern="metapage/**">
  <map:generate src="dummy.xml"/>
  <map:transform src="{global:staging}styles/2selectmeta.xsl">
    <map:parameter name="path" value="repo/{1}"/>
  </map:transform>
  <map:transform type="sql"/>
  <map:transform src="{global:staging}styles/2meta.xsl"/>
  <map:serialize type="xml"/>
</map:match>
```

with "dummy.xml" being:

```
<?xml version="1.0"?>
<page/>
```

"2selectmeta.xsl":

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="path"></xsl:param>
<xsl:template match="/page">
<page>
  <sql:execute-query xmlns:sql="http://apache.org/cocoon/SQL/2.0">
    <sql:use-connection>personnel</sql:use-connection>
    <sql:query>
      select author, category, state from docs
      where path = '<xsl:value-of select="$path"/>'
    </sql:query>
  </sql:execute-query>
</page>
</xsl:template>
</xsl:stylesheet>
```

and "2meta.xsl":

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/page">
  <metapage>
    <xsl:apply-templates/>
  </metapage>
</xsl:template>

<xsl:template match="*[name()='rowset']">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="*[name()='row']">
  <author><xsl:value-of select="*[name()='author']"/></author>
  <category><xsl:value-of select="*[name()='category']"/></category>
  <state><xsl:value-of select="*[name()='state']"/></state>
</xsl:template>

</xsl:stylesheet>
```

Everything works like before but now you can easily generate a list of all new documents.

```
<map:match pattern="getallnew">
  <map:generate src="getallnew.xml"/>
  <map:transform type="sql"/>
  <map:transform src="styles/list2html.xsl"/>
  <map:serialize type="html"/>
</map:match>
```

with "getallnew.xml":

```
<?xml version="1.0"?>
<page>
  <sql:execute-query xmlns:sql="http://apache.org/cocoon/SQL/2.0">
    <sql:use-connection>personnel</sql:use-connection>
    <sql:query>
      select path from docs where state = 'new'
    </sql:query>
  </sql:execute-query>
</page>
```

and "list2html.xsl":

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sql="http://apache.org/cocoon/SQL/2.0">
<xsl:template match="/page">
<html>
  <body>
    <ul>
      <xsl:apply-templates/>
    </ul>
  </body>
</html>
</xsl:template>

<xsl:template match="*[name()='rowset']">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="*[name()='row']">
  <li><a href="{*[name()='path']}"><xsl:value-of select="*[name()='path']"/></a></li>
</xsl:template>
</xsl:stylesheet>
```

"step5" is just another exercise that allows you to create new content.

If you follow the "step5" sample you may notice that the sample starts mixing concerns.

The "business logic" starts creeping into the XSLT Stylesheets.

To see how Cocoon's flow layer helps you to "gain back control" have a look at the flowsample

of the WebDAV block.