ContentAggregationExample

Example for Content Aggregation

Introduction

Consider the problem, that in a directory a number of (identical) XML files are stored; E.g. Articles to be published. It might be desired to create automatically an indexpage for accessing those articles.

The idea is, that everytime a new article is placed into this directory, metainformation about this article is automatically published in an index (table of contents) file.

The "Article" file

The "articles" files can be off course of arbitrary content, but just to get an impression of the idea, a small example here:

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<article>
    <meta>
        <title>Some Article Title</title>
        <abstract>
           Here stands a more concise description of the content
           in this article.
           Maybe some sentences at maximum
        </abstract>
        <category>Test</category>
        <history>
            <created
              date="22.5.2003"
              author="Alexander Schatten"
              email="alexander@xyz.zyx"
            />
            <changed
              date="22.5.2003"
              author="Alexander Schatten"
              email="alexander@xyz.zyx"
            />
            <changed
              date="23.5.2003"
              author="Hans Maier"
              email="hans@xyz.zyx"
            />
        </history>
        <valid ok="yes" />
    </meta>

    <content>

        <s1 title="Introduction">
            <p>This is the introduction of this article.</p>
            <p>Some Text follows here...</p>
        </s1>

...
...
...
```

So we can consider multiple files of this kind in a specific directory, in this example: "user/xml/articles"

The Sitemap Entry

```
    <map:match pattern="articles/browse.html">
        <map:generate src="user/xml/articles" type="directory"/>
        <!-- generate cincludes from directory listing -->
        <map:transform src="style/articles/article-cincludes.xsl">
            <!-- the directory of the files is not hard-coded in the stylesheet! -->
            <map:parameter name="article-dir" value="user/xml/articles" />
        </map:transform>
        <!-- all articles are merged -->
        <map:transform type="cinclude"/>
        <!-- and overview is printed as table -->
        <map:transform src="style/articles/article-list.xsl">
            <map:parameter name="use-request-parameters" value="true"/>
        </map:transform>
        <map:serialize type="html"/>
    </map:match>
```

This sitemap entry creates a list of all documents in the "/user/xml/articles" directory, forwards this list to the cincludes transformer which aggregates the documents. Then the final transformation step extracts the desired meta-information out of the aggregated files and displays it as HTML document.

The "article-cincludes.xsl" file

This stylesheet is responsible for the aggregation of the documents. It can look approximately like this:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:dir="http://apache.org/cocoon/directory/2.0"
   xmlns:ci="http://apache.org/cocoon/include/1.0"
>
   <xsl:param name="article-dir"/>
   <xsl:template match="dir:directory">
           <articles>
           <xsl:for-each select="dir:file">
               <file name="{@name}">
                   <ci:include src="{$article-dir}/{@name}" element="included" />
               </file>
           </xsl:for-each>
       </articles>
   </xsl:template>
</xsl:stylesheet>
```

It is to remark again, that the article directors is passed as parameter from the sitemap.

Check the aggregation result

It is recommended to check whether the aggregation was performed as desired. This can be done be slightly changing the sitemap entry like this:

```
    <map:match pattern="articles/browse.html">
        <map:generate src="user/xml/articles" type="directory"/>
        <!-- generate cincludes from directory listing -->
        <map:transform src="style/articles/article-cincludes.xsl">
            <!-- the directory of the files is not hard-coded in the stylesheet! -->
            <map:parameter name="article-dir" value="user/xml/articles" />
        </map:transform>
        <!-- all articles are merged -->
        <map:transform type="cinclude"/>
        <map:serialize type="xml"/>
    </map:match>
```

So it is stopped after aggregation, and the aggregated document is sent as XML to the browser. Have a look at it using "View Document Source" in the browser. This aggregated document is the basis for the last (visualizing) transformation step:

Final Transformation to HTML "Table of Contents"

The final transformation step should be easy, when the aggregated XML document is analyzed as described above. The following (rather long) XSL document is just to finalize this example: It generates a table that displays the table of content and some meta-information.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>

    <!-- This import is responsible for generation of the basic page design  -->
    <xsl:import href="art_default.xsl" />

    <!--
         This parameter can be passed whether to
         show invalid documents or not
    -->
    <xsl:param name="invalid"/>

    <xsl:template match="articles">
        <h1>Articles Overview</h1>
        <!-- create table header -->
        <table width="100%" border="1" cellpadding="3" cellspacing="0">
            <tr>
                <td><b>Category</b></td>
                <td><b>Article Name</b></td>
                <td><b>Author</b></td>
                <td><b>Created</b></td>
                <td><b>Last Changed</b></td>
                <td><b>Valid</b></td>
            </tr>
            <!-- Sort entries after title of article -->
            <xsl:apply-templates select="file">
                <xsl:sort select="included/article/meta/title"/>
            </xsl:apply-templates>
        </table>
    </xsl:template>

    <xsl:template match="file">
        <xsl:apply-templates select="included"/>
    </xsl:template>

    <xsl:template match="included">
        <xsl:apply-templates select="article"/>
    </xsl:template>


    <xsl:template match="article[meta/valid/@ok='yes']">
        <xsl:call-template name="renderArticle"/>
    </xsl:template>

    <!-- If document is set to invalid, check whether invalid documents should be listed, or not -->
    <xsl:template match="article[meta/valid/@ok='no']">
        <xsl:if test="$invalid='show'">
            <xsl:call-template name="renderArticle"/>
        </xsl:if>
    </xsl:template>


    <xsl:template name="renderArticle">
        <!-- change extension of filename from xml to html -->
        <xsl:variable name="filename"><xsl:value-of select="../../@name"/></xsl:variable>
        <xsl:variable name="filename_html">
            <xsl:value-of select="substring-before($filename, '.')"/>.html
        </xsl:variable>
        <tr>
            <td><xsl:value-of select="meta/category"/></td>
            <td><a href="{$filename_html}"><xsl:value-of select="meta/title"/></a></td>
            <td><xsl:value-of select="meta/history/created/@author"/></td>
            <td><xsl:value-of select="meta/history/created/@date"/></td>
            <td><xsl:value-of select="meta/history/changed[last()]/@date"/> </td>
```

```
            <td align="center">
                <!-- depending on valid flag: color of cell green or red -->
                <xsl:choose>
                    <xsl:when test="meta/valid/@ok='no'">
                        <xsl:attribute name="bgcolor">#FF4444</xsl:attribute>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:attribute name="bgcolor">#99FF66</xsl:attribute>
                    </xsl:otherwise>
                </xsl:choose>
                <xsl:value-of select="meta/valid/@ok"/>
            </td>
        </tr>
    </xsl:template>

</xsl:stylesheet>
```

Performance Issues

I have to note, that I have no idea of the performance implications of such a pipeline, as the aggregated content might become rather big.

I guess it would be a good idea to make this pipeline cached, and I already tried to use the cached cinclude, but it crashed the complete Cocoon setup, when applied as mentioned in the docs (Cocoon 2). Maybe some others have suggestions/experiences about performance/caching.