SimpleContentModel

Author: MarkLundquist

A simple content model for pages generated from static XML sources

In a standard Web server (such as Apache httpd), the default interpretation of the URL path component is as a filesystem pathname relative to the DocumentRoot directory. In Cocoon, there's no default; if a request doesn't match in the sitemap, you get the "No pipeline matched" error.

The Cocoon sitemap gives you complete freedom in mapping requests to processing resources (pipelines). But quite often, it makes sense to organize source documents in the filesystem in a way that maps directly to the request path component, i.e. something very analogous to what a standard Web server does (but not precisely the same, since we're still piping the source XML through some transformations to generate the HTML presentation).

This HOW-TO offers a content model and sitemap implementation that does this, and which also handles the following additional concerns automatically (i. e., without page-specific rules in the sitemap):

- Co-location of additional page-specific resources
- External redirects to a "trailing slash" form of the URL (yes, you **do** want this...)

Assumptions:

- You understand the basics of Cocoon and the (Cocoon sitemap (http:http://cocoon.apache.org/2.1/userdocs/concepts/sitemap.html))
- You know the basic pattern for publishing in Cocoon using XLST (SimpleTransformations).

I'll explain the content model by way of examples, and discuss these additional aspects in context of the examples. A sitemap fragment implementing all of this is given at the end of the article.

You can use this model as a starting point, and modify it according to your needs/tastes/etc.

The Content Model

OK, then... we start with this directory structure within the webapp directory:

```
webapp/
    content/
    static/
        content/
```

`content` is the root of a directory structure that contains the XML source documents.

`static/` contains:

- static resources common to all or most pages, such as
  - layout artwork, e.g. logos
  - site-wide CSS stylesheet
  - common external javascripts
  - etc...
    And...
  - An automatically-generated `content/` subdirectory containing symbolic links into the `webapp/content/` directory structure (see below!)

`static/` does **not** contain any page-specific resources (see below!)

In production, I run Cocoon applications behind an Apache front-end using mod_proxy (see SimpleModProxy). Apache is configured to serve directly URIs starting with `static/`, because it can serve static content so much faster than a servlet container could ever hope to do. But for development on my laptop, I don't want to have to run Apache, so the sitemap includes matchers for serving these static resources, so that I can just point my browser at Cocoon instances running on localhost.

The simplest case

In the simplest case, we just want to take an XML source document and transform it in the standard way. In this case, the request

```
/path/to/something
```

maps to the source document

```
content/path/to/something.xml
```

The implementation looks for this resource first.

Co-located resources

It seems like a standard practice in implementing websites in traditional web servers is to create some subdirectories in the DocumentRoot to contain non-HTML resources, like:

```
images/
css/
flash/
js/
```

I never liked just dumping "all the images" into one big subdirectory, just because they happen to be images. I'd rather have assets co-located with the documents that use them.

In this content model, a page that includes specific resources is represented by a directory whose name ends with ".page". In this case, the same example request '/path/to/dogs/Bowser' maps to this directory structure:

```
content/path/to/dogs
   Bowser.page/
      source.xml       # the document to be transformed
      static/
         images/
            bad_dog.jpg
            favorite_bone.jpg
            # etc..
         style.css        # a page-specific stylesheet
         client.js        # maybe there's some javascript specific to this page...
         flash/
            bowser_chasing_tail.swf
```

We then create a directory structure and a soft link in `webapp/static/content`. A tool is provided (see the implementation notes at the end of this article) that (re)builds `webapp/static/content`, so you don't have to make any of those soft links or the directory scaffolding by hand – just run the tool. But you should know what it does! The following directory structure is created:

```
webapp/static/content/
   path/
      to/
         dogs/
            Bowser      ----> .../content/path/to/dogs/Bowser/static
```

A reference to a page-specific resource looks like this in the HTML, e.g.

```
<img src="/static/content/path/to/dogs/Bowser/bad_dog.jpg" />
```

It would be quite undesirable to hard-code the site structure into our XML source document like that, so we'll add a template to our stylesheet that lets us just write that as `<img src="static/bad_dog.jpg">`; the template will rewrite that to the necessary URI in the generated HTML. See the implementation notes at the end of this article for the XSLT template.

The sitemap looks for `Bowser.page/source.xml` in `content/path/to/dogs` if the resource `Bowser.xml` (see "The simplest case" above) does not exist there. So this is really a "drop-in" content model; the sitemap doesn't have to be told on a page-by-page basis which resources use the "site-specific assets" model and which have just the source document as their only asset.

Note the the sub-structure of `.../Bowser/static` is completely *ad hoc*; all that matters at this level is that references in the page match the directory structure. The sitemap does not know or care about this per-page "static/" directory, and it certainly doesn't care how it's structured internally.

If you have static resources that are specific to a certain area of the site, this is handled too. Suppose we have some imagery that appears on every page in the `path/to/dogs/**`. Simply create a `static/` directory at the appropriate place in the `content/` directory structure, e.g.

```
content/path/to/dogs/
    static/
        some-common-image.jpg
    Bowser.page/
    ''etc...''
```

The tool will create the soft link in `static/content`, and you can reference it in the document like this, e.g.:

```
<img src="/static/content/path/to/dogs/some-common-image.jpg">
```

If you don't like hard-coding so much site structure into the URI, you can use a relative link beginning with "static/". For example, to reference that image from the Bowser.page example above:

```
<img src="static/../some-common-image.jpg" />
```

...and the link rewriting template in your stylesheet will do the right thing. Try it! 🙂

"Trailing slash" resources

Sometimes you really need a resource to be served at a URI that ends with a slash, because it determines whether a relative link in the page denotes a *chil* (if the current URL has a trailing slash), or a *sibling* (current URL has no trailing slash) of the current page. If the URL of the current page doesn't end with a slash (e.g., `path/to/dogs`), then we have an annoying situation if we want to include a link on that page to a child resource (e.g., `path/to/dogs/Bowser`). Our choices are:

- Use a rooted URI, e.g. `<a href="/path/to/dogs/Bowser">`
  This arguably sucks... we'd like to avoid being tightly-coupled to site structure here.
- ...on the other hand, the page-relative link looks like this: `<a href="../dogs/Bowser">`.
  You may (as do I) find this confusing and klunky. Moreover, the path component for "this page" might not be as simple as "dogs"; it might be something we have to generate, as in the URI `CustomerAccounts/94006748`.

So it just seems best to adopt the convention that "a resource with children has a URI ending in '/'".

But it would be dumb to require the user to know which resources need to end in '/'! Only we should need to know this, and then we redirect the user's browser if necessary.

Here's how we handle it. Given the request

`path/to/dogs`

if neither `Bowser.xml` nor `Bowser.page/source.xml` exist in `content/path/to`, then the client is redirected to

`path/to/dogs/`

Request URIs ending in '/' are matched by an internal redirect to the a relative resource named 'main'. So for this example, that would be `path/to/dogs/main`. This request then is handled by our selector as already described, so if the resource `content/path/to/dogs/main.xml` exists, then it is used. If not, then the selector would next try `content/path/to/dogs/main.page/source.xml`. So we can have a structure like this

```
conent/path/to/dogs/
   main.page/                # URL: 'path/to/dogs/'
      source.xml
      thumbnail-images/
          Bowser.jpg
          Doofus.jpg
   Bowser.page/              # URL: 'path/to/dogs/Bowser'
       source.xml
       # etc. per above
   Doofus.page/              # URL: 'path/to/dogs/Doofus'
       source.xml
       # etc.
   vet_story.xml             # URL: 'path/to/dogs/vet_story
                             #    (this one has no page-specific resources)
```

...and of course this kind of structure can be nested in any arbitrary way.

The resource "main.xml" or "main.page/source.xml" here sort of corrseponds to "index.html" or "index.php" or whatever you might be used to in the Apache DirectoryIndex directive (except that I always called mine "main.php", since it usually isn't really an "index", is it?)

Implementation

The Sitemap

This example assumes that you have a template called "to-html.xslt" that does your page layout/styling.

```
<!--
  ++  Static content
  -->

<map:match pattern="static/**.css">
   <map:read src="{0}" mime-type="text/css"/>
</map:match>
<map:match pattern="static/**.js">
   <map:read src="{0}" mime-type="application/x-javascript"/>
</map:match>
<map:match pattern="static/**.gif">
   <map:read src="{0}" mime-type="image/gif"/>
</map:match>
<map:match pattern="static/**.jpg">
   <map:read src="{0}" mime-type="image/jpeg"/>
</map:match>
<map:match pattern="static/**.png">
   <map:read src="{0}" mime-type="image/png"/>
</map:match>
<map:match pattern="static/**.tiff">
   <map:read src="{0}" mime-type="image/tiff"/>
</map:match>
<map:match pattern="static/**.swf">
   <map:read src="{0}" mime-type="application/x-shockwave-flash"/>
</map:match>

<!--
  ++  HTML pages generated from source documents
  -->

<map:match pattern="content//**/">
  <map:redirect-to uri="content//{1}/main" />
</match>

<map:match pattern="content//**">
  <map:select type="resource-exists">
      <map:when test="content/{1}.xml">
        <map:generate src="content/{1}.xml" />
      </map:when>
      <map:when test="content/{1}.page">
        <map:generate src="content/{1}.page/source.xml" />
      </map:when>
      <map:when test="content/{1}">
        <map:redirect-to uri="/{1}/" global="true" />
      </map:when>
      <map:otherwise>
        <!-- we want an error message, so... -->
        <map:generate src="content/{1}" /> <!-- (non-existent) -->
      </map:otherwise>
  </map:select>
  <map:serialize type="xml"/>
</map:match>

<map:match pattern="**">
  <map:generate src="cocoon:/content//{0}" />
  <map:transform src="to-html.xslt">
     <map:parameter name="path" value="{0} />
  </map:transform>
</map:match>
```

XSLT template to rewrite URIs for local resources

Add this to your stylesheet to rewrite relative URIs that begin w/ "static/" (see the discussion of "Co-Located Resources" above). The `path` parameter is passed in by the stylesheet (see above)

```
<xsl:param name="path" />

<!--
  ++ Static content
 -->
<xsl:template
  match="@href[starts-with(., 'static/')] | @src[starts-with(., 'static/')]"
>
  <xsl:attribute name="{name()}">/static/content/<xsl:value-of
                select="concat($path,'/',substring-after(., 'static/'))"
              /></xsl:attribute>
</xsl:template>
```

A script to automate building directories and links in webapp/static/content

You can invoke this script from an ant task or a makefile.

```
#!/bin/csh -f
#
# Rebuild webapp/static/content.
# (See http://wiki.apache.org/cocoon/SimpleContentModel)
#
cd webapp
set content_dir = $cwd/content

rm -rf static/content
mkdir static/content

foreach dir (`cd $content_dir; find * -name static -type d`)
        set page_dir = $dir:h
        set page_parent = $page_dir:h
        set page_name = $page_dir:t
        set static_dir = static/content/$page_parent

        mkdir -p $static_dir
        pushd $static_dir >& /dev/null
        ln -s $content_dir/$page_dir/static $page_name
        popd >& /dev/null
end
```

That's all there is to it! 🙂

Reader Comments

Feedback welcome... add it here, or email me --ML.

See also:
<<FullSearch>>
