Part 3: Anatomy of the pipeline

Table of contents

With Lenya installed, you're itching to figure out how to get around this thing so you can start creating a website. But, you want to put it all together the way you're used to. Well, bring out the frozen dinners, because this is not your momma's cooking. Instead of opening up a file, adding our HTML, and throwing in some CSS, we'll be working with pipelines, XML, XSLT to get the job done.

1. What are pipelines?

Pipelines aren't a Lenya thing - it's a Cocoon thing. If you want to master Lenya, you'll have to get your hands around Cocoon, and that's not an easy thing. Let me remind you that the articles we write here are not because I'm an expert in Lenya or Cocoon. Far from it. But, we have learned some things along the way that we see being asked over and over, so we think it's important to keep the open-source spirit alive and document what I've learned.

OK, so back to pipelines. Pipelines are basically part of a set of items that can be found in a sitemap of your publication, which include the components, views, resources, etc. We'll save all those for another time. Pipelines are a way to match a request coming to your publication and act on them in some way. So, for example, if you are accessing a particular page within your publication using your web browser, a pipeline can find a match for that request and possibly send back a page for you to view.

2. Minimum Requirements

For a pipeline to work, you'll need to match an incoming request, generate something to be used, and then send it back in a format that is recognizable and can be dealt with easily. Here's an example pipeline:

```
1. <map:pipeline>
2. <map:match pattern="example">
3. <map:generate type="file" src="example.xml"/>
4. <map:serialize type="xml"/>
5. </map:match>
6. </map:pipeline>
```

Let's walk through this line by line. Line 1 starts the definition of the pipeline. Everything starts with "map:" because all of this XML is part of the map namespace defined by Cocoon. Line 2 tries to match an incoming request to see if it looks like "example". If it does, we keep going. If not, this pipeline gets skipped over. Line 3 is the generator. In this case, we'll be generating "stuff" from a file, and that file is example.xml. So, what do we do with this stuff inside the file? Well, we need to send it back to the user making the request for "example" in something they (or it) can understand. In line 4, we're doing just that by using a serializer to take all that stuff in example.xml and sending back to the user as XML.

3. How about something more usable?

OK, admittedly, that was a boring example. The file was already XML, so the only thing the serializer did was probably add in our declaration at the top of the page and send it back to the user. Let's add in some spice and relate it to web pages:

```
1. <map:pipeline>
2. <map:match pattern="test.html">
```

```
            3. <map:generate type="file" src="test.xml"/>
            4. <map:transform type="xslt" src="test2html.xsl"/>
            5. <map:serialize type="html"/>
            6. </map:match>
            7. </map:pipeline>
```

OK, so here, we're trying to match the request for test.html. Now, keep in mind, it could very well be that test.html doesn't exist (and in this case it doesn't). That's OK - we're just matching requests for something, and in return, we can send back whatever we like. Think of it as a virtual link to another file we're creating on the fly.

So, if we do match test.html, we'll grab the contents of the file test.xml, but before we send it back, we'll transform that XML into something else using the transformer (line 4). Using XSLT, we can convert that batch of XML into an HTML page! That's done using the file test2html.xsl. When that's all said and done, off we go to serialize it back to the user, but this time as HTML instead of XML.

We won't have time to show you how the XSL transformation works, but we can throw you over to W3Schools (http://www.w3schools.com/xsl/xsl_languages.asp) and they'll give you a nice intro.

4. The Lenya pipeline

So now that we know the basics, how does Lenya use the pipeline in creating it's pages? Well, it's not too much different. While it looks more complicated, the basics are still there.

Below is the pipeline that is used in the publication-sitemap.xmap file in Lenya's default publication:

```
            1. <map:pipeline>
            2.
<!--/lenyabody-{rendertype}/{publication-id}/{area}/{doctype}/{url}-->
            3. <map:match pattern="lenyabody-*/*/*/*/**">
            4. <map:aggregate element="cmsbody">
            5. <map:part src="cocoon://navigation/{2}/{3}/breadcrumb/{5}.xml"/>
            6. <map:part src="cocoon://navigation/{2}/{3}/tabs/{5}.xml"/>
            7. <map:part src="cocoon://navigation/{2}/{3}/menu/{5}.xml"/>
            8. <map:part src="cocoon://navigation/{2}/{3}/search/{5}.xml"/>
            9. <map:part
src="cocoon:/lenya-document-{1}/{3}/{4}/{page-envelope:document-path}"/>>
            10.</map:aggregate>
            11.<map:transform src="xslt/page2xhtml-{4}.xsl">
            12.<map:parameter name="root"
value="{page-envelope:context-prefix}/{2}/{3}"/>
            13.<map:parameter name="url" value="{5}"/>
            14.<map:parameter name="document-id"
value="{page-envelope:document-id}"/>
            15.<map:parameter name="document-type"
value="{page-envelope:document-type}"/>
            16.</map:transform>
            17.<map:select type="parameter">
            18.<map:parameter name="parameter-selector-test" value="{1}"/>
            19.<map:when test="view">
            20.<map:transform type="link-rewrite"/>
            21.</map:when>
            22.</map:select>
            23.<map:serialize type="xml"/>
            24.</map:match>
            25.</map:pipeline>
```

OK, yikes, we know what you're thinking. But seriously, it's not that bad. We still open with a pipeline tag, we match something, we have this aggregation part which I'll explain in a minute, we transform the results, and after another part I'll explain, we serialize the results back the user. Let's start with the matcher.

5. The Matcher

So, um, what exactly are we matching? Without going into too much and getting you swamped with terminology, we're basically trying to match a whole bunch of things at once. The comment right above the matcher tries to tell you what each of the asterisks are. There's the rendertype (whether you're viewing the page, or editing it), the publication ID (which in this case is "default" for the Default Publication), the area (it could be Authoring, or Live, or Admin, etc.), the document type, and the actual URL of the document.

The document type is pretty interesting. In XML, one document could be for describing a shape, while another could be describing a set of books. It doesn't have to be that way, though. For example, the "homepage" and "xhtml" doctypes provided for you in Lenya are exactly the same, except there's another pipeline that says the top index.html page of the publication will be assigned the doctype of "homepage". It's handy because since most homepages have a different design that the secondary or tertiary pages, you can use a different XSLT file to transform it however you want without having to setup a new pipeline for it. Just think of the possibilities with different doctypes...

6. The Aggregator

So, after we've matched all of those options (the asterisks mean anything and everything), we get to this aggregate tag. Basically, it's a generator like we saw in the previous examples, it's just aggregating the results of the generation from all these sources together as one.

So, Lenya separates out the menu (or navigation of the site), the tabs (all the high-level items in the navigation), the breadcrumb trails on the pages, the search box, and the actual content of the page into separate files. See all those {2}'s, {3}'s, and {5}'s? Each one of those points to the value for that numbered asterisk in the matcher. So, whatever happened to have been in the second asterisk in the matcher, we use that in place of {2}. Simple, no?

7. The Transformer

The transformer is pretty straight-forward. We transform the results of all the aggregated content using the file page2xhtml-{4}.xsl. Except, the {4} is replaced with whatever was in the place of the fourth asterisk, or in this case, the doctype. So, if our doctype was "homepage", we would transform our page using the XSL file page2xhtml-homepage.xsl. If our doctype were "xhtml" (which is what most of the pages are in Lenya), then you would transform it with page2xhtml-xhtml.xsl. See how you can differentiate the design with different transformations and doctypes?

The parameters inside of the transform tag are basically setting up variables to be passed to the XSL file. In this case, we're passing along the root location of the publication (perhaps it's just a / in http://www.someplace.com/, for example), the URL of the publication (like "some/where.html"), as well

as the document ID (the latter half of the URL without the .html extension), and the doctype.

8. The Selector

We haven't seen this one yet, but think of the selector as an if/else statement. You have to tell the selector what you are testing against, then test it against some value, and do something. In this case, we're testing the rendertype (that's the first asterisk, or {1}). If the rendertype is "view", as in we're viewing the page and not editing it, then go through one more transformation, called link-rewrite.

The link-rewrite transformer basically checks where you are, then goes through the contents of the page and rewrites all links in relation to what area you are in. For example, if we are in the Authoring environment in Lenya, then our links could be rewritten to look like "/lenya/default/authoring/some/where.html". If we are in the Live area, they would be rewritten to look like "/lenya/default/live/some/where.html". That way, you just keep track of the organization of the site using the Site tab within Lenya, and Lenya will rewrite your links according to what area you are in when viewing the page so that it all just works!

9. The Serializer

In the end, we serialize everything we've done into XML. So, why XML? Because it's later on in the series of pipelines that the results are serialized again into HTML (or XHTML, if you so choose).

10. Fin

So hopefully that gets you cracking on understanding how Lenya is setup to handle pages. There's no doubt we've exposed you to quite a bit that deserves more explanation, and it will certainly come.