CleanerSiteMapsThroughResources

The Lego in Cocoon

The use of Resources can be a great way to clean up your sitemap, and make it more manageable.
In general it allows you to reuse portions of ready made Pipeline components in the Sitemap

To get a feeling of what it can do for you it maybe makes sense to step through a small example. Suppose you have the following scenario (not so uncommon).

- You have some backend-system producing raw data text files (CSV)
- You have a specific generator that converts this into an XML (table) format
- Based on this format you have two specific stylesheets to (1) select one column out of the XML-table and (2) plot out those values on a SVG canvas
- For demo (no back-end) and testing purposes you have a sample CSV file on your local hard-disk

The challenge is to come up with a clean set of pipelines that can serve the various output formats for both the live data as the local copy.

The URI request space for this could look like this

data/*.*		*.txt	*.xml	*.svg
data/live.*	raw CSV format of the live data	xmlized version of the live data	graph-plot of the live data	
data/local.*	raw CSV format of the local copy	xmlized version of the local copy	graph-plot of the local copy	

Which clearly shows the two dimensions in the problem:

- Multiple output-formats: txt, xml, svg, jpg
- Various input-sources that should run through similar pipeline snippets: live-scraping versus local-copy

The most naive attempt at attacking the problem would be to (match for and) define the various pipelines one by one:

```
<map:match pattern="data/live.txt">
  <map:read mime-type="text/plain" src="http://csv-server.domain/getData" />
</map:match>

<map:match pattern="data/live.xml">
  <map:generate type="myCSVGenerator" src="http://csv-server.domain/getData" />
  <map:serialize type="xml" />
</map:match>

<map:match pattern="data/live.svg">
  <map:generate type="myCSVGenerator" src="http://csv-server.domain/getData" />
  <map:transform src="xsl/datafilter.xsl" />
  <map:transform src="xsl/data2svg.xsl" />
  <map:serialize type="svgxml"/>
</map:match>

<map:match pattern="data/live.jpg">
  <map:generate type="myCSVGenerator" src="http://csv-server.domain/getData" />
  <map:transform src="xsl/datafilter.xsl" />
  <map:transform src="xsl/data2svg.xsl" />
  <map:serialize type="svg2jpeg"/>
</map:match>
```

Mind:

1. that there is already a considerable amount of duplication in these pipelines
2. and that these four need to be duplicated for the similar case of the local copy: data/local.*

The duplicate typing assures a high error probability when things start to change (one of the stylesheets, the location of the CSV server backend, ...)

Let us use the duplicate typing as a guide for defining our Resources.

Duplication 1

The SVG and JPEG output pipelines only differ in the serializer, which means we could hope for one single component that wraps up the complete mutual start of the pipelines in question. Such component would be (Starting a pipeline?) taking up the role of a "generator" that looks like this:

```
<map:resource name="generate-data-svg" >
    <map:generate type="myCSVGenerator" src="http://csv-server.domain/getData" />
    <map:transform src="xsl/datafilter.xsl" />
    <map:transform src="xsl/data2svg.xsl" />
</map:resource>
```

Resulting in modified pipelines that are as simple as:

```
<map:match pattern="data/live.svg">
  <map:call resource="generate-data-svg" />
  <map:serialize type="svgxml"/>
</map:match>

<map:match pattern="data/live.jpg">
  <map:call resource="generate-data-svg" />
  <map:serialize type="svg2jpeg"/>
</map:match>
```

Duplication 2

Since the SVG and XML pipelines share the same specific generator there is a similar step more to take by letting both this new resource and the xml-pipe call upon a resource that hides away this generator (and it's hardcoded source.)

Now the combination of:

```
<map:resource name="generate-data-xml" >
  <map:generate type="myCSVGenerator" src="http://csv-server.domain/getData" />
</map:resource>

<map:resource name="generate-data-svg" >
  <map:call resource="generate-data-xml" />
  <map:transform src="xsl/datafilter.xsl" />
  <map:transform src="xsl/data2svg.xsl" />
</map:resource>
```

Allows for even more similarity in the pipelines:

```
<map:match pattern="data/live.xml">
  <map:call resource="generate-data-xml" />
  <map:serialize type="xml"/>
</map:match>

<map:match pattern="data/live.svg">
  <map:call resource="generate-data-svg" />
  <map:serialize type="svgxml"/>
</map:match>

<map:match pattern="data/live.jpg">
  <map:call resource="generate-data-svg" />
  <map:serialize type="svg2jpeg"/>
</map:match>
```

While this does not offer us that much gained economics in the typing effort, it does limit once more the occurance of the part that is the most probable to change (since it is out of our control: the URI-location (server and path) of the CSV file).

Duplication 3

In fact, focussing on that bit we are bound to come up with the last remaining occurance lingering still in the untouched pipeline for the data/live.txt (the read of the raw data)

The question to ask is which 'role' should be taken up by the resource that could fit into both pipelines that still require this field. Given the end-to-end coverage of the single {<map:read>} component in the txt pipeline there is however no smaller granularity then the full pipe to consider. This in turn forces us to reconsider all pipelines to be redefined as resources with end-to-end responsibility. The source to start from, now becomes a parameter `input-src` to these Resources:

```
<!--
    | resources that behave as full-blown end-to-end pipes
     -->
  <map:resource name="pipe-data-txt">
    <map:read mime-type="text/plain" src="{input-src}" />
  </map:resource>

  <map:resource name="pipe-data-xml">
    <map:call resource="generate-data-xml" >
      <map:parameter name="input-src" value="{input-src}" />
    </map:call>
    <map:serialize type="xml" />
  </map:resource>

  <map:resource name="pipe-data-svg">
          <map:call resource="generate-data-svg" >
                  <map:parameter name="input-src" value="{input-src}" />
          </map:call>
    <map:serialize type="svgxml"/>
  </map:resource>

  <map:resource name="pipe-data-jpeg">
    <map:call resource="generate-data-svg" >
      <map:parameter name="input-src" value="{input-src}" />
    </map:call>
    <map:serialize type="svg2jpeg"/>
  </map:resource>
```

Not precisely a small amount of typing, but in return it cleans up the pipelines to a mere:

```
    <map:match pattern="data/live.*">
      <map:call resource="pipe-data-{1}">
        <map:parameter name="input-src"
               value="http://csv-server.domain/getData" />
      </map:call>
    </map:match>
```

And the real gain is yet to come.

Duplication 0

Although we started the other way around, we finally come down to the most obvious duplication in the original setup. i.e. The fact that the local copy (2nd dimension) needed to go through the complete same set of considerations. The only variable in the two cases is the source to start from.

Since we factored that one out to become a simple parameter, the complete 'local' case is dealth with by reusing the resources we already have:

```
    <map:match pattern="data/local.*">
      <map:call resource="pipe-data-{1}">
        <map:parameter name="input-src"
               value="local/sample.txt" />
      </map:call>
    </map:match>
```

Aditional notes:

- When diving into adventures like these, you might want to consider careful attention in ResourceNaming
- Attacking the issue through nested Matchers is a valid option. However, things might not immediately become more readable.
- The followed 'one step at a time'-procedure is also advocated on SurvivalTips (Look for Proceed in "baby steps")
- The test/examples on this page were done with Cocoon 2.0.4 using the TreeProcessor.

Known limitations:

- Not all attributes in the sitemap are subject to variable-resolving. This means that their values can't be parametrized using {sitemap-var} syntax. One notable example is the @status-code of the map:serializer.