How to customize processing of html source

Explains how to do additional processing of html source documents.

Table of contents

1 Intended Audience

Users who want to integrate HTML-pages that require custom adjustments and everybody who wants to learn more about Forrest's pipelines in general.

2 Purpose

Integrating legacy HTML pages is a common task when migrating existing websites to Forrest. This document explains how to implement custom processing which is required when Forrest's standard pipeline for html does not suffice.

3 Prerequisites

To follow these instructions you will need:

1. Know how to use a project sitemap.
2. A basic understanding of Cocoon's pipelines and their components. You should know about matchers, generators, transformers and serializers and have a rough idea how they work together in a pipeline. A good place to start learning about Cocoon is Understanding Apache Cocoon. The Forrest Sitemap Reference will also be helpful.
3. If you want to follow the examples right through your Forrest, a current version of Forrest installed on your system and read-access to Forrest's program directory is necessary.
4. If you plan on creating your own custom processing for html-pages, you'll also need write access to your project directory.

4 Understanding the HTML-Pipeline

The first part of this howto explains the html pipeline, so as to provide the background to enable you to add additional processing for legacy html documents. If you already know how pipelines work, then skip to the section about Customizing the html pipeline.

4.1 Driven by Example

The best way to learn about Forrest pipelines is follow the processing of an imaginary request through the forrest machinery.

So let's see what happens, when a client asks Forrest to serve the document 'http://my.domain.org/mytest/mybad.html'.

4.2 Finding the Sitemap

Like all applications based on Apache Cocoon, each request for a given document is processed by searching a sitemap for a matching processing pipeline. With Forrest, this core sitemap is found in the file 'main/webapp/sitemap.xmap' in Forrest's program directory.

Open the file 'main/webapp/sitemap.xmap' in Forrest's program directory with a text editor of your choice.

> **Note:**
> Any simple text editor will suffice, since the XML in this file is quite simple in structure and easy to read.

Search for `map:sitemap` to find the start of the Sitemap.

This sitemap is the starting point for all requests. So even if there are other sitemaps (which we will see later on), we always start looking for a matching pattern right here.

4.3 Find the Beginning of the Pipelines Section

Modular as everything else in Cocoon, Forrest's sitemap starts with a long list of declarations for all the components used later on. We can safely ignore these at the moment.

So let's skip right to the start of the Pipelines-Section. Search for `map:pipelines`

Within the pipelines-element you will find a long list of pipeline-Elements (no trailing 's'), each one of them defining a processing pipeline within Forrest.

When handling a request, Forrest will check the Pipelines from top to bottom until it encounters a Pipeline that will take care of our request.

4.4 Looking for a Match

Like all Cocoon applications, Forrest knows which pipeline to use for processing a certain request by looking at the entry criteria for each pipeline it comes across. This can be a match against a given pattern, the test if a certain files exists or one of many other possible tests that Cocoon supports.

To better know what we are talking about, let's follow Forrest down the list to the Test for the First Pipeline:

Scroll down to the line `<map:match pattern="cprofile.*">`

Here you can see that very specialized matches need to occur early in the sitemap. The requested file (and pathname) is compared to a pattern 'cprofile.*' that would match if our request started with 'cprofile', endet with any kind of extension and had no pathname. Since it doesn't, we don't have a match and need to keep looking.

Skip forward until you find `<map:match pattern="**/*.html">` .

> **Note:**
>
> While scrolling down you may have noticed the match-pattern `<map:match pattern="*.html">` a couple of lines earlier. This will not match our request since *.something in Cocoon matches only files in the root directory.

4.5 Processing in the '**/*.html' Pipeline

Let's take a quick look at this pipeline to understand what's happening here:

```
<map:match pattern="**/*.html">
        <map:aggregate element="site">
          <map:part src="cocoon:/skinconf.xml"/>
          <map:part src="cocoon:/build-info"/>
          <map:part src="cocoon:/{1}/tab-{2}.html"/>
          <map:part src="cocoon:/{1}/menu-{2}.html"/>
          <map:part src="cocoon:/{1}/body-{2}.html"/>
        </map:aggregate>
        <map:call resource="skinit">
          <map:parameter name="type" value="transform.site.xhtml"/>
          <map:parameter name="path" value="{0}"/>
        </map:call>
      </map:match>
```

In the first part of this pipeline, the aggregate-element assembles information required to build a Forrest page with menu and tabs from different sources. Then the call to the skinit-resource picks up the aggregated info and generates a page in the style of the current skin. That's easy, isn't it?

Well, the complex part begins, when we take a closer look at the sources of the aggregation.

```
<map:part src="cocoon:/{1}/body-{2}.html"/>
```

This mysterious element is most easily explained as a secondary request to the Forrest system.

The 'cocoon:'-part is called a pseudo-protocol and tells the processor to ask Forrest for the resource named behind the colon, process that request and feed the output as input back into our pipeline. (The 'pseudo' goes back to the fact that unlike 'http' or 'ftp', which are real protocols, you can use cocoon: only within the cocoon environments as only Cocoon will know what to do with it.)

So even though we have already seen the end of our pipeline (the skinning), we still don't know, what goes into the skinning and where it comes from. To find out, we have to look at the sources of the aggregation.

4.6 Following the Pseudo-Protocols

To find out what goes into our aggregation, we'll need to look at the pipeline that is called by

```
<map:part src="cocoon:/{1}/body-{2}.html"/>
```

To do that, it's always a good idea to write down what this call actually looks like when all the variables are replaced by real values. A safe way to do that is to look at the matcher to start with, build a list of the numbered variables and their meaning in the current context and then assemble the actual expression(s) from it.

In our example the matcher pattern `**/*.html` is applied to the request-name `mytest/mybad.html`, so we have three variables altogether:

%0	mytest/mybad.html	the whole pathname
%1	mytest/	the first match
%2	mybad	the second match

If we insert that into

```
<map:part src="cocoon:/{1}/body-{2}.html"/>
```

we get

```
<map:part src="cocoon:/mytest/body-mybad.html"/>
```

As you can easily tell, we are suddenly calling for a whole new document. Let's see where that takes us:

4.7 Second Call for Content

Processing of cocoon-calls is not much different from normal requests by a client. When you launch a call like

```
<map:part src="cocoon:/mytest/body-mybad.html"/>
```

Forrest will once again start searching its main sitemap from the beginning and look for a pipeline to match that call.

Search for `**body-*.html` from the beginning of the sitemap to see where we find our next match.

4.8 First Match for '**body-*.html'

Our first match is different to the previous ones because there is a second condition placed inside the matcher. Doing the replacements

```
<map:select type="exists">
         <map:when test="{lm:project.{1}{2}.ehtml}">
```

we quickly discover that there can't be a file of that name in the project-directory.
(The variable '{lm:project.}' is always replaced with the name of your project directory that you can change in the 'forrest.properties'-file.)

So we have a pipeline, but it doesn't do anything. In this case Forrest will simply keep looking for the next match further down.

4.9 Second Match for '**body-*.html'

Continue searching downwards for '**body-*.html' in the sitemap-file.

Looking at the pipeline that handles the request, we see that the cocoon-protocol is once again invoked

```
<map:generate src="cocoon:/{1}{2}.xml"/>
```

this time as a direct generator of input for our pipeline.

So once again we ask Forrest to process a request for content. To know what matcher to look for, let's first expand the variables:

In our example the matcher pattern `**body-*.html` is applied to the request-name `mytest/body-mybad.html`. Which means that we have three variables altogether:

%0	mytest/body-mybad.html	the whole pathname
%1	mytests/	the first match
%2	mybad	the second match

If we insert that into

```
<map:generate src="cocoon:/{1}{2}.xml"/>
```

we get

```
<map:generate src="cocoon:/mytests/mybad.xml"/>
```

4.10 Third Call for Content

So let's scan the main sitemap looking for a match for '/mytests/mybad.xml'.

We find it in the pattern `<map:match pattern="**.xml">` , which is the standard handling for all xml-requests.

Since our request fulfils none of the secondary criteria in this pipeline, it falls right through to the map:mount-element at the end:

```
<map:mount uri-prefix="" src="forrest.xmap" check-reload="yes" />
```

which makes Forrest load and process a secondary sitemap, the file 'forrest.xmap' in the same directory.

Open the file 'forrest.xmap' and continue the search for a matching pattern.

Our search first leads us to the matcher `<map:match type="wildcard" pattern="**.xml">` with a number of submatchers embedded into it.

The first one, `<map:match type="wildcard" pattern="**.xml">` , would handle input coming from Forrest plugins. We won't go into details here.

All further matchers `<map:match type="i18n" pattern="{properties:content.xdocs}{1}.*.xml">` . implement what we call a cascade of matchers. By testing for the existence of one source file after another Forrest will use and process the first of the tested source-formats found.

> **Note:**
> Using the i18n-matcher here, Forrest will do a lot more than just finding content in one of many possible source formats. It will also make sure that the the proper language version of the conent (if there are several) is used. Read more about this matcher http://cocoon.apache.org/2.1/apidocs/org/apache/cocoon/matching/LocaleMatcher.html

Checking each matcher in turn you will find that a pipeline that tests for the existence of the file with different extensions. '.html' is third in this list and leads to the processing steps shown below:

4.11 html-Default Processing

The default processing of html-files consists of three processing steps:

1. `<map:generate src="{source}" type="html" />`
   Using Cocoons html-generator, Forrest reads the html-document from file and uses JTidy to clean up and convert it to xml (which is required for all processing in Cocoon pipelines). At the output of this transformer we will have a valid XHTML-document althought it might still contain some unwanted elements. We'll deal with those later (see When to customize).
2. `<map:transform src="{lm:transform.html.document}" />`
   Using the standard stylesheet 'html-to-document.xsl', this XHTML is transformed into Forrest's Standard Document Format. (refer to a detailed explanation of locationsmaps to understand exactly how and where that stylesheet is found!)

3. `<map:serialize type="xml-document"/>`
   Finally the document is serialized as XML and returned to the calling pipeline.

As a result, we now hand back the content of the html-document in Forrest standard document format to the calling pipeline

> Note:
> To look at the output of this pipeline you can simply point you browser to 'http://localhost:8888/mytest/mybad.xml' (assuming that you are currently running Forrest on your machine and there is an html-page of that name).

4.12 Returning to the '**body-*.html'-Pipeline

On returning into the '**body-*.html'-Pipeline (in sitemap.xmap), processing continues with the next components in this pipeline:

- **idgen** will generate unique IDs for all elements that need to be referenced within a page (mainly headlines).
- **xinclude** would process any xinclude statements in the source. Since HTML does not support this mechanism, nothing happens in our example.
- **linkrewriter** adjusts links between pages so that they will still work in the final Forrest output directory structure. It also resolves any special Forrest links.
- The final transformation, `<map:transform src="{lm:transform.html.broken-links}" />` as the name suggests, will take care of reporting broken site-links.
- The call to 'skinit' prepares the page body for presentation within the selected skin.

> Note:
> To look at the output of this pipeline you can simply point you browser to 'http://localhost:8888/mytest/body-mybad.html' (assuming that you are currently running Forrest on your machine and there is an html-page of that name).

4.13 Returning to the '**/*.html'-Pipeline

At the end of this pipeline, processing returns the results into the aggregation section of the '**/*.html' Pipeline, merges it with other data, skins and serializes for presentation in the requesting client.

4.14 Customizing the html pipeline

In this last part of this document, we will show how to customize the HTML-pipeline to add your additional steps to the default processing.

4.14.1 When to customize?

The html-Pipeline in Forrest is designed to be able to also integrate legacy html-Pages in a Forrest project. In doing so, it will fix common markup errors and convert html to Forrest's intermediate document format.

Due to the nature of html as presentational markup, there is no way this automated process can identify elements in your pages that are not required or even unwanted in the Forrest environment.

A good example are pages from sites where the navigational elements (menus, tabs etc.) are embedded in the html of each page. Since Forrest can't know what is an unwanted menu and what belongs to the page body that you want to keep, you will need to customize the process to remove elements that are

not needed. If you don't, then you will see the original page from your legacy website, menu and all, embedded in your new Forrest site.

4.14.2 How to customize?

To add your own custom processing for a group of pages, you will enhance the project sitemap with pipelines that process your documents according to your specifications.

This project sitemap is located in the file 'src/documentation/sitemap.xmap' in your Forrest project directory and is created automatically whenever you seed a new project. At this point it already contains a few examples for custom pipelines.

To add your own custom processing, edit the file and add a new pipeline to the project sitemap. Since the project sitemap is loaded into the main sitemap right at the top (search for 'This is the user pipeline'), your pipeline intercepts practically all of Forrest's standard pipelines.

4.14.3 What to intercept?

Where to intercept standard processing is really a matter of your choice. A good rule is to replace as little standard handling as possible so that future changes in the Forrest architecture are less likely to break your application.

In our case all we need to do is add a transformation that removes all the unwanted elements. The best place to do this would be right after the generator has converted our document to xhtml.

However, since we can only replace a complete pipeline, we'll create a new pipeline that intercepts **.xml for our pages, copy the steps the original processor is doing and add a transformation of our own to it.

4.14.4 Intercept pattern

Take great care when intercepting very basic pipelines. Instead of designing our custom pipeline to match the original **.xml pattern, try to narrow your matcher down to something that will only match your pages.

If all your pages are located in a directory called mytest, then use a matcher like '/mytest/*.xml' to avoid highjacking the processing for all the other requests.

Add a new pipeline in your project sitemap and set the matcher to '/mytest/*.xml'.

The new pipeline should look like this and does nothing so far.

```
<!--Custom Pipeline for my bad html-pages-->
<map:pipeline>
    <map:match pattern="mytest/*.xml">

    </map:match>
</map:pipeline>
```

Open the 'forrest.xmap', search for `<map:match type="i18n" pattern="{properties:content.xdocs}{1}.*.html">` , copy the three lines for handling *.html files and paste them into your new pipeline.

```
<!--Custom Pipeline for my bad html-pages-->
<map:pipeline>
    <map:match pattern="mytest/*.xml">
      <map:generate src="mytest/{1}.html" type="html" />
```

```
      <map:transform src="{lm:transform.html.document}" />
      <map:serialize type="xml-document"/>
    </map:match>
</map:pipeline>
```

Your custom pipeline will now behave exactly like the standard html-handler. All that is left to be done is creating the custom transformation and adding it the pipeline.

Design and test a new XSL-Transformation that removes the unwanted elements and save it in your project's stylesheet directory, usually src/documentation/resources/stylesheets (defined in project.stylesheets-dir of forrest.properties), which is central storage for all stylesheets in a project.

Add the new transformation as a new line, straight after the generator, and save the changes.

```
<!--Custom Pipeline for my bad html-pages-->
<map:pipeline>
    <map:match pattern="mytest/*.xml">
      <map:generate src="mytest/{1}.html" type="html" />

      <map:transform src="{properties:resources.stylesheets}/fixMyBadHTML.xsl"/>

      <map:transform src="{lm:transform.html.document}" />
      <map:serialize type="xml-document"/>
    </map:match>
</map:pipeline>
```

Done! You have just added your own custom-processing. Don't forget to view the pages to verify that it is working properly.

> **Note:**
>
> Our pipeline does not exactly do what the original pipeline does. To make things easier, we omitted the internationalization part in our pipeline. So if you need to create multi-language sites, make sure that you adjust your pipeline accordingly.