The Sitemap (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. The Sitemap

1.1. Introduction

This document describes the Cocoon sitemap concept in full details, why it's there, what it does, how it works and how you can use it.

See the User documentation for details about configuration of each sitemap component. See the Example sitemap snippets. Explore each sitemap.xmap in the distribution, to see how the various situations are handled.

1.2. Goals

The goal of the sitemap is to allow non-programmers to create web sites and web applications built from logic components and XML documents.

It finds inspiration from both Apache's httpd.conf/.htaccess files as well as from Servlet API 2.2 WAR archives. It uses concepts such as Cascading from W3C CSS, as well as declarative approaches integrated into the W3C XSLT language. It also uses some element/attribute equivalence patterns used in W3C RDF.

The following goals were identified as engineering constraints:

1. minimal verbosity is of maximum importance.
2. the schema should be sufficiently expressive to allow learning by examples.
3. sitemap authoring should not require assistive tools, but be sufficiently future-compatible to allow them.
4. sitemaps must scale along with the site and should not impose growth limitation to the site as a whole nor limit its administration with size increase.
5. sitemaps should contain all the information required to Cocoon to generate all the requests it receives.
6. sitemaps should contain information for both dynamic operation as well as offline generation.
7. uri mapping should be powerful enough to allow every possible mapping need.
8. basic web-serving functionalities (redirection, error pages, resource authorisation) should be provided.
9. sitemaps should not limit Cocoon's intrinsic modular extensibility.
10. resources must be matched with all possible state variables, not only with URI (http parameters, environment variables, server parameters, time, etc...).
11. sitemaps should embed the notion of "semantic resources" to be future-compatible with semantic crawling and indexing.
12. sitemaps should be flexible enough to allow a complete web site to be built with Cocoon.

1.3. The Structure

The Sitemap has the following general structure:

```
<?xml version="1.0"?> <map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
<map:components/> <map:views/> <map:resources/> <map:action-sets/> <map:pipelines/>
</map:sitemap>
```

1.4. The <map:sitemap>

<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

The default namespaces are used mainly for versioning, instead of using attributes such as version="1.0" which could create confusion. People are used to writing URIs with no spelling mistakes, while versioning could be used for their own sitemap versions and this might break operation.

The versioning schema will be "major.minor" where major will be increased by one each time a new release breaks backwards compatibility, while minor is increased each time a change has been made that doesn't create backwards compatibility problems.

1.5. The <map:components>

<map:components> <map:generators/> <map:transformers/> <map:serializers/> <map:readers/> <map:selectors/> <map:matchers/> <map:actions/> <map:pipes/> </map:components>

1.5.1. Common Attributes of Components

All components have some common attributes. The list below will show and explain them:

name
Gives the component an identifying name by which it may be referenced in the pipeline section.

src
Specifies the class implementing this component.

1.5.2. Component Parameters

All components are configured with parameters specified in their child elements at component instantiation time. The following example shows how to specify a <use-request-parameter> parameter for an XSLT transformation component:

<map:components> <map:transformer name="xslt" src="org.apache.cocoon.transformation.TraxTransformer"> <!-- This is a parameter to the transformer component --> <use-request-parameters>false</use-request-parameters> </map:transformer> </map:components>

The name and meaning of the parameters are dependent on the component.

1.5.3. Generators

A Generator generates XML content as SAX events and initializes the pipeline processing.

<map:generators default="file"> <map:generator name="file" src="org.apache.cocoon.generation.FileGenerator"/> <map:generator name="dir" src="MyDirGenerator"/> <map:generator name="serverpages" src="org.apache.cocoon.generation.ServerPagesGenerator"> ... </map:generator> </map:generators>

The default attribute on <map:generators> specifies the type of generator to use if none is specified in a pipeline.

1.5.4. Transformers

A Transformer transforms SAX events into other SAX events.

<map:transformers default="xslt"> <map:transformer name="xslt" src="org.apache.cocoon.transformation.TraxTransformer"> <use-request-parameters>false</use-request-parameters> <use-browser-capabilities-db>false </use-browser-capabilities-db> </map:transformer> <map:transformer name="xinclude" src="org.apache.cocoon.transformation.XIncludeTransformer"/> </map:transformers>

The default attribute on <map:transformers> specifies the type of transformer to use if none is specified in a pipeline.

1.5.5. Serializers

A Serializer transforms SAX events in binary or char streams for final client consumption.

<map:serializers default="html"> <map:serializer name="html" mime-type="text/html" src="org.apache.cocoon.serialization.HTMLSerializer"> <doctype-public>-//W3C//DTD HTML 4.0 Transitional//EN </doctype-public> <doctype-system>http://www.w3.org/TR/REC-html40/loose.dtd </doctype-system> <omit-xml-declaration>true</omit-xml-declaration> <encoding>UTF-8</encoding> <indent>1</indent> </map:serializer> <map:serializer name="wap" mime-type="text/vnd.wap.wml" src="org.apache.cocoon.serialization.XMLSerializer"> <doctype-public>-//WAPFORUM//DTD WML 1.1//EN </doctype-public> <doctype-system>http://www.wapforum.org/DTD/wml_1.1.xml </doctype-system> <encoding>UTF-8</encoding> </map:serializer> <map:serializer name="svg2jpeg" mime-type="image/jpeg" src="org.apache.cocoon.serialization.SVGSerializer"> <parameter name="background_color" type="color" value="#00FF00"/> </map:serializer> <map:serializer name="svg2png" mime-type="image/png" src="org.apache.cocoon.serialization.SVGSerializer"> </map:serializer> </map:serializers>

The default attribute on <map:serializers> specifies the type of serializer to use if none is specified in a pipeline.

1.5.6. Selectors

A Selector is used to implement basic conditional logic (if-then-else or switch) inside the sitemap. As can be seen in the Selector interface, this functionality decomposes into the ability to evaluate a boolean condition.

<map:selectors default="browser"> <map:selector name="load" src="org.apache.cocoon.selection.MachineLoadSelector"> ... </map:selector> <map:selector name="user" src="org.apache.cocoon.selection.AuthenticationSelector"> ... </map:selector> <map:selector name="browser" src="org.apache.cocoon.selection.BrowserSelector"> <browser name="explorer" useragent="MSIE"/> <browser name="lynx" useragent="Lynx"/> <browser name="mozilla5" useragent="Mozilla/5"/> <browser name="mozilla5" useragent="Netscape6/"/> <browser name="netscape" useragent="Mozilla"/> ... </map:selection> </map:selection>

The default attribute on <map:selectors> specifies the type of selector to use if none is specified in a pipeline.

1.5.7. Matchers

A Matcher maps a pattern to a resource.

```
<map:matchers default="wildcard"> <map:matcher name="wildcard"
src="org.apache.cocoon.matching.WildcardURIMatcher"> ... </map:matcher> <map:matcher
name="regexp" src="org.apache.cocoon.matching.RegexpURIMatcher"> ... </map:matcher>
</map:matchers>
```

The default attribute on <map:matchers> specifies the type of matcher to use if none is specified in a pipeline.

1.5.8. Actions

An Action is a sitemap component that manipulates runtime parameters based on request and application state. An Action's result is available in the sitemap as map of name/value pairs. Detailed information on actions may be found in the Actions section.

```
<map:actions> <map:action name="add-employee"
src="org.apache.cocoon.acting.DatabaseAddAction"/> <map:action name="locale"
src="org.apache.cocoon.acting.LocaleAction"/> <map:action name="request"
src="org.apache.cocoon.acting.RequestParamAction"/> <map:action name="form-validator"
src="org.apache.cocoon.acting.FormValidatorAction"/> </map:actions>
```

1.6. The <map:views>

The <map:view> element defines different view of the site. Views are defined independent of pipelines and might be used with any pipeline defined in the sitemap. For more on views read the Views section.

```
<map:views> <map:view name="content" from-label="content"> <map:serialize type="xml"/>
</map:view> <map:view name="links" from-position="last"> <map:serialize type="links"/>
</map:view> </map:views>
```

1.7. The <map:resources>

The <map:resource> element is used as a placeholder for pipelines that are used several times inside the document.

```
<map:resources> <map:resource name="Access refused"> <map:generate
src="./error-pages/restricted.xml"/> <map:transform src="./stylesheets/general-browser.xsl"/>
<map:serialize status-code="401"/> </map:resource> </map:resources>
```

1.8. The <map:action-sets>

The <map:action-set> element is used to arrange actions in groups (See the Actions section for details).

```
<map:action-sets> <map:action-set name="employee"> <map:act type="add-employee"
action="Add"/> <map:act type="del-employee" action="Delete"/> <map:act
type="upd-employee" action="Update"/> <map:act type="sel-employee" action="Select"/>
</map:action-set> </map:action-sets>
```

2. Pipelines

2.1. Introduction

Cocoon relies on the pipeline model: an XML document is pushed through a pipeline, that exists in several transformation steps of your document. Every pipeline begins with a generator, continues with

zero or more transformers, and ends with a serializer. Beside this normal processing each pipeline may define its own error handling, too.

Beside using the various components, you can use matchers, and selectors to choose a specific pipeline processing. Aggregation allows you to build a hierarchy of pipelines.

Using views allows you to define exit points in a pipeline.

2.2. Define a Pipeline

Defining a pipeline is simple. Just write a map:pipeline element inside the map:pipelines element.

Bernhard Huber: Explain optional attribute internal-only

2.3. Pipeline Elements

Having defined a pipeline you can use following sitemap elements:

Element	Description
map:match	Selects pipeline processing depending on matching
map:select, map:when, map:otherwise	Selects pipeline processing depending on selecting
map:mount	Mounts a sub sitemap
map:redirect-to	Redirects to a another URI
map:call	Goto another pipeline fragment
map:parameter	Defines additional parameters for the sitemap components
map:act	Peform action processing
map:generate	Defines the generation step
map:aggregate, map:part	Defines an alternate generation step by mergine pipelines
map:transform	Defines zero or more transformation steps
map:serialize	Defines the final serialization step
map:handle-errors	Handles processing errors

The usage of these sitemap elements is explained in the following sections in more detail.

2.4. Matching

These powerful sitemap components allow Cocoon to associate a pure "virtual" URI space to a given set of instructions that describe how to generate, transform and present the requested resource(s) to the client.

See also Implementing Matchers And Selectors

2.5. Selecting And Testing

Selectors in Apache Cocoon have a role similar to matchers while being more flexible. Like matchers they are designed to test something against a part of the environment (the request URI, headers, cookies and so on), but unlike matchers they can be active decision driving components. A matcher allows only for simple "yes/no" decisions: the match can be succesful or not, if it is the pipeline is executed, if not it's simply ignored. Selectors go a step further allowing for more complex use cases, where there is need for a decision to be made according to a multiple chance scenario. In short you can think of matchers as an "if" statement, while selectors have all the power of an "if-else if-else" or "switch-case" construct. The selector syntax is similar will be familiar to people using the XSLT <xsl:test> statement.

See also Implementing Matchers And Selectors

2.6. Acting

Apache Cocoon has a rich set of tools for publishing web documents, and while XSP and Generators provide alot of functionality, they still mix content and logic to a certain degree. The Action was created to fill that gap. Because the Cocoon Sitemap provides a mechanism to select the pipeline at run time, we surmised that sometimes we need to adjust the pipeline based on runtime parameters, or even the contents of the Request parameter. Without the use of Actions this would make the sitemap almost incomprehensible.

See also Creating And Using Actions

2.7. Generating

A generator is the starting point of an xml pipeline. It generates XML content as SAX events and initialize the pipeline processing.

See also Generators in Cocoon.

2.8. Aggregating

An aggregator produces XML content. It is composed of one or more parts, each of which defined by an XML source. During pipeline processing, all parts of an aggregator are merged. The name of the parent element which contains the merged XML content from each part is defined by the value of the map:aggregate's attribute called element.

You can define an aggregator in places where you define a generator. Defining an aggregator is simple. The example belows defines an aggregate, the merged in parts will become children of element the-aggregated-content.

```
<map:aggregate element="the-aggregated-content"> <!-- define your map:parts here -->
</map:aggregate>
```

Defining an aggregator implicits defining the parts building up the content of an aggregate.

Define parts inside of an aggregate. You can define as source of a part a URL. The following list of examples summarizes some useful part sources:

- Use http://foo/bar to merge in xml content via http protocol, received from machine foo.
- Use context://servlet-context-path/foo/bar to merge in xml content from the servlet context.
- Use cocoon:/current-sitmap-pipeline/foo/bar to merge in xml content from the current sitemap. The appropriate pipeline is selected matching current-sitemap-pipeline.
- Use cocoon://root-sitmap-pipeline/foo/bar to merge in xml content from the root sitemap. The appropriate pipeline is selected matching root-sitemap-pipeline.

- Use resource://class-path-context/foo/bar to merge in xml content from the classpath.
- Use jar:http://www.foo.com/bar/jar.jar!/foo/bar to merge in xml content coming from a jar via http connection.
- Use file:///foo/bar to merge in xml content from the filesystem.
- Use xmldb:<your driver here>://your.xmldb.host/db/foo/bar to merge in xml content from a XML:DB compliant database.
- Depending on your setup you may use nfs:, jndi: protocols, too.

Defining a part element of an aggregate is simple. A part element specifies by its src attribute the source of the xml content.

The following example is taken from the documentation sitemap. The xml content of pipelines matching book-*.xml, and body-*.xml is aggregated having root element site.

```
<map:match pattern="*.html"> <map:aggregate element="site"> <map:part src="cocoon:/book-{1}.xml"/> <map:part src="cocoon:/body-{1}.xml"/> </map:aggregate> ...
```

The aggregated xml content may look like this:

```
<site> <menu> <!-- content of book xml --> ... </menu> <document> <!-- content of body xml --> ... </document> </site>
```

2.9. Transforming

A transformer is the central point in the pipeline. It transform SAX events in SAX events.

See also Transformers in Cocoon.

2.10. Serializing

A serializer is the end point of an xml pipeline. It transform SAX events in binary or char streams for final client consumption.

See also Serializers in Cocoon.

2.11. Handling Errors

Each pipeline may define its error handling. A error handler is specialized pipeline having a pre-configures generator. Each error handler uses the generator named !error-notifier!. Thus you do not define a generator inside the error handler. Beside this issue you configure the error handler like a pipeline. Thus you can choose your transformer, and serializer, and all other features of pipeline processing.

You may define the error handler as last element of a pipeline. A error handler may have a type attribute describing which error is handled. By default an error handler handles status code 500.

The following example defines an error handler, transforming the content of the error content by the xslt, and i18n transformer, and finally serializing html.

```
<map:pipeline> ... <map:error-handler type="500"> <map:transform type="xslt" src="error2html"/> <map:transform type="i18n"/> <map:serialize/> </map:error-handler> </map:pipeline>
```

2.12. Viewing

Basically, views let you specify exit points of your pipelines that are taken whenever a particular view is requested. The processing continues with the definitions in the requested view. The advantage over

selectors that could achieve the same is, that these exit points are not necessarily declared for each pipeline individually, but once per sitemap.

2.13. Redirecting

Redirecting forwards the the request. You may externally send an redirect response to the client. The behaviour is controlled by using the approriate attributes of the element redirect-to.

The attribute uri defines the target of redirect. The target is sent as redirect response to the client. The optional attribute session specifies, if the redirect should happen inside of a session or not. Setting session to yes, or true will persist a session across the redirect response. Use enable the session option if you use http session within your web application.

The following example redirects to a welcome page:

<map:pipeline> <map:match pattern=""> <map:redirect-to uri="welcome"/> </map:match> <map:match pattern="welcome"> ... </map:pipeline>

2.14. Calling resources

Calling resources is dissimilar to redirects as the client does not notice this.

When calling a resource, arbitrary parameters can be specified. They will be available to the processing later on, just like the parameters set by e.g. matchers or actions. Calling a resource always creates a new map of parameters.

The behaviour of resources has slightly changed due to the introduction of the TreeProcessor in Cocoon. Since 2.1 the sitemap interpreter strategy allows for the Resources to be templating any composed portion of a pipeline. (Prior to 2.1 Resources were 'ending' pipelines and as such a call of resource was said 'not to return')

The following example shows how to define and call sitemap resources:

<map:resources> <map:resource name="generate-data" > <map:generate type="my-specific-parser" src="{input-src}" /> </map:resource> <map:resource name="transform-data2svg" > <map:transform src="xsl/data2svg.xsl" /> </map:resource> <map:resource name="transform-data2html" > <map:transform src="xsl/data2html.xsl" /> </map:resource> <map:resource name="pipe-data-raw"> <map:read mime-type="text/plain" src="{input-src}" /> </map:resource> </map:resources> <map:pipeline> <map:match pattern="styled-data/*/*"> <map:call resource="generate-data"> <map:parameter name="input-src" value="{2}"/> </map:call> <map:call resource="transform-data2{1}" /> <map:serialize /> </map:match> <map:match pattern="raw-data/*" > <map:call resource="pipe-data-raw"> <map:parameter name="input-src" value="{1}"/> </map:call> </map:match> </map:pipeline>

2.15. Mounting sitemaps

Mount points allow sitemaps to be cascaded and site management workload to be parallelized. This creates a tree of sitemaps with the main sitemap at the root and possibly several sub-sitemaps as nodes and leaves.

The sub-sitemaps serve two important goals: scalability and simplification of maintenance. The different sub-sitemaps are independent and don't affect each other.

<map:match pattern="faq/*"> <map:mount uri-prefix="faq" check-reload="no" src="faq/sitemap.xmap"/> </map:match>

The src attribute is where the sub-sitemap is located. If it ends in a slash "sitemap.xmap" is appended to find the sitemap, otherwise the src value is used. A check-reload attribute can be used to determine if the modification date of the sub-sitemap file should be checked. The uri-prefix is the part that should be removed from the request URI. The engine will correctly check for a trailing slash (which you may write, of course). If in the example above "faq/cocoon" is requested, "faq/" is removed from the URI and "cocoon" is passed to the sub-sitemap which is loaded from "faq/sitemap.xmap".

Sitemap components (generators, transformers, etc.) in a sitemap are accessible from a sub-sitemap by their names. This is due to the fact that each sitemap has its own Sitemap service manager and they are arranged in the same hierarchical structure as the sitemaps are and thus knows which are their parent sitemap service manager and can ask it for a SitemapComponent it doesn't know about.

2.15.1. Use Cases

Usually you use the same SitemapComponents over and over again in your sub-sitemaps. And because you have a contract between the parent and sub sitemaps (the uri-prefix) you can deliver common SitemapComponents from the parent sitemap to your sub-sitemaps as well. If you break a sitemap all its sub-sitemaps are broken as well (because of the hierarchical arrangement).

However you can create independent sub-sitemaps, which meet the following goals:

1. Simplify site construction
2. Reduce risk of "bad' sitemap entries killing whole site
3. Ease deployment of pre-built applications
4. Keep sitemap files an understandable and manageable size

Just build a main sitemap with the minimum needs: A matcher and a selector. These two components allow to select and direct to mount two other sitemaps. These sub-sitemaps load the components they need (which includes generators and transformers etc...) and have the map elements for that site/application. The benefit is that each sitemap is completely independent of each other and any error in that sitemap does not kill any other sitemap.

Here is an example of a main sitemap. You will notice that it is using a selector that matches on host name of the request, but any matcher or selector would work at this point. Both sub-sitemaps are mounted at the root level.

```
<?xml version="1.0"?> <map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
<!-- Components ================================ --> <map:components>
<map:matchers default="wildcard"> <map:matcher name="wildcard"
src="org.apache.cocoon.matching.WildcardURIMatcher"/> </map:matchers> <map:selectors
default="host"> <map:selector name="host"
src="org.apache.cocoon.selection.HostSelector"> <host name="fee" value="www.foo.com"/>
</map:selector> </map:selectors> </map:components> <!-- Pipelines
================================ --> <map:pipelines> <map:pipeline> <map:select
type="host"> <map:when test="fee"> <map:mount uri-prefix="" src="fee.xmap"/>
</map:when> <map:otherwise> <map:mount uri-prefix="" src="foo.xmap"/>
</map:otherwise> </map:select> </map:pipeline> </map:pipelines> </map:sitemap>
```

2.15.2. Reloading

The reloading of the sub-sitemaps can be configured by two attributes, "check-reload" and "reload-method".

```
<map:match pattern="faq/*"> <map:mount uri-prefix="faq/" check-reload="no"
src="faq/sitemap.xmap" reload-method="asynchron"/> </map:match>
```

The "check-reload" attribute specifies whether the sitemap should be reloaded (regenerated) if it's source XML (sitemap.xmap) is modified. If "check-reload" is set to "no", the sitemap is only generated on the first request for this sitemap. If "check-reload" is set to "yes" (the default), the "reload-method" attribute determines how the sitemap is regenerated if it had changed.

If "reload-method" is set to "asynchron" (the default), then the next request for the changed sitemap causes it to be regenerated in the background, and the request is served with the old one. All subsequent requests are served with the old sitemap until the regeneration in the background has finished. If the reload-method is set to "synchron", the sitemap is first regenerated and then the request is processed.

3. File: URLs

In your sitemaps you may need to refer to some resource that is outside the webapp context (e.g. UNIX /foo/bar/this.xsl e.g. Windows C:\foo\bar\this.xsl). You need to use the file: convention with the following syntax for absolute filesystem pathnames.

- UNIX ... file:///foo/bar/this.xsl
- Windows ... file:///C:/foo/bar/this.xsl

Everything starting with a URI scheme identifer like "file:" or "http:" is an absolute URI. An absolute file URL is file://some.host/some/path/to/file.ext ... the host can be omitted, defaulting to localhost, so you can write file:///some/path/to/file.ext

Further information is at RFC2396: Uniform Resource Identifiers (URI): Generic Syntax

4. Protocols

In the sitemap, you can use all protocols nearly everywhere (except for in map:redirect).

Inside your components, you can also use these protocols whenever you have a SourceResolver handy.

- context:// - get a resource using the servlet context
- cocoon:/ - get a pipeline from the current sitemap
- cocoon:// - get a pipeline using the root sitemap
- resource:// - get a resource from the context classloader

5. Interface specifications

5.1. XMLProducer

This interfaces identifies classes that produce XML data, sending SAX events to the configured XMLConsumer.
It is beyond the scope of this interface to specify a way in which the XML data production is started.

```
public interface XMLProducer { /** * Set the <code>XMLConsumer</code> that will * receive XML data. */ public void setConsumer(XMLConsumer consumer); }
```

5.2. XMLConsumer

This interfaces identifies classes that consume XML data, receiving notification of SAX events.
This interface unites the idea of SAX ContentHandler and LexicalHandler.

```
public interface XMLConsumer extends ContentHandler, LexicalHandler { }
```

5.3. XMLPipe

This interfaces identifies classes that consume XML data, receiving notification of SAX events, and also that produce XML data, sending SAX events to the configured XMLConsumer.
This interface unites the idea of XMLProducer and XMLConsumer.

public interface XMLPipe extends XMLConsumer , XMLProducer { }

5.4. SitemapModelComponent

All sitemap components producing XML must implement this interface:

public interface SitemapModelComponent extends Component { /** * Set the <code>SourceResolver</code>, objectModel * <code>Map</code>, the source and sitemap * <code>Parameters</code> used to process the request. */ void setup(SourceResolver resolver, Map objectModel, String src, Parameters par) throws ProcessingException, SAXException, IOException; }

5.5. SitemapOutputComponent

All sitemap components creating the output must implement this interface:

public interface SitemapOutputComponent extends Component { /** * Set the <code>OutputStream</code> where the requested * resource should be serialized. */ void setOutputStream(OutputStream out) throws IOException; /** * Get the mime-type of the output of this * <code>Component</code>. */ String getMimeType(); /** * Test if the component wants to set the content length */ boolean shouldSetContentLength(); }

5.6. Generator

A Generator must implement at least the following interface:

public interface Generator extends XMLProducer, SitemapModelComponent { String ROLE = "org.apache.cocoon.generation.Generator"; public void generate() throws IOException, SAXException, ProcessingException; }

5.7. Transformer

A Transformer must implement at least the following interface:

public interface Transformer extends XMLPipe, SitemapModelComponent { String ROLE = "org.apache.cocoon.transformation.Transformer"; }

5.8. Serializer

A Serializer gets the OutputStream where the XML should be serialized with the following interface:

public interface Serializer extends XMLConsumer, SitemapOutputComponent { String ROLE = "org.apache.cocoon.serialization.Serializer"; }

5.9. Selector

A Selector gets an expression to evaluate and signals the evaluation with a boolean value.

public interface Selector extends Component { String ROLE = "org.apache.cocoon.selection.Selector"; /** * Selectors test pattern against some objects in a <code>Map</code> * model and signals success with the returned boolean value * @param

expression The expression to test. * @param objectModel The <code>Map</code> containing object of the * calling environment which may be used * to select values to test the expression. * @param parameters The sitemap parameters, as specified by * <parameter/> tags. * @return boolean Signals successfull test. */ boolean select (String expression, Map objectModel, Parameters parameters); }

5.10. Matcher

A Matcher matches a pattern against any value:

public interface Matcher extends Component { String ROLE = "org.apache.cocoon.matching.Matcher"; /** * Matches the pattern against some <code>Request</code> values * and returns a <code>Map</code> object with replacements * for wildcards contained in the pattern. * @param pattern The pattern to match against. Depending on * the implementation the pattern can contain * wildcards or regular expressions. * @param objectModel The <code>Map</code> with object of the * calling environment which can be used * to select values this matchers matches against. * @return Map The returned <code>Map</code> object with * replacements for wildcards/regular-expressions * contained in the pattern. * If the return value is null there was no match. */ Map match (String pattern, Map objectModel, Parameters parameters); }

5.11. Action

An Action processes input objectModel and returns results in a Map:

public interface Action extends Component, ThreadSafe { String ROLE = "org.apache.cocoon.acting.Action"; /** * Controls the processing against some values of the * <code>Dictionary</code> objectModel and returns a * <code>Map</code> object with values used in subsequent * sitemap substitution patterns. * * NOTE: It is important that <code>Action<code> classes are * written in a thread safe manner. * * @param resolver The <code>SourceResolver</code> in charge * @param objectModel The <code>Map</code> with object of the * calling environment which can be used * to select values this controller may need * (ie Request, Response). * @param source A source <code>String</code> to the Action * @param parameters The <code>Parameters</code> for this invocation * @return Map The returned <code>Map</code> object with * sitemap substitution values which can be used * in subsequent elements attributes like src= * using a xpath like expression: * src="mydir/{myval}/foo" * If the return value is null the processing * inside the <map:act> element of the sitemap * will be skipped. * @exception Exception Indicates something is totally wrong */ Map act(Redirector redirector, SourceResolver resolver, Map objectModel, String source, Parameters par) throws Exception; }

6. Additional resources

Learn more about advanced Sitemap features by downloading the free chapter, A User's Look at the Cocoon architecture, from Langham and Ziegeler's *Cocoon: Building XML Applications* available at the New Riders web site.

Check out a draft XML Schema grammar for the Cocoon sitemap, and some external documentation generated from this Schema. A poster diagram of the sitemap structure is also available.

1. Comments

add your comments