# **Forrest Sitemap Reference**

# **Table of contents**

1 Getting started	2
2 Sitemap Overview	
3 Source pipelines (**.xml)	
3.1 forrest.xmap	
3.2 Other source pipelines	4
4 Output pipelines	
4.1 PDF output	
4.2 HTML output	5
5 Intermediate pipelines	6
5.1 Page body	6
5.2 Page menu	7
5.3 Page tabs	7
6 Menu XML generation	7
7 Link rewriting	
7.1 Cocoon foundations: Input Modules	8
7.2 Implementing "site:" rewriting	9

Technically, Forrest can be thought of as a <u>Cocoon</u> distribution that has been stripped down and optimized for people with simple site publishing needs. Central to Cocoon, and hence Forrest, is the **sitemap**. The sitemap defines the site's URI space (what pages are available), and how each page is constructed. Understanding the sitemap is the key to understanding Forrest.

#### Note:

We advise you to spend time to understand the Apache Cocoon sitemap. See <u>Cocoon sitemap</u> and <u>Cocoon concepts</u> and related component documentation. The Forrest sitemap is broken into multiple files. The main one is **sitemap.xmap** which delegates to others.

This document provides an overview of the special sitemap which is used at the core of Apache Forrest.

# 1. Getting started

Forrest's sitemap comprises the \$FORREST\_HOME/main/webapp/\*.xmap files.

You can add pre-processing sitemaps to your project src/documentation directory (or wherever \${project.sitemap-dir} points to). Any match that is not handled, passes through to be handled by the default Forrest sitemaps - obviously extremely powerful. The capability is described in "Using project sitemaps".

Another way to experiment with the sitemap is to do 'forrest run' on a Forrest-using site. Changes to the core \*.xmap files will now be immediately visible at >http://localhost:8888/

# 2. Sitemap Overview

Forrest's sitemap is divided both physically and logically. The most obvious is the physical separation. There are a number of separate \*.xmap files, each defining pipelines for a functional area. Each \*.xmap file has its purpose documented in comments at the top. Here is a brief overview of the files, in order of importance.

sitemap.xmap	Primary sitemap file, which delegates responsibility for serving certain URIs to the others (technically called sub-sitemaps). More about the structure of this file later.
forrest.xmap	Sitemap defining Source pipelines, which generate the body section of Forrest pages. All pipelines here deliver XML in Forrest's intermediate "document-v13" format, regardless of originating source or format.
menu.xmap	Pipelines defining the XML that becomes the menu.
linkmap.xmap	Defines a mapping from abstract ("site:index") to physical ("index.html") links for the current page. See Menus and Linking for a conceptual overview, and the Link rewriting section for technical details.
resources.xmap	Serves "resource" files (images, CSS, Javascript).

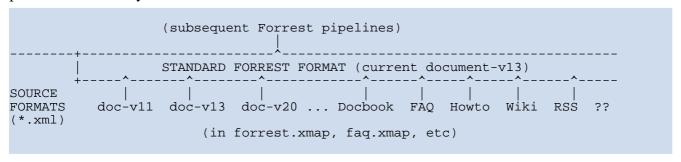
raw.xmap	Serves files located in src/documentation/content/xdocs that are not to be modified by Forrest.
aggregate.xmap	Generates a single page (HTML or PDF) containing all the content for the site.
faq.xmap	Processes FAQ documents.
status.xmap	Generates <u>changes</u> and <u>todo</u> pages from a single status.xml in the project root.
issues.xmap	Generates a page of content from an RSS feed. Used in Forrest to generate a "current issues" list from JIRA.
revisions.xmap	Support for HOWTO documents that want "revisions". Revisions are XML snippets containing comments on the main XML file. The main pipeline here automatically appends a page's revisions to the bottom.
dtd.xmap	A Source pipeline that generates XML from a DTD, using Andy Clark's <u>DTD Parser</u> . Useful for documenting DTD-based XML schemas, such as <u>Forrest's own DTDs</u> .
profiler.xmap	Defines the "profiler" pipeline. allowing pipelines to be benchmarked.

# 3. Source pipelines (\*\*.xml)

Most \*.xmap files (forrest, aggregate, faq, status, issues, revisions, dtd) define Source pipelines. Source pipelines define the content (body) XML for site pages. The input XML format can be any format (document-v13, Docbook, RSS, FAQ, Howto) and from any source (local or remote). The output format is always Forrest's intermediate "document-v13" format.

Source pipelines always have a ".xml" extension. Thus, <u>index.xml</u> gives you the XML source for the index page. Likewise, <u>faq.xml</u> gives you XML for the FAQ (transformed from FAQ syntax), and <u>changes.xml</u> returns XML from the status.xml file. Take any page, and replace its extension (.html or .pdf) with .xml and you'll have the Source XML.

This is quite powerful, because we now have an abstraction layer, or "virtual filesystem", on which the rest of Forrest's sitemap can build. Subsequent layers don't need to care whether the XML was obtained locally or remotely, or from what format. Wikis, RSS, FAQs and Docbook files are all processed identically from here on.



#### 3.1. forrest.xmap

Most of the usual Source pipelines are defined in forrest.xmap which is the default (fallback) handler for \*\*.xml pages. The forrest.xmap uses the <u>SourceTypeAction</u> to determine the type of XML it is processing, and converts it to document-v13 if necessary.

For instance, say we are rendering <u>a Howto document</u> called "howto-howto.xml". It contains this DOCTYPE declaration:

```
<!DOCTYPE howto PUBLIC "-//APACHE//DTD How-to V1.3//EN"
  "http://forrest.apache.org/dtd/howto-v13.dtd">
```

The SourceTypeAction sees this, and applies this transform to get it to document-v13:

```
<map:when test="howto-v13">
     <map:transform src="{forrest:stylesheets}/howto2document.xsl" />
     </map:when>
```

## 3.2. Other source pipelines

As mentioned above, all non-core Source pipelines are distributed in independent \*.xmap files. There is a block of sitemap.xmap which simply delegates certain requests to these subsitemaps:

```
<!-- Body content -->
<map:match pattern="**.xml">
  <map:match pattern="changes.xml">
   <map:mount uri-prefix="" src="status.xmap" check-reload="yes" />
  </map:match>
  <map:match pattern="todo.xml">
    <map:mount uri-prefix="" src="status.xmap" check-reload="yes" />
  </map:match>
  <map:match pattern="**dtdx.xml">
    <map:mount uri-prefix="" src="dtd.xmap" check-reload="yes" />
  </map:match>
 <map:match pattern="forrest-issues.xml">
    <map:mount uri-prefix="" src="issues.xmap" check-reload="yes" />
  </map:match>
  <map:match pattern="**faq.xml">
    <map:mount uri-prefix="" src="faq.xmap" check-reload="yes" />
  </map:match>
  <map:match pattern="site.xml">
    <map:mount uri-prefix="" src="aggregate.xmap" check-reload="yes" />
  </map:match>
  . . . .
  . . . .
```

#### 3.2.1. Late-binding pipelines

One point of interest here is that the sub-sitemap is often not specific about which URLs it handles, and relies on the caller (the section listed above) to only pass relevant requests to it. We term this "binding a URL" to a pipeline.

For instance, the main pipeline in faq.xmap matches \*\*.xml, but only \*\*faq.xml requests are sent to it.

This "late binding" is useful, because the whole URL space is managed in sitemap.xmap and not spread over lots of \*.xmap files. For instance, say you wish all \*.xml inside a "faq/" directory to be processed as FAQs. Just override sitemap.xmap and redefine the relevant source matcher:

```
<map:match pattern="**faq.xml">
     <map:mount uri-prefix="" src="faq.xmap" check-reload="yes" />
</map:match>
```

#### 4. Output pipelines

To recap, we now have a \*.xml pipeline defined for each page in the site, emitting standardized XML. These pipeline definitions are located in various \*.xmap files, notably forrest.xmap

We now wish to render the XML from these pipelines to output formats like HTML and PDF.

## 4.1. PDF output

Easiest case first; PDFs don't require menus or headers, so we can simply transform our intermediate format into XSL:FO, and from there to PDF. This is done by the following matcher in sitemap.xmap...

- 1. The first line uses a matching regexp to break the URL into directory (.\*?) and filename ([^/]\*) parts.
- 2. We then generate XML from a Source pipeline, with the URL cocoon: /{1}{2}.xml
- 3. We then expand any XInclude statements...
- 4. and rewrite links...
- 5. and finally apply the document2fo.xsl stylesheet, to generate XSL:FO XML.

Lastly, we generate a PDF using the fo2pdf serializer.

## 4.2. HTML output

Generating HTML pages is more complicated, because we have to merge the page body with a menu and tabs, and then add a header and footer. Here is the \*.html matcher in sitemap.xmap...

So <u>index.html</u> is formed from aggregating <u>body-index.html</u> and <u>menu-index.html</u> and <u>tab-index.html</u> and then applying the site2xhtml.xsl stylesheet to the result.

There is a nearly identical matcher for HTML files in subdirectories:

```
<map:match pattern="**/*.html">
    <map:aggregate element="site">
    <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="site2xhtml"/>
                <map:parameter name="path"
                 value="{0}"/>
                 </map:call>
                 </map:match>
```

# 5. Intermediate pipelines

#### 5.1. Page body

Here is the matcher which generates the page body:

```
1
    <map:match pattern="**body-*.html">
      <map:generate src="cocoon:/{1}{2}.xml"/>
2
3
      <map:transform type="idgen"/>
4
      <map:transform type="xinclude"/>
5
      <map:transform type="linkrewriter" src="cocoon:/{1}linkmap-{2}.html"/>
6
      <map:call resource="skinit">
7
        <map:parameter name="type" value="document2html"/>
8
        <map:parameter name="path" value="{1}{2}.html"/>
        <map:parameter name="notoc" value="false"/>
      </map:call>
10
    </map:match>
```

- 1. In our matcher pattern, {1} will be the directory (if any) and {2} will be the filename.
- 2. First, we obtain XML content from a source pipeline
- 3. We then apply a custom-written IdGeneratorTransformer, which ensures that every <section> has an "id" attribute if one is not supplied, by generating one from the <title> if necessary. For example, <idgen> will transform:

```
<section>
<title>How to boil eggs</title>
...
```

into:

```
<section id="How+to+boil+eggs">
<title>How to boil eggs</title>
...
```

Later, the document2html.xsl stylesheet will create an <a name> element for every section, allowing this section to be referred to as index.html#How+to+boil+eggs.

- 4. We then expand XInclude elements.
- 5. and rewrite links...
- 6. and then finally apply the stylesheet that generates a fragment of HTML (minus the outer elements like <html> and <body>) suitable for merging with the menu and tabs.

#### 5.2. Page menu

In the sitemap.xmap file, the matcher generating HTML for the menu is:

```
<map:match pattern="**menu-*.html">
    <map:generate src="cocoon:/{1}book-{2}.html"/>
    <map:transform type="linkrewriter" src="cocoon:/{1}linkmap-{2}.html"/>
    <map:call resource="skinit">
        <map:parameter name="type" value="book2menu"/>
        <map:parameter name="path" value="{1}{2}.html"/>
        </map:call>
    </map:match>
```

We get XML from a "book" pipeline, <u>rewrite links</u>, and apply the book2menu.xsl stylesheet to generate HTML.

How the menu XML is actually generated (the \*book-\*.html pipeline) is sufficiently complex to require a section of its own.

#### 5.3. Page tabs

Tab generation is quite tame compared to menus:

```
<map:match pattern="**tab-*.html">
    <map:generate src="content/xdocs/tabs.xml" />
    <map:transform type="linkrewriter" src="cocoon:/{1}linkmap-{2}.html"/>
    <map:call resource="skinit">
        <map:parameter name="type" value="tab2menu"/>
        <map:parameter name="path" value="{1}{2}.html"/>
        </map:call>
    </map:match>
```

All the smarts are in the tab2menu.xsl stylesheet, which needs to choose the correct tab based on the current path. Currently, a "longest matching path" algorithm is implemented. See the tab2menu.xsl stylesheet for details.

# 6. Menu XML generation

The "book" pipeline is defined in sitemap. xmapas:

```
<map:match pattern="**book-*.html">
    <map:mount uri-prefix="" src="menu.xmap" check-reload="yes" />
</map:match>
```

Meaning that it is defined in the menu. xmap file. In there we find the real definition, which is quite complicated, because there are three supported menu systems (see <u>menus and linking</u>). We will not go through the sitemap itself (menu.xmap), but will instead describe the logical steps involved:

- 1. Take site.xml and expand hrefs so that they are all root-relative.
- 2. Depending on the forrest.menu-scheme property, we now apply one of the two algorithms

for choosing a set of menu links (described in menu generation):

• For "@tab" menu generation, we first ensure each site.xml node has a tab attribute (inherited from a parent if necessary), and then pass through nodes whose tab attribute matches that of the "current" node.

For example, say our current page's path is community/howto/index.html. In site.xml we look for the node with this "href" and discover its "tab" attribute value is "howtos". We then prune the site.xml-derived content to contain only nodes with tab="howtos".

All this is done with XSLT, so the sitemap snippet does not reveal this complexity:

• For "directory" menu generation, we simply use an XPathTransformer to include only pages in the current page's directory, or below:

```
<map:transform type="xpath">
  <map:parameter name="include" value="//*[@href='{1}']" />
</map:transform>
```

Here, the "{1}" is the directory part of the current page. So if our current page is community/howto/index.html then "{1}" will be community/howto/ and the transformer will include all nodes in that directory.

We now have a site.xml subset relevant to our current page.

- 3. The "href" nodes in this are then made relative to the current page.
- 4. The XML is then transformed into a legacy "book.xml" format, for compatibility with existing stylesheets, and this XML format is returned (hence the name of the matcher: \*\*book-\*.html).

#### 7. Link rewriting

In numerous places in sitemap.xmap you will see the "linkrewriter" transformer in action. For example:

```
<map:transform type="linkrewriter" src="cocoon:/{1}linkmap-{2}.html"/>
```

This statement is Cocoon's linking system in action. A full description is provided in <u>Menus and Linking</u>. Here we describe the implementation of linking.

#### 7.1. Cocoon foundations: Input Modules

The implementation of site: linking is heavily based on Cocoon <u>Input Modules</u>, a little-known but quite powerful aspect of Cocoon. Input Modules are generic Components which simply allow you to look up a value with a key. The value is generally dynamically generated, or obtained by querying an underlying data source.

In particular, Cocoon contains an XMLFileModule, which lets one look up the value of an XML node, by interpreting the key as an XPath expression. Cocoon also has a SimpleMappingMetaModule, which allows the key to be rewritten before it is used to look up a value.

The idea for putting these together to rewrite "site:" links was described in this thread. The idea is to write a Cocoon Transformer that triggers on encountering <link href="scheme:address">, and interprets the scheme:address internal URI as inputmodule:key. The transformer then uses the named InputModule to look up the key value. The scheme:address URI is then rewritten with the found value. This transformer was implemented as <a href="LinkRewriterTransformer">LinkRewriterTransformer</a>, currently distributed as a "block" in Cocoon 2.1

#### 7.2. Implementing "site:" rewriting

Using the above components, "site:" URI rewriting is accomplished as follows.

#### 7.2.1. cocoon.xconf

First, we declare all the input modules we will be needing:

```
<!-- For the site: scheme -->
<component-instance
   class="org.apache.cocoon.components.modules.input.XMLFileModule"
   logger="core.modules.xml" name="linkmap"/>
<!-- Links to URIs within the site -->
<component-instance
   class="org.apache.cocoon.components.modules.input.SimpleMappingMetaModule"
   logger="core.modules.mapper" name="site"/>
<!-- Links to external URIs, as distinct from 'site' URIs -->
<component-instance
   class="org.apache.cocoon.components.modules.input.SimpleMappingMetaModule"
   logger="core.modules.mapper" name="ext"/>
```

- **linkmap** will provide access to the contents of site.xml; for example, linkmap:/site/about/index/@href would return the value "index.html".
- **site** provides a "mask" over **linkmap** such that site:index expands to linkmap:/site//index/@href
- **ext** provides another "mask" over **linkmap**, such that ext:ant would expand to linkmap:/site/external-refs//ant/@href

However at the moment, we have only declared the input modules. They will be configured in sitemap.xmap as described in the next section.

#### 7.2.2. sitemap.xmap

Now in the sitemap, we define the LinkRewriterTransformer, and insert it into any pipelines which deal with user-editable XML content:

```
</input-module>
    <prefix>/site//</prefix>
    <suffix>/@href</suffix>
  </input-module>
  <input-module name="ext">
    <input-module name="linkmap">
      <file src="{src}" reloadable="false" />
    </input-module>
    <prefix>/site/external-refs//</prefix>
    <suffix>/@href</suffix>
  </input-module>
</map:transformer>
<map:match pattern="**body-*.html">
  <map:generate src="cocoon:/{1}{2}.xml"/>
  <map:transform type="idgen"/>
 <map:transform type="xinclude"/>
  <map:transform type="linkrewriter" src="cocoon:/{1}linkmap-{2}.html"/>
</map:match>
```

As you can see, our three input modules are configured as part of the LinkRewriterTransformer's configuration.

• Most deeply nested, we have:

The "{src}" text is expanded to the value of the "src" attribute in the "linkrewriter" instance, namely "cocoon: /{1}linkmap-{2}.html" Thus the linkmap module reads dynamically generated XML specific to the current request.

- One level out, we configure the "site" and "ext" input modules, to map onto our dynamically configured "linkmap" module.
- Then at the outermost level, we configure the "linkrewriter" transformer. First we tell it which attributes to consider rewriting:

```
<link-attrs>href src</link-attrs>
<schemes>site ext</schemes>
```

So, "href" and "src" attributes starting with "site:" or "ext:" are rewritten.

By nesting the "site" and "ext" input modules in the "linkrewriter" configuration, we tell "linkrewriter" to use these two input modules when rewriting links.

The end result is that, for example, the source XML for the community/body-index.html page has its links rewritten by an XMLFileModule reading XML from cocoon:/community/linkmap-index.html

#### 7.2.3. Dynamically generating a linkmap

Why do we need this "linkmap" pipeline generating dynamic XML from site.xml, instead of just using site.xml directly? The reasons are described in the linkmap RT: we need to concatenate @hrefs and add dot-dots to the paths, depending on which directory the linkee is in. This is done with the following pipelines in linkmap.xmap...

```
<!-- site.xml with @href's appended to be context-relative. --> <map:match pattern="abs-linkmap">
```

You can try these URIs out directly on a live Forrest to see what is going on (for example, Forrest's own <u>abs-linkmap</u>).