

Introducing Cocoon

```
a.menu { color: #FFFFFF; text-align:left; font-size:12px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:plain; text-decoration:none; padding-left: 14px } A.menu:hover { color: #FFCC00 } .menutitle
{ color: #000000; text-align:left; font-size:10px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:bold; padding-left: 8px } .menuselected { color: #FFCC00; text-align:left; font-size:12px;
font-family: Verdana, Arial, Helvetica, sans-serif; font-weight:bold; padding-left: 14px }
```



Introducing Cocoon

XML.APACHE.ORG WWW.APACHE.ORG WWW.W3.ORG w3c

[Index](#)
[News](#)
[License](#)
[Download](#)

[Documentation](#)
[Introduction](#)
[Installing](#)
[Overview](#)
[User Guide](#)
[Dev Guide](#)
[Tutorials](#)
[FAQs](#)
[How-Tos](#)
[Performance](#)
[Snippets](#)
[ToC](#)

[Status](#)
[Changes](#)
[Todo](#)
[Planning notes](#)
[Patch queue](#)

[Community](#)
[Cocoon Bylaws](#)
[Hall of Fame](#)
[Contributing](#)
[Mail Lists](#)
[Mail Archives](#)

[Project](#)
[Bug Database](#)
[Code Repository](#)
[Dev Snapshots](#)

[Links](#)
[Cocoon Links](#)
[Live Sites](#)

The XML Hype

Everybody talks about XML. XML here, XML there. All application servers support XML, everybody wants to do B2B using XML, web services using XML, even databases using XML.

Should you care about it? Given the amount of hype, you can't afford to go around ignoring the argument, would be like ignoring the world wide web 10 years ago: a clear mistake. But why is this so for XML? What is this "magic" that XML seems to have to solve my problems? Isn't this another hype to change once again the IT infrastructure that you spent so much time implementing and fixing in the last few years? Isn't another way to spill money out of your pockets?

If you ever asked yourself one of the above questions, this paper is for you. You won't find singing-and-dancing marketing crap, you won't find boring and useless feature lists, you won't find the usual acronym bombing or those good looking vaporware schemas that connect your databases to your coffee machines via CORBA or stuff like that.

This document will explain you what the Cocoon project is about and what we are doing to solve the problems that we encountered in our web engineering experiences, but from an executive perspective, yes, because we all had the problems of managing a web site, dealing with our colleagues, rushing to the graphical guru to have the little GIF with the new title, or calling the web administrator at night because the database is returning errors without reasons.

It was frustrating to see the best and most clever information technology ever invented (the web) ruined by the lack of engineering practices, tortured by those "let's-reinvent-the-wheel-once-again" craftsmen that were great at doing their jobs as individuals but that couldn't scale and imposed a growth saturation to the whole project.

There had to be a better way of doing things.

Personal Experiences

In 1998, Stefano Mazzocchi volunteered to create the documentation infrastructure for the `java.apache.org` project, which is composed by a bunch of different codebases, maintained by a bunch of different people, with different skills, different geographical locations and different degree of will and time to dedicate to the documentation effort.

But pretty soon he realized that no matter how great and well designed the system was, HTML was a problem: it was **not** designed for those kind of things. Looking at the main page (<http://java.apache.org/>) from the browser and you could clearly identify the areas of the screen: sidebar, topbar, news, status. But if you opened the HTML, boom: a nightmare of table tags and nesting and small little tricks to make the HTML appear the same on every browser.

So he looked around for alternative technologies, but **all** of them were trying to add more complexity at the GUI level (Microsoft Frontpage, Macromedia Dreamweaver, Adobe GoLive, etc...) hoping to "hide" the design problems of HTML under a thick layer of WYSIWYG looks.

What you see is what you get.

But what you see is all you've got.

How can you tell your web server to "extract" the information from the sidebar? How can you have the news feeds out of a complex HTML page?

Damn, it's easy for a human reader: just look at the page and it's very easy to distinguish between a sidebar, a banner, a news and a stock quote. Why is it so hard for a machine?

The HTML Model

HTML is a language that tells your browser how to "draw" things on its window. An image here, a letter there, a color down here. Nothing more. The browser doesn't have the "higher level" notion of "sidebar": it lacks the ability to perform "semantic analysis" on the HTML content.

Semantic analysis? Yeah, it's the kind of thing the human brain is simply great at doing, while computer programs simply suck big time.

So, with HTML, we went a step up and created a highly visual and appealing web of HTML content, but we went two steps back by removing all the higher level semantic information from the content itself.

Introducing Cocoon

Ok, let's make an example... most of you have seen an HTML page... if not, here is an example:

```
<html>
<body>
  <p>Hi, I'm an HTML page</p>
  <p align="center">Written by Stefano</p>
</body>
</html>
```

which says to the browser:

- ◆ I'm a HTML page
- ◆ I have a body
- ◆ I have a paragraph
- ◆ I contain the sentence "Hi, I'm an HTML page."
- ◆ I contain the sentence "Written by Stefano"

Suppose you are a chinese guy that doesn't understand our alphabet, try to answer the following question:

who wrote the page?

You can't perform semantic analysis, you are as blind as a web browser. The only thing you can do is draw it on the screen since this is what you were programmed to do. In other words, your semantic capacity is fixed to the drawing capabilities and a few other things (like linking), thus limited.

Semantic Markup

Suppose you receive this page:

```
<page>
  <author>sflkjoier</author>
  <content>
    <para>sofikdjflksj</para>
  </content>
</page>
```

can you tell me who wrote the page? easy, you say, "sflkjoier" did. Good, but later you receive:

```
<dlkj>
  <ruijfl>sofikdjflksj</ruijfl>
  <wijklkf>
    <oamkfkj>sflkjoier</oamkfkj>
  </wijklkf>
</dlkj>
```

now, who wrote the page? You could guess by comparing the structure, but how do you know the two structures reflect the same semantic information?

The above two pages are both XML documents.

Introducing Cocoon

Are they going to help you? Are they doing to simplify your work?
Are they going to simplify your problems?

At this point, clearly not so, rather the opposite.

So, you could be wondering, why did we spend so much effort to write an XML publishing framework? This document was written exactly to clear your doubts on this, so let's keep going.

The XML Language

XML is most of the times referred to as the "eXtensible Markup Language" specification. A fairly small yet complex specification that indicates how to write languages. It's a syntax. To tell you the truth, nothing fancy at all. So

```
<hello></hello>
```

is correct, while

```
<hello></hi>
```

is not, but

```
<hello><hi /></hello>
```

is correct. That's more than this, but I'll skip the technical details here.

XML is the ASCII for the new millenium, it's a step forward from ASCII or UNICODE (the international extension to ASCII that includes all characters from all modern languages). It defines a "lingua franca" for textual languages.

Ok, great, so now instead of having one uniform language with visual semantics (HTML) we have a babel of languages each with its own semantics. How this can possibly help you?

XML Transformations

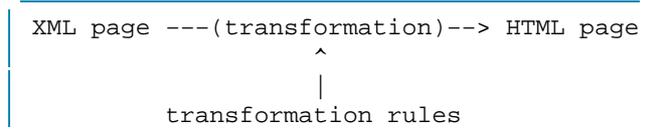
This was the point where Stefano was more or less two years ago for java.apache.org: I could use XML and define my own semantics with <sidebar>, <news>, <status> and all that and I'm sure people would have found those XML documents much easier to write (since the XML syntax is very similar to the HTML one and very user friendly)... but I would have moved from "all browsers" to "no browser".

And having a documentation that nobody can browse is totally useless.

The turning point was the creation of the XSL specification which included a way to "transform" an XML page into something else. (it's more complex than this, but, again, I'll skip the technical details).

Introducing Cocoon

So now you have:



that allows you to write your pages in XML, create your "graphics" as transformation rules and generate HTML pages on the fly directly from your web server.

Apache Cocoon 1.0 did exactly this.

The Model Evolves

If XML is a lingua franca, it means that XML software can work on almost anything without caring about what it is. So, if a cell phone requests the page, Cocoon just has to change transformation rules and send the WAP page to the phone. Or, if you want a nice PDF to printout your monthly report, you change the transformation rules and Cocoon creates the PDF for you, or the VRML, or the VoiceML, or your own proprietary B2B markup.

Anything without changing the basic architecture that is simply based on the simple "angle bracket" XML syntax.

Separation of Concerns (SoC)

Cocoon was not the first product to perform server side XML transformations, nor will be the last one (in a few years, these solutions will be the rule rather than the exception). So, what is the "plus" that the Cocoon project adds?

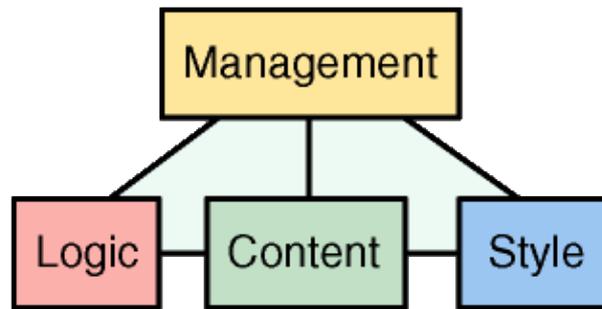
We believe the single most important Cocoon innovation is SoC-based design.

SoC is something that you've always been aware of: not everybody is equal, not everybody performs the same job with the same ability.

It can be observed that separating people with common skills in different working groups increases productivity and reduces management costs, but only if the groups do not overlap and have clear "contracts" that define their operability and their concerns.

For a web publishing system, the Cocoon project uses what we call the *pyramid of contacts* which outlines four major concern areas and five contracts between them. Here is the picture:

Introducing Cocoon



Cocoon is *engineered* to provide you a way to isolate these four concern areas using just those 5 contracts, removing the contract between style and logic that has been bugging web site development since the beginning of the web.

Why? because programmers and graphic people have very different skills and work habits... so, instead of creating GUIs to hide the things that can be harmful (like graphic to programmers or logic to designers), Cocoon allows you to separate the things into different files, allowing you to "seal" your working groups into separate virtual rooms connected with the other rooms only by those "pipes" (the contracts), that you give them from the management area.

Let's have an example:

```
<page>
  <content>
    <para>Today is <dynamic:today/></para>
  </content>
</page>
```

is written by the content writers and you give them the "contract" that states that the tag `<dynamic:today/>` prints out the time of the day when included in the page. Content writers don't care (nor should) about what language has been used for that, nor they can mess up with the programming logic that generates the content since it's stored in another part of the system they don't have access to.

So `<dynamic:today/>` is the "logic – content" contract.

At the same time, the structure of the page is given as a contract to the graphic designers who have to come up with the transformation rules that transform this structure in a language that the browser can understand (HTML, for example).

So, the page structure is the "content – style" contract.

As long as these contract don't change, the three areas can work in a completely parallel way without saturating the human resources used to manage them: costs decrease because time to market is reduced and maintenance costs is decreased because errors do not propagate out of the concern areas.

Introducing Cocoon

For example, you can tell your designers to come up with a "Xmas look" for your web site, without even telling the other people: just switch the XMas transformation rules at XMas morning and you're done.... just imagine how painful it would be to do this on your web site today.

With the Cocoon architecture all this is a couple of line changes away.

Here we go

If you reached this far by reading all sections, you should have grasped the value of the Cocoon Project and distinguish most of the marketing hype that surrounds XML and friends.

Just like you shouldn't care if somebody offers you a software that is "ASCII compliant" or "ASCII based", you shouldn't care about "XML compliant" or "XML based": it doesn't mean anything.

Cocoon uses XML as a core piece of its framework, but improves the model to give you the tools you need and is designed to be flexible enough to follow your needs as well as paradigm shifts that will happen in the future.

Copyright © 1999–2002 The Apache Software Foundation. All Rights Reserved.

Introducing Cocoon

```
a.menu { color: #FFFFFF; text-align:left; font-size:12px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:plain; text-decoration:none; padding-left: 14px } A.menu:hover { color: #FFCC00 } .menutitle
{ color: #000000; text-align:left; font-size:10px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:bold; padding-left: 8px } .menuselected { color: #FFCC00; text-align:left; font-size:12px;
font-family: Verdana, Arial, Helvetica, sans-serif; font-weight:bold; padding-left: 14px }
```



Overview of Apache Cocoon

XML.APACHE.ORG WWW.APACHE.ORG WWW.W3.ORG w3c

- [Index](#)
- [News](#)
- [License](#)
- [Download](#)
- Documentation
 - [Introduction](#)
 - [Installing](#)
 - [Overview](#)
 - [User Guide](#)
 - [Dev Guide](#)
 - [Tutorials](#)
 - [FAQs](#)
 - [How-Tos](#)
 - [Performance](#)
 - [Snippets](#)
 - [ToC](#)
- Status
 - [Changes](#)
 - [Todo](#)
 - [Planning notes](#)
 - [Patch queue](#)
- Community
 - [Cocoon Bylaws](#)
 - [Hall of Fame](#)
 - [Contributing](#)
 - [Mail Lists](#)
 - [Mail Archives](#)
- Project
 - [Bug Database](#)
 - [Code Repository](#)
 - [Dev Snapshots](#)
- Links
 - [Cocoon Links](#)
 - [Live Sites](#)

What is Apache Cocoon

Cocoon is an XML publishing framework. It allows you to define XML documents and transformations to be applied on it, to eventually generate a presentation format of your choice (HTML, PDF, SVG, ...).

Cocoon also gives you the possibility to apply logic to your XML files (so that the XML pipeline can be dynamic).

The [User documentation](#) and especially [Concepts](#) will help to understand Cocoon.

Examples and demonstration applications

There are a whole suite of sample applications to demonstrate the power of Cocoon. These samples are available from the "welcome" page after you have downloaded, built, and installed the distribution. Each example portrays a different aspect of the vast capabilities of Cocoon ... <http://localhost:8080/cocoon/>

 *With the 2.1 version, <http://localhost:8080/cocoon/> goes directly to the documentation, while <http://localhost:8080/cocoon/samples/> is the Samples.*

It will greatly assist your understanding of Cocoon to investigate behind-the-scenes, to find out how each sample is processed. Do this by looking at the actual XML documents provided in the distribution at <src/webapp/samples/> and by consulting each sitemap to see the processing steps that are defined.

Overview of XML document processing

This section gives a general overview of how an XML document is handled by Cocoon. See also the document [Understanding Cocoon](#) for explanation of the separation of content, style, logic and management functions.

Pipeline

[Cocoon Hosting](#)

[Cocoon 1.x](#)

[Old Generation](#)

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. This can be compared to the "servlet-chaining" concept of a servlet engine. We'll explain the components of the pipeline now in more detail.

Generator

The Generator is the starting point for the pipeline. It is responsible for delivering SAX events down the pipeline.

The simplest Generator is the FileGenerator: it takes a local XML document, parses it, and sends the SAX events down the pipeline.

The Generator is constructed to be independent of the concept "file". If you are able to generate SAX events from another source, you can use that without having to go via a temporary file.

Transformer

A Transformer can be compared to an XSL: it gets an XML document (or SAX events), and generates another XML document (or SAX events).

The simplest Transformer is the XalanTransformer: it applies an XSL to the SAX events it receives.

Serializer

A Serializer is responsible for transforming SAX events to a presentation format. For actors looking at the back of the pipeline, it looks like a static file is delivered. So a browser can receive HTML, and will not be able to tell the difference with a static file on the filesystem of the server.

We have Serializers for generating HTML, XML, PDF, VRML, WAP, and of course you can create your own.

The simplest Serializer is the XMLSerializer: it receives the SAX events from up the pipeline, and returns a "human-readable" XML file.

```
a.menu { color: #FFFFFF; text-align:left; font-size:12px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:plain; text-decoration:none; padding-left: 14px } A.menu:hover { color: #FFCC00 } .menutitle
{ color: #000000; text-align:left; font-size:10px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:bold; padding-left: 8px } .menuselected { color: #FFCC00; text-align:left; font-size:12px;
font-family: Verdana, Arial, Helvetica, sans-serif; font-weight:bold; padding-left: 14px }
```



Understanding Apache Cocoon

XML.APACHE.ORG WWW.APACHE.ORG WWW.W3.ORG w3c



Overview

This document is intended for both Users and Developers and presents an overall picture of Apache Cocoon.

- ◆ [Prerequisites](#)
- ◆ [A Little History](#)
- ◆ [What problem does Cocoon solve?](#)
- ◆ [Basic Mechanisms.](#)
- ◆ [Architecture.](#)
- ◆ [Abstraction.](#)
- ◆ [Apache Cocoon Configuration.](#)
- ◆ [Apache Cocoon Work Area.](#)
- ◆ [Use with Tomcat](#)

Prerequisites

What You Should know:

- ◆ XML, XML Namespaces
- ◆ Basics of XPath, XSLT
- ◆ Java language
- ◆ Servlets, HTTP

What You need not know:

- ◆ Cocoon 1

A Little History

Cocoon 1

- ◆ Cocoon project was founded in Jan. 1999 by Stefano Mazzocchi as an open source project under Apache Software Foundation.
- ◆ Started as a simple servlet for XSL styling of XML content.
- ◆ Was based on DOM level 1 API. This choice turned out to be quite limiting for speed/memory efficiency.

- ◆ Used reactor pattern to connect components. This allowed the reaction instructions to be placed inside the documents. Though appealing, it caused difficulties in managing highly dynamic web-sites.
- ◆ Allowed context overlap to happen by having processing instructions in documents/stylesheets.

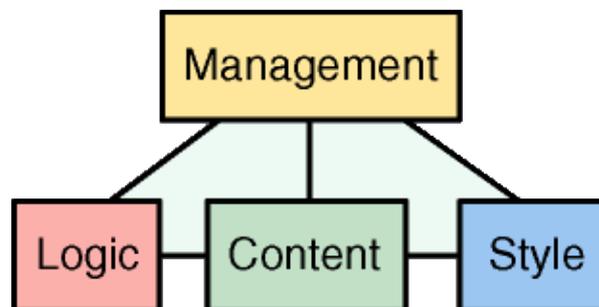
Apache Cocoon

- ◆ A separate codebase to incorporate Cocoon 1 learnings.
- ◆ Designed for execution speed/memory efficiency and scalability to process very large documents by switching processing model from DOM to SAX.
- ◆ Centralizes the management functions by allowing processing pipeline specification in a sitemap (an XML file), replacing the embedded processing instruction model.
- ◆ Better support for pre-compilation, pre-generation and caching for better performance.

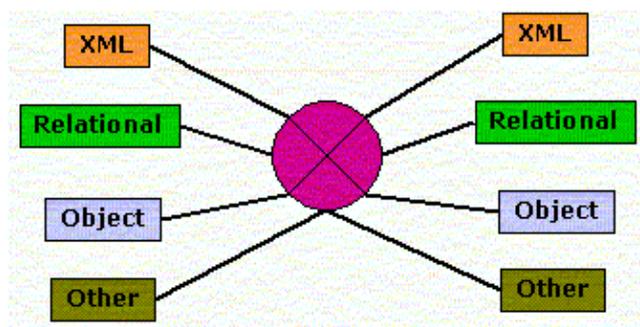
What problem does Cocoon solve?

Basic problem to be solved:

Separation of content, style, logic and management functions in an XML content based web site (and web services).



Data Mapping

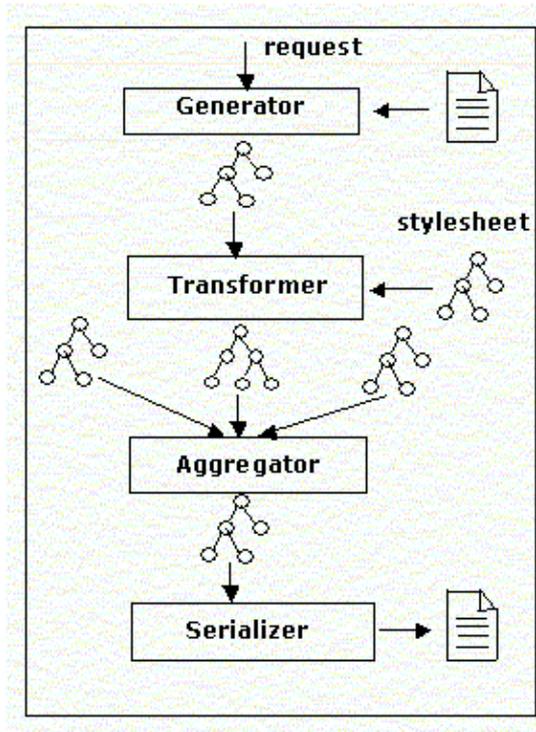


Basic Mechanisms.

Basic mechanisms for processing XML documents:

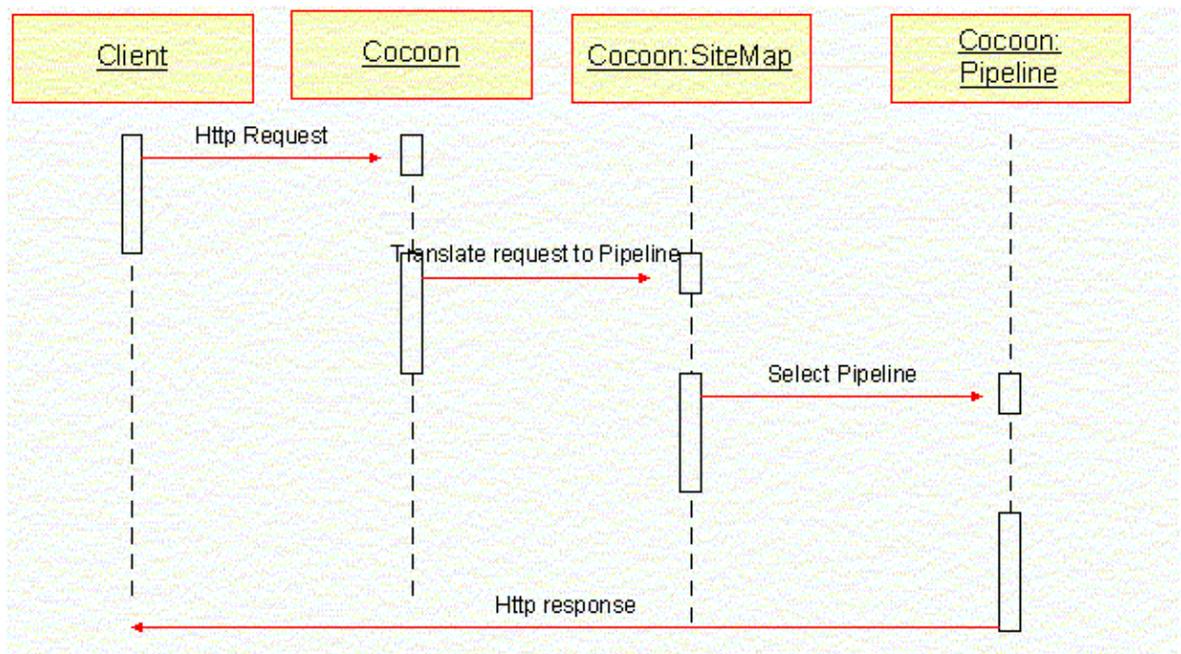
Introducing Cocoon

- ◆ Dispatching based on Matchers.
- ◆ Generation of XML documents (from content, logic, Relation DB, objects or any combination) through Generators
- ◆ Transformation (to another XML, objects or any combination) of XML documents through Transformers
- ◆ Aggregation of XML documents through Aggregators
- ◆ Rendering XML through Serializers

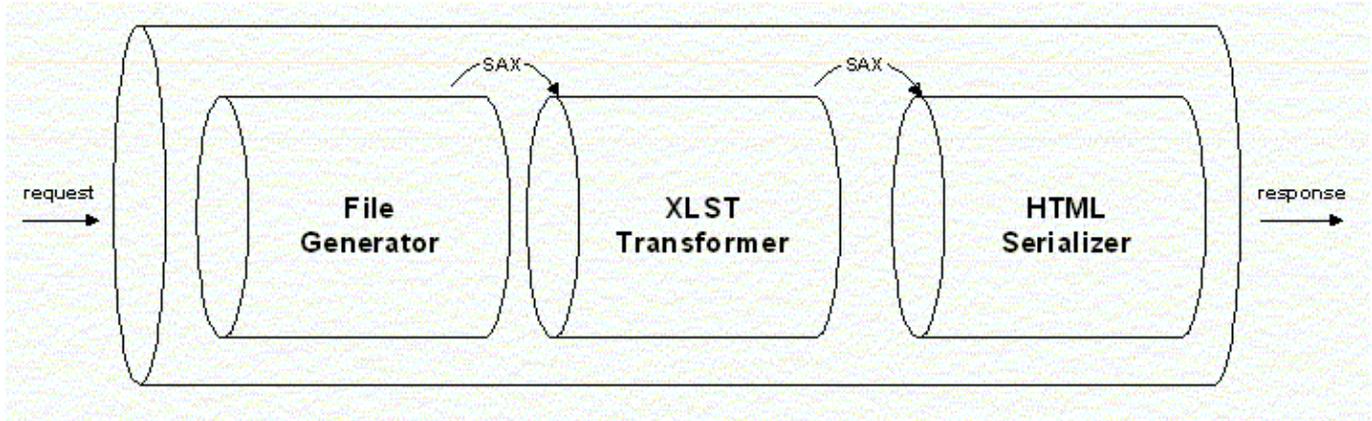


Pipeline Processing

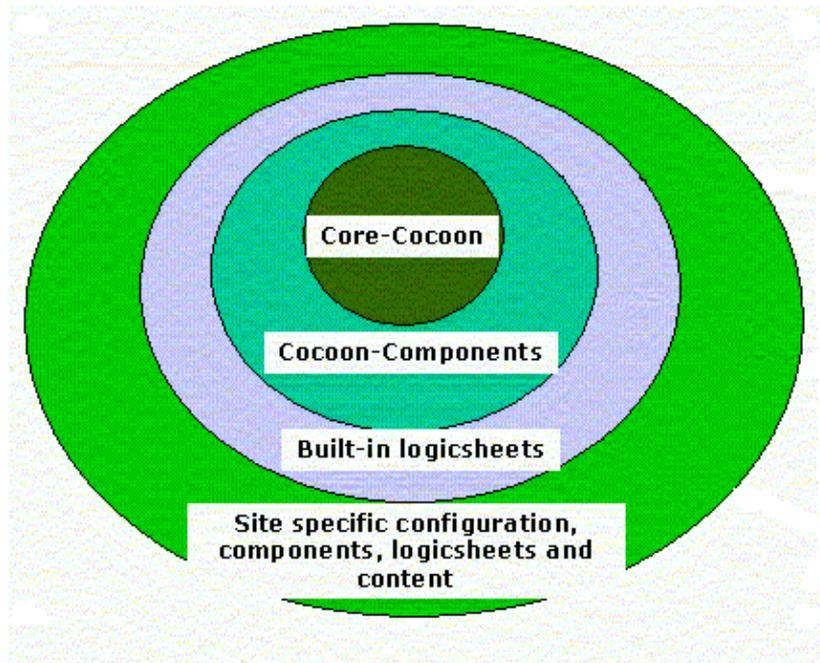
Sequence of Interactions



Pipeline



Architecture.



Core Cocoon

- ◆ Avalon framework for logging, configuration, threading, context etc.
- ◆ Caching mechanism
- ◆ Pipeline handling
- ◆ Program generation, compilation, loading and execution.
- ◆ Base classes for generation, transformation, serialization, components.
- ◆ ...

Cocoon Components

- ◆ Specific generators
- ◆ Specific transformers
- ◆ Specific matchers
- ◆ Specific serializers

◆ ...

Built-in Logicsheets

- ◆ sitemap.xml
- ◆ xsp.xml
- ◆ esql.xml
- ◆ request.xml
- ◆ response.xml
- ◆ ...

Site specific configuration, components, logicsheets and content

◆ ...

Abstraction.

eXtensible Server Pages (XSPs)

An XSP page is an XML page with following requirements:

- ◆ The document root must be `<xsp:page>`
- ◆ It must have language declaration as an attribute in the `<xsp:page>` element.
- ◆ It must have namespace declaration for xsp as an attribute in the `<xsp:page>` element.
- ◆ For an XSP to be useful, it must also require at least an `<xsp:logic>` and an `<xsp:expr>` element.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsp:page language="java" xmlns:xsp="http://apache.org/xsp">
  <xsp:logic>
    static private int counter = 0;
    private synchronized int count()
    {
      return counter++;
    }
  </xsp:logic>
  <page>
    <p>I have been requested <xsp:expr>count()</xsp:expr> times.</p>
  </page>
</xsp:page>
```

An XSP page is used by a generator to generate XML document.

XSP Processing (Code Generation)

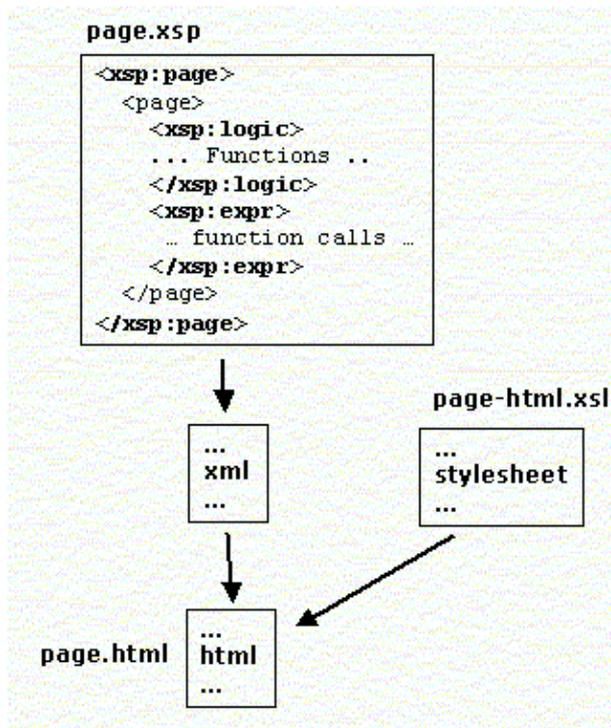
```
package org.apache.cocoon.www.docs.samples.xsp;
import java.io.File;
// A bunch of other imports
```

Introducing Cocoon

```
public class counter_xsp extends XSPGenerator {
    // .. Bookkeeping stuff commented out.
    /* User Class Declarations */
    static private int counter = 0;
    private synchronized int count() {
        return counter++;
    }
    /* Generate XML data. */
    public void generate() throws SAXException {
        this.contentHandler.startDocument();
        AttributesImpl xspAttr = new AttributesImpl();
        this.contentHandler.startPrefixMapping("xsp", "http://apache.org/xsp");
        this.contentHandler.startElement("", "page", "page", xspAttr);
        // Statements to build the XML document (Omitted)
        this.contentHandler.endElement("", "page", "page");
        this.contentHandler.endPrefixMapping("xsp");
        this.contentHandler.endDocument();
    }
}
```

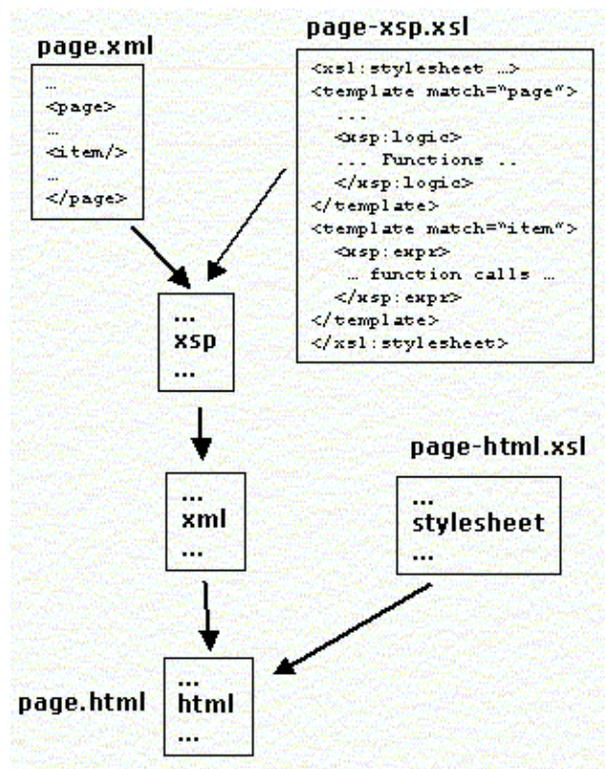
Ways of Creating XSPs Embedded Logic

- ◆ Code is embedded in the XML page
- ◆ No separation of content and logic
- ◆ Okay for small examples but terrible for large systems.



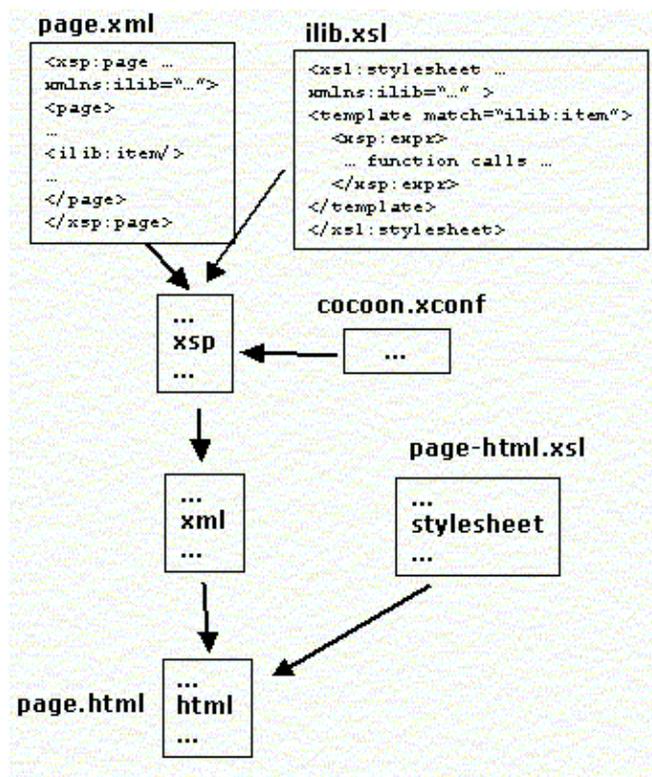
Included Logicsheet

- ◆ Code is in a separate logicsheet (an XSL file)
- ◆ Effective separation of content and logic
- ◆ Preferred way to create XSPs



Logicsheet as tag library

- ◆ The logicsheet is packaged as a reusable tag library and registered with Cocoon in `cocoon.xconf` file.
- ◆ Tag library has a namespace declaration, declared in the original logicsheet and matched in `<xsp:page xmlns:ilib="...">` attribute.
- ◆ Effective separation of content, logic and management



Sitemap

```
<?xml version="1.0"?>
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

<map:components>
...
</map:components>

<map:views>
...
</map:views>
<map:pipelines>
<map:pipeline>
<map:match>
...
</map:match>
...
</map:pipeline>
...
</map:pipelines>
...
</map:sitemap>
```

Sitemap contains configuration information for a Cocoon engine:

- ◆ list of matchers
- ◆ list of generators
- ◆ list of transformers
- ◆ list of readers
- ◆ list of serializers
- ◆ list of selectors
- ◆ list of processing pipelines with match patterns
- ◆ ...

Sitemap is an XML file corresponding to a sitemap DTD.

Sitemap can be edited to add new elements.

Sitemap is generated into a program and is compiled into an executable unit.

Matchers

A Matcher attempts to match an URI with a specified pattern for dispatching the request to a specific processing pipeline.

Different types of matchers:

- ◆ wildcard matcher
- ◆ regexp matcher

More matchers can be added without modifying Cocoon.

Matchers help in specifying a specific pipeline processing for a group of URIs.

Sitemap entries for different types of matchers

```
<map:matchers default="wildcard">
```

Introducing Cocoon

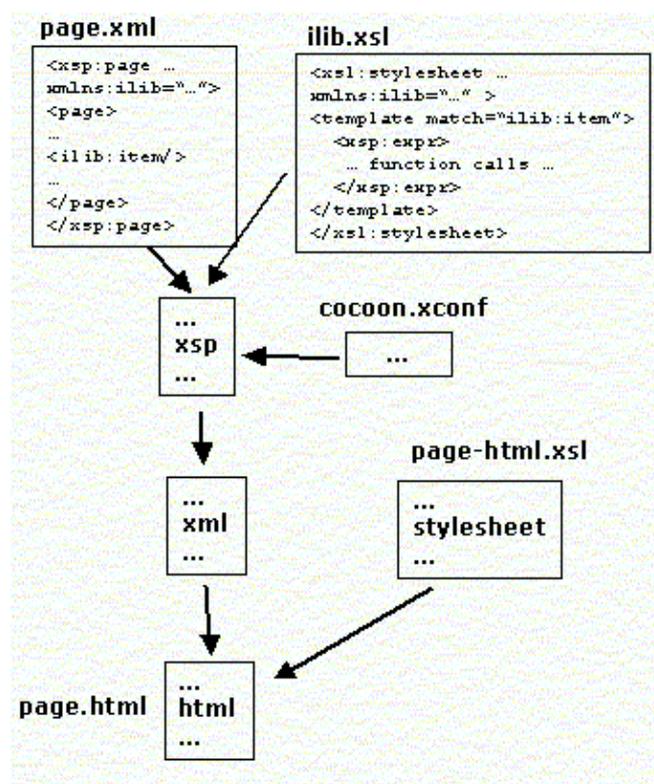
```
<map:matcher name="wildcard" factory="org.apache.cocoon.matching.WildcardURI">
  <map:matcher name="regexp" factory="org.apache.cocoon.matching.RegexpURIMatcher">
  </map:matchers>
```

Pipeline entries in sitemap file

```
<map:match pattern="jsp/*">
  <map:generate type="jsp" src="/docs/samples/jsp/{1}.jsp"/>
  ...
</map:match>
<map:match pattern="hello.pdf">
</map:match
```

Generators

A Generator is used to create an XML structure from an input source (file, directory, stream ...)



Different types of generators:

- ◆ file generator
- ◆ directory generator
- ◆ XSP generator
- ◆ JSP generator
- ◆ Request generator
- ◆ ...

More generators can be added without modifying Cocoon.

Sitemap entries for different types of generators

```
<map:generators default="file">
  <map:generator name="file"
    src="org.apache.cocoon.generation.FileGenerator"
```

Introducing Cocoon

```
        label="content" />
<map:generator name="directory"
  src="org.apache.cocoon.generation.DirectoryGenerator"
  label="content" />
<map:generator name="serverpages"
  src="org.apache.cocoon.generation.ServerPagesGenerator"
  label="content" />
<map:generator name="request"
  src="org.apache.cocoon.generation.RequestGenerator" />
...
</map:generators>
```

A sample generator entries in a pipeline

```
<map:match pattern="hello.html">
  <map:generate src="docs/samples/hello-page.xml" />
  <map:transform src="stylesheets/page/simple-page2html.xsl" />
  <map:serialize type="html" />
</map:match>
```

A Generator turns an XML document, after applying appropriate transformations, into a compiled program whose output is an XML document.

An XSP generator applies all the logicsheets specified in the source XML file before generating the program.

Generators cache the compiled programs for better runtime efficiency.

Transformers

A Transformer is used to map an input XML structure into another XML structure.

Different types of transformers:

- ◆ XSLT Transformer
- ◆ Log Transformer
- ◆ SQL Transformer
- ◆ I18N Transformer
- ◆ ...

Log Transformer is a good debugging tool.

More transformers can be added without modifying Cocoon.

Sitemap entries for different types of transformers

```
<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="log" src="org.apache.cocoon.transformation.LogTransformer"
  ...
</map:transformers>
```

A sample transformer entry in a pipeline

Introducing Cocoon

```
<map:match pattern="hello.html">
  <map:generate src="docs/samples/hello-page.xml" />
  <map:transform src="stylesheets/page/simple-page2html.xsl" />
  <map:serialize type="html" />
</map:match>
```

Serializers

A Serializer is used to render an input XML structure into some other format (not necessarily XML)

Different types of serializers:

- ◆ HTML Serializer
- ◆ FOP Serializer
- ◆ Text Serializer
- ◆ XML Serializer
- ◆ ...

More serializers can be added without modifying Cocoon.

Sitemap entries for different types of serializers

```
<map:serializers default="html">
  <map:serializer name="xml"
    mime-type="text/xml"
    src="org.apache.cocoon.serialization.XMLSerializer" />
  <map:serializer name="html"
    mime-type="text/html"
    src="org.apache.cocoon.serialization.HTMLSerializer" />
  <map:serializer name="fo2pdf"
    mime-type="application/pdf"
    src="org.apache.cocoon.serialization.FOPSerializer" />
  <map:serializer name="vrmf"
    mime-type="model/vrmf"
    src="org.apache.cocoon.serialization.TextSerializer" />
  ...
</map:serializers>
```

A sample serializer entry in a pipeline

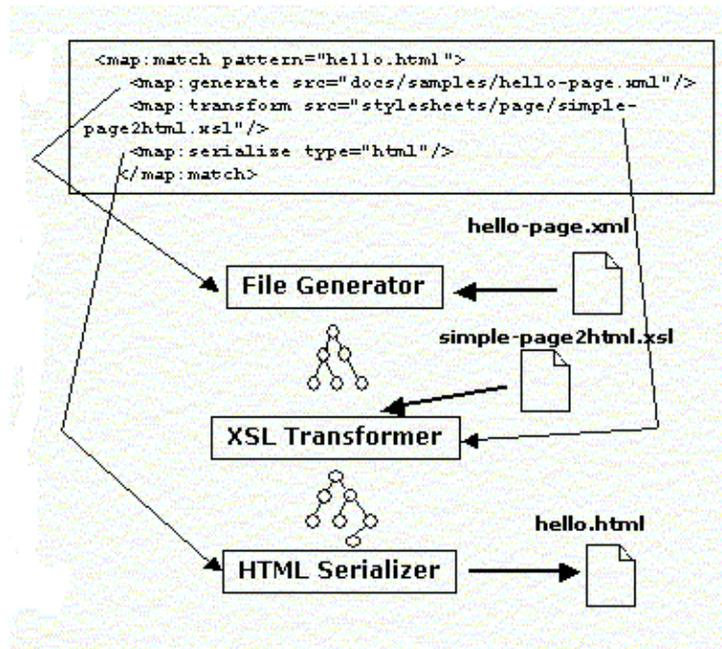
```
<map:match pattern="hello.html">
  <map:generate src="docs/samples/hello-page.xml" />
  <map:transform src="stylesheets/page/simple-page2html.xsl" />
  <map:serialize type="html" />
</map:match>
```

Pipeline Processing

The sitemap configuration allows dynamic setup of processing pipelines consisting of a generator, multiple transformers and a serializer.

Requests are dispatched to a pipeline based on request URI and the pipeline matching pattern (either with wildcards or as a regexp)

The pipeline is setup in the generated file `sitemap_xmap.java` (This file gets generated [possibly asynchronously] everytime the `sitemap.xmap` is modified.



Logicsheets

Logicsheets are XSL files with an associated namespace.

Primary mechanism to add program logic (code) to XSPs.

These need to be registered in configuration file cocoon.xconf.

Logicsheets are used by the generator to transform XML structure before generating program.

Cocoon comes with a no. of built-in logic sheets:

- ◆ request.xsl
- ◆ response.xsl
- ◆ session.xsl
- ◆ cookie.xsl
- ◆ esql.xsl
- ◆ log.xsl
- ◆ ...

Log.xsl structure

```

<xsl:stylesheet version="1.0"
  xmlns:xsp="http://apache.org/xsp"
  xmlns:log="http://apache.org/xsp/log"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="log:logger">
    ... variable and xsp:logic statements ...
  </xsl:template>

  <xsl:template match="log:debug">
    <xsp:logic>
      if(getLogger() != null)
        getLogger().debug("<xsl:value-of select="."/>");
    </xsp:logic>
  </xsl:template>

```

```
</xsl:template>
<xsl:template match="log:error">
...
</xsl:template>
</xsl:stylesheet>
```

A sample use

```
<xsp:page language="java"
          xmlns:xsp="http://apache.org/xsp"
          xmlns:log="http://apache.org/xsp/log">

  <page>
    <log:logger name="test" filename="test.log"/>
    <log:debug>Test Message</log:debug>
  </page>
</xsp:page>
```

Apache Cocoon Configuration.

Cocoon is highly configurable. Main configuration files, assuming Cocoon deployment as a servlet in a servlet container, are (directory locations assume Tomcat servlet container):

- ◆ `sitemap.xmap`: the sitemap file. By default, located in `$TOMCAT_HOME/webapps/cocoon` directory.
- ◆ `cocoon.xconf`: configuration file having logicsheet registrations. Specifies, `sitemap.xmap` location and other such parameters. By default, located in `$TOMCAT_HOME/webapps/cocoon` directory.
- ◆ `web.xml`: servlet deployment descriptor. Specifies location of `cocoon.xconf`, log file location and other such parameters. Located in `$TOMCAT_HOME/webapps/cocoon/WEB-INF` directory.
- ◆ `cocoon.roles`: mapping file for Core Cocoon components name and implementation classes. For example, if you want to use a parser other than the default one, you need to modify this file.

Apache Cocoon Work Area

Cocoon produces execution log entries for debugging/auditing.

- ◆ The amount of data to be logged can be controlled by `log-level` parameter in `web.xml` file. The default is `DEBUG` (maximum data).
- ◆ By default, the log file is:
`$TOMCAT_HOME/webapps/cocoon/WEB-INF/logs/cocoon.log`.

Cocoon keeps the generated `.java` files in a directory tree starting at (by default):
`$TOMCAT_HOME/webapps/work/localhost_8080%2Fcocoon/org/apache/coco`

You can find `sitemap_xmap.java` here.

Files created by `LogTransformer` are kept (by default) in `$TOMCAT_HOME` directory.

Use with Tomcat

Download Tomcat from Apache site.

Download Cocoon sources from Apache CVS. [Command assume UNIX Bourne shell]

```
export CVSROOT=:pserver:anoncvs@cvs.apache.org:/home/cvspublic
cvs login
Password: anoncvs
cvs checkout xml-cocoon2
```

Build sources as per instruction in Install file.

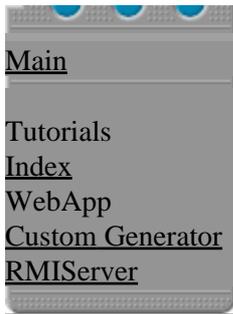
Move the `cocoon.war` file to `$TOMCAT_HOME/webapps` directory.

Start the servlet engine. Type-in the URL `http://localhost:8080/cocoon` in your browser. You should see the Cocoon welcome message.

Consult Install file if you face problems.

Copyright © 1999–2002 The Apache Software Foundation. All Rights Reserved.

```
a.menu { color: #FFFFFF; text-align:left; font-size:12px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:plain; text-decoration:none; padding-left: 14px } A.menu:hover { color: #FFCC00 } .menutitle
{ color: #000000; text-align:left; font-size:10px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:bold; padding-left: 8px } .menuselected { color: #FFCC00; text-align:left; font-size:12px;
font-family: Verdana, Arial, Helvetica, sans-serif; font-weight:bold; padding-left: 14px }
```



Introduction

Apache Cocoon is an XML publishing framework. It allows you to define XML documents and transformations to be applied on it, to eventually generate a presentation format of your choice (HTML, PDF, SVG, etc.). Cocoon also gives you the possibility to have logic in your XML files (so that the XML file itself can become dynamically generated).

Cocoon is developed on top of the Avalon Server Framework, which is a stable and scalable framework. You can find out more about Avalon in this document: (ref: Avalon White Paper). I highly suggest reading this white paper as it covers many concepts that are key to Cocoon, namely Separation of Concerns (SOC) and Inversion of Control (IoC). It also covers foundational aspects of the Avalon Framework, so you can have a better understanding on how Cocoon is structured.

Cocoon helps you separate out concern areas for web development. The areas addressed are Logic, Content, Style, and Management. There are different mechanisms for each.

In order to learn how to use Cocoon, first make sure that you install it properly, then investigate the many samples. The following screenshots come from the "tutorial" that is provided with Cocoon. After you have built the demo webapp as per the installation instructions (`build webapp`) then you can see this tutorial in action via the Samples pages.

Separating Concerns

Cocoon is designed to allow Developers, Business Analysts, Designers, and Administrators to work with each other without breaking the other person's contribution. The problem with using just JSPs, ASPs, or ColdFusion templates is that all of the look, feel, and logic are intertwined. That means that maintenance is much more difficult, and the project's true costs are delayed until the customer wants feature enhancements or bugs fixed. This also means that if the site design is introduced late in the game, the cost of revamping the site becomes much higher.

Developers

Developer's jobs are to create the business logic and object model behind the web application. They are more concerned with functionality than with layout or the words displayed on a screen. These are the people that will develop the Actions (Components that only process information) and the hooks for how to get the necessary information from business objects.

Business Analysts

The Business Analysts are the people who are concerned with the words displayed on the screen, and to a certain extent, the layout. Typically, they will be using the work done by the developer to put together a generic markup that will be transformed into the results. In small development environments, many times the developer takes on both this role and the developer role. Typically, the business analyst will be working with the markup language that goes into the generator.

Designers

The designer is the person or group of people who are responsible to provide the final look and feel of a site. The designer does all the graphics and HTML code. In Cocoon, they will be working with the Transformers that take an input and structure it in a final presentation.

Administrators

The administrator is responsible for the sitemap which maps the URI space to the different pipelines in Cocoon. A pipeline is a path from a Generator to a Serializer. This means, that the administrator decides that all requests for a resource with a ".html" extension starts out as XML and ends up as HTML. The Administrator will work closely with the Designers and the Developers. In the absence of a dedicated administrator, one developer should assume that role. It is important that developers do not get bogged down in this one Component.

Development Style

You have to decide early on whether you will develop from a Business Markup perspective, or develop from a Document Markup perspective. They have different ways of approaching the same problem. Both approaches have its tradeoffs. In the end, you will find that you will need a combination of different aspects of the two approaches.

Business Markup Centric

This approach makes the Business Object the center of attention for development. This approach formalizes your business objects, and makes sure that you always represent a business object in a standard manner. It's limitations come to bear when you have cases when you need two different objects that need to be represented on the same logical page.

Document Markup Centric

This approach feels the most natural to developers who come from

backgrounds with scripting languages. This approach is a bit more flexible in that you represent a page logically, with the wording as the center of attention. With this approach, it is up to the developer to ensure that the business object is represented in a consistent manner.

Hybrid Approach

We will develop a hybrid approach to development in this paper. What this means is that we start with a Document Markup Centric approach, and add in support for specific Business Markup as it is needed. In the end, this is the most flexible and maintainable method for development.

The Concept

For the sake of this paper, we are going to develop a very simple database-backed application that manages users and departments. Each element has a name and an identifier. A department can have many employees, but each employee can only have one department. We will be able to create, change, and delete both employees and departments.

The SQL

```
CREATE TABLE department {
    department_id INT NOT NULL,
    department_name VARCHAR (64) NOT NULL
};

CREATE TABLE employee {
    employee_id INT NOT NULL,
    employee_name VARCHAR (64) NOT NULL,
    department_id INT NOT NULL
};

ALTER TABLE department ADD
    PRIMARY KEY pkDepartment (department_id);

ALTER TABLE employee ADD
    PRIMARY KEY pkEmployee (employee_id);

ALTER TABLE employee ADD
    FOREIGN KEY department_id (department.department_id);
```

Facilities

1. Create Department (need name only)
2. Update Department (change name, reassign potential employees to department, create employee for department)
3. Delete Department
4. Find Department (by name, or by ID)
5. Create Employee (need name and department—create department if needed)
6. Update Employee (change name, reassign department—create department if needed)
7. Delete Employee
8. Find Employees (by name, by ID, or by Department)

Layouts

Various screenshots are available as a separate document, to portray the layout of interfaces and results pages – apply your own style.

Diving In

In order to do anything in Cocoon, you will need a sitemap. At this point we will not go into detail but we will show you how to put an entry in so you can see your stuff. In most development situations, the sitemap will be set up for you. Since we want to start with a clean slate, take the sitemap that comes with Cocoon's samples and clear out everything under the `<map:pipeline>` tag. Next, you will add an entry in the same location that looks like this:

```
<map:pipeline>
  <map:match pattern="">
    <map:redirect-to uri="home.html"/>
  </map:match>

  <map:match pattern="**.xml">
    <map:generate src="docs/{1}.xml"/>
    <map:serialize type="xml"/>
  </map:match>

  <map:match pattern="**.html">
    <map:generate src="docs/{1}.xml"/>
    <map:transform src="stylesheets/apache.xsl"/>
    <map:serialize/>
  </map:match>

  <map:match pattern="images/**/*.gif">
    <map:read src="resources/images/{1}.gif" mime-type="image/gif"/>
  </map:match>

  <map:match pattern="images/**/*.jpg">
    <map:read src="resources/images/{1}.jpg" mime-type="image/jpg"/>
  </map:match>

  <map:match pattern="images/**/*.png">
    <map:read src="resources/images/{1}.png" mime-type="image/png"/>
  </map:match>

  <map:match pattern="resources/**/*.css">
    <map:read src="resources/styles/{1}.css" mime-type="text/css"/>
  </map:match>

  <map:match pattern="resources/**/*.js">
    <map:read src="resource/styles/{1}.js"
      mime-type="application/x-javascript"/>
  </map:match>

  <map:handle-errors>
    <map:transform src="stylesheets/system/error2html.xsl"/>
    <map:serialize status-code="500"/>
  </map:handle-errors>
</map:pipeline>
```

What this does is tell the sitemap that we want to capture all URLs with a

".xml" extension, and find an equivalent file in the "docs" subdirectory. We are not performing any transformations at this time. The Sitemap is really a site administrator's job to maintain. There are some exceptions to this general rule, but we will discuss them when needed. We will use the Document Markup specified in the StyleBook DTD format.

Creating the Pages

Since we are only looking at XML right now, we need to make sure our pages conform to the markup standards. You will see how well this comes in handy for debugging XSP (XML Server Pages) markup. Since we already have the Layout specified, and the database created, we will create our markup.

Our home page is going to be really simple: a list of links that take us to the main pages.

```
<document>
  <header>
    <title>Home Page</title>
  </header>
  <body>
    <sl title="Welcome to Personnel Administrator">
      <p>
        Welcome to our Personnel Administrator. You
        can perform one of the following functions:
      </p>
      <ul>
        <li>
          <link href="search-dept.html">Search Departments</link>
        </li>
        <li>
          <link href="search-empl.html">Search Employees</link>
        </li>
        <li>
          <link href="create-dept.html">Create Departments</link>
        </li>
        <li>
          <link href="edit-dept.html">Edit a Department</link>
        </li>
        <li>
          <link href="create-empl.html">Create Employee</link>
        </li>
        <li>
          <link href="edit-empl.html">Edit an Employee</link>
        </li>
      </ul>
    </sl>
  </body>
</document>
```

Even though this doesn't look like much right now, we have two entries: "***.xml" and "***.html" for the same resource. Look at "home.html", and see how it looks now. Quite a difference. Don't remove the entry for viewing the page as XML yet. We need to use it to debug our XSP pages later.

Our First Form

For now, we are going to skip the search functionality, and jump to our

Introducing Cocoon

"create" templates. It is important to realize the proper method of form handling. While it is possible to create XSP pages that perform the logic for you, this approach is not very maintainable. We also have to choose whether we will directly access the database, or encapsulate that logic in objects.

The tradeoffs are that the direct SQL access is faster to get started, but that it is harder to maintain in the end. You may decide to start with the direct SQL access at the beginning of a project, and build the objects later. With that in mind, we will use some functionality that Cocoon has built in to make this approach a little easier. Cocoon has a group of Database actions that allow you to map form fields to dynamically created SQL calls. It also has a logicsheet that makes creating SQL bound pages a little easier.

Our first form is the "Create a Department" form. The website specification is missing the tags for form building, we will provide an example here:

```
<document>
  <header>
    <title>Department</title>
  </header>
  <body>
    <s1 title="Create a Department">
      <form handler="create-dept.html">
        <p>
          You can create a department by typing in the
          name and pressing the "submit" button.
        </p>
        <p>
          Name: <text name="name" size="30" required="true"/>
        </p>
        <submit name="Create Department"/>
        <note>
          * These fields are required.
        </note>
      </form>
    </s1>
  </body>
</document>
```

It is important to note that the "submit" tag is transformed into an HTML submit button with the name "cocoon-action-ACTIONNAME". The "cocoon-action-ACTIONNAME" form parameter is a magic value that Cocoon uses to select a specific action from a group of actions that only gets executed during that time. You will find that this page displays correctly, but does not do anything yet. The handler is where the navigation goes once you click on the "Create Department" button on the screen. What we are going to do is create one confirmation page for all the Department and Employee pages.

Cocoon has a FormValidatorAction that will take care of ensuring the input results are acceptable. It also has the following database actions for your convenience: DatabaseAddAction, DatabaseUpdateAction, DatabaseDeleteAction, and DatabaseAuthenticatorAction. We will only need the Add, Update, and Delete actions for our simple webapp. In order to prepare them, we create an XML configuration file that tells the actions how to map request parameters to database tables and place constraints on the parameters. For the Department form group, it will look like this:

Introducing Cocoon

```
<root>
  <!--
    The "parameter" elements identify the root constraints for
    the FormValidatorAction. We are specifying that the "id"
    parameter is an integer (it limits to "long", "double",
    "boolean", and "string"). We are specifying that the "name"
    parameter is a string that is at least 5 characters--but no
    more than 64 characters.
  -->
  <parameter name="id" type="long"/>
  <parameter name="name" type="string" min-len="5" max-len="64"/>

  <!--
    Each constraint set is used when we are defining a new way
    of validating a form. We define our constraint sets by
    function. Since we have the same basic form that is driving
    the FormValidator, we have an update set and an add set.

    Note that you can impose additional constraints than the
    default constraints listed above. Also, you do not "have"
    to enforce a constraint. Each "validate" element below
    identifies the parameter constraints we are enforcing.

    For more information view the JavaDocs for
    AbstractValidatorAction
  -->
  <constraint-set name="update">
    <validate name="name"/>
    <validate name="id" nullable="no" min="1"/>
  </constraint-set>

  <constraint-set name="add">
    <validate name="name"/>
  </constraint-set>

  <!--
    This is where we identify our table mappings so that the
    Database Actions can work their magic. Note that the
    parameter names are the same as above--as well as the same
    as form parameter names.

    First we tell the Database Actions that we are using the
    "personnel" connection pool we set up in <code>cocoon.xconf</code>.
    This file should be set up by the site administrator.

    We also tell the Database Actions the structure of the table
    we will be populating. The keys are used to identify which
    columns will be treated as keys--they are treated different
    when the different SQL statements are created. Note that
    there is a "mode" attribute in the key element. The mode
    refers to how new keys will be generated. There are three
    modes: "automatic" keys are generated by the database,
    "manual" keys are generated by manually finding the largest
    value and incrementing it, and finally "form" keys take the
    key value from a parameter on the form.

    Both keys and values serve to map parameter names to table
    columns, converting the value into the native type. For a
    list of supported types check out the JavaDocs for
    AbstractDatabaseAction.
  -->
  <connection>personnel</connection>
  <table name="department">
    <keys>
```

Introducing Cocoon

```
<key param="id" dbcol="department_id" type="int" mode="manual"/>
</keys>
<values>
  <value param="name" dbcol="department_name" type="string"/>
</values>
</table>
</root>
```

After you create the descriptor file, you will have to create some entries in the Sitemap so you can take advantage of the form descriptor. First, the Sitemap has to be able to know how to reference the Actions we want. To do that, alter the "map:actions" section to list all the actions we need:

```
<map:actions>
  <map:action name="dbAdd"
    src="org.apache.cocoon.acting.DatabaseAddAction"/>
  <map:action name="dbDel"
    src="org.apache.cocoon.acting.DatabaseDeleteAction"/>
  <map:action name="dbUpd"
    src="org.apache.cocoon.acting.DatabaseUpdateAction"/>
  <map:action name="form"
    src="org.apache.cocoon.acting.FormValidatorAction"/>
</map:actions>
```

Lastly, we want to create an action set. An action set is a group of actions that will be applied at once. If the action set entry has an "action" parameter, then the specific action is only executed when the ACTIONNAME of the magic "cocoon-action-ACTIONNAME" request parameter matches the value of the "action" parameter. For our purposes, the action set we are defining is listed below (defined in the sitemap):

```
<map:action-sets>
  <map:action-set name="process">
    <map:act type="form" action="Create Department">
      <map:parameter name="validate-set" value="add"/>
      <map:act type="dbAdd"/>
    </map:act>
    <map:act type="form" action="Update Department">
      <map:parameter name="validate-set" value="update"/>
      <map:act type="dbUpd"/>
    </map:act>
    <map:act type="dbDel" action="Delete Department"/>
  </map:action-set>
</map:action-sets>
```

Now that we have defined the actions we want, with the parameters that control them during run-time, we can use it in our pipeline.

```
<map:match pattern="*-dept.html">
  <map:act set="process">
    <map:parameter name="descriptor"
      value="context://docs/department-form.xml"/>
    <map:parameter name="form-descriptor"
      value="context://docs/department-form.xml"/>
    <map:generate type="serverpages" src="docs/confirm-dept.xsp"/>
    <map:transform src="stylesheets/apache.xsl"/>
    <map:serialize/>
  </map:act>
</map:match>
```

Introducing Cocoon

```
</map:act>
<map:generate type="serverpages" src="docs/{1}-dept.xsp"/>
<map:transform src="stylesheets/apache.xsl"/>
<map:serialize/>
</map:match>

<map:match pattern="*-dept.xml">
  <map:act set="process">
    <map:parameter name="descriptor"
      value="context://docs/department-form.xml"/>
    <map:parameter name="form-descriptor"
      value="context://docs/department-form.xml"/>
    <map:generate type="serverpages" src="docs/confirm-dept.xsp"/>
    <map:serialize type="xml"/>
  </map:act>
  <map:generate type="serverpages" src="docs/{1}-dept.xsp"/>
  <map:serialize type="xml"/>
</map:match>
```

This may not seem clear what is happening right now. The way actions work is if they return a null, nothing inside the "map:act" entry will execute, and the request processing will flow through to the second "map:generate" section. This is a side affect of using the FormValidatorAction. If we choose to create our own business objects and form validation framework, we are not constrained by this construct.

In addition, we changed the type of generator we are using: we have made it a "serverpages" (or XSP) generator. We made the transition now so that we can report information on what failed to the user. First, we need to convert our "create-dept.xml" file to an XSP page so that we can see the page again (right now we will get an error). To do this, simply add a new tag to the base of the document called "xsp:page" declaring the XSP namespace. The change will look like this:

```
<xsp:page xmlns:xsp="http://apache.org/xsp">
  <!-- The original document will be embedded here -->
</xsp:page>
```

To complete the transformation, we usually change the extension to ".xsp" so we know what we are dealing with at a glance. Create a new file called "confirm.xsp" with the following contents:

```
<xsp:page xmlns:xsp="http://apache.org/xsp">
<document>
  <header>
    <title>Department</title>
  </header>
  <body>
    <s1 title="Department Processed">
      <p>
        You have successfully processed the department.
      </p>
    </s1>
  </body>
</document>
</xsp:page>
```

Adding support for Error Reporting

In order to successfully report errors processing the page, add another namespace declaration to the "xsp:page" element. The final form page will look like this:

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
          xmlns:xsp-formval="http://apache.org/xsp/form-validator/2.0">
<document>
  <header>
    <title>Department</title>
  </header>
  <body>
    <sl title="Create a Department">
      <form handler="create-dept.html">
        <p>
          You can create a department by typing in the
          name and pressing the "submit" button.
        </p>
        <p>
          Name: <text name="name" size="30" required="true"/><br />
          <xsp:logic>
            if (<xsp-formval:is-toosmall name="name"/>) {
              <xsp:text>"Name" must be at least 5 characters</xsp:text>
            } else if (<xsp-formval:is-toolarge name="name"/>) {
              <xsp:text>"Name" was too long</xsp:text>
            }
          </xsp:logic>
        </p>
        <submit name="Create Department"/>
        <note>
          * These fields are required.
        </note>
      </form>
    </sl>
  </body>
</document>
</xsp:page>
```

Adding Database Support with the ESQL Logicsheet

The "Create Employee" page is going to require database access so that we know which Department a new employee is assigned to. This is fairly easy to accomplish with the ESQL Logicsheet. Again, when you use the ESQL logicsheet, you lose some of your separation of concerns.

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
          xmlns:xsp-formval="http://apache.org/xsp/form-validator/2.0"
          xmlns:esql="http://apache.org/cocoon/SQL/v2">
<document>
  <header>
    <title>Employee</title>
  </header>
  <body>
    <sl title="Create an Employee">
      <form handler="create-empl.html">
        <p>
          You can create a department by typing in the
          name and pressing the "submit" button.
        </p>
      </form>
    </sl>
  </body>
</document>
</xsp:page>
```

Introducing Cocoon

```
</p>
<p>
  Name: <text name="name" size="30" required="true"/><br />
  <xsp:logic>
    if (<xsp-formval:is-null name="name"/>) {
      <xsp:text>"Name" cannot be empty</xsp:text>
    } else if (<xsp-formval:is-toolarge name="name"/>) {
      <xsp:text>"Name" was too long</xsp:text>
    }
  </xsp:logic>
</p>
<p>
  Department:
  <select name="department">
    <esql:connection>

      <!-- declare the connection pool we are using -->
      <esql:pool>personnel</esql:pool>

      <!-- query execution blocks can be repeated -->
      <esql:execute-query>

        <!-- Find all departments and order them -->
        <esql:query>
          SELECT department_id, department_name
          FROM department ORDER BY department_name
        </esql:query>

        <!-- What to do with the results -->
        <esql:results>
          <!--
            A successful query that returns results
            executes this block. You can also embed
            more "execute-query" blocks inside the
            row-results. That way you can have queries
            that filter information based on the results
            of other queries.
          -->
          <esql:row-results>
            <option>
              <xsp:attribute name="name">
                <esql:get-string column="department_id"/>
              </xsp:attribute>
              <esql:get-string column="department_name"/>
            </option>
          </esql:row-results>
          <!--
            Other result types are "no-results" and
            "error-results". A successful query that
            does not return results (an empty resultset)
            will use the XML embedded in the "no-results"
            section. An unsuccessful query that throws
            an exception will use the XML embedded in
            the "error-results" section.
          -->
          </esql:results>
        </esql:execute-query>
      </esql:connection>
    </select>
  </p>
  <submit name="Create Employee"/>
  <note>
    * These fields are required.
  </note>
</form>
```

```
</s1>  
</body>  
</document>  
</xsp:page>
```

As you can see ESQL is flexible and powerful, but the cost of that flexibility is a loss of readability. Using a logicsheet to wrap information in a business object is another alternative. Notice how ESQL works:

- ◆ First, we specify our connection information which will apply to all queries in the ESQL structure.
- ◆ Next, we specify our first query we are going to use. Note that you can nest queries as well as have more than one in an "esql:connection" element.
- ◆ Lastly, we specify how we process the results. There are three different types of results: "esql:row-results", "esql:no-results", and "esql:error-results". This allows you to handle different scenarios easily. It is inside the individual results elements that we can nest new queries to process.

A Note About Actions

Actions are the bread and butter of logic processing in Cocoon. There are a number of approaches that you can take when developing Actions. You can create a specific action for each piece of business logic. This approach is very heavy handed and requires you to spend a lot of development time creating actions.

The preferred method for creating actions is to provide a generic action that can handle a wide range of specific actions. The Database Actions and Validator Actions are examples of this approach. They will read a configuration file specified by a parameter, and they will modify the specific results based on the configuration file. In order to take advantage of this for your own Actions, you can extend the `AbstractComplimentaryConfigurationAction`. Basically what it does is encapsulate the logic for reading and caching the Configuration information for your Action.

Redirects

Most web developers agree that redirecting a user based on input is a valuable and necessary part of web development. In Cocoon there are only two locations where you can issue redirects: the Sitemap and Actions. In essence, Cocoon does require you to plan so that redirects are only used when necessary.

One approach that is good to use is to require all traffic to go through a URL controlling action. The Action will test to see if the user is logged in, and if not will send them to the login page. Another derivation on this approach is to test for a user's role, and if they do not have access redirect them to a different page.

Writing an Action

Writing an action is as simple as writing a Component that conforms to the Action interface. Be sure to examine the different Actions that are in the `org.apache.cocoon.acting` package – you might find some abstract actions that you can extend. Actions are Avalon Components, so you may want to read Avalon's Whitepaper for more information.



Actions will return a map that contains values that the sitemap administrator can use in the sitemap. If the Action returns a null, then anything inside the "map:act" element will not be executed.

Return Values

The Action interface specifies that it returns a Map. This Map is used for value substitution in the sitemap, and communicating information to other Actions. When an Action is specified in the sitemap, it uses the following syntax:

```
<map:act type="my-action">
  <map:generate src="{source}" />
  <map:transform src="doc2{theme}" />
  <map:serialize/>
</map:act>
```

The above code snippet assumes you have an Action with the name "my-action" already specified. It also assumes that there are two "parameters" returned from the action in the Map. The sitemap queries the returned Map for the "source" and "theme" values, and substitutes their values in place of the curly braces that referenced it. In other words, when it sees the "map:generate" with an src attribute of "{source}" it looks in the Map. For our discussion, let us say the value stored is "index.xml". The Sitemap will perform the substitution so that the src attribute now contains "index.xml".

In the case that the above the action might return a null value. In that case, everything inside the "map:act" element is skipped. You can use this to good advantage like the *ValidatorActions do. If everything is validated correctly, they return a Map. If there is an error, they return a null, and place the information in Request attributes.

Cocoon Supplied Components

Cocoon supplies a number of different Components for your use. The types of Components we will discuss here are Generators, Transformers, Serializers, Readers, and Actions. These are the important Components that allow you to do your job.

Generators

A Generator will create SAX events for a SAX stream—whether it reads from an input stream or it generates it on the fly. All built-in generators are in the package "org.apache.cocoon.generation".

DirectoryGenerator

Introducing Cocoon

Reads a directory, and builds an XML document based on the contents. You can pass parameters to it to control how it behaves (note parameter names are case sensitive):

- ◆ `dateFormat` – a format string that you would use in the Java `SimpleDateFormat` object
- ◆ `depth` – the maximum number of directories deep the generator will look (defaults to 1)
- ◆ `root` – a regular expression to find the root directory
- ◆ `include` – a regular expression to declare the files/directories that will be included in the list
- ◆ `exclude` – a regular expression to declare the files/directories that will not be included in the list

When you use this Generator, you must have the Jakarta Regexp package installed in your `WEB-INF/libs` directory. Also, the `DirectoryGenerator` is not Cacheable so the results will be generated fresh each time.

The resulting XML looks like this:

```
<?xml version="1.0"?>
<directory xmlns="http://apache.org/cocoon/directory/2.0"
  name="C:\path\dir\"
  lastModified="135432153351"
  date="11 Jun 2001">
  <file name="C:\path\dir\file.xml" lastModified="135432153351"
    date="11 Jun 2001"/>
</directory>
```

FileGenerator

This generator and the `ServerPagesGenerator` will be your most used generators. The `FileGenerator` reads an XML file from an input source, and converts it into a SAX stream.

When you use this Generator, you must have a JAXP 1.1 compliant parser installed in your `WEB-INF/libs` directory. You may also use the Xerces parser bypassing the JAXP requirement. The `FileGenerator` is Cacheable, so the results will only be re-read when the file changes.

FragmentExtractorGenerator

This generator is used in conjunction with the `FragmentExtractorTransformer` (more on that in the transformers section). The `FragmentExtractorTransformer` splits an XML document into smaller parts so you can treat each smaller part as a unique document. To see this in action, check out the Cocoon supplied [samples](#) and click on the SVG Welcome page.

This Generator caches the results from the `FragmentExtractorTransformer` for quick retrieval later. It is Cacheable, so the fragments are generated once and the cached version is read from that point forward.

HTMLGenerator

This generator is used to read in an HTML file that may not be properly formatted to comply with XML standards. The result is properly formatted XHTML.

This generator requires the Tidy.jar file installed in the WEB-INF/libs directory. The HTMLGenerator is Cacheable, so the results can be cached for application speedup.

ImageDirectoryGenerator

This generator is an extension of the DirectoryGenerator, so it has the same requirements. It extends the markup to include two new attributes for the "file" element: "height" and "width". The ImageDirectoryGenerator reads every GIF and JPEG file to get the dimensions.

This generator is not Cacheable (just like the DirectoryGenerator).

JspGenerator

This generator executes a JSP file and parses the result. The JSP must generate valid XML, and be a file in the context.

This generator requires a JAXP 1.1 compliant parser or Xerces if your environment will not allow you to install one. It is also not cacheable so the results are generated each time.

PhpGenerator

This generator functions just like the JspGenerator, but with PHP templates. The PHP must generate valid XML, and be a file in the context.

This generator requires a JAXP 1.1 compliant parser and the phpservlet.jar file that comes from <http://php.net>. Install the files in the WEB-INF/libs directory. The PhpGenerator is not Cacheable.

RequestGenerator

This generator converts the Request object into an XML representation. It is best used for debugging purposes. The resulting XML follows:

```
<request xmlns="http://xml.apache.org/cocoon/requestgenerator/2.0"
        target="index.html" source="context://docs/index.xml">

  <requestHeaders>
    <header name="HOST_NAME">johny-bravo.infoplanning.com</header>
    <!-- repeat for each header -->
  </requestHeaders>

  <requestParameters>
    <parameter name="form-param">
      <value>1</value>
      <!-- repeat for each value in "form-param" -->
    </parameter>
  </requestParameters>
</request>
```

Introducing Cocoon

```
<!-- repeat for each parameter -->
</requestParameters>

<configurationParameters>
  <parameter
    name="configurations">context://WEB-INF/cocoon.xconf</parameter>
  <!-- repeat for each parameter -->
</configurationParameters>
</request>
```

The RequestGenerator does not have any special requirements for libraries, and it is not Cacheable.

ScriptGenerator

The ScriptGenerator uses the Bean Scripting Framework (BSF) and an associated interpreter to generate valid XML. If you add language support, you will have to embed the following configuration information:

```
<add-languages>
  <!-- repeat the following for each language: -->
  <language name="kawa-scheme"
    src="org.gnu.kawa.bsf.engines.KawaEngine">
    <extension>scm</extension>
  <!-- repeat for each file extension -->
  </language>
</add-languages>
```

The ScriptGenerator requires that you have the bsf.jar in your WEB-INF/libs directory along with any jars for the script interpreters you use. The ScriptGenerator is not Cacheable.

ServerPagesGenerator

The ServerPagesGenerator is the XML Server Pages (XSP) engine. It automatically compiles a new Generator at runtime based on an input XML file.

This generator requires that you have a JAXP 1.1 compliant parser and XSLT engine installed in your WEB-INF/libs directory. It also requires you to have the JDK's tools.jar file in your classpath. If you reference any packages, they must also be in your classpath. The created generator is not Cacheable.

StatusGenerator

The StatusGenerator is another debug tool. It provides status information for the Cocoon engine. The resultant XML is in the following format:

```
<statusinfo xmlns="http://apache.org/cocoon/status/2.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  host="johnny-bravo.infoplanning.com"
  date="7/16/2001 1:16:42 pm">
  <group name="vm">
    <group name="memory">
      <value name="total"><line>5213255</line></value>
```

Introducing Cocoon

```
<value name="free"><line>12321211</line></value>
</group>
<group name="jre">
  <value name="version"><line>1.3.1</line></value>
  <value name="java-vendor"
    xlink:type="simple"
    xlink:href="http://java.sun.com/jdk/1.3/">
    <line>Sun Microsystems Inc.</line>
  </value>
</group>
</group>
<group name="operating-system">
  <value name="name"><line>Windows 2000</line></value>
  <value name="architecture"><line>x86</line></value>
  <value name="version"><line>5.0</line></value>
</group>
</group>
<value name="classpath">
  <line>C:\tomcat\lib\tomcat.jar</line>
  <line>C:\jdk1.3.1\lib\tools.jar</line>
</value>
</statusinfo>
```

The results are not cacheable, and do not require any special libraries.

StreamGenerator

The StreamGenerator is used to convert the Request's InputStream into a SAX XML stream. Alternately, it will accept the magic form parameter "form-name" and read the input stream that the parameter points to.

This generator requires the JAXP 1.1 compliant parser (or Xerces). It is not cacheable.

VelocityGenerator

The VelocityGenerator is used to convert the output from the Velocity template engine to a valid XML stream.

This generator requires Jakarta Velocity and a JAXP 1.1 compliant parser installed in WEB-INF/libs. It is not Cacheable.

Transformers

Transformers read a SAX stream, manipulate the XML stream, and send the results to the next Component in the chain. All built in generators are in the package "org.apache.cocoon.generation".

CIncludeTransformer

The CIncludeTransformer looks for instances of the "ci:include" element, and will embed another XML resource in your document. That resource can be in the sitemap so you can include the results of processed XSP pages. An example follows:

```
<document xmlns:ci="http://apache.org/cocoon/include/1.0">
  <ci:include src="cocoon://my-resource.xml" />
```

Introducing Cocoon

```
        element="body"  
        ns="http://mycompany.com/my-resource/1.0"  
        prefix="res" />  
</document>
```

The Transformer will read the results from the sitemap, and embed it into this document with a new root element "body" using a new namespace (xmlns:res="http://mycompany.com/my-resource/1.0"). The results are not cached.

FilterTransformer

The FilterTransformer will look for instances of an element you specify using parameters, and will not forward any SAX events for that element or any child elements. You can pass parameters to it to control how it behaves (note parameter names are case sensitive):

- ◆ element-name – The name of the element to filter
- ◆ count – the number of times the element will be filtered
- ◆ blocknr – the element number that filtering begins

FragmentExtractorTransformer

This is transformation half of the FragmentExtractor. This transformer sieves an incoming stream of xml with embedded SVG images and replaces the images with a xlink locator pointing to the image. Ultimately this could be much more general, but currently it is mainly an SVG extraction.

I18nTransformer

This is Cocoon's port of Infozone Group's I18nProcessor. The word i18n is a shorthand for the longer word "internationalization" (starts with 'i', ends with 'n', and has 18 letters in the middle). The internationalization transformer allows you to look up references by key in an XML dictionary. This allows you to support your same business processes in many different countries. You have to pass parameters to it so that it knows how to process i18n requests:

- ◆ default_lang – The default language if the requested language does not exist (two character country code)
- ◆ available_lang_X – Language available by the dictionary (two character country code). Replace the 'X' in the attribute with a number (1, 2, 3).
- ◆ src – The location of the dictionary file.

The I18nTransformer reads the request parameter "lang" to determine which language to display to the user. To translate text either embed the text inside the "i18n:text" element, or the attribute name inside the "i18n:attr" attribute.

```
<document xmlns:i18n="http://apache.org/cocoon/i18n/2.0">  
  <body>  
    <s1 title="Test Title" i18n:attr="title">  
      <p>  
        <i18n:text>This is replaceable text.</i18n:text>  
      </p>
```

```
</s1>  
</body>  
</document>
```

LDAPTransformer

The LDAPTransformer is a class that can be plugged into a pipeline to transform the SAX events which passes through this transformer into queries an responses to/from a LDAP interface.

The Sitemap

This section is meant primarily as a reference for the Sitemap Manager. The person in this role needs to have a better understanding of the sitemap than any other role. The sitemap is a relatively new concept, and as such is subject to refinement. There have been a couple of proposals to replace it with something else, but nothing has been started yet.

The Sitemap is composed of three major parts: component declaration, resource declaration, and pipeline declaration. You will only use a few different types of components in the sitemap: Generators, Transformers, Serializers, Readers, Matchers, Selectors, and Actions. Generators create XML and pass the results in a SAX stream. Transformers read a SAX stream and manipulate the results on the way through. Serializers read a SAX stream, and convert it into the servlet's output stream. Readers read an input stream and copy the results to the servlet's output stream. Matchers and Selectors are used to choose how to process an incoming request. Lastly, Actions are used to perform logic only functions (no display logic).

Below is the root element of all sitemaps:

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

Choosing your Components

As previously discussed, you may choose a number of components to use in your own system. This section identifies the different components you can use, and what they do. Before we begin, I must state that every component is declared in the "map:components" element of the Sitemap:

```
<map:components>  
</map:components>
```

Generators

All generators are declared within the "map:generators" element that is a child of the "map:components" element:

```
<map:generators>  
<map:generator name="file"
```

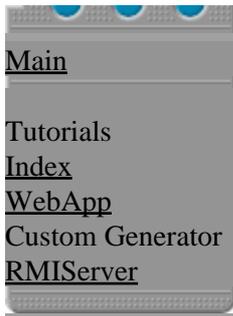
Introducing Cocoon

```
src="org.apache.cocoon.generation.FileGenerator"/>  
</map:generator>
```

Most Generators do not have configuration information, so the "map:generator" element is left empty. If there were configuration information to pass to the generator, it would be placed inside the element. As you can see in the sitemap snippet above, you declare a generator with the "map:generator" element, a "name" attribute, and a "src" attribute. The "name" attribute is how you will refer to this specific type of generator from this point forward. The "src" attribute is the fully qualified class name of the Generator class. In fact this construct is the same for all component types – the only thing that changes is the elements that declare the type of Component we are dealing with.

Copyright © 1999–2002 The Apache Software Foundation. All Rights Reserved.

```
a.menu { color: #FFFFFF; text-align:left; font-size:12px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:plain; text-decoration:none; padding-left: 14px } A.menu:hover { color: #FFCC00 } .menutitle
{ color: #000000; text-align:left; font-size:10px; font-family: Verdana, Arial, Helvetica, sans-serif;
font-weight:bold; padding-left: 8px } .menuselected { color: #FFCC00; text-align:left; font-size:12px;
font-family: Verdana, Arial, Helvetica, sans-serif; font-weight:bold; padding-left: 14px }
```



Introduction

This Tutorial describes the steps necessary to write a basic Cocoon generator. Starting with a quick "Hello World" example and progressing to slightly more involved examples should give a good start to those whose applications call for extending Cocoon with a custom generator.

The intention is to provide:

- ◆ the basics of creating SAX events in a C2 generator
- ◆ a little understanding of the Avalon container contract as it relates to C2 generators
- ◆ a little understanding of the factors that would influence the decision about which xxxGenerator to extend

Purpose

The flexibility to extend the basic "Out of the box" functionality of Cocoon will be an important feature for Cocoon's viability as a broadly used application framework. Though the documentation on "[Extending Cocoon](#)" (at least at this writing) seems to have a hard time imagining applications for custom generators outside of the bizarre, I imagine several scenarios which could call for it:

- ◆ A datasource as yet undeveloped in Cocoon (e.g. event logs)
- ◆ Database driven applications for which XSP is either too awkward or holds too many performance questions. The need for high scalability will drive some (such as myself) to seek optimization in custom generators that just do not seem reasonable to expect out of the auto-generated code that XSPs produce. The current [Performance Tips](#) documentation seems to lead in this direction.
- ◆ Customized control over the caching behaviour if not provided for by other means.

Important

There are other options that should be considered before settling on a new generator. One notable consideration is the option of writing a Source that would fit your needs. See [this discussion](#) from the mailing list for an introduction to the idea. Of course, XSP should be considered – I have not seen any performance comparisons that quantify the benefit that can be had from a custom generator. Finally, be sure you understand the purpose and capabilities of all current standard Generators, as well as those in the scratchpad (for instance, there is a `TextParserGenerator` in the scratchpad at the moment which may be configurable enough to process the event log need mentioned above). Cocoon is a rapidly developing technology that may have anticipated your need. Because the documentation lags behind development, you may find more by examining the source directory and searching the [mail archives](#) for applicable projects.

Intended Audience

This Tutorial is aimed at users who have developed an understanding of the basics of Cocoon and have a need to begin extending it for their own purposes, or desire a deeper understanding of what goes on under the hood.

Prerequisites

Generator developers should have:

- ◆ Read [Cocoon Concepts](#) , as well as [Extending Cocoon](#) , and the broad overview of [Avalon](#) , the framework upon which Cocoon is built.
- ◆ An installed version of Cocoon if you want to follow the examples yourself (obviously).
- ◆ A good understanding of Java.
- ◆ Java SDK (1.2 or later) "installed".

Diving In

Let us start with a simple "Hello World" example:

Simple Example

Our goal will be to build the following document (or, more to the point, the SAX events that would correspond to this document).

```
<example>Hello World!</example>
```

An example of code that will send the correct SAX events down the pipeline:

```
import org.apache.cocoon.generation.AbstractGenerator;
import org.xml.sax.helpers.AttributesImpl;
import org.xml.sax.SAXException;

public class HelloWorldGenerator extends AbstractGenerator
{
    AttributesImpl emptyAttr = new AttributesImpl();
```

Introducing Cocoon

```
/**
 * Override the generate() method from AbstractGenerator.
 * It simply generates SAX events using SAX methods.
 * I haven't done the comparison myself, but this
 * has to be faster than parsing them from a string.
 */

public void generate() throws SAXException
{
    // the org.xml.sax.ContentHandler is inherited
    // through org.apache.cocoon.xml.AbstractXMLProducer

    contentHandler.startDocument();

    contentHandler.startElement("", "example", "example", emptyAttr);

    contentHandler.characters("Hello World!".toCharArray(), 0,
                             "Hello World!".length());

    contentHandler.endElement("", "example", "example");

    contentHandler.endDocument();
}
}
```

So, the basic points are that we extend `AbstractGenerator`, override its `generate()` method, call the relevant SAX methods on the `contentHandler` (inherited from `AbstractGenerator`) to start, fill and end the document. For information on the SAX api, see www.saxproject.org



A performance tip might be to keep an empty instance of `AttributesImpl` around to reuse for each element with no attributes. Also, the `characters(char[] chars, int start, int end)` begs to be overloaded with a version like `characters(String justPutTheWholeThingIn)` that handles the conversion to a character array and assumes you want from beginning to end, as is done in `org.apache.cocoon.generation.AbstractServerPage`. If you are not using namespaces, it is easy to imagine overloaded convenience implementations of the other SAX methods as well. You will probably want to set up a convenient `BaseGenerator` with helpers like this and extend it for your real Generators.

What to Extend?

How did we choose to extend `AbstractGenerator`? Generators are defined by the `org.apache.cocoon.generation.Generator` interface. The only direct implementation of this of interest to us is `AbstractGenerator`, which gives a basic level of functionality. Another option would have been `ComposerGenerator`, which would give us the added functionality of implementing the Avalon interface `Composable`, which would signal the container that handles all the components including our generator to give us a handle back to the `ComponentManager` during the startup of the container. If we needed to lookup a pooled database connection, or some other standard or custom Cocoon component, this is what we would do. Most of the out of the box Generators extend `ComposerGenerator`. Other abstract Generators you may choose to extend include the poorly named (IMHO) `ServletGenerator`, and `AbstractServerPage`. While these both introduce functionality specific to their eventual purpose – the JSP and XSP generators, they do make a convenient starting place for many other Generators.

Running The Sample

In order to run this sample, you will need to compile the code, deploy it into the cocoon webapp, and modify the sitemap to declare our generator and allow access to it via a pipeline.

Compile

Save this source as `HelloWorldGenerator.java` and compile it using

```
javac -classpath %PATH_TO_JARS%\cocoon.jar;%PATH_TO_JARS%\xml-apis.jar
HelloWorldGenerator.java
```

Unfortunately for me, the exact name of your cocoon and xml-apis jars may vary with exactly which distribution, or CVS version you are using, since the community has taken to appending dates or versions at the end of the jar name to avoid confusion. Be sure to find the correct name on your system and substitute it in the classpath. Also, you have several options on where to find jars. If you have a source version that you built yourself, you may want to point to `lib\core\` for them. If you have only the binary version, you can find them in `WEB-INF\lib\`

Deploy

Simply copy the class file into the

`%TOMCAT_HOME%\webapps\cocoon\WEB-INF\classes` directory



If memory serves me, there have been occasional classloading problems in the past that may affect classloading. If your compiled classes are not recognized in the classes directory, try jar-ing them up and place them in `WEB-INF\lib\` instead. That is probably where your real generators would go anyway – with a whole package of all your custom classes in one jar.

Sitemap Modifications

You need to do two things: in the `map:generators` section, add an element for your class:

```
<map:generator name="helloWorld" src="HelloWorldGenerator"/>
```

Then add a pipeline to `sitemap.xml` which uses it:

```
...
  <map:match pattern="heyThere.xml">
    <map:generate type="helloWorld"/>
    <map:serialize type="xml"/>
  </map:match>
...
```

And finally, our creation should be available at

`http://localhost:8080/cocoon/heyThere.xml`

Depending on your exact setup, you may need to restart Tomcat (or whatever your servlet container is) to get there.



Notice that the `<?xml version="1.0" encoding="UTF-8"?>` declaration was added for us by the xml serializer at the beginning. If you need to modify this, the generator is not the appropriate place. The default encoding of UTF-8 could be overridden with `iso-8859-1` for example by specifying an

Introducing Cocoon

<encoding>iso-8859-1</encoding> child parameter inside the declaration for the xml serializer in your sitemap.

A Less Trivial Example

Moving on to a less trivial example, we will take some information out of the Request, and construct a slightly more involved document. This time, our goal will be the following document:

```
<doc>
<uri>...</uri>
<params>
  <param value="...">...</param>
  ...
</params>
<date>..</date>
</doc>
```

The values of course will be filled in from the request, and will depend on choices we make later.

```
import org.apache.cocoon.generation.AbstractGenerator;
import org.xml.sax.helpers.AttributesImpl;
import org.xml.sax.SAXException;

// for the setup() method
import org.apache.cocoon.environment.SourceResolver;
import java.util.Map;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import java.io.IOException;

// used to deal with the request parameters.
import org.apache.cocoon.environment.ObjectModelHelper;
import org.apache.cocoon.environment.Request;
import java.util.Enumeration;

import java.util.Date;

public class RequestExampleGenerator extends AbstractGenerator
{

    // Will be initialized in the setup() method and used in generate()
    Request request = null;
    Enumeration paramNames = null;
    String uri = null;

    // We will use attributes this time.
    AttributesImpl myAttr = new AttributesImpl();
    AttributesImpl emptyAttr = new AttributesImpl();

    public void setup(SourceResolver resolver, Map objectModel,
        String src, Parameters par)
        throws ProcessingException, SAXException, IOException
    {
        super.setup(resolver, objectModel, src, par);
        request = ObjectModelHelper.getRequest(objectModel);
        paramNames = request.getParameterNames();
        uri = request.getRequestURI();
    }

    /**
```

Introducing Cocoon

```
* Implement the generate() method from AbstractGenerator.
*/

public void generate() throws SAXException
{
    contentHandler.startDocument();

    contentHandler.startElement("", "doc", "doc", emptyAttr);

    // <uri> and all following elements will be nested inside the doc element
    contentHandler.startElement("", "uri", "uri", emptyAttr);

    contentHandler.characters(uri.toCharArray(),0,uri.length());

    contentHandler.endElement("", "uri", "uri");

    contentHandler.startElement("", "params", "params", emptyAttr);

    while (paramNames.hasMoreElements())
    {
        // Get the name of this request parameter.
        String param = (String)paramNames.nextElement();
        String paramValue = request.getParameter(param);

        // Since we've chosen to reuse one AttributesImpl instance,
        // we need to call its clear() method before each use. We
        // use the request.getParameter() method to look up the value
        // associated with the current request parameter.
        myAttr.clear();
        myAttr.addAttribute("", "value", "value", "", paramValue);

        // Each <param> will be nested inside the containing <params> element.
        contentHandler.startElement("", "param", "param", myAttr);
        contentHandler.characters(param.toCharArray(),0,param.length());
        contentHandler.endElement("", "param", "param");
    }

    contentHandler.endElement("", "params", "params");

    contentHandler.startElement("", "date", "date", emptyAttr);

    String dateString = (new Date()).toString();
    contentHandler.characters(dateString.toCharArray(),0,dateString.length());

    contentHandler.endElement("", "date", "date");
    contentHandler.endElement("", "doc", "doc");
    contentHandler.endDocument();
}

public void recycle() {
    super.recycle();
    this.request = null;
    this.paramNames = null;
    this.parNames = null;
    this.uri = null;
}
}
```

Compile and Test

Save this code as `RequestExampleGenerator.java` and compile as before. You will need to add both `avalon-framework.jar` and `avalon-excalibur.jar` to your classpath this time. Besides finding the exact name of the jar as described above, you may now also have to ensure

that you have the version of excalibur targeted to your jvm version – there is currently a version for JDK 1.4 and one for 1.2/1.3

For your sitemap, you will need to add a definition for this generator like `<map:generator name="requestExample" src="RequestExampleGenerator" />` and you will need a sitemap pipeline like:

```
<map:match pattern="howYouDoin.xml">
  <map:generate type="requestExample"/>
  <map:serialize type="xml"/>
</map:match>
```

At this point, you should be able to access the example at `http://localhost:8080/cocoon/howYouDoin.xml?anyParam=OK&more=bet`

New Concepts Lifecycle

First, notice that we now override the `setup(...)` and `recycle()` methods defined in `AbstractGenerator`. The `ComponentManager` that handles the lifecycle of all components in Cocoon, calls `setup(...)` before each new call to `generate()` to give the `Generator` information about the current request and its environment, and calls `recycle()` when it is done to enable it to clean up resources as appropriate. Our example uses only the `objectModel` which abstracts the `Request`, `Response`, and `Context`. We get a reference to the `Request` wrapper, and obtain an `Enumeration` of all the `GET/POST` parameters available.

The `src` and `SourceResolver` are provided to enable us to look up and use whatever source is specified in the pipeline setup. Had we specified `<map:generate type="helloWorld" src="someSourceString"/>` we would have used the `SourceResolver` to work with "someSourceString", whether it be a file, or url, etc.

We are also given a `Parameters` reference which we would use to obtain any parameter names and values which are children elements of our `map:generate` element in the pipeline.



It may be good practice to abstract the source of your parameters so that they do not have to come from the Request object. For instance, the following code would allow us to abstract the origin of two parameters, param1 and param2:

```
In RequestExampleGenerator.java,
...
String param1 = null;
String param2 = null;
...
public void setup(SourceResolver resolver, Map objectModel,
                 String src, Parameters par)
    throws ProcessingException, SAXException, IOException
{
    ...
    param1 = par.getParameter("param1");
    param2 = par.getParameter("param2");
}

and in sitemap.xmap,
...

```

Introducing Cocoon

```
<map:match pattern="abstractedParameters.xml" />
  <map:act type="request">
    <map:parameter name="parameters" value="true" />
    <map:generate type="requestExample">
      <parameter name="param1" value="{visibleName1}" />
      <parameter name="param2" value="{visibleName2}" />
    </map:generate>
  </map:act>
</map:match>
...
```

As you can see, we have also hidden the internal name from the outside world who will use `?visibleName1=foo&visibleName2=bar`

Nested Elements

In this example, nested elements are created simply by nesting complete `startElement()/endElement` pairs within each other. If we had a logic failure in our code and sent non-wellformed xml events down the pipeline, nothing in our process would complain (try it!). Of course, any transformers later in the pipeline would behave in an unpredictable manner.

Attributes

Finally, we've introduced the use of attributes. We chose to employ one `attributesImpl`, clearing it before each element. Multiple attributes for an element would simply be added by repeated calls to `addAttribute`.

A Lesson

Before moving on, it is worth noting that after all this work, there is already a generator provided with Cocoon which does much of what we have accomplished here –

`org.apache.cocoon.generation.RequestGenerator` which in the default configuration is probably available at

`http://localhost:8080/cocoon/request`

Moving On

From here, we will move on to cover handling ugly pseudo-xml (like real world html) with CDATA blocks, employing some of the Avalon lifecycle method callbacks (Composable/Disposable), Database access, and Caching.

The Employee SQL Example Reworked

In the samples included with Cocoon, there is an example of a SQL query using XSP and ESQL. We will recreate part of that example below using the same HSQL database, which should be automatically configured and populated with data in the default build. If you find that you do not have that database set up, see the ESQL XSP sample for instructions on setting the datasource up. Do note that this specific task is handled in the ESQL XSP example in just a few lines of code. If your task is really this simple, there may be no need to create your own generator.

```
import org.apache.cocoon.generation.ComposerGenerator;
import org.apache.avalon.framework.component.ComponentManager;
```

Introducing Cocoon

```
import org.apache.avalon.framework.component.ComponentException;
import org.apache.avalon.framework.component.ComponentSelector;
import org.apache.avalon.excalibur.datasource.DataSourceComponent;
import org.apache.cocoon.environment.SourceResolver;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.environment.ObjectModelHelper;
import org.apache.cocoon.environment.Request;
import org.apache.cocoon.caching.Cacheable;
import org.apache.cocoon.caching.CacheValidity;
import org.apache.cocoon.ProcessingException;
import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.AttributesImpl;

import java.sql.*;
import java.util.Map;
import java.util.Date;
import org.apache.avalon.framework.activity.Disposable;

public class EmployeeGeneratorExample extends ComposerGenerator
    implements Cacheable, Disposable
{

    public void dispose() {
        super.dispose();
        manager.release(datasource);
        datasource = null;
    }

    public void recycle() {
        myAttr.clear();
        super.recycle();
    }

    public void setup(SourceResolver resolver, Map objectModel,
        String src, Parameters par) {
        // Not needed for this example, but you would get request
        // and/or sitemap parameters here.
    }

    public void compose(ComponentManager manager)
    throws ComponentException{
        super.compose(manager);
        ComponentSelector selector = (ComponentSelector)
            manager.lookup(DataSourceComponent.ROLE + "Selector");
        this.datasource = (DataSourceComponent) selector.select("personnel");
    }

    public void generate()
    throws SAXException, ProcessingException {
        try {

            Connection conn = this.datasource.getConnection();
            Statement stmt = conn.createStatement();

            ResultSet res = stmt.executeQuery(EMPLOYEE_QUERY);

            //open the SAX event stream
            contentHandler.startDocument();
            myAttr.addAttribute("", "date", "date", "",
                (new Date()).toString());
            //open root element
            contentHandler.startElement("", "content",
                "content", myAttr);
        }
    }
}
```

Introducing Cocoon

```
        String currentDept = "";
        boolean isFirstRow = true;
        boolean moreRowsExist = res.next() ? true : false;

        while (moreRowsExist) {
            String thisDept = attrFromDB(res, "name");
            if (!thisDept.equals(currentDept)) {
                newDept(res, thisDept, isFirstRow);
                currentDept = thisDept;
            }
            addEmployee(res, attrFromDB(res, "id"),
                attrFromDB(res, "empName"));
            isFirstRow = false;

            if (!res.next()) {
                endDept();
                moreRowsExist = false;
            }
        }

        //close root element
        contentHandler.endElement("", "content", "content");
        //close the SAX event stream
        contentHandler.endDocument();

        res.close();
        stmt.close();
        conn.close();
    } catch (SQLException e) {
        throw new ProcessingException(e);
    }
}

public long generateKey()
{
    // Default non-caching behaviour. We will implement this later.
    return 0;
}

public CacheValidity generateValidity()
{
    // Default non-caching behaviour. We will implement this later.
    return null;
}

private DataSourceComponent datasource;
private AttributesImpl myAttr = new AttributesImpl();

private String EMPLOYEE_QUERY =
"SELECT department.name, employee.id, employee.name as empName " +
"FROM department, employee " +
"WHERE department.id = employee.department_id ORDER BY department.name";

private void endDept() throws SAXException {
    contentHandler.endElement("", "dept", "dept");
}

private void newDept(ResultSet res, String dept, boolean isFirstRow)
throws SAXException {
    if (!isFirstRow) {
        endDept();
    }
}
```

Introducing Cocoon

```
myAttr.clear();
myAttr.addAttribute("", "name", "name", "", dept);
contentHandler.startElement("", "dept", "dept", myAttr);
}

private void addEmployee(ResultSet res, String id, String name)
    throws SAXException {
    myAttr.clear();
    myAttr.addAttribute("", "id", "id", "", id);
    contentHandler.startElement("", "employee", "employee", myAttr);
    contentHandler.characters(name.toCharArray(), 0, name.length());
    contentHandler.endElement("", "employee", "employee");
}

private String attrFromDB(ResultSet res, String column)
    throws SQLException {
    String value = res.getString(column);
    return (res.wasNull())?"":value;
}
}
```

Compile and Test

To compile this, you will now need the following on your classpath: `avalon-excalibur.jar`, `avalon-framework.jar`, `cocoon.jar`, `xml-apis.jar` (using whatever names they have in your distribution). When you compile this, you may receive some deprecation warnings. Do not worry about them – we will discuss that later.

To test it, copy it over to your `WEB-INF\classes\` directory as before and add something like the following to your `sitemap.xmap` ...

```
...
<map:generator name="employee" src="EmployeeGeneratorExample"/>
...
<map:match pattern="employee.xml">
    <map:generate type="employee"/>
    <map:serialize type="xml"/>
</map:match>
...
```

New Concepts Composable and Disposable

We've implemented the Avalon lifecycle interfaces `Composable` and `Disposable`. When Cocoon starts up (which happens when the servlet container starts up) the `ComponentManager` will call `compose(ComponentManager m)` for our component as it works its way through all the components declared in the sitemap. The handle to `ComponentManager` is used to look up any other Avalon components that we need. Lookups happen in an abstracted way using a `ROLE` which enables us to change out implementations of each component without affecting previously written code. Our generator's `ROLE` by the way was defined in the `Generator` interface.

Similarly, when this instance of our generator is disposed of by the container, it will call the `dispose()` method to allow us to clean up any resources we held on to between invocations. Note that components can be pooled by the container. If we thought that our employee generator was going to see a lot of traffic, we might change its definition at the top of `sitemap.xmap` to include

Introducing Cocoon

attributes like `pool-grow="2" pool-max="16" pool-min="2"` so that multiple overlapping requests could be serviced without a log jam.

Datasource

We look up our HSQL database here by its name given in `cocoon.xconf`. If we had multiple datasources (say a backup development database and a live one), we could determine which one to use based on a simple configuration parameter in `sitemap.xmap`. We could get at configuration parameters using the Avalon interface `Configurable`.



Notice that we wait until `generate()` to request our connection from the pool – as we should. The problem is that we lose the benefit of using prepared statements since they would be destroyed when we returned the instance to the pool. At present, the implementation of `org.apache.avalon.excalibur.datasource.DataSourceComponent` does not support the pooling of statements.

Caching



FIXME: This is still coming.

Introduce new code to implement Caching, discuss basic logic, and deprecation/move to Avalon. I could use some help here from Carsten, or someone who can quickly give an overview of the changes and plan.

Copyright © 1999–2002 The Apache Software Foundation. All Rights Reserved.