Adding XML Capabilities with Cocoon

Author: Stefano Mazzocchi (stefano@apache.org)

Notice

This document was written as an handout for the author's talk at ApacheCon 2000. This document is written with the collaboration of a number of people from the Cocoon Project and reuses part of the Cocoon documentation reformatted here for a complete presentation. The author wishes to thank all the people that helped in the creation of such documentation as well as typo and language mistakes corrections.

Abstract

This paper introduces the notion of XML publishing to the reader and shows how these new paradigms can be applied to existing web solutions by using the Cocoon publishing framework. Basic knowledge on XML technologies is assumed but nor necessarily required, since I aim to show the power of XML publishing to those that are used to HTML-driven models.

1. Introduction

Cocoon is a publishing framework written using the Java language and based on the Servlet API model. For this reason, it can be installed on any web server that is able to execute Java servlets.

A publishing framework is a software that aims to simplify your work in the creation, design and maintenance of informative systems based on web technology. Such systems are usually composed by server software (web server + functional server modules), by data repositories (file systems, databases, directory servers) and by solution-specific software that glues everything together.

In such systems, the custom logic handles what is normally called the web application, which most of the times turns out to be very complex and rarely reusable.

Cocoon introduces design patterns, evolutionary guidelines and software that allows one to increase the degree of engineering spent in the creation of web services, focusing on human resource management rather then technological details, which are left in the middleware level.

In fact, Cocoon, being a servlet, inherits all the technological benefits that are provided by good servlet platforms while focuses on proposing design patterns for web publishing as well as enforcing them with actual implementations, tools and solutions.

1.1 The need for design patterns

During years of web development and Java programming, I found the notion of design patterns one of the most intriguing and exiting. Patterns represent sort of algorithms for design, collections of rules and ideas that, expressed by humans for humans, allow better understanding and the establishment of common bases.

In the Cocoon Project, we tried to use design patterns for all possible aspects of developing, both internal (for the creation of the program) and external (to show others

how to do web engineering).

Design patterns condense knowledge, mistakes, solutions in a single notion, thus allowing programming to become more solid and solutions more reusable.

While a complete description of the design patterns used in the Cocoon Project is out of the scope of this paper, it is important to note how clear and well established design patterns will be taken for granted, while new design patterns, elaborated to solve new problems in the XML world and proposed either by us or by the XML working groups at W3C, will be discussed and explored.

But let's start from the beginning...

2. The XML model

2.1 What is this XML?

XML (eXtended Markup Language) is a subset of SGML (Standard Generalized Mark-up Language). SGML is the grandparent of all markup languages and a 15-year-old ISO standard for creating languages. You can think of XML as a lighter version of SGML.

The first thing you must understand is that XML is *not* a language (like HTML), but a syntax, in the same way that ASCII defines a standard way to map characters to bytes rather than to character strings.

XML is usually referred to as *portable data* in the sense that its parsing is *application independent*. The same XML parser can read every possible XML document: one describing your bank account, another describing your favorite Italian meal, etc. This is, as you all know, impossible with other text-based or binary file formats. A near-equivalent in the old days was CSV (comma separated values) files, which used a very simple syntax (one record per line, a comma separating fields, and the values in the first row naming the columns). XML, unlike CSV, is much more flexible and structured, even though it's much simpler than SGML.

A particular XML language is defined by its Document Type Definition (DTD). DTDs are described in the XML specification. They describe the syntax of a language implemented in XML. An XML document may be validated against a DTD (if present). If the validation is successful the document is said to be *valid XML based on the particular DTD*. If a DTD is not present and the parser does not encounter syntax errors parsing the file, the XML document is said to be *well-formed* . If errors are found, the document is not XML compliant.

So, any valid XML document is *well-formed* and an XML document *valid* for one particular DTD may not necessarily be valid for another DTD. For example, HTML is not an XML language because some tags such as `<br>` are not XML compliant. In XHTML, an XML compliant reformulation of HTML, `<br>`, for example, is replaced with `<br/>`. While HTML pages are not always well-formed XML documents (some pages might be), XHTML pages are always well-formed and valid XML documents if they match the XHTML DTD.

So much for the technical differences, but why was HTML not good enough? Let's consider an example.

2.2 XML shows its power

Consider how the need for XML came about:

- Everyone starts publishing HTML documents on the web.
- Search engines spring up across the net to help find documents.
- Search engines have a difficult time searching specific pieces of a document since HTML was designed to represent hierarchically how data should be presented, but not what data is being presented.
- Web applications spring up across the net to provide information and *services*.

These services could be web pages that serve up important information about an organization or the structure of the organization. It could be weather information or travel advisories. It could be contact information for people. Stock quotes. It could a book on how to grow the perfect Tomato.

So now we have all this information. Tons of it. Great! Now go and search all those web pages for specific content, like Author or Subject. Find me all abstracts of documents published on the subject of *Big Tomatoes*, since I only want to view abstracts to find the document best for me. An HTML page is not designed for this. It was designed for *how to present* the data.

When I look at a web page I might see that an author chose to make every heading bold with `<font size="+1">`. Yet if I look at another page I might notice that every heading was marked up with `<H1>`. Yet another page may use tables and table headers to format the data. Find me every document that has the word *potato* in the first heading.

Suppose I have a web application that serves up weather information for different parts of the country. Let's say you live in Boston, MA and only want the local weather. Your boss asks you to write an application that goes out and grabs the two-to-three sentence weather summary from my application and display it on your intranet's homepage.

You take a quick jaunt over to my weather application and notice that the summary is in what looks like the second paragraph of the page. So you take a quick peek at the HTML source that my weather application returns. You suddenly realize that it's all on one line and is buried deep within tables.

So you start writing your little application to parse my HTML code to retrieve only the information you were looking for. You pat yourself on the back when—4 hours later—you finally get the information you were looking for. Your code looks for the 2nd `TABLE`, the 6th `TR`, and then the 2nd `TD`. Phew. Your application, which really only wants to retrieve weather data, is forced to parse display markup to get it.

You run over to your boss and demonstrate the application you are so proud of writing. Lo and behold it doesn't work. What happened? The good old page author decided to change the layout and move the weather summary to `TABLE` 1, `TR` 1, `TD` 1. Your application breaks because it is tied to the presentation of the data and not to the data itself. Not very effective, since now your app will break every time the page author drinks too much coffee.

Then you notice something on the page that interests you. The site is automatically generated from XML and you see a link that indicates there is an XML DTD for weather information. And another link that indicates the availability of an XML stream for weather information. Yikes, would you look at that:

```
<weather-information>
 <location>
  <city>Boston</city>
  <state>MA</state>
 </location>
 <summary>
  Beautiful and Sunny, lows 50, highs 65, with the
```

```
  chance of a blizzard and gail force winds.
 </summary>
</weather-information>
```

So you simply download Cocoon and quickly write an XSL stylesheet that looks like the following:

```
<xsl:stylesheet>
 <xsl:template match="/">
  ... presentation info here ...
 </xsl:template>
 <xsl:tempate
   match="weather-information[location/city = 'Boston']">
  <xsl:apply-templates select="summary"/>
 </xsl:template>
</xsl:stylesheet>
```

And your boss gives you your job back!

2.3 The HTML Model

As the above example explains, HTML is a language for describing graphics, behavior, and hyperlinks on web pages. HTML is *not* able to *contextualize* (i.e. *give meaning to some text*). For example, if you look for the *title* of a page, a nice HTML tag gives you that, but if you look for the author or version or something more specific like the author's mail address—even if this information is present in the text—you don't have a way to *isolate* it (contextualize it) from the surrounding information.

In HTML like this

```
<html>
 <head>
  <title>This is my article</title>
 </head>
 <body>
  <h1 align="center">This is my article</h1>
  <h3 align="center">
   by <a href="mailto:stefano@apache.org">
       Stefano Mazzocchi
      </a>
  </h3>
  ...
 </body>
</html>
```

you don't have a guaranteed way to extract the mail address. Whereas in the following XML document

```
<?xml version="1.0"?>
<page>
 <title>This is my article</title>
 <author>
```

```
  <name>Stefano Mazzocchi</name>
  <email>stefano@apache.org</email>
 </author>
 ...
</page>
```

it's trivial and algorithmically certain.

We don't imagine XML overtaking HTML in web publishing since HTML is great for small needs. HTML was born as an SGML-based DTD for scientists' homepages, i.e. to parallelize and simplify the deployment and management of personal information. HTML was *not* designed for the publishing and processing of large quantities of data and complex dynamic information systems.

2.4 The XSL Language

As you can see, XML alone is useless without some defined semantics: even if an application is able to parse a document, it must be able to *understand* what the markup means. This is why XML-only browsers are meaningless and not more useful than text editors from a usability point of view.

This is one of the reasons why XSL (the eXtensible Stylesheet Language) was proposed and designed. XSL is divided into two parts: transformation (XSLT) and formatting objects (sometimes referred to as FO, XSL:FO, or simply XSL). Both are XML DTDs that define a particular XML syntax, so every XSL or XSLT document is a well-formed XML document.

2.5 XSL Transformations (XSLT)

XSLT is a language for transforming one well-formed XML document into something else (which may *not* necessarily be another XML document, although it most often will be). This means that you can use it to go from one DTD to another in a procedural way that is defined inside your XSLT document. XSLT can be used in ways its name might not imply: a transformation may be applied to a document to generate a *graphical description* of its content. This is called *styling*, but, as you can imagine, it is just one of the possible uses of transformation technology.

Back in the earlier example, the HTML file may have been generated from an XML file using *another* XML file as a transformation sheet (which in this case is a stylesheet). The data is all there: we just have to tell the transformer how to come up with the HTML document once all the data is parsed.

Usually, transformation sheets work from one DTD to another and in this way form a chain: transformA goes from DTD1 to DTD2 and transformB from DTD2 to DTD3 or graphically

```
DTD1 ---(transformA)--> DTD2 ---(transformB)---> DTD3
```

We'll call DTD1 the *original DTD*, DTD2 some *intermediate DTD*, DTD3 the *final DTD*. A transformation can always be created to go directly from DTD1 to DTD3, but this might be more complicated and less human-readable/manageable.

2.6 XSL Formatting Objects (XSL:FO)

XSLFO is a language (an XML DTD) for describing 2D layout of text in both printed and digital media. I will not concentrate on the graphical abilities that formatting objects give

you, but rather on the fact that it is mostly used as a *final DTD*, meaning that a transformation is used to generate a formatting object description of a document starting from a general XML file.

An XSLFO document for our ongoing example would be

```
<?xml version="1.0"?>
<fo:root xmlns:fo="http://www.w3.org/XSL/Format/1.0">
 ...
 <fo:flow font-size="14pt" line-height="14pt">
  <fo:block
      text-align="centered"
      font-size="24pt"
      line-height="28pt">This is my article</fo:block>
  <fo:block
      space-before.optimum="12pt"
      text-align="centered">by Stefano Mazzocchi</fo:block>
 </fo:flow>
</fo:root>
```

which tells the formatting object formatter (the rendering engine), how to *draw* and place the text on screen or on paper.

XSL formatting objects and transformations are being specified by the same working group and have a lot of synergy, even though the XSLT specification also includes ways to create HTML and text from XML files.

3. Installing Cocoon

3.1 System Requirements

Cocoon requires the following systems to be already installed in your system:

- **Java Virtual Machine** A Java 1.1 or greater compatible virtual machine must be present for both command line and servlet type usage of Cocoon. Note that all servlet engines require a JVM to run so if you are already using servlets you already have one installed.
- **Servlet Engine** A Servlet 2.x compliant servlet engine must be present in order to support servlet operation and dynamic request handling. Note that this requirement is optional for command line operation.

3.2 Required Components

Cocoon is a publishing framework and was designed to be highly modular to allow users to choose their preferred implementation for the required component and to allow better and faster parallel development.

Here is a list of supported components with their minimum version required to operate with this version of Cocoon:

XML Parsers

Name	**Version**	**Location**

Apache Xerces	1.0.1 (java edition)	xml.apache.org
Sun ProjectX	TR2	java.sun.com

XSLT Processor

Name	Minimum Version	Location
Apache Xalan	0.19.0	xml.apache.org
James Clark's XT	19991102	www.jclark.com

Other Packages

Name	Minimum Version	Location
Apache FOP	0.12.0	xml.apache.org
FESI EcmaScript Engine	1.2.1	home.worldcom.ch/jmlugrin/fesi
GNU Regexp	1.0.8	www.cacas.org/java/gnu/regexp
JNDI API	1.2.1	java.sun.com

Note: *Cocoon is strictly dependent on some printing classes contained into Xerces so, even if you use another parser, you should still keep Xerces visible to Cocoon.*

Being an Apache project, Cocoon focuses on Apache technologies but we are dedicated to support all compatible XML/XSL technologies and will welcome contributions to add support for other components not currently supported.

So this is your shopping list for components for complete operation:

- Apache Xerces (required for the formatting classes *org.apache.xml.serialize*)
- Your favorite XML parser
- Your favorite XSLT processor
- Apache FOP (optional, unless you want PDF rendering)
- GNU Regexp (optional, unless you use *ProducerFromMap*)
- FESI (optional, unless you use *DCP*)
- JNDI (optional, unless you use the *LDAP processor*)

All right. Now that you have downloaded all the components you need, go on and jump to the installation instructions for your servlet engine.

3.3 Installation

Being Cocoon a servlet, you should be able to install it on every compliant servlet engine by associating the "org.apache.cocoon.Cocoon" servlet with the requests you want it to handle. In order to do this, there is no standard way, so we try to provide detailed information for the most used servlet systems.

3.3.1 General considerations

There are some general considerations that apply to all systems.

Since there is no portable way, in a Java platform, to tell how much memory an object is using, the memory cache works in a rather cumbersome manner: you set up a lower limit that the cache must always leave free for the JVM operation. This means, that if the memory limit is 200Kb, Cocoon uses all your JVM heap size to store pages in memory and makes sure that 200Kb are available for other operations.

This does not impact the JVM operation, if enough memory is left for the normal operation. You must be aware of the fact that leaving Cocoon with too little memory for operation does impact peformance since the JVM garbage collector has to do more work to keep up with execution and memory cleanup. Sometimes, you may even end up having OutOfMemoryExceptions if your limit is lower than the memory required for the operation.

A good way to control your memory is to setup your JVM with a fixed heap limit and to give it enough memory to start. This is done by using command line parameters for your java interpreter such as:

- Startup heap size:

  java 1.1 - `-ms[size]`
  java 1.2 - `-Xms[size]`

- Maximum heap size:

  java 1.1 - `-mx[size]`
  java 1.2 - `-Xmx[size]`

A well balanced system should have something like 8Mb start heap, 2Mb Cocoon memory limit and 64Mb or greater max heap, but these depend heavily on your system load/configuration to be generally meaningful.

To change the cocoon object store memory limit open you should change the following property in the `cocoon.properties` file.

```
store.memory = 200000
```

Another important part of Cocoon is the page compiler used inside the XSP processor which store the generated/compiled pages on file system. The default directory is `./repository` which is usually relative to the web server or servlet engine working directory.

If you experience troubles (such as not having reading/writing permissions) or you want to locate this directory somewhere else, you have to change the

```
processor.xsp.repository = ./repository
```

property in the cocoon configuration file.

Warning: Since this directory may contain security sensible information, make sure you deny access (even read-only) to untrusted users.

3.3.2 Installing Cocoon on Apache JServ

Apache JServ has one configuration file for the whole engine (normally called `jserv.properties`) and one for each servlet zone. Please, refer to the Apache JServ documentation for more information on this.

First thing to do is to make sure that Cocoon and all the needed components (as explained in the previous section) are visible. This implies adding this to the servlet engine classpath by adding a line like this in your `jserv.properties` file for each jar package you have to install (after substituting *[path-to-jar]* with the path to the jar file and *[jar-name]* with the package file name).

```
wrapper.classpath=[path-to-jar]/[jar-name].jar
```

Here is an example:

```
wrapper.classpath=/usr/local/java/lib/cocoon.jar
wrapper.classpath=/usr/local/java/lib/xerces.jar
wrapper.classpath=/usr/local/java/lib/xalan.jar
wrapper.classpath=/usr/local/java/lib/fop.jar
wrapper.classpath=/usr/local/java/lib/regexp.jar
wrapper.classpath=/usr/local/java/lib/jndi.jar
```

Note: *from this version of Cocoon the `Cocoon.jar` package should be added to the servlet engine classpath as any other required package (as shown above).*

At this point, you must set the Cocoon configurations. To do this, you must choose the servlet zone(s) where you want Cocoon to reside (Note that Cocoon can safely reside on different servlet zones with different configuration files). If you don't know what a servlet zone is, you probably want to open the `zone.properties` file that represents the default servlet zone.

To configure Cocoon, you must pass the `cocoon.properties` file location to the servlet by adding the following to the `zone.properties` file (or each servlet zone file you want Cocoon to reside):

```
servlet.org.apache.cocoon.Cocoon.initArgs=properties=[path-to-cocoon]/bin/cocoon.properties
```

Note that you should not need to change anything from the template properties file found in the distribution (under `/conf/`), but you must edit it for more complex operation. Please refer directly to the file that contains breaf indications and comments on the different configurations, but you don't need to care about that at this point.

Now your cocoon servlet is properly configured, but you should tell Apache to direct any call to an XML file (or any other file you want Cocoon to process) to the Cocoon servlet. To do this, you should add the following line to your `mod_jserv.conf` or `jserv.conf` file:

```
ApJServAction .xml /servlet/org.apache.cocoon.Cocoon
```

where *.xml* is the file extention you want to map to the servlet and */servlet/* is the mount point of your servlet zone (and the above is the standard name for servlet mapping for Apache JServ).

At this point, you should check if your system matches the global considerations about Cocoon properties. Usually, you might want to give the installation a try as it is and then read again that section if something goes wrong. Most installations don't need any changes to be operational.

Everything should be configured fine. Restart both Apache and Apache JServ and try accessing the samples contained in the distribution to see Cocoon in action or the `/Cocoon.xml` page for Cocoon internal status.

Note: *You may want Cocoon to handle two different extentions with different configurations. To do this, you must set the two different configurations in two different servlet zones, then associate the extentions like this:*

```
ApJServAction .xml1 /zone1/org.apache.cocoon.Cocoon
```

```
ApJServAction .xml2 /zone2/org.apache.cocoon.Cocoon
```

where *.xml1* is handled by the Cocoon residing on *zone1* and *.xml2* is handled by the Cocoon residing on *zone2* which have different configurations and thus different behavior.

3.3.3 Installing Cocoon on Apache Tomcat

Tomcat has two basic methods of locating Java classes for the runtime environment. The first is the overall classpath that Tomcat uses to run, and this is the classpath provided to Java classes that use constructs such as `Class.forName().newInstance()`. The second classpath is associated with a specific context, and is somewhat analagous to the servlet zones used in Apache JServ (see section above).

Because the Cocoon framework utilizes `Class.forName()` and other dynamic instance handling techniques, the Cocoon classes need to have its classpath aware of the component classes used within the framework. To do this, take all the required components (see above) and put them in your *<Tomcat-Root >/lib* directory. This is the standard location for Tomcat core libraries. To ensure that Tomcat will use these, you need to edit the Tomcat startup file.

On Windows, this is *<Tomcat-Root >/tomcat.bat* and on Unix, this is */tomcat.sh*. In this file you must add all the component jar files to Tomcat's classpath.

Note: *from this version of Cocoon the `Cocoon.jar` package should be added to the servlet engine classpath as any other required package (as shown above).*

Next you need to tell your context where Cocoon can find it's properties file, as well as to map Cocoon to XML document requests. Make sure you have a *web.xml* file in your context's *WEB-INF* directory (look in `src/WEB-INF/` to find a template *web.xml*). This file specifies servlet mappings and initial arguments to servlets and looks like this:

```
<servlet>
 <servlet-name>org.apache.cocoon.Cocoon</servlet-name>
 <servlet-class>org.apache.cocoon.Cocoon</servlet-class>
 <init-param>
  <param-name>properties</param-name>
  <param-value>
    [path-to-cocoon.properties]/cocoon.properties
  </param-value>
 </init-param>
</servlet>

<servlet-mapping>
 <servlet-name>org.apache.cocoon.Cocoon</servlet-name>
 <url-pattern>*.xml</url-pattern>
</servlet-mapping>
```

Make sure you replaced the path to the Cocoon.properties file with the actual location of that file on your system. Note that you should not need to change anything from the template properties file found in the distribution, but you must edit it for more complex operation. Please refer directly to the file that contains brief indications and comments on the different configurations, but you don't need to care about that at this point.

At this point, you should check if your system matches the global considerations about

Cocoon properties. Usually, you might want to give the installation a try as it is and then read again that section if something goes wrong. Most installations don't need any changes to be operational.

Everything should be configured fine. Restart both Apache and Tomcat and try accessing the samples contained in the distribution to see Cocoon in action or the `/Cocoon.xml` page for Cocoon internal status.

Note: *Tomcat 3.0 has a bug that prevents Cocoon operation. In order to make Cocoon work under Tomcat you need to download a newer version or, if none is yet avaiable, build it from the latest source code found in the `jakarta-tomcat` CVS module under jakarta.apache.org. We apologize for this, but it's not something we can control or work around.*

4. Hello World

Ok, now that everything is setup, we go on and show it's simple functionality. Here is a well-formed XML file that uses a custom and simple DTD

```
<?xml version="1.0"?>
<page>
 <title>Hello World!</title>
 <content>
  <paragraph>This is my first Cocoon page!</paragraph>
 </content>
</page>
```

Even if this page mimics HTML (in a sense, HTML was born as a simple DTD for homepages), it is helpful to note that there is no style information and all the styling and graphic part is missing. Where do I put the title? How do I format the paragraph? How do I separated the content from the other elements? All these questions do not have answers because in this context they don't need one: this file should be created and maintained by people that don't need to be aware of how this content if further processed to become a served web document.

On the other hand, we need to indicate how the presentation questions will be answered. To do this, we must indicate a *stylesheet* that is able to indicate how to interpret the elements found in this document. Thus, we follow a W3C recommendation and add the XML processing instruction to map a stylesheet to a document:

```
<xml-stylesheet href"hello.xsl" type="text/xsl">
```

Now that our content layer is done, we need to create a stylesheet to convert it to a format readable by our web clients. Since most available web clients use HTML as their *lingua franca*, we'll write a stylesheet to convert our XML in HTML (More precisely, we convert to XHTML which is the XML form of HTML, but we don't need to be that precise at this point).

Every valid stylesheet must start with the root element *stylesheet* and define its own namespace accordingly to the W3C directions. So the skeleton of your stylesheet is:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

Once the skeleton is done, you must include your `template` elements, which are the basic unit of operation for the XSLT language. Each template is matched against the occurrence of some elements in the original document and the element is replaced with the children elements, if they belong to other namespaces, or, if they belong to the XSLT namespace, they are further processed in a recursive way.

Let's make an example: in our `HelloWorld.xml` document `page` is the root element. This must be transformed in all those tags that identify a good HTML page. Your template becomes:

```
<xsl:template match="page">
 <html>
  <head>
   <title><xsl:value-of select="title"/></title>
  </head>
  <body bgcolor="#ffffff">
   <xsl:apply-templates/>
  </body>
 </html>
</xsl:template>
```

were some elements belong to the standard namespace (which we associate to HTML) and some others to the *xsl:* namespace. Here we find two of those XSLT elements: `value-of` and `apply-templates`. While the first *searches* the `page` element direct children for the `title` element and replace itself with the content of the retrieved element, the second indicates to the processor that should continue the processing of the other templates described in the stylesheets from that point.

Other possible templates are:

```
<xsl:template match="title">
 <h1 align="center">
  <xsl:apply-templates/>
 </h1>
</xsl:template>

<xsl:template match="paragraph">
 <p align="center">
  <i><xsl:apply-templates/></i>
 </p>
</xsl:template>
```

After the XSLT processing, the original document is transformed to

```
<html>
 <head>
  <title>Hello</title>
 </head>
 <body bgcolor="#ffffff">
  <h1 align="center">Hello</h1>
  <p align="center">
   <i>This is my first Cocoon page!</i>
  </p>
```

```
 </body>
</html>
```

4.1 Browser Dependent Styling

When a document is processed by an XSLT processor, its output is exactly the same for every browser that requested the page. Sometimes it's very helpful to be able to discriminate the client capabilities and transform content layer into different views/formats. This is extremely useful when we want to serve content do very different types of clients (fat clients like desktop workstations and thin clients like wireless PDAs) but we want to use the same informative source and create the smallest possible impact on the site management costs.

Cocoon is able to discriminate between browsers, allowing the different stylesheets to be applied. This is done by indicating in the stylesheet linking PI the *media* type, for example, continuing with the HelloWorld.xml document, these PIs

```
<?xml version="1.0"?>
<?xml-stylesheet href="hello.xsl" type="text/xsl"?>
<?xml-stylesheet href="hello-text.xsl" type="text/xsl" media="lynx"?>

...
```

would tell Cocoon to apply the `hello.text.xsl` stylesheet if the Lynx browser is requesting the page. This powerful feature allows you to design your content independently and to choose its presentation depending on the capabilities of the browser agent.

The media type of each browser is evaluated by Cocoon at request time, based on their `User-Agent` http header information. Cocoon is preconfigured to handle these browsers:

- **explorer** - any Microsoft Internet Explorer, searches for MSIE (before searching for Mozilla, since they include it too)
- **opera** - the Opera browser (before searching for Mozilla, since they include it too)
- **lynx** - the text-only Lynx browser
- **java** - any Java code using standard URL classes
- **wap** - the Nokia WAP Toolkit browser
- **netscape** - any Netscape Navigator, searches for Mozilla

but you can add your own by personalizing the `cocoon.properties` file modify the `browser` properties. For example

```
browser.0=explorer=MSIE
browser.1=opera=Opera
browser.2=lynx=Lynx
browser.3=java=Java
browser.4=wap=Nokia-WAP-Toolkit
browser.5=netscape=Mozilla
```

indicates that Cocoon should look for the token *MSIE* inside the User-Agent HTTP request

header first, then *Opera* and so on, until *Mozilla*. If you want to recognize different generations of the same browser you should do find the specific string you should look for and indicate the order of searching since more browsers may contain the same string.

4.2 More complex examples

For more complex examples, please refert to the Cocoon distribution that includes many examples that cover all possible usages

5. Dynamic XML Generation

Web publishing is very limited without the ability to create dynamic content. For dynamic XML we refer to the content that is created as a function of request parameters or state of the requested resource. For this reason, a lot of work and design has been put into Cocoon to allow dynamic XML content to be generated.

5.1 The Servlet/JSP model

People are used to write small Java programs to create their dynamic web content. Servlets, and Java in general, are very powerful, easy to write and fast to debug, but they impose (like any other pure-logic solution) a significant management cost. This is due to the fact that programmable components like servlets must include both the logic to generate the dynamic code as well as all static elements (such as static content and style).The need for a more useful solution soon appeared.

To fill the gap between Java programmers and web engineers (groups that rarely overlap), Sun proposed the Java Server Pages (JSP) specification, a markup language (today with both SGML and XML syntax) that allows web engineers to include code in their pages, rather than include pages in their code. The impact of this strategy was significant: servlets were written directly in Java code if very little static content was to be used, otherwise JSP or other compiled server pages technologies were used.

This said, it would seem that using servlets/JPS to create dynamic XML content would be the perfect choice, unfortunately design issues impose that we take a second look to the technology and understand why this isn't so.

5.2 Servlet Chaining Vs. Servlet Nesting

Java Servlets were introduced by the Java Web Server team as a way to allow users to create their own *web plug-ins* . They were designed to handle the HTTP protocol and all possible dynamic web content (including HTML, XML, images, etc. both text and binary streams). Unfortunately, the need for a componentized request handler was not taken into serious consideration in the design phase but only later, when at an implementation phase.

In fact, the Java Web Server provided the ability to *chain* multiple servlets, one becoming the filter of the other. Unfortunately, since the API don't include such possibility in their design, such servlet chain is very limited in its behavior and poses significant restriction on the API use. Something that forced the Servlet API architects to come up with better solutions.

The solution was *servlet nesting*: the ability to include a servlet output inside its own transparently. This allowed programmers to separate different logic on different servlets, thus removing the need for servlet chaining

5.3 The limitations of Servlet Nesting

While servlet nesting was a major advantage over servlet chaining because it allowed servlets to be somewhat modular without loosing the full API power, a common design pattern applies to the Servlet model in general: no servlet is allowed to modify the output of another servlet. This holds true for all servlet API versions up to today (version 2.2).

This limitation is the key: if no further XML processing is needed on the server side, using servlets/JSP for creating XML is a perfect choice, but if this output requires some server side processing (for example XSLT transformations), the Servlet API does not allow another servlet to post process it's output. This other servlet is, in our case, Cocoon.

In a few words, the Servlet API doesn't support *Servlet Piping*.

5.4 The Cocoon model

Rather than turning Cocoon into a servlet engine, thus limiting its portability, this documents outlines some solutions that allow Cocoon users to get the servlet-equivalent functionality with internal Cocoon design ideas.

The Cocoon processing model is based on the separation of

- **Production** - where XML content is generated based on Request parameters (servlet equivalent)
- **Processing** - where the produced XML content is transformed/evaluated
- **Formatting** - where the XML content is finally formatted into the wanted output format for client use.

This separation of working contexts allows Cocoon users to implement their own internal modules to add the functionality they require to the whole publishing system. In fact, while a few of these components are already shipped with Cocoon, the highly modular structure allows you to build your own to fit your particular needs.

5.5 Writing Producers

Producers initiate the request handling phase. They are responsible to evaluate the HttpServletRequest parameters provided and create XML content that is fed into the processing reactor. A servlet logic should be translated into a producer if the request parameters can be used directly to generate the XML content (for example the FileProducer which loads the requested file from disk).

Here follows the code for an example producer distributed with Cocoon:

```
public class DummyProducer
  extends AbstractProducer
  implements Status
{

  String dummy = "<?xml version=\"1.0\"?>"
      + "<?cocoon:format type=\"text/html\"?>"
      + "<html><body>"
      + "<h1 align=\"center\">"
          + "Hello from a dummy page"
      + "</h1>"
```

```
        + "</body></html>";

    public Reader getStream(HttpServletRequest request)
      throws IOException
    {
      return new StringReader(dummy);
    }

    public String getPath(HttpServletRequest request) {
      return "";
    }

    public String getStatus() {
      return "Dummy Producer";
    }
  }
```

The key method is `getStream()` which is responsible to create process the given servlet request and provide an output stream for reading the generated XML document.

Note that Produce has also another method, `getDocument(request)`, which is responsible to return directly a DOM tree. In case you need to render you servlet code Cocoon-aware, the above example should tell you what to do.

Please, look at the shipped producers source code for example code and look at the user guide on how to install and use your own producers.

5.6 Writing Processors

If your servlet needs many parameters to work, it is more reasonable that you write a Processor instead. A Processor transforms a given XML document (which, in this case should contain the needed static parameters) into something else, driven both by the input document and by the request object which is also available.

Here is a simple processor example that should show you what the above means. Suppose you have the following document as input (note that it may have been produced from a file, from other sources or dynamically, see the above paragraph):

```
<?xml version="1.0"?>
<page>
 <p>Current time is <time/></p>
</page>
```

Our simple example processor will look for the %lg;time/%gt; tags and will expand them to the current local time, creating this result document:

```
<?xml version="1.0"?>
<page>
 <p>Current time is 6:48PM</p>
</page>
```

Please, look at the shipped processors source code for example code and look at the user guide on how to install and use your own processors.

5.7 Using Cocoon processors

The above example shows a very simple situation but needs non-trivial code to implement it. For this reason, the Cocoon distribution includes a number of processors that implement common needs and situations. These are:

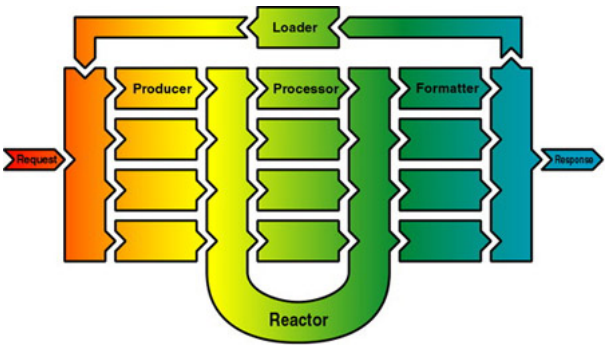
- **The XSLT processor** - the XSLT processor that applies XSLT transformations to the input document. XSLT allows you to solve your transformation needs as well as simple tag evaluation/processing due to its extensible and programmable nature.
- **The XSP processor** - the XSP processor that evaluates XSP pages and compiles them into producers. This processor allows you include programmatic logic into your pages as well as to separate the logic from the content.
- **The DCP processor** - the DCP processor that evaluates XML processing instructions with multi-language (Java and EcmaScript) logic. This processor allows you to do programmatic substitution and inclusion eliminating the need for complex processing logic.
- **The SQL processor** - the SQL processor that evaluates simple tags describing SQL queries to JDBC drivers and formats their result-set in XML depending on given parameters.
- **The LDAP processor** - the LDAP processor that evaluates simple tags describing LDAP queries to directory services and formats their result-set in XML depending on given parameters.

for more information on each of these processors, please, refer to the Cocoon web site or the Cocoon distribution.

6. The Cocoon Architecture

6.1 Cocoon Internals

The Cocoon publishing system has an engine based on the *reactor* design pattern which is described in the picture below:



Let's describe the components that appear on the schema:

- **Request** - wraps around the client's request and contains all the information needed by the processing engine. The request must indicate what client generated the request, what URI is being requested and what producer should handle the request.
- **Producer** - handles the requested URI and produces an XML document. Since producers are pluggable, they work like subservlets for this framework, allowing

users to define and implement their own producers. A producer is responsible of creating the XML document which is fed into the processing reactor. It's up to the producer implementation to define the function that produces the document from the request object.

- **Reactor** - is responsible of evaluating what processor should work on the document by reacting on XML processing instructions. The reactor pattern is different from a processing pipeline since it allows the processing path to the dynamically configurable and it increases performance since only those required processors are called to handle the document. The reactor is also responsible to forward the document to the appropriate formatter.
- **Formatter** - transforms the memory representation of the XML document into a stream that may be interpreted by the requesting client. Depending on other processing instructions, the document leaves the reactor and gets formatted for its consumer. The output MIME type of the generated document depends on the formatter implementation.
- **Response** - encapsulates the formatted document along with its properties (such as length, MIME type, etc..)
- **Loader** - is responsible of loading the formatted document when this is executable code. This part is used for compiled server pages where the separation of content and logic is merged and compiled into a Producer. When the formatter output is executable code, it is not sent back to the client directly, but it gets loaded and executed as a document producer. This guarantees both performance improvement (since the producer are cached) as well as easier producer development, following the common compiled server pages model.

6.2 Cocoon Processing Instructions

The Cocoon reactor uses XML processing instructions to forward the document to the right processor or formatter. These processing instructions are:

```
<?cocoon-process type="xxx"?> for processing

      and

<?cocoon-format type="yyy"?> for formatting
```

These PIs are used to indicate the processing and formatting path that the document should follow to be served. In the example above, we didn't use them and Cocoon wouldn't know (rather than by the presence of the XSL PIs) that the document should be processed by the XSLT processor. To do this, the HelloWorld.xml document should be modified like this:

```
<?xml version="1.0"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="hello.xsl" type="text/xsl"?>
<page>
 <title>Hello World!</title>
 <content>
  <paragraph>This is my first Cocoon page!</paragraph>
 </content>
</page>
```

The other processing instruction is used to indicate what formatter should be used to transform the document tree into a suitable form for the requesting client. For example, in the document below that uses the XSL formatting object DTD, the Cocoon PI indicates that this document should be formatted using the formatter associated to the `text/xslfo` document type.

```
<?xml version="1.0"?>
<?cocoon-format type="text/xslfo"?>

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
 <fo:layout-master-set>
  <fo:simple-page-master
      page-master-name="one"
      margin-left="100pt"
      margin-right="100pt">
   <fo:region-body margin-top="50pt"
       margin-bottom="50pt"/>
  </fo:simple-page-master>
 </fo:layout-master-set>

 <fo:page-sequence>
  <fo:sequence-specification>
   <fo:sequence-specifier-repeating
       page-master-first="one"
       page-master-repeating="one"/>
  </fo:sequence-specification>

  <fo:flow font-size="14pt" line-height="14pt">
   <fo:block>Welcome to Cocoon</fo:block>
  </fo:flow>
 </fo:page-sequence>
</fo:root>
```

6.3 Cocoon Cache System

In a complex server environment like Cocoon, performance and memory usage are critical issues. Moreover, the processing requirement for both XML parsing, XSLT transformations, document processing and formatting are too heavy even for the lightest serving environment based on the fastest virtual machine. For this reason, a special cache system was designed underneath the Cocoon engine and its able to cache both static and dynamically created pages.

This special cache system is required since the page is processed with the help of many components which, independently, may change over time. For example, a stylesheet or a file template may be updated on disk. Every processing logic that may change its behavior over time it's considered *changeable* and checked at request time for change.

Each changeable point is queried at request time and it's up to the implementation to provide a fast method to check if the stored page is still valid. This allows even dynamically generated pages (for example, an XML template page created by querying a database) to be cached and, assuming that request frequency is higher than the resource changes, it

greatly reduces the total server load.

Moreover, the cache system is based on a persistent object storage system which is able to save stored objects in a persistent state that outlives the JVM execution. This is mainly used for pages that are very expensive to generate and last very long without changes, such as compiled server pages.

The store system is responsible of handling the cached pages as well as the pre-parsed XML documents. This is mostly used by XSLT processors which store their stylesheets in a pre-parsed form to speed up execution in those cases where the original file has changed, but the stylesheet has not (which is a rather frequent case).

7. Future Directions

The Cocoon Project has gone a long way since it's creation on January 1999. It started as a simple servlet for static XSL styling and became more and more powerful as new features were added. Unfortunately, design decisions made early in the project influenced its evolution. Today, some of those constraints that shaped the project were modified as XML standards have evolved and solidified. For this reason, those design decisions need to be reconsidered under this new light.

While Cocoon started as a small step in the direction of a new web publishing idea based on better design patterns and reviewed estimations of management issues, the technology used was not mature enough for tools to emerge. Today, most web engineers consider XML as the key for an improved web model and web site managers see XML as a way to reduce costs and ease production.

In an era where services rather than software will be key for economical success, a better and less expensive model for web publishing will be a winner, especially if based on open standards.

7.1 Passive APIs vs. Active APIs

Web serving environments must be fast and scalable to be useful. Cocoon1 was born as a "proof of concept" rather than a production software and had significant design restrictions based mainly on the availability of freely redistributable tools. Other issues were lack of detailed knowledge on the APIs available as well as underestimation of the project success, being created as a way to learn XSL rather than a full publishing system capable of taking care of all XML web publishing needs.

For the above reasons, Cocoon1 was based on the DOM level 1 API which is a *passive* API and was intended mainly for client side operation. This is mainly due to the fact that most (if not all!) DOM implementations require the document to reside in memory. While this is practical for small documents and thus good for the "proof of concept" stage, it is now considered a main design constraint for Cocoon scalability.

Since the goal of Cocoon2 is the ability to process simultaneously multiple 100Mb documents in JVM with a few Mbs of heap size, careful memory use and tuning of internal components is a key issue. To reach this goal, an improved API model was needed. This is now identified in the SAX API which is, unlike DOM, event based (so *active*, in the sense that its design is based the *inversion of control* principle).

The event model allows document producers to trigger producing events that get handled in the various processing stages and get finally formatted in the response stream. This has significant impacts on performance and memory needs:

- **incremental operation** - the response is created during document production. Client's perceived performance is dramatically improved since clients can start receiving data as soon as it is created, not after all processing stages have been performed. In those cases where incremental operation is not possible (for example, element sorting), internal buffers store the events until the operation can be performed. However, even in these cases performance can be increased with the use of tuned memory structures.
- **lowered memory consumption** - since most of the server processing required in Cocoon is incremental, an incremental model allows XML production events to be transformed directly into output events and character written on streams, thus avoiding the need to store them in memory.
- **easier scalability** - reduce memory needs allow more concurrent operation to be possible, thus allowing the publishing system to scale as the load increases.
- **more optimizable code model** - modern virtual machines are based on the idea of hot spots, code fragments that are used often and, if optimized, increase the process execution by far. This new event model allows easier detection of hot spots since it's a method driven operation, rather than a memory driven one. Hot methods can be identified earlier and their optimization performed better.
- **reduced garbage collection** - even the most advanced and lightweight DOM implementation require at least three to five times (and sometimes much more than this) more memory than original document size. This does not only reduce the scalability of the operation, but also impact overall performance by increasing the number of memory garbage that must be collected after the response in sent to the client. Even if modern virtual machines reduced the overhead of garbage collection, less garbage will always have performance and scalability impacts.

The above points, alone, would be enough for the Cocoon2 paradigm shift, even if this event based model impacts not only the general architecture of the publishing system but also its internal processing components such as XSLT processing and PDF formatting. These components will require substantial work and maybe design reconsideration to be able to follow a pure event-based model. The Cocoon Project will work closely with the other component projects to be able to influence their operation in this direction.

Note: *Evolution of current DOM implementations might bring new considerations in the DOM vs. SAX issues. At this point, it's too early to tell which one will be the long time winner.*

7.2 Reactors Reconsidered

Another design choice that should be revised is the reactor pattern that was introduced to allow components to be connected in more flexible way. In fact, opposed to the fixed pipe model used up to Cocoon 1.3.1, the reactor approach allows components to be dynamically connected, depending on reaction instructions introduced inside the documents.

While this at first seemed a very advanced and highly appealing model, it turned out to be a very dangerous approach. The first concern is mainly technical: porting the reactor pattern under an event-based model requires limitations and tradeoffs since the generated events must be cached until a reaction instruction is encountered.

But even if the technical difficulties are solved, a key limitation remains: there is no single point of management.

7.3 Management Considerations

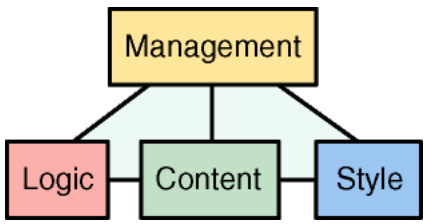
The web was created to reduce information management costs by distributing them back on information owners. While this model is great for user communities (scientists, students, employees, or people in general) each of them managing small amount of personal information, it becomes impractical for highly centralized information systems where distributed management is simply not practical.

While in the HTML web model the page format and URL names where the only necessary contracts between individuals to create a world wide web, in more structured information systems the number of contracts increases by a significant factor due to the need of increased coherence between the hosted information: common style, common design issues, common languages, server side logic integration, data validation, etc...

It is only under this light that XML and its web model reveal their power: the HTML web model had too little contracts to be able to develop a structured and more coherent distributed information system, reason that is mainly imposed by the lack of good and algorithmically certain information indexing and knowledge seeking. Lacks that tend to degrade the quality of the truly distributed web in favor of more structured web sites (that based their improved site structure on internal contracts).

The simplification and engineering of web site management is considered one of the most important Cocoon2 goals. This is done mainly by technologically imposing a reduced number of contracts and place them in a hierarchical shape suitable to replace current high-structure web site management models.

The model that Cocoon2 adopts is the "pyramid model of web contracts" which is outlined in the picture below



and is composed by four different working contexts (the rectangles)

- **Management** - the people that decide what the site should contain, how it should behave and how it should appear
- **Content** - the people responsible to write, own and manage the site content. This context may contain several sub-contexts one for each language used to express page content.
- **Logic** - the people responsible for integration with dynamic content generation technologies and database systems.
- **Style** - the people responsible for information presentation, look & feel, site graphics and its maintenance.

and five contracts contexts (the lines)

- management - content
- management - logic
- management - style
- content - logic

- content - style

7.4 Overlapping contexts and Chain Mapping

The above model can be applied only if the different contexts never overlap, otherwise there is no chance of having a single management point. For example, if the W3C-recommended method to link stylesheets to XML documents is used, the content and style contexts overlap and it's impossible to change the styling behavior of the document without changing it. The same is true for the processing instructions used by the Cocoon1 reactor to drive the page processing: each stage concur to determine the result thus increasing management and debug complexity. Another overlapping in context contracts is the need for URL-encoded parameters to drive the page output. These overlaps break the pyramid model and increase the management costs.

In Cocoon2, the reactor pattern will be abandoned in favor of a chain mapping technique. This is based on the fact that the number of different contracts is limited even for big sites (for example, even if the pages are millions, they probably all share no more than a few different DTDs and each DTD has no more than a couple of stylesheets).

Also, for performance reasons, Cocoon2 will try to compile everything that is possibly compilable (pages/XSP into producers, stylesheets into processors, etc...) so, in this new model, the *processing chain* that generates the page contains (in a direct executable form) all the information/logic that handles the requested resource to generate its response.

This means that instead of using even-driven request-time DTD interpretation (done in all Cocoon1 processors), these will be either compiled into processors directly (XSLT stylesheet compilation) or compiled into producers using logicsheets and XSP which will remove totally the need for request-time interpretation solutions like DCP that will be removed.

7.5 Pre-compilation, Pre-generation and Caching

The cache system in Cocoon1 will be ported with no important design changes since it's very flexible and was not polluted by early design constraints since it appeared in later versions. The issue regards static file caching that, no matter what, will always be slower than direct web server caching.

To be able to put most of the static part job back on the web server (where it belongs), Cocoon2 will greatly improve it's command line operation, allowing the creation of *site makefiles* that will automatically scan the web site and the source documents and will provide a way to *regenerate* the static part of a web site (images and tables included!) based on the same XML model used in the dynamic operation version.

It will be up to the web server administrator to use static regeneration capabilities on a time basis, manually or triggered by some particular event (database update signal) since Cocoon2 will only provide servlet and command line capabilities. The nice integration is based on the fact that there will be no behavioral difference if the files are dynamically generated in Cocoon2 via the servlet operation and cached internally or pre-generated and served directly by the web server, as long as URI contracts are kept the same by the system administrator (via URL-rewriting or aliasing)

Also, it will be possible to avoid on-fly page and stylesheet compilation (which make debugging harder) with command line pre-compilation hooks that will work like normal compilers from a developer's point of view.

8. Conclusions

Cocoon is a big and very ambitious project, not only for the technological issues involved (which will require strong integration with XML components) but also for the significant paradigm shifts imposed by the new technologies. On the other hand, we strongly believe this to be the winner model for future web engineering and if you believe in this yourself, we invite you to join us or help us in any way you can provide.

Thank you.

Appendices

A.1 State of the art

This document refers to the version 1.6 of Cocoon and contains information and details that belong to this particular version. Althought the Cocoon project takes into great consideration back compatibility with previous software versions, it cannot be guaranteed that this paper will remain up-to-date until the conference.

While in the *future directions* part I tried to indicate possible future improvements that might require back incompatible changes, a couple of months may be a very long time in software development, expecially given the lack of well established programming practices and feature collections in the XML world.

A.2 The making of this document

This document was created by Cocoon out of a valid XML file and applying an XSLT stylesheet that transformed it into XSL:FO. Then these formatting objects were formatted by FOP into a PDF file.

I apologize for some rendering mistakes that you might find here and there, also, note that image support was added a couple of days before this document was generated and for this reason, they are not as good as they should be.

Anyway, I like the fact that no other tool rather than Cocoon was used to generate such a document.

A.3 Software

- [Cocoon] "The Cocoon XML Publishing Framework", http://xml.apache.org/cocoon/
- [Xerces] "The Xerces-J XML Parser", http://xml.apache.org/xerces-j/
- [Xalan] "The Xalan XSLT Processor", http://xml.apache.org/xalan/
- [FOP] "The FOP FO + SVG Formatter", http://xml.apache.org/xerces-j/
- [ProjectX] "The Sun ProjectX XML Parser", http://java.sun.com/xml/
- [XT] "James Clark's XSLT Processor", http://www.jclark.com/

A.4 Bibliography

- [XML] "Extensible Markup Language (XML) 1.0 Specification", T. Bray, J. Paoli, C. M. Sperberg-McQueen, 10 February 1998, http://www.w3.org/TR/REC-xml
- [XMLNAMES] "Namespaces in XML", T. Bray, D. Hollander, A. Layman, 14 January 1999, http://www.w3.org/TR/REC-xml-names
- [XSLT] "XSL Transformations (XSLT) Specification Version 1.0", J. Clark, 16 November 1999, http://www.w3.org/TR/xslt

Copyright (c) 2000 Stefano Mazzocchi. All Rights Reserved.