XSP Logicsheet Guide (2.1 legacy document)

Table of contents

Table of contents

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

1. Introduction

This document is intended as an introduction and brief tutorial to using and creating Apache Cocoon XSP logicsheets. It is assumed that the reader has a working knowledge of XML and XSLT, and has worked through at least the basic XSP examples supplied with Cocoon. Although this is not intended as a tutorial for XSP specifically, much of the material may be helpful in gaining a fuller understanding of XSP.

2. Taglibs and logicsheets

There is some confusion over the terms "taglib" and "logicsheet". Many people will use these terms interchangeably. The term "taglib" comes from JSP, which inspired XSP. An XSP logicsheet is a "tag library" in the sense that it defines a set of custom XML tags which can be used within an XSP program to insert whole blocks of code into the file. Cocoon comes with several pre-defined "taglibs", such as the request taglib. Using the request taglib, you can insert the following tag into your XSP code to obtain the HTTP request parameter named "fruit" (e.g., if your URL looks something like "http://myserver.org/index.xml?fruit=apple", the value of "fruit" is "apple"):

<request:get-parameter name="fruit"/>

We will discuss how to use Cocoon's pre-defined taglibs in a later section. The important thing to understand is that all taglibs are defined by a logicsheet. A logicsheet, as we will see, is a special kind of XSL stylesheet, whose output is an XSP file.

3. Hello World!

We'll start with some simple Hello, World! examples, starting with a simple HTML file, and extending it using Cocoon technologies until we have a working (if trivial) example of XSP using a custom logicsheet.

3.1. Simple HTML Example

All of the examples in this section will produce HTML output essentially equivalent to this:

<html> <body> <h1>Hello, world!</h1> </body> </html>

I did say these would be simple examples, didn't I?

3.2. Simple XML/XSL Example

Here's a simple XML file:

greeting.xml <?xml version="1.0"?> <greeting>Hello, world!</greeting>

...and here's the XSL stylesheet that will transform it into an HTML file similar to the one we started this section with:

greeting.xsl <?xml version="1.0"?> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"> <xsl:template match="/"> <html> <body> <h1> <xsl:value-of select="greeting"/> </h1> </body> </html> </xsl:template> </xsl:stylesheet>

So far, nothing exciting. The input XML has a single element, <greeting>, whose text contents gets spit out in HTML. The contents of our particular XML file's greeting is, predictably, "Hello, World!" The point of showing you this is that, as we elaborate our example by adding Java code through XSP, and later with a custom logicsheet, we will continue to use the same stylesheet to format our final output. So, the input XML will generally look much like the XML file in this most recent trivial example.

3.3. Simple XSP Example

Next, we continue in our trivial vein by using trivial Java code to make an XSP program, whose output will mimic that of our XML file above. The output of this file is transformed to HTML by the same XSL stylesheet as above:

greeting2.xml <?xml version="1.0"?> <xsp:page xmlns:xsp="http://apache.org/xsp"> <xsp:logic> // this could be arbitrarily complex Java code, JDBC queries, etc. String msg = "Hello, world!"; </xsp:logic> <greeting> <xsp:expr>msg</xsp:expr> </greeting> </xsp:page>

3.4. Simple XSP Logicsheet Example

Now we are ready to present our final trivial example, using a custom logicsheet. There are two files shown below. The first is an XSP file that uses a custom logicsheet, logicsheet.greeting.xsl, which is the second file shown below.

greeting3.xml <?xml version="1.0"?> <xsp:page xmlns:xsp="http://apache.org/xsp" xmlns:greeting="http://duke.edu/tutorial/greeting"> <greeting> <greeting:hello-world/> </greeting> </xsp:page> logicsheet.greeting.xsl <?xml version="1.0"?> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xsp="http://apache.org/xsp" xmlns:greeting="http://duke.edu/tutorial/greeting" version="1.0"> <xsl:template match="xsp:page"> <xsl:copy> <xsl:apply-templates select="@*"/> <xsl:apply-templates/> </xsl:copy> </xsl:template> <xsl:template match="greeting:hello-world"> <!-- more complex XSLT is possible here as well --> <xsp:logic> // this could be arbitrarily complex Java code, JDBC queries, etc. String msg = "Hello, world!"; </xsp:logic> <xsp:expr>msg</xsp:expr> </xsl:template> <!-- This template simply copies stuff that doesn't match other --> <!-- templates and applies templates to any children. --> <xsl:template match="@*|node()" priority="-1"> <xsl:copy> <xsl:apply-templates select="@*|node()"/> </xsl:copy> </xsl:template> </xsl:stylesheet>

There are several things to note about these two files. First, note that we inform the XSP processor that it should apply our custom logicsheet using the processing instruction

<?xml-logicsheet href="logicsheet.greeting.xsl"?>

There are other ways to associate a logicsheet with an XSP file, which we'll discuss later. Next, note that our logicsheet defines a new namespace, **greeting:**, which must be declared in both files using the same URI:

xmlns:greeting="http://duke.edu/tutorial/greeting"

Note that the URI is completely arbitrary. I've chosen to construct my namespace URI's by using my institution's web address (http://duke.edu/) followed by the project name (tutorial) and namespace name (greeting). You may use any scheme you wish for your namespace URI's; however, the URI declared in the logicsheet **must** match the URI declared in the XSP which uses the logicsheet.

Finally, note that our logicsheet is merely an XSL stylesheet. It transforms one XML file into another. What makes it a logicsheet is that it can be applied not just to any XML file, but specifically to an XSP file, and the end result of its transformation is another XSP file. If you were to apply the logicsheet in

this example to the XML file in this example as just a stylesheet (with no XSP processing), you would end up with something like the following (compare to our earlier XSP example):

<?xml version="1.0"?> <xsp:page xmlns:greeting="http://duke.edu/tutorial/greeting" xmlns:xsp="http://apache.org/xsp"> <greeting> <xsp:logic> // this could be arbitrarily complex Java code, JDBC queries, etc. String msg = "Hello, world!"; </xsp:logic> <xsp:expr>msg</xsp:expr> </greeting> </xsp:page>

4. Using Logicsheets (Taglibs)

There are two ways to apply a logicsheet, once you have written it. First, as in the previous examples, you can tell XSP explicitly what logicsheets to apply, using the <?xml-logicsheet?> processing instruction right after xml header and before <xsp:page> tag:

<?xml version="1.0"?> <?xml-logicsheet href="logicsheet.greeting.xsl"?>

There is another way to apply a logicsheet, which doesn't require a processing instruction for each file that uses the logicsheet. The second way is to declare logicsheet in the cocoon.xconf file. These declarations take the form

<builtin-logicsheet> <parameter name="prefix" value="<logicsheet's prefix>"/> <parameter name="uri" value="<logicsheet's namespace URI>"/> <parameter name="href" value="<URL to file>"/> </builtin-logicsheet>

Cocoon's pre-defined logicsheets are already declared in this file. For instance, the declaration of the XSP request taglib is the following:

<builtin-logicsheet> <parameter name="prefix" value="xsp-request"/> <parameter name="uri" value="http://apache.org/xsp/request/2.0"/> <parameter name="href" value="resource://.../markup/xsp/java/request.xsl"/> </builtin-logicsheet>

This line associates the **http://apache.org/xsp/request/2.0** namespace with the logicsheet named in the URL. This URL points to a file that is stored in the cocoon.jar. To use the request taglib, you must declare the request namespace in your XSP file:

... <xsp:page xmlns:xsp="http://apache.org/xsp" xmlns:xsp-request="http://apache.org/xsp/request/2.0" > ... You should **not** try to apply the xsp-request taglib using the <?xml-logicsheet?> processing instruction, as this will result in the logicsheet being applied twice.

You can add your own logicsheets to the cocoon.xconf file using the same syntax. The only trick is constructing an appropriate URL. If we wanted to declare our **greeting:** namespace and logicsheet from the Hello, World! example above, and if the logicsheet were stored (on a UNIX filesystem) in the location /cocoon/logicsheets/logicsheet.greeting.xsl, we'd add this line to cocoon.xconf:

<builtin-logicsheet> <parameter name="prefix" value="greeting"/> <parameter name="uri" value="http://duke.edu/tutorial/greeting"/> <parameter name="href" value="file:///cocoon/logicsheets/logicsheet.greeting.xsl"/> </builtin-logicsheet>

There are some very important differences between using the <?xml-logicsheet?> processing instruction vs. the cocoon.properties entry to apply a logicsheet. Using cocoon.properties, any time the logicsheet changes, it is necessary to restart Cocoon. If you instead use the processing instruction, Cocoon will detect modifications to your logicsheet, and recompile your XSP programs accordingly. Also, if you need to explicitly control the order in which your logicsheets are applied, you need to use the processing instruction. Logicsheets will be applied in the order in which they appear in processing instructions in your source file.

Whichever method you use, the most important thing to remember is that you must declare, in your

XSP program, the namespace for a logicsheet using the same URI as in the logicsheet itself.

5. Logicsheet Development Tips

5.1. Development Practices

Developing Logicsheets can be a frustrating mental exercise, as it requires you to understand and keep in mind the complex coordination of several different technologies: XML, XSLT, XSP, and Java. A bad assumption in any of these areas can lead to an hour of debugging. Following a few simple practices can reduce the frustration and make logicsheet programming less difficult:

Small Increments
As with any software development, it is much easier to debug a small amount of code than a large amount of code. XSP is no different, except that the complexity of a large amount of code is multiplied by the number of different technologies. So, write a tiny bit of code and get it working, or start with a simple piece of code that is already working. Make small changes, and get each change working before making the next.

Prototype New Ideas
Before trying something you haven't done before (e.g., a new XPath expression, a new Java syntax), prototype it in a simple environment where you can easily see the results of your code. It is more difficult to debug your changes if the output is filtered through multiple stylesheets and rendered into HTML. So instead, write a small XSP that you can use to test your code fragment and see the resulting XML.

Use the Source
After transforming your XSP code with your logicsheet, the XSP processor writes the resulting Java code to a file in your repository. The repository is in a directory specified in cocoon.properties. Make a shortcut to your repository directory and go there often. Read the code that resulted from application of your stylesheet. This lets you debug the Java code as Java code, absent from all of the XML/XSL complications. It also lets you see exactly the results of XSLT transformation using your logicsheet.

Steal Code
The authors of the logicsheets distributed with Cocoon have already solved numerous problems that you may encounter. Read their code (it is in the source tree) and borrow from it liberally. Reading this code is also a good way to gain insight into logicsheet design.

5.2. Standard Templates

As we discussed earlier, a logicsheet is just an XSLT stylesheet which transforms one XSP source file into another. Since we are always expecting to act on an XSP source file, and there is the possibility that other logicsheets may also be acting on the same file (either before or after our logicsheet), there are a few templates which are more or less required in any logicsheet. The templates below were all pulled from the taglib logicsheets distributed with Cocoon.

The first of these is simply a template to copy anything you don't directly act upon yourself. You probably have a template similar to this in most of your stylesheets already.

```
<xsl:template match="@*|node()" priority="-1"> <xsl:copy> <xsl:apply-templates select="@*|node()"/> </xsl:copy> </xsl:template>
```

If your code requires any Java imports, or if you want to declare methods or variables at the class level, you will need to have a way to add elements to the <xsp:page> element that is at the root of the

source file. Here is a template to let you do that (from esql.xsl):

```
<xsl:template match="xsp:page"> <xsp:page> <xsl:apply-templates select="@*"/>
<xsp:structure> // you can put &lt;xsp:include> statements in here to import Java classes
</xsp:structure> <xsp:logic> // put class-level variable declarations and methods here
</xsp:logic> </xsp:page> </xsl:template>
```

Frequently, you may also need to declare variables or perform initialization that needs to occur before any of the code in your custom tags. You could, of course, require that the users of your logicsheet use one particular tag before using any other, and then put your declarations and initializations in the template matching that one tag. This may not be the best solution, however, and may be a source of confusion. Instead, the following template can be used to insert code inside the populateDocument() method, after the standard XSP code (such as declaration of the request and response variables), but before any user code from the source XSP file (including code inserted by your custom tags). The complex XPath expression here just says "match on any child elements of <xsp:page which don't themselves begin with 'xsp:'". Since the <xsp:page> element always has a single element which isn't in the xsp: namespace, this will be matched once and only once.

```
<xsl:template match="xsp:page/*[not(starts-with(name(.), 'xsp:'))]"> <xsl:copy>
<xsl:apply-templates select="@*"/> <xsp:logic> // This code ends up inside
populateDocument() before any user code </xsp:logic> <xsl:apply-templates/> </xsl:copy>
</xsl:template>
```

5.3. Logicsheets Using Logicsheets

Since software tends to build on other software, you will probably find yourself wanting to write logicsheets which make use of other logicsheets, particularly the logicsheets provided with Cocoon. This is very possible. When using one logicsheet inside another, the most important thing to remember is that you must declare the namespace for the second logicsheet not only in the first logicsheet, but also in any XSP program that uses the first logicsheet, even if it doesn't directly reference any of the tags in the second logicsheet.

1. Comments

add your comments