# Working with XSP in Apache Cocoon 2

Presented by developerWorks, your source for great tutorials

**ibm.com/developerWorks**

---

## Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

# Section 1. Tutorial introduction

# Who should take this tutorial?

This tutorial is the second in a series of tutorials on Apache Cocoon 2. It introduces XML Server Pages (XSP), the Cocoon technology for generating dynamic XML content. It is designed to build on the concepts described in the first tutorial, *Introducing Cocoon 2*; it is therefore recommended that you complete the previous tutorial before progressing further.

This tutorial is based on Cocoon 2. While XSP was present in Cocoon 1, architectural differences mean that the majority of the examples are not applicable to that version.

It is assumed that you are familiar with Java Web development frameworks such as Java Servlets and Java Server Pages (JSP). Where appropriate, parallels are drawn between concepts in these technologies and XSP.

This tutorial will be of interest primarily to developers who have progressed beyond the basic features of Cocoon 2 and want to learn how to add dynamic data to their XML documents to create richer Web sites. The tutorial is also relevant for developers who need to integrate Cocoon 2 with existing data sources and/or APIs to publish their data in multiple formats over the Internet.

To achieve these goals the tutorial will:

- Introduce the basic principles behind XSP
- Review the XSP syntax, allowing developers to begin creating dynamic XML documents
- Introduce custom XSP tag libraries, known as *logicsheets*
- Summarize the features provided by the default logicsheets that are built into Cocoon 2
- Describe how to create custom logicsheets

---

# What is XSP?

The first tutorial in this series described the Apache Cocoon 2 architecture, which is based on a pipeline-processing model. This model breaks down the processing of a Web request into the following basic stages:

- Parsing of XML data to *generate* input for the pipeline
- *Transformation* and manipulation of that data
- *Serialization* of the results, for delivery to the user

XML Server Pages (XSP) is a Cocoon 2 technology that enables the creation of

dynamic XML data sources for feeding data into Cocoon 2 pipelines. These data sources are described using a combination of XML markup and application logic that is then automatically compiled into a Java class by the Cocoon 2 engine.

XSP provides a flexible platform for developing applications using Cocoon 2. For example, information in existing application databases can be exposed by Cocoon 2 applications, enabling a greater variety of data-delivery options. XSP allows Cocoon 2 to be used in an application integration environment, such as middleware and document publishing, by providing a means to show the data source through an XML interface.

---

# XSP and JSP

Java Server Pages (JSP) is the most widely used way of generating dynamic Web interfaces using a combination of Java and HTML or XML. The resulting document is interspersed with custom tags and/or snippets of Java code that add the dynamic sections. Preserving the document structure makes JSP pages easier to maintain for non-programmers. However while the markup is emphasized for the end user, a servlet container will compile a JSP page into a standard Java servlet.

XSP has a lot in common with JSP. Both technologies:

- Consist of a mixture of program code and markup
- Are compiled into a binary form for execution
- Allow the creation of custom tag libraries

Yet XSP differs from JSP in two regards. First, XSP provides a framework for mixing code in any programming language with XML markup. XSP pages in Cocoon 2 are primarily created using Java, though other implementations are possible. For example, the AxKit XML application server (see Resources on page 30 ) provides an implementation of XSP that uses Perl. In contrast, JSP is specifically a Java technology.

The second and more practical difference is that XSP generates dynamic *data*, not dynamic *presentations*. JSP, on the other hand, is a presentation layer technology used to produce either HTML or XML, typically at the end of a series of processing steps. XSP pages generate XML data for Cocoon pipelines, which create the desired presentation.

---

# Tools

No specific tools are required to complete this tutorial. To experiment with the technologies, however, it is imperative that you read the instructions on installing Cocoon 2 and Jakarta Tomcat servlet engine in the first tutorial. Refer to the *Installing and configuring Cocoon* section in the first tutorial for complete instructions.

**Note**: The Cocoon 2 project has released at least one new version of the application since the publication of the first tutorial in this series. It is recommended that you upgrade to the new release, which includes a number of bug fixes and enhancements. It is available at *http://xml.apache.org/cocoon/dist/*. Instructions on upgrading are available in the next panel.

You can also *download the source code for examples* in this tutorial.

---

## Upgrading Cocoon 2

When upgrading, take care to back up any existing configuration, style sheets, and XML documents. Do a fresh install of Cocoon 2 while retaining the existing installation. This allows applications to be migrated over one-by-one. Here are the steps:

- Shut down Tomcat
- Rename the Cocoon webapps directory to something like "old-cocoon"
- Copy the new Cocoon 2.0.2 Web application into the Tomcat webapp directory
- Delete the content of the Tomcat work directory
- Restart Tomcat

Once Tomcat has restarted, both the old and the new versions of Cocoon are available -- the old version at `http://localhost:8080/old-cocoon/` and the new version at `http://localhost:8080/cocoon/`.

The following steps demonstrate one way to do this:

```
$CATALINA_HOME/bin/shutdown.sh
cd $CATALINA_HOME/webapps
mv cocoon old-cocoon
cp $COCOON_HOME/cocooon.war $CATALINA_HOME/webapps
rm -r $CATALINA_HOME/work/*
$CATALINA_HOME/bin/startup.sh
```

---

## About the author

Leigh Dodds is the team leader of the Research and Technology Group at Ingenta Ltd. He has five years experience developing on the Java platform, and has spent the last three years working with XML and related technologies. Leigh is also a contributing editor to xmlhack.com, and has been writing the regular "XML-Deviant" column on XML.com since February 2000. He holds a bachelors degree in Biological Science, and a masters in Computing. Leigh can be reached at *leigh@xmlhack.com*.

Having recently become a father, when he's not wrestling with pointy brackets, Leigh can be found making silly noises with his son Ethan.

# Section 2. XSP concepts

## Introduction

To properly understand the role that XSP plays in the Cocoon 2 architecture, a number of important concepts must be mastered. They are detailed in this section:

- What information an XSP document describes
- How an XSP page is processed by Cocoon 2
- How XSP pages are put to use within Cocoon 2 pipelines
- Where to find the automatically generated source code

---

## What an XSP describes

To illustrate what an XSP page actually describes, it's useful to begin with the obligatory "Hello World" example:

```
<xsp:page language="java" xmlns:xsp="http://apache.org/xsp">
   <message>Hello World!</message>
</xsp:page>
```

When processed by Cocoon 2, this page generates the following XML document:

```
<message>Hello World!</message>
```

While this is a trivial example, it demonstrates that the original document contains a mixture of two kinds of markup: *XSP elements*, which are interpreted by Cocoon 2, and *user elements*, which describe the desired output document.

When Cocoon 2 reads an XSP document, it internally converts this markup into the source code for a Cocoon Generator. This source is subsequently compiled and then executed. The two types of markup play different roles in this process. The XSP elements are used as cues to define the *structure* of the generated source code, while the user elements, content, and other application logic are used to create *instructions* for the Generator that describe how its output should be constructed.

XSP is therefore a means to automatically generate program code from a declarative description of what that code should do. The above example declares that the generated code should "create a `message` element, and give it the text content 'Hello World'". In reality, XSP pages aren't always entirely declarative because it's possible to directly include supplementary program code. However, careful use of a feature known as *logicsheets* can reduce or even eliminate the need to do this. Logicsheets on page 21

are discussed in a later section of the tutorial.

The way in which Cocoon 2 processes an XSP page is analogous to how a JSP container processes a JSP page to create the code for a Java Servlet. The author of the page doesn't need to know how to write a servlet; her job is to create HTML elements, add custom tags from a tag library, and perhaps add snippets of Java code. It's the job of the JSP engine to use the structure of the JSP page to create the servlet automatically, just as it is the job of Cocoon 2 to automatically create a Generator from an XSP page.

---
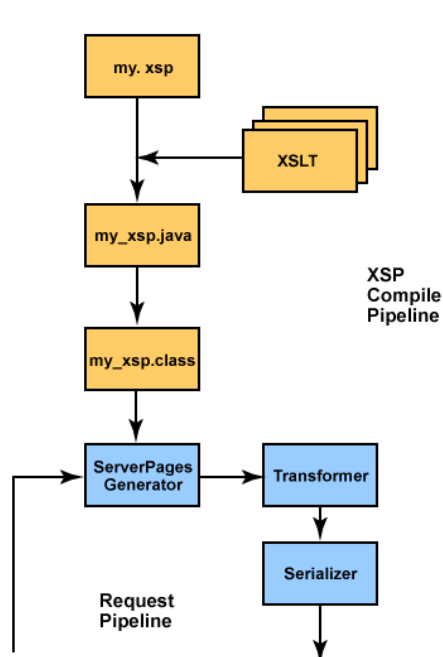
# The XSP compilation process

The process of turning an XSP page into a Generator involves several steps, summarized in the diagram to the left.

A request is received by Cocoon and it determines, from the sitemap configuration, that the XML data for this request should be created by an XSP page. The data is then passed to a Transformer and after that a Serializer, which produces the final response for the user.

As this is the first time a request to this pipeline has been received, Cocoon 2 must compile the required XSP page (an XSP page is also recompiled if its original source changes). This involves the following steps:

1. Parsing the XSP document
2. Transforming the XSP page using a dedicated XSLT style sheet, to generate Java source code
3. Storing the resulting Java file, after neatly indenting and formatting the source using a code formatter
4. Compiling the source into a Generator
5. Loading and executing the compiled Generator within the request pipeline

A key concept to recognize is that this whole process is entirely separate from the request processing pipeline (indicated by the different colors in the diagram). This means that the transformation steps used to create the Generator source code do not have access to the initial request. Therefore, this transformation does not have access to parameters contained in the request, and so it's processing cannot be affected by individual requests. However, when the
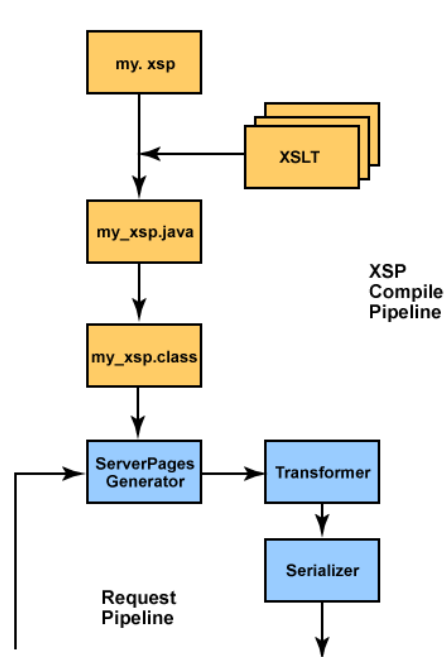
Generator *executes*, it has access to the request environment, in exactly the way that a JSP page has access to its request, etc.

Now that we've reviewed how XSP pages are processed by Cocoon 2, the next step is to understand how to make use of them within pipelines.

# Using XSP pages

Cocoon 2 provides a `ServerPagesGenerator` component that coordinates the compilation and execution of individual XSP pages. This Generator must be added to a pipeline to trigger processing of a particular XSP page.

The `ServerPagesGenerator` is declared in the sitemap as follows:

```
<map:generators default="file">
   <map:generator  name="xsp" src="org.apache.cocoon.generation.ServerPagesGenerator"/>
   <!-- ... other generator declarations ... -->
</map:generators>
```

To use this generator in a pipeline, simply state that it should be used (rather than the default) and indicate where the source for a particular XSP page can be found, as follows:

```
<map:pipeline>
   <map:match pattern="tutorial/*.xml">
      <map:generate type="xsp" src="tutorial/{1}.xsp"/>
      <map:serialize type="xml"/>
   </map:match>
</map:pipeline>
```

This pipeline matches the request for

`http://localhost:8080/cocoon/tutorial/helloworld.xml`. The `ServerPagesGenerator` then reads the file `$CATALINA_HOME/webapps/cocoon/tutorial/helloworld.xsp` and compiles it into a second Generator, which is then executed. The output of the Generator will then be serialized back to the user.

Confirm the exercise by saving the "Hello World" example (which can be downloaded from the Tools on page 3 panel) into `$CATALINA_HOME/webapps/cocoon/tutorial/`. You will need to create the tutorial directory first.

Viewing the URL in Internet Explorer should show the expected document (use "View Source" in Netscape to see the full XML document).

---

# The XSP source code

Once an XSP has been executed, you can examine the source code that was generated during the compilation process. When running Cocoon 2 with Tomcat, all source code for XSP pages is kept in the following directory tree:

`$CATALINA_HOME/work/localhost/<web-app-name>/cocoon-files/org/apache/cocoon/www/`

The source for the "Hello World" Generator will be found in the file:

```
$CATALINA_HOME/work/localhost/cocoon/cocoon-files  \
/org/apache/cocoon/www/tutorial/helloworld_xsp.java
```

Note that the generated Java file name is derived from the original XSP document name (i.e., myfile.xsp becomes myfile_xsp.java).

Examining the source code shows that:

- The class has a number of standard Java and Cocoon 2 class imports
- The generated class is a subclass of `XSPGenerator`
- The class has a single `generate()` method that performs all the processing
- The generate method contains SAX API method calls -- `startElement`, `characters`, and `startElement` methods -- that describe the desired output document

It's useful to know where to find the source code generated for an XSP page. Often errors in the original XSP document result in compilation errors in the generated Java class. In these circumstances, Cocoon will generate an error page that indicates the line number of the problematic code. Having the source at hand can help diagnose the problem.

**Important Note:** Never directly change the source code generated by Cocoon 2. If the original XSP document is subsequently altered, then the source will be overwritten by a freshly generated version, and any other changes will be lost. Use the appropriate XSP elements (e.g., `logic`) to insert extra code where necessary. Further examples of this are given in the next section.

The simple "Hello World" example illustrates the basic principles of how XML Server Pages operate within the Cocoon 2 server framework, and how the pages themselves are used to automatically generate Java classes. The next section describes the XSP syntax.

# Section 3. XSP syntax

## The xsp:page element

This section introduces the elements that make up the complete syntax for XML Server Pages.

The `xsp:page` element is the root element of every XSP document. It must have a language attribute that identifies the programming language that the page contains -- specifically `"java"`. Other implementations of XSP may support different languages, but Java is the best option when using Cocoon 2.

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

   <!-- page contents -->

</xsp:page>
```

Note that the XSP element declares the XSP namespace, `http://apache.org/xsp`, with the prefix `xsp`. All XSP elements must be qualified with this namespace prefix. For clarity, element names are referred to in this format for the rest of the tutorial (e.g. `xsp:page`, `xsp:comment`, etc.).

The page element can contain:

- Any number of `xsp:structure` elements
- Any number of `xsp:logic` elements
- A **single** user element

This last restriction is very important. "User element" means any element not in the XSP namespace, including elements with no namespace. This restriction exists because the user element becomes the root element of the XML document created by the XSP page, and an XML document can only have a single root element.

---

## The xsp:structure and xsp:include elements

When adding programming logic that makes use of standard or custom Java APIs, you must indicate in the XSP that additional import statements are needed in the generated source code to ensure that compilation completes successfully.

The `xsp:structure` and `xsp:include` elements are used to provide these additional cues to the code-generation process. The elements are to be used in tandem, with the `xsp:structure` element grouping together a number of

xsp:include elements. Each xsp:include defines an additional Java package or class to import. The following example illustrates this usage:

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

    <xsp:structure>
        <xsp:include>java.util.Calendar</xsp:include>
        <xsp:include>java.text.*</xsp:include>
    </xsp:structure>

    <!-- page contents -->

</xsp:page>
```

An xsp:include element must contain only text and no additional markup. The contents can be either a fully specified package or a class name, as would be used in a Java import statement. However, neither the import keyword nor the semicolon is required. The code generator adds these automatically, and including them will generate an error.

The above example results in the following lines being added to the generated source code:

```
import java.util.Calendar;
import java.text.*;
```

A number of default imports are also added during the code-generation process. These are for specific classes from the Cocoon, Avalon, and SAX packages, as well as several utility classes from the standard Java API.

## The xsp:logic element

The xsp:logic element is used to add blocks of Java code to an XSP.

Wherever these elements appear as the direct children of the xsp:page element (i.e., outside of the single user element) the block of code should contain method definitions and/or member variables. This occurs because code that appears outside of the user element is not contained within the Generator's generate() method. Normal Java syntax rules mean that this code must be member or class (i.e., static) variables or methods.

The following example illustrates this, defining a method that can be invoked from elsewhere in the XSP that returns the current time:

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

   <xsp:logic>
      public String getTime()
      {
         return java.util.Calendar.getInstance().getTime().toString();
      }
   </xsp:logic>

   <document/>

</xsp:page>
```

`xsp:logic` elements can also be used elsewhere in the XSP page. In these circumstances, they should contain Java statements that will be added to the `generate()` method in the compiled Generator. This makes the `xsp:logic` element similar to the `<% ... %>` scriptlet syntax used in JSP pages.

The following example illustrates how to create blocks of code with the `xsp:logic` element.

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

      <xsp:logic>
         SimpleDateFormat format = new SimpleDateFormat("EEE, MMM d, yyyy");
         String timestamp = format.format(
                                   java.util.Calendar.getInstance().getTime()
                                   );
      </xsp:logic>

      <time><xsp:expr>timestamp</xsp:expr></time>

      <!-- additional elements -->
   </document>
</xsp:page>
```

The code includes logic that creates a timestamp when the XSP document is evaluated. This is then added to the document inside a `time` element, using an `xsp:expr` (described in The xsp:expr element on page 15 ).

## Avoid errors, stay well-formed

Once program code starts to mix with XML markup, problems can arise that can result in the XSP document failing to parse. Any XML reserved characters (<, >, &) used in program code must be properly escaped using one of the predefined entities. For example, this code fragment ...

```
if (a < b && c > d) { ... }
```

must be rewritten as follows when added to an XSP document:

```
if (a &lt; b &amp;&amp; c &gt;) { ... }
```

One way to avoid this problem is to use a CDATA section, which signals to the XML parser to ignore well-formedness rules for that section of content:

```
<xsp:logic>
    <![CDATA[
        if (a < b && c > d) { ... }
    ]]&gt;
</xsp:logic>
```

However use of CDATA sections isn't recommended because the parser will ignore any XSP or user elements that appear inside them, instead treating them as plain text rather than XML markup. This is more likely to cause obscure, time-consuming errors rather than making the effort to properly escape the text using entities.

The second possible cause of error also stems from the failure to obey XML well-formedness rules within `xsp:logic` elements. Consider the following malformed example:

```
<search-results>
    <xsp:logic>
        if (firstResult())
        {
            <result id="first">
        } else
        {
            <result>
        }
            <!-- ...result generation code here... -->
        </result>
    </xsp:logic>
</search-results>
```

In this example, the XSP produces a list of search results and treats the first result as a special case by adding an additional attribute. From a programmer's perspective this may look fine, but to an XML parser it clearly breaks the rules. Notice that there are two opening tags for `result` elements, but only one closing tag. This isn't well-formed because the parser treats the program code as plain text. The XSP compilation process has no understanding of program code nor what it actually does.

Fixing this example is simply a matter of balancing the opening and closing elements. Here is one way to do this:

```
<search-results>
   <xsp:logic>
      if (firstResult())
      {
         <result id="first">
            <!--... handle first result..-->
         </result>
      } else
      {
         <result>
            <!--...handle other results...-->
         </result>
      }
   </xsp:logic>
</search-results>
```

Understanding this relationship between the XML markup and program logic embedded in XSP pages is an important step toward becoming productive with this technology.

___

# The xsp:expr element

The `xsp:expr` element is used to wrap an expression whose value should be added directly to the output document. In contrast, the `xsp:logic` element contains code for the Generator. The `xsp:expr` element is therefore equivalent to the `<%= ... %>` expression syntax that fulfills a similar role in JSP.

It's important to emphasize that the `xsp:expr` element should contain Java *expressions* and not *statements*. The following examples illustrate the critical differences:

```
Expression:   System.currentTimeMillis()
Statement:    System.currentTimeMillis();

Expression:   i++
Statement:    i++;

Expression:    new Date()
Statement:    Date d = new Date();

Expression:    "username"
Statement:    String s = "username";
```

Including a statement as the value of an `xsp:expr` element will not result in an error at parse time, but will cause a compilation error because the contents of an expression element are used as a method parameter in the generated source code. The following examples illustrate this:

```
//Legal
XSP :     <xsp:expr>"username"</xsp:expr>
Java:     XSPObjectHelper.xspExpr(contentHandler, "username");

//ILLEGAL!
XSP :     <xsp:expr>System.currentTimeMillis();</xsp:expr>
Java:     XSPObjectHelper.xspExpr(contentHandler, System.currentTimeMillis(););
```

There are multiple overloaded methods on the `XSPObjectHelper` class that differ in
the parameters they accept. For example, there are methods that accept any of the
Java primitive types, as well as, amongst others, a String, a Collection, or an Object.
The Java expression given in the `xsp:expr` element must therefore return one of
these types for compilation of the generated source code to be successful.

`xsp:expr` elements are often used to refer to variables or call methods that are
defined elsewhere in an XSP inside `xsp:logic` elements. The following example
demonstrates this, showing how a loop variable can be used to add content to an
element:

```
<elements>
    <xsp:logic>
        for (int i=1; i<11; i++)
        {
            <element><xsp:expr>i</xsp:expr></element>
        }
    </xsp:logic>
</elements>
```

An easy mistake to make with an `xsp:expr` is trying to add text content to an element.
String values must always be quoted, otherwise they'll be interpreted as references to
a variable during compilation. This will either result in an error (because the variable
isn't defined) or, worse, work successfully, but result in errors that are difficult to trace.

# Generating dynamic elements using xsp:element

XSLT provides two mechanisms for creating elements as output from a transformation.
The first mechanism, literal result elements, is present in the style sheet itself. The
second involves using the `xsl:element` tag, which allows an element to be
dynamically created during the transformation. For instance, its name and/or attributes
can be calculated by the style sheet.

XSP offers the same two mechanisms. User elements, added directly to the XSP page,
are the equivalent of literal result elements. Elements can also be created dynamically
using the `xsp:element`, which works in a similar way to its XSLT equivalent.

```
<xsp:element>
```

```
    <xsp:param name="name"><xsp:expr>"myElementName"</xsp:expr></xsp:param>
    Element content
</xsp:element>
```

The example above shows that creating an element with a dynamically generated name involves using both the `xsp:element` and `xsp:param` elements. The latter defines a parameter, in this case the name of the element whose value is an expression used to calculate the element's name. Content can be added to the element in the usual ways.

Elements created in this fashion can also be associated with a specific namespace and prefix, as the next example shows. Note that *both* the namespace and prefix parameters are required; otherwise an error will occur.

```
<xsp:element prefix="my" uri="http://www.examples.org">
    <xsp:param name="name"><xsp:expr>"myElementName"</xsp:expr></xsp:param>
    Element content
</xsp:element>
```

The example generates the following XML output:

```
<my:myElementName xmlns:my="http://www.examples.org">Element content</my:myElementName>
```

It's also possible to dynamically generate both the namespace URI and the prefix. Instead of adding the `prefix` and `uri` attributes, use additional `xsp:param` elements with the appropriate names. The following is equivalent to the above example:

```
<xsp:element prefix="my" uri="http://www.examples.org">
    <xsp:param name="name"><xsp:expr>"myElementName"</xsp:expr></xsp:param>
    <xsp:param name="prefix">"my"</xsp:param>
    <xsp:param name="uri">"http://www.examples.org"</xsp:param>
    Element content
</xsp:element>
```

---

# Generating dynamic attributes using xsp:attribute

Just as elements can be created dynamically in an XSP page, so can attributes. The `xsp:attribute` element works in a similar way to `xsp:element`, allowing the name and value of an attribute to be dynamically created:

```
<xsp:element>
   <xsp:param name="name"><xsp:expr>"myElementName"</xsp:expr></xsp:param>
   <xsp:attribute name="myAttribute">myAttributeValue</xsp:attribute>
   Element content
</xsp:element>
```

The name of the attribute is defined within the `name` attribute, although in a similar way to `xsp:element`, it can also be defined using an `xsp:param` child element. The attribute value is specified as the element content. This could be either a simple text value or more usefully generated by an `xsp:expr` element.

The `xsp:attribute` tag doesn't have to be used in conjunction with `xsp:element`, though. It can be placed inside any user element, and an attribute is added in the same way. For example, the URL for an image element might be dynamically created using an expression that calls a method defined elsewhere in the XSP page.

```
<image>
   <xsp:attribute name="href"><xsp:expr>calculateImageURL()</xsp:expr></xsp:attribute>
</image>
```

If the generated attribute is associated with a specific namespace, this can be indicated by using additional `prefix` and `uri` attributes or `xsp:param` elements, similar to the method used for `xsp:element`. Again, it is an error to only define one of these.

---

# Creating comments and processing instructions

The `xsp:comment` and `xsp:pi` elements are used to create comments and processing instructions.

Creating a comment is quite straightforward. Any text provided as the child of an `xsp:comment` element is turned into an XML comment:

```
<xsp:comment>This is a comment</xsp:comment>
```

This then becomes:

```
<!-- This is a comment -->
```

Note that *any* nested markup is ignored, although any text will be added to the comment.

Creating a processing instruction is similar to creating a dynamic element or attribute. The `xsp:pi` element should have a nested parameter that identifies the processing instructions target. The remaining content of the `xsp:pi` element is evaluated as usual. Here's a simple example:

```
<xsp:pi target="myApplication">
   <xsp:expr>"param1=value, param2=value, generatorTimestamp=" +
   System.currentTimeMillis()</xsp:expr>
</xsp:pi>
```

The output looks like this:

```
<?myApplication param1=value, param2=value, generatorTimestamp=1017407796870?>
```

The target for the processing instruction can also be generated automatically by creating it within an `xsp:param` element, as the following example demonstrates:

```
<xsp:pi>
   <xsp:param name="target"><xsp:expr>"myApplication"</xsp:expr></xsp:param>
   <xsp:expr>"param1=value, param2=value, generatorTimestamp=" +
   System.currentTimeMillis()</xsp:expr>
</xsp:pi>
```

# Syntax summary

This section has provided a detailed walk-through of the XSP syntax, illustrating the use of each element with some simple examples. Here's a summary of each of the elements in the XSP syntax and the functionality they provide:

- `xsp:page` -- the root element of an XSP document, which must contain only a single user element
- `xsp:structure`, `xsp:include` -- allow imports of additional Java classes into the compiled version of the XSP
- `xsp:logic` -- allows blocks of additional programming code to be included in the compiled version of the XSP; this can include member variables, methods, or application logic
- `xsp:expr` -- allows Java expressions to be evaluated and their value added to a document
- `xsp:element` -- allows elements to be dynamically created by the XSP; elements can be created with any name and can be associated with any namespace and prefix
- `xsp:attribute` -- allows attributes to be dynamically added to elements; attributes

can be created with any name and value, and can be associated with any namespace

- `xsp:comment` -- allows comments to be added to the generated document
- `xsp:pi` -- allows processing instructions to be dynamically created and added to the generated document

These syntax elements are the basic building blocks of an XSP. It's possible to create dynamic XML content using these elements alone; however, there is an additional important aspect to XSP that allows this syntax to be extended with custom elements. This feature, known as logicsheets, is discussed in the next section.

## Section 4. Logicsheets

## What are Logicsheets?

Although the ability to generate dynamic XML content with XSP is an important feature, there are some disadvantages to mixing program code with XML markup. As explained in Avoid errors, stay well-formed on page 13 , the mixture of the different syntax rules can sometimes create unforeseen problems. The other major downside to this approach is that placing large amounts of application logic in an XSP page makes both the document and the source code difficult to maintain.

These are difficulties that confront any technology that mixes code and markup in this way. In Java Server Pages these problems have been alleviated by introducing the concept of tag libraries, which allow Web developers to add custom tags to JSP pages. The JSP container associates these tags with an implementation created by a Java programmer. This provides a good separation between both the development roles and the underlying code.

Cocoon 2 offers a technology that's similar to JSP tag libraries -- *logicsheets*. A logicsheet is a library of custom elements that can be added to XSP pages. However, unlike JSP tag libraries, which are implemented as Java classes, Cocoon 2 logicsheets are implemented using XSLT transformations. These transformations introduce additional blocks of code and/or XSP markup into an XSP document, thereby extending the capabilities of the resulting Generator class.

A key advantage of logicsheets is that the original documents are much clearer because the code has been removed, but these same logicsheets can be reused across multiple XSP documents. This avoids the need to duplicate code in several places.

The remainder of this section demonstrates how to use logicsheets in an XSP document and reviews the main logicsheets that are built into Cocoon 2. The (Creating logicsheets on page 26 ) section discusses how logicsheets are implemented and how custom logicsheets can be created.

---

## Using logicsheets

Every logicsheet is associated with a particular namespace. Using a logicsheet involves nothing more than declaring the appropriate namespace in the XSP document, and then adding elements from that namespace as necessary

For example, Cocoon 2 includes a *utility* logicsheet that can add the current time to a document. Here's a simple XSP page that uses this functionality:

```
<xsp:page language="java"
```

```
            xmlns:xsp="http://apache.org/xsp"
            xmlns:util="http://apache.org/xsp/util/2.0">
    <clock>
        <day><util:time format="EE"/></day>
        <month><util:time format="MMMM"/></month>
        <year><util:time format="yyyy"/></year>
        <time><util:time format="HH:mm:ss 'on' dd/MM/yyyy"/></time>
    </clock>
</xsp:page>
```

Note that the page declares the utility namespace,
`http://apache.org/xsp/util/2.0`, and uses a single additional element without
inserting any additional Java code. When evaluated, this page produces:

```
<clock>
    <day>Fri</day>
    <month>March</month>
    <year>2002</year>
    <time>15:14:27 on 29/03/2002</time>
</clock>
```

The elements from the `util` namespace do not appear in the output document;
instead, they are used in the XSP compilation process to create code that generates
dates and times in a variety of formats.

Other logicsheets can be used in exactly the same way. An XSP document can make
use of elements from any number of logicsheets, allowing complex behaviors to be
built up from reusable functionality.

# Built-in logicsheets

Cocoon 2 provides a number of predefined logicsheets that offer a great deal of useful
functionality without having to write Java code. These predefined logicsheets can be
loosely categorized by the type of functions they provide:

- Environmental logicsheets -- provide access to the Cocoon processing environment
  (e.g., the request and the response)
- Utility logicsheets -- multi-purpose utility code (e.g., file includes, logging, sending
  mail, etc.)
- Data manipulation -- access to data validation and database-related functionality

The logicsheets included in each of these categories are summarized in the rest of this
section. As an open-source project, the complete list of built-in logicsheets is liable to
change and additional functionality may be added to existing logicsheets. Consult the
Cocoon 2 documentation for more information. A local copy of this can be found at:
`http://localhost:8080/cocoon/documents/userdocs/xsp/index.html` or

you can check the Cocoon project Web site (see Resources on page 30 ) for the latest information.

# Environmental logicsheets

There are four logicsheets in this Environmental category, each of which provides access to particular aspects of the processing environment associated with a Web request.

The *request* logicsheet provides access to properties of the request, including request parameters, the request method (e.g., GET, POST, etc.), and the request headers. This logicsheet is particularly useful when aspects of the request parameters are to be used to alter the generation of the output document.

The *response* logicsheet provides limited access to the HTTP response associated with the current request; it only provides access to response headers. An XSP document cannot perform an include or forward in the same way as a Java Servlet or JSP page because of the separation of concerns that is a core part of the Cocoon 2 architecture. This functionality is described in the sitemap; an XSP page generates XML content and does not direct processing.

The *session* logicsheet provides access to HTTP session information, including the ability to create and delete sessions, as well as add and remove session attributes. This functionality is obviously most useful in Web applications that must maintain a user session for context. Session management in Cocoon 2 is directly equivalent to its JSP counterpart.

The *cookie* logicsheet provides cookie maintenance functionality, such as adding and removing cookies, allowing preferences to be stored in the user's browser.

The functionality provided by these logicsheets is an analogue of that provided by the implicit objects associated with a JSP page (i.e., the request and response objects) and is drawn directly from the HTTP Servlet API.

# Utility logicsheets

There are three logicsheets in this category, each of which provides some simple utility functions.

The previously discussed *utility* logicsheet, in addition to having the ability to get the current time in multiple formats, allows the XSP author to incorporate XML content directly into an XSP document from either a URL or file. This content can be added either as XML (i.e., elements and attributes) or as plain text.

The *log* logicsheet allows log messages to be written from an XSP generator as it processes a user request -- a useful feature when attempting to debug complex XSP

pages. Assuming that the log namespace (`http://apache.org/xsp/log/2.0`) has been defined, the following writes a debug message to the default Cocoon 2 logfile.

```
<log:debug>This is a debug message from an XSP generator</log:debug>
```

There are additional `info`, `warn`, `error`, and `fatal-error` elements for writing other kinds of log statements.

The *sendmail* logicsheet contains a single useful element that provides access to the JavaMail API from an XSP page. The following example demonstrates this and again assumes that the appropriate namespace has been defined -- in this case `http://apache.org/cocoon/sendmail/1.0`:

```
<sendmail:send-mail from="myemail@email.com" to="user@user.com" smtphost="smtphost@emai
    <xsp:param name="subject"><xsp:expr>"The subject of this email..."</xsp:expr></xsp:pa
    <xsp:param name="body"><xsp:expr>"The body of this email..."</xsp:expr></xsp:param>
</sendmail:send-mail>
```

Obviously, the attributes need to be altered to reflect the actual "from" and "to" e-mail addresses, SMTP server, etc. The subject and body of the e-mail can be dynamically generated if required. If the e-mail addresses must also be dynamically constructed (e.g., read from a session parameter) then an `xsp:param` element can be used instead of the attribute. The name attribute of this parameter should reflect the variable it's describing.

---

## Data manipulation

There are two logicsheets that fall into this category, one of which provides significantly more functionality than the other.

The *form validator* logicsheet is never actually used in isolation. It provides a clean interface to the Cocoon Form Validator Action. This action is capable of carrying out basic validation operations on data sent to a Cocoon application from HTML forms, including checking minimum and maximum values for an integer, checking the size of a string, and checking that certain parameters are provided. One notable piece of functionality is the ability to test whether a posted variable matches a given regular expression.

The results of the validation are stored in a request parameter. The *validator* logicsheet provides the means to interpret these results from within an XSP page. This allows error messages to be dynamically generated for users.

Once the data has been confirmed as valid, the next step is to store that data in a database. The *esql* logicsheet provides the means to do this and a great deal more,

including selecting, deleting, and updating a database. Essentially the esql logicsheet provides a means to embed SQL statements directly into an XSP document. The logicsheet then generates the appropriate JDBC code to carry out the SQL operations, simplifying the manipulation and retrieval of database data using Cocoon 2.

Once you've learned how to use logicsheets in XSP documents and reviewed the range of logicsheets already available in Cocoon 2, your next step is to understand how to write custom logicsheets.

## Section 5. Creating logicsheets

## How logicsheets work

Several important facts about logicsheets have already been discussed. First, a logicsheet is associated with a specific namespace and elements from this namespace are associated with specific functionality -- e.g., generating a timestamp or writing a log message. Second, logicsheets are implemented using XSLT transformations, which add Java code to an XSP document to create the desired functionality.

Another piece of the puzzle involves the process of transforming an XSP document into Java source code (see The XSP compilation process on page 7 ), which provides hooks that allow additional transformation steps to be inserted before the actual Java code is created. These hooks are where logicsheets come into play.

When Cocoon 2 first processes an XSP document, it identifies all the namespaces in that document and checks each of them against a configuration file. This configuration associates namespaces with transformations (i.e., a logicsheet). If a match is found, then the additional transformation is carried out. Cocoon 2 will repeatedly apply this step until the only namespaces left in the XSP document are that of the XSP syntax or those which have not been mapped to a transformation. At this point Cocoon 2 carries out the last transformation to generate the final Java code.

Therefore, logicsheets essentially allow sections of code to be factored out of an XSP document and stored in a separate XSLT transformation. The code extracted from the XSP document is replaced by an element or sequence of elements in a custom-defined namespace. When Cocoon 2 processes the XSP document, it recognizes that this namespace is associated with a logicsheet and executes the logicsheet transformation. It is the job of the logicsheet to re-insert the original code back into the document while preserving the rest of the document.

---

## Writing logicsheets

Writing a Cocoon logicsheet is a simple process. The most important thing to remember is that because logicsheet transformations can happen in any order, each logicsheet needs to preserve anything in the input document that it doesn't understand. In other words, the basic template for a logicsheet is an *identity transformation*. This is a transformation that simply copies its input directly to its output unchanged. A sample *identity transformation* has been provided as a starting point (See Tools on page 3 for a link to the source code for this tutorial). This transformation also includes a template to allow additional Java imports and methods to be defined (see the comments in the style sheet for more information).

This basic style sheet can then be extended, adding new templates that match the elements for which this logicsheet provides the functionality. The body of these templates will include XSP syntax that describes the functionality of a given element.

As a simple demonstration, a sample logicsheet has been created that implements the same facility as the `util:time` element. Take a look at the *sample time* logicsheet (see Tools on page 3 ). The important part of this logicsheet is presented below:

```
<xsl:template match="time:time">
   <xsp:logic>
      SimpleDateFormat timeFormat = new SimpleDateFormat("<xsl:value-of select="@format"
   </xsp:logic>

   <xsp:expr>
      timeFormat.format(java.util.Calendar.getInstance().getTime())
   </xsp:expr>
</xsl:template>
```

This single template matches `time` elements and replaces them with code wrapped in `xsp:logic` and `xsp:expr` elements to create the date. The required date format is indicated by the user through the `format` attribute on this element.

## Configuring a logicsheet

Cocoon 2 maintains the mapping between namespaces and the associated logicsheets (style sheets) in its XML configuration file, which can be found at `$CATALINA_HOME/webapps/cocoon/cocoon.xconf`. (Note: substitute `cocoon` for the actual Web application directory if Cocoon 2 is installed in a different directory.)

Search for the `xsp-language` element in the configuration file, and note that it contains a child element called `target-language`. It is within this element that all logicsheets should be declared. It's possible to see that the built-in logicsheets are already predefined. Here's the entry for the *util* logicsheet:

```
<builtin-logicsheet>
   <parameter name="prefix" value="util"/>
   <parameter name="uri" value="http://apache.org/xsp/util/2.0"/>
   <parameter name="href" value="resource://org/apache/cocoon/components/language/markup
</builtin-logicsheet>
```

Notice that the `builtin-logicsheet` element has several parameters. The `prefix` parameter defines the namespace prefix that is commonly associated with or recommended for use within this logicsheet. The second parameter, `uri`, identifies the namespace URI for this logicsheet. Having identified the namespace associated with the logicsheet, the `href` parameter identifies the place where the logicsheet can be found. This logicsheet will be used whenever elements from the indicated namespace are found in an XSP page.

While the built-in logicsheets are bundled into a .jar file along with the Cocoon classes,

user-defined logicsheets can be stored in the file system and identified by the appropriate `resource://` URL. Resource URLs indicate files that live in the Cocoon WEB-INF/classes directory. So having created the sample logicsheet, called time.xsl, one way to store and reference it is:

```
$CATALINA_HOME/webapps/cocoon/WEB-INF/classes/logicsheets/time.xsl
```

Then the `href` parameter should be configured as follows:

```
<parameter name="href" value="resource://logicsheets/time.xsl"/>
```

Once changes have been made to the configuration file, Tomcat must be restarted for the changes to take effect.

---

## Logicsheet development tips

Here are three useful tips and design guidelines to bear in mind when creating new logicsheets:

**Use helper classes**: Although creating a logicsheet will factor program code out of your XSP pages, this code is still locked away in an XSLT style sheet. This makes the code difficult to maintain and test independently of Cocoon. A good design tip is to place as much code as possible in *helper* classes. This reduces the implementation of a logicsheet to simply collecting parameters and then invoking the required helper methods. The built-in logicsheets follow this guideline and have a single helper class per logicsheet.

**Create logicsheet "macros"**: A macro is simply a quick way to record and replay a series of operations. XSP pages in a particular application may repeatedly create the same XML structures or invoke other logicsheets in a consistent way. It's useful to factor out common blocks in XSP pages to create higher-level elements (i.e., macros) that describe this repeated structure. Because Cocoon 2 recursively applies logicsheets until all namespaces have been handled, the implementation of one logicsheet can add elements from another. This kind of grouping can greatly reduce XSP page complexity.

**Think about the user**: When considering what tags to provide in a logicsheet, consider the end users. What flexibility will they require? For example, what parameters can be passed to a tag? Can parameters only be passed through attributes or nested `xsp:param` elements? The latter is more flexible as it allows the user to create parameter values dynamically using `xsp:expr`, which cannot be used inside an attribute. Also consider how to name a tag. Give it a clear and memorable name so it's obvious what it does. Not everyone will be happy to dig through the implementation of a logicsheet to find this out. An obvious extension of this advice is to provide supporting documentation for a logicsheet; this enables a user to quickly find out what a tag does

and how it can be used.

# Section 6. Summary and resources

## Where next?

This tutorial covered the details involved in working with XML Server Pages, the Cocoon technology for generating dynamic XML content.

The introduction contrasted XSP with JSP. Although these are largely equivalent technologies, XSP is about generating data while JSP is mainly used to create presentations.

The foundations of the XSP technology were then described. The process of taking an XSP document and transforming it into Java source code was illustrated. The specifics of how to plug XSP pages into Cocoon 2 pipelines was also covered. Examples were given to demonstrate this process from end to end, including instructions for examining the source code generated by Cocoon based on the XSP document.

A review of the details of the XSP syntax demonstrated how to mix XML markup with Java code within an XSP document, including the ability to dynamically create elements, attributes, comments, and processing instructions. After an examination of how this mixture of code and markup is put to use, the concept of logicsheets was then discussed.

Logicsheets are XSLT transformations that can be hooked into the code generation process to allow the creation of custom tag libraries. These make XSP pages easier to work with, reducing the need to directly embed code. The benefits of logicsheets were reviewed, as were the built-in logicsheets that are bundled with Cocoon 2. Finally instructions were given on how to create new logicsheets, including a number of design guidelines that should be followed to gain the most from the technology.

The next tutorial in this series will take a closer look at how to integrate Cocoon 2 with a database, including the generation of XML documents from database data.

---

## Resources

- Read the first tutorial in this series, *Introduction to Cocoon 2*.
- Follow the continuing development of *Apache Cocoon* at the official project home page.
- The *cocoon-users mailing list* is a great source of advice from expert Cocoon users and developers.
- Read the *user documentation* for more information on the Cocoon logicsheets and XSP technology.
- The *Cocoon Center* is compiling a number of useful tutorials and tips for Cocoon developers.
- *AxKit* is a Perl-based XML application server that supports XSP. Read the *AxKit XSP*

*guide* for more information on building XSP pages using Perl rather than Java.

- For more information on JSP, read Noel Bergman's *Introduction to JavaServer Pages Technology* (*developerWorks*, August 2001).

- Get some great advice on design tag libraries from Bergman's *JSP taglibs: Better usability by design* article. (*developerWorks*, December 2001)

- Read *Understanding SAX* by Nick Chase for a great introduction to the SAX API. (*developerWorks*, September 2001)

- Read *Transforming XML*, a tutorial by Doug Tidwell, for more information on generating HTML, SVG, and PDF using XSLT. (*developerWorks*, May 2000)

- Finally, take a look at *IBM WebSphere Studio Application Developer*, an easy-to-use, integrated development environment for building, testing, and deploying J2EE applications, including generating XML documents from DTDs and schemas.

---

# Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

---

## Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.