XSPSyntax

Describes XML syntax for XSP

To use XSP tags you must include the XSP namespace. Example:

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

xsp:page

Is the root element of every XSP document. Inside this tag you put all the others tags you will need. The use of this tag is mandatory.

Note that, this is *not* the root element of the generated document. During the translation process – in which the XSP page is turned in source code for a particular language – all elements in the XSP namespace are replaced with programming language structures, i.e. imports, code blocks, statements, expressions, etc.

All XSP pages will end up generating a subclass of `XSPGenerator`. See the XSPEnvironment for more details.

The XML Namespaces for all Logicsheets referenced by this XSP page should be declared here, along with the XSP namespace itself. You should avoid using the default namespace on XSP elements as you'll have to redeclare it on all of your own elements should they not be in a namespace of their own.

The page element should have a `language` attribute, indicating the programming language used within that page. Therefore, while XSP may be language neutral, each page is tied to a *specific* language.

The page element may contain several children, but can only have a *single* child which is not in the XSP namespace. This *user* element will become the root of the generated document.

All markup within the user element is generated by the `XSPGenerator.generate()` method.

Children: single user element, `xsp:structure`, `xsp:logic`, `xsp:init-page`

xsp:structure && xsp:include

These elements are used together to allow additional program modules/libraries/classes, required by the code in the rest of the code, to be included in the generated source code.

In Java these elements are used to signify import statements for the generated class. You should include imports for any classes in the Java API, or those from your own class libraries that are required by code in the XSP.

Children: (xsp:structure) `xsp:include`; (xsp:include) none

xsp:init-page

Is a top level tag. It fired just before the call of the **startDocument()** function.

This element can be used to execute any user-defined initializations just before the start of the generation of the target document. Example: can be used to define local variables.

Since it is outside the document scope, this tag **cannot** be used to output anything to the client like headers or footers. Also that means that all the code put here may be only in the target programming languaje (Java, Javascript or Python).

Children: none

xsp:exit-page

Is a top level tag. It fired just after the call of the **endDocument()** function.

This element is used to contain any user-defined necessary code after the end of the generation of the target document. Example: can be used to clean up local variables.

Since it is outside the document scope, this tag **cannot** be used to output anything to the client like headers or footers. Also that means that all the code put here may be only in the target programming languaje (Java, Javascript or Python).

Children: none

xsp:logic

These elements are used to contain blocks of code. This may be method declarations or just sequences of application logic.

xsp:logic elements that appear outside of the *user* element are deemed to be class level methods and declarations – i.e. static methods, member variables, etc.

xsp:logic elements used elsewhere result in the addition of these code blocks to the `generate()` method.

To avoid escaping all of the programming code to ensure that it's well-formed, it's possible to wrap the code in CDATA sections. You must ensure however that the sequence]]> does not occur within the code block – this is a rare occurence in any rate.

The xsp:logic element is much friendlier than the equivalent <% %> JSP syntax. For example in JSP it's common to see blocks of the form:

```
<% if (someTest()) { %>
  <b>Success!</b>
<% } else { %>
  <b>Fail!</b>
<% } %>
```

In an xsp:logic element this would look like:

```
<xsp:logic>
if (someTest()) {
  <b>Success!</b>
} else {
  <b>Fail!</b>
}
</xsp:logic>
```

There's no need for the additional escaping used in JSP to separate program code from elements designed for the output, as the XML parser can already distinguish between text content and child elements.

The XSLT transform used to create the code can then handle these differently: text contained within an xsp:logic element is treated as Java code, whereas elements not in the XSP namespace are substituted by program code that will fire the required SAX events.

An xsp:logic element can contain xsp:expr elements, xsp:content elements, or elements that will be passed directly to the generated document. Ensure however that the content remains well-formed.

For example the following is not legal:

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

The above code, which is generating some results, e.g. from a database query or search, and is attempting to treat the first result differently to subsequent results by adding an additional attribute.

However if you attempt to load this XSP page you'll get a 404 Not Found error. Digging into the `cocoon.log` will show that an exception has been thrown whilst parsing the XSP page:

```
org.xml.sax.SAXException: Stopping after fatal error:
The element type "result" must be terminated by the matching end-tag "</result>".
```

One way to handle this would be:

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

The file is now well-formed.

xsp:expr

An xsp:expr element is used to signify Java expressions, and is equivalent to the <%= %> syntax in JSP.

The contents of an xsp:expr element is passed directly to the XSPObjectHelper.xspExpr(contentHandler, ...) method. Therefore it's an *expression* and not a *statement*. As it is not a statement there is no need to terminate the text content with a semi-colon.

Can be used within both user elements and other XSP elements.

If you want to use `xsp:expr` inside an `xsp:logic`, embed it into an `xsp:content` element.

Children: none

xsp:element

XSLT provides two mechanisms to create elements for output from a transformation. The elements can either be present directly in the stylesheet (literal result elements) or generated dynamically using xsl:element. This allows the precise element to be created to be totally under the control of the stylesheet.

XSP provides a directly equivalent mechanism. We've already seen that user elements can be added to the XSP page and these are sent directly to the output. These are the equivalent to XSL literal result elements. Dynamically created elements can be produced using the xsp:element.

The name of the dynamically created element should be defined by including an xsp:param child of this element. This xsp:param element must have the value of it's name attribute be "name". The contents of this element must be an xsp:expr element which contains the code that will generate the element name, or "someString" (including quotes).

A namespace can be assigned by including two additional parameters with the name "uri" and "prefix". It is an error to only provide one of these.

Children: `xsp:param`, `xsp:attribute`.

xsp:attribute

Used to create an attribute. Similar to xsl:attribute.

Should contain an xsp:param element whose name is "name". This indicates the name of the element. A namespace can be assigned by including two additional parameters with the name "uri" and "prefix". It is an error to only provide one of these.

Children: `xsp:param`, `xsp:expr`

Example:

```
<xsp:logic>
        String val = "a radio button value";
        <input type="radio" name="mod" onclick="muestraAreas()">
                <xsp:attribute>
                        <xsp:param name="name">value</xsp:param>
                        <xsp:expr>val</xsp:expr>
                </xsp:attribute>
        </input>
</xsp:logic>
```

xsp:content

Adds character content to the output. Contents is interpreted as a string. If an element is added as a child then it is passed through as a literal result element.

xsp:pi

Used to create a processing instruction.

Should contain an xsp:param element whose name is "target". This indicates the target of the processing instruction. The content can described using the content of the xsp:pi element, or nested xsp:expr elements.

These are concatenated to form the final content.

xsp:comment

Used to create a comment inside the generated source code.

xsp:param

Used to describe parameters.

Should have a name attribute indicating the name of the parameter.

Using functions

You can also add functions inside your XSP file, like in this example:

```
<?xml version="1.0"?>
<xsp:page xmlns:xsp="http://apache.org/xsp">
	<xsp:structure/>
	<xsp:logic>
		// This is a sample function
		void showValue(String object) throws SAXException {
			// This line is necessary only if generating XML data.
			AttributesImpl xspAttr = new AttributesImpl();
			<p>This is the value of my string: <xsp:expr>object</xsp:expr></p>
		}
	</xsp:logic>
	<html>
		<xsp:logic>showValue("hello");</xsp:logic>
		<xsp:logic>showValue("bye");</xsp:logic>
	</html>
</xsp:page>
```

Output of this sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
	<p>This is the value of my string: hello</p>
	<p>This is the value of my string: bye</p>
</html>
```

Attachment: Log.xsl