Logicsheet Concepts (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. Index

This document introduces logicsheet design and writing principles:

2. Logicsheets

A *logicsheet* is an XML filter used to translate user-defined, dynamic markup into equivalent code embedding directives for a given markup language.

Logicsheets lie at the core of XSP's promise to separate logic from content and presentation: they make dynamic content generation capabilities available to content authors not familiar with (and not interested in) programming.

Logicsheets are *not* programming-language independent. They translate dynamic tags to *actual code* enclosed in code-embedding directives. Fortunately, this dependency can be alleviated by judiciously using helper classes.

Logicsheets are used to translate *dynamic tags* into markup language code-embedding directives. Thus, for example, the dynamic tag:

<util:time-of-day format="hh:mm:ss"/>

would be transformed by the *util* logicsheet into an equivalent XSP expression:

<xsp:expr> SimpleDateFormat.getInstance().format(new Date(), "hh:mm:ss") </xsp:expr>

Note that the output of logicsheet processing is *not* final code, but rather *code-embedding markup language directives* (*<xsp:expr>* in this case).

Logicsheets can be applied in sequence so that it's possible for one logicsheet to produce dynamic tags further processed by another logicsheet. Thus, for example:

<util:time-of-day> <util:param name="format"> <request:get-parameter name="time-format" default="hh:mm:ss"/> </util:param> </util:time-of-day>

would be transformed by the *util* logicsheet into:

<xsp:expr> SimpleDateFormat.getInstance().format(new Date(), <request:get-parameter name="time-format" default="hh:mm:ss"/>) </xsp:expr>

which would be transformed by the *request* logicsheet, in turn, into:

<xsp:expr> SimpleDateFormat.getInstance().format(new Date(), XSPRequestHelper.getParameter("name", "hh:mm:ss")) </xsp:expr>

Note in the examples above that dynamic tags can be "overloaded" in the sense that they can take as parameters either *constant attribute values* or *nested parameter elements*:

<!-- Parameter "format" known at page authoring time --> <util:time-of-day

format="hh:mm:ss"/> <!-- Parameter "format" known at request time --> <util:time-of-day> <util:param name="format"> <request:get-parameter name="time-format" default="hh:mm:ss"/> </util:param> </util:time-of-day>

This means that logicsheets must be able to cope with constant strings, complex expressions and nested parameter processing. Also, logicsheets must be capable of reporting parameter value errors and, possibly, halt code generation altogether.

In order to minimize this complexity (and its associated debugging nightmares!), properly designed logicsheets typically make use of **helper classes**.

3. Logicsheet Helper Classes

A *helper class* is a Java class containing a collection of *static* methods whose arguments correspond (one-to-one) with their dynamic tag counterparts.

Consider the following dynamic tag use-case:

<sql:create-connection name="demo"> <sql:jdbc-driver> oracle.jdbc.driver.OracleDriver </sql:jdbc-driver> <sql:connect-url> jdbc:oracle:thin:@localhost:1521:ORCL </sql:connect-url> <sql:user-name> <request:get-parameter name="user"/> </sql:user-name> <sql:password> <request:get-parameter name="password"/> </sql:password> </sql:create-connection>

A brute-force logicsheet template may be implemented (in XSLT, as discussed below) as:

<xsl:template match="sql:create-connection"> <!-- *** Argument collection skipped for the sake of brevity *** --> <xsp:logic> { Class.forName(<xsl:copy-of select="$jdbc-driver"/>).newInstance(); Connection myConnection = DriverManager.getConnection(<xsl:copy-of select="$connect-url"/>, <xsl:copy-of select="$user-name"/>, <xsl:copy-of select="$password"/>); Session mySession = request.getSession(true); Connection previousConnection = (Connection) mySession.getAttribute("connection." + <xsl:copy-of select="$name"/>); if (previousConnection != null) { previousConnection.commit(); previousConnection.close(); } mySession.setAttribute("connection." + <xsl:copy-of select="$name"/>, myConnection) } </xsp:logic> </xsl:template>

This approach has a number of drawbacks.

- Even when using enclosing braces around the *<xsp:logic>* section, there's always the risk that the page author (or another logicsheet!) has previously defined variables called myConnection, previousConnection or mySession. This will result in multiply-defined variable compilation errors
- Parameter values (like $name) cannot be safely used more than once. As much as they can come from harmless string constants, they can also come from complex expressions involving method/function calls which can have unpredictable side-effects should they be called more than once in the current context
- If another logicsheet (or the page author) has imported a class called Connection the generated code will produce an ambiguous class definition compiler error

It's here that helper classes come to the rescue. By moving all the above logic to a static method createConnection in helper class SQLHelper, we can now rewrite (and simplify!) our logicsheet to read:

<xsl:template match="sql:create-connection"> <!-- *** Argument collection skipped for the sake of brevity *** --> <xsp:logic> SQLHelper.createConnection(<xsl:copy-of select="$name"/>, <xsl:copy-of select="$connect-url"/>, <xsl:copy-of select="$user-name"/>,

<xsl:copy-of select="$password"/>, request); </xsp:logic> </xsl:template>

This simple approach brings several benefits:

- Safer parameter evaluation, with no unpredictable side effects
- Programming language-independence: expressions calling "native" Java code tend to have the same syntax in all programming languages, thus reducing the need to maintain multiple versions of the same logicsheet
- Simpler debugging: syntax errors can now be traced to bad parameter values, rather than invalid code
- Easier logic maintenance: it takes places at the helper class level, rather than at the logicsheet's
- Reduced generated code size.

4. Logicsheets and Objects

Though not required to do so, each logicsheet typically deals with a single *object type*.

What objects must be manipulated by means of logicsheets depends mostly on the calling *host environment*.

Thus, for example, when Cocoon is used as a servlet, XSP pages need access to the underlying servlet engine objects: request, response, session, servlet config, etc.

In general, in order to enable dynamic content generation for each host environment, logicsheets must be written that provide markup-based access to its objects and methods:

<request:get-parameter name="part-number"/> <response:send-redirect location="error-page.xml"/> <cookie:create name="user-preferences"/>

In general, for each object type required by a server pages application a helper class should be written that:

- Provides access to the object's methods and services
- Provides convenience methods to wrap values returned by object methods as XML

Within this discipline, *each object type must define a separate, identifying namespace*.

Finally, logicsheets may require a *preprocessor* that augments its input document with extra information prior to markup transformation.

As an example of logicsheet preprocessing, consider an xbean logicsheet providing services similar to the JSP's intrinsic <jsp:useBean> tag:

. . . <xbean:use-bean id="myCart" class="com.acme.cart.CartBean" scope="session"> <xbean:set-property property-name="type" property-value="promotion"/> <xbean:set-property property-name="customer" parameter-value="custid"/> </xbean:use-bean> . . . Hello <xbean:get-property bean-id="myCart" property-name="customerName"/>, welcome back! You have the following discounts: <xbean:get-property bean-id="myCart" property-name="discount"/> . . .

In this case, code to be emitted by the logicsheet will vary wildly depending on whether a given bean property is indexed, multivalued or object-typed; different conversions and traversing code may be needed for each property based on its Java and bean types.

A logicsheet preprocessor could introspect the given bean at code generation time to augment the input document with additional information as to make it possible for an XSLT-based logicsheet to emit appropriate code:

. . . <xbean:use-bean id="myCart" class="com.acme.cart.CartBean" scope="session">

<xbean:set-property property-name="type" property-value="promotion" java-type="String" is-indexed="false" /> <xbean:set-property property-name="customer" parameter-value="custid" java-type="String" is-indexed="false" /> </xbean:use-bean> . . . Hello <xbean:get-property bean-id="myCart" property-name="customerName" java-type="String" is-indexed="false" />, welcome back! You have the following discounts <xbean:get-property bean-id="myCart" property-name="discount" java-type="float" is-indexed="true" /> . . .

Using this information, the logicsheet can decide, for a given bean property, whether to generate a simple String.valueOf() or an indexed loop displaying individual property values.

Logicsheet preprocessor is still unimplemented. Preprocessing may be performed as well in XSLT by using *extension functions*. Logicsheet preprocessing is meant to be used in those cases where the XSLT processor being used by Cocoon does not support XSLT extensions. (As of this writing, only Xalan is known to support XSLT extensions).

5. Logicsheets and XSLT

XSLT-based transformation is clearly the obvious choice for implementing logicsheets.

XSLT provides all the capabilities needed for dynamic markup transformation as well for final code generation (described in Logicsheet Code Generators).

In fact, logicsheet transformations require only a subset of XSLT much more general capabilities:

- Transforming an input element into other element(s) and nested text (code)
- Collecting and validating parameters as variables
- Substituting variables as either text or nested elements

Paradoxically, though, the XSLT and XPath expressions required to perform these apparently simple tasks can easily become too verbose, and hard-to-read.

This real disadvantage doesn't stem from XSLT not being appropriate or powerful enough to perform the required transformations, but rather from its directives being too low-level for this particular task.

In a classical XML spirit, the solution to this problem is found in the definition of a higher-level language specifically targeted to code-generation transformations. Documents written in this language are transformed into "regular" XSLT stylesheets and subsequently applied to input documents for code generation.

Such language is described in detail below, under The SiLLy Logicsheet Language .

In general, XSLT logicsheets **must** preserve all markup not recognized by its own templates:

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

Parameters should be passed to dynamic tags by means of *both* attributes and nested elements. Also, dynamic tag parameters must be accepted *both* as constant strings and as (potentially complex) expressions. These two requirements are illustrated in the above *util* logicsheet example:

<!-- Parameter "format" known at page authoring time --> <util:time-of-day format="hh:mm:ss"/> <!-- Parameter "format" known at request time --> <util:time-of-day> <util:param name="format"> <xsp:expr> request.getParameter("format"); </xsp:expr> </util:param> </util:time-of-day>

In order to support this, a number of utility templates have been defined:

<!-- Utility templates --> <xsl:template name="get-parameter"> <xsl:param name="name"/>

<xsl:param name="default"/> <xsl:param name="required">false</xsl:param> <xsl:variable name="qname"> <xsl:value-of select="concat($prefix, ':param')"/> </xsl:variable> <xsl:choose> <xsl:when test="@*[name(.) = $name]"> "<xsl:value-of select="@*[name(.) = $name]"/>" </xsl:when> <xsl:when test="(*[name(.) = $qname])[@name = $name]"> <xsl:call-template name="get-nested-content"> <xsl:with-param name="content" select="(*[name(.) = $qname])[@name = $name]"/> </xsl:call-template> </xsl:when> <xsl:otherwise> <xsl:choose> <xsl:when test="string-length($default) = 0"> <xsl:choose> <xsl:when test="$required = 'true'"> <xsl:call-template name="error"> <xsl:with-param name="message"> [Logicsheet processor] Parameter '<xsl:value-of select="$name"/>' missing in dynamic tag <<xsl:value-of select="name(.)"/>> </xsl:with-param> </xsl:call-template> </xsl:when> <xsl:otherwise>""</xsl:otherwise> </xsl:choose> </xsl:when> <xsl:otherwise><xsl:copy-of select="$default"/></xsl:otherwise> </xsl:choose> </xsl:otherwise> </xsl:choose> </xsl:template> <xsl:template name="get-nested-content"> <xsl:param name="content"/> <xsl:choose> <xsl:when test="$content/*"> <xsl:apply-templates select="$content/*"/> </xsl:when> <xsl:otherwise>"<xsl:value-of select="$content"/>"</xsl:otherwise> </xsl:choose> </xsl:template> <xsl:template name="get-nested-string"> <xsl:param name="content"/> <xsl:choose> <xsl:when test="$content/*"> "" <xsl:for-each select="$content/node()"> <xsl:choose> <xsl:when test="name(.)"> + <xsl:apply-templates select="."/> </xsl:when> <xsl:otherwise> + "<xsl:value-of select="translate(.,'	

',' ')"/>" </xsl:otherwise> </xsl:choose> </xsl:for-each> </xsl:when> <xsl:otherwise>"<xsl:value-of select="$content"/>"</xsl:otherwise> </xsl:choose> </xsl:template> <xsl:template name="error"> <xsl:param name="message"/> <xsl:message terminate="yes"><xsl:value-of select="$message"/></xsl:message> </xsl:template>

Given these utility templates, the example <util:time-of-day> template would look like:

<xsl:template match="sql:create-connection"> <xsl:variable name="name"> <xsl:call-template name="get-parameter"> <xsl:with-param name="name">name</xsl:with-param> <xsl:with-param name="required">true</xsl:with-param> </xsl:call-template> </xsl:variable> <xsl:variable name="jdbc-driver"> <xsl:call-template name="get-parameter"> <xsl:with-param name="name">jdbc-driver</xsl:with-param> <xsl:with-param name="required">true</xsl:with-param> </xsl:call-template> </xsl:variable> <xsl:variable name="connect-url"> <xsl:call-template name="get-parameter"> <xsl:with-param name="name">connect-url</xsl:with-param> <xsl:with-param name="required">true</xsl:with-param> </xsl:call-template> </xsl:variable> <xsl:variable name="user-name"> <xsl:call-template name="get-parameter"> <xsl:with-param name="name">user-name</xsl:with-param> <xsl:with-param name="required">true</xsl:with-param> </xsl:call-template> </xsl:variable> <xsl:variable name="password"> <xsl:call-template name="get-parameter"> <xsl:with-param name="name">password</xsl:with-param> <xsl:with-param name="required">true</xsl:with-param> </xsl:call-template> </xsl:variable> <xsp:logic> SQLHelper.createConnection(<xsl:copy-of select="$name"/>, <xsl:copy-of select="$connect-url"/>, <xsl:copy-of select="$user-name"/>, <xsl:copy-of select="$password"/>, request); </xsp:logic> </xsl:template>

This example shows clearly why we need a SiLLy language!

6. XSLT Logicsheets and XSP for Java

The Java programming language defines a source program structure that must be take into account for

properly generating code:

- Package declaration
- Imports
- Class declaration
- Class-level declarations (methods and variables)
- Application-specific method body

The <xsp:page> tag must contain one (and only one) "user" root element.

All markup enclosed within the user root element will be placed inside method generate() of the generated XSPGenerator subclass.

The <xsp:structure> and <xsp:include> tags are used to import "external" classes and **must** be top-level elements (i.e., they must placed directly under the <xsp:page> root element):

<xsp:structure> <xsp:include>java.sql.*</xsp:include> <xsp:include>java.text.SimpleDateFormat</xsp:include> </xsp:structure>

The <xsp:logic> tag can be used to generate *class-level* variable and method declarations when used as a top-level element:

<xsp:page> <xsp:structure> <xsp:include>java.text.SimpleDateFormat</xsp:include> </xsp:structure> <!-- Class-level declarations --> <xsp:logic> private static String timeOfDay(String format) { if (format == null || format.length() == 0) { format = "hh:mm:ss"; } return SimpleDateFormat.getInstance().format(new Date(), format); } </xsp:logic> . . . <user-root> . . . It's now <xsp:expr> timeOfDay(request.getParameter("timeFormat")); </xsp:expr> . . . </user-root> <xsp:page>

Thus, when a logicsheet adds to the import or class-level declaration "sections", it **must** preserve all the declarations possibly generated by previous logicsheets:

<xsl:template match="xsp:page"> <xsp:page> <xsl:apply-templates select="@*"/> <xsp:structure> <xsp:include>java.text.*</xsp:include> </xsp:structure> <xsp:logic> private static int count = 0; private static synchronized int getCounter() { return ++count; } . . . </xsp:logic> <xsl:apply-templates/> </xsp:page> </xsl:template>

7. The SiLLy Logicsheet Language

As of today, the SiLLy Logicsheet Language does not exist. What is described below are thoughts on possible implementation of the language.

In order to overcome the extreme complexity of logicsheet transformations expressed in XSLT, a simpler, higher-level XML transformation language is being defined: *Simple Logicsheet Language* (or *SiLLy*, so baptized by Stefano Mazzocchi in a humorous rejection of its first proposed name).

SiLLy templates are much terser and easier to read and write than the XSLT-based examples presented above:

<sll:logicsheet xmlns:sll="http://xml.apache.org/sll" xmlns:xsl="http://www.w3c.org/1999/XSL/Transform" > <sll:namespace prefix="sql" uri="http://xml.apache.org/sql"/> <sll:prolog> <xsp:structure> <xsp:include>import SQLHelper;</xsp:include> </xsp:structure> <sll:prolog> <sll:element name="create-connection"> <sll:parameter name="name" required="true"/> <sll:parameter name="jdbc-driver" default="oracle.jdbc.driver.OracleDriver"/> <sll:parameter name="connect-url" default="jdbc:oracle:thin:@localhost:1521:ORCL"/> <sll:parameter name="user-name" required="true"/> <sll:parameter name="password" required="true"/> <sll:body> <xsp:logic> SQLHelper.createConnection(<sll:parameter-value select="name"/>,

<sll:parameter-value select="jdbc-driver"/>, <sll:parameter-value select="connect-url"/>, <sll:parameter-value select="user-name"/>, <sll:parameter-value select="password"/>, request); </xsp:logic> </sll:body> </sll:element> </sll:logicsheet>

SiLLy logicsheets are translated into an equivalent XSLT stylesheet using XSLT itself.

It is possible (and, indeed, simple) to generate a stylesheet that uses the xsl namespace without ambiguity: XSLT processors are bound to the XSL namespace *URI*, rather than to its prefix. In addition to this, XSLT defines a <xsl:namespace-alias> directive that can be used to map one namespace's URI to another.

SiLLy provides a limited form of parameter validation: when a dynamic tag parameter is defined as *required* its absence in the source XML document will trigger the abnormal termination of the code generation process producing a (more or less) meaningful message by means of <xsl:message>

It's also possible to specify a list valid values for a parameter. Such list will be used for parameter validation when values passed are constants known at transformation time.

In addition to dynamic *tags*, SiLLy can also match attributes, and processing instructions.

In absence of a schema or DTD, the following examples illustrate the basic SiLLy matching and transformation capabilities (use-cases are shown as XML comments):

<!-- <util:time-of-day format="hh:mm aa"/> --> <sll:element name="time-of-day" uri="http://www.plenix.org/util" prefix="util"> <sll:parameter name="format" default="hh:mm:ss"/> <sll:body> <xsp:expr> SimpleDateFormat.getInstance().format(new Date(), <sll:parameter-value name="format"/>) </xsp:expr> </sll:body> </sll:element> <!-- --> <sll:attribute name="src" uri="http://www.plenix.org/translator" prefix="flag"> <sll:body> <xsp:attribute name="src"><xsp:expr> request.getParameter("<sll:attribute-value/>"); </xsp:expr>.gif</xsp:attribute> </sll:body> </sll:attribute> <!-- <?log Commit point reached?> --> <sll:processing-instruction target="log"> <sll:body> <xsp:logic> logger.log("<sll:pi-data/>"); </xsp:logic> </sll:body> </sll:attribute> <sll:processing-instruction> is probably overkill

1. Comments

add your comments