Roxen CMS 5.4

XSLT Tutorial

2014-03-11
www.roxen.com

Contents

1 Introduction

Welcome to the Roxen CMS Tutorials. This section is dedicated to all users of Roxen CMS. The tutorials are intended for both beginners and experienced users, and we hope that all find some interesting reading and get creative ideas.

It is assumed that the reader is familiar with HTML and have some knowledge of XML.

As always, if you have any suggestions, comments or complaints regarding these tutorials do not hesitate to send an email to manuals@roxen.com and if the issue is an obvious bug do not hesitate to report it to Bug Crunch, our bug tracking system.

1.1 XSLT Tutorial

Roxen CMS includes a template system based on the Extensible Stylesheet Language Transformations (XSLT) standard version 1.0. It's beyond the scope of this manual to document XSLT in its entirety but we will present a tutorial on how to start using it. We will also describe some important Roxen CMS-specific issues on how XSLT is implemented.

Related to XSLT is another standard named XPath. It defines the syntax for expressions in XSLT stylesheets and is also used by Roxen CMS. The XSLT and XPath standards can be found on the W3C web site at www.w3.org/TR/xslt and www.w3.org/TR/xpath, respectively. It should be noted that while these documents are the main references for technical details they are not particularly reader-friendly; instead we recommend www.xslt.com to find news, tutorials, FAQs, book references, mailing lists and other valuable resources.

2 Using HTML Contents

XML and HTML may look similar at first but there are syntactic differences that make them incompatible. In normal use XSLT templates are restricted to XML input only, meaning that HTML content isn't suitable until it's rewritten to adhere to XML standards. However, in Roxen CMS this will happen automatically whenever a HTML document is requested by a client browser. The conversion will perform these changes:

- Convert all empty tags to properly terminated XML tags
- Terminate all containers which are optional in HTML
- Add surrounding quotes to all tag attribute values
- Convert tag names and attributes to lower-case
- Rewrite empty attributes
- Rewrite `&` as `&amp;`
- Add a top-level container element if one is missing

2.1 HTML to XML conversion example

Take the following HTML page as an example. This is what the original code might look like:

```
<HTML><BODY>
  <H1>Sales</H1>
  This month's sales data:<br>
  <table>
  <tr>
    <td bgcolor=#c0c0c0>Product<td>Quantity
  <tr>
    <td nowrap>Nuts & bolts<td>2,500 pcs
  <tr>
    <td>Drills<td>185 pcs
  </table>
</BODY></HTML>
```

After automatic XML conversion Roxen CMS will get this page:

```
<html><body>
  <h1>Sales</h1>
  This month's sales data:<br/>
  <table>
  <tr>
    <td bgcolor="#c0c0c0">Product</td><td>Quantity</td>
  </tr><tr>
    <td nowrap="nowrap">Nuts &amp; bolts</td><td>2,500 pcs</td>
  </tr><tr>
    <td>Drills</td><td>185 pcs</td>
  </tr>
  </table>
</body></html>
```

This page is now sufficiently XML compliant to be accepted by the XML parser and can then be formatted using an XSLT stylesheet.

The converter will also try to repair some simple syntax errors such as erroneous tag nesting. One common construct is:

```
<b><i>Hi there!</b></i>
```

which is converted into:

```
<b><i>Hi there!<!--</b>--></i></b>
```

thus making the containers properly nested.

3 An XML Document is a Tree

Throughout this introduction we will use a tree analogy for XML documents. This is convenient since it lets us talk about parents, siblings, children and other relations.

3.1 A sample document tree

To exemplify this we can start with an XML file:

```
<?xml version="1.0"?>
<ancestor>
  <parent>
    <sibling-1/>
    <sibling-2/>
    <sibling-3>
      <!-- a comment -->
      <child/>
      Some text!
    </sibling-3>
  </parent>
</ancestor>
```

From this file we can draw a tree:

```
                    /
                    |
                <ancestor>
                    |
                 <parent>
                    |
     +--------------+--------------+
     |              |              |
 <sibling-1>    <sibling-2>    <sibling-3>
                                   |
                    +--------------+------------+
                    |              |            |
           <!-- a comment -->   <child>    Some text!
```

Each item pictured in this tree is called a *node*. There are element nodes, comment nodes, text nodes and so on. At the very top is a *root node* called `/`. In addition to the input data the output is also represented as a tree during the XSLT transformation. This model is very helpful when trying to understand what goes on.

One node in particular, the *current node*, is of great importance. This is the data item in the input file where the XSLT engine is currently looking for input. At the beginning the current node is set to the document root but it will change over time as processing takes place.

By starting at the root and tracing the lines downward we can refer to any of the elements in the tree. E.g. `/ancestor/parent/sibling-3/child` is a path from the root to the `<child>` element. We will later on see how we can find nodes by starting at any node, tracing the lines e.g. left or right and counting the number of nodes we pass by.

4 Where to Place Stylesheets

Roxen CMS gives you a lot of power in the way it locates XSLT stylesheets in your site hierarchy. It will search for stylesheets in the directory where the requested content file is stored and, if not found, it will move on to the parent directory, the parent's parent directory and so on. This search strategy has a major benefit in that site-wide stylesheets can be placed at the root of the site and then be overridden, either in part or completely, as needed in selected sub-trees.

4.1 Roxen CMS extensions for locating stylesheets

This implementation supports absolute or relative import paths in the `<xsl:import>` statement. The latter type can be prefixed with `ancestor::` or `ancestor-or-self::` (concepts borrowed from XPath) to search relative to the stylesheet directory. By using `ancestor::` one can have a stylesheet named, for instance, `footer.xsl` which imports another stylesheet using the same name located higher up in the site hierarchy.

The import statement has been extended to handle parametric import, in other words expressions which let the import file be determined at run-time. See the section Template Parameterization for more examples on this.

```
<!-- absolute path -->
<xsl:import href="/articles/footer.xsl"/>

<!-- use smart search starting in content file directory -->
<xsl:import href="footer.xsl"/>

<!-- use smart search starting in same directory as this stylesheet -->
<xsl:import href="ancestor-or-self::footer.xsl"/>

<!-- use smart search starting in parent directory of this stylesheet -->
<xsl:import href="ancestor::footer.xsl"/>

<!-- use parametric import -->
<xsl:import href="{concat($footer, '.xsl')}"/>
<xsl:param name="footer" rxml:type="select:footer1,footer2"
select="'footer1'"/>
```

Since stylesheets may contain valuable information that should not be visible in plain text to visitors of the site it can be useful to prevent direct downloading of an XSLT file. This is easily achieved by setting the Externally Visible metadata for the stylesheet file to Never. The stylesheet will still work as expected but can no longer be accessed from a browser.

5 Setting up the DemoLabs Site

Creating a new Platform virtual server in the Roxen CMS configuration interface will let you initialize your site repository with a copy of a demo site for a fictitious company called DemoLabs. This site will serve as an example for this introduction on XSLT so we recommend that you set up a DemoLabs site before continuing.

To accomplish this, go to the Roxen WebServer administration interface, select the Sites tab and click *Create new site*; a wizard will guide you through this process step by step. One of the wizard pages asks you to select the preinstalled content for your site, and this is where you should select DemoLabs. Once you answer the remaining questions your site will be created and ready for you to use.

Note that the code snippets included in this tutorial are not always complete files. Many times only a part of the file is shown, and there may be sections marked ... where text not relevant to the discussion at hand has been removed for brevity.

6 A First Look at a Stylesheet File

Let's take a look at the DemoLabs stylesheet called common.xsl (stored in the root of the DemoLabs site):

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0" rxml:copy-unknown-elements="yes">

<xsl:import href="xhtmllayout.xsl"/>
<xsl:import href="buttons.xsl" rxml:customize-params="no"/>
<xsl:import href="containers.xsl" rxml:customize-params="no"/>
<xsl:import href="{$palette-file}" rxml:customize-params="no"/>

<xsl:output method="html" />

<xsl:param name="palette-file"
           rxml:type="file:/color-palette*.xsl"
           select="'color-palette1.xsl'"
           rxml:group="Colors"/>
<xsl:param name="company-logo"
           rxml:type="string"
           select="'DemoLabs'"
           rxml:group="Strings"/>
<xsl:param name="company-full"
           rxml:type="string"
           select="'DemoLabs, Inc.'"
           rxml:group="Strings"/>
  ...
</xsl:stylesheet>
```

The first line declares that this is an XML file. Although not required it is a good idea to indicate which XML version one is using. Following that is the `<xsl:stylesheet>` tag which encapsulates the entire stylesheet. Again a version attribute which in fact is required by the XSLT standard (but not enforced by Roxen CMS), and thereafter a Roxen CMS attribute called `rxml:copy-unknown-elements` which we will explain later on.

6.1 Importing code from separate files

All stylesheet data need not be contained in one single file since the XSLT standard supports importing of one or more stylesheets into another. The `<xsl:import>` tags in this file imports other XSL stylesheets, thereby making all of the rules and variables declared in those stylesheets available as well. The given names may be absolute pathnames when a specific file is wanted, but here we see simple filenames which enable the smart search strategy described recently. The exception is the last import directive which takes advantage of a parametric import.

6.2 Controlling output format

The `<xsl:output>` tag controls various aspects of the output generation. The XSLT engine can operate in one of three modes; XML, HTML or text, and although the default mode is derived from a set of heuristics we here tell the engine that we always want it to return HTML data.

6.3 Global parameters

Finally there are a number of `<xsl:param>` tags declaring global parameters. Parameters give the stylesheet author an easy way to control various aspects of how stylesheets work. In this case the `<xsl:param>` tags use Roxen CMS-specific attributes called `rxml:type` and `rxml:group` to include type information and parameter grouping titles, respectively, both of which are used by the Customize Template feature in the Content Editor. More on that later in this introduction.

7 Applying Template Rules

A stylesheet file normally contains a number of *template rules* where each rule describes how a particular tag in the content file should be processed. The code inside a template rule is executed like a computer program and the resulting data is sent to the web page.

7.1 Basic match pattern

So far we have not seen any DemoLabs template rules. The inner workings of the stylesheet is stored in, for instance, `xhtmllayout.xsl` and its import files, and here's one of the least complex template rules from `containers.xsl`:

```
<xsl:template match="h2">
  <p/>
  <font size="+2"><b><i><xsl:apply-templates/></i></b></font>
  <br/>
</xsl:template>
```

The *match* attribute tells the XSLT engine which input tags this template rule should be applied to. For this rule any `<h2>` tag is acceptable.

The rule's body will be parsed and tags not belonging to the XSLT namespace (i.e. prefixed by `xsl:`) will be copied to the output. In this case all but one tag is copied; meanwhile, the `<xsl:apply-templates/>` tag tells the engine to process the contents of the `<h2>` input element recursively.

7.2 Matching with predicates

Another rule from the same stylesheet file:

```
<xsl:template match="p[@initial]">
  <xsl:apply-templates/>
  <br clear="all"/>
</xsl:template>
```

The major change here is the more advanced match attribute. The `p[@initial]` string is a short-hand way of writing `p[attribute::initial]`. Square brackets surround a predicate which places additional restrictions on when the template rule is invoked. This particular rule only applies to `<p initial="initial">` tags, leaving regular `<p>` tags unprocessed.

As illustrated by these examples `<xsl:apply-templates/>` is the magic tag which drives the template application. In its simplest form it will process all children of the current input element but by supplying a *select* attribute it can be directed to read any other input data.

If the XSLT engine finds input data for which no template rule exists one of two things will happen:

- All traditional XSLT stylesheets contain a built-in rule which is applied to unknown tags. This rule consists of a call to `<xsl:apply-templates/>`. Another built-in rule is used for text elements which are copied to the output.

- Stylesheets declared with `rxml:copy-unknown-elements="yes"` (as seen earlier in one of the examples) will behave slightly differently. The built-in rule has been modified to not only call `<xsl:apply-templates/>` recursively but also copy the name of the current tag and its attributes to the output. It will also copy comments and processing instructions in both the content file and the stylesheet file.

  This special mode is not compliant with the XSLT standard but is useful when lots of HTML code is included in the input files and you want it preserved in the output without adding template rules for each and every HTML tag.

8 Reusing Template Code

All non-XSLT tags which are placed in the stylesheet file are treated as *literal result elements*. As already mentioned they will be copied to the output whenever the rule they are located in is processed. One might think that this can be used to call one of the template rules directly, which unfortunately is wrong. In other words this code fragment will not work:

```
<xsl:template match="greeting">
  <p>Hi there!</p>
</xsl:template>

<xsl:template match="message">
  <!-- this doesn't work -->
  <greeting/>
</xsl:template>
```

8.1 Calling named templates

How can this be accomplished then? `<xsl:call-template/>` is the answer, but before we can use it we must assign unique names to those template rules we wish to call:

```
<xsl:template match="greeting" name="show-greeting">
  <p>Hi there!</p>
</xsl:template>

<xsl:template match="message">
  <!-- this works -->
  <xsl:call-template name="show-greeting"/>
</xsl:template>
```

You can find more examples on this in `xhtmllayout.xsl`.

8.2 Handling unbalanced output

At times it may be necessary to split code into separate template rules which can be called individually. Often this leads to an XML syntax problem since all tags must be well-balanced. Once more we'll start by looking at a non-functional code fragment:

```
<xsl:template name="start-table">
  <!-- won't work -->
  <table>
</xsl:template>

<xsl:template name="end-table">
  <!-- neither will this -->
  </table>
</xsl:template>

<xsl:template match="my-table">
  <xsl:call-template name="start-table"/>
  <tr><td>...</td></tr>
  <xsl:call-template name="end-table"/>
</xsl:template>
```

Understandably this will never be accepted by the XML parser. What must be done in such cases is to use a special XML wrapper called `<![CDATA[...]]>` which makes the XML parser accept the input as pure text instead of markup, and finally add `<xsl:text>` to tell the XSLT engine to output this text unmodified:

```
<xsl:template name="start-table">
  <xsl:text disable-output-escaping="yes"><![CDATA[<table>]]></xsl:text>
</xsl:template>

<xsl:template name="end-table">
  <xsl:text disable-output-escaping="yes"><![CDATA[</table>]]></xsl:text>
</xsl:template>
```

9 Template Modes and Expressions

XSLT defines a concept called *mode*. This is a way to control which template rules that gets applied when `<xsl:apply-templates/>` is executed.

For instance, consider these template rules for `<h1>`:

```
<xsl:template match="h1" mode="toc">
  ...
</xsl:template>

<xsl:template match="h1">
  ...
</xsl:template>
```

Here `toc` is an abbreviation for *table of contents*. By defining two rules for the same tag and separating them by a *mode* attribute we can decide that `<h1>` tags should be processed in a different manner when we are builting the table of contents:

```
<xsl:template match="document">
  <h1>Table of contents</h1>
  <xsl:apply-templates select="h1" mode="toc"/>

  <hr/>
  <xsl:apply-templates/>
</xsl:template>
```

The first `<xsl:apply-templates>` will select all `<h1>` elements and process them using the rule designated for table of contents. In contrast, the second `<xsl:apply-templates/>` will select any child element of the current node (i.e. all elements inside the `<document>` container) and process them recursively in normal mode.

One important detail here is that we are reusing the same `<h1>` tags in the input file for different purposes. No matter which one of the rules we invoke they will get the same `<h1>` tags in sequence as their input data. The `containers.xsl` and `page-components.xsl` files relies on this fact to pull off another trick, namely creating relative references from the table of contents to the headings in the document. Let's return to the `<h1>` example again, this time expanding it to introduce cross-reference links:

```
<xsl:template match="h1" mode="toc">
  ...
  <a href="#{generate-id(.)}">
    <xsl:value-of select="."/>
  </a>
  ...
</xsl:template>

<xsl:template match="h1">
  <a name="{generate-id(.)}"/>
  ...
</xsl:template>
```

The expression `generate-id(.)` is an XPath expression which generates a unique identifier string for the node ., i.e. the current node. It is guaranteed to generate the same identifier for the same node within one activation of the XSLT engine. Knowing this we can build a relative URL in the table of contents by inserting `<a`

`href="#id_4711">...</a>` and have the section we link to start with a corresponding `<a name="id_4711"/>` anchor.

9.1 Attribute value templates

The curly brackets are needed to tell the XSLT engine that parts of the literal result element attributes should be treated as an expression. They are not needed when passing expressions in normal XSLT tags.

This example also introduces the tag `<xsl:value-of select="."/>`. Its purpose is to extract the text value of the current node. The reason why `<xsl:apply-templates/>` is not used is that we are not interested in any nested tags in the `<h1>` container; we want `<h1>Foo <i>Bar</i></h1>` to return `Foo Bar` excluding any formatting details.

10 XPath Axes and Predicates

We have touched on the subject of expressions earlier but we will now take a closer look. One of the previous examples used the syntax `p[attribute::initial]`; as mentioned then square brackets denote a predicate which in this case is an existence test for an attribute named *initial*. This predicate is tested against the candidate node in the input file, and if the result is the boolean value true the rule is processed.

Of special interest here is the `attribute::` axis. An axis defines a direction in which we can look for data related to the current node. These are the axes available in XPath:

- `parent::`
- `ancestor::`
- `ancestor-or-self::`
- `child::`
- `descendant::`
- `descendant-or-self::`
- `preceding::`
- `preceding-sibling::`
- `following::`
- `following-sibling::`
- `attribute::`
- `namespace::`
- `self::`

Recall for a minute the tree picture presented in the section An XML Document is a Tree. With the current node (which can be any node) as a starting point the axes provide a means of describing how to reach any other node in the tree.

10.1 Abbreviated form

Some axes are frequently used and may be abbreviated into shorter forms. The following table lists a number of popular constructs, and when the expanded form refers to `node()` it means any tag, text, comment etc:

Abbreviated form	Expanded form
`@foo`	`attribute::foo`
`foo/bar`	`child::foo/child::bar`
`foo//bar`	`child::foo/descendant-or-self::node()/child::bar`
`..`	`parent::node()`
`.`	`self::node()`

10.2 Predicates

Nodes on an axis can be located in different ways. Usually one uses a name or other type of node identifier, possibly combining this with a relative position counter. E.g., `preceding-sibling::tr[1]` retrieves the closest preceding `<tr>` element at the same nesting level as the current node. This works due to the fact that `[1]` is treated as the expression `[position() = 1]`, i.e. a position predicate.

Predicates can be nested and/or sequenced like `tr[td[1][@align='center']]` which finds a `<tr>` element whose first `<td>` child element has an `align='center'` attribute.

11 Iteration and Conditional Processing

In earlier examples we have seen the use of `<xsl:apply-templates>` to process data recursively. Instead of writing the template code as a stand-alone rule you can process data directly using the loop construct. In this example we will use both `<xsl:for-each>` and `<xsl:if>` to present a comma-separated list of name and email addresses:

```
<!-- data file -->
<?xml version="1.0"?>
<customers>
  <customer>
    <name>Alice</name>
    <email>alice@somewhere.com</email>
  </customer>
  <customer>
    <name>Bob</name>
    <email>bob@elsewhere.com</email>
  </customer>
</customers>
```

```
<!-- template file -->
<?xml version="1.0"?>
<xsl:stylesheet>
  <xsl:template match="customers">
    <h3>Contact List</h3>
    <xsl:for-each select='customer'>
      <xsl:value-of select='name'/>
      <a href="mailto:{email}"><xsl:value-of select="email"/></a>
      <xsl:if test="position() != last()">, </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Sharp-eyed observers will again recognize the use of curly brackets in the `href` attribute of the `<a>` element. In XSLT an attribute where expressions can be embedded is called attribute value template. In this case the expression is a selection statement which inserts the `<email>` data into the link.

The `<xsl:if>` statement ensures that the list item separator is not inserted after the final item. To accomplish this we use a boolean expression which compares the position of the current `<customer>` element to the position of the last `<customer>` element. Both of these positions are computed with respect to the the set of nodes selected in the `<xsl:for-each>` statement. Also noteworthy is the `!=` operator which stands for *not equal* and is borrowed from programming languages such as C and C++.

For situations where a single `<xsl:if>` statement isn't powerful enough XSLT also offers `<xsl:choose>` where any number of `<xsl:when>` test cases as well as an `<xsl:otherwise>` fallback case can be handled. Interested readers can look in e.g. `navigation-components.xsl` which contains numerous `<xsl:choose>` statements.

The Roxen CMS implementation has also been extended to allow attribute value templates in the `<xsl:output>` attributes. This enables, among other things, browser-specific output modes:

```
<?xml version="1.0"?>
<xsl:stylesheet>
  <!-- check for WAP 1.1 browser -->
  <xsl:variable name="output-method">
    <xsl:choose>
      <xsl:when test="contains($rxml:supports, ' wml1.1 ')">xml</xsl:when>
      <xsl:otherwise>html</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <!-- set output mode dynamically -->
  <xsl:output method="{$output-method}"/>
</xsl:stylesheet>
```

The code examines the `rxml:supports` variable which is explained in the section Roxen Extensions.

12 Variables and Parameters

By creating a *variable* you can store strings, numbers, booleans, node-sets and even *result tree fragments* and recall them later on. A result tree fragment represents nodes in a tree structure which later on will be used to generate the output.

Variables are only accessible from the point where they are declared and onward within the parent element. The exception is variables declared at the top-most level, i.e. outside template rules, in which case they become global. Another concept related to variables are parameters which are needed to send values to template rules.

12.1 Variable example

The file `print.xsl` in the DemoLabs site includes a template rule which we will use to demonstrate variables:

```
<xsl:template match="p[@initial]/text()[1]">
  <xsl:variable name="char" select="substring(., 1, 1)"/>
  <xsl:variable name="rest" select="substring(., 2)"/>
  <font size="+1">
    <b><xsl:value-of select="$char"/></b>
  </font>
  <xsl:value-of select="$rest"/>
</xsl:template>
```

This rule has a rather advanced match pattern whose purpose is to find the first text child of a `<p initial="initial"/>` element. The template rule declares two local variables, char and rest, where we place non-overlapping substrings of the current text node. The first substring holds the first character and the second holds the remaining text. To access the values of char and rest we use the expressions `$char` and `$rest`, respectively, which is the XPath syntax for variable references. The end result is that the initial character in the `<p/>` container will be displayed boldfaced and in larger size and the remaining string is displayed normally. A similar rule is found in `containers.xsl` but that one takes advantage of `<gtext/>` to render a drop caps graphically.

12.2 Passing parameters in template calls

Sending parameters requires a bit more attention. Both the caller and the template rule which expects parameters as input must agree on parameter names:

```
<!-- from buttons.xsl -->
<xsl:template name="gnutton" match="gnutton">
  <xsl:param name="align" select="@align"/>
  <xsl:param name="valign" select="@valign"/>
  <xsl:param name="text" select="text()"/>
  ...
</xsl:template>

<!-- from page-components.xsl -->
<xsl:call-template name="gnutton">
  <xsl:with-param name="text" select="concat($string-print, '!')"/>
  <xsl:with-param name="align">left</xsl:with-param>
  <xsl:with-param name="valign">bottom</xsl:with-param>
</xsl:call-template>
```

As indicated from this example the order of parameters is not important but the names are. The select attributes of the `<xsl:param>` tags for the parameters `align`, `valign`, and `text` set default values in case the caller doesn't specify an actual value.

To pass values `<xsl:with-param>` elements are inserted into the `<xsl:call-template>` container. They may also be used in companion with `<xsl:apply-templates>` elements. Any parameter for which the invoked template rule fails to declare a matching `<xsl:param>` are ignored.

13 Template Parameterization

In the previous section we mentioned local and global variables. In fact there can be global parameters as well. The XSLT specification lets the software implementation decide on how this is realized; in Roxen CMS we use special markup syntax to make this possible. We will now return to the code displayed in the section A First Look at a Stylesheet File, this time highlighting the use of `<xsl:param>` at a global level:

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0" rxml:copy-unknown-elements="yes">
  ...
  <xsl:import href="{$palette-file}" rxml:customize-params="no"/>
  ...

  <xsl:param name="palette-file" select="'/color-palette1.xsl'"
             rxml:type="file:/color-palette*.xsl" rxml:group="Colors"/>

  <xsl:param name="company-logo" select="'DemoLabs'"
             rxml:type="string" rxml:group="Strings"/>

  <xsl:param name="company-full" select="'DemoLabs, Inc.'"
             rxml:type="string" rxml:group="Strings"/>

  <xsl:param name="domain-name" select="'demolabs.com'"
             rxml:type="string" rxml:group="Strings"/>

  <xsl:param name="logo-font" select="'quadrangle'"
             rxml:type="font" rxml:group="Font settings"/>

  <xsl:param name="logo-font-size" select="36"
             rxml:type="int" rxml:group="Font settings"/>

  <xsl:param name="h1-font" select="'yikes!'"
             rxml:type="font" rxml:group="Font settings"/>

  <xsl:param name="h1-font-size" select="36"
             rxml:type="int" rxml:group="Font settings" />

  <xsl:param name="initial-font" select="'quadrangle'"
             rxml:type="font" rxml:group="Font settings"/>

  <xsl:param name="initial-font-size" select="32"
             rxml:type="int" rxml:group="Font settings" />

  <xsl:param name="h2-fonts" select="'arial, helvetica, sans-serif'"
             rxml:type="string" rxml:group="Font settings"/>

</xsl:stylesheet>
```

13.1 Assigning type information

Any global parameters declared with an Roxen CMS-specific `rxml:type` attribute will be noticed by the Content Editor and included in the user-friendly *Customize Template* wizard. The `select` attributes are once again setting default values but values entered in the wizard will override the defaults. The possible types for `rxml:type` are:

- `string, text`: Two variants of textual parameters which will be instantiated as values of the XPath *string* type.
- `int, float`: Numeric parameter types which will be instantiated as values of the XPath *number* type.

- `Checkbox`: Boolean toggle. Set the initial value to 1, `on`, `true` or `enable` to give the XPath *boolean* variable a value of *true*. Any other value will yield *false*.
- `color`: Color selection parameter which will be instantiated as a value of the XPath *string* type.
- `font`: Font selection parameter which will be instantiated as a value of the XPath *string* type. The list of available fonts depends on what is installed in the Roxen server.
- `select:item,item,...`: String selection popup parameter which will be instantiated as a value of the XPath *string* type. Choices should be listed in a comma-separated string (literal comma can be included if escaped as `\,`).
- `file:path,path,...`: Similar to `select` but will index files at the given locations and present them in a popup menu. The absolute path of the chosen file will be instantiated as a value of the XPath *string* type. A path may be relative to the current stylesheet or absolute. The last segment of an absolute path, i.e. after the last `/` character, may contain wildcards such as `*`and `?`. Note that this is not supported for relative paths.

13.2 Parameter example

Here are some parameter type examples:

```
<!-- string -->
<xsl:param name="company-logo" rxml:type="string" select="'DemoLabs'"/>

<!-- int -->
<xsl:param name="logo-font-size" rxml:type="int" select="36"/>

<!-- checkbox -->
<xsl:param name="news-headlines" rxml:type="checkbox" select="1"/>

<!-- color -->
<xsl:param name="bgcolor-primary" rxml:type="color" select="'#ffffc9'"/>

<!-- font -->
<xsl:param name="h1-font" rxml:type="font" select="'yikes!'"/>

<!-- select -->
<xsl:param name="product" rxml:type="select:Roxen WebServer,Roxen CMS"
           select="'Roxen CMS'"/>

<!-- file -->
<xsl:param name="palette-file" rxml:type="file:/color-palette*.xsl"
           select="'/color-palette1.xsl'"/>
```

The file type and the Roxen-enhanced `<xsl:import>` is a powerful combination. As seen in this example we use it to determine which color palette we should inherit at run-time. Since all import directives must come before anything else inside a `<xsl:stylesheet>` element the XSLT engine has been relaxed to allow forward-referencing of a parameter name.

The double quoting seen in string values in XSLT expressions is important. If not present the string is treated as an expression resulting in a syntax error or unexpected behavior. Example:

```
<!-- incorrect use of select -->
<xsl:param name="company-logo" rxml:type="string" select="DemoLabs"/>
```

The incorrect code in this example tells the XSLT engine to initialize the `company-logo` parameter to the value found in the `<DemoLabs>` element, which is not at all what we wanted.

13.3 Other documentation options

Two other attributes provided by Roxen CMS are `rxml:group` and `rxml:doc`. As seen above the `rxml:group` is used to place a number of related parameters into the same logical group; this grouping is visible in the Customize Templates wizard in the Content Editor. Likewise, `rxml:doc` documents the parameter and displays its value in the customization wizard.

14 Multiple Source Documents

One of the less obvious but extremely powerful features of XSLT is the ability to read input from several documents at once. This is done through the `document()` function which returns a node-set of all nodes in the given file.

The document contents can be used directly in expressions, or it may be stored in a variable for later processing. Consider an input file called `my-authors.xml` which contains the following data:

```
<?xml version="1.0"?>
<authors>
  <author name="Astrid Lindgren" country="Sweden"/>
  <author name="H.C. Andersen" country="Denmark"/>
  <author name="Selma Lagerlöf" country="Sweden"/>
</authors>
```

Next, this stylesheet wants to display all authors in two groups, Swedish and international:

```
<?xml version="1.0"?>
<xsl:stylesheet>

  <xsl:template match="/">
    <!-- read author data -->
    <xsl:variable name="data" select="document('my-authors.xml')"/>

    <h3>Swedish authors</h3>
    <xsl:for-each select="$data/authors/author[@country = 'Sweden']">
      <xsl:value-of select="@name"/><br/>
    </xsl:for-each>

    <h3>International authors</h3>
    <xsl:for-each select="$data/authors/author[@country != 'Sweden']">
      <xsl:value-of select="@name"/><br/>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

The `document()` function would not be needed if the file `my-authors.xml` was the one being accessed by the client browser and if that content file was set to use the stylesheet above as its template. However, for the purpose of this example we can imagine this stylesheet processed on behalf of another content file and as part of this task loads the author file.

In Roxen CMS all relative file paths will be interpreted as starting at the current content file. It does not currently support a second argument to `document()` specifying another base for relative paths as described in the XSLT specification.

15 Roxen Extensions

To conclude this XSLT introduction we will look at the Roxen Platform-specific extensions to XSLT and XPath variables, functions and elements.

15.1 Variables

When the XSLT engine processes a stylesheet in Roxen CMS you will always have access to a set of predefined global variables, all of string type:

`$rxml:supports`

A string containing a space-separated string (including leading/trailing spaces) of all support flags relevant to the current client browser. A hypothetical example would be " java javascript tables images stylesheets ". This can be used to optimize the stylesheet for a particular device or browser:

```
<xsl:template match="h1">
  <xsl:choose>

    <xsl:when test="contains($rxml:supports, ' images ')">
      <!-- render <h1> using Gtext graphics -->
      <gh1><xsl:apply-templates/></gh1>
    </xsl:when>

    <xsl:otherwise>
      <!-- output <h1> unmodified -->
      <h1><xsl:apply-templates/></h1>
    </xsl:otherwise>

  </xsl:choose>
</xsl:template>
```

Note the spaces which are necessary to differentiate e.g. "` java `" and "`javascript `" since "`java`" alone is a proper substring of "`javascript`".

A list of supports flags in Roxen CMS can be found in the section <if supports>.

`$rxml:host`
`$rxml:domain`
`$rxml:user`
`$rxml:name`

String values taken from entites in the Client Scope.

`$rxml:is-editarea`

Boolean flag that is true when the page is rendered in the edit area for a user, i.e. when a user is viewing his/hers uncommitted changes. This is the same as the `&user.is-editarea;` variable in the User Scope.

15.2 Functions

Here are the Roxen CMS extension functions:

`rxml:cookie(name)`

Returns the value of the given cookie.

`rxml:variable(name)`

Returns the value of the given variable in the form scope.

`rxml:pike-expression(program)`

The argument should be a Pike program encoded as string. To overcome string quoting problems it's usually a wise choice to put the code into a variable first:

```
<xsl:template match="/">
  <!-- define Pike program -->
  <xsl:variable name="my-program">
    mapping m = ([ "vendor"  : "Roxen",
                   "product" : "CMS" ]);
    return m->vendor + " " + m->product + " on host " + id->misc->host;
  </xsl:variable>

  <!-- execute it -->
  <p>You are running
    <xsl:value-of select="rxml:pike-expression($my-program)"/>.
  </p>
</xsl:template>
```

When adding `rxml:pike-expression` to a stylesheet it will mark all transformed pages as non-cacheable since the Pike code may have side-effects when run.

By default the `rxml:pike-expression()` function is disabled since it can be used to access privileged information about the server (specifically, the `RequestID` object named `id`). The server administrator can activate it by changing a configuration setting for the XSLTransform module.

`rxml:node-set(result-tree-fragment)`

Converts a result tree fragment into a node-set containing the same data. The operations allowed on a result tree fragment is very limited as described in the XSLT specification, but when converted into a node-set you can use e.g. `/`, `//`, `[]` and similar operators.

```
<xsl:template match="/">
  <!-- create a result tree fragment -->
  <xsl:variable name="fruit-fragment">
    <fruits>
      <fruit name="apple"/>
      <fruit name="banana"/>
    </fruits>
  </xsl:variable>

  <!-- convert fragment into node-set and find "banana" -->
  <xsl:value-of
       select="rxml:node-set($fruit-fragment)/fruits/fruit[2]/@name"/>
</xsl:template>
```

`rxml:metadata()`
`rxml:metadata(path)`

Retrieves metadata information about the current content file (when no argument is given) or a specific file/directory. The set of metadata as well as their names are taken from entities listed in <emit dir> section.

The node-set returned for the current file will be a number of elements grouped directly under a root node:

```
<!-- output from "rxml:metadata()" -->
<filename>index.xml</filename>
```

```
<type>text/xml</type>
<template>common.xsl</template>
<author-name>doris</author-name>
...
```

When a path is given the returned node-set consists of a number of `<file>` and `<dir>` elements grouped directly under a root node. The `<file>` elements are identical to what was described above, while the <dir> elements include only `<dirname>`, `<url>` and `<selected>` elements:

```
<!-- output from "rxml:metadata('/news/')" -->
<file>
  <filename>index.xml</filename>
  <type>text/xml</type>
  <template>common.xsl</template>
  <author-name>doris</author-name>
  ...
</file>
<file>
  <filename>common.xsl</filename>
  <type>sitebuilder/xsl-template</type>
  <author-name>doris</author-name>
  ...
</file>
<dir>
  <dirname>news</dirname>
  <url>/news/</url>
  <selected>no</selected>
</dir>
...
```

The `<selected>` element contains yes or no depending on whether the directory is part of the path to the current content page.The order of the file and directory items is undefined so the `<xsl:sort>` is recommended for sorting the data:

```
<!-- generate sorted directory listing -->
<ul>
  <xsl:for-each select="rxml:metadata('/')/*">
    <xsl:sort select="filename | dirname"/>
    <li><xsl:value-of select="filename | dirname"/></li>
  </xsl:for-each>
</ul>
```

15.3 Elements

There are three Roxen CMS-specific extension elements:

`<rxml:parse>`

Runs the RXML parser on the element and copies the resulting data to the output:

```
<xsl:variable name="crypted-password">
  <rxml:parse>
    <crypt>hello world</crypt>
  </rxml:parse>
</xsl:variable>
```

Note that the output data is not fed into the XSLT parser so you cannot create dynamic XSLT stylesheets this way. Since RXML tags may have side-effects all caching of transformed web pages is disabled.

<rxml:copy-attributes>
<rxml:copy-attributes ignore="attr1,attr2,...">

Copies all attributes of the context node to the output. Can only be used before other children are added to the the output element (just as `<xsl:attribute>`). The optional *ignore* attribute should be a comma-separated list of attributes to ignore.

```
<xsl:template name="img">
  <img>
    <!-- copy all <img> attributes except JavaScript event handlers -->
    <rxml:copy-attributes ignore="onclick,onmousedown,onmouseup,
                                  onmouseover,onmousemove,onmouseout,
                                  onkeypress,onkeydown,onkeyup"/>
  </img>
</xsl:template>
```

This can be rewritten in standard XSLT using `<xsl:copy-of>` which is the recommended solution for maximum compatibility:

```
<xsl:template name="img">
  <img>
    <!-- copy all <img> attributes except JavaScript event handlers -->
    <xsl:copy-of select="@*[name(.) != 'onclick' and
                            name(.) != 'onmousedown' and
                            name(.) != 'onmouseup' and
                            ...]"/>

    <!-- even better solutions may exist in specific cases -->
    <xsl:copy-of select="@*[not(starts-with(name(.), 'on'))]"/>
  </img>
</xsl:template>
```

<rxml:helptext>
<rxml:helptext match="element">,
<rxml:helptext match="element/@attr">

Provides stylesheets documentation which is presented in the Roxen CMS user interface. Use of HTML markup is permitted as shown in this example:

```
<?xml version="1.0"?>
<xsl:stylesheet>
  <rxml:helptext>
    This stylesheet generates recurring page elements such
    as headers and footers.
  </rxml:helptext>

  <!-- document <footer> tag and its attributes -->
  <rxml:helptext match="footer">
    Template rule which outputs the page footer.
  </rxml:helptext>

  <rxml:helptext match="footer/@align">
    Sets footer alignment to <tt>left</tt>, <tt>center</tt>,
    or <tt>right</tt>. Default value is <tt>left</tt>.
  </rxml:helptext>

  <!-- template rules -->
  <xsl:template match="footer">
    <hr/>
    <div align="left">
      <xsl:if test="@align">
        <xsl:attribute name="align" select="@align"/>
      </xsl:if>
      <p>Last page update: <date/>.</p>
    </div>
  </xsl:template>
</xsl:stylesheet>
```

15.4 Final words

We hope that this introduction has helped you in becoming familiar with XSLT in Roxen CMS. More tutorial and reference resources are available at www.xslt.com which is well worth a visit.