Sitecore CMS 6.2-6.4

Presentation Component XSL Reference

A Conceptual Overview for Developers

Table of Contents

Chapter 1

Introduction

This document provides an overview of common XML, XSL, and XPath concepts and syntax as used in Sitecore XSL renderings.[1] Sitecore developers and designers should read this document before implementing XSL renderings.

This document begins with an overview of XSL and a description of basic XSL and XPath constructs. This document then describes the Sitecore XML structure that most XSL renderings process. This document next discusses XSL and XPath in the Sitecore context before explaining how to implement custom .NET extensions to XSL.

This document contains the following chapters:

- Chapter 1 — Introduction
- Chapter 2 — Basic XSL and XPath Constructs
- Chapter 3 — The Sitecore XML Structure
- Chapter 4 — XSL and XPath with Sitecore
- Chapter 5 — Custom XSL Extension Libraries

[1] For more information about XSL, see http://www.w3.org/Style/XSL/. For more information about XML, see http://www.w3.org/XML/. For more information about XPath, see http://www.w3.org/TR/xpath. For more information about XSL renderings, see the Presentation Component Reference manual at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Reference.aspx. For information about troubleshooting XSL renderings, see the Presentation Component Troubleshooting Guide at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Troubleshooting%20Guide.aspx.

Chapter 2

Basic XSL and XPath Constructs

This chapter provides an overview of basic XML, XSL, and XPath concepts and syntax.

This chapter begins with an overview of XSL, and then describes specific XML, XSL, and XPath tokens. This chapter then documents common XSL elements before discussing XPath and XSL functions.

This chapter contains the following sections:

- XSL Overview
- XML, XSL, and XPath Tokens
- XSL Elements
- XPath and XSL Functions

2.1 XSL Overview

XSL is a declarative programming languages intended for transforming XML data sources into other formats, including HTML, text, other XML, or even binary formats.

XML is a tag-based, hierarchical data store similar to HTML markup, but with stricter syntax and greater flexibility in structure and content. XSL is a dialect of XML. XSL files are XML files that contain elements that XSL transformation engines recognize.

XSL renderings access an XML representation of a Sitecore database. For more information about the XML representation of a Sitecore database, see Chapter 3, The Sitecore XML Structure.

The XSL context element is the location of the XSL transformation engine in an XML document. For more information about the context element, see the section *The Context Element*.

XSL code uses XPath statements to select nodes in the XML document. For more information about XPath, see the sections *XML, XSL, and XPath Tokens* and *XPath Navigation*.

An XSL template is a block of code, similar to a procedure or function. You can invoke XSL templates by name, and you can invoke XSL templates by selecting nodes in the XML document that match specific criteria. For more information about XSL templates, see the section *The <xsl:template> , <xsl:call-template>, and <xsl:with-param>, and <xsl:apply-templates> XSL Elements*.

XPath statements include implicit and explicit use of axes. Axes are directions for traversing the XML document. For more information about axes, see the section *Implicit Relationships (XPath Axes)*.

Important
ASP.NET supports XSL 1.0. ASP.NET, and hence Sitecore, does not support XSL 1.1 or XSL 2.0. When researching XSL syntax, avoid constructs that are specific to XSL 1.1 or XSL 2.0.

Important
XSL looping constructs index elements starting with position `1` rather than `0`.

Note
You can use ASP.NET exclusively in favor of XSL. [2] ASP.NET provides a superset of the features available to XSL. Use .NET where XSL or XPath syntax is unwieldy, where XSL code is difficult to manage, where XSL performance is poor, and for components that involve significant logic.

Note
This document focuses on extensions to XSL for working with Sitecore content. Consult external documentation for more information on XML, XSL, and XPath.[3]

[2] For more information about the advantages and disadvantages of XSL and .NET rendering technologies, see the Presentation Component Reference manual at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Reference.aspx, the Presentation Component Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Cookbook.aspx, and the Presentation Component API Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20API%20Cookbook.aspx.

[3] Sitecore recommends the book XSLT: Programmer's Reference, by Michael Kay (http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764543814.html).

2.2 XML, XSL, and XPath Tokens

This document provides further detail about the tokens summarized in the following table, which commonly appear in XSL renderings.

Token	Meaning
`.`	A single *dot* character represents the self axis, which is the current location of the XSL transformation engine in an XML document (also known as the context element). XSL assumes the self axis; `@id`, `./@id`, and `self::@id` all select the attribute named `id` of the context element. Synonym: `self`.
`..`	Two *dot* characters represent the parent axis, which contains the parent element of the context element. Synonym: `parent`.
`/`	The *slash* character is the path operator, representing the root of the XML document (`/`) and separates elements from its attributes (`./@id`) and child elements (`./item`). Synonym: `child`.
`//`	Two *slash* characters represent the `descendant` axis, which selects all descendants of the context element. Synonym: `descendant`.
`@`	The *at* character specifies an attribute. For example, `@key` selects the attribute named `key`.
`*`	The *star* character matches any element. For example, `/*` selects the root element of any XML document.
`$`	The *dollar sign* character indicates a named parameter or variable, which can be a simple value or a location in the XML document. For example, `$sc_currentitem` represents the context item.
`[]`	The *square bracket* characters indicate an XPath predicate, which filter a selection of elements. For example, `./item` selects all children of the context element; `./item[@template='templatename']` selects the children of the context element that are based on the template named `templatename`.
`::`	Two *colons* represent the axis resolution operator, which separates an axis name from other tokens.
`&amp;`	XML-escaped ampersand (“`&`”).
`&lt;`	XML-escaped left angle bracket (“<”).
`&gt;`	XML-escaped right angle bracket (“>”).
`&quot;`	XML-escaped quote (").
`&apos;`	XML-escaped apostrophe (“'”).
`&####;`	XML-escaped character specified by hexadecimal value.
`and`	Logical and operator.
`or`	Logical or operator.
`*`	The **star** character is a wildcard that can select any node in the XML document.

Important
To optimize performance, avoid using any variant of the `descendant` axis.

Note
XML supports only the five named character entities listed above. In XSL, you cannot use other named entities as you would in HTML, such as ` `. Instead, use the corresponding numerical entity, such as ` `.[4]

[4] For a table mapping text entity codes to the corresponding numerical equivalents, see http://www.webmonkey.com/reference/Special_Characters.

2.3 XSL Elements

This section provides an overview of XSL elements commonly used in Sitecore XSL renderings. XSL elements use the `xsl` namespace.

2.3.1 The <xsl:variable> XSL Element

The `<xsl:variable>` XSL element creates a named variable. For example, each of the following will create the variable `$content` representing the `/Sitecore/Content` item (an `<item>` element in the XML document that represents a Sitecore database).

```
<xsl:variable name="content" select="/item[@key='sitecore']/item[@key='content']" />
<xsl:variable name="content" select="/*/item[@key='content']" />
<xsl:variable name="content" select="sc:item('/sitecore/content',.)" />
```

The first statement uses a fully-qualified XPath statement to select an element.

The second statement uses a wildcard to select a child of the root element.

The third statement uses a .NET XSL extension method to select the element at the specified path.

If you create a variable that represents a Sitecore item, you can retrieve values from that item, iterate over its children, or perform other operations on the item. The following example creates a variable named `$content`, and then iterates its children.

```
<xsl:variable name="content" select="sc:item('/sitecore/content',.)" />
<xsl:for-each select="$content/item">
  Child: <xsl:value-of select="@name" /><br />
</xsl:for-each>
```

Avoid the overhead of variables when possible. Store IDs instead of elements, and when necessary, reference `<item>` elements directly instead of using variables. For example, the following example achieves the same result as the previous example, without using a variable.

```
<xsl:for-each select="sc:item('/sitecore/content',.)/item">
  Child: <xsl:value-of select="@name" /><br />
</xsl:for-each>
```

Note
You cannot change the value of an XSL variable. You can sometimes work around this issue by passing variables as parameters to recursive XSL templates. For more information about XSL templates, see the section *The <xsl:template> , <xsl:call-template>, and <xsl:with-param>, and <xsl:apply-templates> XSL Elements*.

2.3.2 The <xsl:value-of> XSL Element

The `<xsl:value-of>` XSL element retrieves a value. You can use `<xsl:value-of>` to write a value to the output stream. For example, to write the raw value of the field named **FieldName** to the output stream:

```
<xsl:value-of select="sc:fld('FieldName',$sc_currentitem)" />
```

You can use `<xsl:value-of>` to populate a variable using a named template:

```
<xsl:template name="GetContentID">
  <xsl:value-of select="sc:item('/sitecore/content',.)/@id" />
</xsl:template>
...
<xsl:variable name="contentid">
  <xsl:call-template name="GetContentID" />
</xsl:variable>
```

You disable escaping of XML special characters using the `disable-output-escaping` attribute. For example, you can disable output escaping when you process a URL that contains multiple query string parameters separated by ampersand characters.

```
<xsl:value-of
  select="sc:fld('FieldName', $sc_currentitem)" disable-output-escaping="yes" />
```

Unless you set `disable-output-escaping` to `yes`, the XSL transformation engine encodes special characters in the source value. For example, the system outputs `&` for any ampersand ("`&`") characters in the source value.

2.3.3 The <xsl:if> XSL Element

The `<xsl:if>` XSL element invokes the enclosed code if the condition specified by the `test` attribute is True.

Because XSL treats Null and empty strings as False, a common shortcut to check whether a field exists and contains a value is to check whether the `sc:fld()` XSL extension method returns True.

```
<xsl:if test="sc:fld('FieldName',$sc_currentitem)">
  <!-- FieldName exists and contains a value in the specified item-->
</xsl:if>
```

Note
XSL does not include elements such as `<xsl:elseif>` or `<xsl:else>`. For multiple conditions, use `<xsl:choose>` as described in the section *The <xsl:choose, <xsl:when>, and <xsl:otherwise> XSL Elements*.

2.3.4 The <xsl:choose, <xsl:when>, and <xsl:otherwise> XSL Elements

The `<xsl:choose>` element processes the contents of the first `<xsl:when>` element with a `test` condition that evaluates to True, or the contents of the `<xsl:otherwise>` element if no `test` evaluates to True.

```
<xsl:variable name="random" select="sc:random(10)" />
<xsl:choose>
  <xsl:when test="$random &gt; 6">
    <!--random is greater than 6-->
  </xsl:when>
  <xsl:when test="$random &gt; 3">
    <!--random is greater than three but less than or equal to 6-->
  </xsl:when>
  <xsl:otherwise>
    <!--random is less than or equal to 3-->
  </xsl:otherwise>
</xsl:choose>
```

Note
For each `<xsl:choose>` element, the system will process the segment of code contained in only one `<xsl:when>` element or the `<xsl:otherwise>` element. If there is no `<xsl:otherwise>` element, and none of the `test` conditions evaluate to True, then the system will not process any code within the `<xsl:choose>` element.

2.3.5 The <xsl:for-each> XSL Element

The `<xsl:for-each>` XSL element iterates over zero or more elements as specified by the XPath expression in the `select` attribute. Within each iteration of the `<xsl:for-each>` element, the context element ("`.`") contains the selected element.

The following example iterates over the children of the context item, outputting the name of each:

```
<xsl:for-each select="$sc currentitem/item">
  <xsl:value-of select="@name" /><br />
</xsl:for-each>
```

2.3.6 The <xsl:sort> XSL Element

The `<xsl:sort>` XSL element sorts a list of elements, such as within `<xsl:for-each>`. For more information about `<xsl:for-each>`, see The <xsl:for-each> XSL Element.

The following example sorts the children of the context item in descending order by a date field value.

```
<xsl:for-each select="$sc_currentitem/item">
  <xsl:sort select="sc:fld('__updated',.)" order="descending" />
  <xsl:value-of select="@name" /><br />
</xsl:for-each>
```

You can reverse the default sort using the `@sortorder` attribute:

```
<xsl:for-each select="$sc currentitem/item">
  <xsl:sort select="@sortorder" order="descending" />
  <xsl:value-of select="@name" /><br />
</xsl:for-each>
```

Note
By default, items appear in XML in the same order that they appear in the content tree.

2.3.7 The <xsl:template> , <xsl:call-template>, and <xsl:with-param>, and <xsl:apply-templates> XSL Elements

The `<xsl:template>` XSL element encapsulates a segment of XSL code, similar to a method, procedure, or function in other languages. The XSL transformation engine may write the output generated by the content of an `<xsl:template>` element to the output stream, or encase that output in an XSL variable. In XSL, the term template refers to a segment of XSL code contained within an `<xsl:template>` element.

If you invoke a template using `<xsl:call-template>`, then the XSL transformation engine writes the output of the template to the output stream. For example:

```
<xsl:template name="TemplateName">
  <xsl:value-of select="$sc currentitem/@template" />
</xsl:template>
<xsl:call-template name="TemplateName" />
```

Alternatively, you can put the output of a template in a variable:

```
<xsl:variable name="VariableName">
  <xsl:call-template name="TemplateName" />
</xsl:variable>
```

You can pass parameters to an XSL template using the `<xsl:param>` and `<xsl:with-param>` XSL elements, and you can specify default values for parameters using the `select` attribute of `<xsl:param>`.

```
<xsl:template name="TemplateName">
  <xsl:param name="ParamName" select="$sc currentitem" />
  <xsl:value-of select="$ParamName/@template" />
</xsl:template>
...
<xsl:call-template name="TemplateName">
  <xsl:with-param name="ParamName" select="." />
</xsl:call-template>
```

Note
The context element within an XSL template is the context element from the calling context.

You can also use the `<xsl:apply-templates>` XSL element to invoke XSL templates using XPath statements to select items.

2.4 XPath and XSL Functions

This section provides an overview of functions available in XPath statements used in XSL programming.

2.4.1 The position() Function

The `position()` function returns the location of the element in a list. For more information about the `position()` function, see the section *The ancestor and ancestor-or-self Axes*.

A common pattern is to compare `position()` to `last()` to determine whether a loop has reached the last element, such as to insert a spacing element between output elements. For example, the following code outputs an HTML line break (`<br>`) after each link to a child item of the context item except for the last link in the list.

```
<xsl:for-each select="$sc_currentitem/item">
  <sc:link><sc:text field="FieldName" /></sc:link>
  <xsl:if test="position() != last()">
    <br />
  </xsl:if>
</xsl:for-each>
```

2.4.2 The last() Function

The `last()` function returns the number of elements in a list. For an example using the `last()` function, see the previous section, *The position() Function*.

2.4.3 The current() Function

The `current()` function represents the current element in the XML document For more information about the `current()` function, see *The Current Element*.

2.4.4 The document() Function

The `document()` function retrieves an external XML source, such as an RSS feed. The following sample code reformats an RSS feed.

```
<xsl:variable name="rssurl" select="'http://www.asp.net/news/rss.ashx'" />
<xsl:variable name="rss" select="document($rssurl)" />
<xsl:if test="$rss">
  <xsl:for-each select="$rss//item">
    <a>
      <xsl:attribute name="href">
        <xsl:value-of select="link"/>
      </xsl:attribute>
      <xsl:value-of select="title"/>
    </a>
    <p>
      <xsl:value-of select="description" />
    </p>
  </xsl:for-each>
</xsl:if>
```

Note
You can store values such as the URL of the RSS feed in a field of an item:

```
<xsl:variable name="rssurl" select="sc:fld('FieldName', $sc_currentitem)" />
```

2.4.5 The concat() Function

The `concat()` function concatenates strings.

```
        <xsl:variable name="VariableName" select="concat('A', 'B')" />
        <xsl:variable name="VariableName" select="concat(concat('A', 'B'), 'C')" />
        <xsl:variable name="VariableName" select="concat(concat(concat('A', 'B'), 'C'), 'D')"
/>
```

2.4.6 The translate() Function

The `translate()` function converts characters to alternate characters or removes characters from a value. For example, the following outputs the raw value of the field named **FieldName** after replacing hyphen characters ("-") with underscore characters ("_"):

```
        <xsl:value-of select="translate(sc:fld('FieldName', $sc_currentitem), '-', '_')" />
```

You can use the `translate()` function to convert a string representation of a date or date and time to a number. Sitecore stores date and time values using the ISO date format corresponding to the .NET format pattern `yyyyMMddTHHmmss`, where `T` is a literal character that separates the date portion of the value from the time portion. To convert a date in this ISO format to a number for comparisons or other purposes, you can remove the `T` character using the `translate()` function.

```
        <xsl:variable name="updated"
          select="translate(sc:fld('__updated', $sc_currentitem), 'T', '')" />
```

2.4.7 The true() Function

The `true()` function returns a True value. You can use this function to inverse a value.

2.4.8 The false() Function

The `false()` function returns a false value. You can use this function to inverse a value.

2.4.9 The not() Function

The `not()` function negates a condition. The following condition is never true:

```
        <xsl:if test="not(true())">
          <!--the XSL transformation engine will never invoke this code.-->
        </xsl:if>
```

2.4.10 The count() Function

The `count()` function returns the number of elements in a list. For example, to determine if the context item has more than five child `<item>` elements:

```
        <xsl:if test="count($sc_currentitem/item) &gt; 5">
```

2.4.11 The contains() Function

The `contains()` function returns True if the first argument contains the second argument.

You can use the `contains()` function to determine if a list of IDs, such as that stored in selection field, contains the ID of a specific item. For example, to determine if the **FieldName** field in the context item contains the ID of the context item:

```
        <xsl:variable name="ids" select="sc:fld('FieldName', $sc currentitem/@id)" />
        <xsl:if test="contains($ids, $sc_currentitem/@id)">
          <!--The specified field in the context item contains the ID of the context item-->
        </xsl:if>
```

You can implement this condition without creating a variable:

```
        <xsl:if test="contains(sc:fld('FieldName', $sc_currentitem), $sc_currentitem/@id)">
          <!--The specified field in the context item contains the ID of the context item-->
        </xsl:if>
```

Chapter 3

The Sitecore XML Structure

This chapter provides an overview of the XML representation of a Sitecore database that XSL renderings use.

This chapter begins by describing how to access items in the XML document. This chapter then explains how to use XPath to navigate around the XML document and select items in that document.

This chapter contains the following sections:

- Working with Items
- XPath Navigation
- Selecting Items

3.1 Working with Items

XSL renderings access an XML document that represents the structure of a Sitecore database.

Important
Use your knowledge of the information architecture and the Sitecore XSL extensions described in this document rather than investigating the raw XML representation of a Sitecore database.

The XML document consists of a hierarchy of `<item>` elements. Each `<item>` element corresponds to an item in the Sitecore content tree. Each `<item>` element has attributes and contains a collection of elements that define the fields of the item.

Each `<item>` can contain any number of child `<item>` elements. The XML document contains <item> elements nested as deeply as you nest items in the content tree.

The root `<item>` element in the XML document represents the `/Sitecore` item in the content tree. The following XML fragment includes some of the data in the default `/Sitecore` and `/Sitecore/Content` items. All `<item>` elements have a similar structure.

```
        <item name="sitecore" key="sitecore" id="{11111111-1111-1111-1111-111111111111}"
          tid="{C6576836-910C-4A3D-BA03-C277DBD3B827}"
          mid="{00000000-0000-0000-0000-000000000000}"
          sortorder="100" language="en" version="1" template="root"
          parentid="{00000000-0000-0000-0000-000000000000}">
          <fields>
            <field tfid="{5DD74568-4D4B-44C1-B513-0AF5F4CDA34F}" key="  created by"
            type="text" />
            <!-- additional field definition elements -->
           <field tfid="{9C6106EA-7A5A-48E2-8CAD-F0F693B1E2D4}" key="  read only"
            type="checkbox" />
          </fields>
          <item name="content" key="content" id="{0DE95AE4-41AB-4D01-9EB0-67441B7C2450}"
          tid="{E3E2D58C-DF95-4230-ADC9-279924CECE84}"
          mid="{00000000-0000-0000-0000-000000000000}"
          sortorder="100" language="en" version="1" template="main section"
          parentid="{11111111-1111-1111-1111-111111111111}">
            <fields>
              <field tfid="{BADD9CF9-53E0-4D0C-BCC0-2D784C282F6A}" key="  updated by"
              type="text"  />
              <!-- additional field definition elements -->
              </field>
            </fields>
            <!— item elements at this level represent children of /Sitecore/Content-->
            ...
          </item>
          <!-- additional item elements at this level represent siblings of /Sitecore/Content-
->
        </item>
```

3.1.1 Item Attributes

Each `<item>` element has a specific set of attributes, including:

- **`@name`**: The name of the item.
- **`@key`**: The lowercase of the name of the item.
- **`@id`**: The ID of the item.
- **`@tid`**: The ID of the data template definition item associated with the item.
- **`@mid`**: The ID of the branch template or command template used to insert the item, or the null GUID.
- **`@sortorder`**: The numerical sort order of the item relative to its sibling items.
- **`@language`**: The context language.

- **`@version`**: The version number of the item within the language.
- **`@template`**: The lowercase name of the data template associated with the item.
- **`@parentid`**: The ID of the parent of the item, or the null GUID for the root item.

Warning
For best performance, always compare the values of `@id` attributes to determine whether two elements represent the same item. For example, test whether `$sc_currentitem/@id = $sc_item/@id`, not whether `$sc_currentitem = $sc_item`.

3.1.2 Item Fields

Each `<item>` element in the XML representation of a Sitecore database contains a `<fields>` element. Each `<fields>` element contains a number of `<field>` elements. Each `<field>` element represents a field definition in the data template associated with the item, or one of the base templates associated with that template, including the standard template.

Note
For efficiency, the XML representation of a Sitecore database does not contain field values; `<field>` elements do not contain values. For information about XSL extensions that you can use to access field values, see the sections *The <sc:date> XSL Extension Control, The <sc:html> XSL Extension Control, The <sc:image> XSL Extension Control, The <sc:link> XSL Extension Control, The <sc:memo> XSL Extension Control, The <sc:text> XSL Extension Control, The <sc:wordStyle> XSL Extension Control, The sc:fld() XSL Extension Method, and The sc:field() XSL Extension Method*.

Each `<field>` element has a specific set of attributes including:

- **`@tfid`**: The ID of the data template field definition item.
- **`@key`**: The lowercase name of the data template field definition item.
- **`@type`**: The lowercase name of the data template field data type.

3.2 XPath Navigation

You can use XPath to navigating through the XML document available to XSL renderings.

3.2.1 Specific Items

XSL renderings define parameters that you can use to access items.

The Context Item: $sc_currentitem

The context item is the item that corresponds to the path in the URL requested by the client. The context item is the default data source for all renderings for which you do not specify a data source. The `$sc_currentitem` variable represents the `<item>` element that corresponds to the context item in the XML representation of a Sitecore database.

The Data Source Item : $sc_item

A rendering can retrieve data from its data source item. The `$sc_item` variable represents the data source item for an XSL rendering. If the developer does not specify a data source item for a rendering, the default data source item is the context item, and `$sc_item` and `$sc_currentitem` are the same item.

XSL rendering logic begins with the first `<xsl:template>` XSL element in the code file. The XSL rendering boilerplate file uses the `select` attribute of the `<xsl:apply-templates>` XSL element to set the context element to the data source item and invoke the XSL template with `mode` attribute `main`:

```
<xsl:template match="*">
  <xsl:apply-templates select="$sc item" mode="main"/>
</xsl:template>
<xsl:template match="*" mode="main">
  <!--the context element is the data source item-->
  <!--developers typically insert code here-->
</xsl:template>
```

The Context Element

The context element is the location of the XSL transformation engine within an XML document. The context element is the location from which the XSL transformation engine interprets relative XPath statements. The dot character (“.”) represents the context element. XSL constructs such as `<xsl:for-each>` change the context element, but do not change the context item (`$sc_currentitem`).

The Current Element

The `current()` function returns the current element, which you can use in XPath predicates to access the context element at the current scope. Within the predicate of an XPath statement, the current element is the element that was the context element at the point that the XSL transformation engine began evaluating the XPath statement. Otherwise, the current element and the context element are often the same element. Consider the following example:

```
<xsl:for-each select="$sc_item/item">
  <xsl:choose>
    <xsl:when test="$sc currentitem/ancestor-or-self::item[@id=current()/@id">
      <!—the context element is the iteration item or one of its descendants-->
    </xsl:when>
  </xsl:choose>
</xsl:for-each>
```

The outer `<xsl:for-each>` element iterates over the `<item>` elements that are children of the data source item. Within the outer `<xsl:for-each>`, the context element is a child of the data source item, and the `current()` function returns that item. The `<xsl:when>` element tests if the context

item (`$sc_currentitem`) is or has an ancestor that is the context element (`current()`), for instance to determine if the context item is within that section of the information architecture. Within the predicate, the context element is the `<item>` element on the `ancestor-or-self` axis, while the current element remains the child of the data source item.

Important
It is important to understand the difference between the context item, the context element, and the current element, which may all represent the same `<item>`. The context item is the item requested by the browser. The context element is the location of the XML transformation engine in an XML document. The context item is the default context element for renderings that do not have an explicit data source. The context element is generally an `<item>` in an XML document that represents a Sitecore database, but could be any type of element and could be in a different XML document. The current element refers to the context element except in the predicate of looping constructs, where the current element is the element that was the context element at the opening of the loop.

Item Variables Using XPath

You can reference any item using a fully-qualified XPath statement. For instance, you can create a variable representing the `/Sitecore/Content` item using the following XSL construct:

```
<xsl:variable name="content" select="/item[@key='sitecore']/item[@key='content']" />
```

Note
Because an XML document always has exactly one root element, you can shorten this expression using the star character ("`*`"), or wildcard, to match that root element:

```
<xsl:variable name="content" select="/*/item[@key='content']" />
```

Item Variables Using sc:item()

You can access an item by passing its path or ID to the `sc:item()` XSL extension method:

```
<xsl:variable name="content" select="sc:item('/sitecore/content',.)"/>
<xsl:variable name="content"
  select="sc:item('{110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9}',.)" />
<xsl:for-each select="$content/item">
  <xsl:value-of select="@name" /><br />
</xsl:for-each>
```

Pass Items to XSL Renderings Using the <xsl:param> XSL Element

You can pass additional item paths or IDs to XSL renderings using the `<xsl:param>` XSL element, and use the `sc:item()` XSL extension method to select the corresponding items.[5] Add rendering parameter definitions in the header of the XSL rendering near the default parameter definitions.

```
<xsl:param name="ParamNamePathOrID"><!--default parameter value--></xsl:param>
<xsl:variable name="VariableName" selet="sc:item($ParamNamePathOrID, .)" />
```

3.2.2 Item References

An item can contain fields that contain the IDs of other Sitecore items, representing references from one item to another.

If a field contains a single ID, then you can use the `sc:fld()` XSL extension method to retrieve that ID, and pass that ID to the `sc:item()` XSL extension method to select the corresponding `<item>` element.

[5] For more information about passing parameters to renderings, see the Presentation Component Reference manual at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Reference.aspx.

```
<xsl:variable name='IDVariableName' select='sc:fld('FieldName", $sc_item)' />
<xsl:if test="$IDVariableName">
  <xsl:variable name='ItemVariableName' select='sc:item($IDVariableName, $sc item)' />
  <xsl:if test="$ItemVariableName">
    <xsl:value-of select="$ItemVariableName/@name" />
  </xsl:if>
</xsl:if>
```

If a field can contain a list of IDs referencing multiple items, then you can use the `sc:SplitFieldValue()` XSL extension method to iterate over that ordered list.

```
<xsl:for-each select="sc:SplitFieldValue('FieldName',.) ">
  <xsl:for-each select="sc:item(text(),.)">
    <xsl:value-of select="@name" /><br />
  </xsl:for-each>
</xsl:for-each>
```

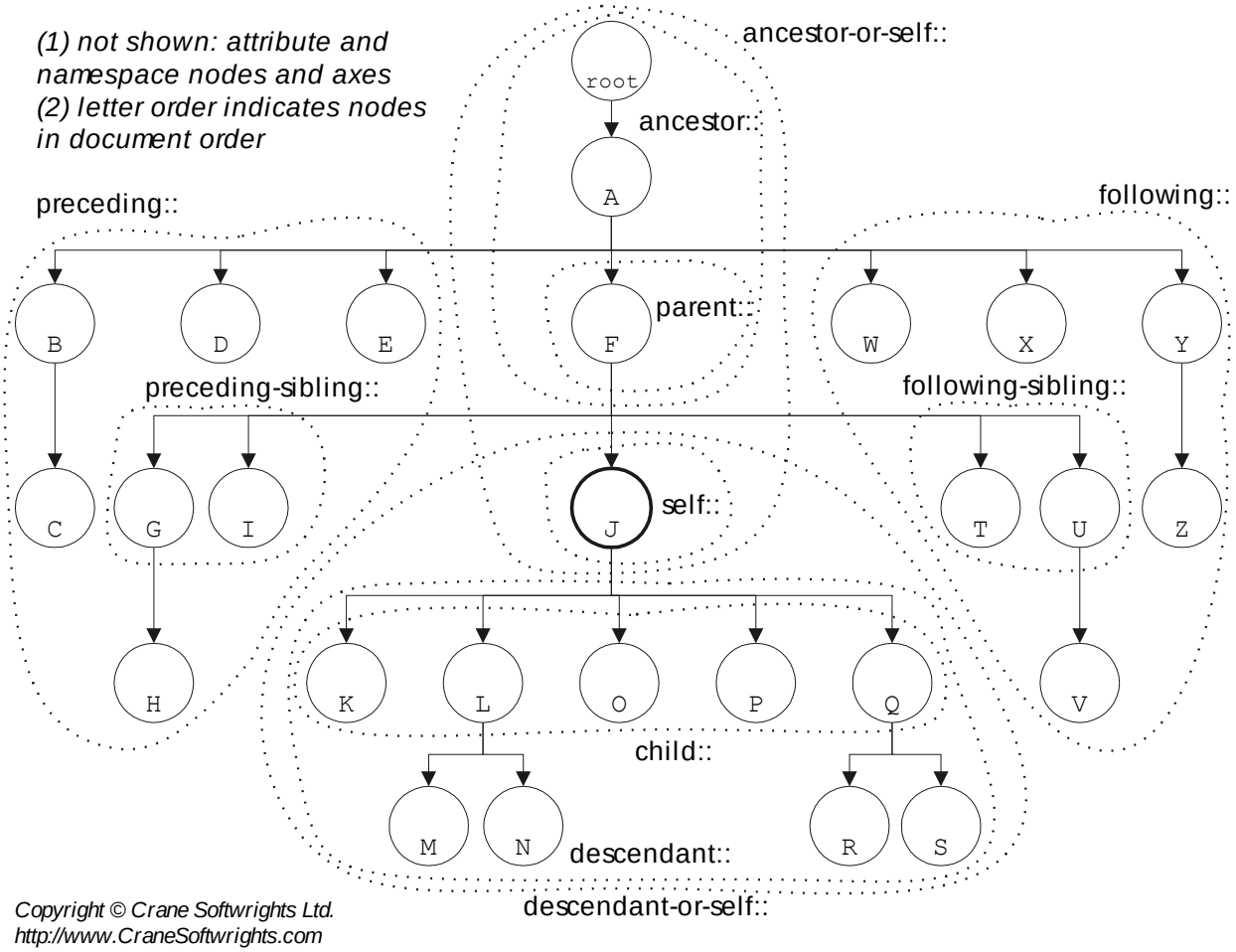
Note
Use `<xsl:for-each>` to process the item returned by the `sc:item()` XSL extension function sets context element, resulting in shorter, more flexible, and more consistent syntax.

Tip
You can use the `sc:SplitFieldValue()` XSL extension method to process a field that contains the ID of a single item as well as fields that contain the IDs of multiple items.

3.2.3 Implicit Relationships (XPath Axes)

Each `<item>` element in an XML document that represents a Sitecore database has a number of implicit relationships with other `<item>` elements in that document through the various XPath axes. The following diagram, in which J represents the context element, provides an overview of the various XPath axes.[6]

[6] Published with permission of Crane Softwrights Ltd., http://www.CraneSofwrights.com.



The Self Axis

The `self` axis contains only the context element. In XPath expressions, the dot character (“`.`”) represents the `self` axis. The `self` axis is the default axis if you do not explicitly specify an axis. Developers typically reference the context element implicitly or explicitly rather than using the `self` axis explicitly. The following three constructs are equivalent, with the shortest being generally preferable.

```
<xsl:value-of select="@id" />
<xsl:value-of select="./@id" />
<xsl:value-of select="self::*/@id" />
```

The XPath statement in the `select` attribute in the first example does not specify an axis, and therefore implicitly retrieves the value of an attribute of the context element. The second example explicitly references the context element (`.`), and uses the path operator (`/`) to select a named attribute of that item. The third example explicitly matches any element (`*`) on the self axis (`self::`), and retrieves the value of the attribute named `id` from any such element.

The Child Axis

The `child` axis contains the children of the context element. In XPath expressions, the path operator (“`/`”) represents the `child` axis.

Each `<item>` element can contain nested `<item>` elements representing the children of the item. To exclude the `<fields>` element when selecting the children of an item, explicitly select child elements named item. In each of the following XPath statements, the token `item` matches `<item>` elements.

```
<xsl:for-each select="./item">
<xsl:for-each select="item">
<xsl:for-each select="./child::item" />
<xsl:for-each select="child::item">
```

The first example matches `<item>` elements that are children of the context element. The second example matches the same items, demonstrating that XPath defaults to processing the child axis from the context element. The third example matches `<item>` elements on the child axis of the context element. The fourth example matches `<item>` elements on the child axis of the context element.

You can use a construct such as the following to determine whether an item has children:

```
<xsl:if test="$sc_currentitem/item">
  <!--the context item has child items-->
</xsl:if>
```

The Parent Axis

The `parent` axis includes the parent of the context element. Except for the root element, each element in an XML document has a parent element. In XPath expressions, two dots (“`..`”) represent the `parent` axis.

The following examples test whether the name of the data template associated with the parent item of the context element is `homepage`.

```
<xsl:if test="./parent::item[@template='homepage']">
<xsl:if test="parent::item[@template='homepage']">
<xsl:if test="parent::*[@template='homepage']">
<xsl:if test="./../[@template='homepage']">
<xsl:if test="../[@template='homepage']">
```

Note
The following expression determines not whether the parent item uses the specified data template, but whether that parent item has a child that uses that data template.

```
<xsl:if test="../item[@template='homepage']">
```

The ancestor and ancestor-or-self Axes

The `ancestor` axis returns the ancestors of the context element in document order. Document order refers to the order of elements in the XML document, from the root element down. The `ancestor-or-self` axis returns the context element and its ancestors in document order.

In the diagram shown in the section *Implicit Relationships (XPath Axes)*, if the context item is S, then the `ancestor` axis includes A, F, J, and Q, while the `ancestor-or-self` axis includes A, F, J, Q, and S.

XSL renderings often use the `ancestor` and `ancestor-or-self` axes to process the items that enclose another item or to determine whether an item is a descendant of another item, such as to generate a breadcrumb or highlight a navigational element corresponding to an item that contains the context item.

Within the `select` attribute of the `<xsl:for-each>` element, the `position()` function references the index of the element in the list of elements. For the `ancestor` and `ancestor-or-self` axes, this is the opposite of XML document order. In the predicate of the `select` attribute of the `<xsl:for-each>` element, the `last()` function returns the number of elements in the list of elements to process.

The `ancestor` axis is often useful in breadcrumbs. Breadcrumbs should never include links to A and F, which represent the `/Sitecore` and `/Sitecore/Content` items. Breadcrumbs typically include the name of the context item, but that step in the breadcrumb is typically not a link. The `ancestor` axes excludes the context item. To exclude A and F, ignore the two furthest elements from the context item:

```
<xsl:for-each select="$sc_currentitem/ancestor::item[position() &lt; last() - 1]">
```

You can use the `ancestor-or-self` axis to determine if an item is an ancestor of another, or is itself that item. For instance, a rendering might iterate over the sections beneath the home item, generating links to each section and highlighting the section containing the current page.

```
<xsl:for-each select="$home/item[@template='section']">
  <xsl:choose>
    <xsl:when test="$sc currentitem/@id=@id">
      <!--the client has requested this section item-->
    </xsl:when>
    <xsl:when test="$sc_currentitem/ancestor-or-self::item[@id=current()/@id">
      <!--the client has requested an item that is a descendant of this item-->
    </xsl:when>
    <xsl:otherwise>
      <!--the client has not requested this item or any of its descendants-->
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
```

You can use of the `ancestor-or-self` axis to retrieve field values from an item or its nearest ancestor that contains a value for that field. The following code processes the field named **FieldName** in the context item or its nearest ancestor that defines a value for that field.

```
<sc:image field="FieldName"
  select="$sc_currentitem/ancestor-or-self::item[sc:fld('FieldName',.,'src')][1]" />
```

The token `$sc_currentitem/ancestor-or-self::item` causes the XSL transformation engine to evaluate the predicate against the context element and each of its ancestor `<item>` elements. The first predicate restricts the selection to only those `<item>` elements that define a value for the **FieldName** field (`[sc:fld('FieldName',.,'src')]`). The second predicate (`[1]`) selects the matching `<item>` element that is closest to the context item. The `<sc:image>` control generates an HTML `<img>` tag using the specified field value in that item.

Descendants and Recursion

The `descendant` axis includes all descendants of an element, recursively, in document order. The `descendant-or-self` axis includes the context element and all of its descendants, recursively in document order.

In the diagram shown in the section *Implicit Relationships (XPath Axes)*, if the context element is J, then the `descendant` axis includes the K, L, M, N, O, P, Q, R, and S elements, while the `descendant-or-self` axis includes the J, K, L, M, N, O, P, Q, R, and S elements.

Recursion, including the `descendant` and `descendant-or-self` axes, can be expensive, but can also be effective, especially for small amounts of data. The following example generates a data-driven site map.

```
<xsl:call-template name="SiteMapStep" />
...
<xsl:template name="SiteMapStep">
  <xsl:param name="level" select="1" />
  <xsl:param name="start" select="$home" />
  <ul class="{concat('sitemap',$level)}">
    <xsl:for-each select="$start/item">
      <li>
        <sc:link><sc:text field="title" /></sc:link>
        <xsl:if test="item">
          <xsl:call-template name="SiteMapStep">
            <xsl:with-param name="level" select="$level+1" />
            <xsl:with-param name="start" select="." />
          </xsl:call-template>
```

```
            </xsl:if>
          </li>
        </xsl:for-each>
      </ul>
    </xsl:template>
```

Warning
For performance, avoid excessive use of the `descendant` and `descendant-or-self` axes including the `//` construct, especially to process large numbers of items. Structure the information architecture and user search indexes and other features to limit the number of items processed by any single piece of code.

3.3 Selecting Items

This section provides techniques for selecting items to process.

3.3.1 How to Select Items Based on a Specific Data Template

You can use the `@template` attribute in the XPath predicate to process items that are based on a specific data template.

```
<xsl:for-each select="$sc_item/item[@template='templatename']">
```

Important
The `@template` attribute contains the data template key, which is the lowercase of the name of the data template.

To process items based on one or more templates, consider the XPath `contains()` function.

```
<xsl:for-each select="$sc item/item[contains('!templatename1!templatename2!',
  concat(concat('!',@template),'!')]">
```

To process items based on data templates that share a common base template:

```
<xsl:for-each select="$sc_currentitem/item[sc:IsItemOfType('basetemplate',.)]">
```

To process items based on data templates that share a common base template, including items based directly on that base template:

```
<xsl:for-each select="$sc currentitem/item[@template='basetemplate' or
  sc:IsItemOfType('basetemplate',.)]">
```

3.3.2 How to Select Items with a Version in the Context Language

In some sites supporting multiple languages, CMS users do not translate every item before publishing. To select items that have a version for the context language, check for a value in the creation date field. For example:

```
<xsl:for-each select="$sc currentitem/item[sc:fld('  created',.)]">
  <!--the context element is an item with a version in the context language-->
</xsl:for-each>
```

3.3.3 How to Select Items that Have Children

To select items that have children, include a predicate that specifies the existence of child `<item>` elements. For example, to process all the children of the context item that have at least one child item:

```
<xsl:for-each select="$sc currentitem/item[item]">
  <!--the context element, a child of the context item, has at least one child item-->
</xsl:for-each>
```

To process all children of the context item that have at least one child item based on or that inherits from a specific data template that has a version for the context language:

```
<xsl:for-each select="$sc currentitem/item[item[@template='basetemplate' or
  sc:IsItemOfType('basetemplate',.)) and sc:fld('__created',.)]]">
```

Chapter 4

XSL and XPath with Sitecore

This chapter describes considerations and techniques for developers working with XSL renderings, including XSL, XPath, and Sitecore XSL extensions written in .NET.

This chapter first describes the boilerplate file that you can use to create new XSL renderings. This chapter then documents how Sitecore manages errors in XSL renderings. The following section describes how to access fields in Sitecore items. The remaining chapters describe XSL extension methods and XSL extension controls.

This chapter contains the following sections:

- The Sitecore XSL Boilerplate File
- XSL Error Management
- Working with Fields
- Overview of XSL Extension Controls and Methods
- Sitecore XSL Extension Controls
- Sitecore XSL Extension Methods

4.1 The Sitecore XSL Boilerplate File

When you use the create a new XSL rendering in the **Developer Center**, Sitecore duplicates the XSL rendering boilerplate file, which provides a starting point for the new code.

The boilerplate file for new XSL renderings contains the following lines.

```
<?xml version="1.0" encoding="UTF-8"?>
```

This line indicates that the file contains XML. XSL rendering files contain XSL code. XSL is a dialect of XML; all XSL files are XML files.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sc="http://www.sitecore.net/sc"
  xmlns:dot="http://www.sitecore.net/dot"
  exclude-result-prefixes="dot sc">
```

This line, which contains the XML root element of the XSL file, identifies the file contents as XSL code. Similar to how ASP.NET maps tag prefixes to assemblies containing controls, XSL maps namespace identifiers to URLs containing XSL processing facilities. The `xsl` namespace exposes the XSL language. Sitecore by default defines the `sc` and `dot` namespaces, which correspond to .NET classes in assemblies installed by Sitecore. The `/configuration/sitecore/xslExtension/extension` elements in `web.config` provide signature for each class, effectively mapping XSL namespaces to .NET assemblies. For more information about .NET XSL extensions, see the Chapter 5, Custom XSL Extension Libraries.

```
<xsl:output method="html" indent="no" encoding="UTF-8" />
```

This line indicates that the rendering outputs markup using the HTML syntax, which does not require closing elements for `<hr>` and other elements. For an XHTML site, the value of the `method` attribute should be `xml`, resulting in output such as `<hr />` or `<hr></hr>`.

```
<xsl:param name="lang" select="'en'"/>
<xsl:param name="id" select="''"/>
<xsl:param name="sc item"/>
<xsl:param name="sc_currentitem"/>
```

These lines define several parameters that Sitecore passes to all XSL renderings:

- **`$lang`**: The context language.
- **`$id`**: The GUID of the data source item for the rendering.
- **`$sc_item`**: The data source item for the rendering.
- **`$sc_currentitem`**: The context item.

```
<!--<xsl:variable name="home" select="sc:item('/sitecore/content/home',.)" />-->
<!--<xsl:variable name="home" select="/*/item[@key='content']/item[@key='home']" />-->
<!--<xsl:variable name="home"
  select="$sc_currentitem/ancestor-or-self::item[@template='site root']" />-->
```

These lines demonstrate three ways to access the home item of the context site.[7]

```
<xsl:template match="*">
  <xsl:apply-templates select="$sc item" mode="main"/>
</xsl:template>
```

These lines set the context element to the data source of the rendering (`$sc_item`) before invoking the following XSL template with a value of `main` for the `mode` attribute.

```
<xsl:template match="*" mode="main">

</xsl:template>
```

Developers generally add code within this `<xsl:template>` element.

[7] For an example of using logic to determine the home item of the context site, see the `GetHomeItem()` method at http://trac.sitecore.net/XslHelper/browser/Trunk/Xml/Xsl/XslHelper.cs.

```
        </xsl:stylesheet>
```

This line closes the <xsl:stylesheet> root element.

4.2 XSL Error Management

If the layout engine encounters a syntax error or an exception while processing an XSL rendering, it embeds information about the error in the output stream, which is visible to Web clients.[8]

The layout engine does not compile XSL renderings until runtime. XSL does not support compile-time error detection.

Extensions to XSL written in .NET can throw exceptions. If an XSL rendering calls an extension method that throws an exception, then the layout engine adds information about that exception to the output stream, which is visible to Web clients. This output appears after any output generated by the rendering prior to the exception. XSL renderings do not write to the output stream after encountering an exception.

[8] For more information about handling errors with Sitecore, including how to override error management, see the Presentation Component API Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20API%20Cookbook.aspx.

4.3 Working with Fields

The following table summarizes the Sitecore XSL extension controls and methods that you can use to access the various data template field types in XSL renderings as described further in the following sections of this document.

Field Types	Construct	Notes
Attachment Icon IFrame Integer Internal Link Layout Number Password Security Template Field Source Tristate	`N/A`	Do not use XSL to process these field types.
Checkbox Droplist File Grouped Droplist	`sc:fld()`	You can use the `sc:fld()` XSL extension method to process these types of fields.
Checklist Multilist Treelist TreelistEx	`sc:SplitFieldValue()` and `sc:item()`	You can use the `sc:SplitFieldValue()` XSL extension method with the `sc:item()` XSL extension method to retrieve the items referenced by a field that allows the user to select multiple items.
Date DateTime	`<sc:date>`	You can use the `<sc:date>` XSL extension control to process Date and Datetime fields.
Droplink Droptree Grouped Droplink Internal Link	`sc:fld()` and `sc:item()`	You can use the `sc:item()` XSL extension method with the `sc:fld()` XSL extension method to retrieve the item referenced by a field that allows the user to select a single item.
File Drop Area (FDA)	`sc:fld()` and `sc:item()`	You can use the `sc:fld()` XSL extension method to retrieve `mediaid` attribute of the FDA field and the `sc:item()` XSL extension method to access the children of that item. For more information about accessing FDA fields, see the following section, *File Drop Area Fields*.
General Link	`<sc:link>`	You can use the `<sc:link>` XSL extension control to process General Link fields.

Field Types	Construct	Notes
Image	`<sc:image>`	You can use the `<sc:image>` XSL extension control to process Image fields.
Multi-Line Text	`<sc:memo>`	You can use the `<sc:memo>` XSL extension control to process Multi-Line text fields.
Rich Text Editor (RTE)	`<sc:html>`	You can use the `<sc:html>` XSL extension control to process Rich Text Editor fields.
Single-Line Text	`<sc:text>`	You can use the `<sc:text>` XSL extension control to process Single-Line Text fields.
Word Document	`<sc:text>`	You can use the `<sc:text>` XSL extension control to process Word Document fields.

Note
You can use the `sc:fld()` and `sc:field()` XSL extension methods to access any type of field.

Note
You can use the `<sc:wordstyle>` XSL extension element to embed an HTML `<style>` element containing the CSS styles required to support each field of type Word Document. For more information about the <sc:wordStyle> XSL extension control, see the section *The <sc:wordStyle> XSL Extension Control*.

Tip
To view the value stored in a field, such as the attributes available for an Image, File, or General Link field, view raw field values.[9]

4.3.1 File Drop Area Fields

You can implement code based on the following example that accesses the media items in the FDA File Drop Area (FDA) field named **FileDopAreaField**.

```
<xsl:variable name="folderid"
  select="sc:fld('filedropareafield', $sc item, 'mediaid')" />
<xsl:if test="$folderid and sc:item($folderid, $sc item)/item">
  <xsl:for-each select="sc:item($folderid, $sc item)/item">
    <a href="{sc:GetMediaUrl(.)}">
      <xsl:value-of select="@name"/>
    </a>
  </xsl:for-each>
</xsl:if>
```

The `$folderid` variable contains the GUID of the attribute named `mediaid` in the FDA field named **FileDropAreaField** in the data source for the rendering. If the value of the **FileDropAreaField** is not empty, and the corresponding item has children, then this code generates links to each of those children.

[9] For instructions to access raw field values, see the Client Configuration Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206/Client%20Configuration%20Cookbook.aspx.

4.4 Overview of XSL Extension Controls and Methods

XSL extensions expose .NET logic to XSL renderings. There are two types of XSL extensions: XSL extension controls and XSL extension methods.

XSL extension controls are XML elements in XSL renderings that correspond to .NET classes. For example, the `<sc:text>` XSL extension control corresponds to the `Sitecore.Web.UI.XslControls.Text` .NET class. XSL extension controls are standalone elements in the XSL code.

XSL extension methods correspond to methods in a .NET class. For example, the `sc:fld()` XSL extension method corresponds to the `fld()` method of the `Sitecore.Xml.Xsl.XslHelper` class represented by the `sc` namespace. XSL extension methods appear within attribute values and cannot stand alone as XML elements.

In general, XSL extension methods are easier to write, more flexible to use, and expose more functionality than XSL extension controls more efficiently. XSL extension controls result code syntax more consistent with XSL.

Note
Unless otherwise specified, all XSL extensions controls and methods work with the current version of each item in the context language.

4.5 Sitecore XSL Extension Controls

Sitecore provides several .NET XSL extension controls that you can use to simplify various rendering functions.

4.5.1 Common Attributes

Several of the XSL extension controls described in this chapter accept the common attributes described in the following sections.

The field Attribute

The field attribute specifies the name of a field to process.

```
        <sc:text field="FieldName" />
```

The select Attribute

The `select` attribute specifies the item on which the control operates. If you do not specify a value for the `select` attribute, the control operates on the context element.

```
        <sc:text field="FieldName" select="$sc_item" />
```

The show-title-when-blank Attribute

If the field value is empty and the `show-title-when-blank` attribute is `true`, then the layout engine outputs the name of the data template field before the inline editing control in the Page Editor. This can help CMS users locate empty fields.

```
        <sc:text field="FieldName" show-title-when-blank="true" select="$sc_currentitem" />
```

The disable-web-editing Attribute

If the `disable-web-editing` attribute is `true`, the layout engine disables inline editing for the field in the Page Editor. You can also disable inline editing by passing a third parameter to the `sc:field()` XSL extension method. The following two statements are equivalent.

```
        <sc:text field="FieldName" disable-web-editing="true" select="$sc_currentitem" />
        <xsl:value-of select="sc:field('FieldName', $sc_currentitem, 'disable-web-
editing=true')"
          disable-output-escaping="yes" />
```

Arbitrary Attributes

The `<sc:image>` and `<sc:link>` XSL extension controls pass any unrecognized attributes to the HTML element they generate.

```
        <sc:image border="1" ...
        <sc:link class="ClassName" ...
```

4.5.2 The Sitecore XSL Extension Controls

This section describes the individual Sitecore XSL extension controls.

Note
The `<sc:html>`, `<sc:memo>`, and `<sc:text>` XSL extension controls are very similar. The only significant implementation difference between these XSL extension controls is that `<sc:memo>` supports the `line-breaks` attribute.

The <sc:date> XSL Extension Control

The `<sc:date>` XSL extension control outputs the value of a Date or Datetime field.

The <sc:date> XSL extension control requires the following attribute:

- **field**: The name of the Date or Datetime field. For more information about the field attribute, see the preceding section, *The field Attribute*.

The <sc:date> XSL extension control accepts the following optional attributes:

- **format**: The .NET format pattern.[10]
- **select**: The item containing the Date or Datetime field. For more information about the select attribute, see the preceding section, *The select Attribute*.
- **disable-web-editing**: Enables or disables inline editing. For more information about the disable-web-editing attribute, see the preceding section, *The disable-web-editing Attribute*.

The sc:formatdate() XSL extension method provides functionality equivalent to that of the <sc:date> XSL extension control.

```
<sc:date field="FieldName" format="d" select="$sc_currentitem" />
<xsl:value-of select="sc:formatdate(sc:fld("FieldName",$sc_currentitem),'d')" />
```

The <sc:editFrame> XSL Extension Control

The <sc:editFrame> XSL extension control inserts an edit frame.[11]

Note
This document does not describe the <sc:editFrame> XSL extension control.

The <sc:dot> XSL Extension Control

The <sc:dot> XSL extension control generates a content marker in the Page Editor. Content markers support editing content while browsing a virtual copy of the web site.

The <sc:dot> XSL extension control does not require any attributes.

The <sc:dot> XSL extension control accepts the following optional attribute:

- **select**: The item to associate with the content marker. For more information about the select attribute, see the preceding section, *The select Attribute*.

The dot:Render() XSL extension method provides functionality equivalent to that of the <sc:dot> XSL extension control.

```
<sc:dot select="$sc_currentitem" />
<xsl:value-of select="dot:Render($sc_currentitem)" />
```

Important
Content markers are obsolete. When possible, enable inline editing instead of using content markers.

The <sc:html> XSL Extension Control

The <sc:html> XSL extension control outputs the value of a field of type Rich Text Editor or Word Document.

[10] For information about .NET date format patterns, see http://msdn.microsoft.com/en-us/library/97x6twsz.aspx.
[11] For more information about edit frames, see the Client Configuration Reference at http://sdn.sitecore.net/Reference/Sitecore%206/Client%20Configuration%20Cookbook.aspx.

The <sc:html> XSL extension control requires the following attribute:

- **field**: The name of the Rich Text field. For more information about the field attribute, see the preceding section, *The field Attribute*.

The <sc:html> XSL extension control accepts the following optional attributes:

- **select**: The item containing the Rich Text field. For more information about the select attribute, see the preceding section, *The select Attribute*.
- **disable-web-editing**: Enables or disables inline editing. For more information about the disable-web-editing attribute, see the preceding section, *The disable-web-editing Attribute*.
- **show-title-when-blank**: Output the field title if the field value is blank. For more information about the show-title-when-blank attribute, see the preceding section, *The field Attribute*.

The sc:field() and sc:fld() XSL extension methods provide functionality equivalent to that of the <sc:html> XSL extension control. For more information about the sc:field() XSL extension method, see the section *The sc:field() XSL Extension Method*. For more information about the sc:fld() XSL extension method, see the section *The sc:fld() XSL Extension Method*.

```
<sc:html field="FieldName" select="$sc currentitem" />
<xsl:value-of select="sc:field('FieldName',$sc currentitem)"
  disable-output-escaping="yes" />
<sc:html field="FieldName" select="$sc currentitem" disable-web-editing="true" />
<xsl:value-of select="sc:fld('FieldName',$sc_currentitem)" />
```

The <sc:image> XSL Extension Control

The <sc:image> XSL extension control outputs an HTML image (<img>) element using the image referenced in an Image field.

The <sc:image> XSL extension control requires the following attribute:

- **field**: The name of the Image field. For more information about the field attribute, see the preceding section, *The field Attribute*.

The <sc:image> XSL extension control accepts the following optional attributes:

- **select**: The item containing the Image field. For more information about the select attribute, see the preceding section, *The select Attribute*.
- **w**: Width in pixels.
- **h**: Height in pixels.
- **mw**: Maximum width in pixels.
- **mh**: Maximum height in pixels.
- **la**: Language (defaults to context language).
- **vs**: Version (defaults to latest version).
- **db**: Database name (defaults to context database).
- **bc**: Background color (defaults to black).
- **as**: Allow stretch (defaults to false, set to 1 for true).
- **sc**: Scale by floating point number (.25 = 25%).
- **thn**:Thumbnail (set to 1 for true).

- **disable-web-editing**: Enables or disables inline editing. For more information about the disable-web-editing attribute, see the preceding section, *The disable-web-editing Attribute*.

Note
Attributes of the <sc:image> XSL extension control that affect height and width do not correspond to the HTML height and width attributes of the <img> element. Image manipulation such as resizing occurs on the server to minimize network traffic to transfer the image from the server to the client.

The <sc:image> XSL extension control passes unrecognized attributes to the <img> element it generates.

```
<sc:image field="FieldName" select="$sc_currentitem" border="1" thn="1"/>
```

The sc:field() XSL extension method provides functionality equivalent to the <sc:image> XSL extension control. For more information about the sc:field() XSL extension method, see the section *The sc:field() XSL Extension Method.*

```
<xsl:value-of select="sc:field('FieldName', $sc_currentitem)"
  disable-output-escaping="yes" />
```

You can use the sc:fld() XSL extension method to access individual properties of an Image field.

```
<img src="{sc:fld('FieldName', $sc currentitem, 'src')}"
  alt="{sc:fld('FieldName', $sc_currentitem, 'alt')}" />
```

The <sc:link> XSL Extension Control

The <sc:link> XSL extension control generates an HTML anchor (<a>) element.

The <sc:link> XSL extension control does not require any attributes.

The <sc:link> XSL extension control accepts the following optional attributes:

- **field**: The name of the Image field. For more information about the field attribute, see the preceding section, *The field Attribute*.
- **select**: The item containing the field. For more information about the select attribute, see the preceding section, *The select Attribute*.
- **text:** The text content that the user will click in the HTML <a> tag.
- **disable-web-editing**: Enables or disables inline editing. For more information about the disable-web-editing attribute, see the preceding section, *The disable-web-editing Attribute*.

By default, the <sc:link> XSL extension control generates a link to the item represented by the context element. To link to a specific item, specify that item using the select attribute. To link as specified in a field of type General Link in the context element, pass the name of the field to the control using the field attribute. To link as specified in a General Link field in a specific item, pass both the select and the field attributes.

You can specify the text of the link using the text attribute, or the text value of the <sc:link> element. If you specify both, the layout engine ignores the text attribute, even if the text value evaluates to an empty string.

```
<sc:link text="click here" />
<sc:link text="this is ignored">this is output</sc:link>
```

The <sc:link> XSL extension control passes unrecognized attributes to the <a> element that it generates.

```
<sc:link class="CSSClass" />
```

You can use the `sc:fld()` XSL extension method to access the individual properties of a General Link field. For more information about the `sc:fld()` XSL extension method, see the section *The sc:fld() XSL Extension Method*.

```
<xsl:if test="sc:fld('FieldName',$sc_currentitem,'linktype')='mailto'">
```

Note
In the Page Editor, using `sc:field()` as a parameter to `<sc:link>` XSL extension control may result in a link looking incorrect because of including markup. To prevent this, use `sc:fld()` instead:

```
<sc:link select="$home">
<xsl:value-of select="sc:fld('Title',$home)" />
</sc:link>
```

Also, you can pass the following attribute to `<sc:link>` to disable inline editing in the Page Editor:

```
disable-web-editing="true"
```

The <sc:memo> XSL Extension Control

The `<sc:memo>` XSL extension control outputs the value of a Multi-Line Text field.

The `<sc:memo>` XSL extension control requires the following attribute:

- **`field`**: The name of the Multi-Line Text field. For more information about the `field` attribute, see the preceding section, *The field Attribute*.

The `<sc:memo>` XSL extension control accepts the following optional attributes:

- **`select`**: The item containing the Multi-Line Text field. For more information about the `select` attribute, see the preceding section, *The select Attribute*.
- **`line-breaks`**: Replacement characters for line feeds in the Multi-Line Text field.
- **`disable-web-editing`**: Enables or disables inline editing. For more information about the `disable-web-editing` attribute, see the preceding section, *The disable-web-editing Attribute*.
- **`show-title-when-blank`**: Output the field title if the field value is blank. For more information about the `show-title-when-blank` attribute, see the preceding section, *The show-title-when-blank Attribute*.

The `sc:fld()` and `sc:field()` XSL extension methods provide functionality equivalent to that of the `<sc:memo>` XSL extension control. For more information about the `sc:field()` XSL extension method, see the section *The sc:field() XSL Extension Method*. For more information about the `sc:fld()` XSL extension method, see the section *The sc:fld() XSL Extension Method*.

```
<sc:memo field="FieldName" select="$sc_currentitem" />
<xsl:value-of select="sc:field('FieldName',$sc_currentitem)"
  disable-output-escaping="yes" />
<xsl:value-of select="sc:fld('FieldName',$sc_currentitem)" />
<sc:memo field="FieldName" select="$sc_currentitem" line-breaks="&lt;br /&gt;" />
```

The <sc:sec> XSL Extension Control

The `<sc:sec>` XSL extension control causes the XSL transformation engine to process the enclosed segment of code if the context user has the designated access right to an item.

The `<sc:sec>` XSL extension control accepts the following attributes:

- **`require`**: Access right code.
- **`select`**: The item for which to evaluate the access right.

The `require` attribute supports the following access rights:

- **`item:admin`**: Administer access right.
- **`item:create`**: Create access right: `item:create`.
- **`item:delete`**: Delete access right.
- **`item:read`**: Read access right.
- **`item:rename`**: Rename access right.
- **`item:write`**: Write access right.

For example:

```
        <sc:sec req="item:delete" select="$sc_currentitem">
          <sc:sec req="item:create" select="$sc_currentitem">
            <!--the context user has both delete and create access rights to the context item-
->
          </sc:sec>
        </sc:sec>
```

The `sc:HasRight()` XSL extension method provides functionality equivalent to that of the `<sc:sec>` XSL extension control.

```
        <xsl:if test="sc:HasRight("item:delete",$sc_currentitem)">
```

The <sc:text> XSL Extension Control

The `<sc:text>` XSL extension control outputs the value of a Single-Line Text field or other simple text field.

The `<sc:text>` XSL extension control requires the following attribute:

- **`field`**: The name of the field. For more information about the field attribute, see the preceding section, *The field Attribute*.

The `<sc:text>` XSL extension control accepts the following optional attributes:

- **`select`**: The item containing the field. For more information about the select attribute, see the preceding section, *The select Attribute*.
- **`disable-web-editing`**: Enables or disables inline editing. For more information about the `disable-web-editing` attribute, see the preceding section, *The disable-web-editing Attribute*.
- **`show-title-when-blank`**: Output the field title if the field value is blank. For more information about the show-title-when-blank attribute, see the preceding section, *The show-title-when-blank Attribute*.
- **`editormode`**: When editing an item that contains a field of type Word Document in the Page Editor, if the value of the `editormode` attribute is `inline`, then the editor appears embedded within the page rather than as a popup window.
- **`editorwidth`**: When the value of the `editormode` attribute is `inline`, the value of the `editorwidth` attribute specifies the width of the inline editor in pixels. For example, `770px`.
- **`editorheight:`** When the value of the `editormode` attribute is `inline`, the value of the `editorheight` attribute specifies the height of the inline editor in pixels. For example, `450px`.

The `sc:fld()` and `sc:field()` XSL extension methods provide functionality equivalent to that of the `<sc:text>` XSL extension control.

```
        <sc:text field="FieldName" select="$sc currentitem" />
        <xsl:value-of select="sc:fld('FieldName', $sc_currentitem)" />
```

```
        <xsl:value-of select="sc:field('FieldName', $sc_currentitem)"
          disable-output-escaping="yes" />
```

The <sc:disableSecurity> XSL Extension Control

The `<sc:disableSecurity>` XSL extension control causes the XSL transformation engine to disable security checks while evaluating the enclosed XSL code, causing that code to execute in the security context of an administrative user.

The `<sc:disableSecurity>` XSL extension control does not accept any attributes.

For more information about the `<sc:disableSecurity>` XSL extension control, see the following section, *The <sc:enableSecurity> XSL Extension Control*.

The <sc:enableSecurity> XSL Extension Control

The `<sc:enableSecurity>` XSL extension control causes the XSL transformation engine to apply security while evaluating the enclosed XSL code, causing that code to execute in the security context of the context user.

Because security applies by default, you do not need to use `<sc:enableSecurity>` except within `<sc:disableSecurity>`. For example:

```
        <!--the system enforces security while processing this segment of code-->
        <sc:disableSecurity>
          <!--the system ignores security while processing this segment of code-->
          <sc:enableSecurity>
            <!--the system enforces security while processing this segment of code-->
          </sc:enableSecurity>
          <!--the system ignores security while processing this segment of code-->
        </sc:disableSecurity>
        <!--the system enforces security while processing this segment of code-->
```

The `<sc:enableSecurity>` XSL extension control does not accept any attributes.

The `sc:EnterSecurityState()` and `sc:ExitSecurityState()` XSL extension methods provide functionality equivalent to that of the `<sc:disableSecurity>` and `<sc:enableSecurity>` XSL extension controls.

```
        <xsl:value-of select="sc:EnterSecurityState(false())" />
          <!--the system ignores security while processing this segment of code-->
        <xsl:value-of select="sc:ExitSecurityState()" />
```

The `<xsl:value-of>` elements call the XSL extension methods, but generate no output.

The <sc:wordStyle> XSL Extension Control

The `<sc:wordStyle>` XSL extension control generates an HTML `<style>` element containing the CSS styles associated with the content of a field of type Word Document.

The `<sc:wordStlye>` XSL extension control requires the following attribute:

- **`field`**: The name of the Word Document field. For more information about the field attribute, see the preceding section, *The field Attribute*.

The `<sc:wordStyle>` XSL extension control accepts the following optional attributes:

- **`select`**: The item containing the Word Document field. For more information about the select attribute, see the preceding section, *The select Attribute*.

For example, to embed the styles and content of the Word Document field named **WordField** in the context item:

```
        <sc:wordstyle field="WordField" select="$sc currentitem" />
        <sc:text field="WordField" select="$sc_currentitem" />
```

4.6 Sitecore XSL Extension Methods

This section describes some of the Sitecore XSL extension methods available to XSL renderings.

4.6.1 The sc Namespace: The Sitecore.Xml.Xsl.XslHelper Class

This section describes the most frequently used Sitecore XSL extension methods, which are in the `Sitecore.Xml.Xsl.XslHelper` class represented by the `sc` namespace.

Important
The `Sitecore.Xml.Xsl.XslHelper` class exposes a number of XSL extension methods not documented in this section. For more information about these methods, see the Sitecore API documentation.[12]

The sc:feedUrl() XSL Extension Method

The `sc:feedUrl()` XSL extension method returns the RSS URL of an item.[13] The first parameter is the RSS item. The second parameter indicates whether the RSS feed requires Sitecore authentication, in which case query string parameters in the URL contain encrypted user identification information.

For example, you can generate a link to the RSS URL of the item identified by the variable named `$item` with syntax such as the following:

```
<xsl:variable name="rssUrl" select="sc:feedUrl($item, false())" />
<xsl:if test="$rssUrl">
  <a href="{$rssUrl}">RSS</a>
</xsl:if>
```

The sc:field() XSL Extension Method

The `sc:field()` XSL extension method returns the value of a field and includes markup to support inline editing if the user is inline editing in the Page Editor.

```
<xsl:value-of select="sc:field('FieldName',$sc_currentitem)"
  disable-output-escaping="yes" />
```

You can pass parameters, including those used to resize images, using a third parameter. For example, to process an image field using parameters equivalent to the attributes supported by the `<sc:image>` XSL extension control:

```
<xsl:value-of select="sc:field('FieldName', $sc currentitem,
  'disable-web-editing=yes&amp;thn=1&amp;border=1')" disable-output-escaping="yes" />
```

The sc:fld() XSL Extension Method

The `sc:fld()` XSL extension returns the raw value of a field, or the value of an attribute within an XML field value. The following example creates a variable using the value of a field in the context item:

```
<xsl:variable name="VariableName" select="sc:fld('FieldName',$sc_currentitem)" />
```

A common use of `sc:fld()` is to determine if a Checkbox field is selected. A Checkbox field stores the value `1` if the user selects the checkbox. Always check for this value to determine whether the user has selected the checkbox.

[12] For access to the Sitecore API documentation, see http://sdn5.sitecore.net/Reference/Sitecore%206.aspx.
[13] For more information about RSS, see the Client Configuration Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206/Client%20Configuration%20Cookbook.aspx and the Content Author's Cookbook at http://sdn.sitecore.net/End%20User/Sitecore%206%20Cookbooks.aspx.

```
<xsl:if test="sc:fld("FieldName",$sc_currentitem)!='1')">
  <!--checkbox field does not exist in context item or the user has not selected it-->
</xsl:if>
```

Certain types of fields, including Image, File, and General Link, represent their value using a single XML element with a number of attributes. You can pass a third parameter to `sc:fld()` to retrieve the value of a specific attribute. For example, to generate a link based on field of type File, you could use the `sc:fld()` method to retrieve the `src` attribute from the field value:

```
<xsl:variable name="src" select="sc:fld('FieldName',$sc_currentitem,'src')" />
<xsl:if test="$src">
  <a href="{concat('/',$src)}">
    <xsl:value-of select="concat('/',$src)" />
  </a>
</xsl:if>
```

To access the media item referenced by a File field, use the `sc:item()` method to retrieve the item referenced by the `mediaid` attribute.

```
<xsl:variable name="mediaid" select="sc:fld('FieldName',$sc_currentitem,'mediaid')" />
<xsl:if test="$mediaid">
  <xsl:variable name="mediaitem" select="sc:item($mediaid,$sc currentitem)" />
  <xsl:if test="$mediaitem">
    <a href="{concat('/',sc:GetMediaUrl($mediaitem))}">
      <xsl:choose>
        <xsl:when test="sc:fld('title',$mediaitem)">
          <sc:text field="title" select="$mediaitem" />
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="$mediaitem/@name" />
        </xsl:otherwise>
      </xsl:choose>
    </a>
  </xsl:if>
</xsl:if>
```

Important
The `sc:fld()` XSL extension method does not rewrite links to use friendly URLs, for example when retrieving the value of a Rich Text field. To rewrite links, use the `sc:field()` XSL extension method as described in the following section, *The sc:item() XSL Extension Method*.

The sc:item() XSL Extension Method

The `sc:item()` XSL extension method returns the `<item>` element corresponding to the ID or short path specified by the first parameter.

```
<xsl:variable name="content" select="/item[@key='sitecore']/item[@key='content']" />
<xsl:variable name="content" select="sc:item('/sitecore/content',$sc_currentitem)"/>
<xsl:variable name="content"
  select="sc:item('{110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9}',.)" />
<xsl:value-of select="$content/@name" />
```

Important
You must pass an element in the XML document containing the referenced item as the second parameter to the `sc:item()` extension method.

You can determine if the referenced item exists and the context user has the `item:read` access right to it by checking the value returned by the `sc:item()` XSL extension method. For example:

```
<xsl:variable name="content" select="sc:item('/sitecore/content',.)" />
<xsl:if test="$content">
  <xsl:value-of select="$content/@name" />
</xsl:if>
```

The sc:path() XSL Extension Method

The `sc:path()` XSL extension method returns the friendly URL of a content item.

```
<a href="{sc:path($sc_currentitem)}">
```

The sc:GetMediaUrl() XSL Extension Method

The `sc:GetMediaUrl()` XSL extension method returns the friendly URL of a media item. Replace the `FieldName` parameter in the sample code below with a valid field name for a field of Field Type Image or File.

```
<xsl:variable name="mediaid" select="sc:fld('FieldName',$sc_currentitem,'mediaid')" />
<xsl:if test="$mediaid">
  <xsl:variable name="mediaitem" select="sc:item($mediaid,$sc currentitem)" />
  <xsl:if test="$mediaitem">
    <a href="{concat('/',sc:GetMediaUrl($mediaitem))}">
    <xsl:value-of select="$mediaitem/@name" /></a>
  </xsl:if>
</xsl:if>
```

Important
Sitecore does not prefix the URL of the media item with a slash ("/") character automatically. This can result in URLs that exceed browser or server limits. Prefix media URLs with slashe characters when necessary.

The sc:pageMode() XSL Extension Method

The `sc:pageMode()` XSL extension method returns an XML structure indicating the client mode, such as Preview, Page Editor, or the Debugger, with and without different features enabled. You can use this information to output markup exposing different features in different modes. For more information about using the page mode, see the Client Configuration Cookbook.[14]

The sc:IsItemOfType() XSL Extension Method

The `sc:IsItemOfType()` XSL extension method returns true if an item is based on a data template that inherits from a specific base data template.

```
<xsl:if test="sc:IsItemOfType('basetemplate',$sc currentitem)">
<xsl:if test="$sc_currentitem/@template='basetemplate'
  or sc:IsItemOfType('basetemplate',$sc_currentitem)">
```

Important
The `sc:IsItemOfType()` method returns false for items that inherit directly from the specified base template. Both call the `sc:IsItemOfType()` XSL extension method and compare the template name directly when necessary:

```
<xsl:if test="sc:IsItemOfType('basetemplate',$sc_currentitem)
  or $sc_currentitem/@template='basetemplate'">
```

The sc:SplitFieldValue() XSL Extension Method

The `sc:SplitFieldValue()` XSL extension method returns an XML structure containing the IDs of the items selected in a selection field. You can use the `sc:SplitFieldValue()` XSL extension method to process the values in a Tree, Multilist, Treelist, or other field allowing selection of zero or more Sitecore items. For example:

```
<xsl:for-each select="sc:SplitFieldValue('FieldName',$sc_currentitem)">
  <xsl:for-each select="sc:item(text(),$sc currentitem)">
    <xsl:value-of select="@name" /><br />
  </xsl:for-each>
</xsl:for-each>
```

[14] To access the Client Configuration Cookbook, see http://sdn5.sitecore.net/Reference/Sitecore%206.aspx.

Note
The `sc:SplitFieldValue()` XSL extension method does not confirm the existence of items corresponding to the IDs contained in the field value.

The sc:formatdate() XSL Extension Method

The `sc:formatdate()` XSL extension method returns a formatted string based on a date value stored in the ISO format used by Sitecore. For more information about formatting dates, see the previous sections *The translate() Function* and *The <sc:date> XSL Extension Control*.

```
<xsl:value-of select="sc:formatdate(sc:fld('FieldName',$sc_currentitem),'d')" />
```

The sc:ToLower() XSL Extension Method

The `sc:ToLower()` XSL extension method returns the lowercase value of a string. The following condition is always true:

```
<xsl:if test="sc:ToLower($sc_currentitem/@name)=$currentitem/@key">
```

Important
XPath is case-sensitive. Always convert values to a consistent character case before comparison.

The sc:trace() XSL Extension Method

The `sc:trace()` XSL extension method writes a message to the trace log visible in the Sitecore debugger. For example:

```
<xsl:value-of select="sc:trace(concat('Context element item path: ',sc:path(.)))" />
```

In this case, the `<xsl:value-of>` element generate no output, but calls the `sc:trace()` XSL extension method to write a message to the trace.

The sc:qs() XSL Extension Method

The `sc:qs()` XSL extension method returns the value of a URL query string parameter.

```
<xsl:choose>
  <xsl:when
    test="sc:ToLower(sc:qs('ParameterName'))='true' or sc:qs('ParameterName')='1'">
    <!--URL query string parameter is true-->
  </xsl:when>
  <xsl:otherwise>
    <!--URL query string parameter is not true-->
  </xsl:otherwise>
</xsl:choose>
```

The sc:random() XSL Extension Method

The `sc:random()` XSL extension method returns a somewhat random integer as returned by `System.Random.Next(int)`.[15] For example, to generate a number between one and ten:

```
<xsl:variable name="VariableName" select="sc:random(11)" />
```

4.6.2 Additional XSL Extension Method Classes

This section describes several additional classes that you can use as XSL extension method libraries.[16]

[15] For information about `System.Random.Next(int)`, see http://msdn.microsoft.com/en-us/library/system.random.next.aspx.
[16] For more information about the Sitecore Application Programmer Interfaces, see http://sdn5.sitecore.net/Reference/Sitecore%206.aspx.

Excluding content marker functionality, the only class containing XSL extension methods provided by the default boilerplate file used to create XSL renderings is the `Sitecore.Xml.Xsl.XslHelper` class represented by the `sc` namespace. This class contains the most frequently used XSL extension methods.

To use additional extension libraries, update the XSL rendering header as shown for each additional extension library.

The dateutil Namespace : Sitecore.DateUtil

You can use some of the methods in the `Sitecore.DateUtil` class as XSL extension methods.

```
xmlns:dateutil="http://www.sitecore.net/dateutil"
exclude-result-prefixes="dot sc dateutil"
```

Note
The `sc` namespace contains the most frequently used methods for manipulating dates.

The stringutil Namespace : Sitecore.StringUtil

While there are some methods for manipulating strings in `Sitecore.Xml.Xsl.XslHelper`, there are additional helpful methods for manipulating strings in the `Sitecore.StringUtil` class that you can use as XSL extension methods.

```
xmlns:stringutil="http://www.sitecore.net/stringutil"
exclude-result-prefixes="dot sc stringutil"
```

The mainutil Namespace : Sitecore.MainUtil

There are some miscellaneous methods in the `Sitecore.MainUtil` class that you can use as XSL extension methods.

```
xmlns: mainutil ="http://www.sitecore.net/mainutil"
exclude-result-prefixes="dot sc mainutil"
```

The sql Namespace : Sitecore.Xml.Xsl.SqlHelper

There are some helpful methods for working with SQL databases in the `Sitecore.Xml.Xsl.Sqlhelper` class that you can use as XSL extension methods.

```
xmlns:sql="http://www.sitecore.net/sql"
exclude-result-prefixes="dot sc sql"
```

Chapter 5

Custom XSL Extension Libraries

This chapter provides techniques for implementing custom .NET XSL extensions.

You can use custom XSL extensions:

- To perform resource-intensive operations.
- To increase readability of the code used for complex operations
- To access data in a system other than the current Sitecore database.

This chapter contains the following section:

- Custom XSL Extension Methods

5.1 Custom XSL Extension Methods

This section provides procedures for implementing custom XSL extension methods using .NET. You can register your own XSL namespace containing custom XSL extension methods, or add methods to the default `sc` namespace.

Tip
Consider adding the namespace definition to the boilerplate file used for XSL renderings.

5.1.1 How to Add Methods to the sc Namespace

To add methods to the `sc` namespace, override `Sitecore.Xml.Xsl.XslHelper`.

1. Create a class that inherits from `Sitecore.Xml.Xsl.XslHelper`.
2. In `web.config`, in the `/configuration/sitecore/xslExtensions/extension` element with `namespace http://www.sitecore.net/sc`, replace the value of the `type` attribute with the signature of your class. The `sc` namespace then exposes the methods in your class as well as the methods in the `Sitecore.Xml.Xsl.XslHelper` base class.

```
<extension mode="on" type="Namespace.Class,Assembly"
  namespace="http://www.sitecore.net/sc" singleInstance="true" />
```

5.1.2 How to Access Properties of an XSL Extension Method Library Object

To access properties of an XSL extension class library object, use explicit `get_Property()` and `set_Property()` methods. For example:

```
<xsl:if test="get_PropertyName()">
  <xsl:value-of select="set_PropertyName('PropertyValue')" />
</xsl:if>
```

In this case, the `<xsl:value-of>` XSL element sets the property, and does not generate any output.

5.1.3 XSL Extension Method Examples

This section contains examples of custom .NET XSL extension methods.

GetHome() — Return a Sitecore.Data.Items.Item

The boilerplate file for XSL renderings defines a variable named `$home` using an XPath statement. This variable is invalid if you use an XSL rendering on a site that does not have `/Sitecore/Content/Home` as its start item. You can use an XSL extension method to determine the home item using logic rather than hard-coding a path.

First determine the home item for the site. Then use the `Sitecore.Configuration.Factory.GetItemNavigator()` method to convert the `Sitecore.Data.Items.Item` to the `System.Xml.XPath.XPathNodeIterator` representation used by XSL renderings.

```
namespace Namespace.Xml.Xsl
{
  private Sitecore.Data.Items.Item GetHomeItem()
  {
    Sitecore.Data.Database db = Sitecore.Context.Database;
    Sitecore.Data.Items.Item home = database.GetItem(Sitecore.Context.Site.StartPath);
    return(home);
  }
  public class XslHelper
  {
    public Sitecore.Xml.XPath.ItemNavigator GetHome()
```

```
        {
          return(Sitecore.Configuration.Factory.CreateItemNavigator(GetHomeItem()));
        }
        public string GetHomeID()
        {
          return(GetHomeItem().ID.ToString());
        }
      }
    }
```

Update the `$home` variable definition in XSL rendering and the boilerplate file used for new XSL renderings.

```
    <xsl:variable name="home" select="namespace:GetHome()" />
```

Note
It is generally more efficient to process a string than it is to process an XML structure. When possible, use a method that returns an ID as a `string` instead of returning an item as a `System.Xml.XPath.XPathNavigator`. For example, unless you are already using the `$home` variable and just need to update the logic used to define that variable, avoid defining the `$home` variable. When possible, use the `GetHomeID()` method instead of `GetHome()`. If you update the XSL rendering boilerplate file as suggested above, comment out this variable declaration to avoid unnecessary overhead. Developers can uncomment this line if they need this variable.

GetRandomSiblings() — Return Multuple Values Using XML

You can return a list from an XSL extension using a delimited string, or using XML. You can use this technique to return a list of item IDs, which you can process using XSL code similar to that used with the `sc:SplitFieldValue()` XSL extension method.

For example, a rendering needs to generate links to five random siblings of the context item, but never to the context item itself, and without ever generating two links to the same sibling. The following extension library class inherits from the `Sitecore.Xml.Xsl.XslHelper` class in order to use its `GetItem()` method to retrieve the `Sitecore.Data.Items.Item` corresponding to a `System.Xml.XPath.XPathNodeIterator`.

```
    namespace Namespace.Xml.Xsl
    {
      public class XslHelper : Sitecore.Xml.Xsl.XslHelper
      {
        public XPathNodeIterator GetRandomSiblings(XPathNodeIterator iterator,int max)
        {
          Sitecore.Xml.Packet packet = new Sitecore.Xml.Packet("values","");
          iterator.MoveNext();
          Sitecore.Data.Items.Item item = GetItem(iterator);
          if(item != null )
          {
            Sitecore.Collections.ChildList children = item.Parent.Children;
            if(children.Count>1)
            {
              if(max>children.Count-1)
              {
                max = children.Count-1;
              }
              List<Sitecore.Data.ID> ids = new List<Sitecore.Data.ID>();
              Random rand = new Random();
              while(ids.Count<max)
              {
                int index = rand.Next(children.Count);
                if(children[index].ID!=item.ID && !ids.Contains(children[index].ID))
                {
                  packet.AddElement("value",children[index].ID.ToString());
                  ids.Add(children[index].ID);
                }
              }
            }
          }
          XPathNavigator navigator = packet.XmlDocument.CreateNavigator();
          if (navigator == null)
```

```
            {
              navigator = new XmlDocument().CreateNavigator();
            }
            navigator.MoveToRoot();
            navigator.MoveToFirstChild();
            return navigator.SelectChildren(XPathNodeType.Element);
          }
        }
      }
```

This code will return an XML structure such as the following:

```
      <values>
        <value>{ID}</value>
        ...
        <value>{ID}</value>
      </values>
```

You can process this structure using code such as the following:

```
      <xsl:for-each select="namespace:GetRandomSiblings(.,5)">
        <xsl:for-each select="sc:item(text(),$sc_currentitem)">
          <sc:link>
            <xsl:value-of select="@name" />
            <br />
          </sc:link>
        </xsl:for-each>
      </xsl:for-each>
```

Note
This code is provided only for demonstration purposes and could be very inefficient with a small number of siblings.