



The Silverlight technique — or Ode to Steve Muench

Shedding a Little Light on XML

Wouldn't it be great to have something named after you? Up to now I've had no luck. Off the top of my head, the only way for this to happen is to (1) have children; (2) invent a lifesaving technique like the Heimlich maneuver; (3) die of a painful disease (Lou Gehrig's disease, for example);...



**SILVERLIGHT
HOTEL & RESORTS**

... (4) name a star after yourself (the "XML Star" was still available the last time I checked); (5) invent a clever method of grouping XML data (aka the Muenchian Method).

Hopefully I will never qualify for option 3. Steve Muench, however, has managed to gain immortality by developing a very innovative way to group XML data.

Grouping XML data is a very common programming dilemma, which will be resolved in this month's column using his technique, that is, the Muenchian Method. The Muenchian Method is not inherently obvious, primarily because it uses two functions, `key()` and `generate-id()`, neither of which is widely used. However, once they're demonstrated as part of a solution, their usefulness will stand out.

Black Belt What Is the Muenchian Method?

The Muenchian Method is an innovative technique used for grouping XML data using the XSLT `key()` function. The technique offers a clever solution to the questions "How do I group unsorted XML data efficiently?" and "How do I find the unique values in a set of values?"

While the method is used for grouping data, it's also used to find the unique values in a set of values. It has widespread applications beyond grouping. For example, you might want to count the number of states represented by a list of salesmen's territories. Though it may not sound like a grouping problem (in

fact, it *isn't* a grouping problem), you'd still use the key tables in the Muenchian Method to obtain the answer.

To further explain this technique, I'll use a sample XML document containing unsorted product information. The output generated will produce a list of products grouped by region, then by product name. Both groupings will be sorted.

Overview: A 50,000-Ft View of the Algorithm

Two loops will be used to process the product information.

The outer loop will iterate through all unique "regions" in the XML document. Each iteration of the loop will display the region name, sorted alphabetically, as the header for the HTML table that's generated. At this point the `key()` and `generate-id()` functions will be integral in generating a distinct set of regions.

The inner loop will iterate through all products for the current region and will display product information alphabetically. Here, the `key()` function will be integral in allowing us to quickly access the products for a given region.

Breakdown

Step 1: Define the primary key to be used in the Muenchian grouping.

Defining the primary key is the first step in Muenchian grouping. Using the `xsl:key` element sets up the groundwork for later accessibility to all "products" for a given "region" via a key table. At this point we're essentially defining how products for a region will be retrieved later on using the `key()` function, but no work is actually being done. All that's accomplished here is that the XSLT processor is learning how to populate

the key table so that later on in the process the `key()` function can be used to retrieve the elements.

The `xsl:key` element is defined as follows:

```
<xsl:key name="products" match="product" use="region" />
```

There are three parts to the `xsl:key` element:

1. **Name:** The name that will be used to refer to this key table.
2. **Match:** The pattern that refers to the nodes that populate the key table. In this example "product" nodes are those that can be accessed using this key.
3. **Use:** This "use" expression defines the value associated with the node used in the key table. In this example the "product" nodes will be accessed using the "region" as the key. This will allow us to retrieve all of the "products" for a specific "region."

Step 2: Loop through the unique regions (the primary key) in the document.

To fully appreciate what happens in Step 2, it's important to understand how the `generate-id()` function works. `generate-id(nodeset)` generates a unique identifier for the nodeset that's passed to it. It requires a nodeset as a parameter, but will return only the identifier for the first node (in document order) of the nodeset.

Note: It might appear on the surface that repeated calls to this function would produce a new `generate-id()` value each time they were called, thus causing the algorithm to fail. However, the same node will always return the same identifier during a single process. In a sense it behaves more like a "return-id" function than a "generate-id" function.

It's important to remember that values returned from `generate-id()` must be used blindly. There's no rhyme or reason

AUTHOR BIO

David Silverlight is chief XML evangelist for Infoteria Corporation (www.infoteria.com), an XML software development company based in Tokyo and Beverly, Massachusetts. He also maintains www.XMLPitstop.com, a resource for XML examples and everything else XML.

for the values generated other than that they be lexically valid as name tokens in XML. The processor is required to return the same ID for a given node every time it's requested in the running of a single process, but the processor isn't obliged to return the same value the next time around, even with the same data.

In some of the examples the values returned from the generate-id() function are displayed as part of the output. This is shown strictly to give a demonstration of what's happening under the covers. Please note that the generate-id() values should never be "interpreted" or used for analysis and/or comparison. You can't rely on the pattern generated since it will change from execution to execution, even if the data hasn't changed. The values are interesting to look at, however.

On to Step 2...

The end result of the xsl:for-each element below will be a return of product elements for the value of the region that's passed to it. More specifically, it's the result of examining all product elements and filtering out those whose "region" child values are duplicates, then filtering in the first element in document order with each unique value for a "region" child.

```
<xsl:for-each select="product
[generate-id(.)=generate-id
(key('products',region))]">
```

The logic behind this statement isn't inherently obvious, but a further breakdown should clarify it. Essentially, the expression will return the elements from which the IDs from the product elements "generate-id(.)" match the IDs returned from the indexed key lookup function "generate-id(key('products',region))".

The goal in this equality test is uniqueness: find "only one" of each value, then act on the found ones. The technique accomplishes this requirement by using the key table to find "the first of each in document order" of each value. This works because (1) the key() function returns the set of nodes of the given value (in document order), and (2) the generate-id() function works on the first node (in document order) when given more than one node.

By combining (1) and (2) above, the result is the first node of each key table value in document order, thus giving us the first node representing each unique value.

The actual IDs generated from the two sets are given in Figure 1, which shows the IDs that correspond to both. It's important to observe that each call

to generate-id(key('products',region)) will return a single value. However, when visually scanning the entire list of values displayed in Figure 1, it becomes apparent that there are only two unique IDs in the list (having values of "IDAN-HZDB" and "IDATGZDB"). On the other hand, the generate-id(.) expression will generate a unique ID for each product.

When visually scanning this column in Figure 1, we see that each element returns a unique value. When evaluating these two expressions in the equality test, only two elements are returned, producing only the unique "regions" from the XML document. Take a moment to look at the values in Listing 1 and Figure 1 to see what actual IDs are generated using these predicates. The downloadable source code for this article also contains this example and the code for Listing 2 (www.sys-con.com/xml).

Step 3: Display the products for each region.

As each region is iterated through, the key() function comes into play once again. This time it's used as part of the "for-each" loop to iterate through the product elements for the given region. Rather than looping through the document, the key function uses the current region value to return a nodeset of all the corresponding product elements from the named key table. Though it's up to the processor to implement the key() function quickly, it's assumed the processor will obtain values from the key table much faster than revisiting the source node tree.

```
<xsl:for-each select
="key('products',
region)">
```

The expression "key('products',region)" will return all of the "product" elements from the key table whose "use=" expression defined in xsl:key (see Step 1) evaluated to the same value as the "region"

child of the current element. In the example we specified use="region". If region has a value of "SouthWest", then all the product elements from the key table that contain a child element with a value of "SouthWest" will be returned.

Q: What are some of the cautions regarding the Muenchian Method?

A: The Muenchian Method requires an XML processor that supports keys, which not all parsers do. Although MSXML 3.0, Xalan, and Saxon support them, XT does not at the time of this writing. Alternative techniques that don't use the key() function do exist, but they're inherently slower based on the assumption that a processor will optimize the access to key tables (see **Elements of Design** section, below).

Keys can be memory intensive in their own way as a result of how they index document information. Using keys requires the information to remain in memory, thus demanding extra overhead for support.

The complete code for the Muenchian method example can be seen in Listing 2. Refer to Figure 2 to see the output generated.

generate-id()	Product Name	generate-id(key('products',region))	Region
IDAN17DB	Rock n Roll Supercart	IDAN17DB	SouthWest
IDATG7DB	Digital Tire Gauge	IDATG7DB	NorthWest
IDAN17DB	Auto Pilot Road Navigator	IDAN17DB	SouthWest
IDW1E7DB	SteadyFinger GPS for Laptop	IDAN17DB	SouthWest
IDW1E7DB	Auto Map Light	IDATG7DB	NorthWest
IDW1E7DB	PowerTip Signal Processor	IDATG7DB	NorthWest
IDW1E7DB	Star Ocean Watch Pendant	IDATG7DB	NorthWest
IDW1E7DB	Promotional Credit Card Lighter	IDATG7DB	NorthWest
IDAN17DB	Mentolite Set	IDATG7DB	NorthWest
IDAN17DB	Beckhammen Floor game set	IDATG7DB	NorthWest
IDW1E7DB	Blendable Sunglasses	IDAN17DB	SouthWest
IDW1E7DB	Sports Wrist Watch	IDAN17DB	SouthWest
IDW1E7DB	Eye Folding Leather Shopper	IDAN17DB	SouthWest
IDW1E7DB	Machino Slot Machine	IDAN17DB	SouthWest
IDW1E7DB	Swimming Cross Machine	IDAN17DB	SouthWest
IDW1E7DB	World Time Calculator	IDAN17DB	SouthWest
IDW1E7DB	Screen-Top Monitor Shelf	IDATG7DB	NorthWest
IDW1E7DB	Women's Bracelet Watch	IDAN17DB	SouthWest
IDW1E7DB	Teletube Portable DVD Player	IDATG7DB	NorthWest

FIGURE 1 Display of keys associated with product elements

Product Name	Price	Region
Auto Map Light	\$18	NorthWest
Beckhammen Floor game set	\$55	NorthWest
Digital Tire Gauge	\$40	NorthWest
Mentolite Set	\$30	NorthWest
Promotional Credit Card Lighter	\$20	NorthWest
PowerTip Signal Processor	\$15	NorthWest
Screen-Top Monitor Shelf	\$50	NorthWest
Star Ocean Watch Pendant	\$30	NorthWest
Teletube Portable DVD Player	\$1,299	NorthWest

Product Name	Price	Region
Auto Pilot Road Navigator	\$19	SouthWest
Blendable Sunglasses	\$30	SouthWest
Eye Folding Leather Shopper	\$39	SouthWest
Machino Slot Machine	\$69	SouthWest
Rock n Roll Supercart	\$150	SouthWest
Sports Wrist Watch	\$60	SouthWest
Swimming Cross Machine	\$100	SouthWest
World Time Calculator	\$30	SouthWest

FIGURE 2 Final output of products grouped by region, then by name

Elements of Design

A World Without Muench

This month's Elements of Design highlights other methods of grouping. The two methods showcased here were submitted by developers in the field. Each demonstrates a variation on the original message, has its own pros and cons, and simply offers alternative techniques for grouping XML data.

Grouping Without the Use of the key() function (G. Ken Holman)

As mentioned in the question above, not all XSLT processors support the key() function. This solution becomes especially useful in scenarios in which the key() function isn't an option. Following are a few comments explaining the differences between Ken's method and the Muenchian Method, but to fully appreciate it, download the source code for the full example.

The essence of using the variable instead is very similar to a key table. In this

solution all products are put into the variable (which itself is in document order):

```
<xsl:variable name="products" select=
  "//product"/>
```

Visit every product (which essentially happens when using a predicate in the Muenchian Method) in sorted order, but act only on the first product in document order (again using generate-id()), but this time with the variable instead of the lookup table):

```
<xsl:for-each select="$products">
  <xsl:sort select="name" order=
    "ascending"/>
  <xsl:variable name="region" select=
    "region"/>

  <xsl:if test="generate-id(.)=generate-
    id($products[region=$region])">
```

The rest is similar, except to grab the products of the same region from the

variable instead of the key table:

```
<xsl:for-each select=
  "$products[region=$region]">
```

Grouping Using the Distinct Template (EXSLT.ORG)

This solution varies from both Ken's approach and the Muenchian Method because it doesn't use key() or generate-id(). Instead, it uses the set:distinct template defined by the community initiative for commonly required extensions not available in XSLT.

This particular template can be used to return the distinct "region" elements instead of implementing the generate-id() function to return "product" elements with distinct "region" children. The focus changes slightly since we're obliged to deal with nodes being distinguished. ☹

DSILVERLIGHT @ INFOTERIA.COM

LISTING 1 XSLT code used to generate output in Figure 1

```
<h2>Listing of Keys associated with Product elements </h2>
<table border="1">
  <tr>
    <th>generate-id(.)</th>
    <th>Product Name</th>
    <th>generate-id(key('products',region))</th>
    <th>Region</th>
  </tr>
  <xsl:for-each select="product">
    <tr>
      <td><xsl:value-of select="generate-id(.)"/></td>
      <td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="generate-
        id(key('products',region))"/></td>
      <td><xsl:value-of select="region"/></td>
    </tr>
  </xsl:for-each>
</table>
```

LISTING 2 Source code for the solution using the Muenchian Method of grouping

```
<?xml version='1.0' encoding='utf-8' ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" ver-
  sion="1.0">
  <xsl:output method="html"/>
  <xsl:include href="Style1.xsl" />

  <!--
  Step 1: Define the primary key to be used in the Muenchian
  grouping. The beautiful thing about the xsl:key element in
  our example is that once we know the "region," we can easily
  find all of the products that match that region.
  The xsl:key element (different from the key() function) is
  defined as follows:-->

  <xsl:key name="products" match="product" use="region"/>

  <!-- Template for our root rule -->
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <!-- Template for our "products" rule -->
  <xsl:template match="products">

  <xsl:call-template name="Style"/>
```

```
<h2>Grouping of Products by Region, then by Product Name
</h2>
```

```
<!-- Step 2: Loop through the unique regions (the primary
key) in our document. -->
<xsl:for-each select="//product[generate-id(.)=generate-
id(key('products',region))]">
```

```
<!-- Sort Primary key by name in ascending order -->
  <xsl:sort select="name" order="ascending"/>
```

```
<!-- Display the region as our table header -->
  <h3><xsl:value-of select="region"/> region</h3>
```

```
<!-- Display all nodes for a given region in a table-->
```

```
<table border="1">
  <tr>
    <th>Product Name</th>
    <th>Price</th>
    <th>Region</th>
  </tr>
```

```
<!-- For each value in our key collection for the given
region,
display values -->
```

```
<xsl:for-each select="key('products',region)">
```

```
<!--
The expression "key('products',region)" will return all of
the "product" elements from the key table whose "use="
expression defined in xsl:key (see xsl:key at top) evaluat-
ed to the same value as the "region" child of the current
element. In the example we specified use="region". If
region has a value of "SouthWest", then all of the product
elements from the key table that contain a child element
with a value of "SouthWest" will be returned.
-->
```

```
<!-- Sort our secondary key, product nodes, by name-->
  <xsl:sort select="name"/>
```

```
<tr>
  <td><xsl:value-of select="name"/></td>
  <td><xsl:value-of select="price"/></td>
  <td><xsl:value-of select="region"/></td>
</tr>
```

```
</xsl:for-each>
</table>
<br/><br/>
</xsl:for-each>
```

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

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

DOWNLOAD THE CODE @
www.xml-journal.com