Extracted from:

XSLT Jumpstarter

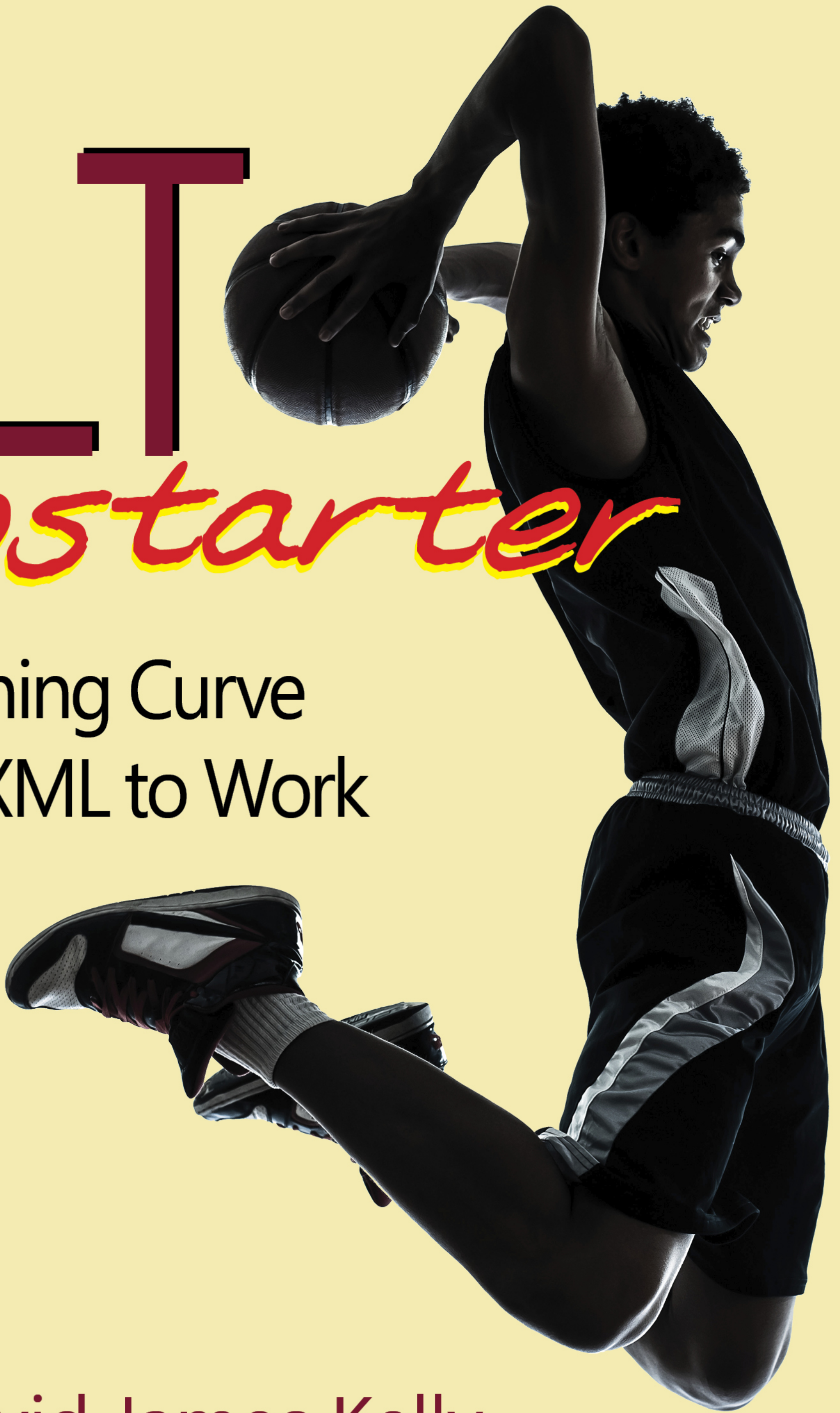
Level the Learning Curve and Put Your XML to Work

Peloria Press

Raleigh, North Carolina

XSLT Jumpstarter

Level the Learning Curve and Put Your XML to Work



David James Kelly

Foreword by Dave Thomas

Output Equals Input: the Simplest Identity Transform

The simplest version of the identity transform is a single template that makes an identical copy of everything in the source XML:

to-identity/identity.xsl

```
Line 1 <?xml version="1.0" encoding="utf-8"?>
     2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
     3   <xsl:template match="@* | node()">
     4     <xsl:copy>
     5       <xsl:apply-templates select="@* | node()"/>
     6     </xsl:copy>
     7   </xsl:template>
     8 </xsl:stylesheet>
```

Figure 13—Stylesheet for the Identity Transform

That's it. You can run a document of a million XML nodes through this stylesheet, and the output will be the same as the input.

So what's going on here? How does a single template perform such a magical trick? We saw some of the stylesheet structures in Chapter 2, *XSLT in Action*, on page ?, but there are some new things, too.

First, the `match=` attribute in line 3 includes *@** and *node()*, separated by a pipe (|) character. The @ at the beginning of a string signifies an attribute. For instance, you would refer to an `id=` attribute by saying *@id*. In this case we add the asterisk (*) wildcard to get it to refer to *all* attributes.

The *node()* is a kind of wildcard itself: it tells the `match=` attribute to match any and all XML nodes in the current context. "All nodes" means text, elements, comments, and processing instructions. Pretty much everything except attributes.

The | character represents a union of node sets, although it is sometimes thought of as a logical OR. It is used between XPath statements that return node sets, and it acts to combine those node sets. This means that the `match=` attribute in line 3 says, essentially, "Match all nodes or attributes found in the current context."

There is also an `or` operator that is used in expressions, but that one is used as part of an expression to return a Boolean value.

A Walk through the Identity Transform's Operations.

To follow what happens when this stylesheet meets an XML document, let's take a look at a new bit of XML:

to-identity/bibliography.xml
```
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet type="text/xsl" href="bib1.xsl"?>
<bibliography id="i54321">
  <book>
    <author>
      <given>David</given>
      <family>Kelly</family>
    </author>
    <title>XSLT Jumpstarter: Level the Learning Curve
      and Put Your XML to Work</title>
  </book>
</bibliography>
```

The XSLT processor looks in the stylesheet for matches starting from the XML document's root node. In our example, it matches the root node in the XML because our template matches any and all nodes in the current context with that node() wildcard.

Great, we're in. But what happens next?

Line 4 contains a single XSLT instruction: <xsl:copy>. This instruction tells the XSLT processor to make a copy of the current node, and that's it. So we copy the root node into the output and keep going.

Next, line 5 tells the XSLT processor to apply any templates, within the current context, that match any XML elements defined by the select= attribute. In this case, the contents of the select= attribute look exactly like the contents of our original match= attribute. The <xsl:apply-templates> instruction is telling the XSLT processor, "Go find any templates that provide a match for attributes or nodes in the current context," and the match is provided by the match statement at the head of the same template.

Since the <xsl:apply-templates> tag is inside the <xsl:copy> instruction, the results of applying any subsequent templates will also be inside the node that we just copied. That's as it should be for our example, but make a note: by moving the <xsl:apply-templates> tag outside the <xsl:copy> tag, we could change the structure of the output document—and there may be times when you need that.

So far, the stylesheet has processed the root node, <bibliography> and created a copy of it. Note that the XSLT processor has only output the opening tag at this point. It won't create the closing tag for <bibliography> until it reaches the closing tag of <xsl:copy> in the template.

Then, when the processor reaches <xsl:apply-templates>, the select= attribute tells it to apply any template that matches any XML attribute or node in the current context. In the XML, there is one attribute, id="i54321". So the XSL processor looks for a template to match that attribute.

Well, since there's only one template in the stylesheet, it doesn't have to look far. And since the template matches any attribute or node, it matches our particular id= attribute. So the template starts over again: the <xsl:copy> instruction copies the attribute and its value, the <xsl:apply-templates> tag keeps up moving through the XML..

This time there are no more attributes to match, so the <xsl:apply-templates> instruction matches on the next item in the XML document, in document order. The next thing is the <book> tag, so that gets matched and copied. Then the <author> tag. Then the <given> tag, and then the "David" piece of text.

There's no more text to copy, nor any other XML elements inside <given>, so the XSL process begins to unwind—remember, we've been executing the <xsl:apply-templates> tag and haven't reached the closing </xsl:copy> tag yet. The processor reaches the end tag for <given>, so the end tag for <xsl:copy> kicks in and outputs </given>. The processor moves on to the next tag (<family>), the processor finds the match for it with the node() part of match="@* | node()" in line 3 of Figure 13, *Stylesheet for the Identity Transform*, on page 3, and the process continues.

You get the pattern. This goes on and on until the end of the document is reached. The template for the identity transform uses a pattern in which it calls itself to continue processing, and it continues processing until it runs out of XML nodes and attributes. It's just a quiet, efficient little copying machine.

The result is a fine, upstanding replica of the XML file we started with.

Why Use an Identity Transform?

What was the point of all that? How useful can a copy be?

It turns out the identity transform has a lot of uses. Sometimes we want to make a copy of a document with just a slight variation—changing British spellings to American spellings, for instance. Sometimes we want to make a copy of only some parts of a document while discarding other parts. I once needed to show all the tags and structure of a document that was causing me problems, but the content was proprietary, so I couldn't share it. I used an identity transform with slight variation to translate all letters and numbers to Xs, and off it went.

To make the point, let's try a few variations of our own. In the following sections we're going to:

- Remove all tags from a document
- Remove all text from a document
- Uppercase a specific word in the document
- Change all instances of one word to another word

We'll find that the identity transform serves as a starting point for a lot of helpful purposes. And with each new function we demand from the identity transform, we'll also learn some new XSLT techniques.

Variation on a Theme #1: Removing All Tags

One variation on the identity transform occurs when we want to remove all the tags from the document. It sounds a bit drastic, but it does happen: you might want to run the text through a voice translator, for instance, or you may need to prepare the text for an audio transcription. You may simply want a list of text values in a file for parsing by some other software that works better with text than with XML.

In any case, stripping the tags out is a piece of cake. We'll just need to add one template to act as an exception to the template that copies everything.

Let's start with an XML file that will run nicely in a browser (see Figure 14, *XSLT Code for an Acknowledgements Web Page*, on page 7

Sure, it's XHTML, but that's XML, too, so it works.

Notice that the second line of this XML file points to identity.xsl. If you open this file in a browser (and the identity.xsl file is in the same directory as the XML file), the stylesheet in identity.xsl is invoked. This stylesheet contains the identity transform, so the output looks like Figure 15, *The Output from the Identity Transform in the Browser*, on page 7.

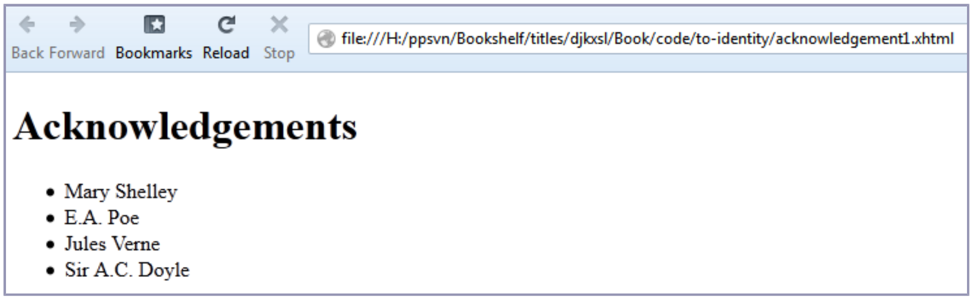
The XSL stylesheet transforms the XML into a perfect copy of itself, as we saw in the previous section. The resulting XHTML appears in the browser with all the text formatted as it should be.

Suppose we want to count the number of words in the text, or we need a version to give to someone who will read it for an audio book. All we want is the text. To get that, we'll make our transformation a copy operation, but with

to-identity/acknowledgement1.xhtml

```
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet type="text/xsl" href="identity.xsl"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Acknowledgements</title>
  </head>
  <body>
    <h1 class="acknowledgements" id="d24e12098">Acknowledgements</h1>
    <ul>
      <li>Mary Shelley</li>
      <li>E.A. Poe</li>
      <li>Jules Verne</li>
      <li>Sir A.C. Doyle</li>
    </ul>
  </body>
</html>
```

Figure 14—XSLT Code for an Acknowledgements Web Page



file:///H:/ppsvn/Bookshelf/titles/djkxsl/Book/code/to-identity/acknowledgement1.xhtml

Acknowledgements

- Mary Shelley
- E.A. Poe
- Jules Verne
- Sir A.C. Doyle

Figure 15—The Output from the Identity Transform in the Browser

a slight difference from the identity transform: it will drop all the tags during the copy process.

So let's modify our identity transform to get that working (see Figure 16, *A Stylesheet to Strip Tags from XML*, on page 8.

It's similar to the identity transform, but with a difference. Check out the differences between this piece of code and Figure 13, *Stylesheet for the Identity Transform*, on page 3. As you can see, the first template looks like the identity transform, but with a change to the match= and select= attributes. And then we've added a second template that matches on the asterisk (*), but doesn't make a copy of anything.

to-identity/acknowledgement2.xsl

```
Line 1  <?xml version="1.0" encoding="utf-8"?>
     -  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     -    version="1.0">
     -    <xsl:template match="/ | text()">
     5      <xsl:copy>
     -        <xsl:apply-templates select="* | text()"/>
     -      </xsl:copy>
     -    </xsl:template>
     -
    10     <xsl:template match="*">
     -        <xsl:apply-templates select="* | text()"/>
     -    </xsl:template>
     -
     -  </xsl:stylesheet>
```

Figure 16—A Stylesheet to Strip Tags from XML

(If you want to learn more about wildcards right this red-hot minute, you can skip ahead to *Wildcards and General Representations for Nodes*, on page ? —but come back straightly.)

We use * instead of node() because node() includes text nodes, and we want to distinguish between text nodes and element nodes. The * matches only elements, not text nodes. So we split up that node() from the previous example into * and text().

Let's go through it in detail to understand what we've accomplished, and how it works.

In line 4 of our tag-remover, we're now matching on the document root ("/") or any text node ("text()"). Before, we used *node()* to represent the root node, element nodes, and text nodes, because we wanted to treat all those kinds of nodes the same way—we just wanted to copy them. Now we want to treat the element nodes differently, so we discard *node()* from the match and we substitute / and *text()*. These are the two kinds of nodes we want to continue copying.

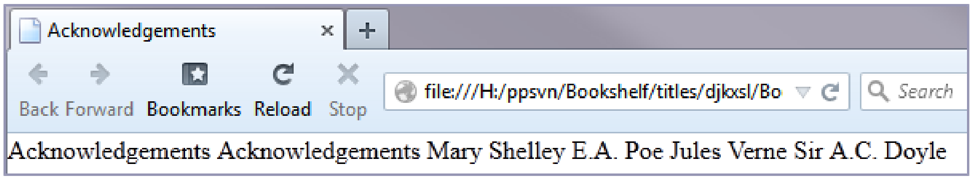
We've also discarded the @* from the match, because if we're not going to copy elements, we certainly don't need to copy attributes. And since there is nothing inside an attribute, we don't need to process them, either.

In line 6, our <xsl:apply-templates> tag has a new select= value. It includes *text()*, but it also includes *, the two of them separated by a pipe. As we mentioned, the * is a wildcard for XML elements. So now we're telling the XSL processor, "Apply any templates that match the element nodes or text nodes that you find within the current context."

In the last version, we had only one template. In this version, we have two templates. Now, when the XSLT processor finds an element in the document, it will find the new template in the stylesheet that matches elements (the one on line 10). And when it processes that template it won't have the <xsl:copy> instruction. Instead, it finds only the <xsl:apply-templates> on line 11.

The XSLT processor goes off and finds a template to match whatever it finds next—element or text node. If it's a text node, the XSLT processor finds a matching template on line 4. If it's an element, the match is with the template on line 10, which doesn't perform a copy—so the element is stripped. The templates continue churning through the XML document, as before, as long as the XSLT processor continues to find matches.

With the tags stripped out, you'll see this in the browser:



Acknowledgements Acknowledgements Mary Shelley E.A. Poe Jules Verne Sir A.C. Doyle

Success! None of that extraneous XHTML formatting for us.

Okay, that was fun. We did something interesting with the identity transform. And now there shouldn't be many surprises when we see the template that strips out text in the next section.