Introduction to XSLT

Version 1.3.2
March 2020

nikos dimitrakas

Table of contents

1 INTRODUCTION 3
1.1 XSLT 3
1.2 PREREQUISITES 3
1.3 STRUCTURE 3
2 SAMPLE DATA 4
3 XSLT 6
4 EXAMPLES 7
4.1 HELLO WORLD 7
4.1.1 XML 8
4.1.2 html 10
4.2 SIMPLE ITERATION 11
4.3 MULTIPLE TEMPLATES 13
4.4 ELIMINATING DUPLICATES 16
4.5 MULTIPLE SOURCES 18
4.6 CONDITIONALS 20
4.7 DEFAULT VALUES 21
4.8 ATTRIBUTE VALUE TEMPLATES 22
4.9 GROUPING (XSLT 2) 24
4.10 DYNAMIC XPATH EXPRESSIONS (XSLT 3) 26
5 EXECUTING XSLT 2 AND XSLT 3 32
5.1 XSLTRANSFORM.NET 32
5.2 BASEX 32
6 EPILOGUE 32

1 Introduction

This compendium gives an introduction to XSLT and how to use some of its most common features. Most of the XSLT features described in the examples later on are part of XSLT 1. Some additional features were introduced in XSLT 2 and XSLT 3, but since this compendium relies on products (Web browsers) that only support XSLT 1, only some XSLT 2 and XSLT 3 examples are covered in the last section of chapter 4. There are ways to test XSLT 2 and XSLT 3, but most require commercial products or libraries that require writing a client to use them. The best free option is the site xslttest.appspot.com that has created such a program that uses one of the available libraries. BaseX has also support for executing XSLT by using the Saxon XSLT Processor.

The latest version of this compendium is available at http://coursematerial.nikosdimitrakas.com/xslt/ where all other relevant files can also be found.

1.1 XSLT

XSLT (or XSL Transformations) is a language that can be used to transform XML data from one structure to another. The result may be XML, but it doesn't have to be. A popular output format is html.
XSLT 1 relies on XPath 1, while XSLT 2 relies on XPath 2 and XSLT 3 relies on XPath 3. Consequently, XSLT 2 and 3 are far more powerful and flexible. Unfortunately the major web browsers only support XSLT 1. Most of the examples in this compendium are in XSLT 1, so they can be tested in any web browser. A few examples use XSLT 2 and XSLT 3 features. They require an execution environment that supports XSLT 2/3, like xsltransform.net, xslttest.appspot.com or BaseX.

1.2 Prerequisites

It is highly recommended that the reader has a good understanding of XML as well as some knowledge of html and basic programming concepts like loops, conditionals, variables, etc. Most of the examples can be executed in any major web browser, so having a computer available while reading this compendium is recommended. Since XSLT relies heavily on XPath, prior familiarity with XPath and even XQuery is a plus.

1.3 Structure

In the next chapter we will look at the sample data used for all the examples. In chapter 3 we will take a look at different ways to connect XSLT files to XML files. Then in chapter 4, we will go through several examples using the sample data.

2 Sample Data

In this chapter we will take a look at the XML data that we will use in all the examples to follow. We will use one XML document with information about books and one XML document with information about publishers. The two XML documents are available in the files books.xml and publishers.xml and they are accompanied by books.dtd and publishers.dtd which describe the structure of the two documents. The two DTDs are presented below.

Books.dtd:

```
<!ELEMENT Books (Book+)>
<!ELEMENT Book (Author+,Edition+)>
<!ATTLIST Book Title CDATA #REQUIRED
    OriginalLanguage CDATA #REQUIRED
    Genre CDATA "N/A">
<!ELEMENT Edition (Translation*)>
<!ATTLIST Edition Year CDATA #REQUIRED
    Price CDATA #REQUIRED>
<!ELEMENT Translation EMPTY>
<!ATTLIST Translation Language CDATA #REQUIRED
    Publisher CDATA "N/A"
    Price CDATA #REQUIRED>
<!ELEMENT Author EMPTY>
<!ATTLIST Author Name CDATA #REQUIRED
    Email CDATA #REQUIRED
    YearOfBirth CDATA #REQUIRED
    Country CDATA #REQUIRED>
```

The definition in books.dtd can be graphically represented according to the following figure.

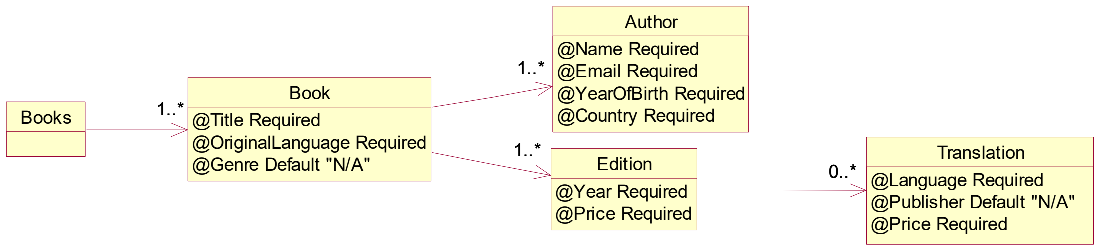


Figure 1 Graphical representation of books.dtd

Publishers.dtd:

```
<!ELEMENT Publishers (Publisher+)>
<!ELEMENT Publisher (Address)>
<!ATTLIST Publisher Name CDATA #REQUIRED>
<!ELEMENT Address (Street, City, PostalCode, Country)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT PostalCode (#PCDATA)>
<!ELEMENT Country (#PCDATA)>
```

The definition in publishers.dtd can be graphically represented according to the following figure.

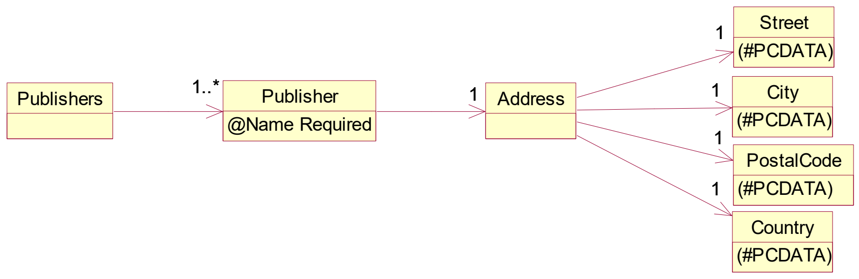


Figure 2 Graphical representation of publishers.dtd

DTDs can unfortunately not express certain details like data types, unique constraints and other complex business rules. For the remainder of this compendium we will consider the following rules to also be valid in the two XML documents:

- Every book has a unique title
- Every author has a unique name and a unique e-mail
- Two editions of the same book may not have the same year
- Two translations of the same edition may not have the same language
- Every publisher has a unique name
- The value of the attribute Publisher in the element Translation always corresponds to the name of a publisher
- The attributes Price, Year and YearOfBirth allow only positive integers as their value

3 XSLT

An XSLT file is usually a file with the extension .xsl. Such a file contains all the transformation rules to be applied to an XML file. One way to link the two files to each other is to insert a reference to the XSLT file in the XML file. This is done with the following processing instruction just before the root element of the XML document:

```
<?xml-stylesheet type="text/xsl" href="filename.xsl"?>
```

Where "filename.xsl" is of course the name of the XSLT file. This reference can be relative to the location of the XML file or an absolute filename reference.

This method would of course imply that you can only have one XSLT file associated with each XML file, which would be quite inflexible. The linking of the XSLT file could of course be done dynamically for example if your XML file is generated by a program.

Another possible solution is to let the XSLT file reference the XML file it wants to use. This can be done by using the XPath function "document". This gives us the possibility to also have an XSLT file that works with two or more sources. The disadvantage is, of course, that the XSLT files cannot be reused for different XML files. So choose your solution wisely!

For the examples in the next chapter, we will use any XSLT capable web browser as our execution environment. These browsers will apply the referenced XSLT file when an XML file (with such a reference) is opened. Should we try to open an XSLT file directly in a browser, then the XSLT file would be considered to be the opened XML file (since XSLT files are also XML files) and it would just be displayed. So in order to have the content of the XSLT file executed and not just displayed, we need to always open an XML file with a reference to the XSLT file. So we have the following two options (which correspond to the alternatives discussed earlier):

1. Add a reference to the XSLT file in the XML file that contains the data and open it in a browser.
2. Write an XSLT file that uses the "document" function in order to access the XML file with the data, and then use another XML file with just a reference to the XSLT file and an empty root element. Such a file could look like this:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <?xml-stylesheet type="text/xsl" href="filename.xsl"?>
   <root/>
   ```

If we instead want to use a different execution environment the XSLT file and the XML file may be linked dynamically by the execution environment. In BaseX for example, the XML document and the XSLT document are provided as parameters to the function xslt:transform (when the output should be XML) and or the function xslt:transform-text (when the output should be text). This is similar to how this works in other products like DB2, SQL Server and Oracle. xsltransform.net and xslttest.appspot.com use a similar principal where the two documents are provided independently in two form components.

4 Examples

In this chapter we will go through several examples in order to test some of the most common declarations and instructions that are part of XSLT. In most examples we will only use one XML file as the data source.

4.1 Hello World

Let's start with a very basic example just to check that everything works correctly. The goal is to show the text "Hello World" as the output if there is a root element. Our XSLT file (named hello.xsl) could have the following content:

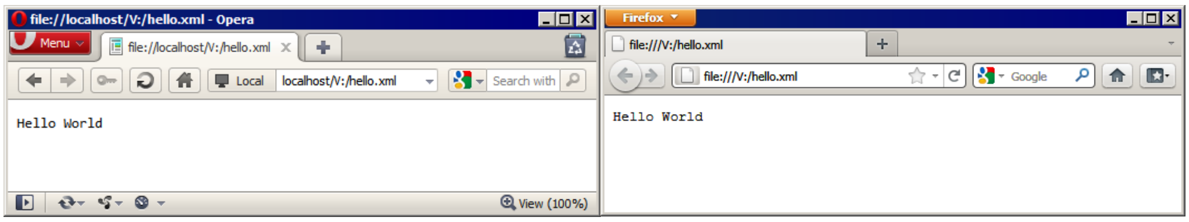
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text"/>
<xsl:template match="/">
    <xsl:text>Hello World</xsl:text>
</xsl:template>
</xsl:transform>
```

Then we need an XML file (let's call it hello.xml) with a reference to the XSLT file. The content of the XML file should be the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="hello.xsl"?>
<root/>
```

By opening the XML file in a web browser, the XSLT file will be called and the contained templates will be applied if they match. Our XSLT file has only one template and it matches the root element of the XML document. The content of the template element will thus start constructing the output. The element xsl:output is a declaration that informs the executing environment that the result will be of a certain type, in our case plain text. Other common output methods are xml and html. The element xsl:text is an instruction that creates text in the output.

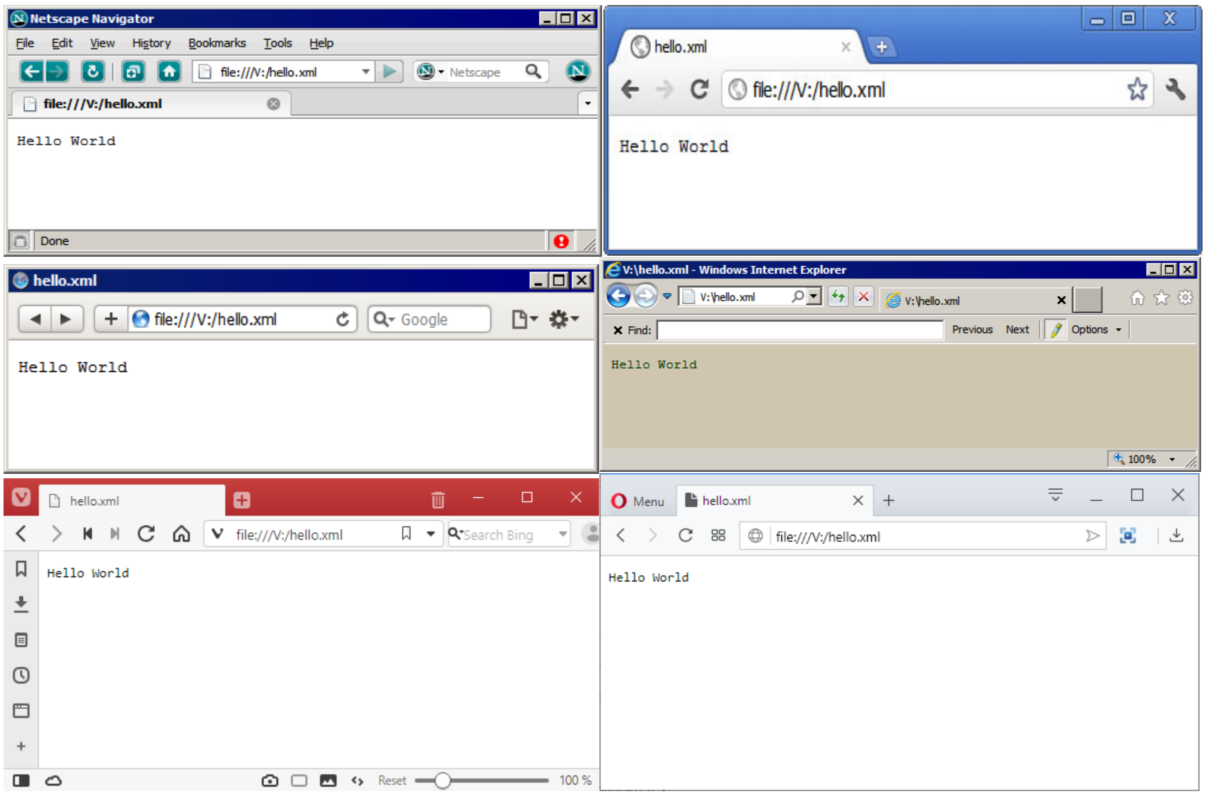
The result would look like the following screenshots from some major web browsers[1]:



```
Hello World
```

```
Hello World
```

[1] For Chrome and other browsers using chromium (like Opera since version 15 and Vivaldi) to work with local files, they must be started with the following command line parameter --allow-file-access-from-files. This is due to some security design decision by Chromium developers.


Hello World

Since all the elements in the XSLT file belong to the same namespace, it would be correct to rewrite the file's content as the following:

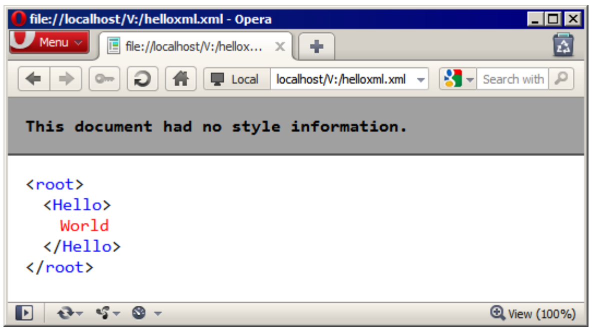
```
<?xml version="1.0" encoding="UTF-8"?>
<transform version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
<output method="text"/>
<template match="/">
    <text>Hello World</text>
</template>
</transform>
```

4.1.1 XML

We may of course want the output to be XML instead of plain text. In that case we can change the output method to xml and construct the desired XML structure inside the template element:

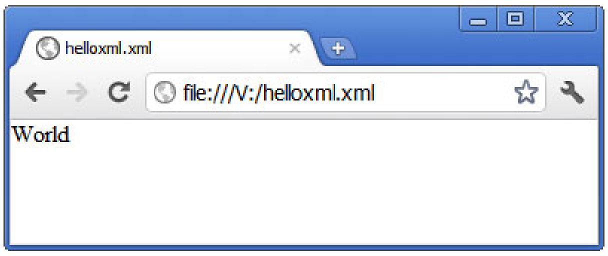
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" />
<xsl:template match="/">
    <root><Hello>World</Hello></root>
</xsl:template>
</xsl:transform>
```

The result may look different in each browser depending on the setting for displaying XML. Here are two examples:


file://localhost/V:/helloxml.xml - Opera
Menu
file://localhost/V:/hellox...
Local
localhost/V:/helloxml.xml
Search with
View (100%)

This document had no style information.

```
<root>
  <Hello>
    World
  </Hello>
</root>
```

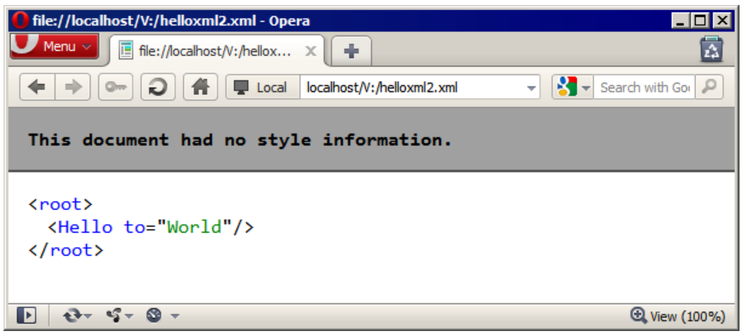


World

In the example above we created our elements directly, but we could instead use XSLT instructions for that. In the example below we create the elements "root" and "Hello" and we create an attribute "to" with "World" as its value:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" />
<xsl:template match="/">
   <xsl:element name="root">
     <xsl:element name="Hello">
        <xsl:attribute name="to">World</xsl:attribute>
     </xsl:element>
   </xsl:element>
</xsl:template>
</xsl:transform>
```

The result would be the following:


file://localhost/V:/helloxml2.xml - Opera
Menu
file://localhost/V:/hellox...
Local
localhost/V:/helloxml2.xml
Search with Go
View (100%)

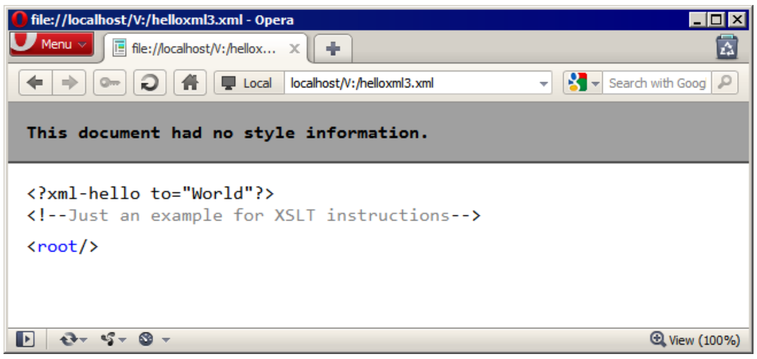
This document had no style information.

```
<root>
  <Hello to="World"/>
</root>
```

The main components in an XML document are elements and attributes, but we may also want to create comments or processing instructions. The XSLT instructions "comment" and "processing-instruction" can help us with that. Here is a simple example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" />
   <xsl:template match="/">
      <xsl:processing-instruction name="xml-hello">to="World"</xsl:processing-instruction>
      <xsl:comment>Just an example for XSLT instructions</xsl:comment>
      <xsl:element name="root">
      </xsl:element>
   </xsl:template>
</xsl:transform>
```

And the result would look like this:


file://localhost/V:/helloxml3.xml - Opera
Menu
file://localhost/V:/hellox...
Local
localhost/V:/helloxml3.xml
Search with Goog
View (100%)

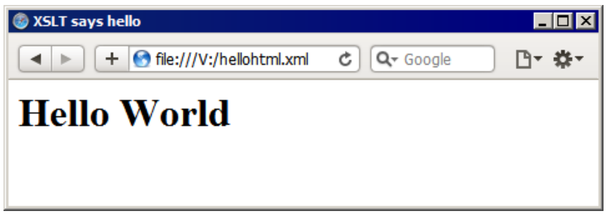
```
This document had no style information.

<?xml-hello to="World"?>
<!--Just an example for XSLT instructions-->
<root/>
```

4.1.2 html

Another popular output method is html. Since we are using web browsers as our execution environment and display facility, it is only reasonable to use html for the output. The web browsers will automatically render the result as html. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="html" />
   <xsl:template match="/">
      <html>
         <head>
            <title>XSLT says hello</title>
         </head>
         <body>
            <h1>Hello World</h1>
         </body>
      </html>
   </xsl:template>
</xsl:transform>
```

The result is a very simple html page that a browser could render like this:



Hello World

We could of course use the XSLT instruction "element" in order to create the elements "html", "head", "title", "body" and "h1" but the result would be the same.

4.2 Simple iteration

It's now time to use some XML input data in the output. We can start by creating an XSLT file (booktitles.xsl) with the following content:

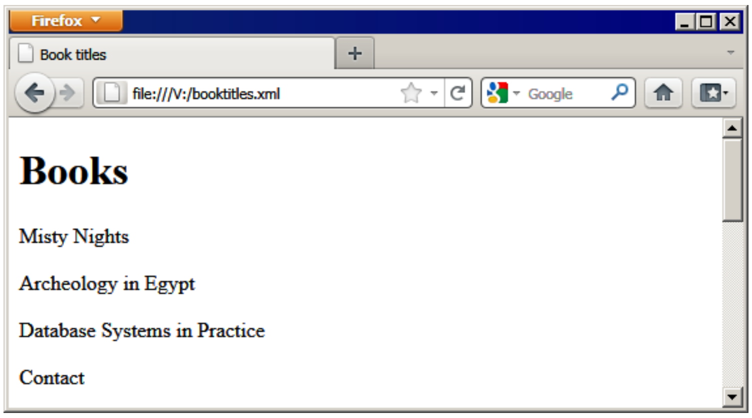
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <html>
         <head><title>Book titles</title></head>
         <body>
            <h1>Books</h1>
            <xsl:for-each select="document('books.xml')//Book">
               <p><xsl:value-of select="@Title"/></p>
            </xsl:for-each>
         </body>
      </html>
   </xsl:template>
</xsl:transform>
```

The template that matches the root of the calling document will create the html content of the output. The new thing here is the XSLT instruction "for-each", where we retrieve a sequence of Book elements from the XML file "books.xml". For each such node, we construct an html paragraph with the currect book's title (the value of the Title attribute node inside the Book element node). The reference to @Title is relative to the current book in the for-each loop.

Now we need an XML file to call the XSLT file. We can create an XML file (booktitles.xml) with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="booktitles.xsl"?>
<root/>
```

When we open this file in a browser, the browser will use the XSLT in booktitles.xsl in order to style the opened file. The XSLT will in turn open the XML file books.xml (which contains our sample data) and create one paragraph for each Book element found there. The result will look like this:



Books

Misty Nights

Archeology in Egypt

Database Systems in Practice

Contact

It is of course possible to add the reference to the XSLT file directly in the XML file books.xml. In that case the XSLT file does not have to use the XPath function "document", since it is already in that context. We can create a new version of the XSLT file as booktitles2.xsl with the content slightly modified:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head><title>Book titles</title></head>
            <body>
                <h1>Books</h1>
                <xsl:for-each select="//Book">
                    <p><xsl:value-of select="@Title"/></p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:transform>
```

Then we just add the following processing instruction to books.xml just before the opening tag <Books>:
<?xml-stylesheet type="text/xsl" href="booktitles2.xsl"?>

Now if we open the XML file books.xml in a browser we will get the same result as before.

4.3 *Multiple templates*

Another way to work with XSLT is to use templates recursively instead of using for-each loops. So let's try to create the same result as in the previous example by using templates instead of loops. We can create a new XSLT file (booktitles3.xsl) with the following content:

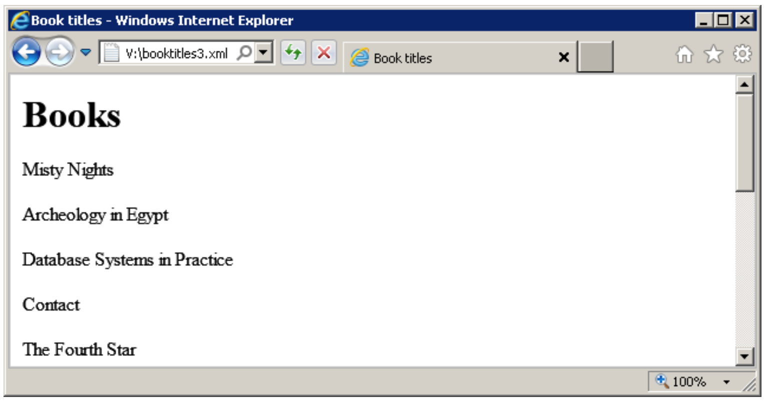
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head><title>Book titles</title></head>
            <body>
                <h1>Books</h1>
                <xsl:apply-templates select="document('books.xml')//Book" />
            </body>
        </html>
    </xsl:template>
    <xsl:template match="Book">
        <p><xsl:value-of select="@Title"/></p>
    </xsl:template>
</xsl:transform>
```

This XSLT file has two templates: one matching the root of the document, and one that matches Book elements. As usual, the first template will serve as the initiator of the process. It will start creating the html output and after the h1 element, it will ask for all the relevant templates to be applied to the selection. Since the selection is a sequence of Book element nodes, the relevant templates will be applied to each node in the sequence. The only relevant template for such nodes is our second template, which creates an html paragraph.

We can test our XSLT by creating a new XML file (booktitles3.xml) to call the XSLT. Its content should be the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="booktitles3.xsl"?>
<root/>
```

And the result will once again be the same as before:



Books

Misty Nights

Archeology in Egypt

Database Systems in Practice

Contact

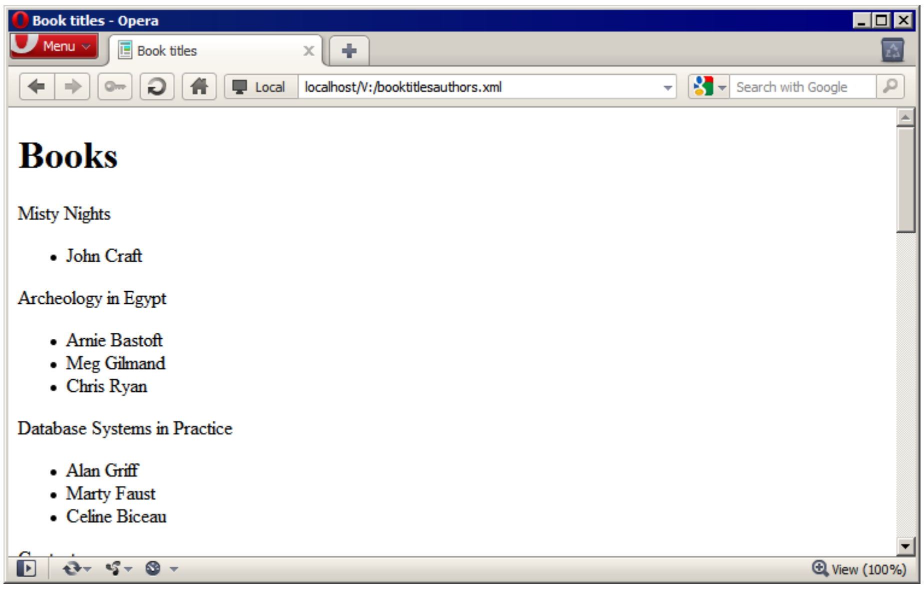
The Fourth Star

We can now expand this solution to also show the names of all the authors as an unordered list after each title. We can create a new XSLT file (booktitlesauthors.xsl) with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head><title>Book titles</title></head>
            <body>
                <h1>Books</h1>
                <xsl:apply-templates select="document('books.xml')//Book" />
            </body>
        </html>
    </xsl:template>
    <xsl:template match="Book">
        <p>
            <xsl:value-of select="@Title"/>
            <ul><xsl:apply-templates select="Author" /></ul>
        </p>
    </xsl:template>
    <xsl:template match="Author">
        <li><xsl:value-of select="@Name"/></li>
    </xsl:template>
</xsl:transform>
```

We have added a third template which matches Author elements and creates html list items. We have also modified the second template so that it not only creates a paragraph with the book title, but it also places, in the paragraph, an html unordered list whose content is the result of applying all relevant templates to the node sequence returned by the XPath expression "Author" (which of course is relative to the current Book element).

Using a new XML file (booktitlesauthors.xml) with a reference to the new XSLT file, we can get the result:



Books

Misty Nights

- John Craft

Archeology in Egypt

- Arnie Bastoft
- Meg Gilmand
- Chris Ryan

Database Systems in Practice

- Alan Griff
- Marty Faust
- Celine Biceau

The same result can of course be achieved by using nested for-each instructions like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head><title>Book titles</title></head>
            <body>
                <h1>Books</h1>
                <xsl:for-each select="document('books.xml')//Book">
                    <p>
                        <xsl:value-of select="@Title"/>
                        <ul>
                            <xsl:for-each select="Author">
                                <li><xsl:value-of select="@Name"/></li>
                            </xsl:for-each>
                        </ul>
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:transform>
```

4.4 Eliminating duplicates

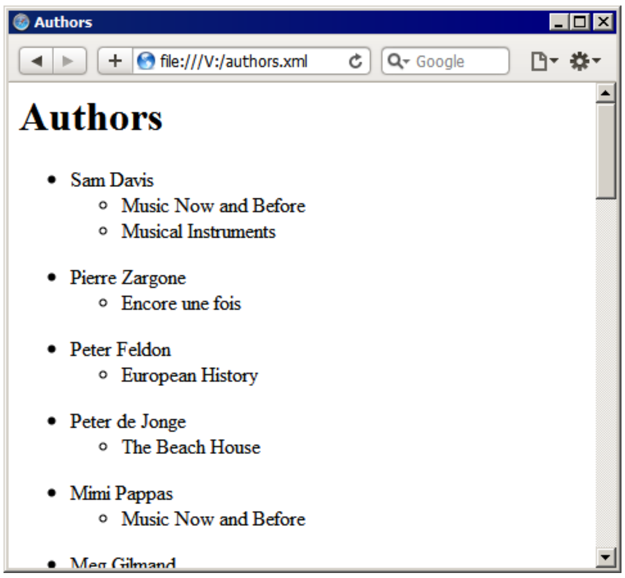
In XPath 2 we can use the function "distinct-values" in order to eliminate duplicate values. In XPath 1 we need to do this in a different way. We can use the "preceding" axis and check that no previous value is the same as the current one, thus only keeping the first occurrence of each value. We may want to create a list of all the authors with a sublist with each author's books. We could do this by finding all the authors (without duplicates) and then looking for books for the current author in a loop:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head><title>Authors</title></head>
            <body>
                <h1>Authors</h1>
                <xsl:variable name="doc" select="document('books.xml')" />
                <xsl:for-each select="$doc//Author[not(@Name = preceding::Author/@Name)]">
                    <xsl:sort select="@Name" order="descending" />
                    <xsl:variable name="n" select="@Name"/>
                    <ul>
                        <li>
                            <xsl:value-of select="@Name"/>
                            <ul>
                                <xsl:for-each select="$doc//Book[Author/@Name = $n]">
                                    <li><xsl:value-of select="@Title"/></li>
                                </xsl:for-each>
                            </ul>
                        </li>
                    </ul>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:transform>
```

Here we use a few new features. The first one is the XSLT instruction "variable". This is a simple instruction for assigning values to variables. The value is the result of the evaluation of the XPath expression specified in the select attribute. The variable can later be referenced in any XPath expression (by using the standard XPath variable mechanism: adding a dollar sign in front of the variable name). The second one is the "sort" instruction[2], which allows us to order the result of any loop. In this example we order the result according to the author names in descending order.

[2] Technically xsl:sort is not classified as an instruction in the specification. It is instead a support subelement to other instructions.

The result would look like this:



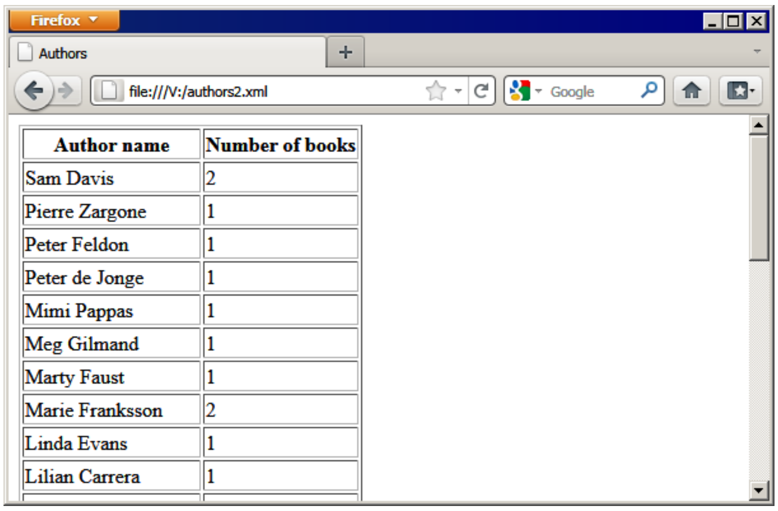
Authors

- Sam Davis
  - Music Now and Before
  - Musical Instruments
- Pierre Zargone
  - Encore une fois
- Peter Feldon
  - European History
- Peter de Jonge
  - The Beach House
- Mimi Pappas
  - Music Now and Before
- Meg Gilmand

We may want to show the number of books each author has written, instead of showing the titles. And we may want to put the result in an html table instead of a plain list. For that we can use the following XSLT:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <xsl:variable name="doc" select="document('books.xml')" />
      <html>
         <head><title>Authors</title></head>
         <body>
            <table border="1">
               <tr>
                  <th>Author name</th>
                  <th>Number of books</th>
               </tr>
               <xsl:for-each select="$doc//Author[not(@Name = preceding::Author/@Name)]">
                  <xsl:sort select="@Name" order="descending" />
                  <xsl:variable name="n" select="@Name"/>
                  <tr>
                     <td><xsl:value-of select="@Name"/></td>
                     <td><xsl:value-of select="count($doc//Book[Author/@Name = $n])"/></td>
                  </tr>
               </xsl:for-each>
            </table>
         </body>
      </html>
   </xsl:template>
</xsl:transform>
```

Here we use the XPath function "count", which returns the number of nodes (or values) in the sequence provided as the function's parameter. The result looks like this:



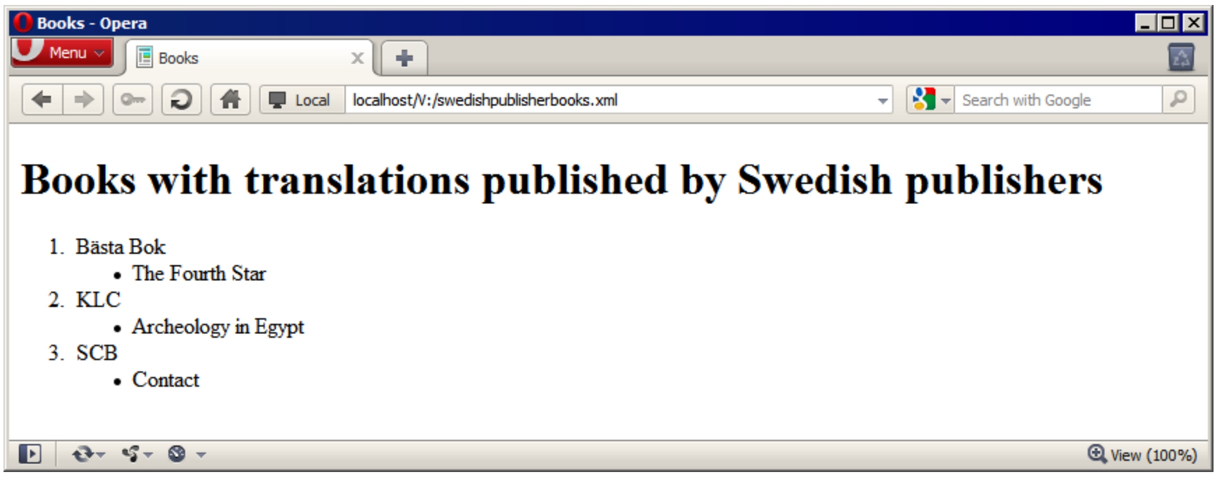
Author name	Number of books
Sam Davis	2
Pierre Zargone	1
Peter Feldon	1
Peter de Jonge	1
Mimi Pappas	1
Meg Gilmand	1
Marty Faust	1
Marie Franksson	2
Linda Evans	1
Lilian Carrera	1

4.5 *Multiple sources*

Sometimes the data we need may be available in different sources. As described earlier, our data is in two files: books.xml and publishers.xml. We may need to combine these two files, for example if we want to find books published by Swedish publishers. We can start by finding all the Swedish publishers in publishers.xml and then use the result in order to find books in books.xml that have been published by the right publishers. It is actually the translations of the different editions that have a publisher, so that's where we are going to look. The following XSLT does that and returns the titles of the right books.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:variable name="books" select="document('books.xml')" />
    <xsl:variable name="publishers" select="document('publishers.xml')" />
    <xsl:template match="/">
        <html>
            <head><title>Books</title></head>
            <body>
                <h1>Books with translations published by Swedish publishers</h1>
                <ol>
                    <xsl:apply-templates select="$publishers//Publisher[Address/Country = 'Sweden']"/>
                </ol>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="Publisher">
        <xsl:variable name="p" select="@Name" />
        <li>
            <xsl:value-of select="$p"/>
            <ul>
                <xsl:apply-templates select="$books//Book[Edition/Translation/@Publisher = $p]"/>
            </ul>
        </li>
    </xsl:template>
    <xsl:template match="Book">
        <li>
            <xsl:value-of select="@Title"/>
        </li>
    </xsl:template>
</xsl:transform>
```

As usual, the first template is used to initiate the result and it retrieves a sequence of Publisher element nodes (the publishers from Sweden) and applies all the relevant templates. The second template takes care of the Publisher elements and creates one list item for each. It also retrieves (from the other file) the relevant books and applies the relevant templates on them. The third template creates a list item for each Book element. The result looks like this:

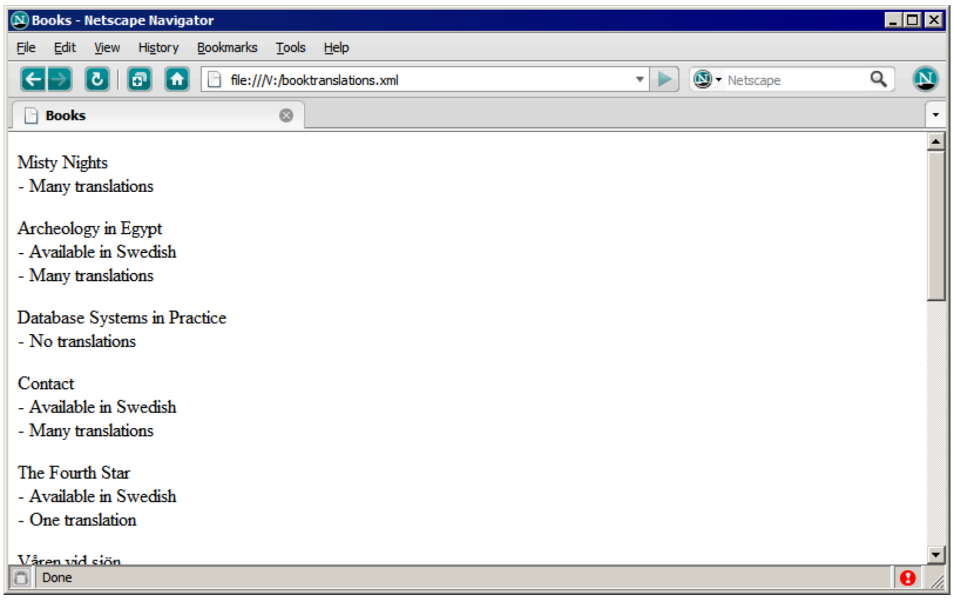


Books with translations published by Swedish publishers

1. Bästa Bok
    - The Fourth Star
2. KLC
    - Archeology in Egypt
3. SCB
    - Contact

4.6 Conditionals

XSLT, just as many other languages, has an "if" instruction and a "choose" instruction. The "if" instruction does not support an "else" clause, while the "choose" instruction can have several "when" clauses and possibly an "otherwise" clause. We could use these instructions when we want to vary the output based on certain conditions. Let's create an output that shows all the books and highlights the books that are available in Swedish, with some additional information about the number of translations. We want the output to look like this:


Books - Netscape Navigator
File Edit View History Bookmarks Tools Help
file:///V:/booktranslations.xml
Netscape
Books
Done

```
Misty Nights
- Many translations

Archeology in Egypt
- Available in Swedish
- Many translations

Database Systems in Practice
- No translations

Contact
- Available in Swedish
- Many translations

The Fourth Star
- Available in Swedish
- One translation

Våren vid sjön
```

We can achieve the desired output with the following XSLT:

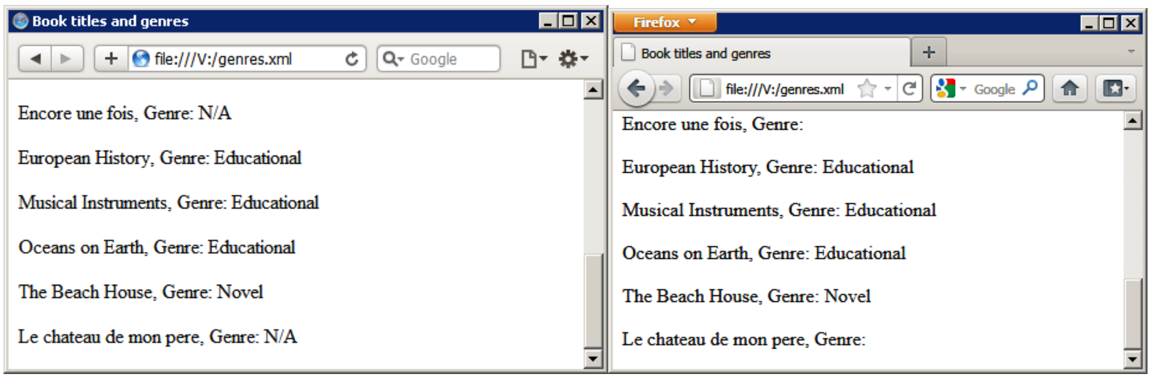
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:variable name="books" select="document('books.xml')" />
    <xsl:template match="/">
        <html>
            <head><title>Books</title></head>
            <body>
                <xsl:apply-templates select="$books//Book"/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="Book">
        <p>
            <xsl:value-of select="@Title"/>
            <xsl:if test="Edition/Translation/@Language = 'Swedish'">
                <br /> - Available in Swedish
            </xsl:if>
            <xsl:choose>
                <xsl:when test="count(Edition/Translation)=0">
                    <br /> - No translations
                </xsl:when>
                <xsl:when test="count(Edition/Translation)=1">
                    <br /> - One translation
                </xsl:when>
                <xsl:otherwise>
                    <br /> - Many translations
                </xsl:otherwise>
            </xsl:choose>
        </p>
    </xsl:template>
</xsl:transform>
```

4.7 Default values

One interesting aspect of XML is that the associated DTD or XML Schema may provide default values for attributes that may be missing in the XML document. In our sample data we have an attribute Genre that may be missing, and in that case the default value is "N/A". XSLT can use the information in the DTD and return the default value instead of nothing. We can try this by selecting all the book titles and genres with this simple XSLT (file genres.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head><title>Book titles and genres</title></head>
            <body>
                <h1>Books</h1>
                <xsl:for-each select="document('books.xml')//Book">
                    <p><xsl:value-of select="@Title"/>, Genre: <xsl:value-of select="@Genre"/></p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:transform>
```

Unfortunately, not all browsers implement the default lookup, so the result may differ like this:



```
Encore une fois, Genre: N/A
European History, Genre: Educational
Musical Instruments, Genre: Educational
Oceans on Earth, Genre: Educational
The Beach House, Genre: Novel
Le chateau de mon pere, Genre: N/A
```

```
Encore une fois, Genre:
European History, Genre: Educational
Musical Instruments, Genre: Educational
Oceans on Earth, Genre: Educational
The Beach House, Genre: Novel
Le chateau de mon pere, Genre:
```

4.8 Attribute value templates

When creating output, there are two main ways of creating nodes (elements, attributes, etc.). The one way is to use node creation instructions (like xsl:element, xsl:attribute, xsl:text, xsl:comment, etc). The other way is to write directly the textual representation of the nodes (as we have done in the examples earlier in this chapter). Let's look at a simple example with both these methods. We can use the XML document with the publishers and produce the following structure:

```
<Publishers>
    <Publisher Name="" Country="" />
    <Publisher Name="" Country="" />
</Publishers>
```

The first solution is with node creation instructions:

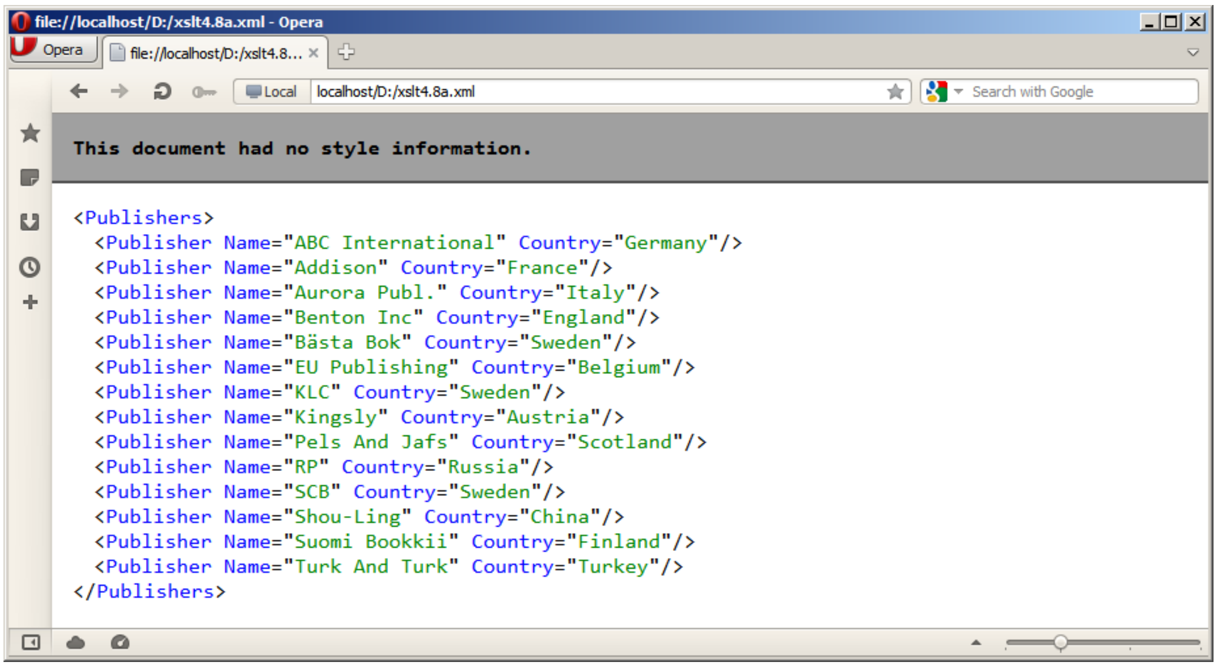
```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml"/>
    <xsl:variable name="publishers" select="document('publishers.xml')" />
    <xsl:template match="/">
        <xsl:element name="Publishers">
            <xsl:for-each select="$publishers//Publisher">
                <xsl:element name="Publisher">
                    <xsl:copy-of select="@Name"/>
                    <xsl:attribute name="Country">
                        <xsl:value-of select=".//Country"/>
                    </xsl:attribute>
                </xsl:element>
            </xsl:for-each>
        </xsl:element>
    </xsl:template>
</xsl:transform>
```

The second solution is to just type the xml content directly:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="xml"/>
   <xsl:variable name="publishers" select="document('publishers.xml')" />
   <xsl:template match="/">
      <Publishers>
         <xsl:for-each select="$publishers//Publisher">
            <Publisher Name="{@Name}" Country="{.//Country}" />
         </xsl:for-each>
      </Publishers>
   </xsl:template>
</xsl:transform>
```

Here we have to use attribute value templates like {@Name}. It is not possible to use xsl:value-of since that would cause the XSLT file to not be well-formed.

Both solutions produce the same result:


file://localhost/D:/xslt4.8a.xml - Opera
Opera
file://localhost/D:/xslt4.8...
Local
localhost/D:/xslt4.8a.xml
Search with Google

This document had no style information.

```
<Publishers>
  <Publisher Name="ABC International" Country="Germany"/>
  <Publisher Name="Addison" Country="France"/>
  <Publisher Name="Aurora Publ." Country="Italy"/>
  <Publisher Name="Benton Inc" Country="England"/>
  <Publisher Name="Bästa Bok" Country="Sweden"/>
  <Publisher Name="EU Publishing" Country="Belgium"/>
  <Publisher Name="KLC" Country="Sweden"/>
  <Publisher Name="Kingsly" Country="Austria"/>
  <Publisher Name="Pels And Jafs" Country="Scotland"/>
  <Publisher Name="RP" Country="Russia"/>
  <Publisher Name="SCB" Country="Sweden"/>
  <Publisher Name="Shou-Ling" Country="China"/>
  <Publisher Name="Suomi Bookkii" Country="Finland"/>
  <Publisher Name="Turk And Turk" Country="Turkey"/>
</Publishers>
```

Attribute value templates can of course be combined with node creation instructions in order to produce element names or attribute names that are dynamic. We could, for example, use the country as the element name instead of "Publisher". Try it!

4.9 *Grouping (XSLT 2)*

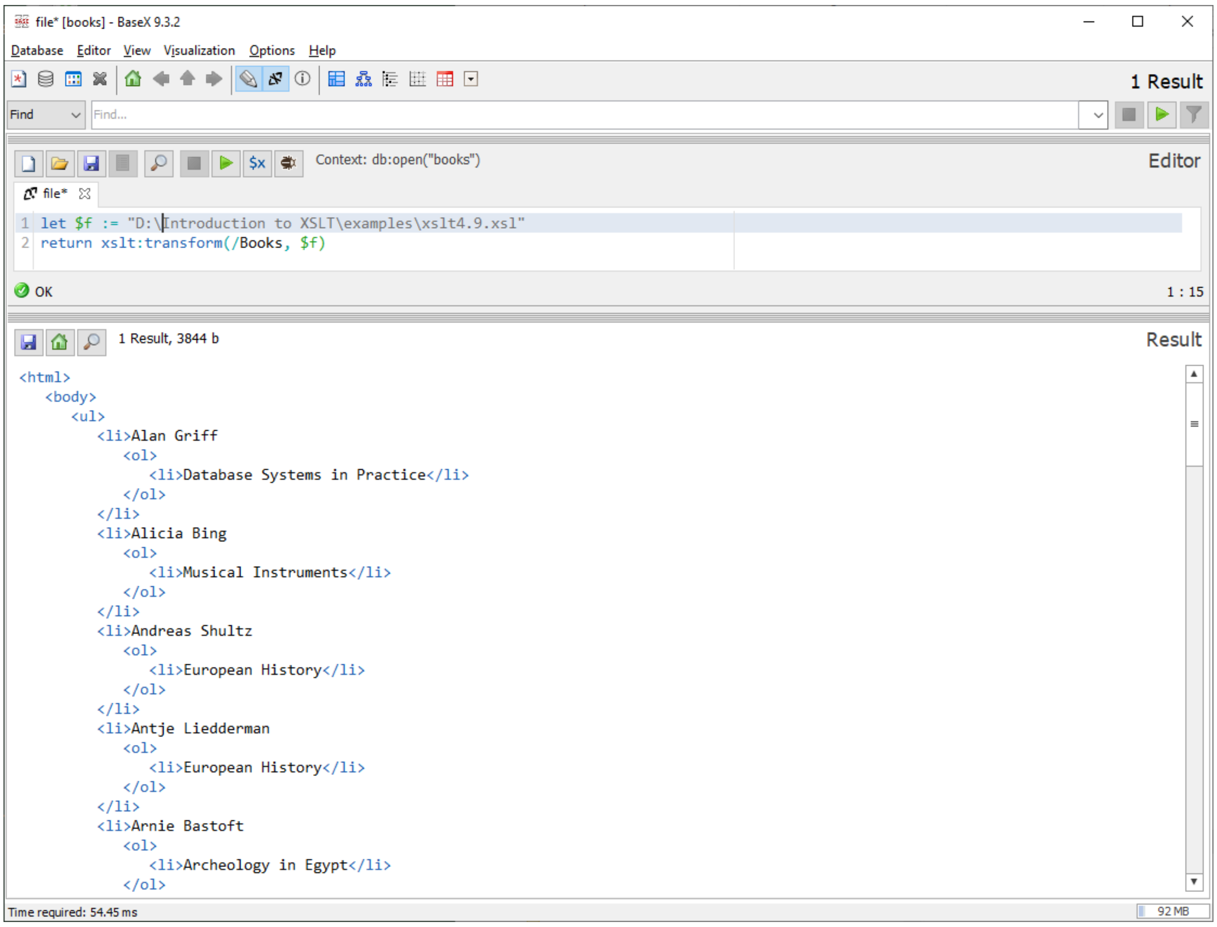
XSLT 2 introduces an instruction for grouping the iterations over a sequence. The instruction for-each-group can be used to group the iterations and perhaps produce some content once per group. Let's see how this instruction can help create an html page with a list of books per author.

```
<xsl:transform version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html"/>
    <xsl:template match="/">
        <html>
            <body>
            <ul>
                <xsl:for-each-group select="//Author" group-by="@Name">
                    <xsl:sort select="current-grouping-key()" />
                    <li>
                        <xsl:value-of select="current-grouping-key()"/>
                        <ol>
                            <xsl:for-each select="current-group()">
                                <li><xsl:value-of select="../@Title"/></li>
                            </xsl:for-each>
                        </ol>
                    </li>
                </xsl:for-each-group>
            </ul>
            </body>
        </html>
    </xsl:template>
</xsl:transform>
```

Here we take all the Author elements and group them by the value of the Name attribute. The function current-grouping-key() contains the value for the current group and can be used to sort the groups and later as output. The function current-group() contains all the Author elements that belong to the current group. Thus, to access the book titles of an author (which is represented by a group), we loop through the Author sequence returned by the current-group() and access the attribute Title of the parent element (Book is the parent element of the Author element).

Executing this example requires an XSLT 2 processor. Some ways to execute XSLT 2 and XSLT 3 are described in chapter 5.

Here is the result in BaseX:

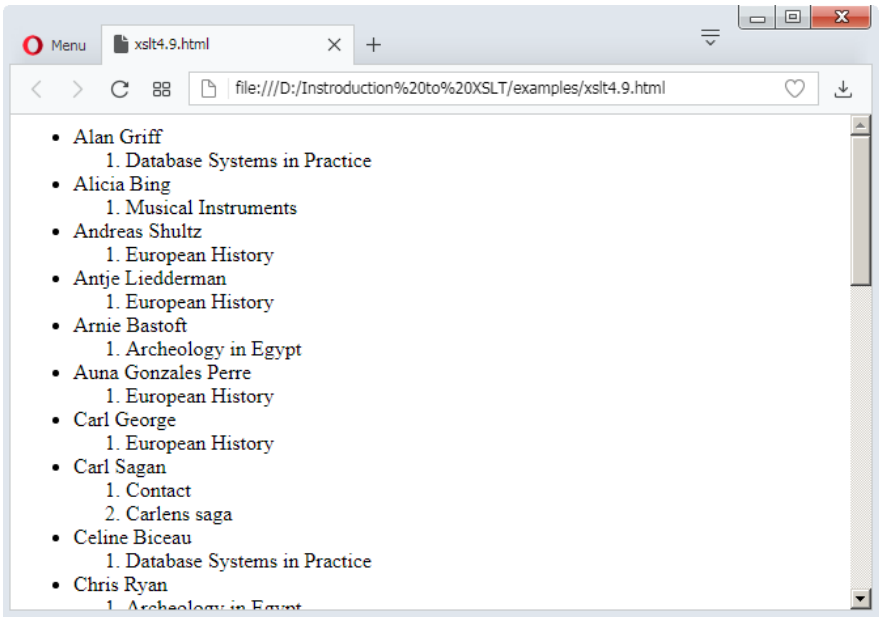

file* [books] - BaseX 9.3.2
Database Editor View Visualization Options Help
1 Result
Find
Context: db:open("books")
Editor
file*
OK
1 : 15
1 Result, 3844 b
Result
Time required: 54.45 ms
92 MB

```
let $f := "D:\Introduction to XSLT\examples\xslt4.9.xsl"
return xslt:transform(/Books, $f)
```

```
<html>
   <body>
      <ul>
         <li>Alan Griff
            <ol>
               <li>Database Systems in Practice</li>
            </ol>
         </li>
         <li>Alicia Bing
            <ol>
               <li>Musical Instruments</li>
            </ol>
         </li>
         <li>Andreas Shultz
            <ol>
               <li>European History</li>
            </ol>
         </li>
         <li>Antje Liedderman
            <ol>
               <li>European History</li>
            </ol>
         </li>
         <li>Arnie Bastoft
            <ol>
               <li>Archeology in Egypt</li>
            </ol>
```

Since the result is html and not necessarily well-formed XML, we could/should use the function xslt:transform-text instead.

BaseX does not provide a rendered version. By saving the result as an HTML file and opening it in a web browser we can see the rendered version:



- Alan Griff
  1. Database Systems in Practice
- Alicia Bing
  1. Musical Instruments
- Andreas Shultz
  1. European History
- Antje Liedderman
  1. European History
- Arnie Bastoft
  1. Archeology in Egypt
- Auna Gonzales Perre
  1. European History
- Carl George
  1. European History
- Carl Sagan
  1. Contact
  2. Carlens saga
- Celine Biceau
  1. Database Systems in Practice
- Chris Ryan
  1. Archeology in Egypt

4.10 Grouping with composite grouping key (XSLT 3)

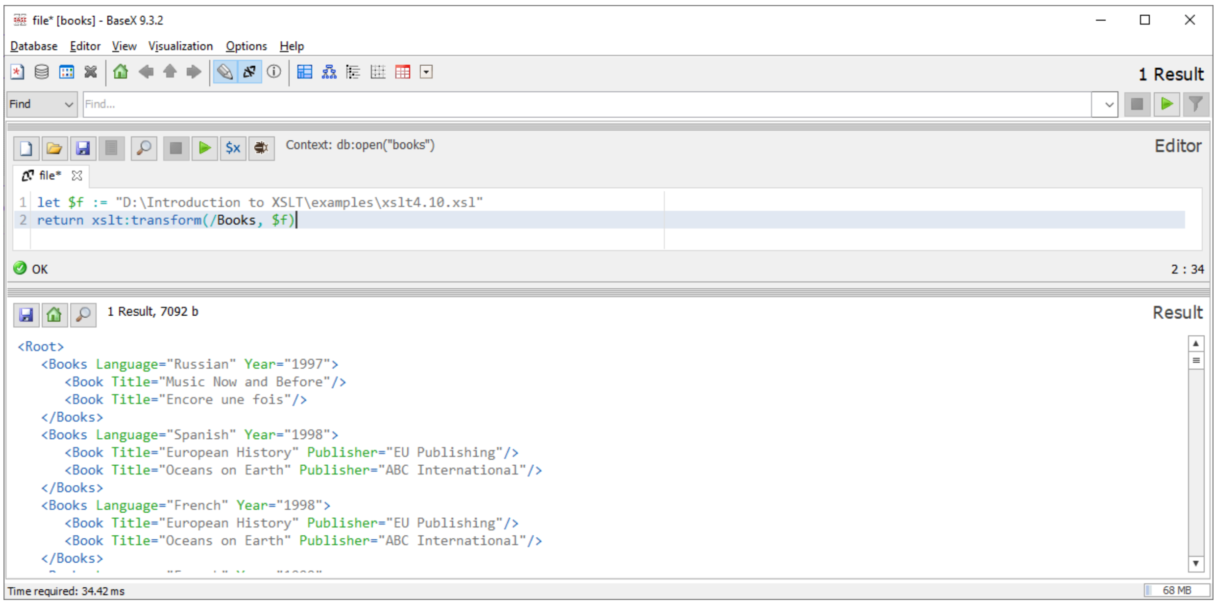
In the previous section we saw an example of a grouping where the value of only one attribute was used as the grouping key. But what if we need to group on a combination of things? Maybe we want to group the book translations based on the language and the year. And then, for each combination of language and year, show the title and publisher. This can be achieved by using a composite for-each-group. The element for-each-group has an attribute called composite that can be set to true. And then the attribute group-by can have a sequence representing the components of the composite grouping key. The following would then create one group for each combination of a translation language and a year:

```
<xsl:for-each-group select=".//Translation" group-by="(@Language, ../@Year)" composite="true">
```

But what happens to the grouping-key that we can retrieve with the current-grouping-key() function? Since the grouping is composite, the function returns a sequence consisting of the corresponding values in the same order as defined in the group-by attribute. So, the full solution could be the following (including some sorting showing the largest groups first and equal sized groups sorted by year, which is the second part of the grouping key):

```
<xsl:transform version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="/">
        <Root>
            <xsl:for-each-group select=".//Translation" group-by="(@Language, ../@Year)"
                                        composite="true">
                <xsl:sort select="count(current-group())" order="descending"/>
                <xsl:sort select="current-grouping-key()[2]"/>
                <Books>
                    <xsl:attribute name="Language" select="current-grouping-key()[1]"/>
                    <xsl:attribute name="Year" select="current-grouping-key()[2]"/>
                    <xsl:for-each select="current-group()">
                        <Book>
                            <xsl:copy-of select="ancestor::Book/@Title"/>
                            <xsl:copy-of select="@Publisher"/>
                        </Book>
                    </xsl:for-each>
                </Books>
            </xsl:for-each-group>
        </Root>
    </xsl:template>
</xsl:transform>
```

And here is the result in BaseX:


file* [books] - BaseX 9.3.2
Database Editor View Visualization Options Help
1 Result
Find
Editor
Result
68 MB

Context: db:open("books")

```
let $f := "D:\Introduction to XSLT\examples\xslt4.10.xsl"
return xslt:transform(/Books, $f)
```

OK 2 : 34

1 Result, 7092 b

```
<Root>
   <Books Language="Russian" Year="1997">
      <Book Title="Music Now and Before"/>
      <Book Title="Encore une fois"/>
   </Books>
   <Books Language="Spanish" Year="1998">
      <Book Title="European History" Publisher="EU Publishing"/>
      <Book Title="Oceans on Earth" Publisher="ABC International"/>
   </Books>
   <Books Language="French" Year="1998">
      <Book Title="European History" Publisher="EU Publishing"/>
      <Book Title="Oceans on Earth" Publisher="ABC International"/>
   </Books>
```

Time required: 34.42 ms

4.11 Dynamic XPath expressions (XSLT 3)

In some situations, it may be necessary to evaluate strings as XPath expressions. This is illustrated in the following example, even though it is possible to achieve the same result without evaluation of dynamic XPath expressions.

We can create a template that can be reused with different parameters in order to filter and sort books.

```
<xsl:template name="SortedFilteredBooks">
    <xsl:param name="filter"/>
    <xsl:param name="sort"/>
    <xsl:variable name="books">
        <xsl:evaluate xpath="'//Book[' || $filter || ']'" context-item="."/>
    </xsl:variable>
    <xsl:for-each select="$books/Book">
        <xsl:sort>
            <xsl:evaluate xpath="$sort" context-item="."/>
        </xsl:sort>
        <Book title="{@Title}" firstEdition="{min(Edition/@Year)}"/>
    </xsl:for-each>
</xsl:template>
```

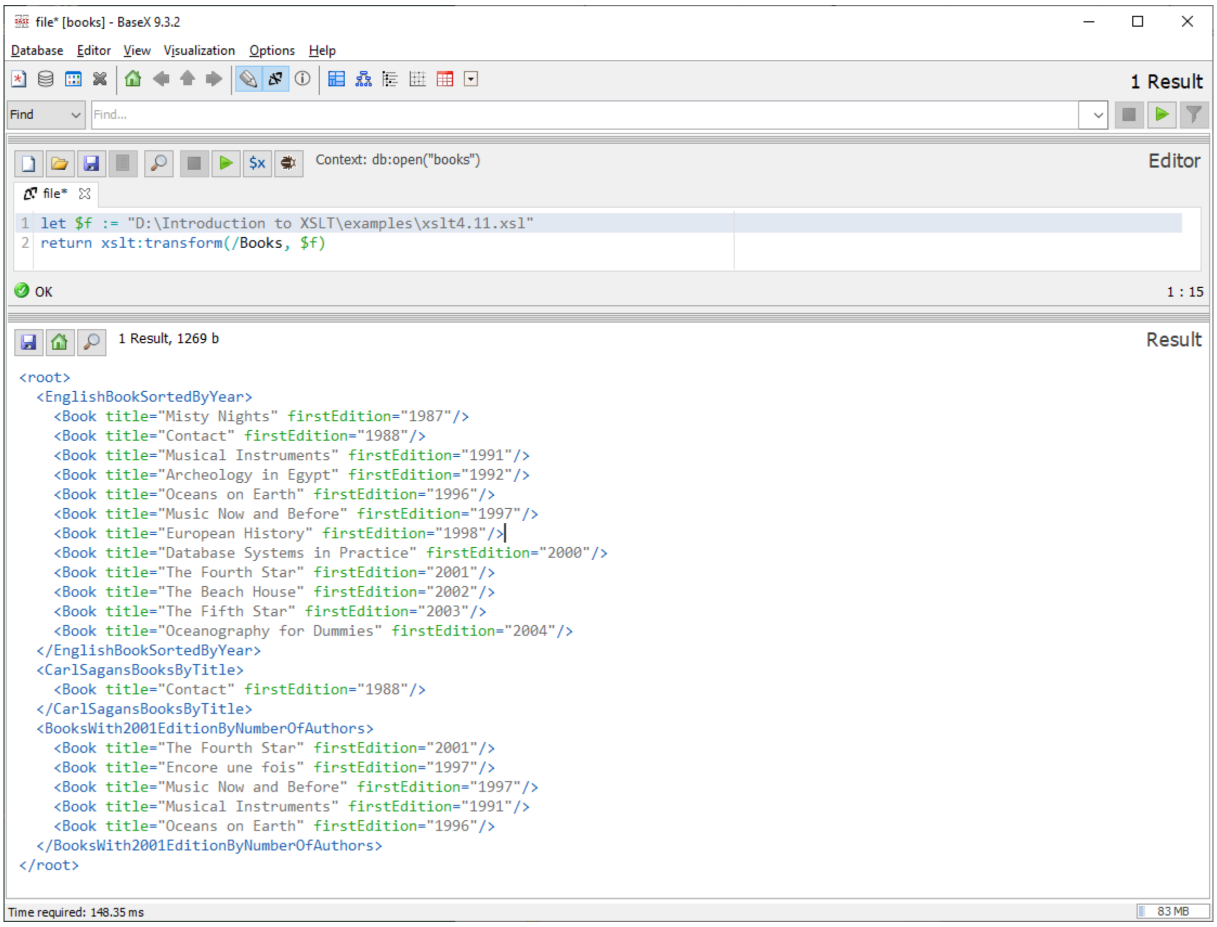
This template must be called with two parameters. The filter parameter must be a string that can be a predicate to filter books and the sort parameter must be a string that contains an XPath expression that can be applied to a book so that books can be sorted based on the result.

The filter parameter is concatenated in an XPath expression that is evaluated and the result is placed in the variable books. The sort parameter is evaluated inside an xsl:sort.

We can now call the template multiple times with different parameters. Here is a complete XSLT document where the template is called three times. It is necessary to create a variable "quote" in order to concatenate quotes inside a string that is inside an attribute.

```
<xsl:transform version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml"/>
    <xsl:variable name="quote">"</xsl:variable>
    <xsl:template match="/">
        <root>
            <EnglishBookSortedByYear>
                <xsl:call-template name="SortedFilteredBooks">
                    <xsl:with-param name="filter" select="'@OriginalLanguage='||$quote||'English'||$quote"/>
                    <xsl:with-param name="sort" select="'min(Edition/@Year)'"/>
                </xsl:call-template>
            </EnglishBookSortedByYear>
            <CarlSagansBooksByTitle>
                <xsl:call-template name="SortedFilteredBooks">
                    <xsl:with-param name="filter" select="'Author/@Name='||$quote||'Carl Sagan'||$quote"/>
                    <xsl:with-param name="sort" select="'@Title'"/>
                </xsl:call-template>
            </CarlSagansBooksByTitle>
            <BooksWith2001EditionByNumberOfAuthors>
                <xsl:call-template name="SortedFilteredBooks">
                    <xsl:with-param name="filter" select="'Edition/@Year=2001'"/>
                    <xsl:with-param name="sort" select="'count(Author)'"/>
                </xsl:call-template>
            </BooksWith2001EditionByNumberOfAuthors>
        </root>
    </xsl:template>
    <xsl:template name="SortedFilteredBooks">
        <xsl:param name="filter"/>
        <xsl:param name="sort"/>
        <xsl:variable name="books">
            <xsl:evaluate xpath="'//Book[' || $filter || ']'" context-item="."/>
        </xsl:variable>
        <xsl:for-each select="$books/Book">
            <xsl:sort>
                <xsl:evaluate xpath="$sort" context-item="."/>
            </xsl:sort>
            <Book title="{@Title}" firstEdition="{min(Edition/@Year)}"/>
        </xsl:for-each>
    </xsl:template>
</xsl:transform>
```

Here is the result in BaseX:


file* [books] - BaseX 9.3.2
Database Editor View Visualization Options Help
1 Result
Find
Editor
Context: db:open("books")
file*
OK
1 : 15
1 Result, 1269 b
Result
Time required: 148.35 ms
83 MB

```
let $f := "D:\Introduction to XSLT\examples\xslt4.11.xsl"
return xslt:transform(/Books, $f)
```

```
<root>
  <EnglishBookSortedByYear>
    <Book title="Misty Nights" firstEdition="1987"/>
    <Book title="Contact" firstEdition="1988"/>
    <Book title="Musical Instruments" firstEdition="1991"/>
    <Book title="Archeology in Egypt" firstEdition="1992"/>
    <Book title="Oceans on Earth" firstEdition="1996"/>
    <Book title="Music Now and Before" firstEdition="1997"/>
    <Book title="European History" firstEdition="1998"/>
    <Book title="Database Systems in Practice" firstEdition="2000"/>
    <Book title="The Fourth Star" firstEdition="2001"/>
    <Book title="The Beach House" firstEdition="2002"/>
    <Book title="The Fifth Star" firstEdition="2003"/>
    <Book title="Oceanography for Dummies" firstEdition="2004"/>
  </EnglishBookSortedByYear>
  <CarlSagansBooksByTitle>
    <Book title="Contact" firstEdition="1988"/>
  </CarlSagansBooksByTitle>
  <BooksWith2001EditionByNumberOfAuthors>
    <Book title="The Fourth Star" firstEdition="2001"/>
    <Book title="Encore une fois" firstEdition="1997"/>
    <Book title="Music Now and Before" firstEdition="1997"/>
    <Book title="Musical Instruments" firstEdition="1991"/>
    <Book title="Oceans on Earth" firstEdition="1996"/>
  </BooksWith2001EditionByNumberOfAuthors>
</root>
```

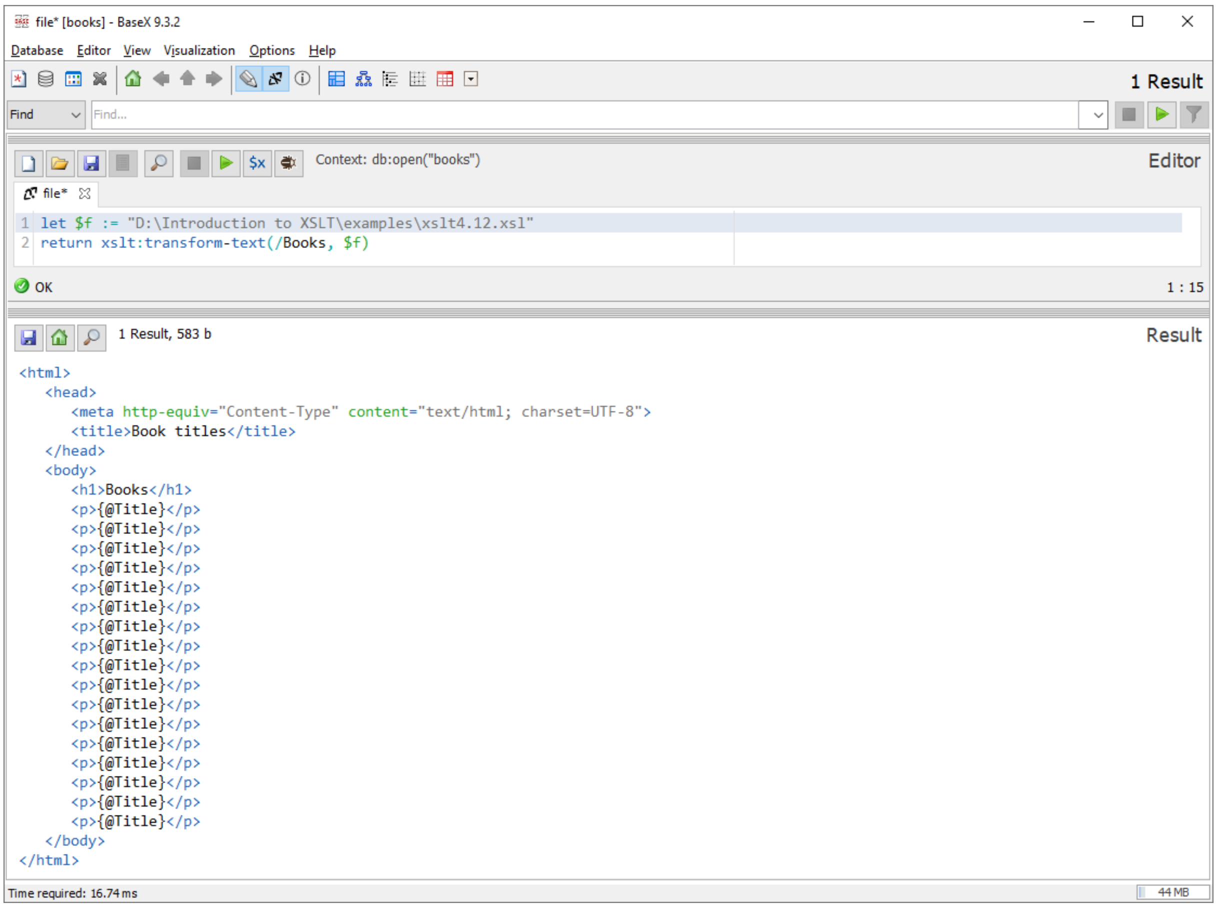
4.12 Text value templates (XSLT 3)

In section 4.2 we saw how to use the element value-of in order to get the value of a node and use it in the result being created. Later, in section 4.8, we saw how we could simplify our solution by using attribute value templates. In XSLT 1 and 2, we could use this convenient syntax only for attributes. In XSLT 3 we can use a similar syntax for text nodes as well.

Based on the example from section 4.2, we would expect the following to work:

```
<xsl:transform version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <html>
         <head><title>Book titles</title></head>
         <body>
            <h1>Books</h1>
            <xsl:for-each select="//Book">
               <p>{@Title}</p>
            </xsl:for-each>
         </body>
      </html>
   </xsl:template>
</xsl:transform>
```

But running this gives the following result:


file* [books] - BaseX 9.3.2
Database Editor View Visualization Options Help
1 Result
Find
Context: db:open("books")
Editor
file*
OK
1 : 15
1 Result, 583 b
Result
Time required: 16.74 ms
44 MB

```
let $f := "D:\Introduction to XSLT\examples\xslt4.12.xsl"
return xslt:transform-text(/Books, $f)
```

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Book titles</title>
   </head>
   <body>
      <h1>Books</h1>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
      <p>{@Title}</p>
   </body>
</html>
```

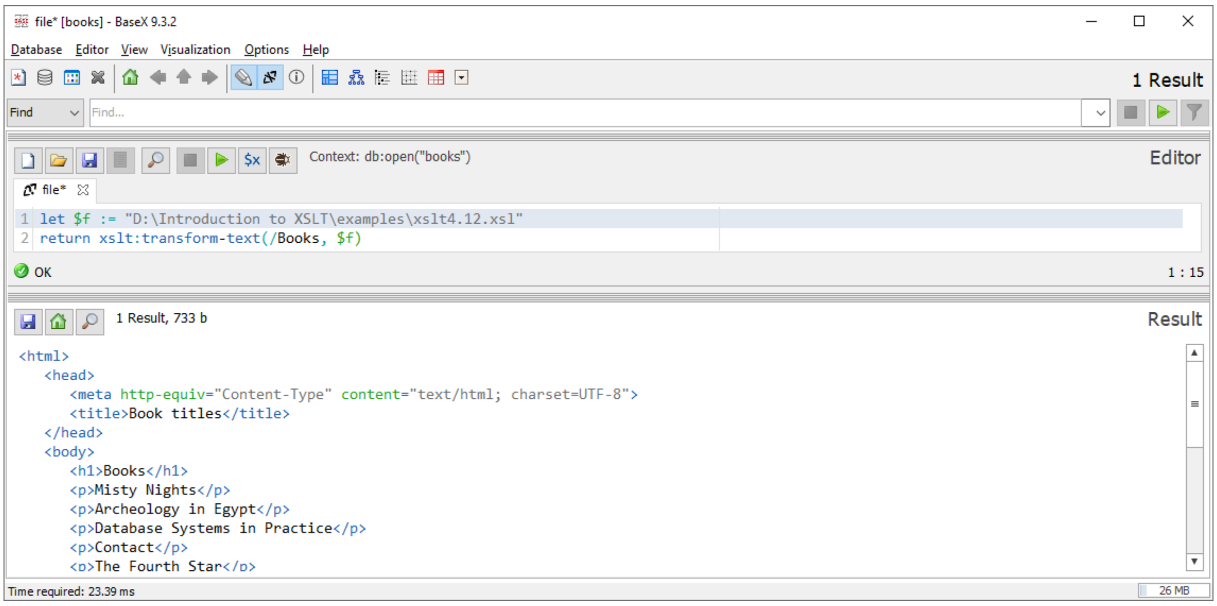
The expression {@Title} has not been evaluated. It has been treated as text. In some cases, this may be what we want. In order to force the expressions to be evaluated, we must add the attribute expand-text. This attribute can be added at any level, so we could add it to the root element transform:

```
<xsl:transform expand-text="yes" version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Or we could add it to the element p (with the right namespace):

```
<p xsl:expand-text="yes">
```

Or we could add it to any other suitable level. Either way the result will now be as expected:


file* [books] - BaseX 9.3.2
Database Editor View Visualization Options Help
1 Result
Find
Editor
Result
26 MB

Context: db:open("books")

```
let $f := "D:\Introduction to XSLT\examples\xslt4.12.xsl"
return xslt:transform-text(/Books, $f)
```

OK 1 : 15

1 Result, 733 b

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Book titles</title>
   </head>
   <body>
      <h1>Books</h1>
      <p>Misty Nights</p>
      <p>Archeology in Egypt</p>
      <p>Database Systems in Practice</p>
      <p>Contact</p>
      <p>The Fourth Star</p>
```

Time required: 23.39 ms

Using text value templates can make our code easier to read and reduce problems with white spaces and line breaks. Consider the following example:

```
<xsl:transform version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="true"/>
    <xsl:template match="/">
        <xsl:element name="Root">
            <xsl:for-each select="//Book">
                <xsl:element expand-text="yes" name="Book">
                    <xsl:attribute name="By">{string-join(Author/@Name, ", ")}</xsl:attribute>
                    "{@Title}" was written in {@OriginalLanguage} and exists in {count(Edition)} edition(s).
                </xsl:element>
            </xsl:for-each>
        </xsl:element>
    </xsl:template>
</xsl:transform>
```

5 Executing XSLT 2 and XSLT 3

Executing XSLT 2 and XSLT 3 requires a processor that supports the corresponding version. The web sites xsltransform.net and xslttest.appspot.com provide an interface that on the inside uses such processors. BaseX can also be configured to use an XSLT 2 or XSLT 3 Saxon processor.

5.1 xsltransform.net & xslttest.appspot.com

xsltransform.net and xslttest.appspot.com are two websites that provides a form where an XSLT document and an XML document can be written and then the result (of applying the XSLT document to the XML document) is computed and presented either as XML or as rendered HTML.

5.2 BaseX

BaseX comes with support for XSLT 1 by using the Saxon processor. BaseX can be configured to use a later version of the Saxon processor in order to support XSLT 2 or XSLT 3. To do this, download the latest Saxon processor jar file (saxon9he.jar, saxon9pe.jar or saxon9ee.jar) and place it in the lib folder of BaseX. Then start the BaseX GUI by running the file basexgui.bat (found in the bin folder of BaseX). The jar file will not load if BaseX GUI is started by the executable BaseX.exe. Some advanced functionality requires a license. A trial license can be acquired from saxonica.com (where the jar files are also available).

6 Epilogue

There are many more details about XSLT that we could discuss, but the goal of this compendium is only to serve as a "Getting started" introduction. There are many good resources about XSLT that could be of value if you want to become an expert. First and foremost, the XSLT specification. In many cases the specification may be too formal and hard to understand. So, books like "The XML 1.1 Bible" could serve you well.

In the examples that generate html as output, we used only very basic html components. It is of course possible to use the full power of html with images, css, javascript, etc. But the focus of the examples has been on the XSLT and not on the graphical design of the output.

I hope you have found this introduction educational and fun. Do not hesitate to send comments and suggestions that may help improve the next version of the compendium!

The Author
nikos dimitrakas