Studia UBB Digitalia, Volume 62 (LXII) 2017, June, Issue 1, 100-111

Published Online: 2017-06-30

DOI:10.24193/subbdigitalia.2017.1.08

eXist DB or Saxon/C in PHP. A comparison between two approaches for XSLT 2.0 based websites[*]

Christian Schwaderer

Eberhard Karls Universität Tübingen, Germany,
E-mail: c_schwaderer@hotmail.com

Abstract: XML and XSLT are very popular technologies among Digital Humanists. However, when it comes to deploying an XSLT 2.0 processor into a website infrastructure some difficulties arrise. This paper discusses two possible approaches for doing XSLT 2.0 transformations on the fly on a Web (Application) Server: Saxon as a PHP extension and Saxon within the popular XML database eXist. The conclusion shows, that both solutions have their pros and cons. In the end, however, the PHP solution wins.

Keywords: XML; XSLT; PHP; eXist.

There is no need to prove that XML is popular among Digital Humanists. While in recent times, JSON became probably the most important exchange format for structured data in commercial applications, it doesn't meet the needs of humanists, who often have to deal with semi-structured texts and thus with mixed content. Therefore, there is no real alternative to XML and none is in sight in the near future either.

This being the case, there is a need for practical ways to present XML structured data to end users in a web interface. While there are nowadays various ways to process and transform XML data, at least in a DH context, using XSLT seems to be among the most common methods.[1]

[*] I'd like to thank Christopher Johnson and O'Neil Delpratt for their valuable help and feedback. Of course, any remaining errors are mine.

[1] Take for example last year's programme of the most important summer school in Digital Humanities. In the workshop “Text to Tech” (http://dhoxss.humanities.ox.ac.uk/2015/text2tech.html) “a hands-on introduction” to ”XPath, and XSLT” was announced.

While deciding how input data should be displayed on the screen and writing XSLT code for that purpose, might be regarded as the main and major work process in building a digital humanities website, it is – unfortunately – not all that is to be done. By having sophisticated XSLT templates transforming input XML to pretty HTML pages, you are just half way there in setting up a web application. If you only want to transform data locally on your own machine, then you maybe an IDE like oXygen fits your needs best. However, once you decide to build a website:

1. You need to make sure that the user gets the appropriate HTML file for her request. In other words: You have to find solutions for session management.
2. You might want to give your users a site internal search.
3. Finally, you need some management solution for your XSLT processing. While you could start the XSLT processor of your choice from the command line on your local machine and copy the generated HTML files manually to the place you want to have them, this process becomes annoying if you are dealing with dozens of transformation processes and hundreds of result pages. If you want interactive components on your website with the help of XSLT, you have no choice anyway: You need a program for starting XSLT transformations on the server.

Probably, there are at least half a dozen solutions for the needs briefly outlined above. Many things depend on which XSLT processor you are using. However, if you don't want to miss the features of XSLT 2.0 and XPath 2.0 the Saxon XSLT processor family might be the choice for you. Thus, the number of methods narrows down. If you don't want to sue Saxon on the command line or write your web application in Java, there are not too many options on how to integrate Saxon into a website infrastructure which highly relies on using XSLT 2.0 for creating HTML files the end users get to see.

In this text, I want to outline my experience with two different approaches.

Roughly speaking, the first way (which I did for my Database of the Letters of Pope Gregory VII project[2]) is as follows:

- Run a web server like Apache on a server.
- Store your XML data on a server's file system.
- Install Saxon/C as a PHP extension.
- Develop PHP scripts for starting Saxon, storing the result HTML files on the file system and/or delivering them to the client.

In the “Editiones Electronicae Guelferbytanae”, the XSLT code itself is estimated as of being of such relevance as scholarly data that they even publish it together with the XML.

Of course, not all people like XSLT: “I don’t know about you, but I honestly can’t stand XSLT.” Jesse Alama, XSLTXT: A more compact form of XSL, April 20, 2015 http://goxrxyourself.com/2015/04/20/xsltxt-a-more-compact-form-of-xsl/.

[2] http://www.g7ldb.history.uni-tuebingen.de/.

- For the setting up a search engine on your website, build a PHP script which turns what user input in the search form into an XPath expression, then put this XPath into an XSLT script and process it.

The alternative (which is what I did in my edition of “Adelbert of Heidenheim”[3]) would be:

- Run eXist XML database on a server.
- Put your XML data into eXist. eXist will then store it internally as binary files.
- Develop XQuery scripts for starting Saxon, storing the HTML results within eXist and/or delivering them to the client. (As of now, Saxon already comes with eXist.)
- For the search, set up the eXist index options and develop an XQuery script which queries your data using the Lucene query engine deployed in eXist.

In the sections to follow, I'd like to compare both approaches and discuss their pro's and con's. The versions used are Saxon/C 0.3.1 beta and eXist db 3.0RC1.

Installation and set-up

eXist ships almost ready to use out of the box. All you have to do is run the installation jar and “answer a few questions” on the way through the installation process. Afterwards, in order to have eXist installed as a service, you need to run `$EXIST_HOME/tools/wrapper/bin/exist.sh install` – and you're done. (Unless something goes wrong like a port conflict with Tomcat or something like that.)

In order to use Saxon within PHP (and not via the command line) you have to install the Saxon/C PHP extension provided by Saxonica. And this is not so easy, since yet, there is no installation script for the various Linux distributions (I did not try it on Windows). So, you have to do a few steps by hand, like compiling, making, and setting things in PHP.ini. Personally, I had a lot of trouble and needed a lot of help from the Saxonica guys, before Saxon/C finally worked on openSUSE 13.1.

However, once the software runs, in the Saxon/C PHP extension scenario, you are done. There is pretty nothing to configure or set up. Either it runs or it doesn't. But if it does, there is nothing left to do.

Though you can start using eXist right after its installation, for XSLT users, there are some things to do. For example, in order minimize the white space problem (see below), in `$EXIST_HOME/conf.xml`, you have to ensure that the `preserve-whitespace-mixed-content` attribute is set to `yes`.

[3] http://www.forschungsdatenarchiv.escience.uni-tuebingen.de/adlils/

Also, there is a problematic default setting called “XSLT caching”. You can easily get rid of it by finding `<transformer class="net.sf.saxon.Transformer FactoryImpl"..` element and setting `caching` to `no`. Otherwise importing or including XSLT files into other XSLT files (`<xsl:import href="...xsl"/>`) will cause you trouble, because the included/imported files will be taken from the cache, thus often being in an outdated state if you are currently working with them.

File management and data handling

We have already mentioned that eXist does not directly use XML documents on the file system but stores the XML iput in internally as binary data. While this is necessary for any XML database to work efficiently, it may affect the things you can do with your files and which you can't.

First of all, the eXist data directory consumes double the hard disk space that HTML files on the file system do. While in most cases, this might not be a problem at all, some other issues are.

If your XML data and your automatically created HTML files are stored on the file system you remain in full control over them. You can manipulate files with non XML-aware script languages like PHP or whatever. That's important in case you want to achieve something which is difficult in pure XSLT.

Take, for instance, the HTML5 doctype definition `<!DOCTYPE html>`. There is no “official” way to output it in XSLT 2.0.[4] However, you can easily write a small script which simply prepends the doctype as a string to all your HTML files. PHP doesn't care about the content of your files, you can tamper in any way you want with your files, regardless if you produce well formed XML or not. It's all up to you.

eXist on the other hand, is an XML database, so it cares if your data is well formed XML or not. It also cares about doctypes, putting them out according to its serialization settings (and removing them from the input). So, if you want to do your own thing, you probably will have a hard time. Additionally, XQuery is not a language designed and suitable for manipulating files and strings. (I didn't succeed in prepending `<!DOCTYPE html>` to all my HTML files stored in eXist.)

Concerning backup and version control, as far as I know there are no external tools to do this in eXist. Sure, any application that is able to deal with the XMLDB API[5], could make use of the data stored in eXist, but I don't know of any. (Additionally, even if you have an external backup of eXist's binary files there is no guaranty that you will be able to get them to work within eXist again, as I have experienced myself.)

[4] For some tricks, see http://www.whoop.ee/posts/2014/08/22/valid-html5-doctype-with-xslt.html
[5] See http://xmldb-org.sourceforge.net/xapi/ . As far as I know, there is not much support for this API anyway.

In the end, you have to rely on eXist's own built-in functions.

One really enervating issue is that eXist's doesn't exactly store XML documents the way you put them in. Not only does it remove the doctype definition (as mentioned forehand) it also tampers with white space.

For example, putting in

`<tei:rdg><tei:supplied>et</tei:supplied></tei:rdg>` will by eXist be turned into:

```
<tei:rdg>
   <tei:supplied>et</tei:supplied>
</tei:rdg>
```

Now imagine, you have a template for `tei:rdg` which wraps its contents in brackets and one for `tei:supplied` which puts quotation marks around the nodes inside. You probably expect a result like `(“et”)`. However, with the added white space by eXist you will get `( “et” )` which at least looks silly, is simply not what you want, and an absolutely unnecessary inconvenience. (In my case, I had developed two routines for my apparatus entries: One for single words and one for more than one word. The distinction was made by the existence of white space: If there was any the input was treated as two words. That worked perfectly till I began using eXist. Now, because of the added white space all single words were treated like they were two words. It took some days to figure out a solution here.)

XSLT support and error handling

Despite Saxon being in both solutions the XSLT processor in the back end, there are some issues to point out in the handling and integration of XSLT.

When speaking of any code processor, compiler, and interpreter, one of the first things developers are interested in is the information you get if your code produces an error. Getting detailed error reports is absolutely crucial for debugging.

Using an XSLT script which is terminated by an XSL message (`<xsl:message terminate="yes">I am a terminate message!</xsl:message>`), we test the output we get. For both solutions, the most detailed information is to be found in the log files.

For Saxon/C in PHP we will see the following in the Apache “error_log” file:

```
I am a terminate message!
Error at xsl:message on line 15 of param.xsl:
  XTMM9000: Processing terminated by xsl:message at line 15 in param.xsl
Exception in thread "main" net.sf.saxon.s9api.SaxonApiException: Processing
terminated by xsl:message at line 15 in param.xsl
```

```
      at net.sf.saxon.s9api.XsltTransformer.transform(Unknown Source)
      at net.sf.saxon.option.cpp.XsltProcessorForCpp.xsltApplyStylesheet(Unknown
Source)
Caused by: net.sf.saxon.expr.instruct.TerminationException: Processing terminated
by xsl:message at line 15 in param.xsl
      at net.sf.saxon.expr.instruct.Message.processLeavingTail(Unknown Source)
```

(Four further lines of Java stack trace omitted.)

For other errors, you even get the number of the column:

```
Error at xsl:variable on line 14 column 69 of param.xsl: XTTE0570: Required
item type of value of variable $my_val is xs:integer; supplied value has
item type xs:string
```

In eXist's main log file “exist.log” the output is slightly different:

```
2015-11-12 15:04:57,243 [eXistThread-95] WARN  (Transform.java
[fatalError]:817) - XSL transform reports fatal error: Processing
terminated by xsl:message at line -1 in null
net.sf.saxon.expr.instruct.TerminationException: Processing terminated by
xsl:message at line -1 in null
      at net.sf.saxon.expr.instruct.Message.processLeavingTail(Message.java:223)
~[?:?]
      at net.sf.saxon.expr.instruct.Choose.processLeavingTail(Choose.java:796)
~[?:?]
      at net.sf.saxon.expr.instruct.Choose.processLeavingTail(Choose.java:796)
~[?:?]
      at net.sf.saxon.expr.instruct.Instruction.process(Instruction.java:131)
~[?:?]
```

And then you get about two hundred lines of Java stack trace which I omit here since it is of no use for debugging XSLT code.

You see, there are two main differences:

- In eXist, the actual message is not there. It's not logged in the main log file, but passed to the general system output, thus being either in the console window or in “EXIST_HOME/tools/wrapper/logs/wrapper.log”. That's a little bit inconvenient, but once you know where to look for it, it's just a few clicks extra.
- The line number of the XSLT snippet causing the problem is missing in eX-ist. That's really painful and makes debugging for a file with 5041 lines (my main XSLT file) almost impossible.

Still speaking of error handling, it is not only important what is put out to the log files but also how the wrapping PHP respectively XQuery script itself behaves in case there are some XSLT errors.

With Saxon/C in PHP, if calling `transformToString()` results in an XSLT error, nothing happens. You will just get an empty string as a result and the rest of the PHP script is executed as normal. (If Saxon/C itself throws an error like running out of memory that's another story. However, in version 0.3.1 I have never experienced something like that.)

After the execution of `transformToString()`, you can check the result and possibly access any error messages like so:

```
if($xslt_result_string == NULL) {
         $errCount = $xslt_processor_instance->getExceptionCount();
         if($errCount > 0 ) {
          for ($i = 0; $i < $errCount; $i++) {
             $errC = $xslt_processor_instance->getErrorCode(intval(i));
             $errMessage = $xslt_processor_instance-
>getErrorMessage(intval(i));
             if($errC != NULL) { echo 'Error: '. $errC.' :'.$errMessage;}
             $xslt_processor_instance->exceptionClear();
            }
          }
         }
```

eXist's XQuery function for XSLT transformation (namely `transform: transform()`) behaves slightly different. If the processor runs into an error while processing your XSLT, eXist will display an error page with the XSLT error and the rest of the XQuery script will be skipped. In order to keep the XQuery running even after it encountered an XSLT error you have to wrap the function call in a try-catch-block. If `catch *` is triggered you can put out the error messages by accessing the variables `$err:code`, `$err:description`, and `$err:value`.

However, the problem is: Not all XSLT errors are caught in eXist. Errors of the type: `exerr:ERROR XSL transform reported error:....` are treated as expected, while errors of the type `exerr:ERROR Exception while transforming node...` are not caught at all. In the latter case, the query puts out everything which is noted in the XQuery script before the block with the `transform:transform()` function call – and nothing more. The rest of the query seems not to be processed at all (including the error message).

Aside from error handling, any XSLT wrapper should provide a convenient way to pass parameters to the XSLT script. Generally, that works with both approaches pretty well, though not completely intuitive. In PHP, you first have to create an `XdmValue` before passing it to Saxon:

```
$xdmval_for_my_param = $xslt_processor_instance->createXdmValue("example str");
$xslt_processor_instance->setParameter('my_param', $xdmval_for_my_param);
```

(For reasons I don't not know you cannot directly pass `$xslt_processor_instance->createXdmValue("example string")` as the second parameter of `setParameter` when using a string. For integers, it works that way.)

In eXist, you have to wrap all your XSLT parameters in a `<parameters>` element like:

```
let $transform_parameters :=
 <parameters>
   <param name="my_param" value="'example str'"/>
 </parameters>
and then pass $transform_parameters as a parameter to transform:transform().
```

However, the problem in eXist is that, as of now, you can only pass string values as parameters to Saxon. (That should change once the `fn:transform` function specified in XQuery 3.1 will be implemented in eXist.)

Not a real problem, but a fact one should be aware of when using eXist for doing XSLT, is that all the options set in `xsl:output` are ignored by eXist (instead, the settings you pass as serialization parameters are taken into account).

Furthermore, there seems to be a problem in eXist's XSLT handling when loading XML documents which are stored inside eXist into XSLT variables (like: `<xsl:variable name="my_var" select="doc('foo.xml')"/>`): An XSLT transformation started in eXist returns for the XSLT command `xsl:value-of` not only text nodes (as you would expect) but also the values of XML attribute nodes.[6]

Additionally, in order to get elements in the order they are stored in your document you need a silly XSLT line like `<xsl:sort select="(count(preceding::*) + count(ancestor::*))"/>`. And furthermore, using variables of these kind significantly slows down the performance. A transformation which took 20 seconds before using a variable document of this kind, took 10 minutes after adding `$my_var` into an expression like `<xsl:for-each select="//some_element">` (thus accessing the document in the variable instead of the input document).

Speaking of performance, aside from the phenomenon described above, I did not do any thorough tests. So, I cannot assess whether Saxon runs generally faster when started from within eXist or from within PHP. However, even without having done any real measuring I'd dare to say that the difference (if there is any) is rather small.[7]

[6] See https://github.com/eXist-db/exist/issues/791 for a description of this and other XSLT issues in eXist.

[7] For the performance of Saxon/C in general, see Michael Kay and Debbie Lockett, Benchmarking XSLT Performance. Presented at XML London 2014, June 7-8th, 2014. doi:10.14337/XMLLondon14.Kay01.

Setting up a search

Almost every website offers its users a full text search and there is no reason why XML/XSLT based ones should be an exception. However, realizing an on-the-fly querying of XML files with X Technologies requires a little bit of work to do. (I won't discuss other approaches for setting up a website search like Apache Solr, Google Custom Search, and so on.)

As of now, in Saxon/C you cannot query XML files with pure XPath expressions. (That will change in next version of Saxon/C, however). This being the case, there is only one way to search and query XML input based on user input: creating an XSLT or XQuery file with a random string inside a select expression, turning the user input to an XPath expression or an "XPath compatible" text snippet via PHP, replacing your random string with the XPath expression just created and then running the XSLT or XQuery as string. (In the lines to follow, I will only show XSLT, though since Saxon 0.3.1, XQuery is also an option and probably an easier one.)

For example, your XSLT could look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="2.0">
 <xsl:template match="/">
  <h1>Search results</h1>
  <xsl:for-each select="//some_element[contains(.,'Az9HpPC')]">
   <p><xsl:value-of select="."/></p>
  </xsl:for-each>
 </xsl:template>
</xsl:stylesheet>
```

In PHP, you have to do the following:

```
$search_term = $_GET["search_term"]
$xslt_raw = file_get_contents("my_xslt_file.xsl");
$xslt_with_actual_search_term =
str_replace('Az9HpPC',$search_term,$xslt_raw);
$xslt_processor_instance = new SaxonProcessor();
$xslt_processor_instance->setSourceFile('my_xml_file.xml');
$xslt_processor_instance-
>setStylesheetContent($xslt_with_actual_search_term);
$xslt_result_string = $xslt_processor_instance->transformToString();
```

While this is feasible, it gets a little bit more complicated, if you want to offer your users an “advanced” search, i.e. combining different input fields with checkboxes, dropdown inputs, and so on. In this case, you have to put together a more complex XPath expression. In PHP, those XPath expressions will be treated as strings, so during development you will never get any hints on XPath syntax errors you may produce.

For advanced full text search options like similarity search, truncation and boolean searching you either have to build your own XPath functions or find existing libraries appropriate for your use case. Either way, there is some work to do.

In eXist, of course you can also use your own XPath functions for offering a full text search. However, there is no need to do it, since eXist has a great built-in solution for this. So, eXist can play to its strength in this category. eXist fully integrates Lucene, thus providing you and your users with a full developed query syntax. Also, there are built-in functions for ordering search results and highlighting a user's search term within a search result text block.

Configuring the indexes for the Lucene full text search in eXist is not completely intuitive, but requires a little bit of reading through the docs. It is not complicated, though.

Since your search management script will be written in XQuery, you won’t have any trouble building complex XPath expressions. You have full XPath support while writing an XQuery search script in the XML IDE of your choice.

Long term maintenance

Especially for humanities projects, long term availability matters. We all hope, that even in decades (not to talk about centuries) our web applications will still be there in the open web and still be running. As of now, nobody knows whether this will be the case and who should be in charge for keeping projects up. However, one thing is for sure: The faster and easier it is to keep a web application running and a website alive, the more likely it is that someone will actually do it – be it a member of the original project team or a somebody of library or computing service stuff.

So, the question here is: Which solution is more convenient when it comes to maintenance and keeping things alive?

Basically, there are two cases:

- The current software on which a web application relies needs an update (for whatever reason).
- The current software does not run anymore and is to be replaced by another software with similar functionality.

eXist is definitely more common than Saxon/C is. This being the case and keeping in mind what we have said about the installation process, eXist wins in the first scenario: It is more likely that you will get support and the necessary information on how to replace one eXist version with another than you will receive on Saxon/C. Triggering a full backup, uninstalling the current eXist version, installing a new one and restoring the backup will in most cases be rather smooth and won't cause too much trouble – unless you use eXist features which will be removed in newer versions.

However, in the second scenario, things are different. eXist has created its own XQuery dialect and thus offers much functionality which is not part of the XQuery standard and which is not even among the things for which XQuery originally was designed for, I.e. things like `request:get-cookie-value()`. It is very unlikely that there will ever be any XML database with the very same XQuery functions. So, in most cases, the end of eXist will be the end of most web applications running in it – even if the data itself will be rescued.

However, even if there will be a similar XML database with similar XQuery functions, then still, there will be a lot of manual maintenance work to do. Someone has to check not only all internal paths (like `/db/apps/foo/`) inside a web application but also every single function call, replacing eXist's function names with those of the new database and – more complicated – adapting the XQuery function parameters to the new XML database.

On the other hand, chances are, that in 30 years, there will still be web servers with PHP processors. Replacing Apache HTTP Server should be easy anyway (there are millions of instances, one can be sure that as long as there are still servers one will find instructions how to replace Apache with another web server). In case you still find an XSLT 2.0 processor which runs as a PHP extension then after installing it your job will be almost done. You do not have to go through the whole code, finding functions of a special programming language dialect like for eXist's XQuery functions. All you have to do is replacing one single block of code with another.

```
$xslt_processor_instance = new SaxonProcessor();
$xslt_processor_instance->setSourceFile(...);
$xslt_processor_instance->setStylesheetFile(...);
$xslt_processor_instance->setParameter(...);
$xslt_processor_instance->transformToString();
```

Those five lines are everything which is specific to Saxon in PHP, everything else you need for a “normal” web application in PHP is just part of the PHP standard.

So, the PHP solution wins in the second scenario.

Conclusion

First of all, I'd like to point out that my comparison might be unfair, since eXist is much more than just a wrapper for XSLT. It offers many features in addition to what is outlined above, e.g. URL rewriting/mapping.

However, when trying to summarise the aspects discussed we get the synopsis below:

	eXist-db and XQuery	**Saxon/C as PHP extension**
Installation and set-up	+ Easy A few things to do in the configuration file.	- Difficult and complicated. + Nothing to do in configuration.
File management and data handling	- Complete dependence on eXist's built-in tools for manipulating data, backup, and version control. - Adds white space in XML data.	+ Full control over files. External tools may be used for version control, backup, and manipulating files.
XSLT support and error handling	- Only strings as parameters. - Bug in handling documents in variables. - No line numbers in error messages. - Not all XSLT errors catchable.	+ Full detailed error message directly in Apache's error_log. + All types of parameters. + PHP scripts never stop because of an XSLT error.
Setting up a search	+ Full developed search engine with advanced syntax. + built-in functions for search result highlighting and ordering. + Native XPath support since XQuery is used.	- You have to develop your own XPath functions for search and result highlighting. - No syntax help while building XPath expressions in PHP.
Long term maintenance	+ More widely used. - Hard to replace.	- Probably rarely used. + Easier to replace completely.

As you can see, the PHP solution wins in the most important category: XSLT support. Among the rest, the pluses and minuses are distributed rather evenly. Depending on what you are willing to do, convenient built-in functions for setting up a search or full control over your input files might be more important for you. Personally, I'm in favour of Saxon/C as a PHP extension.