

Proof Of Concept: XSLTProcessor extension

Marcin Kurzyna

February 24, 2008

Synopsis

Proof Of Concept idea is to check whether it's possible to provide custom (identified by prefix) processing tags to XSL transformations done with PHP's XSL extension.

Contents

Proof Of Concept: XSLTProcessor extension.....	1
Synopsis.....	1
Introduction.....	3
Problem description.....	3
Solution idea.....	4
Problem: complex, data dependent controls.....	5
Problem: pre- and postprocessing – how do we chose?.....	7
Problem: order of appearance and nested tags.....	8
Proof of Concept Implementation.....	9
Usage example.....	11

Introduction

Some time ago, while researching template technologies decision was made to use XML+XSL transformations as a template engine for [neFR series of frameworks](#). It was argued that generic XML data provided by application could be then transformed to any output required whether it'd be XHTML or other. Another benefit of XSL transformations was that template writers didn't have to be programmers. Although steeper than for Smarty templates the learning curve for XSL templates wasn't a problem. The syntax was "HTML document like" with only few extra control tags; thus it was easy to understand for people used to writing HTML pages.

Problem description

neFR framework version 2.x used XML+XSL transformation with much success. It quickly became apparent however that the built-in functions and processes weren't sufficient. Many calls were made to PHP's internal functions or application objects using `<php:function />` tag. While this method did solve the lack of certain functionality it wasn't considered "the right way". It violated the presumption of not having to resort to programming skills and application function knowledge to write templates. It also clouded the overall template readability. Another thing was that enabling direct PHP calls opened the possibility to modify application data which, under certain circumstances, could be a security risk (publicly edible templates for once).

Most available template engines allow for writing plugins to add new functionality. This is the most obvious way of providing means to do things that weren't foreseen by original developers or are application specific and wouldn't be required in general implementation. While this is also possible with XSL processor used by PHP's extension it's only available on the C language level – this is how php prefixed tags were added. There are unfortunately no PHP level hooks provided for adding new control tags or functions.

Solution idea

Because there is no way to directly add new functionality an idea came up to preprocess XSL stylesheet before applying final transformation. Prefixed tags should be replaced by output provided by application level function resulting in raw XSL document. Identified were following basic assumptions:

- marking tags for replacement,
- binding application functions to tags.

For the first assumption XML contained already present solution: tag prefixes and namespaces. Function bindings have to be implemented in PHP. An example XSL document could look as follows:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:my="http://proof-of-concept.eu/xsl/postprocess">

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    <my:date/>
  </xsl:template>
</xsl:stylesheet>
```

As each XSL stylesheet is first loaded as DOMDocument it would be fairly easy to find all tags with “my” prefix and “date” local name. Those tags should be then replaced by output of date() function. The resulting stylesheet could look like:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:my="http://proof-of-concept.eu/xsl/process">

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    2008-02-24 00:20
  </xsl:template>
</xsl:stylesheet>
```

Such stylesheet could be later applied to XML data producing desired end result.

Problem: complex, data dependent controls

First problem with this method is when trying to create more complex controls: ones that depend on XML data or more generally on the result of XSL transformation. Consider following implementation of `sprintf()` access tag:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:my="http://proof-of-concept.eu/xsl/process">

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    <my:sprintf format="%01.2f € - %s">
      <variable><xsl:value-of select="price"/></variable>
      <variable><xsl:value-of select="title"/></variable>
    </my:sprintf>
  </xsl:template>
</xsl:stylesheet>
```

The XML data could be:

```
<data>
  <price>34.1</price>
  <title>XSL extension</title>
</data>
```

The processing result should look like:

```
34.10 € - XSL extension
```

There are two possible solutions. First is to write a complex parser that would understand XSL controls execute them and then substitute `<my:sprintf />` tag with proper string; this is insane. Second possibility is not to preprocess XSL stylesheet but postprocess it. First transform document and then replace “my” prefixed tags.

This would allow for simple task of handling following XML code:

```
<my:sprintf format="%01.2f € - %s">  
  <variable>34.1</variable>  
  <variable>XSL extension</variable>  
</my:sprintf>
```

This can be done with any of typical methods (DOMDocument transformations, XMLReader/XMLWriter, etc) and is outside the scope of this document.

Problem: pre- and postprocessing – how do we chose?

So now we have two types of possible processing: before and after XSL transformation. We have to be able to decide how to process a tag. This can be done with namespaces. By defining different namespaces we can select and process tags in different manner independently of the prefix they're assigned.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:pre="http://proof-of-concept.eu/xsl/preprocess"
  xmlns:post="http://proof-of-concept.eu/xsl/postprocess">

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    <post:sprintf format="%01.2f € - %s - %s">
      <variable><xsl:value-of select="euro"/></variable>
      <variable><xsl:value-of select="title"/></variable>
      <variable><pre:date/></variable>
    </post:sprintf>
  </xsl:template>
</xsl:stylesheet>
```

By selecting tags by namespace we first process “pre” tags changing `<pre:date />` to some date. Then we apply XSL transformation retrieving values from XML data. Finally we replace `<post:sprintf/>` tag with final string.

Problem: order of appearance and nested tags

There is yet another problem: order of appearance. By default we get our tags “as they appear” in the document in a “flat” manner. In the previous example we’ll first get `<post:sprintf/>` and later the `<pre:date />` tag.

But when we have nested tags (like above) we should process them inside out. Start with the deepest node and move upwards so that if the `<pre:date />` tag was also a postprocess tag it would be processed before `<post:sprintf/>` one as it requires date’s output being it’s input. There is no automated way of doing this. It has to be manually implemented by counting node’s depth and processing them in depth order. Node depth can be obtained with Xpath query by counting node’s ancestors:

```
$depth = $Xpath->query("ancestor-or-self::*")->length;
```

Proof of Concept Implementation

The initial implementation that was written to test whether it's possible to hook into XSL processing allows for post processing tags and function binding. The test code implements:

- tag postprocessing
- possibility to use any valid PHP callback for processing (processed node is passed as only argument)
- prefix to processing callback mapping
- binding of multiple prefixes
- processor class implements strategy pattern so it's possible to handle different tags with one class

The code does not take node depth into account so complex nested tags substitutions will fail.

```

<?php

class processor {
    public function transform(DOMNode & $node) {
        call_user_func(array($this, $node->localName), $node);
    }

    private function sprintf(DOMNode & $node) {
        $sxe = simplexml_import_dom($node);

        $doc = $node->ownerDocument;
        $new = $doc->createTextNode(vsprintf(
            (string)$sxe['format'],
            $sxe->xpath('variable')
        ));

        $node->parentNode->insertBefore($new, $node);
        $node->parentNode->removeChild($node);
    }
}

class MyXSLProcessor extends XSLTProcessor {
    private $_extensions = array();

    public function registerExtension($name, $callback) {
        $this->_extensions[$name] = $callback;
    }

    public function transformToXML(DOMDocument $dom) {
        $partial = $this->transformToDoc($dom);
        $nodes = $partial->getElementsByTagName(
            'http://proof-of-concept.eu/xsl/process','*');

        foreach($nodes as $node) if(isset($this->_extensions[$node->prefix]))
            call_user_func($this->_extensions[$node->prefix], $node);

        return $partial->saveXML();
    }
}

?>

```

Usage example

XSL stylesheet:

```
<?xml version="1.0" standalone="true" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:my="http://proof-of-concept.eu/xsl/process">

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    <my:sprintf format="%01.2f € - %s">
      <variable><xsl:value-of select="price"/></variable>
      <variable><xsl:value-of select="title"/></variable>
    </my:sprintf>
  </xsl:template>
</xsl:stylesheet>
```

XML document:

```
<?xml version="1.0" standalone="true" ?>
<data>
  <price>34.1</price>
  <title>XSL extension</title>
</data>
```

PHP code:

```
<?php
  $dom = new DOMDocument();

  $xsl = new MyXSLProcessor();
  $xsl->registerExtension('post', array(new processor, 'transform'));

  $dom->load($xsldocument);
  $xsl->importStyleSheet($dom);

  $dom->load($xmldocument);

  echo $xsl->transformToXML($dom);
?>
```