cobj_xpath

version 1.3.0

Torsten Schrade

2013-08-11 12:48

Contents

XPATH Content Object 1
- Introduction 1
  - What does it do? 1
  - Features 1
  - Credits 1
  - Please rate 1
- Installation 1
  - Requirements 1
  - Installation 2
- Reference 2
- Tutorials 5
  - XPATH and stdWrap 5
  - Using XPATH to read Flexform values 7
  - Transform & display an external RSS feed 8
  - FLUIDTEMPLATE, XPATH & XSLT 11
  - <xpath> TypoTag 17
- Known problems 19
- To-Do list 19
- ChangeLog 19

XPATH Content Object

Classification:	cobj_xpath
Version:	1.3.0
Language:	en
Description:	Manual for the extension cobj_xpath
Keywords:	forEditors, forAdmins, forBeginners, forIntermediates
Copyright:	2013

Author:	Torsten Schrade
Email:	schradt@uni-mainz.de

Rendered:	2013-08-11 12:48

The content of this document is related to TYPO3, a GNU/GPL CMS/Framework available from www.typo3.org.

Introduction

What does it do?

This extension adds a new content object XPATH to the classic TypoScript cObjects. The XPATH cObject allows for a flexible, TypoScript based treatment of any kind of XML data. The data can be read from database fields, files or external resources.

Features

- TypoScript based approach to XML processing
- Many possibilities to work on the XML data with stdWrap or parseFunc
- Works with database fields, files or external XML resources
- Get values from TYPO3 Flexforms or Gridelements with TypoScript
- Offers different return formats for the results matched by the XPath query
- Debug errors with the TSFE admin panel

Many things can be realized like chaining of XPATH objects or transformation of the XML data either with TypoScript, PHP (userFuncs) or XSLT . Have a look at the tutorials in this manual.

Credits

This extension has been developed for the Digital Humanities projects of the Digital Academy Mainz where we have to deal with lots of XML. Glad if you find it useful too.

Please rate

We're always looking forward to your ratings in TER. Good ratings encourage us to go on, bad ratings encourage us to make the stuff better ;)

Installation

Requirements

- You need at least PHP 5.2+

- You need libxml and SimpleXML enabled
- The extension has been tested on TYPO3 versions 4.5 up to 6.1

Installation

Simply import it from TER and install. Nothing else needs to be done.

Reference

This section gives an overview on all TypoScript properties of the XPATH content object. You can also have a look into the README.txt file within the doc folder to get a configuration example.

Property

source

Data type

string/stdWrap

Description

This fetches the XML data from a source. Can be a XML string, a field in the database, a file (path or via TypoScript FILE cObject) or an external resource.

Example (field):

```
page.10 = XPATH
page.10 {
   source.data = page : my_xml_field
   [...]
}
```

Fetches the XML from the field 'my_xml_field' of the current page record.

Example (stdWrap / FILE):

```
page.10 = XPATH
page.10 {
   source.cObject = FILE
   source.cObject.file = fileadmin/myfile.xml
   [...]
}
```

This fetches the XML from a file included by TypoScript's FILE content object. Important: Due to FILE's internal settings, the data can't be larger than 1024kb. See TSref.

Example (external):

```
page.10 = XPATH
page.10 {
   source = http://news.typo3.org/rss.xml
   [...]
}
```

This draws the XML from an external source. It can be an URL like above or an external file resource of any size.

Property

registerNamespace

Data type

string/+ subproperties

Description

Registers a namespace for use with the XPATH expression. Syntax is **prefix|namespace** . The namespace must match a namespace in the source, otherwise the XPATH query will return false.

Example:

```
page.10 = XPATH
page.10 {
registerNamespace = c|http://example.org/chapter-title
expression = //c:title
[...]
}
```

It's possible to extract the namespaces of the XML source with the following subproperties:

Subproperties:

```
.getFromSource [boolean]

.getFromSource.debug [boolean]

.getFromSource.listNum [integer]
```

getFromSource will retrieve the namespaces from the source rather than taking the string given in the parent property. With **debug** it's possible to see what namespaces are returned. **listNum** is a TypoScript listNum with which you can select any of the (possibly several) namespaces returned from the XML source.

Property

expression

Data type

string/stdWrap

Description

XPATH expression.

Example:

```
page.10 = XPATH
page.10 {
   expression = //item
   [...]
}
```

Gets all <item> nodes from the XML source.

Example (with stdWrap):

```
page.10 = XPATH
page.10 {
   expression = //item[{register:count}]
   expression.insertData = 1
   [...]
}
```

Fetches the item by the number found in the TypoScript register.

Property

return

Data type

keyword/stdWrap

Description

This sets the return value for the XPATH query. Can be one of the following keywords:

count

Returns the number of the nodes/attributes matched by the XPATH expression

boolean

Returns true or false depending if the XPATH expression matched any nodes/attributes

xml

Returns all matched nodes and their child nodes as XML. Should be used in conjunction with *.resultObj* or *.directReturn*

array

Converts and returns all nodes matched by the XPATH expression in an array structure. Should be used in conjunction with *.resultObj* or *.directReturn*

json

Converts and returns all nodes matched by the XPATH expression in json format. Should be used in conjunction with *.resultObj* or *.directReturn*

string

Converts and returns all items matched by the XPATH expression as strings (atomic node values). Should be used in conjunction with *.resultObj* or *.directReturn*

Example:

```
page.10 = XPATH
page.10 {
  source.data = page : my_xml_field
  expression = //title
  return = string
  [...]
}
```

Default

string

Property

resultObj

Data type

→ see TSref split

Description

As the name says, the result object contains the result of the XPATH query (i.e. all matched nodes, attributes, etc). The resultObj works similar to the well known TypoScript split property. This makes the handling of the returned items very flexible. You can use option split, stdWrap, parseFunc and all the other nice stuff from TSref :)

Example:

```
page.10 = XPATH
page.10 {

  source.data = page : my_xml_field
  expression = //title
  return = string

  resultObj {
    cObjNum = 1 || 2
```

```
    1.current = 1
    1.wrap = <h1 style="color:red">|</h1>

    2.current = 1
    2.wrap = <h1 style="color:green">|</h1>
  }
}
```

Property

implodeResult

Data type

boolean/+token

Description

Instead of processing the XPATH result set with resultObj, this setting directly returns the whole set imploded around a token. This way you can split or explode the result yourself and do further processing, depending on your usecase. Can be useful for passing on result arrays to a FLUIDTEMPLATE for example.

token (string/stdWrap)

Sets the token around which the result set is imploded.

Default

###COBJ_XPATH###

Property

stdWrap

Data type

stdWrap

Description

stdWrap properties for the XPATH cObject

Example:

```
page.10 = XPATH
page.10 {

  [...]

  stdWrap {
    outerWrap = <code>|</code>
    htmlSpecialChars = 1
  }
}
```

Tutorials

XPATH and stdWrap

Let's explain some basic techniques for selecting nodes from an XML source and afterwards displaying them in the FE using stdWrap. We have the following XML file:

```
<?xml version="1.0"?>
<collection>
 <cd>
  <title>Fight for your mind</title>
  <artist>Ben Harper</artist>
  <year>1995</year>
```

```
 </cd>
 <cd>
   <title>Electric Ladyland</title>
   <artist>Jimi Hendrix</artist>
   <year>1997</year>
 </cd>
</collection>
```

We now want to select all CD titles and display them using alternative colors. Wow, that's a real usecase, isn't it? ;). We decide to load the source using the FILE object within our XPATH object. The XPATH expression for selecting all title attributes within the document is simply: //title. Since the titles are atomic node values, we decide to return our result as strings. We end up with the following TypoScript:

```
page.10 = COA
page.10 {

  10 = XPATH
  10 {
    source.cObject = FILE
    source.cObject.file = fileadmin/xpath/collection.xml

    expression = //title

    return = string
  }
}
```

We have used source's stdWrap properties to apply a FILE object that loads the contents of our XML file from fileadmin. But wait, something is wrong, we don't see anything on the website yet... Let's check what's wrong with the TSFE admin panel.

XPATH object debug output in TSFE Admin Panel

As we can see from the debug output, loading the XML data was successful but we forgot to configure a resultObj for the XPATH query. The resultObj is a key component of the XPATH object. Since each XPATH query returns an array of matches, we need an instrument to process the results. So it is a bit like using renderObj for CONTENT elements. The resultObj property works similar to the TypoScript split property and has the same subproperties. That provides us with a flexible tool for handling our results.

Let's output the first title in red, the second in green. The resultObj for doing that looks like this:

```
page.10 = COA
page.10 {

  10 = XPATH
  10 {
    source.cObject = FILE
    source.cObject.file = fileadmin/xpath/collection.xml

    expression = //title

    return = string
```

```
    resultObj {
      cObjNum = 1 || 2

      1.current = 1
      1.wrap = <h1 style="color:red;">|</h1>

      2.current = 1
      2.wrap = <h1 style="color:green;">|</h1>
    }
  }
}
```

The option split determines that the first match is treated with object 1, the rest with object 2. This makes it possible to configure different wraps for our atomic node values. And here it is in all it's beauty:

Using XPATH to read Flexform values

This tutorial shows you how you can use the XPATH content object to read XML data from a database field and retrieve values from a TYPO3 Flexform without much hazzle. Lets say we want to get the value of the summary attribute of a table content element:

Flexform of the table content element

Let's have a look at this Flexform's XML data to locate the value we are targeting:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<T3FlexForms>
   <data>
      <sheet index="sDEF">
         <language index="lDEF">
            <field index="acctables_caption">
               <value index="vDEF">This is the caption</value>
            </field>
            <field index="acctables_summary">
               <value index="vDEF">This is the summary</value>
            </field>
            <field index="acctables_tfoot">
               <value index="vDEF">0</value>
            </field>
            <field index="acctables_headerpos">
               <value index="vDEF"></value>
            </field>
            <field index="acctables_nostyles">
```

```
            <value index="vDEF">0</value>
          </field>
          <field index="acctables_tableclass">
            <value index="vDEF">myclass</value>
          </field>
        </language>
      </sheet>
      <sheet index="s_parsing">
        <language index="lDEF">
          <field index="tableparsing_quote">
            <value index="vDEF"></value>
          </field>
          <field index="tableparsing_delimiter">
            <value index="vDEF">124</value>
          </field>
        </language>
      </sheet>
    </data>
</T3FlexForms>
```

For retrieving the value, we need to match the index attribute of the <field> tag and then get to the atomic node of the <value> tag using it's index attribute. Our TypoScript looks as follows:

```
page.10 = XPATH
page.10 {

  source.data = DB:tt_content:2:pi_flexform

  return = string

  expression = //field[@index='acctables_summary']/value

  resultObj {
    cObjNum = 1
    1.current = 1
    1.wrap = <p><em>|</em></p>
  }
}
```

Using stdWrap on the source property gets us the content element. The XPATH expression targets the atomic node in the Flexform. Of course everything could have been made much more flexible. But that's for you to try out ;)

Transform & display an external RSS feed

It is quite common to display content from an external XML feed. There are many good extensions in TER that cater for this need. Most of the time they take the approach to import the external source into the TYPO3 database (for example into tt_news which is then used for record display). When you just want to retrieve, parse, format and display a feed the XPATH content object can step in as "TypoScript only" solution.

Say we want to display the official TYPO3 newsfeed from http://news.typo3.org/rss.xml on our website. Using the XPATH object, we can retrieve the XML feed, select it's items and format them with TypoScript's parseFunc. Let's start with the basics:

```
page.10 = XPATH
page.10 {

  # first we set the source to the newsfeed url
  source = http://news.typo3.org/rss.xml

  # each news entry is wrapped in <item> tags, fetch them with XPATH expression
  expression = //item
```

```
  # return the <item>s as XML which is going to be formatted with parseFunc later on
  return = xml

  # before we do the parseFunc stuff, just return the content to see what we've got
  resultObj {
    cobjNum = 1
    1.current = 1
  }

  # let's us display the output on the website for analysis
  stdWrap.htmlSpecialChars = 1
}
```

When we reload the page we can see the items matched by our XPATH query:

```
<item>
  <title>FLOW3 1.0.3 has been released</title>
  <link>
    http://news.typo3.org/news/article/flow3-103-has-been-released/
  </link>
  <description>
    FLOW3 1.0.3, the third patch release of the PHP application framework has been released.
  </description>
  <category>Development</category>
  <category>FLOW3</category>
  <category>www.typo3.org</category>
  <pubDate>Sat, 25 Feb 2012 21:30:00 +0100</pubDate>
</item>
```

We want to translate this to the following HTML

```
<div>
  <h1>FLOW3 1.0.3 has been released</h1>
  <p>Tags:
    <span class="category">Development</span>,
    <span class="category">FLOW3</span>,
    <span class="category">www.typo3.org</span>
  </p>
  <p>
    FLOW3 1.0.3, the third patch release of the PHP application framework has been released.
  </p>
  <p>
    <a href="http://news.typo3.org/news/article/flow3-103-has-been-released/">
    Read more...
  </a>
  </p>
</div>
```

Several things need to be considered. First the easy ones: The <item>, <title> and <description> tags need to be transformed to , <h1> and <p> respectively. That shouldn't be too hard. The <link> tag needs to be transformed to an <a> tag where the href attribute has to be set to the former tag's content. The content for the <a> tag needs to be set to “Read more”. Finally, there are several <category> tags that need to be collected within one <p> and transformed to tags with a class “category” assigned. Let's see what TSRef and parseFunc have to offer for this scenario.

parseFunc's “externalBlocks” property comes to our help. In tt_content, “externalBlocks” is used to pre-split bodytext content and parse <table> and <blockquote> tags and their according children. In our case, we can use it to replace the incoming <item> tags and pass the content once again into parseFunc to do the <category> collection and process the <link> tag.

The next step shows you the finished setup:

```
page.10 = XPATH
page.10 {

  # first we set the source to the newsfeed url
  source = http://news.typo3.org/rss.xml

  # each news entry is wrapped in <item> tags, fetch them with XPATH expression
  expression = //item

  # return the <item>s as XML which is going to be formatted with parseFunc later on
  return = xml

  # configure the resultObj
  resultObj {

    cObjNum = 1

    1.current = 1
    1.parseFunc {

      # use externalBlocks to select the <item> tags
      externalBlocks = item
      externalBlocks.item {

        # and send their content once more into parsFunc
        callRecursive  = 1
        # take out <item> tag
        callRecursive.dontWrapSelf = 1

        # use stdWrap to wrap with <div>
        stdWrap {

          wrap = <div> | </div>

          # and now load a COA to work on the rest of the XML content
          cObject = COA
          cObject {

            # get the current XML data first
            5 = LOAD_REGISTER
            5.item.data = current:1

            # and now use some XPATH cobj to select the content; <title> first
            10 = XPATH
            10 {
              # item register from .5
              source.data = register:item
              return = string
              expression = //title
              resultObj {
                cObjNum = 1
                1.wrap = <h1>|</h1>
                1.current = 1
              }
            }

            # <category> collection next
            15 < .10
            15 {
              expression = //category
```

```
                resultObj {
                  # use option split, so the last <category> doesn't get a ,
                  cObjNum = |*|1|*|2

                  1.wrap >
                  1.noTrimWrap = |<span class="category">|</span>, |

                  2.current = 1
                  2.wrap = <span class="category">|</span>

                  stdWrap.noTrimWrap = |<p>Tags: |</p>|
                }
              }

              # next select the <description> and wrap in <p>
              20 < .10
              20 {
                expression = //description
                resultObj.1.wrap = <p>|</p>
              }

              # and finally select the <link> and wrap this in an <a> tag
              30 < .10
              30 {
                expression = //link
                resultObj.1.wrap = <p><a href="|">Read more...</a></p>
              }
            }
          }
        }
      }
    }
  }
}
```

Admittedly, this is a bit of TypoScript ;) On the other hand, it only uses standard functionality and at the same time demonstrates how you can “chain” XPATH objects to flexibly work on your XML data.

The transformation could have been achieved much simpler using an XSL stylesheet. This is precisely what the XSLT content object is all about. Check it out in TER, you'll find a tutorial very similar to this one where the transformation is done with an XSL stylesheet.

FLUIDTEMPLATE, XPATH & XSLT

This tutorial will show you how to display a TEI encoded XML file on your website using the XPATH content object and a FLUIDTEMPLATE. You will learn how to select parts of the XML structure with XPATH and assign them as variables to your FLUIDTEMPLATE. Further you will see how the Xpath view helper for Fluid works. And finally we will have a quick look at the extension XSLT content object that can be used to transform large chunks of XML for display on your TYPO3 website. Before we start, please make sure that you have installed Extbase, Fluid and cobj_xslt.

As an example text we would like to display Edgar Allan Poe's "Tales" on our website. The book is offered as a TEI encoded XML file at http://docsouth.unc.edu/southlit/poe/poe.xml .

What we want to achieve:

- Use a FLUIDTEMPLATE as basis for the FE output
- Display the title and subtitle of the book and a table of contents
- Link the table of contents to the tales; if the user clicks on a story, the selected story should be displayed with a backlink to the table of contents
- Transform the selected story's XML text to HTML using a XSL transformation

Let's have a look at the XML structure and consider some basic things:

- We can get the title and author from the data inside the <titlePage> tag. This is basically static information that does not change on user interaction
- We can generate the table of contents from the list items inside <div1 type="contents">. But we need to find a mechanism that a) generates links around the items and b) provides targets for the links so that we can display the according tale to the toc item that was clicked (basically the same as a "single view")
- So basically we have a "listview" (author, title, table of contents) and a "singleview" (author, title, single story). This should be doable with a combination of a <f:if> Fluid condition and a TypoScript condition. As a trigger for list- and singleview we will use a custom link parameter: &tale=n.
- In sum we will use four different Fluid variables: {author}, {title}, {toc} and {tale}

Let's start with a basic FLUIDTEMPLATE. Save the following to fileadmin/templates/Poe.html:

```
<hgroup>
  <h1>{title}</h1>
  <h2>{author}</h2>
</hgroup>
<f:if condition="{tale}">
  <f:then>
  </f:then>
  <f:else>
  </f:else>
</f:if>
```

Next we construct a basic XPATH cObject:

```
lib.xpath = XPATH
lib.xpath {
  # fetch the source
  source = http://docsouth.unc.edu/southlit/poe/poe.xml

  # set return format
  return = string

  # result handling
  resultObj {
    cObjNum = 1
    1.current = 1
  }
}
```

As you can see we defined it as a TS library. That saves us some typing when assigning the {author} and {title} variables to the FLUIDTEMPLATE afterwards. We can just copy the library into the variable declaration and change the XPATH expression to match the values we want:

```
page.10 = FLUIDTEMPLATE
page.10 {

  file = fileadmin/templates/Poe.html

  variables {

    title < lib.xpath
    title.expression = /TEI.2/text/front/titlePage/docTitle/titlePart

    author < lib.xpath
    author.expression = /TEI.2/text/front/titlePage/docAuthor
  }
}
```

If we now reload the frontend, we can already see the correct title and author in the <hgroup> element:

Note: Since we matched atomic node values (the contents of the <titlePart> and <docAuthor> tags), we set the return format of our XPATH library to string. We have to remember that when we use it for other purposes (like getting the {tale} variable later on).

Since there is no {tale} variable yet, the condition in our FLUIDTEMPLATE evaluates to false and no other content is generated. Time for a change! Next up is the {toc}. The toc will consist of several elements that we will wrap in an unordered list.

Having Fluid at our hands, the easiest mechanism for that would be to use a <f:for> loop. But now we hit one of the rare limitations of TypoScript... it's only possible to pass on string values and not arrays to the FLUIDTEMPLATE (if you're interested further in that topic, have a read of the following ML thread: http://lists.typo3.org/pipermail/typo3-dev/2011-January/042417.html).

For this reason version 1.2.0 of cobj_xpath provides a Xpath query view helper that can be used inside Fluid templates to execute queries. The view helper passes back the "raw" result. This makes it possible to get arrays or other multi value results directly within Fluid.

To use the view helper, we now register a namespace at the beginning of our template:

```
{namespace xpath = Tx_CobjXpath_ViewHelpers}
```

Next we construct a <f:for> loop in the <f:else> branch and set up the Xpath query:

```
<f:else>
  <section id="toc">
    <h3>Select your tale:</h3>
    <ul>
      <f:for each="{xpath:query(source: 'fileadmin/xpath/poe.xml', expression: '//div1[@type=\'contents\']/list/item', return: 'stri
        <li>
          <f:link.action additionalParams="{tale : key.cycle}">{item}</f:link.action>
        </li>
      </f:for>
    </ul>
  </section>
</f:else>
```

The view helper takes three attributes: source (your XML source), expression (your XPATH expression) and return (the return format keyword). The result of the above call will be an array of strings with the atomic nodes of the <item> tags that will be passed on to the <f:for> loop. Notice how we use the <f:for> iterator and cycle variable in <f:link> to append the &tale parameter with incremented numbers. This will come in handy in our last step, the {tale} variable. But first have a look at the result so far:

{tale} will be filled with a whole story depending on which toc-link the user has clicked. Since we don't know it in the beginning, it makes sense to only fill this variable when a link was clicked and the &tale parameter is filled. A classic case for a TypoScript condition:

```
[globalVar = GP : tale >= 1]
page.10.variables {

  tale = XSLT
  tale {
    source.cObject < lib.xpath
    source.cObject {
      expression = /TEI.2/text/body/div1/div2[{GP : tale}]
      expression.insertData = 1
      return = xml
    }
    transformations.1.stylesheet = fileadmin/xpath/poe.xsl
  }

}
[GLOBAL]
```

As you can see, we load the tale variable with a XSLT content object. This makes sense because the result will be a large chunk of XML and the easiest way to transform this is to employ a XSL stylesheet. Of course we could also use TypoScript's parseFunc or a PHP userFunc to parse the XML and get to the same result but let's stay in the world of XML technologies for this tutorial ;)

The source property of our XSLT object is just the same as for the other variables, so we can use our XPATH library. We use stdWrap on the XPATH expression to dynamically insert the value of &tale parameter from GET/POST. Because the number of the &tale parameter matches the amount of stories in the XML file we get the correct <div2> belonging to the respective item in the {toc}.

Notice that we set the return format to "xml" this time.

Next we need a XSL stylesheet for the transformation:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" encoding="utf-8" indent="yes" omit-xml-declaration="yes"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="head">
    <h3><xsl:value-of select="."/></h3>
  </xsl:template>

  <xsl:template match="p">
    <p><xsl:apply-templates/></p>
  </xsl:template>

  <xsl:template match="epigraph">
    <blockquote><xsl:apply-templates/></blockquote>
  </xsl:template>

  <xsl:template match="lg">
    <p><xsl:apply-templates/></p>
  </xsl:template>

  <xsl:template match="l">
    <xsl:apply-templates/><br/>
  </xsl:template>

  <xsl:template match="foreign">
    <span><xsl:apply-templates/></span>
  </xsl:template>
```

```
  <xsl:template match="hi[@rend='italics']">
    <em><xsl:apply-templates/></em>
  </xsl:template>

  <xsl:template match="pb"/>

  <xsl:template match="table">
    <table><xsl:apply-templates/></table>
  </xsl:template>

  <xsl:template match="row">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

  <xsl:template match="cell">
    <td><xsl:apply-templates/></td>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet transforms each XML tag in the document to an appropriate HTML equivalent. We can now insert the {tale} variable in our FLUIDTEMPLATE:

```
<f:then>
  <p><f:link.page>[Table of Contents]</f:link.page></p>
  <section id="tale">
    <f:format.html parseFuncTSPath="">{tale}</f:format.html>
  </section>
</f:then>
```

One important thing to remember is that Fluid sends it's output through htmlSpecialChars on default unless we format the output with <f:format.html> and parseFunc. But in our case we don't want that since the formating has already taken place in the XSLT object. The solution is to leave the parseFuncTSPath empty. In TYPO3 4.6 and higher you can use the <f:format.raw> view helper alternatively. And thats the output if the first link of the toc is clicked:

To get the complete picture, here is the the full TS setup:

```
lib.xpath = XPATH
lib.xpath {
  #source = http://docsouth.unc.edu/southlit/poe/poe.xml
  source = fileadmin/xpath/poe.xml
  return = string
  resultObj {
    cObjNum = 1
    1.current = 1
  }
}

page.10 = FLUIDTEMPLATE
page.10 {
```

```
  file = fileadmin/xpath/Poe.html
  variables {

    title < lib.xpath
    title.expression = /TEI.2/text/front/titlePage/docTitle/titlePart

    author < lib.xpath
    author.expression = /TEI.2/text/front/titlePage/docAuthor
  }
}

[globalVar = GP : tale >= 1]
page.10.variables {

  tale = XSLT
  tale {
    source.cObject < lib.xpath
    source.cObject {
      expression = /TEI.2/text/body/div1/div2[{GP : tale}]
      expression.insertData = 1
      return = xml
    }
    transformations.1.stylesheet = fileadmin/xpath/poe.xsl
  }

}
[global]
```

And the FLUIDTEMPLATE:

```
{namespace xpath = Tx_CobjXpath_ViewHelpers}

<hgroup>
   <h1>{title}</h1>
   <h2>{author}</h2>
</hgroup>

<f:if condition="{tale}">
   <f:then>
      <p><f:link.page>[Table of Contents]</f:link.page></p>
      <section id="tale">
         <f:format.html parseFuncTSPath="">{tale}</f:format.html>
      </section>
   </f:then>
   <f:else>
      <section id="toc">
         <h3>Select your tale:</h3>
         <ul>
         <f:for each="{xpath:query(source: 'fileadmin/xpath/poe.xml', expression: '//div1[@type=\'contents\']/list/item', return: 'stri
            <li>
               <f:link.action additionalParams="{tale : key.cycle}">{item}</f:link.action>
            </li>
         </f:for>
         </ul>
       </section>
   </f:else>
</f:if>
```

<xpath> TypoTag

From a developers point of view using the XPATH content object in a TypoScript template or in a FLUIDTEMPLATE is perfectly ok. But imagine you have some power users that want to display values from XML files they upload themselves. How could the cObject be made “reusable” when we have to deal with XML a lot? Writing a small extension comes to mind immediately. Or maybe introducing a new content element. In this tutorial, we will look at another possibility that is quite reusable, flexible and convenient: a <xpath> TypoTag. It will look like this:

This is the source when the RTE is disabled:

```
<xpath expression="//item" return="string">fileadmin/xpath/poe.xml</xpath>
```

Optionally for our editors we could provide a userElement:

The obvious advantage of a TypoTag in comparison to the other approaches is that it can be used everywhere in the system. You could also use it in a news record or an address element. Only an input field is needed that is treated with the good old lib.parseFunc/lib.parseFunc_RTE. But first things first. Lets configure the RTE with PageTSConfig for the XPATH custom tag:

```
RTE.default {

  showButtons := addToList(user)
  hideButtons := removeFromList(user)

  userElements {
    747 = XML Functions
    747 {
      10 = XPATH
      10.description = Executes a XPath query
      10.mode = wrap
      10.content = <xpath>|</xpath>

      20 = XSLT
      20.description = Executes a XSLT transformation
      20.mode = wrap
      20.content = <xslt>|</xslt>
    }
  }

  proc {
    allowTagsOutside := addToList(xpath)
    allowTags := addToList(xpath)
    entryHTMLparser_db {
      htmlSpecialChars = -1
```

```
        allowTags := addToList(xpath)
      }
    }

}
```

We add the custom <xpath> tag to the various allowedTag lists in the default configuration of the RTE. This makes it possible to enter the tag directly without switching off the editor. The configuration of a XML section in the userElements is optional and included here just for completeness. Notice: If you use this, you will have to implement the parsing of the custom tag slightly different than shown below, because its not possible to set tag attributes in the userElements dialogue.

Next we need to configure lib.parseFunc and lib.parseFunc_RTE for FE rendering of our tag:

```
lib.parseFunc {
  allowTags := addToList(xpath)
}

lib.parseFunc_RTE {
  allowTags := addToList(xpath)
}

# add typotag to parseFunc
lib.parseFunc.tags.xpath = XPATH
lib.parseFunc.tags.xpath {

  # tag is breaking up nonTypoTag content, content after must be re-wrapped
  breakoutTypoTagContent = 1

  # strip new lines before and after the tag
  stripNL = 1

  # get current content of tag as source (either XML or a path)
  source.data = current : 1

  # get the Xpath expression from the expression attribute of the tag
  expression.data = parameters : expression

  # get the return format from the format attribute of the tag
  return.data = parameters : return

  # configuration of the result
  resultObj = 1
  resultObj.cObjNum = 1
  resultObj 1.current = 1
}

lib.parseFunc_RTE.tags.xpath < lib.parseFunc.tags.xpath
```

First we added the <xpath> tag to the allowTags lists of both parsing libraries. Then we configured the tag itself. Notice that its important to set the breakoutTypoTagContent property, otherwise you will have <p>s wrapped around your result. Another thing to remember is that it is possible to get the attribute values of custom tags with getText from the $cobj->parameters array. We can set them directly in the corresponding properties of the XPATH content object by using stdWrap. That's it. Now you can enter XPATH queries in the RTE and display the results on your website.

All that is left is to improve the display of the tag in the RTE like in the screenshot above. This is of course optional. For the example above we inserted the following CSS rule in a custom RTE stylesheet:

```
xpath:before {
  content: "XPATH ["attr(expression)"] ["attr(return)"] :";
  display: inline-block;
```

```
  padding: 0 0.5em 0 0;
  font-family: monospace;
  font-weight: bold;
}
```

The RTE normally will not display any tag attributes. But in our case it can be helpful to see which expression is set. This can be achieved with pure CSS using the :before pseudo-selector and the content property in combination with CSS's attr() function. Nice :)

Known problems

You can report bugs at our TYPO3 Forge bugtracker .

To-Do list

Feature requests on the TYPO3 Forge are always welcome.

ChangeLog

Version	Changes
1.3.0	• Version compatibility set to 4.5.0-6.1.99 • ReST based manual
1.2.0	• New XPATH view helper for Fluid templates • New TypoScript property implodeResult • New tutorial about XPATH, FLUIDTEMPLATE and XSLT • New tutorial about <xpath> TypoTag
1.1.1	• Loading XML files from a path could fail sometimes
1.1.0	• source.url property fused into parent property source
1.0.0	• First public version