XMLTreeWalker

Mario Theodoridis <`mario@schmut.com`>

	Revision History
Revision 1.0	12-30-2006
	Initial Draft

Table of Contents

Purpose

This document describes how to use the XmlTreeWalker product. XmlTreeWalker works in conjunction with expat. It can be used for filtering and content modification of XML documents by employing various Operators and having them activate based on specified Filter rules. Uses include content extraction, suppression and rewriting.

Status

This is an initial release. It is also my first distribution ever, so it's likely to have various issues.

Requirements

- python 2.4
- expat

Usage

The following example takes an xhtml document and parses the table of contents. The PrintOperator and the DocbookTocOperator both filter the same part of the tree. This part is outlined below.

```
<html>
    <body>
        <div class="article">
            <div class="toc">
                <dl>
                    parsed content including the dl tag
                </dl>
            </div>
        </div>
    </body>
</html>
```

The PrintOperator reprints everything except the parsed content. The DocbookTocOperator only deals with the parsed content and extracts out a

```
<dl>
    <dt>
        <a href="url">
            name
        </a>
    </dt>
    <dd>
        <dl>
            <dt>
                ....
```

type definition list and returns a recursive list of TocItems representing the TOC data parsed.

The following is the code that accomplishes that.

```
import codecs
# xmltreewalker uses unicode so we need to decode the utf-8
in = codecs.open(path, 'rb', 'utf-8')
orig = in.read()
in.close()

# setup the filter for toc processing
flt = Filter.getInstance([['html'], ['body'],
                         ['div', {'class': 'article'}],
                         ['div', {'class':'toc'}],
                         ['dl'],
                         ])
dto = DocbookTocOperator(False, [flt])

# this filter allows everything but the TOC
flt2 = Filter.getInstance([['html'], ['body'],
                          ['div', {'class': 'article'}],
                          ['div', {'class':'toc'}],
                          ])

# the output of the PrintOperator will be written to a file like object
# using data.write(content). For larger datasets we could just as well use the
# file opened below and write right to that.
data = StringIO()

# set up the PrintOperator to print unless the filter is met
# the filter list is subsequently specified along with where the
# data goes to.
pto = PrintOperator(True, [flt2], data)])

# this sets up the treewalker with it's operators
wlk = XMLTreeWalker([dto, pto])

# this makes things happen
wlk.parse(orig)

# retrieve the parsed data from the PrintOperator
got = data.getvalue()
data.close()

# write it back remembering that it is unicode and needs to
# be re-encoded into utf-8
output = codecs.open(outPath, 'w', 'utf-8')
output.write(got)
output.close()
```

```
# This prints a pretty version of the TOC
print >>> sys.stdout(dto.rootItem.prettyPrint())
```

A pretty printed version of that could look something like:

```
URL: None - Name: Root - Children:
    URL: #d0e24 - Name: Purpose
    URL: #d0e29 - Name: Status
    URL: #d0e34 - Name: Requirements
    URL: #d0e44 - Name: Usage
```

Details

XMLTreeWalker

XMLTreeWalker registers its handlers with expat and dispatches the calls to all active operators in the order in which they were specified in the list. Each operator returns a Boolean indicating whether to continue to process the other operators (`True`) or to abort calling the other operators (`False`). This is useful for masking. One can place an operator to remove data first in the list and activate it as needed. This way when it's active it can terminate processing for all subsequent operators, effectively masking sections of a document out.

Operators

Operators act on the data they get when they're active. Whether they are or not depends on their filters and the initial state. The initial state is specified in the constructor. If `isActive` is `True` the operator starts out active and processes data until one of the filters is met. At that time the state is toggled to inactive until the filter condition no longer applies. Starting an operator with `isActive` set to `False` accomplishes the exact opposite.

Filters

Multiple filters can be specified for an operator to activate or deactivate. They are specified as a list and are processed in that order. When any of the filters match the `isActive` state of the operator is toggled.

Filter specification always starts at the root node of the document. They may be specified as a series of `TagRequirements` which include the tag name and a dictionary of property value pairs. The tag name may be set to `None` in which case any tag will match. The properties may be specified along with values or with their value set to `None` for any. For a `TagRequirement` to be met, all specified properties must match unless they're set to `None`.