Thinking XML: Serving up WordNet as XML

Build the basic WordNet/XML facilities into a Web server framework

Uche Ogbuji

August 30, 2005

A few articles back, Uche Ogbuji discussed WordNet 2.0, a Princeton University project that aims to build a database of English words and lexical relationships between them. He showed how to extract XML serializations from the word database. In this article he continues the exploration, demonstrating code to serve up these WordNet/XML documents over Web protocols and showing you how to access these from XSLT.

View more content in this series

XML is SGML on the Web, and it is a rare XML project that doesn't find a connection to the Web in one way or another. A little while ago in this column, I wrote "Querying WordNet as XML," which examined WordNet 2.0 and how it can be used with XML and RDF technology. If you haven't read that article I suggest you review it before continuing with this one. Here, I build on the basic code to extract the lexical information from WordNet in XML form. I show how to serve up word information on the Web in HTML and XML forms.

The Web server code

I use Python and XSLT for the code in this article, although you can certainly translate the concepts into most other programming languages. I try to explain the Python code so that you need no more than a basic understanding of the language to follow along. Python supports many ways of writing Web applications, ranging from the super simple BaseHTTPServer module in the standard library to third-party large scale systems such as Zope. In between, you'll find numerous choices for Python Web frameworks. One very popular option, and my personal favorite, is CherryPy (see Resources). It is simple and very well suited to Python idioms. Listing 1 (wnserver.py) is a program that uses CherryPy to serve up information about words based on a very simple URL scheme.

Listing 1. CherryPy server code to serve up WordNet information (wnserver.py)

```python
import cherrypy
from picket import Picket, PicketFilter
from wnxmllib import *
```
class root:
    _cpFilterList = [ PicketFilter(defaultStylesheet="viewword.xslt") ]

class wordform_handler:
    def __init__(self, applyxslt=False):
        self.applyxslt = applyxslt
    return

    @cherrypy.expose
    def default(self, word):
        synsets = serialized_synsets_for_word(word)
        result = ''.join(synsets) # Concatenate strings in result list
        # Wrap up the XML fragments into a full document
        wordxml = '<word-senses text="'+word+'">'+result+'</word-senses>'
        if self.applyxslt:
            picket = Picket()
            picket.document = wordxml
            return picket # apply the XSLT and return the result
        return wordxml

class pointer_handler:
    @cherrypy.expose
    def default(self, pos, target):
        synset = getSynset(pos, int(target))
        synsetxml = serialize_synset(synset)
        picket = Picket()
        picket.document = synsetxml
        return picket # apply the XSLT and return the result

cherrypy.root = root()
cherrypy.root.view = wordform_handler(applyxslt=True)
cherrypy.root.raw = wordform_handler()
cherrypy.root.pointer = pointer_handler()
# Disable debugging messages in Web responses
cherrypy.config.update({'logDebugInfoFilter.on': False})
cherrypy.server.start()

The code is very simple, but it does an impressive amount of work, as you shall see. It requires Python 2.4 at a minimum because it uses decorators, a new feature in that version. First, take a look at the prerequisites. One of the imports is from wnxxmllib.py, which is basically the code I presented in the last column on WordNet but bundled into one file and updated a bit to use a simpler 4Suite API. All the code except for the following prerequisites is bundled in the download for this article.

- **CherryPy** (import cherrypy): Minimum version CherryPy 2.1, currently in beta.
- **Picket** (from picket import Picket, PicketFilter): This is a straightforward tool that simplifies the API for invoking XSLT from a CherryPy server. Minimum version Picket 0.5.
- **4Suite** (imported from wnxxmllib.py): The core XML library. Minimum version 4Suite 1.0b1.

The listing wnserver.py contains three classes, each of which handles a different part of the URL space for the site. This is controlled by the code at the bottom of the file. The near empty class root is really a placeholder for the root URL of your site. On a testing machine, this will
be something like http://localhost:8080/. All the real action starts with more specific URLs. The following list describes the mapping from URL stems to classes:

- URLs that start with http://localhost:8080/view/ are handled by an instance of wordform_handler that applies XSLT to WordNet XML results. This is for general Web browser viewing.
- URLs that start with http://localhost:8080/raw/ are handled by an instance of wordform_handler that passes WordNet XML results directly back. This is for use by applications that can process the XML.
- URLs that start with http://localhost:8080/pointer/ are handled by an instance of the pointer_handler class. This is for interpreting links from one sense of a word to another, as discussed near the end of my last WordNet article. It also uses XSLT to turn XML results into HTML for browsers.

The root class sets up a filter, which is a CherryPy construct that performs some action on data that comes in from the HTTP request, and again on data that goes out as the HTTP response. With this particular filter, PicketFilter, you can apply an XSLT transform on the output, if you need to. CherryPy is designed so that this filter on the root handler instance also applies to other handler objects on the chain from cherrypy.root. I'll discuss the filter's action more in a moment.

The wordform_handler class has an initializer and keeps track of the parameter passed in to determine whether to apply XSLT to XML results. The special method default is marked as a CherryPy handler function by the @cherrypy.expose decorator. It takes a word string as a parameter from the URL and calls wnxmlibserialized_synsets_for_word to get the resulting XML chunk. It then uses string concatenation to create a complete, well-formed XML document from it. Be aware that simple string concatenation can be a dangerous way to construct XML documents in general (see Resources for an article that expands on this point). I use this approach only because I have a very clear understanding of the data in WordNet 2.0 and an expert understanding of XML serialization issues. If you're not absolutely sure you know what you're doing, avoid such a shortcut and use a proper XML output toolkit (such as 4Suite's MarkupWriter).

Make the transform

If wordform_handler is set up to apply XSLT, then it creates a Picket object. Such an object is caught by the filter I discussed above and used to apply an XSLT transform to generate the actual output. The source document is the XML constructed from the WordNet synonym set information. Listing 2 (viewword.xslt) is the XSLT transform.

Listing 2. XSLT to render WordNet XML as HTML (viewword.xslt)

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html"/>

    <xsl:template match="/">
        <html>
            <head>
                <title><xsl:value-of select="word-senses/@text"/></title>
            </head>
```

Thinking XML: Serving up WordNet as XML
This XSLT takes the sense information for a word, and displays the gloss (brief definition) for each sense. It also renders a set of links to other senses based on pointers in WordNet. Seeing the result of this XSLT should help make its operation clear. Figure 1 is what you see when the Web
server in Listing 1 is running and you browse to the page for the word "code". The Web browser I used for the screenshot is Mozilla Firefox. As you can see in the image, the URL I visited was http://localhost:8080/view/code.

Figure 1. Browser view of the page for the word "code"

When you click on a link you go to a pointer, which refers to a synset, rather than a word. In this case, I chose to give a simpler display that includes all the word forms for the synset in square brackets, the gloss, and then any further links to other synsets. I designed the XSLT in Listing 2 so that it can be used to display full word form information as well as information for an individual synset. The code that handles pointer URLs in the Web server is the pointer_handler.default class in Listing 1. As you can see, it takes two values from the URL. A URL such as http://localhost:8080/pointer/noun/5955443 becomes a call to pointer_handler.default, with part of speech noun and WordNet offset 5955443. One very important point is that I render the word forms that appear before each gloss as links, which means that you can navigate the depth and breadth of the WordNet database entirely by clicking through the many relationships between English words.

Just the XML, ma'am
This WordNet Web server example also allows you to grab the raw XML for a word using a URL of the form http://localhost:8080/raw/... I think it is very important for Web applications that use XML to expose this XML directly over the Web, as well as HTML prepared for the Web
browser. This allows you or others to easily build other applications that extend the functionality by processing this XML directly, rather than screen-scraping from HTML. As an example of how useful this can be, Listing 3 demonstrates simple XSLT code that queries raw XML from the server and then processes it. It takes a list of words, and outputs the same list updated with one of the possible definitions for each word.

Listing 3. Example XSLT based on query from the main WordNet server

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="word-list">
        <xsl:copy>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="word">
        <!-- The word form to look up is the element content -->
        <xsl:variable name="wordform" select="."/>
        <!-- Use the word form to construct the query URL -->
        <xsl:variable name="wordnet-url" select="concat('http://localhost:8080/raw/', $wordform)"/>
        <!-- Query the WordNet server, retrieving an XML document -->
        <xsl:variable name="wordnet-info" select="document($wordnet-url)"/>
        <!-- Grab the first gloss from the retrieved WordNet document -->
        <xsl:variable name="gloss" select="$wordnet-info//gloss[1]"/>
        <xsl:copy>
            <form><xsl:value-of select="."/></form>
            <sample-gloss><xsl:value-of select="$gloss"/></sample-gloss>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

Running this XSLT against the test document in Listing 4 (wordlist.xml) yields the result in Listing 5.

Listing 4. Example word list document for the transform in Listing 3 (wordlist.xml)

```xml
<word-list>
    <word>animal</word>
    <word>vegetable</word>
    <word>mineral</word>
</word-list>
```

Listing 5. Result from applying Listing 3 XSLT against Listing 4 test XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<word-list>
    <word><form>animal</form><sample-gloss>a living organism characterized by voluntary movement</sample-gloss></word>
    <word><form>vegetable</form><sample-gloss>edible seeds or roots or stems or leaves or bulbs or tubers or nonsweet fruits of any of numerous herbaceous plant</sample-gloss></word>
    <word><form>mineral</form><sample-gloss>solid homogeneous inorganic substances occurring in nature having a definite chemical composition</sample-gloss></word>
</word-list>
```
I added white space to Listing 5 for formatting purposes. The content of the sample-gloss comes from the dynamic query to the WordNet server. This is just one more example of the power of WordNet and XML combined.

**Wrap-up**

I plan to continue building on practical uses of WordNet made available in XML form. Future topics include exposing WordNet as an RDF database and using it to enhance searching. Meanwhile, do please post your thoughts on the Thinking XML discussion forum.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code for WordNet as XML</td>
<td>x-think33-code.zip</td>
<td>6 KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2005  

**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)