Processing WSDL in Python

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

Tutorial purpose

Web Services Description Language (WDSL), originally developed by IBM, Microsoft, and others, is an XML format for technical description of Web services. In this tutorial, Mike Olson and Uche Ogbuji introduce WSDL4Py, an open-source Python library for WSDL 1.1 hosted by IBM developerWork's open-source zone. Usage of the library is explained, as well as discussion of its development.

About the authors

Mike Olson and Uche Ogbuji are co-founders of and principal consultants at Fourthought Inc. where they develop the open-source tools 4Suite and 4Suite Server, and provide commercial consulting, training and 4Suite customization for clients working with XML and Web services. They reside in Boulder, Colorado.

Section 2. Getting started

Setting up WSDL4Py

The prerequisites for using the WSDL4Py library and the examples in this tutorial are as follows:

* A working knowledge of the Python programming language, and WSDL.
* Python: version 2.0 or more recent is highly recommended, but 1.5.2 will work as well.
* 4Suite: version 0.11 or more recent.

The software should work under UNIX and Windows. Unfortunately, Macintosh users must wait for someone to port the XPath library of 4Suite to that platform, or wait for `4Suite-0.11.1` that will have a pure Python XPath parser

Installation is quite simple. Download and unpack the archive, then run the command `python setup.py install`, which will copy the library to the appropriate location in your `PYTHONPATH`. To test the installation, run the following command:

```
$ python -c "import wsdllib"
```

If you see no error messages, you're ready to go.

Section 3. Reading and examining descriptions

Reading WSDL descriptions

WSDL provides the means of giving descriptions to Web services. These descriptions are normally stored in a UDDI registry so that they can be looked up by other services. For the sake of simplicity in our examples, we will consider a standalone implementation without the use of a UDDI registry.

The first use of the WSDL library implemented in our project is to read and parse the contents of the WSDL description for a service.

Reading and examining descriptions

You can read in WSDL from external sources in stream, string, or content retrieved from URI.

```
#import the library
import wsdllib

#Read the WSDL source directly from URI
wsdl = wsdllib.ReadFromUri('http://uche.ogbuji.net/articles/wsdl/endorsementservice.xml')

#Open the URI directly as a stream, and read the WSDL source therefrom
import urllib
stream = urllib.urlopen('http://uche.ogbuji.net/articles/wsdl/endorsementservice.xml')
wsdl = wsdllib.ReadFromStream(stream)
```

`ReadFromUri` and `ReadFromStream` are factory functions that return a new object of class `Wsdl`. This class is the core of the library: each instance represents a separate WSDL description and provides methods for examining and modifying the description.

If you have the WSDL information in string form, the factory function `ReadFromString` is similar to the above.

Wsdl instances

Each instance of the `Wsdl` class contains the components of the WSDL description as illustrated in Figure 1 shown in the next panel. The messages, portTypes, bindings and services are all stored as instance variables in the form of objects that act like Python dictionaries. The key of each dictionary entry corresponds to the name attribute in the XML source. The value is an instance of class `Message`, `PortType`, `Binding`, or `Service` respectively. These classes in turn encapsulate the data specific to each element.

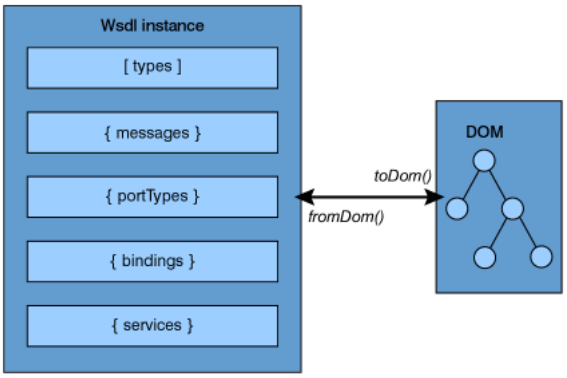
Diagram of a Wsdl instance

The type information is stored as an instance of class Types, which stores a Python list of the XML schema fragments included in the description.

This leads to very "pythonic" access to the guts of a WSDL description. For instance, the program in Listing 1 displays the names of all the messages and message parts for a WSDL

description at a URL given on the command line.

Figure 1: An example of a WSDL instance



Listing 1: Display a summary of messages and message parts from a WSDL description

```
import sys
import wsdllib

def SummarizeMessages(url):
    #Read in the WSDL
    wsdl = wsdllib.ReadFromUri(url)
    #Python idiom for iterating over the key/value pairs in a dict
    for mname, message in wsdl.messages.items():
        #Print out the current message name
        print "Message: %s"%mname
        for pname, part in message.parts.items():
            #Print out the current part name in the current message
            print "\tPart: %s"%pname
    return

if __name__ == "__main__":
    usage = "Usage: %s uri"%sys.argv[0]
    if len(sys.argv) < 2:
        print usage
        sys.exit(-1)
    SummarizeMessages(sys.argv[1])
```

Displaying a summary from a WSDL description

You're probably familiar with the general form of this script (if not, get ye to a Python tutorial). All the action is in the `SummarizeMessages` function. You can see from the sparseness of this function how simple it is to use the WSDL4Py library. Hopefully the in-line comments clearly explain the logic. The result of running this program against an online WSDL document follows:

```
$ python listing2.py http://uche.ogbuji.net/articles/wsdl/endorsementservice.xml
Message: GetEndorsingBoarderRequest
        Part: body
Message: GetEndorsingBoarderResponse
        Part: body
```

Most of the API for examining the details of WSDL follows this simple pattern of accessing

objects representing the element information.

Section 4. Generating and modifying WSDL

Generating and modifying WSDL

The initializer for the `Wsdl` class allows you to create "empty" descriptions to which elements can be added as needed to build the description dynamically. Each of the element classes, such as `Message` or `PortType` provides convenient methods to add sub-elements. The function in Listing 2: A function to create a WSDL object from scratch on page 7 illustrates this dynamic building of descriptions.

Listing 2: A function to create a WSDL object from scratch

```
def GenerateEsWsdl():
    #Create a new "empty" description
    wsdl = wsdllib.Wsdl()

    #Add a message element to the description, by name
    #Also return the new message object
    msg = wsdl.addMessage("GetEndorsingBoarderRequest")

    #Add standard documentation to the message element
    msg.documentation = "Query the pro who endorses the given gear"

    #Add a new part to the message, specifying the
    #message body element it represents
    part = msg.addPart("body", "esxsd:GetEndorsingBoarder")

    return wsdl
```

Creating a WSDL object from scratch

After you make your modifications to the descriptions, WSDL allows you to turn the resulting `Wsdl` instance into an XML DOM document for processing in XML form. For example, the following code would take a Wsdl instance and print its XML serialization to standard output:

```
#wsdl is an object created, say by calling GenerateEsWsdl()
#the toDom() method returns an XML DOM representing the document
doc = wsdl.toDom()
#Import the machinery for printing the XML serialization of DOMs
from xml.dom.ext import PrettyPrint
#Pretty-print the document, and presto: familiar WSDL
PrettyPrint(doc)
```

The program in Listing 3: A complete program for generating and displaying a WSDL document on page 7 illustrates this whole process of generating and printing a description.

Listing 3: A complete program for generating and displaying a WSDL document

```
import wsdllib
from xml.dom.ext import PrettyPrint
```

```
def GenerateEsWsdl():
    wsdl = wsdllib.Wsdl()
    msg = wsdl.addMessage("GetEndorsingBoarderRequest")
    msg.documentation = "Query the pro who endorses the given gear"
    part = msg.addPart("body", "esxsd:GetEndorsingBoarder")
    return wsdl

wsdl = GenerateEsWsdl()
doc = wsdl.toDom()
PrettyPrint(doc)
```

Generating a WSDL document

The following is the result of running the program from Listing 3:

```
$ python listing3.py
<wsdl:definitions xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'>
  <wsdl:message name='GetEndorsingBoarderRequest'>
    <wsdl:documentation>Query the pro who endorses the given gear</wsdl:documentation>
    <wsdl:part name='body' element='esxsd:GetEndorsingBoarder'/>
  </wsdl:message>
</wsdl:definitions>
```

Of course, the result is missing some of the good stuff that goes with WSDL descriptions, such as types, port types, and bindings, but these can be added to the `Wsdl` instance in similar fashion to how we added the elements and parts.

Section 5. Notes from implementation

Notes from implementation

Since WSDL is completely defined by its XML representation, one possible approach to creating a WSDL library would be to mimic the DOM itself. Complications such as adding a path to a message can be avoided by subclassing the DOM with special helper methods for WSDL such as the following code:

```
WSDL_NS = 'http://schemas.xmlsoap.org/wsdl/'
new_part_node = wsdl_doc.createElementNS(WSDL_NS, "wsdl:part")
new_part_node.setAttributeNS(None, "name", "body")
new_part_node.setAttributeNS(None, "element", "esxsd:GetEndorsingBoarder")
message_node.appendChild(new_part_node)
```

However, even then, the resulting interface would be far from the most natural way to view the organization of the WSDL model in Python.

Since most of WSDL consists of named structures, Python's dictionaries are a natural fit for handling them. However, this means a complete break from the DOM for the WSDL model and API.

However, since XML is the actual currency of WSDL descriptions, conversion to and from XML forms is essential. We chose to use DOM as the basis for this conversion since it is so widely implemented and would fit into a large number of toolkits. Serialized XML can easily be obtained from DOM.

For conversion from DOM to WSDL4Py model, standalone XPath expressions (in 4XPath) are used to extract key elements at the top level, and the class for each type of element is invoked for specialized deserialization of each component of the description, using further XPath expressions. This approach is much more readable than direct DOM manipulation, although we do manipulate DOM for things such as iterating over node lists that result from XPath evaluation.

Section 6. Web services easy as py

Web services easy as py

Please feel free to use, comment on, and contribute to WSDL4Py. The project home page includes a newsgroup and public CVS. In general, Python is the perfect language for Web services development. A mailing list for Python Web services activity is in the works, and the number of libraries to choose from is always expanding. See the column " *The Python Web services developer* ," also by the authors of this article, for more information on the topic.

Section 7. resources

Resources

* *Python* is a popular general-purpose application-development language famed for its readability, sensible design and comprehensive standard library.
* There are many online Python introductions and tutorials. IBM developerWorks presents *Python 101* by Evelyn Mitchell.
* There is *Python Programming for Beginners* on Linux Journal.
* The Python documentation comes with a *tutorial* .
* Magnus Lie Hetland has tutorials for *programmers* and *non-programmers* , which have been translated to quite a few languages.
* *Josh Cogliati's tutorial* is also a very gentle introduction.
* *WSDL 1.1* is a W3C Note.
* Other WSDL tools include *IBM's WSDL Toolkit* and *Using SOAP with Tcl* .
* The *inaugural* installment of *The Python Web services developer* listed Python tools for Web service. The *second* and *third* discuss the development of a software repository Web service by example.
* Uche Ogbuji's earlier articles on WSDL include *"Using WSDL in SOAP applications"* , *"Supercharging WSDL with RDF* ," and *"WSDL processing with XSLT* ."

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. Feel free to suggest improvements or other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.