

Tutorial: Introduction to XML messaging

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.

1. Tutorial introduction	2
2. Problem-solving with XML messaging	4
3. The XML messaging listener code	6
4. The XML messaging requestor code	12
5. Two-way processing	14
6. You try it	19
7. Resources and feedback	22

Section 1. Tutorial introduction

Who should take this tutorial?

This tutorial gives a hands-on introduction to the basic building blocks for applications that communicate two ways using Web protocols. If you are working on dynamic Web applications or distributed programming, this tutorial will get you started.

XML is quickly emerging as the standard data format for Internet technology. It is also very popular for the broader range of component technologies. XML is a useful way to introduce structure into the body of HTTP requests. This tutorial gives hands-on examples of XML messaging. It will be useful for anyone contemplating ways of communicating between components using Web protocols.

Navigation

Navigating through the tutorial is easy:

- * Use the Next and Previous buttons to move forward and backward through the tutorial.
 - * When you're finished with a section, select Next section for the next section. Within a section, use the Section menu button to see the contents of that section. You can return to the main menu at any time by clicking the Main menu button.
 - * If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.
-

Prerequisites

You should be familiar with the HTTP protocol. If you aren't, the previous tutorial in this series will provide the necessary background.

For the code examples, you will need some skill in any object-oriented language, such as C++ or Java. The code will itself be in Python, and you will need Python 2.0, which is available for UNIX, Windows, and Macintosh.

Note, however, that you do not need any prior knowledge of Python in order to understand and complete this tutorial. Any other object-oriented language will do.

Getting help and finding out more

For technical questions about the content of this tutorial, contact the author, [Uche Ogbuji](#).

Uche Ogbuji is a computer engineer, co-founder and principal consultant at [Fourthought, Inc](#). He has worked with XML for several years, co-developing [4Suite](#), a library of open-source tools for XML development in [Python](#) and [4Suite Server](#), an open-source, cross-platform XML data server providing standards-based XML solutions. He writes articles on XML for IBM *developerWorks*, LinuxWorld, SunWorld and XML.com. Mr. Ogbuji is a Nigerian immigrant living in Boulder, CO.

Section 2. Problem-solving with XML messaging

Motivation

XML messaging is very useful when you have a system that can do specialized and distributed computing using components that already use XML, or are easily plugged into XML processing systems. This makes available to you all the data-management facilities that have made XML so popular.

This is why XML messaging is becoming such an important part of the emerging field of Web services. So many applications and systems now have XML-based interfaces that XML messaging is often the most straightforward way to expose required functionality in a distributed manner.

The problem

In this tutorial we'll examine a sample application offered as a Web service: an adding machine which takes a series of numbers encoded in XML, sums them up, and returns the result in XML form.

This relatively simple example allows us to focus on the request and response messaging protocol rather than the details of how the request is formulated, or how the service is performed.

This will demonstrate the process of building a request in XML format. For the listener to handle this we have to be able to extract the numbers from the XML format.

The request body

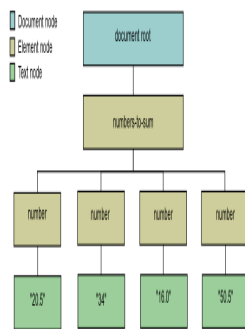
This is the XML request we'll use as our example.

```
<?xml version="1.0"?>
<numbers-to-sum>
  <number>20.5</number>
  <number>34</number>
  <number>16.0</number>
  <number>50.5</number>
</numbers-to-sum>
```

Retrieving the data from XML

The extraction of the numbers from this structure requires some specialized XML processing. Luckily, there are two standards for such XML processing. One is the Simple API for XML (SAX), which treats the XML as a stream of events representing the data. The second is the Document Object Model (DOM), which treats the XML as a tree where each node is a different part of the document.

SAX is generally more efficient for simple processing such as extracting the numbers in our example, but it is a bit more involved to program. We'll use the DOM, since it results in more concise code.



Picturing the XML

To understand the DOM code we'll use, it is best to visualize the XML source as a tree, as illustrated here.

Section 3. The XML messaging listener code

XML listener for summing numbers

This section includes the listener code for our basic XML messaging example.

Setting up the DOM reader

First some imports from the standard library:

```
import string, BaseHTTPServer
```

Now we create a reader object:

```
from xml.dom.ext.reader import PyExpat
```

This import allows us to use the DOM facilities of 4Suite. PyExpat is an XML parser that comes with Python.

```
reader = PyExpat.Reader()
```

This creates a reader object, which can be used and reused for parsing XML documents into a DOM instance resembling the tree in the illustration in the last panel of Section 2.

Defining the handler class

The listing below is a familiar beginning of the definition of an HTTP handler class. If it doesn't seem familiar to you, you may want to break here and review the earlier HTTP tutorial in this series.

```
class XmlSumHttpServer(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        clen = self.headers.getheader('content-length')
        if clen:
            clen = string.atoi(clen)
        else:
            print 'POST ERROR: missing content-length'
            return
        input_body = self.rfile.read(clen)
```

Turning text into a DOM object

This line takes the XML string from the body of the HTTP request and parses it into DOM form using the reader object we created earlier.

```
xmlDoc = reader.fromString(input_body)
```

Gathering all number elements

This line uses a handy DOM method which retrieves all elements with a given name.

```
nums = xmlDoc.getElementsByTagNameNS('', 'number')
```

The first argument is the namespace. If you look back at the request body, you'll notice that we don't use namespaces here, so we pass in an empty string as the first argument.

The second argument is the "local name" of the element, which is the name with the prefix removed. Since we don't use namespaces in this example, there are no prefixes, so we just use the full element name "number". The `nums` variable now references a list of all elements with name "number" in the document.

Preparing to compute the sum

Now we process this list to compute the sum.

```
total = 0
```

The variable `total` will contain the results when we're done and is initialized to 0.

```
for num_elem in nums:
```

Now we set up a loop which iterates over each member of the `nums` list.

Retrieving the first child

Within the body of the loop we will update the running total. `num_elem` is assigned by the for loop to the current `number` element in the list.

```
num_node = num_elem.firstChild
```

The first (and only) child node of the current element is assigned to `num_node`. If you review the illustration of the XML source as a tree (on the last page of Section 2), you'll see that this is the text node representing the string with each number, for instance "20.5".

Accessing a text node's character data

The `data` attribute of a text node gives the string it represents, and we assign this to `num_str`.

```
num_str = num_node.data
```

Updating the total

Next we convert this to a floating-point value using the `string.atof` function, and add the value to the running total.

```
total = total + string.atof(num_str)
```

Python programmers will note that there are many shortcuts that could be employed to simplify the list processing just described -- but this code was designed to walk any OO programmer through the operations needed to process the source XML in careful detail.

Building the output body

Once we have the total, we can quite easily put together the result, which we'll just render as the number we computed inside a single element of name "total". We do this using simple Python string operations, specifically adding three substrings together.

```
output_body = '<?xml version="1.0"?>\n<total>' + str(total) + '</total>'
```

The first substring contains the XML declaration, then the '\n' character which represents an ASCII new-line (as it does in C), then the start tag of the "total" element. The second substring is obtained by converting the `total` floating-point value to string. The third substring consists of the end tag of the "total" element.

Single or double quotes?

You might notice that we use single quotation marks (apostrophes) rather than double quotes for the literal Python strings. Python allows you to use single or double quotes for string delimiters. In this case, the XML string we're specifying `<?xml version="1.0"?>\n<total>` contains double quotes, so we must use single quotes to delimit the Python string itself. It just so happens that XML also allows you to use single or double quotes interchangeably, so this line of Python code could be written:

```
output_body = "<?xml version='1.0'?>\n<total>" + str(total) + "</total>"
```

And it would be legal Python and XML.

An alternative method for constructing XML

Note that concatenating strings together piece by piece is just one way of generating an XML document. We chose it here for simplicity, but it could become cumbersome and error prone if we were rendering more complex output. Using special DOM methods for creating and structuring XML nodes is a cleaner though less efficient way to generate XML in many cases. See Resources to find discussions of DOM that cover these methods.

The output body length

Here we compute the length of the body we just constructed.

```
olen = len(output_body)
```

Once we've constructed the XML string, which will be used for the HTTP response body, we move on to familiar territory. The rest of the code is just as in the echo listener example.

Sending the response

The rest of the method sends the HTTP response headers and body in familiar fashion.

```
self.send_response(200, 'OK')
self.send_header('Content-type', 'text/plain; charset=iso-8859-1')
self.send_header('Content-length', olen)
self.end_headers()
self.wfile.write(output_body)
return
```

Putting the handler to work

The remainder of the listener code is again familiar, setting up a server instance with the handler we have defined, and then listening for an indefinite number of requests.

```
server_address = ('127.0.0.1', 1066)
httpd = BaseHTTPServer.HTTPServer(server_address, XmlSumHttpServer)
print "Now we wait for requests..."
httpd.serve_forever()
```

The complete listener code

Remember that Python is sensitive to indentation, so leave the formatting exactly as you see it here or you might get syntax errors.

```
import string, BaseHTTPServer
from xml.dom.ext.reader import PyExpat
reader = PyExpat.Reader()

class XmlSumHttpServer(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_POST(self):
        clen = self.headers.getheader('content-length')
        if clen:
            clen = string.atoi(clen)
        else:
            print 'POST ERROR: missing content-length'
            return
        input_body = self.rfile.read(clen)

        xmldoc = reader.fromString(input_body)
        nums = xmldoc.getElementsByTagNameNS('', 'number')
        total = 0
        for num_elem in nums:
            num_node = num_elem.firstChild
            num_str = num_node.data
            total = total + string.atof(num_str)

        output_body = '<?xml version="1.0"?>\n<total>' + str(total) + '</total>'
        olen = len(output_body)
        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/plain; charset=iso-8859-1')
        self.send_header('Content-length', str(olen))
        self.end_headers()
        self.wfile.write(output_body)
        return

server_address = ('127.0.0.1', 1066)
httpd = BaseHTTPServer.HTTPServer(server_address, XmlSumHttpServer)
print "Now we wait for a request..."
#httpd.handle_request()
httpd.serve_forever()
```

Section 4. The XML messaging requestor code

XML requestor for summing numbers

Here is code which sends the HTTP requests to sum numbers structured as XML. It starts with the library imports.

```
import httplib, mimetools
```

The request body

The request body is an XML string as described in the request body.

```
body = """<?xml version="1.0"?>
<numbers-to-sum>
  <number>20.5</number>
  <number>34</number>
  <number>16.0</number>
  <number>50.5</number>
</numbers-to-sum>"""
```

The `body` element once again contains the string to be sent in the body of the HTTP request, but this time the string is quite different. It uses a special Python syntax for defining strings whose content covers multiple lines. The triple quotation marks mean that all characters on all following lines are part of the string until another set of triple quotes is encountered. The body is set to the XML code we already presented in the request body.

Other than this, the rest of the requestor code should be familiar.

The complete requestor code

Once again mind the indentation.

```
import httplib, mimetools

body = """<?xml version="1.0"?>
<numbers-to-sum>
  <number>20.5</number>
  <number>34</number>
  <number>16.0</number>
  <number>50.5</number>
</numbers-to-sum>"""

blen = len(body)
requestor = httplib.HTTP('127.0.0.1', 1066)
requestor.putrequest('POST', '/echo-request')
requestor.putheader('Host', '127.0.0.1')
requestor.putheader('Content-Type', 'text/plain; charset="iso-8859-1"')
requestor.putheader('Content-Length', str(blen))
requestor.endheaders()
requestor.send(body)
(status_code, message, reply_headers) = requestor.getreply()
reply_body = requestor.getfile().read()
print status_code
print message
print reply_body
```

Section 5. Two-way processing

Get set

The trick to having two-way conversations between components is to use multitasking.

We'll use Python's support for threads to expand the adding machine components to allow two-way exchanges.

Python's threading is very simple. You simply arrange each path of execution into a single function and ask the threading module that comes in Python's standard library to launch the function as a separate thread.

The code described in this section illustrates a single component code base which combines the listener and requestor functions for the adding machine.

The code

The two-way component code is too large to present in its entirety in these tutorial panels. Please [view the code](#). It is best to open the link in a separate browser window so that you can follow along with the explanations in the following panels.

The requestor section

At the top of the file is the comment `#Requestor code`. This portion is similar to the HTTP requestor code with which you have become familiar.

```
thisport = 1067
otherport = 1066
```

The global variables `thisport` and `otherport` set, respectively, the TCP/IP port for us to use for the listener in this process, and the port where we expect to find the listener in the other process. We used to have port values hard-coded, but we'll need to be able to modify their values in order to have both processes running on the same machine (since you can only have one process to listen on a given port at a time).

The requestor thread function

The `requestor_thread_function` function will be invoked as a separate thread.

```
def requestor_thread_function():
```

It will sit in a loop accepting simple input from the user which will then be sent to the listener in an XML-encoded HTTP request just as in our last example.

The main request loop

`requestor_thread_function` starts out by establishing the loop.

```
while 1:
```

A *while* loop is a Python construct which loops until the condition that follows the `while` keyword is not true. In Python, the number 1 is true, so `while 1` has the effect of repeating forever.

Getting user input interactively

In the body of our loop we start with a request for user input.

```
num_series = raw_input("Please enter a series of numbers, separated by spaces")
```

The `raw_input` function puts out a prompt message, which is the argument passed in, and then waits for the user to type in input, concluding when the user hits "enter." In this case we expect a series of numbers separated by spaces.

Splitting the input line into numbers

This line takes the string entered by the user and using the `string.split` function constructs a list of substrings according to the separating spaces.

```
num_list = string.split(num_series)
```

If the user provided correct input, this should now be a list of individual numbers in string form. So, for instance, if the user input the string "14.75 12.50 36.25 41", at this point we would have a list of the following individual strings: "14.75", "12.50", "36.25", and "41".

Building the request body

Next we build the XML request body, starting with the XML declaration and start tag of the `numbers-to-sum` element.

```
body = '<?xml version="1.0"?>\n<numbers-to-sum>\n'
```

Then we loop over the list of number strings obtained from the user input.

```
for num_str in num_list:
```

Appending an element for each number

For each number we add an element to the XML string by adding the `number` start tag, appending the current number string and finishing with the `number` end tag, then a line feed.

```
body = body + "" + num + "\n"
```

Completing the XML string

Once we have added a `numbers` element for each of the input numbers, the loop terminates and we can close the `numbers-to-sum` element.

```
body = body + "</numbers-to-sum>"
```

The listener code

The rest of the requestor thread function is the familiar code for sending the HTTP request and listening for the response.

Browse the code until you get to the line `#Listener code`.

The listener code hasn't changed much. We still define a `XmlSumHttpServer` class, and now we have placed the code that sets up the HTTP server code into a function (`listener_thread_function`) so we can run it in a separate thread.

The main section

Now that we have set up functions to represent each thread, we need some code to set things up and launch the threads. This forms the remainder of our two-way messaging code.

Look for the section starting with the comment `#Binding it together`.

The Thread module

First we import the `Thread` class from Python's standard `threading` module.

```
from threading import Thread
```

Creating Thread objects

Next we create instances of `Thread` objects from the functions we already defined. `Thread` objects represent the machinery for managing and handling threads.

```
listener_thread = Thread(None, listener_thread_function)
requestor_thread = Thread(None, requestor_thread_function)
```

There are two parameters to the `Thread` initializer. The first in each is *None*, which is a special Python object typically used as a stand-in when one doesn't have a valid object for a particular purpose. It is also often passed into function parameters to mean "I don't have an argument for you here. Just use whatever default was set up."

It is rare that you will need to pass anything except for *None* as the first argument to `Thread`'s initializer. The second argument is the function that the thread invokes when spawned. There are other optional arguments to the initializer which are outside the scope of this tutorial.

Executing the threads

Finally the threads are kicked off. The `start` method on a thread object actually spawns the new thread and begins execution at the function that was passed into the initializer.

```
listener_thread.start()
requestor_thread.start()
```

Section 6. You try it

Copying the code

Copy the [two-way adding-machine code](#) into a file called `twoway-addnum1.py`.

Now copy the code from [this different listing](#) into a file called `twoway-addnum2.py`.

The only difference between these two scripts is that the values of `thisport` and `otherport` are swapped. As we discussed, this allows both scripts to be listening at the same time without their ports clashing.

Running the programs

Open up two command-line consoles.

In the first execute the first script by entering `python twoway-addnum1.py`.

```
[uogbuji@borgia xml-messaging]$ python twoway-addnum1.py
Listening on port 1067
Please enter a series of numbers, separated by spaces:
```

As you can see, the program begins, announces the start of the listener, and then prompts the user for input.

In the second console execute the second script by entering `python twoway-addnum2.py`.

```
[uogbuji@borgia xml-messaging]$ python twoway-addnum2.py
Listening on port 1066
Please enter a series of numbers, separated by spaces:
```

Messages one way

Now in the first console, enter the series of number we've been using as our example. Here is what you should see:

```
[uogbuji@borgia xml-messaging]$ python twoway-addnuml.py
Listening on port 1067
Please enter a series of numbers, separated by spaces: 14.75 12.50 36.25 41
200
OK
```

```
<?xml version="1.0"?>
<total>104.5</total>
Please enter a series of numbers, separated by spaces:
```

It took your input, sent it to the server, and got the result back: "104.5" encoded as XML.

Then it prompted you for input again. This is due to the loop in the requestor thread function.

Messages the other way

Over to the second console. You'll notice that an HTTP status message was scrawled across the console. Ignore this: you are still at the prompt to enter a list of number. So do so.

```
[uogbuji@borgia xml-messaging]$ python twoway-addnuml.py
Listening on port 1066
Please enter a series of numbers, separated by spaces: localhost.localdomain - - [10/Jan/2001 16:39:45] "POS
1 2 3 4 5 6
200
OK

<?xml version="1.0"?>
<total>21.0</total>
Please enter a series of numbers, separated by spaces:
```

And you can see that we got the answer back from the other listener thread.

You can send messages one way or another all you want. When you're done, press CTRL-C in each console to end the listeners.

Troubleshooting

If you run into trouble, here are some tips that might help.

- * If you get any message containing "Syntax Error", you might not have copied over the code listings properly. Mind the indentation, especially, and the ports specified in each file.
- * Make sure you have Python installed properly. Enter `python` and you should be left at a prompt of the form "`>>>`".
- * Make sure you can `ping 127.0.0.1` and not hang or get any error messages.
- * If the programs don't stop when you press `CTRL-C`, you might have to forcibly kill them. In UNIX, try `CTRL-Z` and then `kill %1`. In Windows, try `CTRL-ALT-DELETE` and then kill the listed command prompt task list entries.

Section 7. Resources and feedback

Resources

These resources are in addition to those listed [the HTTP tutorial](#) .

- * [The DOM specification](#)
 - * [XML 1.0 specification](#) , thoroughly annotated by Tim Bray
 - * [Introduction to XML: a tutorial](#)
 - * [Introduction to XML tutorial at Microsoft](#)
 - * [IBM XML Zone](#)
 - * , by Gordon Van Huizen
-

Giving feedback and finding out more

For technical questions about the content of this tutorial, contact the author, [Uche Ogbuji](#) .

Uche Ogbuji is a computer engineer, co-founder and principal consultant at [Fourthought, Inc](#) . He has worked with XML for several years, co-developing [4Suite](#) , a library of open-source tools for XML development in [Python](#) , and [4Suite Server](#) , an open-source, cross-platform XML data server providing standards-based XML solutions. He writes articles on XML for IBM developerWorks, LinuxWorld, SunWorld, and XML.com. Mr. Ogbuji is a Nigerian immigrant living in Boulder, CO.

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.