Advanced Technology Research

Using ZSI

Revision 1.0

July 30, 2007

Document prime: Chris Hobbs
Document Number: IS-2007-0008
Contributors:

CHAPTER 1

Introduction

1.1 What is ZSI?

The *Zolera Soap Infrastucture* (ZSI) is a Python Package that implements the SOAP 1.1 specification.

Starting with the WSDL definition of a web service it generates the necessary Python stubs, skeletons and helper classes to speed the creation of consumers (clients) and servers.

The ZSI homepage is at http://pywebsvcs.sourceforge.net/.

1.2 Version

This manual refers to version 2.0 of ZSI published on 2nd February 2007.

1.3 Context

This document assumes that you want to create a web services server and consumer starting with WSDL that you already have available: either because it forms part of a standard (e.g., Parlay-X) or because you have written it yourself. As reference [1] points out, producing the server from the WSDL rather than the WSDL from the server avoids locking an implementation in to a particular vendor's tool set and promotes interoperability.

1.4 Other Documentation

Various documents are available related to ZSI but those aimed at using (rather than developing) ZSI tend to be either excellent but somewhat out-of-date (e.g., reference [1]) or excellent but only addressing the *rpc/literal* style. There also seems to be various versions

of the user and developer manuals with the same version number but different content: make sure that you have the versions dated February 2007.

Useful documents include:

- Reference [2]: the Developer's Guide
- Reference [3]: the User's Guide
- Reference [1]: a description of implementing both *rpc/literal* and *document/literal* servers and then accessing them from Excel XP Visual Basic, gSOAP C/C++ and Applix spreadsheet consumers. Reference [1] describes the use of ZSI version 1.6.1 (released in December 2004). The current version at the time of writing is 2.0 (released in February 2007).
- Reference [4]: a description (in German) of developing a consumer and server (albeit using *rpc/literal*) with ZSI. It also contains information on the use of the Eclipse Web Tools Platform (WTP) WSDL editor.

CHAPTER 2

Installation

Installation of ZSI follows the normal method for installing Python modules:

- Download the tar file from Sourceforge. This will probably be called something like ZSI-2.0.tar.gz.
- Unzip the tar file:

  ```
  gunzip ZSI-2.0.tar.gz
  tar xvf ZSI-2.0.tar
  ```

- Go into the directory where the files have been created and invoke the Python installer:

  ```
  cd ZSI-2.0-rc3
  python setup.py install
  ```

  If you are not running as *root* then you may need to enter the command as

  ```
  sudo python setup.py install
  ```

  or read the documentation of the Python installer to direct the installation to a directory to which you have write access.

The installation is now complete. It includes a large number of test cases that should be run to ensure the integrity of the system. These test cases are also useful for hints on how to use the ZSI software.

CHAPTER 3

Code Generation

Note: the descriptions in this section and the examples in chapter 4 all assume that the services being created and invoked are of the "document/literal" rather than "rpc/literal" style. Reference [4] deals exclusively with "rpc/literal" services and reference [1] covers both.

3.1 Generating the Consumer (Client)

Given the WSDL, the code outline is generated by using the command

```
wsdl2py --complexType --file=ZonedLocation.wsdl
```

This extracts the service name from the `wsdl:service` element in the WSDL ("ZonedLocation" in this case) and uses it to create two files:

1. **ZonedLocation_services.py** that contains a consumer stub. In particular, it contains:
   - a `Locator` class. An instance of this class can be used to create an instance of a binding through which the service can be invoked.
   - a class for each operation that can be invoked on the server. This can be used to construct a message to be sent to the server.
2. **ZonedLocation_services_types.py** that contains various helper classes associated with the types defined in the WSDL.

The `-complexType` parameter causes wsdl2py to generate helper methods (getters, setters and factory methods) for each complex type in the WSDL. As can be seen from the examples in sections 4.1 and 4.2, it is not necessary to have a local copy of the WSDL: it can be accessed across the Internet.

3.2 Generating the Server

To create the basic code for a server, the following command should then be run. Note that `wsdl2py` needs to be run first.

```
wsdl2dispatch --extended --file=ZonedLocation.wsdl
```

This creates the file:

ZonedLocation_services_server.py that contains a framework for the server.

The use of these generated files is illustrated in examples in chapter 4.

CHAPTER 4

Examples

4.1 Creating a Simple Client

The site http://www.xmethods.net/ contains a number of links to web services of different sorts. These can provide test services for simple consumers.

This example uses the "Mighty Maxims" service that provides a Document (rather than rpc) interface to a daily quotation. To access the service carry out the following process:

1. Use the service WSDL to generate consumer software:

   ```
   wsdl2py -bu http://saintbook.org/MightyMaxims/MightyMaxims.asmx?WSDL
   ```

   The `b` option is the short-form of `-complexType` and `u` specifies a remote location (rather than local file) for the WSDL file. This will generate the two files `Maxim_services.py` and `Maxim_services_types.py`.

2. Write the necessary consumer code. In this case there are no elements to be sent to the service—it simply needs to be invoked and the result printed. A suitable program might be:

   ```
   # import the generated class stubs
   from Maxim_services import *

   # get a port proxy instance
   loc = MaximLocator()
   port = loc.getMaximSoap()

   # create a new request
   ```

```
req = ForTodaySoapIn()

# call the remote method
resp = port.ForToday(req)

# print the resulting quotation
print resp.ForTodayResult
```

3. Run the program. The quotation of the day will be displayed.

Note that the invocation `port = loc.getMaximSoap()` could have provided a URL for the service. The default is to take the URL from the WSDL.

4.2 Creating a Slightly More Sophisticated Client

As with the consumer in section 4.1, this consumer invokes a remote Document service: a service to convert Celsius temperatures to Fahrenheit. This service is one of the ones listed on http://www.xmethods.net/. It is slightly more complex than the consumer in section 4.1 as it requires an input parameter. The basic process is the same as before:

1. Use the service WSDL to generate consumer software:

   ```
   wsdl2py -bu http://webservices.daelab.net/temperatureconversions/TemperatureConversions.wso?WSDL
   ```

   This generates the two files `TemperatureConversions_services.py` and `TemperatureConversions_services_types.py`.

2. Write the necessary consumer code. A suitable program might be as follows (note the misspelling of Celsius in the WSDL and therefore in the program):

   ```
   # import the generated class stubs
   from TemperatureConversions_services import *

   celsiusTemp = 32

   # get a port proxy instance
   loc = TemperatureConversionsLocator()
   port = loc.getTemperatureConversionsSoapType()

   # create a new request
   req = CelciusToFahrenheitSoapRequest()
   req._nCelcius = celsiusTemp
   ```

```
# call the remote method
resp = port.CelciusToFahrenheit(req)

# print the result in Fahrenheit
print "%s degrees C = %s degrees F" % (celsiusTemp,
                resp.CelciusToFahrenheitResult)
```

3. Run the program. The result of the conversion is displayed:

   32 degrees C = 89.6 degrees F

4.3 Creating a Simple Server

In this example we create a server to respond with quotations using the same WSDL-defined interface as the remote server in section 4.1. We then use the same consumer code (with the new URL added) as the consumer to access it.

To create this server:

1. Run

   ```
   wsdl2py -bu http://saintbook.org/MightyMaxims/MightyMaxims.asmx?WSDL
   ```

   to generate the necessary stubs (unless you already did this following the example in section 4.1).

2. Run

   ```
   wsdl2dispatch -u http://saintbook.org/MightyMaxims/MightyMaxims.asmx?WSDL
   ```

   to generate the server skeleton.

3. Modify the *"port="* line in the client listed in section 4.1 to read

   ```
   port = loc.getMaximSoap("http://localhost:8080/cwlh")
   ```

   so that it uses a server on the local host rather than the remote server.

4. Write a server to return a quotation when invoked. Suitable code for this is as follows

   ```
   import sys
   import random
   from ZSI.ServiceContainer import AsServer
   from Maxim_services_server import *
   ```

```
quotations = [ "All men by nature desire knowledge. Aristotle",
   "Man is by nature a political animal. Aristotle",
   "Many people would sooner die than think. In fact they do. Russell",
   "Science may be described as the art of systematic over-simplification. Popper",
   "The smallest minority on earth is the individual. Rand",
   "The limits of my language mean the limits of my world. Wittgenstein" ]

class Service(Maxim):

    def soap_ForToday(self, ps):
        response = ns0.ForTodayResponse_Dec().pyclass()
        response.ForTodayResult = random.choice(quotations)
        return response

if __name__ == "__main__" :
    AsServer(8080, (Service('cwlh'),))
```

You may, if you wish, replace the quotations with your own.

This code makes use of the `AsServer` class contained in the ZSI libraries. It creates a web server and automatically dispatches incoming requests.

5. Point a browser at `http://localhost:8080/cwlh?wsdl` and note that the WSDL for the service is returned.
6. Run the consumer and note the wise quotation provided.

CHAPTER 5

Gotchas and Tips

This chapter contains a few tricks that the current author found necessary when using ZSI.

5.1 Tracing

Tracing of the exchanged messages can be switched on at the client by adding the
`tracefile=<filepointer>`
argument to the call to create the binding. So, in the client example given in section 4.2, the line

```
port = loc.getTemperatureConversionsSoapType()
```

would be coded as

```
fp = file("/home/cwlh/tmp/tracefile","w")
port = loc.getTemperatureConversionsSoapType(tracefile=fp)
```

Do not forget to close the file at some convenient point.

5.2 DateTime Fields

ZSI does not expect a datetime to be a string in the standard `xs:datetime` format. Instead it expects a list of 9 elements in the format returned by `datetime.datetime.now()` or equivalent methods. Note that this is *not* the format returned by `time.gmtime()`.

Consider, for example, an element called `EventTime` in a WSDL specification of type DateTime. Setting this value in the consumer might be done as follows:

```
dt = list(datetime.datetime.now().timetuple())
dt[6] = 0
request._EventTime = dt
```

The rather strange command to set `dt[6]` to zero is to ensure that the milliseconds field is not rendered in the XML—apparently necessary to ensure compatibility across various platforms.

The field is received at the server in the form indicated by this example:

```
eventTime = request.EventTime
print eventTime
(2007, 7, 16, 13, 47, 40, 0, 0, 0)
```

Note that the time appears in the form of a tuple rather than a list.

5.3 Duration Fields

ZSI does not expect a duration to be a string in the standard `xs:duration format`. Instead it expects a list of 6 elements in the format

```
[ year, month, day, hour, minute, second ]
```

so a duration of 3 days, 4 hours 5 minutes and 2 seconds would be represented as [0, 0, 3, 4, 5, 2].

At the server, a duration is delivered as a 9-tuple with the first six elements as delivered at the client and the last three elements apparently set to 0.

5.4 Exceptions

Consider a service operation that can return either a message or a SOAP Fault. A portion of the WSDL may appear as follows:

```
<wsdl:operation name="ExchangeMessages">
<wsdl:input message="tns:ExchangeMessagesRequest"/>
    <wsdl:output message="tns:ExchangeMessagesResponse"/>
    <wsdl:fault name="fault" message="tns:ExchangeMessages_faultMsg"></wsdl:fault>
    <wsdl:fault name="fault1" message="tns:ExchangeMessages_fault1Msg"></wsdl:fault>
</wsdl:operation>
```

A thorny question is how to generate the faults at the server. With the ZSI v2.0 code as it is provided, this is not possible. Different work-arounds are available and the simplest (provided by Joshua Boverhof: many thanks!) is probably to add the following lines to the `FaultFromException` method in the source file `fault.py` (remembering to regenerate the fault.pyc as appropriate):

```
if isinstance(ex, Fault):
    return ex
```

immediately before `elt = ZSIFaultDetail(string=exceptionName....`

This has the effect of allowing the server to generate any sort of SOAP fault by creating an instance of the `ZSI.Fault` class and then raising an exception.

For example, at the point in the server where a response message would normally be created and it is intended to produce a fault instead:

```
from ZSI import Fault

....

    response = ns0.NoResponseFault_Dec().pyclass()
    response.ItemIdentifier = "FredBloggs"
    fault = Fault(Fault.Server,
                          "NoResponseFault",
                          None,
                          response)
    raise fault
```

In this case `ns0.NoResponseFault_Dec` has been generated automatically from the WSDL fault definition by `wsdl2py` and `ItemIdentifier` is one of its elements.

If the consumer (client) is also written using ZSI then the handling of this SOAP fault is simple:

```
try:
    response = port.ExchangeMessages(request)
except Exception, ex:
    # pull the exception apart
    typeOfFault = ex.fault.string
    itemIdentifier = ex.fault.detail[0]._ItemIdentifier
```

Bibliography

[1] Holger Joukl. Interoperable WSDL/SOAP Web Services: Python ZSI, 2005.

[2] Rich Salz. ZSI: The Zolera Soap Infrastructure developer's guide, 2007.

[3] Joshua Boverhof. ZSI: The Zolera Soap Infrastructure user's guide, 2007.

[4] Richard Mutschler. PythonSOAP Tutorial: Erstellung eines WEB-Service mit ZSI, 2007.

>BUSINESS MADE **SIMPLE**