ZSI: The Zolera Soap Infrastructure User's Guide

Release 2.0.0

October 25, 2006

Acknowledgments

We are grateful to the members of the `soapbuilders` mailing list (see http://groups.yahoo.com/soapbuilders), Fredrik Lundh for his `soaplib` package (see http://www.secretlabs.com/downloads/index.htm#soap), Cayce Ullman and Brian Matthews for their `SOAP.py` package (see http://sourceforge.net/projects/pywebsvcs).

We are particularly grateful to Brian Lloyd and the Zope Corporation (http://www.zope.com) for letting us incorporate his ZOPE WebServices package and documentation into `ZSI`.

Abstract

ZSI, the Zolera SOAP Infrastructure, is a Python package that provides an implementation of SOAP messaging, as described in *The SOAP 1.1 Specification*. In particular, ZSI parses and generates SOAP messages, and converts between native Python datatypes and SOAP syntax. It can also be used to build applications using *SOAP Messages with Attachments*. ZSI is "transport neutral", and provides only a simple I/O and dispatch framework; a more complete solution is the responsibility of the application using ZSI. As usage patterns emerge, and common application frameworks are more understood, this may change.

ZSI requires Python 2.0 or later and PyXML version 0.6.6 or later.

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

CONTENTS

CHAPTER

ONE

Introduction

`ZSI`, the Zolera SOAP Infrastructure, is a Python package that provides an implementation of the SOAP specification, as described in *The SOAP 1.1 Specification*. This guide demonstrates `ZSI`'s typecode generation facilities, these typecodes are then used to parse and generates SOAP messages, and converts between native Python datatypes and SOAP syntax.

`ZSI` requires Python 2.3 or later and PyXML version 0.8.3 or later.

The `ZSI` project is maintained at SourceForge, at http://pywebsvcs.sf.net. `ZSI` is discussed on the Python web services mailing list, visit http://lists.sourceforge.net/lists/listinfo/pywebsvcs-talk to subscribe.

For a low-level treatment of typecodes, and a description of SOAP-based processing see the ZSI manual.

Within this document, `tns` is used as the prefix for the application's target namespace, and the term *element* refers to a DOM element node.)

1.1 How to Read this Document

CHAPTER

TWO

WSDL/XMLSchema python code generation

Handling XML Schema (see *XML Schema specification*) is one of the more difficult aspects of using WSDL (see *The Web Services Description Language*. Using wsdl2py generates a module with stub code for the client interface, and a ”types” module that contains typecode representations of the XML Schema types and elements. The generated typecodes are registered in a global schema instance, and once the ”types” module is imported by an application all the global elements declarations and type definitions are available everywhere (see section ??).

2.1 wsdl2py

2.1.1 Command Line Flags

General Flags

-h, —help Display the help message and available command line flags that can be passed to wsdl2py.

-f FILE, —file=FILE Create bindings for the WSDL which is located at the local file path.

-u URL, —url=URL Create bindings for the remote WSDL which is located at the provided URL.

-x, —schema Just process a schema (xsd) file and generate the types mapping file.

-d, —debug Output verbose debugging messages during code generation.

-o OUTPUT_DIR, —output-dir=OUTPUT_DIR Write generated files to OUTPUT_DIR.

Typecode Extensions (Stable)

-b, —complexType (more in subsection) Generate convenience functions for complexTypes. This includes getters, setters, factory methods, and properties. ** Do NOT use with –simple-naming **

Development Extensions (Unstable)

-a, —address WS-Addressing support. The WS-Addressing schema must be included in the corresponding WSDL.

-w, —twisted Generate a twisted.web client. Dependencies: python>=2.4, Twisted>=2.0.0, TwistedWeb>=0.5.0

Customizations (Unstable)

-e, —extended Do extended code generation.

-z ANAME, —aname=ANAME Use a custom function, ANAME, for attribute name creation.

-t TYPES, —types=TYPES Dump the generated type mappings to a file named, “TYPES.py”.

-s, —simple-naming Simplify the generated naming.

-c CLIENTCLASSSUFFIX, —clientClassSuffix=CLIENTCLASSSUFFIX The suffic to use for service client class. (default “SOAP”)

-m PYCLASSMAPMODULE, —pyclassMapModule=PYCLASSMAPMODULE Use the existing existing type mapping file to determine the “pyclass” objects to be used. The module should contain an attribute, “mapping”, which is a dictionary of form, schemaTypeName: (moduleName.py, className).

2.1.2 Basics of Code Generation

client stub module

Using only the *General Flags* options one can generate a **client stub module** from a WSDL description, consisting of representations of the WSDL information items *service*, *binding*, *portType*, and *message*.

These four items are represented by three abstractions, consisting of a ***Locator*** class, ***PortType*** class, and several ***Message*** classes. The ***Locator*** will have two methods for each *service port* declared in the WSDL definition. One method returns the address specified in the *binding*, and the other is a factory method for returning a ***PortType*** instance. Each ***Message*** class represents the aspects of the *binding* at the operation level and below, and any type information specified by *message part* items.

types module

The **types module** is generated with the **client module** but it can be created independently. This is especially useful for dealing with schema definitions that aren’t specified inside of a WSDL document.

The module level class defintions each represent a unique namespace, they are simply wrappers for the contents of individual namespaces. The inner classes are the typecode representations of global *type definitions* (suffix ***_Def***), and *element declarations* (suffix ***_Dec***).

understanding the generated typecodes

The generated inner typecode classes come in two flavors, as mentioned above. *element declarations* can be serialized into XML, generally *type definitions* cannot. In very simple terms, the *name* attribute of an *element declaration* is serialized into an XML tag, but *type definitions* lack this information so they cannot be directly serialized into an XML instance. Most *element declaration*s declare a *type* attribute, this must reference a *type definition*. Considering the above scenario, a generated *TypeCode* class representing an *element declaration* will subclass the generated *TypeCode* class representing the *type definition*.

pyclass All instances of generated *TypeCode* classes will have a *pyclass* attribute, instances of the *pyclass* can be created to store the data representing an *element declaration*. The *pyclass* itself has a *typecode* attribute, which is a reference to the *TypeCode* instance describing the data, thus making *pyclass* instances self-describing. When parsing an XML instance the data will be marshalled into a *pyclass* instance.

aname The *aname* is a *TypeCode* instance attribute, its value is a string representing the attribute name used to reference data representing an element declaration. The set of *XMLSchema* element names is *NCName*, this is a superset of ordinary identifiers in *python*.

Namespaces in XML

```
From Namespaces in XML
NCName  ::= (Letter | '_') (NCNameChar)*
NCNameChar  ::= Letter | Digit | '.' | '-' | '_' | CombiningChar | Extender

From Python Reference Manual (2.3 Identifiers and keywords)
identifier ::= (letter|"_") (letter | digit | "_")*

Default set of anames
ANAME ::= ("_") (letter | digit | "_")*
```

transform *NCName* into an *ANAME*

1. preprend "_"
2. character not in set (letter | digit | "_") change to "_"

Attribute Declarations: attrs_aname The *attrs_aname* is a *TypeCode* instance attribute, its value is a string representing the attribute name used to reference a dictionary, containing data representing attribute declarations. The keys of this dictionary are the `(namespace,name)` tuples, the value of each key represents the value of the attribute.

Mixed Text Content: mixed_aname

2.1.3 Typecode Extensions

–complexType

The *complexType* flag provides many conveniences to the programmer. This option is tested and reliable, and highly recommended by the authors.

low-level description When enabled the `__metaclass__` attribute will be set on all generated *pyclass*es. The metaclass will introspect the *typecode* attribute of *pyclass*, and create a set of helper methods for each element and attribute declared in the *complexType* definition. This option simply adds wrappers for dealing with content, it doesn't modify the generation scheme.

Getters/Setters A getter and setter function is defined for each element of a complex type. The functions are named `get_element_ANAME` and `set_element_ANAME` respectively. In this example, variable *wsreq* has functions named `get_element__Options` and `set_element__Options`. In addition to elements, getters and setters are generated for the attributes of a complex type. For attributes, just the name of the attribute is used in determining the method names, so get_attribute_NAME and set_attribute_NAME are created.

Factory Methods If an element of a complex type is a complex type itself, then a conveniece factory method is created to get an instance of that types holder class. The factory method is named, `newANAME`, so *wsreq* has a factory method, `new_Options`.

Properties *Python class properties* are created for each element of the complex type. They are mapped to the corresponding getter and setter for that element. To avoid name collisions the properties are named, `PNAME`, where the first letter of the type's pname attribute is capitalized. In our running example, *wsreq* has class property, `Options`, which calls functions `get_element__Options` and `set_element__Options` under the hood.

```
<xsd:complexType name='WolframSearchOptions'>
  <xsd:sequence>
    <xsd:element name='Query' minOccurs='0' maxOccurs='1' type='xsd:string'/>
    <xsd:element name='Limit' minOccurs='0' maxOccurs='1' type='xsd:int'/>
  </xsd:sequence>
  <xsd:attribute name='timeout' type='xsd:double' />
</xsd:complexType>
<xsd:element name='WolframSearch'>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name='Options' minOccurs='0' maxOccurs='1' type='ns1:WolframSearchOptions'/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

```
# Create a request object to operation WolframSearch
#   to be used as an example below
from WolframSearchService_services import *

port = WolframSearchServiceLocator().getWolframSearchmyPortType()
wsreq = WolframSearchRequest()
```

```
# sample usage of the generated code

# get an instance of a Options holder class using factory method
opts = wsreq.new_Options()
wsreq.Options = opts

# assign values using the properties or methods
opts.Query = 'Newton'
opts.set_element__Limit(10)

# don't forget the attribute
opts.set_attribute_timeout(1.0)

# At this point the serialized wsreq object would resemble this:
# <WolframSearch>
#   <Options timeout="1.0" xsi:type="tns:WolframSearchOptions">
#     <Query xsi:type="xsd:string">Newton</Query>
#     <Limit xsi:type="xsd:double">10.0</Limit>
#   </Options>
# </WolframSearch>

# ready call the remote operation
wsresp = port.WolframSearch(wsreq)

# returned WolframSearchResponse type holder also has conveniences
print 'SearchTime:', wsresp.Result.SearchTime
```

2.2 Code Generation from WSDL and XML Schema

This section covers wsdl2py, the second way ZSI provides to access WSDL services. Given the path to a WSDL service, two files are generated, a 'service' file and a 'types' file, that one can then use to access the service. As an example, we will use the search service provided by Wolfram Research Inc.©, http://webservices.wolfram.com/wolframsearch/, which provides a service for searching the popular MathWorld site, http://mathworld.wolfram.com/, among others.

```
wsdl2py --complexType --url=http://webservices.wolfram.com/services/SearchServices/WolframSearc
```

Run the above command to generate the service and type files. wsdl2py uses the *name* attribute of the *wsdl:service* element to name the resulting files. In this example, the service name is *WolframSearchService*. Therefore the files *WolframSearchService_services.py* and *WolframSearchService_services_types.py* should be generated.

The 'service' file contains locator, portType, and message classes. A locator instance is used to get an instance of a portType class, which is a remote proxy object. Message instances are sent and received through the methods of the portType instance.

The 'types' file contains class representations of the definitions and declarations defined by all schema instances imported by the WSDL definition. XML Schema attributes, wildcards, and derived types are not fully handled.

2.2.1 Example Use of Generated Code

The following shows how to call a proxy method for *WolframSearch*. It assumes wsdl2py has already been run as shown in the section above. The example will be explained in greater detail below.

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

# get a port proxy instance
loc = WolframSearchServiceLocator()
port = loc.getWolframSearchmyPortType()

# create a new request
req = WolframSearchRequest()
req.Options = req.new_Options()
req.Options.Query = 'newton'

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

# print results
print 'Search Time:', resp.Result.SearchTime
print 'Total Matches:', resp.Result.TotalMatches
for hit in resp.Result.Matches.Item:
    print '--', hit.Title
```

Now each section of the code above will be explained.

```
from WolframSearchService_services import *
```

We are primarily interested in the service locator that is imported. The binding proxy and classes for all the messages are additionally imported. Look at the *WolframSearchService_services.py* file for more information.

```
loc = WolframSearchServiceLocator()
port = loc.getWolframSearchmyPortType()
```

Using an instance of the locator, we fetch an instance of the port proxy which is used for invoking the remote methods provided by the service. In this case the default *location* specified in the *wsdlsoap:address* element is used. You can optionally pass a url to the port getter method to specify an alternate location to be used. The *portType* - *name* attribute is used to determine the method name to fetch a port proxy instance. In this example, the port name is *WolframSearchmyPortType*, hence the method of the locator for fetching the proxy is *getWolframSearchmyPortType*.

The first step in calling *WolframSearch* is to create a request object corresponding to the input message of the method. In this case, the name of the message is *WolframSearchRequest*. A class representing this message was imported from the service module.

```
req = WolframSearchRequest()
req.Options = req.new_Options()
req.Options.Query = 'newton'
```

Once a request object is created we need to populate the instance with the information we want to use in our request.

This is where the `--complexType` option we passed to wsdl2py will come in handy. This caused the creation of functions for getting and setting elements and attributes of the type, class properties for each element, and convenience functions for creating new instances of elements of complex types. This functionality is explained in detail in subsection 2.1.3.

Once the request instance is populated, calling the remote service is easy. Using the port proxy we call the method we are interested in. An instance of the python class representing the return type is returned by this call. The *resp* object can be used to introspect the result of the remote call.

```
resp = port.WolframSearch(req)
```

Here we see that the response message, *resp*, represents type *WolframSearchReturn*. This object has one element, *Result* which contains the search results for our search of the keyword, `newton`.

```
print 'Search Time:', resp.Result.SearchTime
...
```

Refer to the wsdl for *WolframSearchService* for more details on the returned information.

2.3 Advanced Usage Patterns

Not done.