ZSI: The Zolera Soap Infrastructure User's Guide

Release 2.0.0

Joshua Boverhof,
Charles Moad

February 01, 2007

jrboverhof@lbl.gov

CONTENTS

CHAPTER

ONE

Introduction

`ZSI`, the Zolera SOAP Infrastructure, is a Python package that provides an implementation of the SOAP specification, as described in *SOAP 1.1 Specification*.

This guide demonstrates how to use ZSI to develop *Web Service* applications from a *Web Services Description Language* document.

This document is primarily concerned with demonstrating and documenting how to use a *Web Service* by creating and accessing Python data for the purposes of sending and receiving SOAP messages. Typecodes are used to marshall Python datatypes into XML, which can be included in a SOAP Envelope. The typecodes are generated from information provided in the WSDL document, and generally describe SOAP and XML Schema data types. For a low-level treatment of typecodes, and a description of SOAP-based processing see the ZSI manual.

1.1 Acronyms and Terminology

SOAP
: Usually refering to the content and format of a message ultimately sent and received by a *Web Service*, see *SOAP 1.1 Specification*

WSDL
: A document describing a *Web Service*'s interface, see *Web Services Description Language*

XMLSchema
: Standard for modeling XML document structure. See *XML Schema Specification*

schema document
: a file containing a schema definition.

schema (instance)
: The set of rules or components contained in the assemblage of one or more schema documents.

Element Declaration
: A schema component that associates a name with a type definition. eg. *¡element name="age" type="xsd:int"¿,*

GED
: *Global Element Declaration, an element declared at the top-level of a schema.*

ComplexType
: The parent of all type definitions that can specify attributes and children.

SimpleType
: A simple data type like a string or integer. The *XML Schema Specification* defines many built-in types.

The XML Schema type library
: The `http://www.w3.org/2001/XMLSchema` namespace, which contains definitions of various primitive types like string and integer, as well as a compound type *complexType used to create aggregate types. Conventionally the* xsd *prefix is used to map to this schema.*

doc/literal
: document style with literal encoding

rpc/enc
: rpc style with specified encoding, not compatible with *Basic Profile (WS-Interop)*

rpc/literal
: rpc style with literal encoding.

1.2 Overview

The ZSI *Web Service* tools are for top-Down *Web Service* development, using an existing WSDL Document to create client and server applications (see 1.3). A *Web Service*, in the context of this document, exposes a WSDL Document describing the service's interface, this document is typically available at a published URL (see *Uniform Resource Locator*). The WSDL document defines SOAP bindings for communicating with the service. These bindings will be used to exchange SOAP messages, the contents of these messages must adhere to the document structure specified by the schema. The schema is either included in the WSDL Document, imported by it, or represented by the available built-in types (such as *xsd:int, xsd:string, etc*).

1.2.1 soap bindings

The two styles of SOAP bindings are *rpc* and *document*. The use of *literal* encoding is encouraged and the recommended way to develop new *Web Service* applications (see *Basic Profile (WS-Interop)*). The SOAP *encoded* support is maintained for use with old apps, and other SOAP toolkits restricted to *rpc/enc* development. A *doc/literal* service is typically described as an exchange of documents, while a *rpc/enc* or *rpc/literal* service is thought of in terms of remote procedure calls. Whether this distinction of purpose is meaningful or useful is debatable. `ZSI` supports all three types, but *rpc/literal* and *doc/literal* are the focus of ongoing development.

1.2.2 python tools

wsdl2py

The **wsdl2py** script generates python code representing the various components defined in a WSDL Document. The second chapter introduces **wsdl2py** and demonstrates how to create a client for a simple *Web Service* from a WSDL document.

wsdl2dispatch

The **wsdl2dispatch** script should be run after running **wsdl2py** , since the module it generates will attempt to import all **wsdl2py** generates. This sciprt generates a module containing a service interface generated from the WSDL Document. This interface is typically subclassed and invoked through an HTTP server.

1.3 Not Covered

1. How to create a WSDL document

2. How to write XML Schema
3. Interoperability
4. How to use Web Services without WSDL

1.4 References

1. Web services development patterns `http://www-128.ibm.com/developerworks/websphere/library/techarticles/0511_flurry/0511_flurry.html`

CHAPTER

TWO

WSDL to Python Code

The **wsdl2py** script is the primary tool. It will generate all the code needed to access a Web Service through an exposed WSDL document, usually this description is available at a URL which is provided to the script.

wsdl2py generates a ”stub” module from the WSDL SOAP Bindings. It contains various classes, including a `Locator` which represents the bindings to the actual Web Service, and several `port` classes that are used to remotely invoke operations on the *Web Service*, as well as various `Message` classes that represent the SOAP and XML Schema data types. A `Message` instance is serialized as a XML instance. A `Message` passed as an argument to a `port` method is then serialized into a SOAP Envelope and transported to the *Web Service*, the client will then wait for an expected response, and finally the SOAP response is marshalled back into the `Message` returned to the user.

A second module the ”types”, contains typecodes representing all the components of each schema specified by the target WSDL Document (not including built-in types). Each schema component declared at the top-level, the immediate children of the *schema* tag, are global in scope and by importing the ”types” module an application has access to the GEDs and global type definitions either directly by reference or through the convenience functions `GED` and `GTD`, respectively.

2.1 wsdl2py

2.1.1 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*.

class `Locator`(***keywords*)
The following keyword arguments may be used:

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

2.1.2 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 -bu http://webservices.wolfram.com/services/SearchServices/WolframSearch2.wsdl
```

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 A.1.2.

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.

CHAPTER

THREE

Using ServiceContainer with wsdl2dispatch

The **wsdl2dispatch** script generates a service skeleton, typicallly this interface will be subclassed and the base class methods will be invoked by the implementation code. Functionally the methods of the base class will parse the SOAP request into the expected message, which is available by referencing the instance's *request* attribute, and return an initialized response message.

3.1 running the script

Running the command below generates the file *WolframSearchService_services_server.py*

```
wsdl2dispatch -u http://webservices.wolfram.com/services/SearchServices/WolframSearch2.wsdl
```

3.2 class ServiceInterface

In the `ServiceContainer` infrastructure all generated service skeletons subclass `ServiceInterface`. A service instance's *post* attribute specifies the path used to contact the service.

```
class ServiceInterface:
    '''Defines the interface for use with ServiceContainer Handlers.
    '''

    def __init__(self, post):
        self.post = post
```

3.3 service skeleton

The skeleton is generated by the **wsdl2dispatch** script. The `WolframSearch` operation is shown below, the request SOAP message is parsed into a python instance and the response is initialized and returned.

```
# WolframSearchService_services_server.py
class WolframSearchService(ServiceSOAPBinding):
    ...
    def soap_WolframSearch(self, ps):
        self.request = ps.Parse(WolframSearchRequest.typecode)
        return WolframSearchResponse()
    ...
```

3.4 service implementation

The user must implement the service and dump it into a container that understands how to correctly dispatch messages to the implementation's various methods.

```
#!/usr/bin/env python
import sys
from ZSI.ServiceContainer import AsServer
from WolframSearchService_services_server import *

class Service(WolframSearchService):

    def soap_WolframSearch(self, ps):
        rsp = WolframSearchService.soap_WolframSearch(self, ps)
        msg = self.request

        t1 = time.time()
        opts = msg.Options

        rsp.Result = result = rsp.new_Result()
        if opts.Query == 'Newton':
            result.TotalMatches = 1
            result.Matches = match = result.new_Matches()
            item = match.new_Item()
            item.Title = "Fig Newtons"
            item.Score = 10
            item.URL = 'http://www.nabiscoworld.com/newtons/'
            match.Item = [item]

        result.SearchTime = time.time() - t1

        return rsp

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

The `ZSI.ServiceContainer.AsServer` function is a convenient way to start a HTTP server that will dispatch to your various services. In a container all services should have unique paths, here the service is available at "test". A URL to contact this service at port 8080 would be *http://localhost:8080/test*

APPENDIX

A

Wsdl2py script

A.1 Command Line Flags

usage: wsdl2py [options]

A.1.1 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.

A.1.2 Typecode Extensions (Stable)

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

A.1.3 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

-l, —lazy recursion error solution, lazy evalution of typecodes

A.1.4 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).

APPENDIX

B

Wsdl2dispatch script

B.1 Command Line Flags

usage: wsdl2dispatch [options]

B.1.1 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.

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

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

B.1.2 Development Extensions (Unstable)

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

B.1.3 Customizations (Unstable)

-e, —extended Do extended code generation.

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

-t TYPES, –types=TYPES file to load types from