ZSI: The Zolera Soap Infrastructure User's Guide

Release 2.1.0

Joshua Boverhof,
Charles Moad

November 01, 2007

jrboverhof@lbl.gov

COPYRIGHT

Copyright © 2001, Zolera Systems, Inc.
All Rights Reserved.

Copyright © 2002-2003, Rich Salz.
All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

Copyright © (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from the U.S. Dept. of Energy). All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. (3) Neither the name of the University of California, Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code (”Enhancements”) to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such Enhancements or derivative works thereof, in binary and source code form.

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]

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 older applications, and other SOAP toolkits restricted to rpc/encdevelopment. A doc/literalservice is typically described as an exchange of documents, while a rpc/encor rpc/literalservice 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/literaland doc/literalare 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. Most of the remaining guide focuses on how to use this tool and understand its output.*

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] The xsd prefix refers to namespace *"http://www.w3.org/2001/XMLSchema"*

1.4 References

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

CHAPTER

TWO

wsdl2py basics

The **wsdl2py** *script is used to 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 *will generate a client, types, and service module. From the the WSDL SOAP Bindings, the client and service modules are created. The types module contains typecodes for the schema defined by the WSDL.*

2.1 Modules

2.1.1 client stub module

classes

The service item in the Web Services Description Language *definition contains one or more port items.*

locator *Defines a factory method for each port item, and stores the service's address. Use to grab a client(port) to the* Web Service.

```
# Example Locator
class WhiteMesaSoapRpcLitTestSvcLocator:
    SoapTestPortTypeRpc_address = "http://www.whitemesa.net/test-rpc-lit"
    def getSoapTestPortTypeRpcAddress(self):
        return WhiteMesaSoapRpcLitTestSvcLocator.SoapTestPortTypeRpc_address
    def getSoapTestPortTypeRpc(self, url=None, **kw):
        return Soap11TestRpcLitBindingSOAP(url or WhiteMesaSoapRpcLitTestSvcLocator.SoapTestPor
}
```

port *Each port item will be represented by a single class definition, grab a port through one of the locator's factory methods.*

```
loc = WhiteMesaSoapRpcLitTestSvcLocator()
port = loc.getSoapTestPortTypeRpc(tracefile=sys.stdout)
```

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.*

```
msg = echoBooleanRequest()
msg.InputBoolean = True
rsp = port.echoBoolean(msg)
```

2.1.2 types module

Defines typecodes for all components of all 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 or with the unique (namespace,name) combination thru convenience functions.

classes

Global Type Definition

```
class ns1:
    ..
    ..
    class HelpRequest_Def(ZSI.TCcompound.ComplexType, TypeDefinition):
        schema = "http://webservices.amazon.com/AWSECommerceService/2006-11-14"
        type = (schema, "HelpRequest")
        def __init__(self, pname, ofwhat=(), attributes=None, extend=False, restr
            ..
```

Global Element Declaration

```
class ns1:
    ..
    ..
    class Help_Dec(ZSI.TCcompound.ComplexType, ElementDeclaration):
        literal = "Help"
        schema = "http://webservices.amazon.com/AWSECommerceService/2006-11-
        def __init__(self, **kw):
            ..
```

helper functions

Global Type Definition

```
klass = ZSI.schema.GTD(\
    "http://webservices.amazon.com/AWSECommerceService/2006-11-14",
    "HelpRequest")
typecode = klass("Help")
```

Global Element Declaration

```
typecode = ZSI.schema.GED(\
    "http://webservices.amazon.com/AWSECommerceService/2006-11-14",
    "Help")
```

Each module level class defintion represents a unique namespace, they're simply wrappers of individual namespaces. In the example above, the two inner classes of `ns1` *are the typecode representations of a global type definition* ***HelpRequest_Def****, and a global element declaration* ***Help_Dec****. In most cases a* `TypeCode` *instance represents either a global or local element declaration.*

In the example `GED` *returns a* `Help_Dec` *instance while* `GTD` *returns the class definition* `HelpRequest_Def`*. Why this asymmetry? The element name is serialized as the XML tag name, while the type definition describes the contents (children, text node).*

In the generated code an element declaration either defines all its content in its contructor or it subclasses a global type definition, which is another generated class.

2.1.3 service module

skeleton class, normally subclassed and invoked by implementation code. The skeleton defines a callback method for each operation defined in the SOAP Binding. These methods marshal/unmarshall XML into python types.

example: DateService

server skeleton code

```
class simple_Date_Service(ServiceSOAPBinding):
    ..
    ..
    def soap_getCurrentDate(self, ps):
        self.request = ps.Parse(getCurrentDateRequest.typecode)
        return getCurrentDateResponse()

    soapAction['urn:DateService.wsdl#getCurrentDate'] = 'soap_getCurrentDate'
    root[(getCurrentDateRequest.typecode.nspname,getCurrentDateRequest.typecode.pr
```

server implementation code

```
DS = simple_Date_Service
class Service(DS):
    def soap_getCurrentDate(self, ps):
        response = DS.soap_getCurrentDate(self, ps)
        response.Today = today = response.new_today()
        self.request.Input
        dt = time.localtime(time.time())
        today.Year = dt[0]
        today.Month = dt[1]
        today.Day = dt[2]
        today.Hour = dt[3]
        today.Minute = dt[4]
        today.Second = dt[5]
        today.Weekday = dt[6]
        today.DayOfYear = dt[7]
        today.Dst = dt[8]
        return response
```

2.2 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.[1] *Basically, 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 declarations 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.

2.2.1 special handling of instance attributes

The attributes discussed below are common to all TypeCodes, for more information see the ZSI manual. I'm reintroducing them to point out certain conventions adhered to in the generated code, necessary for reliably dealing with WSDL and various messaging patterns and usages.

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.[2]*. 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 an instance of the class specified in the typecode's pyclass attribute.

```
typecode = ZSI.schema.GED(\
    "http://webservices.amazon.com/AWSECommerceService/2006-11-14",
    "Help")
msg = typecode.pyclass()
```

[1] The *pname can be set to None when a XML tag name is not needed (eg. attributes).*
[2] Exceptions include the Union TypeCode, may need multiple pyclasses to make it work

aname

The aname is a TypeCode instance attribute, its value is a string representing the attribute name used to reference data representing an element declartion. The set of XMLSchema element names is NCName, this is a superset of ordinary identifiers in python. Keywords like return and class are legal NCNames.

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

Its value represents the attribute name used to store text content that some ComplexType definitions allow.

2.2.2 Metaclass Magic: pyclass_type

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

When –complexType is enabled the `__metaclass__` *attribute will be set on all generated pyclasses. 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.*

Use help in a python interpreter to view all the properties and methods of these typecodes. Looking at the generated code is not very helpful.

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 msg 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`.

Properties Python class properties *are created for each element of the complex type. They are initialized with 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, msg has class property,* `Options`*, which calls functions* `get_element__Options` *and* `set_element__Options` *under the hood.*

example

schema *Taken from the WolframSearch WSDL.*

```
<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>
```

help *(WolframSearchRequest)*

```
Help on WolframSearch_Holder in module WolframSearchService_types object:

class WolframSearch_Holder(__builtin__.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  get_element_Options(self)
 |
 |  new_Options(self)
 |      returns a mutable type
 |
 |  set_element_Options(self, value)
 |
 |  ----------------------------------------------------------------------
 |  Properties defined here:
 |
 |  Options
 |      property for element (None,Options), minOccurs="0" maxOccurs="1" nillable="False"
 |
 |      <get> = get_element_Options(self)
 |
 |      <set> = set_element_Options(self, value)
 |
```

request

```
from WolframSearchService_client import *
msg = WolframSearchRequest()
# get an instance of a Options holder class using factory method
msg.Options = opts = msg.new_Options()

# 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)
```

invoke

```
port = WolframSearchServiceLocator().getWolframSearchmyPortType()
rsp = port.WolframSearch(msg)
print 'SearchTime:', rsp.Result.SearchTime
```

XML *XML approximation of our WolframSearchRequest instance.*

```
<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>
```

CHAPTER

THREE

Security

3.1 HTTP Basic Authorization

auth=dict(style=ZSI.AUTH.httpbasic, user=USERNAME, password=PASSWORD)

3.2 HTTP Digest Authorization

auth=dict(style=ZSI.AUTH.httpdigest, user=USERNAME,, password=PASSWORD)

3.3 Message Security

CHAPTER

FOUR

SOAP Headers

CHAPTER

FIVE

Type Substitution

APPENDIX

A

Wsdl2py scrpt

A.1 Command Line Flags

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*

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

Example: WolframSearch

B.1 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 http://webservices.wolfram.com/services/SearchServices/WolframSearch2.wsd
```

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.

B.1.1 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_client 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_client 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.