
ZSI: The Zolera Soap Infrastructure Developer's Guide

Release 2.1.0

Rich Salz,
Christopher Blunck

November 01, 2007

rsalz@datapower.com
blunck@python.org

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.

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 2.3 or later.

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

CONTENTS

| | | |
|-----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | How to Read this Document | 2 |
| 2 | Examples | 3 |
| 2.1 | Server Side Examples | 3 |
| 2.2 | Client Side Examples | 7 |
| 3 | Exceptions | 11 |
| 4 | Utilities | 13 |
| 4.1 | Low-Level Utilities | 13 |
| 5 | The <code>ParsedSoap</code> module — basic message handling | 15 |
| 6 | The <code>TypeCode</code> classes — data conversions | 19 |
| 6.1 | <code>TC.TypeCode</code> | 19 |
| 6.2 | <code>TC.Any</code> — the basis of dynamic typing | 21 |
| 6.3 | <code>TC.SimpleType</code> | 23 |
| 6.4 | Strings | 23 |
| 6.5 | Integers | 24 |
| 6.6 | Floating-point Numbers | 25 |
| 6.7 | Dates and Times | 26 |
| 6.8 | Boolean | 26 |
| 6.9 | XML | 27 |
| 6.10 | <code>ComplexType</code> | 27 |
| 6.11 | <code>Struct</code> | 28 |
| 6.12 | Arrays | 28 |
| 6.13 | Apache Datatype | 29 |
| 7 | The <code>SoapWriter</code> module — serializing data | 31 |
| 8 | The <code>Fault</code> module — reporting errors | 33 |
| 9 | The <code>resolvers</code> module — fetching remote data | 35 |
| 10 | Dispatching and Invoking | 37 |
| 10.1 | Dispatching | 37 |
| 10.2 | The <code>client</code> module — sending SOAP messages | 40 |
| 11 | Bibliography | 43 |

| | | |
|----------|-----------------------------------|-----------|
| A | CGI Script Array | 47 |
| A.1 | Intro | 47 |
| A.2 | CGI Script | 47 |
| A.3 | client test script | 48 |
| A.4 | SOAP Trace | 48 |
| B | CGI Script Struct | 53 |
| B.1 | Intro | 53 |
| B.2 | CGI Script | 53 |
| B.3 | client test script | 54 |
| B.4 | SOAP Trace | 54 |
| C | Complete Low Level Example | 55 |
| C.1 | Intro | 55 |
| C.2 | code | 55 |
| C.3 | SOAP Trace | 59 |
| D | pickler example | 61 |
| D.1 | Intro | 61 |
| D.2 | code | 61 |

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*. In particular, ZSI parses and generates SOAP messages, and converts between native Python datatypes and SOAP syntax.

ZSI requires Python 2.3 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 those interested in using the **wSDL2py** tool see the *Users Guide*, it contains a detailed example of how to use the code generation facilities in ZSI.

For those interested in a high-level tutorial covering ZSI and why Python was chosen, see the article <http://www.xml.com/pub/a/ws/2002/06/12/soap.html>, written by Rich Salz for xml.com.

SOAP-based processing typically involves several steps. The following list details the steps of a common processing model naturally supported by ZSI (other models are certainly possible):

1. ZSI takes data from an input stream and *parses* it, generating a DOM-based parse tree as part of creating a `ParsedSoap` object. At this point the major syntactic elements of a SOAP message — the `Header`, the `Body`, etc. — are available.
2. The application does *header processing*. More specifically, it does local dispatch and processing based on the elements in the SOAP `Header`. The SOAP `actor` and `mustUnderstand` attributes are also handled (or at least recognized) here.
3. ZSI next *parses* the `Body`, creating local Python objects from the data in the SOAP message. The parsing is often under the control of a list of data descriptions, known as *typecodes*, defined by the application because it knows what type of data it is expecting. In cases where the SOAP data is known to be completely self-describing, the parsing can be *dynamic* through the use of the `TC.Any` class.
4. The application now *dispatches* to the appropriate handler in order to do its “real work.” As part of its processing it may create *output objects*
5. The application creates a `SoapWriter` instance and outputs an initial set of namespace entries and header elements.
6. Any local data to be sent back to the client is *serialized*. As with `Body` parsing, the datatypes can be described through typecodes or determined dynamically (here, through introspection).
7. In the event of any processing exceptions, a `Fault` object can be raised, created, and/or serialized.

Note that ZSI is “transport neutral”, and provides only a simple I/O and dispatch framework; a more complete solution is available through the use of included WSDL tools (**wSDL2py**), but otherwise this is the responsibility of the

application using ZSI. As usage patterns emerge, and common application frameworks are more understood, this may change.

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

Readers interested in using WSDL and clients to web services, and those intending on implementing web services based on existing WSDL should refer to the *Users Guide*. Others interested in developing the simplest SOAP applications, or spending the least amount of time on building a web services infrastructure should read chapters 2, 3, and 10 of this document. Readers who are developing complex services, and who are familiar with XML Schema and/or WSDL, should read this manual in order. This will provide them with enough information to implement the processing model described above. They can skip probably skip chapters 2 and 10.

ZSI has the capability to process WSDL definitions and XML Schema documents (described in [The Web Services Description Language](#) and [XMLSchema 1.0](#)) and generate typecodes automatically. For more information see the *Users Guide*.

Examples

This chapter contains a number of examples to show off some of ZSI's features. It is broken down into client-side and server-side examples, and explores different implementation options ZSI provides.

2.1 Server Side Examples

2.1.1 Simple example

Using the `ZSI.dispatch` module, it is simple to expose Python functions as web services. Each function is invoked with all the input parameters specified in the client's SOAP request. Any value returned by the function will be serialized back to the client; multiple values can be returned by returning a tuple.

The following code shows some simple services:

```
#!/usr/local/bin/python2.4
# SOAP Array

def hello():
    return ["Hello, world"]

def echo(*args):
    return args

def sum(*args):
    sum = 0
    for i in args: sum += i
    return [sum]

def average(*args):
    return [sum(*args) / len(args)]

from ZSI import dispatch
dispatch.AsCGI(rpc=True)
```

Each function defines a SOAP request, so if this script is installed as a CGI script, a SOAP message can be posted to that script's URL with any of `hello`, `echo`, or `average` as the request element, and the value returned by the function will be sent back. These functions expect and return SOAP-ENC:arrayType instances which are marshalled into python list instances, this script interoperates with the `client.Binding`. For more information see *Appendix A*.

The ZSI CGI dispatcher catches exceptions and sends back a SOAP fault. For example, a fault will be sent if the

hello function is given any arguments, or if the average function is given a non-integer.

Here is another example but using SOAP-ENC:Struct instances which are marshalled into python dict instances, this script interoperates with the client.NamedParamBinding. For more information see *Appendix B*.

```
#!/usr/local/bin/python2.4
# SOAP Struct

def hello():
    return {"value":"Hello, world"}

def echo(**kw):
    return kw

def sum(**kw):
    sum = 0
    for i in kw.values(): sum += i
    return {"value":sum}

def average(**kw):
    d = sum(**kw)
    return d["value"] = d["value"]/len(kw)

from ZSI import dispatch
dispatch.AsCGI(rpc=True)
```

2.1.2 low level soap processing example

We will now show a more complete example of a robust web service implemented at the SOAP layer. It takes as input a player name and array of integers, and returns the average. It is presented in sections, following the steps detailed above. A complete working example of this service is available in *Appendix C*.

The first section reads in a request, and parses the SOAP header.

```

from ZSI import *
import sys
IN, OUT = sys.stdin, sys.stdout
try:
    ps = ParsedSoap(IN)
except ParseException, e:
    OUT.write(FaultFromZSIException(e).AsSOAP())
    sys.exit(1)
except Exception, e:
    # Faulted while processing; we assume it's in the header.
    OUT.write(FaultFromException(e, 1).AsSOAP())
    sys.exit(1)

# We are not prepared to handle any actors or mustUnderstand elements,
# so we'll arbitrarily fault back with the first one we found.
a = ps.WhatActorsArePresent()
if len(a):
    OUT.write(FaultFromActor(a[0]).AsSOAP())
    sys.exit(1)
mu = ps.WhatMustIUnderstand()
if len(mu):
    uri, localname = mu[0]
    OUT.write(FaultFromNotUnderstood(uri, localname).AsSOAP())
    sys.exit(1)

```

This section defines the mappings between Python objects and the SOAP data being transmitted. Recall that according to the SOAP specification, RPC input and output are modeled as a structure.

```

class Player:
    def __init__(self, *args):
        if not len(args): return
        self.Name = args[0]
        self.Scores = args[1:]
Player.typecode = TC.Struct(Player, [
    TC.String('Name'),
    TC.Array('Integer', TC.Integer(), 'Scores', undeclared=True),
    ], 'GetAverage')

class Average:
    def __init__(self, average=None):
        self.average = average
Average.typecode = TC.Struct(Average, [
    TC.Integer('average'),
    ], 'GetAverageResponse')

```

This section parses the input, performs the application-level activity, and serializes the response.

```

try:
    player = ps.Parse(Player.typecode)
except EvaluateException, e:
    OUT.write(FaultFromZSIException(e).AsSOAP())
    sys.exit(1)

try:
    total = 0
    for value in player.Scores: total = total + value
    result = Average(total / len(player.Scores))
    sw = SoapWriter()
    sw.serialize(result, Average.typecode)
    sw.close()
    OUT.write(str(sw))
except Exception, e:
    OUT.write(FaultFromException(e, 0, sys.exc_info()[2]).AsSOAP())
    sys.exit(1)

```

In the `serialize()` call above, the second parameter is optional, since `result` is an instance of the `Average` class, and the `Average.typecode` attribute is the typecode for class instances.

2.1.3 A `mod_python` example

The Apache module `mod_python` (see <http://www.modpython.org>) embeds Python within the Apache server. In order to expose operations within a module via `mod_python`, use the `dispatch.AsHandler()` function. The `dispatch.AsHandler()` function will dispatch requests to any operation defined in the module you pass it, which allows for multiple operations to be defined in a module. The only trick is to use `__import__` to load the XML encodings your service expects. This is a required workaround to avoid the pitfalls of restricted execution with respect to XML parsing.

The following is a complete example of a simple handler. The soap operations are implemented in the `MyHandler` module:

```

def hello():
    return {"value":"Hello, world"}

def echo(**kw):
    return kw

def sum(**kw):
    sum = 0
    for i in kw.values(): sum += i
    return {"value":sum}

def average(**kw):
    d = sum(**kw)
    d["value"] = d["value"]/len(kw)
    return d

```

Dispatching from within `mod_python` is achieved by passing the aforementioned `MyHandler` module to `dispatch.AsHandler()`. The following code exposes the operations defined in `MyHandler` via SOAP:

```

from ZSI import dispatch
from mod_python import apache

import MyHandler
mod = __import__('encodings.utf_8', globals(), locals(), '*')
mod = __import__('encodings.utf_16_be', globals(), locals(), '*')

def handler(req):
    dispatch.AsHandler(modules=(MyHandler,), request=req, rpc=True)
    return apache.OK

```

2.2 Client Side Examples

2.2.1 Simple Example

ZSI provides two ways for a client to interactive with a server: the `Binding` or `NamedParamBinding` class and the `ServiceProxy` class. The first is useful when the operations to be invoked are not defined in WSDL or when only simple Python datatypes are used; the `ServiceProxy` class can be used to parse WSDL definitions in order to determine how to serialize and parse the SOAP messages.

During development, it is often useful to record “packet traces” of the SOAP messages being exchanged. Both the `Binding` and `ServiceProxy` classes provide a `tracefile` parameter to specify an output stream (such as a file) to capture messages. It can be particularly useful when debugging unexpected SOAP faults.

The first example provided below demonstrates how to use the `NamedParamBinding` class to connect to a remote service and perform an operation.

```

#!/usr/bin/env python
import sys,time
from ZSI.client import NamedParamBinding as NPBinding

b = NPBinding(url='http://127.0.0.1/cgi-bin/soapstruct', tracefile=sys.stdout)
print "Hello: ", b.hello()
print "Echo: ", b.echo(name="josh", year=2006, pi=3.14, time=time.gmtime())
print "Sum: ", b.sum(one=1, two=2, three=3)
print "Average: ", b.average(one=100, two=200, three=300, four=400)

```

2.2.2 Complex Example: pickler.py

If the operation invoked returns a `ComplexType`, typecode information must be provided in order to tell ZSI how to deserialize the response. Here is a sample server-side implementation (for the complete example see *Appendix D*):

```

class Person:
    def __init__(self, name=None, age=0):
        self.name = name
        self.age = age

Person.typecode = TC.Struct(Person,
    [TC.String('name'),
     TC.InonNegativeInteger('age')],
    'myApp:Person')

# my web service that returns a complex structure
def getPerson(name):
    fp = open('%s.person.pickle', % name, 'r')
    return pickle.load(fp)

# my web service that accepts a complex structure
def savePerson(person):
    fp = open('%s.person.pickle' % person.name, 'w')
    pickle(person, fp)
    fp.close()

```

In order for ZSI to transparently deserialize the returned complex type into a `Person` instance, a module defining the class and its typecode can be passed into the `Binding`. It is also possible to explicitly tell ZSI what typecode to use by passing it as a parameter to the `Binding.Receive()` method.

The following fragment shows both styles:

```

import sys
from ZSI.client import Binding
from MyComplexTypes import Person

b = Binding(url='http://localhost/test3/pickler.py', tracefile=sys.stdout)
person = Person('christopher', 26)
rsp = b.savePerson(person)

```

Because the returned complex type is defined in a class present in *typesmodule*, transparent deserialization is possible. When sending complex types to the server, it is not necessary to list the module in *typesmodule*:

```

import sys
import MyComplexTypes
from ZSI.client import NamedParamBinding as NPBinding, Binding
from ZSI import TC

kw = {'url':'http://localhost/test3/pickler.py', 'tracefile':sys.stdout}
b = NPBinding(**kw)
rsp = b.getPerson(name='christopher')
assert type(rsp) is dict, 'expecting a dict'
assert rsp['Person']['name'] == 'christopher', 'wrong person'

b = NPBinding(typesmodule=MyComplexTypes, **kw)
rsp = b.getPerson(name='christopher')
assert isinstance(rsp['Person'], MyComplexTypes.Person), (
    'expecting instance of %s' %MyComplexTypes.Person)

b = Binding(typesmodule=MyComplexTypes, **kw)
class Name(str):
    typecode = TC.String("name")

rsp = b.getPerson(Name('christopher'))
assert isinstance(rsp['Person'], MyComplexTypes.Person), (
    'expecting instance of %s' %MyComplexTypes.Person)

```


Exceptions

exception ZSIException

Base class for all ZSI Exceptions, it is a subtype of the Python `Exception` class.

exception ParseException

ZSI can raise this exception while creating a `ParsedSoap` object. It is a subtype of the `ZSIException` class. The string form of a `ParseException` object consists of a line of human-readable text. If the backtrace is available, it will be concatenated as a second line.

The following attributes are read-only:

inheader

A boolean that indicates if the error was detected in the SOAP Header element.

str

A text string describing the error.

trace

A text string containing a backtrace to the error. This may be `None` if it was not possible, such as when there was a general DOM exception, or when the `str` text is believed to be sufficient.

exception EvaluateException

This exception is similar to `ParseException`, except that ZSI may raise it while converting between SOAP and local Python objects.

The following attributes are read-only:

str

A text string describing the error.

trace

A text backtrace, as described above for `ParseException`.

Utilities

ZSI defines some utility methods that general applications may want to use.

Version()

Returns a three-element tuple containing the numbers representing the major, minor, and release identifying the ZSI version. New in version 1.1.

4.1 Low-Level Utilities

ZSI also defines some low-level utilities for its own use that start with a leading underscore and must be imported explicitly. They are documented here because they can be useful for developing new typecode classes.

These functions are mostly used in in parse methods and the `ParsedSoap` class. The serialization routines use the `ElementProxy` class to encapsulate common DOM-level operations.

Some `lambda`'s are defined so that some DOM accessors will return an empty list rather than `None`. This means that rather than writing:

```
if elt.childNodes:
    for N in elt.childNodes:
        ...
```

One can write:

```
for N in _children(elt):
    ...
```

Other `lambda`'s return SOAP-related attributes from an element, or `None` if not present.

attrs (*element*)

Returns a list of all attributes of the specified `element`.

backtrace (*elt, dom*)

This function returns a text string that traces a “path” from `dom`, a DOM root, to `elt`, an element within that document, in XPath syntax.

child_elements (*element*)

Returns a list of all children elements of the specified `element`.

children (*element*)

Returns a list of all children of the specified `element`.

copyright_empty_nsuri_list

find_arraytype (*element*)

The value of the SOAP `arrayType` attribute. New in version 1.2.

`_find_attr` (*element*, *name*)

The value of the unqualified `name` attribute.

`_find_attrNS` (*element*, *namespaceURI*, *localName*)

The value of a `name` attribute in a namespace `namespaceURI`.

`_find_attrNodeNS` (*element*, *namespaceURI*, *localName*)

Works just like `_find_attrNS`, but this function grabs the attribute `Node` to distinguish between an unspecified attribute (`None`) and one set to empty string (`""`).

`_find_default_namespace` (*element*)

Returns the value of the default namespace.

`_find_encstyle` (*element*)

The value of the SOAP `encodingStyle` attribute.

`_find_href` (*element*)

The value of the unqualified `href` attribute.

`_find_type` (*element*)

The value of the XML Schema `type` attribute.

`_find_xmlns_prefix` (*element*, *prefix*)

The value of the `xmlns:prefix` type attribute.

`_find_xsi_attr` (*element*, *attribute*)

Find the attribute in any of the XMLSchema namespaces.

`_get_element_nsuri_name` (*element*)

Returns a (`namespace`, `name`) tuple representing the element tag.

`_get_idstr` (*obj*)

Substitute for `id` function. Python 2.3.x generates a `FutureWarning` for negative IDs, so we use a different prefix character to ensure uniqueness, and call `abs()` to avoid the warning.

`_get_postvalue_from_absoluteURI` (*url*)

Returns POST value from `url`, and caches these values.

`_resolve_prefix` (*element*, *prefix*)

resolve `prefix` to a `namespaceURI`. If `None` or empty `str`, return default namespace or `None` if not defined.

`_valid_encoding` (*elt*)

Return true if the element `elt` has a SOAP encoding that can be handled by ZSI (currently Section 5 of the SOAP 1.1 specification or an empty encoding for XML).

The `ParsedSoap` module — basic message handling

This class represents an input stream that has been parsed as a SOAP message.

class `ParsedSoap` (*input* [, ***keywords*])

Creates a `ParsedSoap` object from the provided input source. If *input* is not a string, then it must be an object with a `read()` method that supports the standard Python “file read” semantics.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|---------|---|
| <code>envelope</code> | True | expect a SOAP Envelope |
| <code>keepdom</code> | False | Do not release the DOM when this object is destroyed. To access the DOM object, use the <code>GetDomAndReader()</code> method. The reader object is necessary to properly free the DOM structure using <code>reader.releaseNode(dom)</code> . New in version 1.2. |
| <code>readerclass</code> | None | Class used to create DOM-creating XML readers; described below. New in version 1.2. |
| <code>resolver</code> | None | Value for the <code>resolver</code> attribute; see below. |
| <code>trailers</code> | False | Allow trailing data elements to appear after the <code>Body</code> . |

The following attributes of a `ParsedSoap` are read-only:

body

The root of the SOAP `Body` element. Using the `GetElementNSdict()` method on this attribute can be useful to get a dictionary to be used with the `SoapWriter` class.

body_root

The element that contains the SOAP serialization root; that is, the element in the SOAP `Body` that “starts off” the data.

data_elements

A (possibly empty) list of all child elements of the `Body` other than the root.

header

The root of the SOAP `Header` element. Using the `GetElementNSdict()` method on this attribute can be useful to get a dictionary to be used with the `SoapWriter` class.

header_elements

A (possibly empty) list of all elements in the SOAP `Header`.

trailer_elements

Returns a (possibly empty) list of all elements following the `Body`. If the `trailers` keyword was not used when the object was constructed, this attribute will not be instantiated and retrieving it will raise an exception.

The following attribute may be modified:

resolver

If not `None`, this attribute can be invoked to handle absolute `href`'s in the SOAP data. It will be invoked as follows:

resolver (*uri, tc, ps, **keywords*)

The `uri` parameter is the URI to resolve. The `tc` parameter is the typecode that needs to resolve `href`; this may be needed to properly interpret the content of a MIME bodypart, for example. The `ps` parameter is the `ParsedSoap` object that is invoking the resolution (this allows a single resolver instance to handle multiple SOAP parsers).

Failure to resolve the URI should result in an exception being raised. If there is no content, return `None`; this is not the same as an empty string. If there is content, the data returned should be in a form understandable by the typecode.

The following methods are available:

Backtrace (*elt*)

Returns a human-readable “trace” from the document root to the specified element.

FindLocalHREF (*href, elt*)

Returns the element that has an `id` attribute whose value is specified by the `href` fragment identifier. The `href` *must* be a fragment reference — that is, it must start with a pound sign. This method raises an `EvaluateException` exception if the element isn't found. It is mainly for use by the parsing methods in the `TypeCode` module.

GetElementNSdict (*elt*)

Return a dictionary for all the namespace entries active at the current element. Each dictionary key will be the prefix and the value will be the namespace URI.

GetMyHeaderElements (*[actorlist=None]*)

Returns a list of all elements in the `Header` that are intended for *this* SOAP processor. This includes all elements that either have no SOAP `actor` attribute, or whose value is either the special “next actor” value or in the `actorlist` list of URI's.

GetDomAndReader ()

Returns a tuple containing the dom and reader objects, (`dom, reader`). Unless `keepdom` is true, the dom and reader objects will go out of scope when the `ParsedSoap` instance is deleted. If `keepdom` is true, the reader object is needed to properly clean up the dom tree with `reader.releaseNode(dom)`.

IsAFault ()

Returns true if the message is a SOAP fault.

Parse (*how*)

Parses the SOAP `Body` according to the `how` parameter, and returns a Python object. If `how` is not a `TC.TypeCode` object, then it should be a Python class object that has a `typecode` attribute.

ResolveHREF (*uri, tc[, **keywords]*)

This method is invoked to resolve an absolute URI. If the typecode `tc` has a `resolver` attribute, it will use it to resolve the URI specified in the `uri` parameter, otherwise it will use its own `resolver`, or raise an `EvaluateException` exception.

Any keyword parameters will be passed to the chosen resolver. If no content is available, it will return `None`. If unable to resolve the URI it will raise an `EvaluateException` exception. Otherwise, the resolver should return data in a form acceptable to the specified typecode, `tc`. (This will almost always be a file-like object holding opaque data; for XML, it may be a DOM tree.)

WhatActorsArePresent ()

Returns a list of the values of all the SOAP `actor` attributes found in child elements of the SOAP `Header`.

WhatMustIUnderstand ()

Returns a list of `(uri, localname)` tuples for all elements in the SOAP `Header` that have the SOAP `mustUnderstand` attribute set to a non-zero value.

ZSI supports multiple DOM implementations. The `readerclass` parameter specifies which one to use. The default is to use `minidom`, provided through the `DefaultReader` class in the `ZSI.parse` module.

The specified reader class must support the following methods:

fromString (*string*)

Return a DOM object from a string.

fromStream (*stream*)

Return a DOM object from a file-like stream.

releaseNode (*dom*)

Free the specified DOM object.

The DOM object must support the standard Python mapping of the DOM Level 2 specification. While only a small subset of specification is used, the particular methods and attributes used by ZSI are available only by inspecting the source.

To use the `cDomlette` DOM provided by the `4Suite` package, use the `NonvalidatingReader` class in the `Ft.Xml.Domlette` module. Due to name changes in the 1.0 version of `4Suite`, a simple adapter class is required to use this DOM implementation.

```
from 4Suite.Xml.Domlette import NonvalidatingReaderBase

class 4SuiteAdapterReader(NonvalidatingReaderBase):

    def fromString(self, str):
        return self.parseString(str)

    def fromStream(self, stream):
        return self.parseStream(stream)

    def releaseNode(self, node):
        pass
```

The `TypeCode` classes — data conversions

The `TypeCode` module defines classes used for converting data between SOAP data and local Python objects. Python numeric and string types, and sequences and dictionaries, are supported by ZSI. The `TC.TypeCode` class is the parent class of all datatypes understood by ZSI.

All typecodes classes have the prefix `TC.`, to avoid name clashes.

ZSI provides fine-grain control over the names used when parsing and serializing XML into local Python objects, through the use of two attributes: the `pname`, the `aname`. The `pname` specifies the name expected on the XML element being parsed and the name to use for the output element when serializing. The `aname` is the name to use for the analogous attribute in the local Python object.

The `pname` is the parameter name. It specifies the incoming XML element name and the default values for the Python attribute and serialized names. All typecodes take the `pname` argument. This name can be specified as either a list or a string. When specified as a list, it must have two elements which are interpreted as a “(namespace-URI, localname)” pair. If specified this way, both the namespace and the local element name must match for the parse to succeed. For the Python attribute, and when generating output, only the “localname” is used. If a namespace-URI is specified then the full qualified name is used for output, and it is required for input; this *requires* the namespace prefix to be specified.

The `aname` is the attribute name. This parameter overrides any value implied by the `pname`. Typecodes nested in a `TC.Struct` or `TC.ComplexType` can use this parameter to specify the tag, dictionary key, or instance attribute to set.

The `nsdict` parameter to the `SoapWriter` construct can be used to specify prefix to namespace-URI mappings, these are otherwise handled automatically.

6.1 `TC.TypeCode`

The `TypeCode` class is the parent class of all typecodes.

```
class TypeCode (**keywords)
```

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|-----------------------|---|
| <code>pname</code> | <code>None</code> | parameter name of the object |
| <code>aname</code> | <code>None</code> | attribute name of the object |
| <code>minOccurs</code> | <code>1</code> | schema facet minimum occurrences |
| <code>maxOccurs</code> | <code>1</code> | schema facet maximum occurrences |
| <code>nillable</code> | <code>False</code> | schema facet is this nillable (<code>xsi:nil="true"</code>) |
| <code>typed</code> | <code>True</code> | Output type information (in the <code>xsi:type</code> attribute) when serializing. By special dispensation, typecodes within a <code>TC.Struct</code> object inherit this from the container. |
| <code>unique</code> | <code>0</code> | If true, the object is unique and will never be “aliased” with another object, so the <code>id</code> attribute need not be output. |
| <code>pyclass</code> | <code>None</code> | when parsing data, instances of this class can be created to store the data. Default behavior is reflective of specific <code>TypeCode</code> classes. |
| <code>attrs_aname</code> | <code>'_attrs'</code> | attribute name of the object where attribute values are stored. Used for serialization and parsing. |

Optional elements are those which do not have to be an incoming message, or which have the XML Schema `nil` attribute set. When parsing the message as part of a `Struct`, then the Python instance attribute will not be set, or the element will not appear as a dictionary key. When being parsed as a simple type, the value `None` is returned. When serializing an optional element, a non-existent attribute, or a value of `None` is taken to mean not present, and the element is skipped.

typechecks

This is a class attribute. If true (the default) then all typecode constructors do more rigorous type-checking on their parameters.

tag

This is a class attribute. Specifies the global element declaration this typecode represents, the value is a `(namespace, name)` tuple.

type

This is a class attribute. Specifies the global type definition this typecode represents, the value is a `(namespace, name)` tuple.

attribute_typecode_dict

This is a class attribute. This is a dict of `(URI, NCName)` tuple keys, the values of each is a typecode. This is how attribute declarations other than SOAP and XMLSchema attribute declarations (eg. `xsi:type`, `id`, `href`, etc) are represented.

logger

This is a class attribute. logger instance for this class.

The following methods are useful for defining new typecode classes; see the section on dynamic typing for more details. In all of the following, the `ps` parameter is a `ParsedSoap` object.

checkname (*elt, ps*)

Checks if the name and type of the element `elt` are correct and raises a `EvaluateException` if not. Returns the element's type as a `(uri, localname)` tuple if so.

checktype (*elt, ps*)

Like `checkname()` except that the element name is ignored. This method is actually invoked by `checkname()` to do the second half of its processing, but is useful to invoke directly, such as when resolving multi-reference data.

nilled (*elt, ps*)

If the element `elt` has data, this returns `False`. If it has no data, and the typecode is not optional, an `EvaluateException` is raised; if it is optional, a `True` is returned.

simple_value (*elt, ps, mixed=False*)

Returns the text content of the element `elt`. If no value is present, or the element has non-text chil-

dren, an `EvaluateException` is raised. If `mixed` is `False` if child elements are discovered an `EvaluateException` is raised, else join all text nodes and return the result.

6.2 TC.Any — the basis of dynamic typing

SOAP provides a flexible set of serialization rules, ranging from completely self-describing to completely opaque, requiring an external schema. For example, the following are all possible ways of encoding an integer element `i` with a value of 12:

6.2.1 simple data

– requires type information

```
<tns:i xsi:type="SOAP-ENC:integer">12</tns:i>
<tns:i xsi:type="xsd:nonNegativeInteger">12</tns:i>
<SOAP-ENC:integer>12</SOAP-ENC:integer>
<tns:i>12</tns:i>
```

The first three lines are examples of *typed* elements. If `ZSI` is asked to parse any of the above examples, and a `TC.Any` typecode is given, it will properly create a Python integer for the first three, and raise a `EvaluateException` for the fourth.

6.2.2 compound data

– Struct or Array Compound data, such as a `struct`, may also be self-describing (namespace are omitted for clarity):

```
<tns:foo>
  <tns:i xsi:type="SOAP-ENC:integer">12</tns:i>
  <tns:name xsi:type="SOAP-ENC:string">Hello world</tns:name>
</tns:foo>
```

If this is parsed with a `TC.Any` typecode, either a Python `dict` is created or if `aslist` is `True` a list:

```
ps = ParsedSoap(xml, envelope=False)
print ps.Parse(TC.Any())
{ 'name': u'Hello world', 'i': 12 }

print ps.Parse(TC.Any(aslist=True))
[ 12, u'Hello world' ]
```

Note that one preserves order, while the other preserves the element names.

6.2.3 class description

class Any (*name* [, ***keywords*])

Used for parsing incoming SOAP data (that is typed), and serializing outgoing Python data.

The following keyword arguments may be used:

| Keyword | Default | Description |
|---------------------|--------------------|--|
| <code>aslist</code> | <code>False</code> | If true, then the data is (recursively) treated as a list of values. The default is a Python dictionary, which preserves parameter names but loses the ordering. New in version 1.1. |

In addition, if the Python object being serialized with an `Any` has a `typecode` attribute, then the `serialize` method of the typecode will be invoked to do the serialization. This allows objects to override the default dynamic serialization.

Referring back to the compound XML data above, it is possible to create a new typecode capable of parsing elements of type `mytype`. This class would know that the `i` element is an integer, so that the explicit typing becomes optional, rather than required.

6.2.4 Adding new types

Most of the `TypeCode` classes in TC are registered with `Any`, making an instance of itself available for dynamic typing. New `TypeCode` classes can be created and registered with `Any` by using `RegisterType`. In order to override an existing entry in the registry call `RegisterType` with `clobber=True`. The serialization entries are mappings between builtin Python types and a `TypeCode` instance, it is not possible to have one Python type map to multiple typecodes. The parsing entries are mappings between `(namespaceURI, name)` tuples, representing the `xsi:type` attribute, and a `TypeCode` instance. Thus, only one instance of a `TypeCode` class can represent a XML Schema type. So this mechanism is not appropriate for representing XML Schema element information.

class `NEWTYPESCODE (TypeCode)` (...)

The new typecode should be derived from the `TC.TypeCode` class, and `TypeCode.__init__()` must be invoked in the new class's constructor.

`parselist`

This is a class attribute, used when parsing incoming SOAP data. It should be a sequence of `(uri, localname)` tuples to identify the datatype. If `uri` is `None`, it is taken to mean either the XML Schema namespace or the SOAP encoding namespace; this should only be used if adding support for additional primitive types. If this list is empty, then the type of the incoming SOAP data is assumed to be correct; an empty list also means that incoming typed data cannot be dynamically parsed.

`errorlist`

This is a class attribute, used when reporting a parsing error. It is a text string naming the datatype that was expected. If not defined, ZSI will create this attribute from the `parselist` attribute when it is needed.

`seriallist`

This is a class attribute, used when serializing Python objects dynamically. It specifies what types of object instances (or Python types) this typecode can serialize. It should be a sequence, where each element is a Python class object, a string naming the class, or a type object from Python's `types` module (if the new typecode is serializing a built-in Python type).

`parse (elt, ps)`

ZSI invokes this method to parse the `elt` element and return its Python value. The `ps` parameter is the `ParsedSoap` object, and can be used for dereferencing `href`'s, calling `Backtrace()` to report errors, etc.

`serialize (sw, pyobj[, **keywords])`

ZSI invokes this method to output a Python object to a SOAP stream. The `sw` parameter will be a `SoapWriter` object, and the `pyobj` parameter is the Python object to serialize.

The following keyword arguments may be used:

| Keyword | Default | Description |
|-----------------------|---------------------|--|
| <code>attrtext</code> | None | Text (with leading space) to output as an attribute; this is normally used by the <code>TC.Array</code> class to pass down indexing information. |
| <code>name</code> | None | Name to use for serialization; defaults to the name specified in the typecode, or a generated name. |
| <code>typed</code> | <i>per-typecode</i> | Whether or not to output type information; the default is to use the value in the typecode. |

Once the new typecode class has been defined, it should be registered with ZSI's dynamic type system by invoking the following function:

RegisterType (*class* [, *clobber=0* [, ***keywords*]])

By default, it is an error to replace an existing type registration, and an exception will be raised. The `clobber` parameter may be given to allow replacement. A single instance of the `class` object will be created, and the `keyword` parameters are passed to the constructor.

If the class is not registered, then instances of the class cannot be processed as dynamic types. This may be acceptable in some environments.

6.3 TC.SimpleType

Parent class of all simple types.

empty_content

This is a class attribute. Value returned when tag or node is present, is not nilled, and without text content.

6.4 Strings

SOAP/XMLSchema Strings are Python strings.

class String (*name* [, ***keywords*])

The parent type of all strings.

The following keyword arguments may be used:

| Keyword | Default | Description |
|-----------------------|---------|--|
| <code>resolver</code> | None | A function that can resolve an absolute URI and return its content as a string, as described in the <code>ParsedSoap</code> description. |
| <code>strip</code> | True | If true, leading and trailing whitespace are stripped from the content. |

class Enumeration (*value_list*, *name* [, ***keywords*])

Like `TC.String`, but the value must be a member of the `choices` sequence of text strings

In addition to `TC.String`, the basic string, several subtypes are provided that transparently handle common encodings. These classes create a temporary string object and pass that to the `serialize()` method. When doing RPC encoding, and checking for non-unique strings, the `TC.String` class must have the original Python string, as well as the new output. This is done by adding a parameter to the `serialize()` method:

| Keyword | Default | Description |
|-------------------|---------|--|
| <code>orig</code> | None | If deriving a new typecode from the string class, and the derivation creates a temporary Python string (such as by <code>Base64String</code>), then this parameter is the original string being serialized. |

class Base64String (*name* [, ***keywords*])

The value is encoded in Base-64.

class HexBinaryString (*name* [, ***keywords*])

Each byte is encoded as its printable version.

class URI (*name* [, ***keywords*])

The value is URL quoted (e.g., %20 for the space character).

It is often the case that a parameter will be typed as a string for transport purposes, but will in fact have special syntax and processing requirements. For example, a string could be used for an XPath expression, but it is more convenient for the Python value to actually be the compiled expression. Here is how to do that:

```
import xml.xpath.pyxpath
import xml.xpath.pyxpath.Compile as _xpath_compile

class XPathString(TC.String):
    def __init__(self, name, **kw):
        TC.String.__init__(self, name, **kw)

    def parse(self, elt, ps):
        val = TC.String.parse(self, elt, ps)
        try:
            val = _xpath_compile(val)
        except:
            raise EvaluateException("Invalid XPath expression",
                                    ps.Backtrace(elt))
        return val
```

In particular, it is common to send XML as a string, using entity encoding to protect the ampersand and less-than characters.

class XMLString (*name* [, ***keywords*])

Parses the data as a string, but returns an XML DOM object. For serialization, takes an XML DOM (or element node), and outputs it as a string.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|---------|---|
| <code>readerclass</code> | None | Class used to create DOM-creating XML readers; described in the ParsedSoap chapter. |

6.5 Integers

SOAP/XMLSchema integers are Python integers.

class Integer ([***keywords*])

The parent type of all integers. This class handles any of the several types (and ranges) of SOAP integers.

The following keyword arguments may be used:

| Keyword | Default | Description |
|---------------------|-----------------|--|
| <code>format</code> | <code>%d</code> | Format string for serializing. New in version 1.2. |

class IEnumeration (*choices* [, ***keywords*])

Like `TC.Integer`, but the value must be a member of the `choices` sequence.

A number of sub-classes are defined to handle smaller-ranged numbers.

class Ibyte ([***keywords*])

A signed eight-bit value.

class `UnsignedByte` (`**keywords`)
 An unsigned eight-bit value.

class `Ishort` (`**keywords`)
 A signed 16-bit value.

class `UnsignedShort` (`**keywords`)
 An unsigned 16-bit value.

class `Iint` (`**keywords`)
 A signed 32-bit value.

class `UnsignedInt` (`**keywords`)
 An unsigned 32-bit value.

class `Ilong` (`**keywords`)
 A signed 64-bit value.

class `UnsignedLong` (`**keywords`)
 An unsigned 64-bit value.

class `IpositiveInteger` (`**keywords`)
 A value greater than zero.

class `InegativeInteger` (`**keywords`)
 A value less than zero.

class `InonPositiveInteger` (`**keywords`)
 A value less than or equal to zero.

class `InonNegativeInteger` (`**keywords`)
 A value greater than or equal to zero.

6.6 Floating-point Numbers

SOAP/XMLSchema floating point numbers are Python floats.

class `Decimal` (`**keywords`)
 The parent type of all floating point numbers. This class handles any of the several types (and ranges) of SOAP floating point numbers.

The following keyword arguments may be used:

| Keyword | Default | Description |
|---------------------|-----------------|--|
| <code>format</code> | <code>%f</code> | Format string for serializing. New in version 1.2. |

class `FPenumeration` (`value_list`, `name`, `**keywords`)
 Like `TC.Decimal`, but the value must be a member of the `value_list` sequence. Be careful of round-off errors if using this class.

Two sub-classes are defined to handle smaller-ranged numbers.

class `FPfloat` (`name`, `**keywords`)
 An IEEE single-precision 32-bit floating point value.

class `FPdouble` (`name`, `**keywords`)
 An IEEE double-precision 64-bit floating point value.

6.7 Dates and Times

SOAP dates and times are Python time tuples in UTC (GMT), as documented in the Python `time` module. Time is tricky, and processing anything other than a simple absolute time can be difficult. (Even then, timezones lie in wait to trip up the unwary.) A few caveats are in order:

1. Some date and time formats will be parsed into tuples that are not valid time values. For example, 75 minutes is a valid duration, although not a legal value for the minutes element of a time tuple.
2. Fractional parts of a second may be lost when parsing, and may have extra trailing zero's when serializing.
3. Badly-formed time tuples may result in non-sensical values being serialized; the first six values are taken directly as year, month, day, hour, minute, second in UTC.
4. Although the classes `Duration` and `Gregorian` are defined, they are for internal use only and should not be included in any `TypeCode` you define. Instead, use the classes beginning with a lower case `g` in your typecodes.

In addition, badly-formed values may result in non-sensical serializations.

When serializing, an integral or floating point number is taken as the number of seconds since the epoch, in UTC.

class `Duration` (`**keywords`)

A relative time period. Negative durations have all values less than zero; this makes it easy to add a duration to a Python time tuple.

class `Gregorian` (`**keywords`)

An absolute time period. This class should not be instantiated directly; use one of the `gXXX` classes instead.

class `gDateTime` (`**keywords`)

A date and time.

class `gDate` (`**keywords`)

A date.

class `gYearMonth` (`**keywords`)

A year and month.

class `gYear` (`**keywords`)

A year.

class `gMonthDay` (`**keywords`)

A month and day.

class `gDay` (`**keywords`)

A day.

class `gTime` (`**keywords`)

A time.

6.8 Boolean

SOAP Booleans are Python integers.

class `Boolean` (`**keywords`)

When marshaling zero or the word “false” is returned as 0 and any non-zero value or the word “true” is returned as 1. When serializing, the number 0 or 1 will be generated.

6.9 XML

XML is a Python DOM element node. If the value to be serialized is a Python string, then an href is generated, with the value used as the URI. This can be used, for example, when generating SOAP with attachments. Otherwise, the XML is typically put inside a wrapper element that sets the proper SOAP encoding style.

For efficiency, incoming XML is returned as a “pointer” into the DOM tree maintained within the `ParsedSoap` object. If that object is going to go out of scope, the data will be destroyed and any XML objects will become empty elements. The class instance variable `copyit`, if non-zero indicates that a deep copy of the XML subtree will be made and returned as the value. Note that it is generally more efficient to keep the `ParsedSoap` object alive until the XML data is no longer needed.

class XML (`[**keywords]`)

This typecode represents a portion of an XML document embedded in a SOAP message. The value is the element node.

The following keyword arguments may be used:

| Keyword | Default | Description |
|-----------------------|----------------------------|---|
| <code>copyit</code> | <code>TC.XML.copyit</code> | Return a copy of the parsed data. |
| <code>comments</code> | <code>0</code> | Preserve comments in output. |
| <code>inline</code> | <code>0</code> | The XML sub-tree is single-reference, so can be output in-place. |
| <code>resolver</code> | <code>None</code> | A function that can resolve an absolute URI and return its content as an element node, as described in the <code>ParsedSoap</code> description. |
| <code>wrapped</code> | <code>1</code> | If zero, the XML is output directly, and not within a SOAP wrapper element. New in version 1.2. |

When serializing, it may be necessary to specify which namespace prefixes are “active” in the XML. This is done by using the `unsuppressedPrefixes` parameter when calling the `serialize()` method. (This will only work when XML is the top-level item being serialized, such as when using typecodes and document-style interfaces.)

| Keyword | Default | Description |
|-----------------------------------|-----------------|---|
| <code>unsuppressedPrefixes</code> | <code>[]</code> | An array of strings identifying the namespace prefixes that should be output. |

6.10 ComplexType

Represents the XMLSchema `ComplexType`. New in version 2.0.

class ComplexType (`pyclass, ofwhat[, **keywords]`)

This class defines a compound data structure. If `pyclass` is `None`, then the data will be marshaled into a Python dictionary, and each item in the `ofwhat` sequence specifies a (possible) dictionary entry. Otherwise, `pyclass` must be a Python class object. The data is then marshaled into the object, and each item in the `ofwhat` sequence specifies an attribute of the instance to set.

Note that each typecode in `ofwhat` must have a name.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|----------------------|---|
| <code>inorder</code> | False | Items within the structure must appear in the order specified in the <code>ofwhat</code> sequence. |
| <code>inline</code> | False | The structure is single-reference, so ZSI does not have to use <code>href/id</code> encodings. |
| <code>mutable</code> | False | If an object is going to be serialized multiple times, and its state may be modified between serializations, then this keyword should be used, otherwise a single instance will be serialized, with multiple references to it. This argument implies the <code>inline</code> argument. New in version 1.2. |
| <code>type</code> | None | A <code>(uri, localname)</code> tuple that defines the type of the structure. If present, and if the input data has a <code>xsi:type</code> attribute, then the namespace-qualified value of that attribute must match the value specified by this parameter. By default, type-checking is not done for structures; matching child element names is usually sufficient and senders rarely provide type information. |
| <code>mixed</code> | False | using a mixed content model, allow text and element content. |
| <code>mixed_aname</code> | <code>'_text'</code> | if <code>mixed</code> is True, text content is set in this attribute (key). |

If the `typed` keyword is used, then its value is assigned to all typecodes in the `ofwhat` parameter. If any of the typecodes in `ofwhat` are repeatable, then the `inorder` keyword should not be used and the `hasextras` parameter *must* be used.

For example, the following C structure:

```
struct foo {
    int i;
    char* text;
};
```

could be declared as follows:

```
class foo:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return str((self.name, self.i, self.text))

foo.typecode = TC.Struct(foo,
    ( TC.Integer('i'), TC.String('text') ),
    'foo')
```

6.11 Struct

SOAP Struct is a complex type for accessors identified by name. No element may have the same name as any other, nor may any element have a `maxOccurs` \leq 1. SOAP Structs are either Python dictionaries or instances of application-specified classes.

6.12 Arrays

SOAP arrays are Python lists; multi-dimensional arrays are lists of lists and are indistinguishable from a SOAP array of arrays. Arrays may be *sparse*, in which case each element in the array is a tuple of `(subscript, data)` pairs. If an array is not sparse, a specified *fill* element will be used for the missing values.

Currently only singly-dimensional arrays are supported.

class `Array` (*atype*, *ofwhat*[, ***keywords*])

The *atype* parameter is a (URI, NCName) tuple representing the SOAP array type. The *ofwhat* parameter is a typecode describing the array elements.

The following keyword arguments may be used:

| Keyword | Default | Description |
|-------------------------|----------------|--|
| <code>childnames</code> | None | Default name to use for the child elements. |
| <code>dimensions</code> | 1 | The number of dimensions in the array. |
| <code>fill</code> | None | The value to use when an array element is omitted. |
| <code>mutable</code> | False | If an object is going to be serialized multiple times, and its state may be modified between serializations, then this keyword should be used, otherwise a single instance will be serialized, with multiple references to it. |
| <code>nooffset</code> | 0 | Do not use the SOAP <code>offset</code> attribute so skip leading elements with the same value as <code>fill</code> . |
| <code>sparse</code> | False | The array is sparse. |
| <code>size</code> | None | An integer or list of integers that specifies the maximum array dimensions. |
| <code>undeclared</code> | False | The SOAP <code>'arrayType'</code> attribute need not appear. |

6.13 Apache Datatype

The Apache SOAP project, [urlhttp://xml.apache.org/soap/index.html](http://xml.apache.org/soap/index.html), has defined a popular SOAP datatype in the <http://xml.apache.org/xml-soap> namespace, a `Map`.

The `Map` type is encoded as a list of `item` elements. Each `item` has a `key` and `value` child element; these children must have SOAP type information. An Apache `Map` is either a Python dictionary or a list of two-element tuples.

class `Apache.Map` (*name*[, ***keywords*])

An Apache map. Note that the class name is dotted.

The following keyword arguments may be used:

| Keyword | Default | Description |
|---------------------|----------------|--|
| <code>aslist</code> | 0 | Use a list of tuples rather than a dictionary. |

The SoapWriter module — serializing data

The SoapWriter class is used to output SOAP messages. Note that its output is encoded as UTF-8; when transporting SOAP over HTTP it is therefore important to set the `charset` attribute of the `Content-Type` header.

The SoapWriter class reserves some namespace prefixes:

| Prefix | URI |
|----------|---|
| SOAP-ENV | http://schemas.xmlsoap.org/soap/envelope/ |
| SOAP-ENC | http://schemas.xmlsoap.org/soap/encoding/ |
| ZSI | http://www.zolera.com/schemas/ZSI/ |
| xsd | http://www.w3.org/2001/XMLSchema |
| xsi | http://www.w3.org/2001/XMLSchema-instance |

class SoapWriter (*optional**keywords*)

The following keyword arguments may be used:

| Keyword | Default | Description |
|----------------------------|---------------------------|---|
| <code>encodingStyle</code> | None | If not None, then use the specified value as the value for the SOAP <code>encodingStyle</code> attribute. New in version 1.2. |
| <code>envelope</code> | True | Create a SOAP Envelope. New in version 1.2. |
| <code>nsdict</code> | {} | Dictionary of namespaces to declare in the SOAP <code>Envelope</code> |
| <code>header</code> | True | create a SOAP <code>Header</code> element |
| <code>outputclass</code> | <code>ElementProxy</code> | wrapper around DOM or other XML library. |

Creating a SoapWriter object with `envelope` set to `False` results in an object that can be used for serializing objects into a string.

serialize (*pyobj*[, *typecode=None*[, *root=None*[, *header_pyobjs=None*[, ***keywords*]]]])

This method serializes the `pyobj` Python object as directed by the `typecode` `typecode` object. If `typecode` is omitted, then `pyobj` should be a Python object instance of a class that has a `typecode` attribute. It returns `self`, so that serializations can be chained together, or so that the `close()` method can be invoked. The `root` parameter may be used to explicitly indicate the root (main element) of a SOAP encoding, or indicate that the item is not the root. If specified, it should have the numeric value of zero or one. Any other keyword parameters are passed to the `typecode`'s `serialize` method.

close ()

Invokes all the callbacks, if any. The `close` operations can only happen once, if invoked a second time it will just return. This method will be invoked automatically if the object is deleted.

__str__ ()

Invokes the `close` method, and returns a string representation of the serialized object. Assumes that `serialize` has been invoked.

The following methods are primarily useful for those writing new typecodes.

AddCallback (*func, arg*)

Used by typecodes when serializing, allows them to add output after the SOAP Body is written but before the SOAP Envelope is closed. The function `func()` will be called with the `SoapWriter` object and the specified `arg` argument, which may be a tuple.

Forget (*obj*)

Forget that `obj` has been seen before. This is useful when repeatedly serializing a mutable object.

Known (*obj*)

If `obj` has been seen before (based on its Python `id`), return 1. Otherwise, remember `obj` and return 0.

ReservedNS (*prefix, uri*)

Returns true if the specified namespace `prefix` and `uri` collide with those used by the implementation.

writeNSDict (*nsdict*)

Outputs `nsdict` as a namespace dictionary. It is assumed that an XML start-element is pending on the output stream.

The Fault module — reporting errors

SOAP defines a *fault* message as the way for a recipient to indicate it was unable to process a message. The `ZSI Fault` class encapsulates this.

class `Fault` (*code*, *string* [, ***keywords*])

The *code* parameter is a text string identifying the SOAP fault code, a namespace-qualified name. The class attribute `Fault.Client` can be used to indicate a problem with an incoming message, `Fault.Server` can be used to indicate a problem occurred while processing the request, or `Fault.MU` can be used to indicate a problem with the SOAP `mustUnderstand` attribute. The *string* parameter is a human-readable text string describing the fault.

The following keyword arguments may be used:

| Keyword | Default | Description |
|---------------------|---------|--|
| <i>actor</i> | None | A string identifying the <code>actor</code> attribute that caused the problem (usually because it is unknown). |
| <i>detail</i> | None | A sequence of elements to output in the <code>detail</code> element; it may also be a text string, in which case it is output as-is, and should therefore be XML text. |
| <i>headerdetail</i> | None | Data, treated the same as the <code>detail</code> keyword, to be output in the SOAP header. See the following paragraph. |

If the fault occurred in the SOAP `Header`, the specification requires that the detail be sent back as an element within the SOAP `Header` element. Unfortunately, the SOAP specification does not describe how to encode this; `ZSI` defines and uses a `ZSI:detail` element, which is analogous to the SOAP `detail` element.

The following attributes are read-only:

actor

A text string holding the value of the SOAP `faultactor` element.

code

A text string holding the value of the SOAP `faultcode` element.

detail

A text string or sequence of elements containing holding the value of the SOAP `detail` element, when available.

headerdetail

A text string or sequence of elements containing holding the value of the `ZSI` header detail element, when available.

string

A text string holding the value of the SOAP `faultstring` element.

AsSOAP ([, ***kw*])

This method serializes the `Fault` object into a SOAP message. The message is returned as a string. Any

keyword arguments are passed to the `SoapWriter` constructor. New in version 1.1; the old `AsSoap()` method is still available.

If other data is going to be sent with the fault, the following two methods can be used. Because some data might need to be output in the `SOAP Header`, serializing a fault is a two-step process.

DataForSOAPHeader()

This method returns a text string that can be included as the `header` parameter for constructing a `SoapWriter` object.

serialize(*sw*)

This method outputs the fault object onto the `sw` object, which is a `SoapWriter` instance.

Some convenience functions are available to create a `Fault` from common conditions.

FaultFromActor(*uri*, *actor*)

This function could be used when an application receives a message that has a `SOAP Header` element directed to an actor that cannot be processed. The `uri` parameter identifies the actor. The `actor` parameter can be used to specify a URI that identifies the application, if it is not the ultimate recipient of the SOAP message.

FaultFromException(*ex*, *inheader*, *tb*, *actor*)

This function creates a `Fault` from a general Python exception. A SOAP “server” fault is created. The `ex` parameter should be the Python exception. The `inheader` parameter should be true if the error was found on a `SOAP Header` element. The optional `tb` parameter may be a Python traceback object, as returned by `'sys.exc_info()[2]'`. The `actor` parameter can be used to specify a URI that identifies the application, if it is not the ultimate recipient of the SOAP message.

FaultFromFaultMessage(*ps*)

This function creates a `Fault` from a `ParsedSoap` object passed in as `ps`. It should only be used if the `IsAFault()` method returned true.

FaultFromNotUnderstood(*uri*, *localname*, *actor*)

This function could be used when an application receives a message with the `SOAP mustUnderstand` attribute that it does not understand. The `uri` and `localname` parameters should identify the unknown element. The `actor` parameter can be used to specify a URI that identifies the application, if it is not the ultimate recipient of the SOAP message.

FaultFromZSIException(*ex*, *actor*)

This function creates a `Fault` object from a `ZSI` exception, `ParseException` or `EvaluateException`, passed in as `ex`. A SOAP “client” fault is created. The `actor` parameter can be used to specify a URI that identifies the application, if it is not the ultimate recipient of the SOAP message.

The `resolvers` module — fetching remote data

The `resolvers` module provides some functions and classes that can be used as the `resolver` attribute for `TC.String` or `TC.XML` typecodes. They process an absolute URL, as described above, and return the content. Because the `resolvers` module can import a number of other large modules, it must be imported directly, as in `'from ZSI import resolvers'`.

These first two functions pass the URI directly to the `urlopen` function in the `urllib` module. Therefore, if used directly as resolvers, a client could direct the SOAP application to fetch any file on the network or local disk. Needless to say, this could pose a security risks.

Opaque (*uri*, *tc*, *ps*[, ***keywords*])

This function returns the data contained at the specified `uri` as a Python string. Base-64 decoding will be done if necessary. The `tc` and `ps` parameters are ignored; the `keywords` are passed to the `urlopen` method.

XML (*uri*, *tc*, *ps*[, ***keywords*])

This function returns a list of the child element nodes of the XML document at the specified `uri`. The `tc` and `ps` parameters are ignored; the `keywords` are passed to the `urlopen` method.

The `NetworkResolver` class provides a simple-minded way to limit the URI's that will be resolved.

class NetworkResolver ([*prefixes=None*])

The `prefixes` parameter is a list of strings defining the allowed prefixes of any URI's. If asked to fetch the content for a URI that does start with one of the prefixes, it will raise an exception.

In addition to `Opaque` and `XML` methods, this class provides a `Resolve` method that examines the typecode to determine what type of data is desired.

If the SOAP application is given a multi-part MIME document, the `MIMEResolver` class can be used to process SOAP with Attachments.

The `MIMEResolver` class will read the entire multipart MIME document, noting any `Content-ID` or `Content-Location` headers that appear on the headers of any of the message parts, and use them to resolve any `href` attributes that appear in the SOAP message.

class MIMEResolver (*ct*, *f*[, ***keywords*])

The `ct` parameter is a string that contains the value of the MIME `Content-Type` header. The `f` parameter is the input stream, which should be positioned just after the message headers.

The following keyword arguments may be used:

| Keyword | Default | Description |
|----------------|----------------|--|
| seekable | 0 | Whether or not the input stream is seekable; passed to the constructor for the internal <code>multifile</code> object. Changed in version 2.0: default had been 1. |
| next | None | A resolver object that will be asked to resolve the URI if it is not found in the MIME document. New in version 1.1. |
| uribase | None | The base URI to be used when resolving relative URI's; this will typically be the value of the <code>Content-Location</code> header, if present. New in version 1.1. |

In addition to the `Opaque`, `Resolve`, and `XML` methods as described above, the following method is available:

GetSOAPPart ()

This method returns a stream containing the SOAP message text.

The following attributes are read-only:

parts

An array of tuples, one for each MIME bodypart found. Each tuple has two elements, a `mimertools.Message` object which contains the headers for the bodypart, and a `StringIO` object containing the data.

id_dict

A dictionary whose keys are the values of any `Content-ID` headers, and whose value is the appropriate `parts` tuple.

loc_dict

A dictionary whose keys are the values of any `Content-Location` headers, and whose value is the appropriate `parts` tuple.

Dispatching and Invoking

New in version 1.1.

ZSI is focused on parsing and generating SOAP messages, and provides limited facilities for dispatching to the appropriate message handler. This is because ZSI works within many client and server environments, and the dispatching styles for these different environments can be very different.

Nevertheless, ZSI includes some dispatch and invocation functions. To use them, they must be explicitly imported, as shown in the example at the start of this document.

The implementation (and names) of these classes reflects the orientation of using SOAP for remote procedure calls (RPC).

Both client and server share a class that defines the mechanism a client uses to authenticate itself.

class AUTH ()

This class defines constants used to identify how the client authenticated: `none` if no authentication was provided; `httpbasic` if HTTP basic authentication was used, or `zbasic` if ZSI basic authentication (see below) was used.

The ZSI schema (see the last chapter of this manual) defines a SOAP header element, `BasicAuth`, that contains a name and password. This is similar to the HTTP basic authentication header, except that it can be used independently from an HTTP transport.

10.1 Dispatching

The `ZSI.dispatch` module allows you to expose Python functions as a web service. The module provides the infrastructure to parse the request, dispatch to the appropriate handler, and then serialize any return value back to the client. The value returned by the function will be serialized back to the client. If an exception occurs, a SOAP fault will be sent back to the client.

10.1.1 Dispatch Behaviors

By default the callback is invoked with the `pyobj` representation of the body root element, and it is expected to return a self-describing request (`w/typecode`). Parsing is done via a `typecode` from `typesmodule`, or `Any`. Other keyword options are available in dispatch mechanisms (see below) that result in different behavior.

`rpc`

An `rpc` service will ignore the body root (RPC Wrapper) of the request, and parse all "parts" of message via individual `typecodes`. The callback function is expected to return the parts of the message in a dict or a list. The dispatch

mechanism will try to serialize it as a Struct but if this is not possible it will be serialized as an Array. Parsing done via a typecode from typesmodule, or Any. Not compatible with *docstyle*.

docstyle

Callback is invoked with a ParsedSoap instance representing the request, and the return value is serialized with an XML typecode (DOM). The result is wrapped as an rpc-style message, with *Response* appended to the request wrapper. Not compatible with *rpc*.

10.1.2 Special Modules

These are keyword options available to all dispatch mechanism (see below).

modules

Dispatch is based solely on the name of the root element in the incoming SOAP request; the request URL is ignored. These modules will be searched for a matching function. If no modules are specified, only the `__main__` module will be searched.

typesmodule

Used for parsing. This module should contain class definitions with the `typecode` attribute set to a `TypeCode` instance. By default, a class definition matching the root element name will be retrieved or the `Any` typecode will be used. If using *rpc*, each child of the root element will be used to retrieve a class definition of the same name.

10.1.3 Dispatch Mechanisms

Three dispatch mechanisms are provided: one supports standard CGI scripts, one runs a dedicated server based on the `BaseHTTPServer` module, and the third uses the `JonPY` package, <http://jonpy.sourceforge.net>, to support FastCGI.

AsServer (`[**keywords]`)

This creates a `HTTPServer` object with a request handler that only supports the “POST” method. Dispatch is based solely on the name of the root element in the incoming SOAP request; the request URL is ignored.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|--------------------------|--|
| <code>port</code> | <code>80</code> | Port to listen on. |
| <code>addr</code> | <code>''</code> | Address to listen on. |
| <code>docstyle</code> | <code>False</code> | Exhibit the <i>docstyle</i> behavior. |
| <code>rpc</code> | <code>False</code> | Exhibit the <i>rpc</i> behavior. |
| <code>modules</code> | <code>(__main__,)</code> | List of modules containing functions that can be invoked. |
| <code>typesmodule</code> | <code>(__main__,)</code> | This module is used for parsing, it contains class definitions that specify the <code>typecode</code> attribute. |
| <code>nsdict</code> | <code>{}</code> | Namespace dictionary to send in the SOAP Envelope |

AsCGI (`[**keywords]`)

This method parses the CGI input and invokes a function that has the same name as the top-level SOAP request element.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|--------------------------|--|
| <code>rpc</code> | <code>False</code> | Exhibit the <i>rpc</i> behavior. |
| <code>modules</code> | <code>(__main__,)</code> | List of modules containing functions that can be invoked. |
| <code>typesmodule</code> | <code>(__main__,)</code> | This module is used for parsing, it contains class definitions that specify the <code>typecode</code> attribute. |
| <code>nsdict</code> | <code>{}</code> | Namespace dictionary to send in the SOAP Envelope |

AsHandler (*request=None*[, ***keywords*])

This method is used within a JonPY handler to do dispatch.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|--------------------------|--|
| <code>request</code> | <code>None</code> | modpython HTTPRequest instance. |
| <code>modules</code> | <code>(__main__,)</code> | List of modules containing functions that can be invoked. |
| <code>docstyle</code> | <code>False</code> | Exhibit the <i>docstyle</i> behavior. |
| <code>rpc</code> | <code>False</code> | Exhibit the <i>rpc</i> behavior. |
| <code>typesmodule</code> | <code>(__main__,)</code> | This module is used for parsing, it contains class definitions that specify the <code>typecode</code> attribute. |
| <code>nsdict</code> | <code>{}</code> | Namespace dictionary to send in the SOAP Envelope |

AsJonPy (*request=None*[, ***keywords*])

This method is used within a JonPY handler to do dispatch.

The following keyword arguments may be used:

| Keyword | Default | Description |
|--------------------------|--------------------------|--|
| <code>request</code> | <code>None</code> | jonpy Request instance. |
| <code>modules</code> | <code>(__main__,)</code> | List of modules containing functions that can be invoked. |
| <code>docstyle</code> | <code>False</code> | Exhibit the <i>docstyle</i> behavior. |
| <code>rpc</code> | <code>False</code> | Exhibit the <i>rpc</i> behavior. |
| <code>typesmodule</code> | <code>(__main__,)</code> | This module is used for parsing, it contains class definitions that specify the <code>typecode</code> attribute. |
| <code>nsdict</code> | <code>{}</code> | Namespace dictionary to send in the SOAP Envelope |

The following code shows a sample use:

```
import jon.fcgi
from ZSI import dispatch
import MyHandler

class Handler(cgi.Handler):
    def process(self, req):
        dispatch.AsJonPy(modules=(MyHandler,), request=req)

jon.fcgi.Server({jon.fcgi.FCGI_RESPONDER: Handler}).run()
```

10.1.4 Other Dispatch Stuff

GetClientBinding()

More sophisticated scripts may want to use access the client binding object, which encapsulates all information about the client invoking the script. This function returns `None` or the binding information, an object of type `ClientBinding`, described below.

class ClientBinding(...)

This object contains information about the client. It is created internally by `ZSI`.

GetAuth ()

This returns a tuple containing information about the client identity. The first element will be one of the constants from the `AUTH` class described above. For HTTP or ZSI basic authentication, the next two elements will be the name and password provided by the client.

GetNS ()

Returns the namespace URI that the client is using, or an empty string. This can be useful for versioning.

GetRequest ()

Returns the `ParsedSoap` object of the incoming request.

The following attribute is read-only:

environ

A dictionary of the environment variables. This is most useful when `AsCGI ()` is used.

10.2 The `client` module — sending SOAP messages

ZSI includes a module to connect to a SOAP server over HTTP, send requests, and parse the response. It is built on the standard Python `httplib` and `Cookie` modules. It must be explicitly imported, as in `'from ZSI.client import AUTH, Binding'`.

10.2.1 `_Binding`

class `_Binding` (*keywords*)**

This class encapsulates a connection to a server, known as a *binding*. A single binding may be used for multiple RPC calls. Between calls, modifiers may be used to change the URL being posted to, etc.

Cookies are also supported; if a response comes back with a `Set-Cookie` header, it will be parsed and used in subsequent interactions.

The following keyword arguments may be used:

| Keyword | Default | Description |
|---------------------------|---|---|
| <code>auth</code> | <code>(AUTH.none,)</code> | A tuple with authentication information; the first value should be one of the constants from the <code>AUTH</code> class. |
| <code>nsdict</code> | <code>{}</code> | Namespace dictionary to send in the SOAP Envelope |
| <code>soapaction</code> | <code>''</code> | Value for the <code>SOAPAction</code> HTTP header. |
| <code>readerclass</code> | <code>None</code> | Class used to create DOM-creating XML readers; see the description in the <code>ParsedSoap</code> class. |
| <code>writerclass</code> | <code>None</code> | <code>ElementProxy</code> Class used to create XML writers; see the description in the <code>SoapWriter</code> class. |
| <code>tracefile</code> | <code>None</code> | An object with a <code>write</code> method, where packet traces will be recorded. |
| <code>transport</code> | <code>HTTPConnection/HTTPSConnection</code> | transport class |
| <code>transdict</code> | <code>{}</code> | keyword arguments for connection initialization |
| <code>url</code> | <code>n/a</code> | URL to post to. |
| <code>wsAddressURI</code> | <code>None</code> | URI, identifies the WS-Address specification to use. By default it's not used. |
| <code>sig_handler</code> | <code>None</code> | XML Signature handler, must sign and verify. |

If using SSL, the `cert_file` and `key_file` keyword parameters may be passed to the transport's constructor via the `transdict`. For details about `HTTPSConnection` see the documentation for the `httplib` module.

Once a `_Binding` object has been created, the following modifiers are available. All of them return the binding object, so that multiple modifiers can be chained together.

AddHeader (*header, value*)

Output the specified `header` and `value` with the HTTP headers.

SetAuth (*style, name, password*)

The `style` should be one of the constants from the `AUTH` class described above. The remaining parameters will vary depending on the `style`. Currently only basic authentication data of name and password are supported.

SetNS (*uri*)

Set the default namespace for the request to the specified `uri`.

SetURL (*url*)

Set the URL where the post is made to `url`.

ResetHeaders ()

Remove any headers that were added by `AddHeader()`.

The following attribute may also be modified:

trace

If this attribute is not `None`, it should be an object with a `write` method, where packet traces will be recorded.

Once the necessary parameters have been specified (at a minimum, the URL must have been given in the constructor or through `SetURL`), invocations can be made.

RPC (*url, opname, pyobj, replytype=None*, ***keywords*)

This is the highest-level invocation method. It calls `Send()` to send `pyobj` to the specified `url` to perform the `opname` operation, and calls `Receive()` expecting to get a reply of the specified `replytype`.

This method will raise a `TypeError` if the response does not appear to be a SOAP message, or if it is valid SOAP but contains a fault.

Send (*url, opname, pyobj*, ***keywords*)

This sends the specified `pyobj` to the specified `url`, invoking the `opname` method. The `url` can be `None` if it was specified in the `Binding` constructor or if `SetURL` has been called. See below for a shortcut version of this method.

The following keyword arguments may be used:

| Keyword | Default | Description |
|-----------------------------------|--|--|
| <code>auth_header</code> | <code>None</code> | String (containing presumably serialized XML) to output as an authentication header. |
| <code>SOAP Envelope nsdict</code> | <code>{}</code> | Namespace dictionary to send in the SOAP Envelope |
| <code>requesttypecode</code> | <code>n/a</code> | Typecode specifying how to serialize the data. |
| <code>soapaction</code> | Obtained from the <code>Binding</code> | Value for the <code>SOAPAction</code> HTTP header. |

Methods are available to determine the type of response that came back:

IsSOAP ()

Returns true if the message appears to be a SOAP message. (Some servers return an HTML page under certain error conditions.)

IsAFault ()

Returns true if the message is a SOAP fault.

Having determined the type of the message (or, more likely, assuming it was good and catching an exception if not), the following methods are available to actually parse the data. They will continue to return the same value until another message is sent.

ReceiveRaw ()

Returns the unparsed message body.

ReceiveSoap()

Returns a `ParsedSOAP` object containing the parsed message. Raises a `TypeError` if the message wasn't SOAP.

ReceiveFault()

Returns a `Fault` object containing the SOAP fault message. Raises a `TypeError` if the message did not contain a fault.

Receive(*replytype=None*)

Parses a SOAP message. The `replytype` specifies how to parse the data. If it's `None`, dynamic parsing will be used, usually resulting in a Python list. If `replytype` is a Python class, then the class's `typecode` attribute will be used, otherwise `replytype` is taken to be the `typecode` to use for parsing the data.

Once a reply has been parsed (or its type examined), the following read-only attributes are available. Their values will remain unchanged until another reply is parsed.

reply_code

The HTTP reply code, a number.

reply_headers

The HTTP headers, as a `mimertools` object.

reply_msg

A text string containing the HTTP reply text.

10.2.2 Binding

If an attribute is fetched other than one of those described in `_Binding`, it is taken to be the `opname` of a remote procedure, and a callable object is returned. This object dynamically parses its arguments, receives the reply, and parses that.

class Binding([keywords])**

For other keyword arguments see `_Binding`.

| Keyword | Default | Description |
|--------------------------|-------------------|---|
| <code>typesmodule</code> | <code>None</code> | See explanation in <code>Dispatching</code> |

opname(*args)

Using this shortcut requires that the `url` attribute is set, either through the constructor or `SetURL()`.

10.2.3 NamedParamBinding

If an attribute is fetched other than one of those described in `_Binding`, it is taken to be the `opname` of a remote procedure, and a callable object is returned. This object dynamically parses its arguments, receives the reply, and parses that.

class NamedParamBinding([keywords])**

For other keyword arguments see `_Binding`.

| Keyword | Default | Description |
|--------------------------|-------------------|---|
| <code>typesmodule</code> | <code>None</code> | See explanation in <code>Dispatching</code> |

opname(kwargs)**

Using this shortcut requires that the `url` attribute is set, either through the constructor or `SetURL()`.

Bibliography

BIBLIOGRAPHY

- [1] This is the first item in the Bibliography.
- [2] This is the second item in the Bibliography.

CGI Script Array

A.1 Intro

This is an example of a simple web service CGI Script. The service returns and expects SOAP Arrays (python `list`). A sample soap trace is provided below. In this example the CGI script is dispatched as a *rpc* service.

A.1.1 rpc wrapper

The wrapper element of the request is the `dispatch` key to the callback function, the child elements are passes as a `list` or `dict` of values to the callback function. The callback function is expected to return a `list` or `dict` of values, the response wrapper is by default set to the request wrapper name appended *Response*.

A.2 CGI Script

```
#!/usr/local/bin/python2.4
# SOAP Array

def hello():
    return ["Hello, world"]

def echo(*args):
    return args

def sum(*args):
    sum = 0
    for i in args: sum += i
    return [sum]

def average(*args):
    return [sum(*args) / len(args)]

from ZSI import dispatch
dispatch.AsCGI(rpc=True)
```

A.3 client test script

```
#!/usr/bin/env python
# client.py
import sys
from ZSI.client import Binding
b = Binding(url='http://127.0.0.1/cgi-bin/simple', tracefile=sys.stdout)
print b.hello()
try:
    print b.hello(1)
except Exception, ex:
    print "Fault: ", ex

print b.echo("whatever", "hi", 1, 2)
print b.sum(*[2*i for i in range(5)])
print b.average(*[2*i for i in range(5)])
```

A.4 SOAP Trace

A.4.1 hello

```
$ ./client.py
Hello: _____ Wed Oct 4 17:36:33 2006 REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<hello SOAP-ENC:arrayType="xsd:anyType[0]"></hello>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

_____ Wed Oct 4 17:36:34 2006 RESPONSE:
200

<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<helloResponse SOAP-ENC:arrayType="xsd:anyType[1]">
<element id="o671b0" xsi:type="xsd:string">Hello, world</element>
</helloResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
[u'Hello, world']
```

A.4.2 hello fault

```
Wed Oct 4 17:36:34 2006 REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<hello SOAP-ENC:arrayType="xsd:anyType[1]">
<element id="o1803988" xsi:type="xsd:int">1</element>
</hello>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
Wed Oct 4 17:36:35 2006 RESPONSE:
500
```

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Processing Failure</faultstring>
<detail>
<ZSI:FaultDetail>
<ZSI:string>exceptions:TypeError hello() takes no arguments (1 given)</ZSI:string>
<ZSI:trace>build/bdist.darwin-8.8.0-Power_Macintosh/egg/ZSI/dispatch.py:86:_Dispatch</ZSI:trace>
</ZSI:FaultDetail>
</detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
Fault: Processing Failure
exceptions:TypeError
hello() takes no arguments (1 given)
[trace: build/bdist.darwin-8.8.0-Power_Macintosh/egg/ZSI/dispatch.py:86:_Dispatch]
```

A.4.3 echo

```
Wed Oct 4 17:36:35 2006 REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<echo SOAP-ENC:arrayType="xsd:anyType[4]">
<element id="o644c0" xsi:type="xsd:string">whatever</element>
<element id="o644e0" xsi:type="xsd:string">hi</element>
<element id="o1803988" xsi:type="xsd:int">1</element>
<element id="o180397c" xsi:type="xsd:int">2</element>
</echo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
Wed Oct 4 17:36:36 2006 RESPONSE:
200
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<echoResponse SOAP-ENC:arrayType="xsd:anyType[4]">
<element id="o4f4290" xsi:type="xsd:string">whatever</element>
<element id="o4f4338" xsi:type="xsd:string">hi</element>
<element id="o1803988" xsi:type="xsd:int">1</element>
<element id="o180397c" xsi:type="xsd:int">2</element>
</echoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
[u'whatever', u'hi', 1, 2]
```

A.4.4 sum

Wed Oct 4 17:36:36 2006 REQUEST:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<sum SOAP-ENC:arrayType="xsd:anyType[5]">
<element id="o1803994" xsi:type="xsd:int">0</element>
<element id="o180397c" xsi:type="xsd:int">2</element>
<element id="o1803964" xsi:type="xsd:int">4</element>
<element id="o180394c" xsi:type="xsd:int">6</element>
<element id="o1803934" xsi:type="xsd:int">8</element>
</sum>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Wed Oct 4 17:36:37 2006 RESPONSE:

200

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<sumResponse SOAP-ENC:arrayType="xsd:anyType[1]">
<element id="o18038a4" xsi:type="xsd:int">20</element>
</sumResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
[20]
```

A.4.5 average

```
Wed Oct 4 17:36:37 2006 REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<average SOAP-ENC:arrayType="xsd:anyType[5]">
<element id="o1803994" xsi:type="xsd:int">0</element>
<element id="o180397c" xsi:type="xsd:int">2</element>
<element id="o1803964" xsi:type="xsd:int">4</element>
<element id="o180394c" xsi:type="xsd:int">6</element>
`<element id="o1803934" xsi:type="xsd:int">8</element>
</average>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
Wed Oct 4 17:36:38 2006 RESPONSE:
200
```

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<averageResponse SOAP-ENC:arrayType="xsd:anyType[1]">
<element id="o1803964" xsi:type="xsd:int">4</element>
</averageResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
[4]
```

CGI Script Struct

B.1 Intro

This is an example of a simple web service CGI Script. The service returns and expects SOAP Structs (python dict). A sample soap trace is provided below. In this example the CGI script is dispatched as a *rpc* service.

B.1.1 rpc wrapper

The wrapper element of the request is the dispatch key to the callback function, the child elements are passes as a list or dict of values to the callback function. The callback function is expected to return a list or dict of values, the response wrapper is by default set to the request wrapper name appended *Response*.

B.2 CGI Script

```
#!/usr/local/bin/python2.4
# SOAP Struct

def hello():
    return {"value":"Hello, world"}

def echo(**kw):
    return kw

def sum(**kw):
    sum = 0
    for i in kw.values(): sum += i
    return {"value":sum}

def average(**kw):
    d = sum(**kw)
    return d["value"] = d["value"]/len(kw)

from ZSI import dispatch
dispatch.AsCGI(rpc=True)
```

B.3 client test script

```
#!/usr/bin/env python
import sys,time
from ZSI.client import NamedParamBinding as NPBinding

b = NPBinding(url='http://127.0.0.1/cgi-bin/soapstruct', tracefile=sys.stdout)
print "Hello: ", b.hello()
print "Echo: ", b.echo(name="josh", year=2006, pi=3.14, time=time.gmtime())
print "Sum: ", b.sum(one=1, two=2, three=3)
print "Average: ", b.average(one=100, two=200, three=300, four=400)
```

B.4 SOAP Trace

B.4.1 hello

Complete Low Level Example

C.1 Intro

This is a complete example of using the low level soap utilities in ZSI to implement a web service.

C.2 code

C.2.1 httpserver script

Minimal http server example, opens up a new process to do the SOAP processing.

```
#!/usr/bin/env python
# file: httpserver.py
import os
from subprocess import Popen, PIPE
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer

class RequestHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        length = int(self.headers['content-length'])
        xml_in = self.rfile.read(length)
        p = Popen(os.path.join(os.path.curdir, 'player.py'),
                 shell=True, stdin=PIPE, stdout=PIPE)

        (stdout, stderr) = p.communicate(xml_in)
        code = 200
        if stdout.find('Fault') >= 0: code = 500
        self.send_response(code)
        self.send_header('Content-type', 'text/xml; charset="utf-8"')
        self.send_header('Content-Length', str(len(stdout)))
        self.end_headers()
        self.wfile.write(stdout)
        self.wfile.flush()

if __name__ == '__main__':
    server = HTTPServer(('localhost', 8080), RequestHandler)
    server.serve_forever()
```

C.2.2 typecode module

```
# file: typecode.py
# CHECK PYTHONPATH: Must be able to import
class Player:
    def __init__(self, *args):
        if not len(args): return
        self.Name = args[0]
        self.Scores = args[1:]
Player.typecode = TC.Struct(Player, [
    TC.String('Name'),
    TC.Array('Integer', TC.Integer(), 'Scores', undeclared=True),
    ], 'GetAverage')

class Average:
    def __init__(self, average=None):
        self.average = average
Average.typecode = TC.Struct(Average, [
    TC.Integer('average'),
    ], 'GetAverageResponse')
```

C.2.3 player script

```
#!/usr/bin/env python
# file: player.py
from ZSI import *
import sys
IN, OUT = sys.stdin, sys.stdout
try:
    ps = ParsedSoap(IN)
except ParseException, e:
    OUT.write(FaultFromZSIException(e).AsSOAP())
    sys.exit(1)
except Exception, e:
    # Faulted while processing; we assume it's in the header.
    OUT.write(FaultFromException(e, 1).AsSOAP())
    sys.exit(1)

# We are not prepared to handle any actors or mustUnderstand elements,
# so we'll arbitrarily fault back with the first one we found.
a = ps.WhatActorsArePresent()
if len(a):
    OUT.write(FaultFromActor(a[0]).AsSOAP())
    sys.exit(1)
mu = ps.WhatMustIUnderstand()
if len(mu):
    uri, localname = mu[0]
    OUT.write(FaultFromNotUnderstood(uri, localname).AsSOAP())
    sys.exit(1)

from typecode import Player, Average
try:
    player = ps.Parse(Player.typecode)
except EvaluateException, e:
    OUT.write(FaultFromZSIException(e).AsSOAP())
    sys.exit(1)

try:
    total = 0
    for value in player.Scores: total = total + value
    result = Average(total / len(player.Scores))
    sw = SoapWriter()
    sw.serialize(result, Average.typecode)
    sw.close()
    OUT.write(str(sw))
except Exception, e:
    OUT.write(FaultFromException(e, 0, sys.exc_info()[2]).AsSOAP())
    sys.exit(1)
```

C.2.4 client test script

```
#!/usr/bin/env python2.4
#file: client.py
from ZSI import *
from ZSI.wstools.Namespaces import SCHEMA
from typecode import Player, Average

if __name__ == '__main__':
    import sys
    from ZSI.client import Binding
    b = Binding(url='http://localhost:8080', tracefile=sys.stdout)
    pyobj = b.RPC(None, None, Player("Josh",10,20,30), replytype=Average)
    print pyobj
    print pyobj.__dict__
```

C.3 SOAP Trace

C.3.1 GetAverage

```
./client.py
----- Thu Oct 5 14:57:39 2006 REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<GetAverage>
<Name xsi:type="xsd:string">Josh</Name>
<Scores>
<element>10</element>
<element>20</element>
<element>30</element>
</Scores>
</GetAverage>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
----- Thu Oct 5 14:57:39 2006 RESPONSE:
200
OK
-----
Server: BaseHTTP/0.3 Python/2.5
Date: Thu, 05 Oct 2006 21:57:39 GMT
Content-type: text/xml; charset="utf-8"
Content-Length: 431

<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<GetAverageResponse>
<average>20</average>
</GetAverageResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

<__main__.Average instance at 0x5f9760>
{'average': 20}
```

C.3.2 fault

Purposely send a incorrect *Nae* element for the *Name*.

```
./client.py
```

```
_____ Thu Oct 5 14:33:25 2006 REQUEST:  
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"  
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
<SOAP-ENV:Header></SOAP-ENV:Header>  
<SOAP-ENV:Body>  
<GetAverage>  
<Nae xsi:type="xsd:string">Josh</Nae>  
<Scores>  
<element>10</element>  
<element>20</element>  
<element>30</element>  
</Scores>  
</GetAverage>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

```
_____ Thu Oct 5 14:33:26 2006 RESPONSE:  
500  
Internal Server Error
```

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"  
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
<SOAP-ENV:Header></SOAP-ENV:Header>  
<SOAP-ENV:Body>  
<SOAP-ENV:Fault>  
<faultcode>SOAP-ENV:Client</faultcode>  
<faultstring>Unparseable message</faultstring>  
<detail><Eoe440>&lt;ZSI:ParseFaultDetail&gt;  
&lt;ZSI:string&gt;Element "Name" missing from complexType&lt;/ZSI:string&gt;  
&lt;ZSI:trace&gt;/SOAP-ENV:Envelope/SOAP-ENV:Body/GetAverage&lt;/ZSI:trace&gt;  
&lt;/ZSI:ParseFaultDetail&gt;</Eoe440></detail>  
</SOAP-ENV:Fault>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

```
Traceback (most recent call last):
```

```
File "./player_client.py", line 25, in ?  
  pyobj = b.RPC(None, None, Player("Josh",10,20,30), replytype=Average)  
File "/private/var/www/htdocs/guide/client.py", line 176, in RPC  
  
File "/private/var/www/htdocs/guide/client.py", line 420, in Receive
```

```
ZSI.FaultException: Unparseable message  
<Element Node at 5f9f58: Name='Eoe440' with 0 attributes and 1 children>
```

Pickler example

D.1 Intro

This is an example of a stateful mod_python web service.

D.2 code

D.2.1 typecode module

Module containing complex type typecode.

```
# Complex type definition
from ZSI import *
class Person:
    def __init__(self, name=None, age=0):
        self.name = name
        self.age = age

Person.typecode = TC.Struct(Person,
                             [TC.String('name'),
                              TC.InonNegativeInteger('age')],
                             pname=('urn:MyApp', 'Person'))
```

D.2.2 pickler script

Configure appache to use this script with mod_python PythonHandler.

```

# pickler.py
import pickle, new
from mod_python import apache
from ZSI import dispatch
import MyComplexTypes

# my web service that returns a complex structure
def getPerson(name=None):
    #fp = open('/tmp/%s.person.pickle'%Person.name, 'r')
    fp = open('/tmp/%s.person.pickle'%name, 'r')
    #return pickle.load(fp)
    p = pickle.load(fp)
    print "PERSON: ", p
    print "typecode: ", p.typecode
    return p

# my web service that accepts a complex structure
def savePerson(Person):
    print "PERSON: ", Person
    fp = open('/tmp/%s.person.pickle'%Person.name, 'w')
    pickle.dump(Person, fp)
    fp.close()
    return {}

mod = __import__('encodings.utf_8', globals(), locals(), '*')
mod = __import__('encodings.utf_16_be', globals(), locals(), '*')

handles = new.module('handles')
handles.getPerson = getPerson
handles.savePerson = savePerson
def handler(req):
    dispatch.AsHandler(modules=(handles,), request=req, typesmodule=MyComplexTypes, rpc=True)
    return apache.OK

```

D.2.3 client: invoke savePerson

script

```

import sys
from ZSI.client import Binding
from MyComplexTypes import Person

b = Binding(url='http://localhost/test3/pickler.py', tracefile=sys.stdout)
person = Person('christopher', 26)
b.savePerson(person)

```

SOAP Trace

Wed Oct 11 13:10:05 2006 REQUEST:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<savePerson xmlns:ns1="urn:MyApp">
<ns1:Person><name xsi:type="xsd:string">christopher</name>
<age xsi:type="xsd:nonNegativeInteger">26</age>
</ns1:Person>
</savePerson>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Wed Oct 11 13:10:05 2006 RESPONSE:

```
Server: Apache/2.0.53-dev (Unix) mod_ruby/1.2.4 Ruby/1.8.2 (2004-12-25)
mod_python/3.1.4 Python/2.4.1
Transfer-Encoding: chunked
Content-Type: text/xml
```

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<savePersonResponse></savePersonResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

D.2.4 client: invoke getPerson 3 different ways

script

```
import sys
import MyComplexTypes
from ZSI.client import NamedParamBinding as NPBinding, Binding
from ZSI import TC

kw = {'url':'http://localhost/test3/pickler.py', 'tracefile':sys.stdout}
b = NPBinding(**kw)
rsp = b.getPerson(name='christopher')
assert type(rsp) is dict, 'expecting a dict'
assert rsp['Person']['name'] == 'christopher', 'wrong person'

b = NPBinding(typesmodule=MyComplexTypes, **kw)
rsp = b.getPerson(name='christopher')
assert isinstance(rsp['Person'], MyComplexTypes.Person), (
    'expecting instance of %s' %MyComplexTypes.Person)

b = Binding(typesmodule=MyComplexTypes, **kw)
class Name(str):
    typecode = TC.String("name")

rsp = b.getPerson(Name('christopher'))
assert isinstance(rsp['Person'], MyComplexTypes.Person), (
    'expecting instance of %s' %MyComplexTypes.Person)
```

SOAP Trace

All responses are exactly the same, for comparison the three requests are presented first and only the last response is included.

Wed Oct 11 13:19:00 2006 REQUEST:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<getPerson>
<name id="o6c2a0" xsi:type="xsd:string">christopher</name>
</getPerson>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

** OMIT RESPONSE **

Wed Oct 11 13:19:00 2006 REQUEST:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<getPerson>
<name id="o6c2a0" xsi:type="xsd:string">christopher</name>
</getPerson>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

** OMIT RESPONSE **

Wed Oct 11 13:19:00 2006 REQUEST:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body>
<getPerson>
<name xsi:type="xsd:string">christopher</name>
</getPerson>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Wed Oct 11 13:19:00 2006 RESPONSE:
Server: Apache/2.0.53-dev (Unix) mod_ruby/1.2.4 Ruby/1.8.2 (2004-12-25)
mod_python/3.1.4 Python/2.4.1
Transfer-Encoding: chunked
Content-Type: text/xml

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header></SOAP-ENV:Header>
```