ZSI: The Zolera Soap Infrastructure

Release 2.0.0

Rich Salz,
Christopher Blunck

January 04, 2006

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 2.0 or later and PyXML version 0.6.6 or later.

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

CONTENTS

CHAPTER

ONE

Introduction

ZSI, the Zolera SOAP Infrastructure, is a Python package that provides an implementation of the SOAP specification, as described in *The SOAP 1.1 Specification*. In particular, ZSI parses and generates SOAP messages, and converts between native Python datatypes and SOAP syntax.

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

This release of `ZSI` adds the capability to process WSDL definitions (described in *The Web Services Description Language*) and generate typecodes automatically. See chapter 11 for details.

CHAPTER

TWO

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.cgi` 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:

```
def hello():
    return "Hello, world"

def echo(*args):
    return args

def average(*args):
    sum = 0
    for i in args: sum += i
    return sum / len(args)

from ZSI import dispatch
dispatch.AsCGI()
```

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.

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.

2.1.2 More complex example

We will now show a more complete example of a robust web service. 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.

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:
    FaultFromZSIException(e).AsSOAP(OUT)
    sys.exit(1)
except Exception, e:
    # Faulted while processing; we assume it's in the header.
    FaultFromException(e, 1).AsSOAP(OUT)
    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):
    FaultFromActor(a[0]).AsSOAP(OUT)
    sys.exit(1)
mu = ps.WhatMustIUnderstand()
if len(mu):
    uri, localname = mu[0]
    FaultFromNotUnderstood(uri, localname).AsSOAP(OUT)
    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, name):
        pass
Player.typecode = TC.Struct(Player, [
                                TC.String('Name'),
                                TC.Array('Integer', TC.Integer(), 'Scores'),
                                ], 'GetAverage')
class Average:
    def __init__(self, average):
        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:
    FaultFromZSIException(e).AsSOAP(OUT)
    sys.exit(1)

try:
    total = 0
    for value in player.Scores: total = total + value
    result = Average(total / len(player.Scores))
    sw = SoapWriter(OUT)
    sw.serialize(result, Average.typecode)
    sw.close()
except Exception, e:
    FaultFromException(e, 0, sys.exc_info()[2]).AsSOAP(OUT)
    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. In addition, since the `SoapWriter` destructor will call `close()` if necessary, sending a SOAP response can often be written like this one-liner:

```
SoapWriter(OUT).serialize(result)
```

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 "Hello, world"

def echo(*args):
    return args

def average(*args):
    sum = 0
    for i in args: sum += i
    return sum / len(args)
```

Dispatching from within mod_python is achieved by passing the aforementined 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)
    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` 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 `Binding` class to connect to a remote service and perform an operation. It assumes that the simple server-side example shown above is installed on the webserver running on the local host, and if the URL is `/cgi-bin/simple-test`:

```
from ZSI.client import Binding
fp = open('debug.out', 'a')
b = Binding(url='/cgi-bin/simple-test', tracefile=fp)
fp.close()
a = b.average(range(1,11))
assert a == 5
print b.hello()
```

2.2.2 Complex Example

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:

```
# Complex type definition
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 must be appended to the *ZSI.Path* list. It is also possible to explicitly tell `ZSI` what class and typecode to use by passing the class as a parameter to the `Binding.Receive()` method. The first method is often preferred, particularly for publically-distributed libraries.

The following fragment shows both styles:

```
from ZSI.client import Binding
from ZSI import Path

# Explicitly stating what to get back.
from MyComplexTypes import Person
a = apply(b.getPerson, 'christopher')
person = b.Receive(Person)

# Transparent deserialization
import MyComplexTypes
b = Binding(url='/cgi-bin/complex-test', typesmodule=MyComplexTypes)
person = b.getPerson('christopher')
```

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

```
from ZSI.client import Binding
b = Binding(url='/cgi-bin/complex-test')
person = Person('christopher', 26)
b.savePerson(person)
```

CHAPTER

THREE

Exceptions

ZSI defines two exception classes.

exception ParseException
: ZSI can raise this exception while creating a ParsedSoap object. It is a subtype of Python's Exception 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.

CHAPTER

FOUR

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.

_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).

_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.

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):
   ...
```

_children(*element*)
Returns a list of all children of the specified element.

_attrs(*element*)
Returns a list of all attributes of the specified element.

_child_elements(*element*)
Returns a list of all children elements of the specified element.

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

_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_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.

CHAPTER

FIVE

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
keepdom	0	Do not release the DOM when this object is destroyed. To access the DOM object, use the GetDomAndReader() method. The reader object is necessary to properly free the DOM structure using reader.releaseNode(dom). New in version 1.2.
readerclass	None	Class used to create DOM-creating XML readers; described below. New in version 1.2.
resolver	None	Value for the resolver attribute; see below.
trailers	0	Allow trailing data elements to appear after the Body.

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 the DOM provided with the PyXML package developed by the Python XML SIG, provided through the PyExpat.Reader class in the xml.dom.ext.reader 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
```

CHAPTER

SIX

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 three attributes: the pname, the aname, and the oname (in approximate order of importance). They specify the name expected on the XML element being parsed, the name to use for the analogous attribute in the local Python object, and the name to use for the output element when serializing.

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 name argument, known as name, for the pname. 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. (Because the output name is not namespace-qualified, it may be necessary to set the default namespace, such as through the nsdict parameter of the SoapWriter class. When the name is specified as a string, it can be either a simple XML name (such as "foo"), or a colon-separated XML qualified name (such as "tns:foo"). If a qualified name is used, the namespace prefix is ignore on input and for the Python attribute, but the full qualified name is used for output; 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 the TC.Struct or TC.Choice can use this parameter to specify the tag, dictionary key, or instance attribute to set.

The final name, oname, specifies the name to use for the XML element when serializing. This is most useful when using the same typecode for both parsing and serializing operations. It can be any string, and is output directly; a name like "tns:foo" implies that the nsdict parameter to the SoapWriter construct should have an entry for "tns," otherwise the resulting output will not be well-formed XML.

class TypeCode(*name, **keywords*)

The name parameter is the name of the object; this is only required when a typecode appears within a TC.Struct as it defines the attribute name used to hold the data, or within a TC.Choice as it determines the data type. (Since SOAP RPC models transfer as structures, this essentially means that a the name parameter can never be None.)

The following keyword arguments may be used:

Keyword	Default	Description
aname		See name discussion above.
default	n/a	Value if the element is not specified.
optional	0	The element is optional; see below.
oname		See name discussion above.
repeatable	0	If multiple instances of this occur in a TC.Struct, collect the values into a list. New in version 1.2.
typed	1	Output type information (in the xsi:type attribute) when serializing. By special dispensation, typecodes within a TC.Struct object inherit this from the container.
unique	0	If true, the object is unique and will never be "aliased" with another object, so the id attribute need not be output.

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.

typed
This is a class attribute. This sets the default value for whether or not typecodes output typing (xsi:type attribute) information. The default is none-zero.

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 0. If it has no data, and the typecode is not optional, an EvaluateException is raised; if it is optional, a 1 is returned.

simple_value(*elt, ps*)
Returns the text content of the element elt. If no value is present, or the element has non-text children, an EvaluateException is raised.

6.1 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:

```
<tns:i xsi:type="SOAP-ENC:integer">12</tns:i>
<tns:i xsi:type="xsi: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 `ParseException` for the fourth.

Compound data, such as a `struct`, may also be self-describing:

```
<tns:foo xsi:type="tns:mytype">
    <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 dictionary or a sequence will be created:

```
{   'name': u'Hello world',    'i': 12   }

[ 12, u'Hello world' ]
```

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

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
`aslist`	`0`	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.

The rest of this section describes how to add new types to the `ZSI` typecode engine.

class *NEWTYPECODE*(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 by 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
`attrtext`	`None`	Text (with leading space) to output as an attribute; this is normally used by the `TC.Array` class to pass down indexing information.
`name`	`None`	Name to use for serialization; defaults to the name specified in the typecode, or a generated name.
`typed`	*per-typecode*	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.

6.2 Void

A SOAP void is a Python `None`.

class Void(*name*[, ***keywords*])
A `Void` is an item without a value. It is of marginal utility, mainly useful for interoperability tests, and as an optional item within a `Struct`.

6.3 Strings

SOAP Strings are Python strings. If the value to be serialized is a Python sequence, then an `href` is generated, with the first element of the sequence used as the URI. This can be used, for example, when generating SOAP with attachments.

class String(*name*[, ***keywords*])
The parent type of all SOAP strings.

The following keyword arguments may be used:

Keyword	Default	Description
resolver	None	A function that can resolve an absolute URI and return its content as a string, as described in the ParsedSoap description.
strip	1	If true, leading and trailing whitespace are stripped from the content.
textprotect	1	If true, less-than and ampersand characters are replaced with < and &, respectively. New in version 1.1.

class Enumeration(*value_list, name*[, ***keywords*])
Like TC.String, but the value must be a member of the value_list 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
orig	None	If deriving a new typecode from the string class, and the derivation creates a temporary Python string (such as by Base64String), than 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
readerclass	None	Class used to create DOM-creating XML readers; described in the ParsedSoap chapter.

6.4 Integers

SOAP integers are Python integers.

class Integer(*name*[, ***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
format	%d	Format string for serializing. New in version 1.2.

class IEnumeration(*value_list, name*[, ***keywords*])
Like TC.Integer, but the value must be a member of the value_list sequence.

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

class Ibyte(*name*[, ***keywords*])
A signed eight-bit value.

class IunsignedByte(*name*[, ***keywords*])
An unsigned eight-bit value.

class Ishort(*name*[, ***keywords*])
A signed 16-bit value.

class IunsignedShort(*name*[, ***keywords*])
An unsigned 16-bit value.

class Iint(*name*[, ***keywords*])
A signed 32-bit value.

class IunsignedInt(*name*[, ***keywords*])
An unsigned 32-bit value.

class Ilong(*name*[, ***keywords*])
An signed 64-bit value.

class IunsignedLong(*name*[, ***keywords*])
An unsigned 64-bit value.

class IpositiveInteger(*name*[, ***keywords*])
A value greater than zero.

class InegativeInteger(*name*[, ***keywords*])
A value less than zero.

class InonPositiveInteger(*name*[, ***keywords*])
A value less than or equal to zero.

class InonNegativeInteger(*name*[, ***keywords*])
A value greater than or equal to zero.

6.5 Floating-point Numbers

SOAP floating point numbers are Python floats.

class `Decimal`(*name*[, ***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
`format`	`%f`	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.

6.6 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`(*name*[, ***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`(*name*[, ***keywords*])
An absolute time period. This class should not be instantiated directly; use one of the `gXXX` classes instead.

class `gDateTime`(*name*[, ***keywords*])
A date and time.

class `gDate`(*name*[, ***keywords*])
A date.

class `gYearMonth`(*name*[, ***keywords*])
A year and month.

class `gYear`(*name*[, ***keywords*])
A year.

class gMonthDay(*name*[, ***keywords*])
A month and day.

class gDay(*name*[, ***keywords*])
A day.

class gTime(*name*[, ***keywords*])
A time.

6.7 Boolean

SOAP Booleans are Python integers.

class Boolean(*name*[, ***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.

6.8 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 returend 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 longerneeded.

class XML(*name*[, ***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
`copyit`	TC.XML.copyit	Return a copy of the parsed data.
`comments`	`0`	Preserve comments in output.
`inline`	`0`	The XML sub-tree is single-reference, so can be output in-place.
`resolver`	`None`	A function that can resolve an absolute URI and return its content as an element node, as described in the `ParsedSoap` description.
`wrapped`	`1`	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
`unsuppressedPrefixes`	[]	An array of strings identifying the namespace prefixes that should be output.

6.9 Struct

SOAP structs are either Python dictionaries or instances of application-specified classes.

class `Struct`(*pyclass, typecode_seq, name*[, ***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 `typecode_seq` sequence specifies a (possible) dictionary entry. Otherwise, `pyclass` must be a Python class object whose constructor takes a single parameter, which will be the value of the `name` parameter given in the `TC.Struct` constructor. (This allows a single `pyclass` to be used for different typecodes.) The data is then marshaled into the object, and each item in the `typecode_seq` sequence specifies an attribute of the instance to set.

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

The following keyword arguments may be used:

Keyword	Default	Description
`hasextras`	`0`	Ignore any extra elements that appear in the in the structure. If `inorder` is true, extras can only appear at the end.
`inorder`	`0`	Items within the structure must appear in the order specified in the `TCseq` sequence.
`inline`	`0`	The structure is single-reference, so ZSI does not have to use `href/id` encodings.
`mutable`	`0`	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 `inline` argument. New in version 1.2.
`type`	`None`	A '`(uri, localname)`' tuple that defines the type of the structure. If present, and if the input data has a `xsi:type` 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.

If the `typed` keyword is used, then its value is assigned to all typecodes in the `typecode_seq` parameter. If any of the typecodes in `typecode_seq` 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')
```

6.10 Choice

A choice is a Python two-element '`(name, value)`' tuple, representing a union of different types. The first item is a string identifying the type, and the second is the actual data.

class `Choice`(*typecode_seq, name*[, ***keywords*])

When parsing, ZSI will look at the element name in the SOAP message, and determine which of the choices to create.

When serializing Python objects to SOAP messages, `ZSI` must be explicitly told which of the choices define the data. This is done by passing a two-element tuple. The first item is a string identifying the name of a typecode from the `typecode_seq` list of typecodes. The second is the object to be serialized.

6.11 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-dimensioned arrays are supported.

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

The `atype` parameter is a text string 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
`childnames`	`None`	Default name to use for the child elements.
`dimensions`	`1`	The number of dimensions in the array.
`fill`	`None`	The value to use when an array element is omitted.
`mutable`	`0`	Same as `TC.Struct` New in version 1.2.
`nooffset`	`0`	Do not use the SOAP `offset` attribute so skip leading elements with the same value as `fill`.
`sparse`	`0`	The array is sparse.
`size`	`None`	An integer or list of integers that specifies the maximum array dimensions.
`undeclared`	`0`	The SOAP '`arrayType`' attribute need not appear.

6.12 Apache Datatype

The Apache SOAP project, urlhttp://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
`aslist`	`0`	Use a list of tuples rather than a dictionary.

CHAPTER

SEVEN

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`(*out*[, ***keywords*])

The `out` parameter is an object that has a `write()` method for generating the output.

The following keyword arguments may be used:

Keyword	Default	Description
`encoding`	`SOAP-ENC` value	If not `None`, then use the specified value as the value for the SOAP `encodingStyle` attribute. New in version 1.2.
`envelope`	`1`	Write the SOAP envelope elements. New in version 1.2.
`nsdict`	`{}`	Dictionary of namespaces to declare in the SOAP `Body`. Note that earlier versions of ZSI put the declarations on the SOAP `Envelope`; they have been moved to the `Body` for greater interoperability.
`header`	`None`	A sequence of elements to output in the SOAP `Header`. It may also be a text string, in which case it is output as-is, and should therefore be XML text.

Creating a `SoapWriter` object with a `StringIO` object for the `out` parameter and `envelope` set to false results in an object that can be used for serializing objects into a string.

`serialize`(*pyobj*[, *typecode*[, *root=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`([*trailer=None*[, *nsdict=None*]])

Close off the SOAP message, finishing all the pending serializations. If `trailer` is a string or list of elements,

it is output after the close-tag for the `Body`. The `close()` method of the originally provided out object is NOT called. (If it were, and the original `out` object were a `StringIO` object, there would be no way to collect the data.) This method will be invoked automatically if the object is deleted.

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.

`write`(*arg*)
This is a convenience method that calls `self.out.write()` on `arg`, with the addition that if `arg` is a sequence, it iterates over the sequence, writing each item (that isn't `None`) in turn.

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

CHAPTER

EIGHT

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
actor	`None`	A string identifying the `actor` attribute that caused the problem (usually because it is unknown).
detail	`None`	A sequence of elements to output in the `detail` element; it may also be a text string, in which case it is output as-is, and should therefore be XML text.
headerdetail	`None`	Data, treated the same as the `detail` 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`([*output*[, ***kw*]])

This method serializes the `Fault` object into a SOAP message. If the *output* parameter is not specified or `None`,

the message is returned as a string. Any other keyword arguments are passed to the `SoapWriter` constructor. Otherwise `AsSOAP()` will call *output*`.write()` as needed to output the message. 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 must support a `write()` method.

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.

CHAPTER

NINE

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 multifile 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 Content-Location header, if present. New in version 1.1.

In addition to 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 mimetools.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.

CHAPTER

TEN

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 the 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 zsibasic 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. To return multiple values, return a list.

If an exception occurs, a SOAP fault will be sent back to the client.

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.

AsCGI([*module_list*])

This method parses the CGI input and invokes a function that has the same name as the top-level SOAP request element. The optional module_list parameter can specify a list of modules (already imported) to search for functions. If no modules are specified, only the __main__ module will be searched.

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
docstyle	0	If true, then all methods are invoked with a single argument, the unparsed body of the SOAP message.
modules	(__main__,)	List of modules containing functions that can be invoked.
nsdict	{}	Namespace dictionary to send in the SOAP Envelope
port	80	Port to listen on.

AsJonPy(*request=req*[, ***keywords*])

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

The following keyword arguments may be used:

Keyword	Default	Description
request	(__main__,)	List of modules containing functions that can be invoked.

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()
```

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`'.

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
`auth`	`(AUTH.none,)`	A tuple with authentication information; the first value should be one of the constants from the `AUTH` class.
`host`	`'localhost'`	Host to connect to.
`ns`	n/a	Default namespace for the request.
`nsdict`	`{}`	Namespace dictionary to send in the SOAP `Envelope`
`port`	`80` or `443`	Port to connect on.
`soapaction`	`http://www.zolera.com`	Value for the `SOAPAction` HTTP header.
`readerclass`	`None`	Class used to create DOM-creating XML readers; see the description in the `ParsedSoap` class. New in version 1.2.
`ssl`	`0`	Use SSL if non-zero.
`tracefile`	`None`	An object with a `write` method, where packet traces will be recorded.
`url`	n/a	URL to post to.

If using SSL, the `cert_file` and `key_file` keyword parameters may also be used. For details 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 are 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 is valid SOAP but contains a fault.

Send(*url*, *opname*, *pyboj*[, ***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
`auth_header`	`None`	String (containing presumably serialized XML) to output as an authentication header.
SOAP `Envelope` `nsdict`	`{}`	Namespace dictionary to send in the SOAP `Envelope`
`requestclass`	n/a	Python `class` object with a `typecode` attribute specifying how to serialize the data.
`requesttypecode`	n/a	Typecode specifying how to serialize the data.
`soapaction`	Obtained from the `Binding`	Value for the `SOAPAction` 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 `mimetools` object.

reply_msg

A text string containing the HTTP reply text.

Finally, if an attribute is fetched other than one of those described above, 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.

opname(*args...*)

Using this shortcut requires that the `SetURL()` was invoked first. This method is then equivalent to: '`RPC(None, opname, tuple(args), TC.Any())`'

CHAPTER

ELEVEN

WSDL Support

The `ZSI` and `ZSI.wstools` modules provide client tools for using WSDL 1.1 (see *WSDL 1.1 specification*).

ZSI provides two ways of accessing a WSDL service. The first provides an easy-to-use interface, but requires setting all type codes manually. It is easier to use with simple services than with those specifying many complex types. The second method requires the use of a more complex interface, but automatically generates type codes and classes that correspond to XML Schema types, as well as client stub code. Both use a `WSDL` instance internally to send and receive messages (see section 11.4 for more information on the `WSDL` class).

The first way of accessing a service is through the `ServiceProxy` class. Once the proxy has been created, each remote operation is exposed as a method on the object. The user must handle the generation of type codes. Note that while `ServiceProxy` is part of `ZSI`, it must be explicitly imported.

The second method uses wsdl2py. Handling XML Schema (see *XML Schema specification*) is one of the more difficult aspects of using WSDL. The class `WriteServiceModule`, which wsdl2py uses, helps to hides these details. It generates a module with stub code for the client interface, and a module that encapsulates the handling of XML Schema, automatically generating type codes.

11.1 WSDLReader

The `WSDLReader` class in `ZSI.wstools.WSDLTools` provides methods for loading WSDL service descriptions from URLs, XML files or XML string data, and creating a `WSDL` object. It is used by `ServiceProxy` and `WriteServiceModule`.

`WSDL` instances represent WSDL service descriptions and provide a low-level object model for building and working with those descriptions.

The WSDL reader is implemented as a separate class to make it easy to create custom readers that implement caching policies or other optimizations.

class `WSDLReader`()
The following methods are available:

`loadFromStream`(*file*)
Return a `WSDL` instance representing the service description contained in *file*. The *file* argument must be a file-like object.

`loadFromString`(*data*)
Returns a `WSDL` instance loaded from the XML string *data*.

`loadFromFile`(*filename*)
Returns a `WSDL` instance loaded from the file named by *filename*.

`loadFromURL`(*url*)
Returns a `WSDL` instance loaded from the given *url*.

11.2 ServiceProxy

The `ServiceProxy` class provides calls to web services. A WSDL description must be available for the service. `ServiceProxy` uses `WSDLReader` internally to load a `WSDL` instance.

The user may build up a type codes module for use by `ServiceProxy`.

class `ServiceProxy`(*wsdl*,[, *service*[, *port*]])

The *wsdl* argument may be either the URL of the service description or an existing `WSDL` instance. The optional *service* and *port* name the service and port within the WSDL description that should be used. If not specified, the first defined service and port will be used.

The following keyword arguments may be used:

Keyword	Default	Description
`nsdict`	`{}`	Namespace dictionary to send in the SOAP `Envelope`
`tracefile`	`None`	An object with a `write` method, where packet traces will be recorded.

A `ServiceProxy` instance, once instantiated, exposes callable methods that reflect the methods of the remote web service it represents. These methods may be called like normal methods, using *either* positional or keyword arguments (but not both).

The methods can be called with either positional or keyword arguments; the argument types must be compatible with the types specified in the WSDL description.

When a method of a `ServiceProxy` is called with positional arguments, the arguments are mapped to the SOAP request based on the parameter order defined in the WSDL description. If keyword arguments are passed, the arguments will be mapped based on their names.

11.2.1 Example

The following example, using ServiceProxy, shows a simple language translation service that makes use of the complex type structures defined in the module BabelTypes:

```
from ZSI import ServiceProxy
import BabelTypes

service = ServiceProxy('http://www.xmethods.net/sd/BabelFishService.wsdl',
       typesmodule=BabelTypes)
value = service.BabelFish('en_de', 'This is a test!')
```

The return value from a proxy method depends on the SOAP signature. If the remote service returns a single value, that value will be returned. If the remote service returns multiple "out" parameters, the return value of the proxy method will be a dictionary containing the out parameters indexed by name. Because `ServiceProxy` makes use of the ZSI serialization / deserialization engine, complex return types are supported. This means that an aggregation of primitives can be returned from or passed to a service invocation according to any predefined hierarchical structure.

11.3 Code Generation from WSDL and XML Schema

This section covers wsdl2py, the second way ZSI provides to access WSDL services. Given the path to a WSDL service, two files are generated, a 'service' file and a 'types' file, that one can then use to access the service. For example, to generate code to access the TerraServer database, the script can be called as follows:

```
wsdl2py http://terraservice.net/TerraService.asmx?WSDL
```

To generate the 'service' file, wsdl2py uses the `WriteServiceModule` class in `ZSI.wsdl2python`. `WriteServiceModule` transforms the definitions in a WSDL instance to remote proxy interfaces in the 'service' file. To generate the 'types' file, wsdl2py transforms the XML Schema instances in the WSDL types section to type codes that describe the data.

The `WSDL` (see section 11.4) class and `ZSI.wstools.XMLSchema` module provide API's into the definitions, which `ModuleWriter` and its underlying generator classes use to interpret WSDL and XML Schema into class definitions.

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

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

11.3.1 WriteServiceModule Class Description

class `WriteServiceModule`(*wsdl,* [*importlib,* [*typewriter*]])

This class generates a module containing client stub code, and a module encapsulating the use of XML Schema instances, given a `WSDL` instance generated by `WSDLReader`. It handles import, namespace, and schema complexities, and class definition order.

`WriteServiceModule` delegates to `ServiceDescription` the interpretation of the service definition, and to `SchemaDescription` the interpretation of the schema definition. (These two classes are only intended to be called by `WriteServiceModule`, but are described here to indicate what is going on behind the scenes.)

The following method is available:

`write`()

Generates the client code.

class `ServiceDescription`()

Interprets the service definition, and creates the client interface and port descriptions code. It delegates to `MessageWriter`, which generates a message's class definition, and to `PartWriter`, which generates a message part's description.

class `SchemaDescription`()

Interprets the schema definition, generating typecode module code for all global definitions and declarations in a schema instance. It delegates to `TypeWriter`, which generates a type's description/typecode.

The following excerpt is from wsdl2py, and illustrates how `WriteServiceModule` is used. The input is a path name of a WSDL file or URL.

```
...
reader = WSDLTools.WSDLReader()
if args_d['fromfile']:
    wsdl = reader.loadFromFile(args_d['wsdl'])
elif args_d['fromurl']:
    wsdl = reader.loadFromURL(args_d['wsdl'])
wsm = ZSI.wsdl2python.WriteServiceModule(wsdl)
wsm.write()
...
```

11.3.2 Example Use of Generated Code

The following shows how to call a proxy method for ConvertPlaceToLonLatPt, a method provided by a service named TerraService. It assumes that wsdl2py has already been called. In this case, the 'services' and 'types' files generated would be named TerraService_services.py and TerraService_services_types.py, respectively.

```
from TerraService_services import *

import sys

def main():
    loc = TerraServiceLocator()

       # prints messages sent and received if tracefile is set
    kw = { 'tracefile' : sys.stdout }
    portType = loc.getTerraServiceSoap(**kw)

    # --------------------------------
    # From "TerraService_services.py"
    #
    #  ConvertPlaceToLonLatPtSoapIn = ns0.ConvertPlaceToLonLatPt_Dec().pyclass

    # --------------------------------
    # From "TerraService_services_types.py"
    #
    #  class ConvertPlaceToLonLatPt_Dec(ZSI.TCcompound.Struct, ElementDeclaration):
    #      ...
    #      class Holder(SetAddByAname):
    #          typecode = self
    #          def __init__(self):
    #              # pyclass
    #              self._place = None
    #
    class Holder: pass
    request = ConvertPlaceToLonLatPtSoapIn()
    request._place = Holder()
    request._place._City = 'Oak Harbor'
    request._place._State = 'Washington'
    request._place._Country = 'United States'

    response = portType.ConvertPlaceToLonLatPt(request)
    print "Latitude = %s" % response._ConvertPlaceToLonLatPtResult._Lat
    print "Longitude = %s" % response._ConvertPlaceToLonLatPtResult._Lon
...
```

One needs to look at the associated WSDL file to see how to use the classes and methods in the generated code. In this example, `TerraServiceLocator` is a class with the name of the WSDL service, plus 'Locator'. It contains the information necessary to contact the service, using `getTerraServiceSoap(**kw)`.

That method's name is generated by 'get' plus the name of the WSDL portType for the service. It returns a class which encapsulates the information in the portType, and contains the proxies for the methods associated with it.

`ConvertPlaceToLonLatPtSoapInWrapper()`'s name is generated using the name of the WSDL input message for the ConvertPlaceToLonLatPt WSDL operation, plus 'Wrapper'. The actual call to the service is a method of the class encapsulating the portType, with the same name as the WSDL operation.

The name of the response field is '_' plus the name of the WSDL element (or one contained by the element) returned by the call. Parameters that are input and output are subfields of the request object and the response field, respectively. Their names can be determined by looking at the part sub-element of a message.

If the user wishes to set authorization headers in a request, using the previous example, it would be accomplished like this:

```
def main():
    loc = TerraServiceLocator()
    portType = loc.getTerraServiceSoap(
        tracefile=sys.stdout,
        auth=(soap.ZSI.AUTH.httpbasic, 'logname', 'password'))
    # ...
```

11.4 WSDL objects

The following classes described encapsulate the upper-level objects in a WSDL file. Note that most users will not need to use these, given the availability of `WriteServiceModule` and `ServiceProxy`, which are built on top of these objects.

There are quite many classes defined here to implement the WSDL object model. Instances of those classes are generally accessed and created through container objects rather than instantiated directly. Most of them simply implement a straightforward representation of the WSDL elements they represent. The interfaces of these objects are described in the next section.

An exception is defined for errors that occur while creating WSDL objects.

exception `WSDLError`
: This exception is raised when errors occur in the parsing or building of WSDL objects, usually indicating invalid structure or usage. It is a subtype of Python's `Exception` class.

class `WSDL()`
: `WSDL` instances implement the WSDL object model. They are created by loading an XML source into a `WSDLReader` object.

  A `WSDL` object provides access to all of the structures that make up a web service description. The various "collections" in the WSDL object model (services, bindings, portTypes, etc.) are implemented as `Collection` objects that behave like ordered mappings.

  The following attributes are read-only:

  `name`
  : The name of the service description (associated with the *definitions* element), or `None` if not specified.

  `targetNamespace`
  : The target namespace associated with the service description, or `None` if not specified.

  `documentation`
  : The documentation associated with the *definitions* element of the service description, or the empty string if not specified.

  `location`
  : The URL from which the service description was loaded, or `None` if the description was not loaded from a URL.

  `services`
  : A collection that contains `Service` objects that represent the services that appear in the service description. The items of this collection may be indexed by name or ordinal.

messages
A collection that contains `Message` objects that represent the messages that appear in the service description. The items of this collection may be indexed by name or ordinal.

portTypes
A collection that contains `PortType` objects that represent the portTypes that appear in the service description. The items of this collection may be indexed by name or ordinal.

bindings
A collection that contains `Binding` objects that represent the bindings that appear in the service description. The items of this collection may be indexed by name or ordinal.

imports
A collection that contains `ImportElement` objects that represent the import elements that appear in the service description. The items of this collection may be indexed by ordinal or the target namespace URI of the import element.

types
A `Types` instance that contains `XMLSchema` objects that represent the schemas defined or imported by the WSDL description. The `Types` object may be indexed by ordinal or by targetNamespace to lookup schema objects.

extensions
A sequence of objects that represent WSDL *extension elements*. These objects may be instances of classes that represent WSDL-defined extensions (`SoapBinding`, `SoapBodyBinding`, etc.), or DOM `Element` objects for unknown extensions.

class `Service()`
A `Service` object represents a WSDL `<service>` element.

The following attributes are read-only:

name
The name of the service.

documentation
The documentation associated with the element, or an empty string.

ports
A collection that contains `Port` objects that represent the ports defined by the service. The items of this collection may be indexed by name or ordinal.

extensions
A sequence of any contained WSDL extensions.

The following method is available:

`getWSDL()`
Return the parent `WSDL` instance of the object.

class `Port()`
A `Port` object represents a WSDL `<port>` element.

The following attributes are read-only:

name
The name of the port.

documentation
The documentation associated with the element, or an empty string.

binding
The name of the binding associated with the port.

extensions
A sequence of any contained WSDL extensions.

The following methods are available:

getAddressBinding()
A convenience method that returns the address binding extension for the port, either a `SoapAddressBinding` or `HttpAddressBinding`. Raises `WSDLError` if no address binding is found.

getService()
Return the parent `Service` instance of the object.

getBinding()
Return the `Binding` instance associated with the port.

getPortType()
Return the `PortType` instance associated with the port.

class PortType()
A `PortType` object represents a WSDL `<portType>` element.

The following attributes are read-only:

name
The name of the portType.

documentation
The documentation associated with the element, or an empty string.

operations
A collection that contains `Operation` objects that represent the operations in the portType. The items of this collection may be indexed by name or ordinal.

The following method is available:

getWSDL()
Return the parent `WSDL` instance of the object.

class Operation()
A `Operation` object represents a WSDL `<operation>` element within a `portType` element.

The following attributes are read-only:

name
The name of the operation.

documentation
The documentation associated with the element, or an empty string.

parameterOrder
A string representing the `parameterOrder` attribute of the operation, or `None` if the attribute is not defined.

input
A `MessageRole` instance representing the `<input>` element of the operation binding, or `None` if no input element is present.

output
A `MessageRole` instance representing the `<output>` element of the operation, or `None` if no output element is present.

faults
A collection of `MessageRole` instances representing the `<fault>` elements of the operation.

The following methods are available:

getPortType()
Return the parent `PortType` instance of the object.

getInputMessage()
Return `Message` object associated with the input to the operation.

getOutputMessage()
Return `Message` object associated with the output of the operation.

getFaultMessage(*name*)
Return `Message` object associated with the named fault.

class MessageRole()
`MessageRole` objects represent WSDL `<input>`, `<output>` and `<fault>` elements within an operation.

The following attributes are read-only:

name
The name attribute of the element.

type
The type of the element, one of `'input'`, `'output'` or `'fault'`.

message
The name of the message associated with the object.

documentation
The documentation associated with the element, or an empty string.

class Binding()
A `Binding` object represents a WSDL `<binding>` element.

The following attributes are read-only:

name
The name of the binding.

documentation
The documentation associated with the element, or an empty string.

type
The name of the portType the binding is associated with.

operations
A collection that contains `OperationBinding` objects that represent the contained operation bindings.

extensions
A sequence of any contained WSDL extensions.

The following methods are available:

getWSDL()
Return the parent `WSDL` instance of the object.

getPortType()
Return the `PortType` object associated with the binding.

findBinding(*kind*)
Find a binding extension in the binding. The *kind* can be a class object if the wanted extension is one of the WSDL-defined types (such as `SoapBinding` or `HttpBinding`). If the extension is not one of the supported types, *kind* can be a tuple of the form `(namespace-URI, localname)`, which will be used to try to find a matching DOM `Element`.

findBindings(*kind*)
The same as `findBinding()`, but will return multiple values of the given *kind*.

class OperationBinding()
A `OperationBinding` object represents a WSDL `<operation>` element within a binding element.

The following attributes are read-only:

name
The name of the operation binding.

documentation
The documentation associated with the element, or an empty string.

input
A `MessageRoleBinding` instance representing the `<input>` element of the operation binding, or `None` if no input element is present.

output
A `MessageRoleBinding` instance representing the `<output>` element of the operation binding, or `None` if no output element is present.

faults
A collection of `MessageRoleBinding` instances representing the `<fault>` elements of the operation binding.

extensions
A sequence of any contained WSDL extensions.

The following methods are available:

getBinding()
Return the parent `Binding` instance of the operation binding.

getOperation()
Return the abstract `Operation` associated with the operation binding.

findBinding(*kind*)
Find a binding extension in the operation binding. The *kind* can be a class object if the wanted extension is one of the WSDL-defined types (such as `SoapOperationsBinding` or `HttpOperationBinding`).

If the extension is not one of the supported types, *kind* can be a tuple of the form `(namespace-URI, localname)`, which will be used to try to find a matching DOM `Element`.

findBindings(*kind*)
The same as `findBinding()`, but will return multiple values of the given *kind*.

class MessageRoleBinding()
`MessageRoleBinding` objects represent WSDL `<input>`, `<output>` and `<fault>` elements within an operation binding.

The following attributes are read-only:

name
The name attribute of the element, for fault elements. This is always `None` for input and output elements.

type
The type of the element, one of `'input'`, `'output'` or `'fault'`.

documentation
The documentation associated with the element, or an empty string.

extensions
A sequence of any contained WSDL extensions.

The following methods are available:

findBinding(*kind*)
Find a binding extension in the message role binding. The *kind* can be a class object if the wanted extension is one of the WSDL-defined types.

If the extension is not one of the supported types, *kind* can be a tuple of the form `(namespace-URI, localname)`, which will be used to try to find a matching DOM `Element`.

findBindings(*kind*)
The same as `findBinding()`, but will return multiple values of the given *kind*.

class Message()
A `Message` object represents a WSDL `<message>` element.

The following attributes are read-only:

name
The name of the message.

documentation
The documentation associated with the element, or an empty string.

parts
A collection that contains `MessagePart` objects that represent the parts of the message. The items of this collection may be indexed by name or ordinal.

class `MessagePart`()
A `MessagePart` object represents a WSDL `<part>` element.

The following attributes are read-only:

name
The name of the message part.

documentation
The documentation associated with the element, or an empty string.

type
A tuple of the form `(namespace-URI, localname)`, or `None` if the `type` attribute is not defined.

element
A tuple of the form `(namespace-URI, localname)`, or `None` if the `element` attribute is not defined.

class `Types`()
The following attributes are read-only:

A `Types` object represents a WSDL `<types>` element. It acts as an ordered collection containing `XMLSchema` instances associated with the service description (either directly defined in a `<types>` element, or included via import). The `Types` object can be indexed by ordinal or by the `targetNamespace` of the contained schemas.

documentation
The documentation associated with the element, or an empty string.

extensions
A sequence of any contained WSDL extensions.

The following method is available:

`getWSDL`()
Return the parent `WSDL` instance of the object.

class `ImportElement`()
A `ImportElement` object represents a WSDL `<import>` element.

The following attributes are read-only:

namespace
The namespace attribute of the import element.

The following method is available:

location
The location attribute of the import element.

11.4.1 Binding Classes

The `WSDLTools` module contains a number of classes that represent the binding extensions defined in the WSDL specification. These classes are straightforward, reflecting the attributes of the corresponding XML elements, so they are not documented exhaustively here.

class `SoapBinding`(*transport*[, *style*])
Represents a `<soap:binding>` element.

class `SoapAddressBinding`(*location*)
Represents a `<soap:address>` element.

class `SoapOperationBinding`()
Represents a `<soap:operation>` element.

class `SoapBodyBinding`()
Represents a `<soap:body>` element.

class `SoapFaultBinding`()
Represents a `<soap:fault>` element.

class `SoapHeaderBinding`()
Represents a `<soap:header>` element.

class `SoapHeaderFaultBinding`()
Represents a `<soap:headerfault>` element.

class `HttpBinding`()
Represents a `<http:binding>` element.

class `HttpAddressBinding`()
Represents a `<http:address>` element.

class `HttpOperationBinding`()
Represents a `<http:operation>` element.

class `HttpUrlReplacementBinding`()
Represents a `<http:urlReplacement>` element.

class `HttpUrlEncodedBinding`()
Represents a `<http:urlEncoded>` element.

class `MimeMultipartRelatedBinding`()
Represents a `<mime:multipartRelated>` element.

class `MimePartBinding`()
Represents a `<mime:part>` element.

class `MimeContentBinding`()
Represents a `<mime:content>` element.

class `MimeXmlBinding`()
Represents a `<mime:mimeXml>` element.

CHAPTER

TWELVE

ZSI Schema

The `ZSI` schema defines two sets of elements. One is used to enhance the SOAP Fault `detail` element, and to report header errors. The other is used to define a header element containing a name and password, for a class of basic authentication.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.zolera.com/schemas/ZSI/"
  xmlns:SOAPFAULT="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://www.zolera.com/schemas/ZSI/">

  <import namespace="http://schemas.xmlsoap.org/soap/envelope/"
    schemaLocation="http://schemas.xmlsoap.org/soap/envelope/"/>

  <!--  Soap doesn't define a fault element to use when we want
        to fault because of header problems. -->
  <element name="detail" type="SOAPFAULT:detail"/>

  <!--  A URIFaultDetail element typically reports an unknown
        mustUnderstand element. -->
  <element name="URIFaultDetail" type="tns:URIFaultDetail"/>
  <complexType name="URIFaultDetail">
    <sequence>
      <element name="URI" type="anyURI" minOccurs="1"/>
      <element name="localname" type="NCName" minOccurs="1"/>
      <any minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <!--  An ActorFaultDetail element typically reports an actor
        attribute was found that cannot be processed. -->
  <element name="ActorFaultDetail" type="tns:ActorFaultDetail"/>
  <complexType name="ActorFaultDetail">
    <sequence>
      <element name="URI" type="anyURI" minOccurs="1"/>
      <any minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <!--  A ParseFaultDetail or a FaultDetail element are typically
        used when there was parsing or "business-logic" errors.
        The TracedFault type is intended to provide a human-readable
        string that describes the error (in more detail then the
        SOAP faultstring element, which is becoming codified),
        and a human-readable "trace" (optional) that shows where
```

```
        within the application that the fault happened. -->
    <element name="ParseFaultDetail" type="tns:TracedFault"/>
    <element name="FaultDetail" type="tns:TracedFault"/>
    <complexType name="TracedFault">
      <sequence>
        <element name="string" type="string" minOccurs="1"/>
        <element name="trace" type="string" minOccurs="0"/>
        <!-- <any minOccurs="0" maxOccurs="unbounded"/> -->
      </sequence>
    </complexType>

    <!--  An element to hold a name and password, for doing basic-auth. -->
    <complexType name="BasicAuth">
      <sequence>
        <element name="Name" type="string" minOccurs="1"/>
        <element name="Password" type="string" minOccurs="1"/>
      </sequence>
    </complexType>

  </schema>
```