ZSI: The Zolera Soap Infrastructure

Release 1.2.6

Rich Salz

May 14, 2002

Zolera Systems, http://www.zolera.com
E-mail: rsalz@zolera.com

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/.

COPYRIGHT

Acknowledgments

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

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` homepage is at http://www.zolera.com/resources/opensrc/zsi. `ZSI` is discussed on the Python web services mailing list, visit http://lists.sourceforge.net/lists/listinfo/pywebsvcs-talk to subscribe.

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.

Currently, the most cumbersome part of using `ZSI` is defining the typecode objects. A future release of `ZSI` may be able to process WSDL definitions (described in *The Web Services Description Language*) and generate typecodes automatically.

CHAPTER

TWO

Examples

This section contains two examples. The first shows how to use `ZSI` to expose conventional CGI scripts through SOAP. The input parameters and return value are Python lists and the SOAP messages must contain all type information. This is appropriate for building simple schema-less applications. The second example shows how to to create a more comprehensive application that uses `ZSI` to validate its input and output against a schema.

2.1 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.

If the above script is installed on the webserver running on the local host, and if the URL is `/cgi-bin/simple-test`, then the following code shows simple binding and access to the server:

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

2.2 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 EvaluateError, 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)
```

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. 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.

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. To use the cDomlette DOM provided by the 4Suite package, use the RawExpatReader class in the Ft.Lib.cDomlette 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.

CHAPTER

SIX

The `TypeCode` classes — data conversions

The `TypeCode` module defines classes used for converting data between SOAP data and local Python objects. 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.

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, local-name)`' 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.

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, and we want 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):
    # We don't set parselist, since this data is typed as a string
    # for interoperability with other SOAP implementations.
    #parselist = [ ('tns', 'xpath') ]

    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
```

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.

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.

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.typecodes = 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 write in the SOAP `Header`.
`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.

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=None*[, ***kw*]])

This method serializes the `Fault` object into a SOAP message. If the `output` parameter is not specified, 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=None*]`)`
: 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=None*[*, actor=None*]]`)`
: 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. It should only be used if the `IsAFault()` method returned true.

`FaultFromNotUnderstood``(`*uri, localname,*[*, actor=None*]`)`
: 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=None*]`)`
: This function creates a `Fault` object from a `ZSI` exception, `ParseException` or `EvaluateException`. 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.

Two dispatch mechanisms are provided: one supports standard CGI scripts, and the other runs a dedicated server based on the `BaseHTTPServer` module.

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.
port	80	Port to listen on.

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` module. 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.

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 th 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 th `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
`nsdict`	`{}`	Namespace dictionary to send in the SOAP `Envelope`
`requestclass`	n/a	Python `class` object with a `typecode` attribute specifying how to alize 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

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>
```