suds-py3 Documentation

Release 1.3.1

cackharot

Jan 08, 2021

Contents:

Suds is a lightweight SOAP python client that provides a service proxy for Web Services.

CHAPTER 1

Overview

The goal of suds is to present an RPC-'like' interface into soap-based web services. This means that in most cases, users do not need to be concerned with the complexities of the WSDL and referenced schemas. Regardless of which soap message style is specified, the signature of the service methods remain the same. Uses that do examine the WSDL will notice that even with the 'document' soap message style, the signature of each method resembles an RPC. The method signature contains the contents of the 'document' defined for the message instead of the document itself.

Basic features:

- No class generation
- Provides an object-like API.
- Reads wsdl at runtime for encoding/decoding
- **Provides for the following SOAP (style) binding/encoding:**
    - Document/Literal
    - RPC/Literal
    - RPC/Encoded (section 5)

The goal of suds is to present an RPC-'like' interface into soap-based web services. This means that in most cases, users do not need to be concerned with the complexities of the WSDL and referenced schemas. Regardless of which soap message style is specified, the signature of the service methods remain the same. Uses that do examine the WSDL will notice that even with the 'document' soap message style, the signature of each method resembles an RPC. The method signature contains the contents of the 'document' defined for the message instead of the document itself.

CHAPTER 2

Logging

The ‘suds’ package use the Python standard lib logging package, all messages are at level DEBUG or ERROR.

To register a console handler you can use basicConfig:

```
#!python
import logging
logging.basicConfig(level=logging.INFO)
```

Once the console handler is configured, the user can enable module specific debugging doing the following:

```
logging.getLogger(<desired package>).setLevel(logging.<desired-level>)
```

A common example (show sent/received soap messages):

```
#!python
logging.getLogger('suds.client').setLevel(logging.DEBUG)
```

Suggested modules for debugging:

suds.client:

Set the logging level to ‘DEBUG’ on this module to see soap messages (in & out) and http headers.

suds.transport:

Set the logging level to ‘DEBUG’ on this module to see more details about soap messages (in & out) and http headers.

suds.xsdschema:

Set the logging level to ‘DEBUG’ on this module to see digestion of the schema(s).

suds.wsdl:

Set the logging level to ‘DEBUG’ on this module to see digestion WSDL.

CHAPTER 3

Basic Usage

The 'suds' [suds.client.Client-class.html Client] class provides a consolidated API for consuming web services. The object contains (2) sub-namespaces:

service:

> The [suds.client.Service-class.html service] namespace provides a proxy for the consumed service. This object is used to invoke operations (methods) provided by the service endpoint.

factory:

> The [suds.client.Factory-class.html factory] namespace provides a factory that may be used to create instances of objects and types defined in the WSDL.

You will need to know the url for WSDL for each service used. Simply create a client for that service as follows:

```
#!python
from suds.client import Client
url = 'http://localhost:7080/webservices/WebServiceTestBean?wsdl'
client = Client(url)
```

You can inspect service object with: `__str__()` as follows to get a list of methods provide by the service:

```
#!python
print client

Suds - version: 1.3.3.1 build: IN 20180220

Service (WebServiceTestBeanService) tns="http://test.server.enterprise.rhq.org/"
 Prefixes (1):
   ns0 = "http://test.server.enterprise.rhq.org/"
 Ports (1):
   (Soap)
     Methods:
       addPerson(Person person, )
       echo(xs:string arg0, )
       getList(xs:string str, xs:int length, )
```

(continues on next page)

(continued from previous page)

```
        getPercentBodyFat(xs:string name, xs:int height, xs:int weight)
        getPersonByName(Name name, )
        hello()
        testExceptions()
        testListArg(xs:string[] list, )
        testVoid()
        updatePerson(AnotherPerson person, name name, )
 Types (23):
   Person
   Name
   Phone
   AnotherPerson
```

Note: See example of service with multiple ports below.

The sample output lists that the service named `WebServiceTestBeanService` has methods such as `getPercentBodyFat()` and `addPerson()`.

3.1 Simple Arguments

Let's start with the simple example. The `getPercentBodyFat()` method has the signature of `getPercentBodyFat('xs:string' name, 'xs:int' height, 'xs:int' weight)`. In this case, the parameters are 'simple' types. That is, they not objects. This method would be invoked as follows:

```
#!python
result = client.service.getPercentBodyFat('jeff', 68, 170)
print result

You have 21% body fat.

#!python
result = client.service.getPercentBodyFat(name='jeff', height=68, weight=170)
print result

You have 21% body fat.

#!python
d = dict(name='jeff', height=68, weight=170)
result = client.service.getPercentBodyFat(**d)
print result

You have 21% body fat.
```

3.2 Complex Arguments

The `addPerson()` method takes a 'person' argument of type: 'Person' and has a signature of: `addPerson('Person' person, )` where parameter type is printed followed by it's name.

There is a type (or class) named 'person' which is coincidentally the same name as the argument. Or in the case of `getPercentBodyFat()` the parameters are **string** of type *xs:string* and **integer** of type *xs:int*.

So, to create a 'Person' object to pass as an argument we need to get a person argument using the 'factory' sub-namespace as follows:

```
#!python
person = client.factory.create('Person')
print person

(Person)=
{
 phone = []
 age = NONE
 name(Name) =
     {
         last = NONE
         first = NONE
     }
}
```

As you can see, the object is created as defined by the WSDL. The list of phone number is empty so we'll have to create a 'Phone' object:

```
#!python
phone = client.factory.create('Phone')
phone.npa = 202
phone.nxx = 555
phone.number = 1212
```

... and the name (Name object) and age need to be set and we need to create a name object first:

```
#!python
name = client.factory.create('Name')
name.first = 'Elmer'
name.last = 'Fudd'
```

Now, let's set the properties of our 'Person' object:

```
#!python
person.name = name
person.age = 35
person.phone = [phone]
```

or:

```
#!python
person.phone.append(phone)
```

... and invoke our method named `addPerson()` as follows:

```
#!python
try:
    person_added = client.service.addPerson(person)
except WebFault, e:
    print e
```

It's that easy.

Users may **not** use python 'dict' for complex objects when they are subclasses (or extensions) of types defined in the wsdl/schema. In other words, if the schema defines a type to be an 'Animal' and you wish to pass a 'Dog' (assumes Dog 'isa' Animal), you may **not** use a 'dict' to represent the dog. In this case, suds needs to set the *xsi:type="Dog"* but cannot because the python 'dict' does not provide enough information to indicate that it is a 'Dog' not an 'Animal'. Most likely, the server will reject the request and indicate that it cannot instantiate a abstract 'Animal'.

3.3 Complex Arguments Using Python (dict)

Just like the factory example, let's assume the `addPerson()` method takes a 'person' argument of type: 'Person'. So, to create a 'Person' object to pass as an argument we need to get a person object and we can do so by creating a simple python 'dict'.:

```
#!python
person = {}
```

According to the WSDL we know that the Person contains a list of Phone objects so we'll need 'dict's for them as well:

```
#!python
phone = {
    'npa':202,
    'nxx':555,
    'number':1212,
}
```

... and the name (Name object) and age need to be set and we need to create a name object first:

```
#!python
name = {
    'first':'Elmer',
    'last':'Fudd'
}
```

Now, let's set the properties of our 'Person' object:

```
#!python
person['name'] = name
person['age'] = 35
person['phone'] = [phone,]
```

... and invoke our method named `addPerson()` as follows:

```
#!python
try:
   person_added = client.service.addPerson(person)
except WebFault, e:
  print e
}}}
```

3.4 Faults

The Client can be configured to throw web faults as `WebFault` or to return a tuple `(<status>, <returned-value>)` instead as follows:

```
#!python
client = client(url, faults=False)
result = client.service.addPerson(person)
print result

( 200, person ...)
```

3.5 Options

The 'suds' [suds.client.Client-class.html client] has many that may be used to control the behavior of the library. Some are [suds.options.Options-class.html general options] and others are [suds.transport.options.Options-class.html transport options]. Although, the options objects are exposed, the preferred and supported way to set/unset options is through

- The [suds.client.Client-class.html Client] constructor
- The [suds.client.Client-class.html Client].set_options()
- The [suds.transport.Transport-class.html Transport] constructor(s).

They are as follows:

faults:

Controls web fault behavior.

service:

Controls the default service name for multi-service wsdls.

port:

Controls the default service port for multi-port services.

location:

This overrides the service port address 'URL' defined in the WSDL.

proxy:

Controls http proxy settings.

transport:

Controls the 'plugin' web [suds.transport.Transport-class.html transport].

cache:

Provides caching of documents and objects related to loading the WSDL. Soap envelopes are never cached.

cachingpolicy:

The caching policy, determines how data is cached. The default is 0.

- 0 = XML documents such as WSDL & XSD.
- 1 = WSDL object graph.

headers:

Provides for 'extra' http headers.

soapheaders:

Provides for soap headers.

wsse:

Provides for WS-Security object.

__inject:

Controls message/reply message injection.

doctor:

The schema 'doctor' specifies an object used to fix broken schema(s).

xstq:

The 'X'ML 's'chema 't'ype 'q'ualified flag indicates that 'xsi:type' attribute **values** should be qualified by namespace.

prefixes:

Elements of the soap message should be qualified (when needed) using XML prefixes as opposed to xmlns="" syntax.

timeout:

The URL connection timeout (seconds) default=90.

retxml:

Flag that causes the I{raw} soap envelope to be returned instead of the python object graph.

autoblend:

Flag that ensures that the schema(s) defined within the WSDL import each other.

nosend:

Flag that causes suds to generate the soap envelope but not send it. Instead, a [suds.client.RequestContext-class.html RequestContext] is returned Default: False.

CHAPTER 4

Enumerations

Enumerations are handled as follows

Let's say the wsdl defines the following enumeration:

```
#!xml
<xs:simpleType name="resourceCategory">
<xs:restriction base="xs:string">
  <xs:enumeration value="PLATFORM"/>
  <xs:enumeration value="SERVER"/>
  <xs:enumeration value="SERVICE"/>
</xs:restriction>
</xs:simpleType>
```

The client can instantiate the enumeration so it can be used.

Misspelled references to elements of the 'enum' will raise a `AttrError` exception as:

```
#!python
resourceCategory = client.factory.create('resourceCategory')
client.service.getResourceByCategory(resourceCategory.PLATFORM)
```

CHAPTER 5

Factory

The [suds.client.Factory-class.html factory] is used to create complex objects defined the the wsdl/schema. This is **not** necessary for parameters or types that are specified as 'simple' types such as *xs:string, xs:int, etc...*

The `create()` method should always be used because it returns objects that already have the proper structure and schema-type information. Since xsd supports nested type definition, so does `create()` using the `(.)` dot notation. For example suppose the `(Name)` type was not defined as a top level "named" type but rather defined within the (Person) type. In this case creating a `(Name)` object would have to be qualified by it's parent's name using the dot notation as follows:

```
#!python
name = client.factory.create('Person.Name')
```

If the type is in the same namespace as the wsdl `(targetNamespace)` then it may be referenced without any namespace qualification. If not, the type must be qualified by either a namespace prefix such as:

```
#!python
name = client.factory.create('ns0:Person')
```

Or, the name can be fully qualified by the namespace itself using the full qualification syntax as:

```
#!python
name = client.factory.create('{http://test.server.enterprise.rhq.org/}person')
```

Qualified names can only be used for the 'first' part of the name, when using `(.)` dot notation to specify a path.

CHAPTER 6

Services with multiple ports

Some services are defined with multiple ports as:

```
#!xml
<wsdl:service name="BLZService">
<wsdl:port name="soap" binding="tns:BLZServiceSOAP11Binding">
  <soap:address location="http://www.thomas-bayer.com:80/axis2/services/BLZService"/>
</wsdl:port>
<wsdl:port name="soap12" binding="tns:BLZServiceSOAP12Binding">
  <soap12:address location="http://www.thomas-bayer.com:80/axis2/services/BLZService"/
↪>
</wsdl:service>
```

And are reported by suds as:

```
#!python
url = 'http://www.thomas-bayer.com/axis2/services/BLZService?wsdl'
client = Client(url)
print client

Suds - version: 1.3.3.1 build: IN 20180220

Service (BLZService) tns="http://thomas-bayer.com/blz/"
 Prefixes (1)
   ns0 = "http://thomas-bayer.com/blz/"
 Ports (2):
    (soap)
      Methods (1):
        getBank(xs:string blz, )
    (soap12)
      Methods (1):
        getBank(xs:string blz, )
 Types (5):
    getBankType
    getBankResponseType
```

(continues on next page)

(continued from previous page)

```
    getBankType
    getBankResponseType
    detailsType
```

This example only has (1) method defined for each port but it could very likely have may methods defined. Suds does not require the method invocation to be qualified (as shown above) by the port as:

```
#!python
client.service.<port>.getBank()
```

unless the user wants to specify a particular port. In most cases, the server will work properly with any of the soap ports. However, if you want to invoke the getBank() method on this service the user may qualify the method name with the port.

There are (2) ways to do this:

- Select a default port using the 'port' [suds.options.Options-class.html option] before invoking the method as:

  ```
  #!python
  client.set_options(port='soap')
  client.service.getBank()
  ```

- fully qualify the method as:

  ```
  #!python
  client.service.soap.getBank()
  ```

Support multiple-services within (1) WSDL as follows:

This example only has (1) method defined for each port but it could very likely have may methods defined. Suds does not require the method invocation to be qualifed (as shown above) by the port as:

```
#!python
client.service[port].getBank()
```

unless the user wants to specify a particular port. In most cases, the server will work properly with any of the soap ports. However, if you want to invoke the `getBank()` method on this service the user may qualify the method name with the port. The 'port' may be subscribed either by name (string) or index(int).

There are many ways to do this:

- Select a default port using the 'port' [suds.options.Options-class.html option] before invoking the method as:

  ```
  #!python
  client.set_options(port='soap')
  client.service.getBank()
  ```

- fully qualify the method using the port 'name' as:

  ```
  #!python
  client.service['soap'].getBank()
  ```

- fully qualify the method using the port 'index' as:

  ```
  #!python
  client.service[0].getBank()
  ```

CHAPTER 7

WSDL with multiple services & multiple ports

Some WSDLs define multiple services which may (or may not) be defined with multiple ports as:

```
#!xml
<wsdl:service name="BLZService">
<wsdl:port name="soap" binding="tns:BLZServiceSOAP11Binding">
  <soap:address location="http://www.thomas-bayer.com:80/axis2/services/BLZService"/>
</wsdl:port>
<wsdl:port name="soap12" binding="tns:BLZServiceSOAP12Binding">
  <soap12:address location="http://www.thomas-bayer.com:80/axis2/services/BLZService"/
↪>
</wsdl:service>
<wsdl:service name="OtherBLZService">
<wsdl:port name="soap" binding="tns:OtherBLZServiceSOAP11Binding">
  <soap:address location="http://www.thomas-bayer.com:80/axis2/services/
↪OtherBLZService"/>
</wsdl:port>
<wsdl:port name="soap12" binding="tns:OtherBLZServiceSOAP12Binding">
  <soap12:address location="http://www.thomas-bayer.com:80/axis2/services/
↪OtherBLZService"/>
</wsdl:service>
```

And are reported by suds as:

```
#!python
url = 'http://www.thomas-bayer.com/axis2/services/BLZService?wsdl'
client = Client(url)
print client

Suds - version: 1.3.3.1 build: IN 20180220

Service (BLZService) tns="http://thomas-bayer.com/blz/"
 Prefixes (1)
   ns0 = "http://thomas-bayer.com/blz/"
 Ports (2):
```

(continues on next page)

(continued from previous page)

```
    (soap)
      Methods (1):
        getBank(xs:string blz, )
    (soap12)
      Methods (1):
        getBank(xs:string blz, )
 Types (5):
    getBankType
    getBankResponseType
    getBankType
    getBankResponseType
    detailsType

Service (OtherBLZService) tns="http://thomas-bayer.com/blz/"
 Prefixes (1)
   ns0 = "http://thomas-bayer.com/blz/"
 Ports (2):
    (soap)
      Methods (1):
        getBank(xs:string blz, )
    (soap12)
      Methods (1):
        getBank(xs:string blz, )
 Types (5):
    getBankType
    getBankResponseType
    getBankType
    getBankResponseType
    detailsType
```

This example only has (1) method defined for each port but it could very likely have may methods defined. Suds does **not** require the method invocation to be qualifed (as shown above) by the service and/or port as:

```
#!python
client.service[service][port].getBank()
```

unless the user wants to specify a particular service and/or port. In most cases, the server will work properly with any of the soap ports. However, if you want to invoke the `getBank()` method on the `OtherBLZService` service the user may qualify the method name with the service and/or port. If not specified, suds will default the service to the 1st server defined in the WSDL and default to the 1st port within each service. Also, when a WSDL defines (1) services, the [] subscript is applied to the port selection. This may be a little confusing because the syntax for subscripting can seem inconsistent.

Both the 'service' **and** 'port' may be subscripted either by name (string) or index (int).

There are many ways to do this:

- Select a default service using the 'service' option and default port using 'port' option [suds.options.Options-class.html option] before invoking the method as:

  ```
  #!python
  client.set_options(service='OtherBLZService', port='soap')
  client.service.getBank()
  ```

- method qualified by 'service' and 'port' as:

```
#!python
client.service['OtherBLZService']['soap'].getBank()
```

- method qualified by ‘service’ and ‘port’ using indexes as:

```
#!python
client.service[1][0].getBank()
```

- method qualified by ‘service’ (by name) only as:

```
#!python
client.service['OtherBLZService'].getBank()
```

- method qualified by ‘service’ (by index) only as:

```
#!python
client.service[1].getBank()
```

Note, that if a WSDL defines more then one service, you **must** qualify the ‘service’ via [suds.options.Options-class.html option] or by using the subscripting syntax in order to specify the ‘port’ using the subscript syntax.

CHAPTER 8

SOAP Headers

SOAP headers may be passed during the service invocation by using the ‘soapheaders’ [suds.options.Options-class.html option] as follows:

```
#!python
client = client(url)
token = client.factory.create('AuthToken')
token.username = 'Elvis'
token.password = 'TheKing'
client.set_options(soapheaders=token)
result = client.service.addPerson(person)
```

OR:

```
#!python
client = client(url)
userid = client.factory.create('Auth.UserID')
userid.set('Elvis')
password = client.factory.create('Auth.Password')
password.set('TheKing')
client.set_options(soapheaders=(userid,password))
result = client.service.addPerson(person)
```

OR:

```
#!python
client = client(url)
userid = 'Elmer'
passwd = 'Fudd'
client.set_options(soapheaders=(userid,password))
result = client.service.addPerson(person)
```

The ‘soapheaders’ option may also be assigned a dictionary for those cases when optional headers are specified and users don’t want to pass None place holders. This works much like the method parameters. Eg:

```
#!python
client = client(url)
myheaders = dict(userid='Elmer', passwd='Fudd')
client.set_options(soapheaders=myheaders)
result = client.service.addPerson(person)
```

CHAPTER 9

Custom SOAP Headers

Custom SOAP headers may be passed during the service invocation by using the 'soapheaders' [suds.options.Options-class.html option]. A 'custom' soap header is defined as a header that is required by the service by **not** defined in the wsdl. Thus, the 'easy' method of passing soap headers already described cannot be used. This is done by constructing and passing an [suds.sax.element.Element-class.html Element] or collection of [suds.sax.element.Element-class.html Elements] as follows:

```
#!python
from suds.sax.element import Element
client = client(url)
ssnns = ('ssn', 'http://namespaces/sessionid')
ssn = Element('SessionID', ns=ssnns).setText('123')
client.set_options(soapheaders=ssn)
result = client.service.addPerson(person)
```

Do **not** try to pass the header as an XML 'string' such as:

```
#!python
client = client(url)
ssn = '<ssn:SessionID>123</ssn:SessionID>'
client.set_options(soapheaders=ssn)
result = client.service.addPerson(person)
```

It will not work because:

1. Only [suds.sax.element.Element-class.html Elements] are processed as 'custom' headers. 1. The XML string would be escaped as &lt;ssn:SessionID&gt;123&lt;/ssn:SessionID&gt; anyway.

CHAPTER 10

Ws-Security

ws-security provided by `UsernameToken` with 'clear-text' password (no digest).:

```
#!python
from suds.wsse import *
security = Security()
token = UsernameToken('myusername', 'mypassword')
security.tokens.append(token)
client.set_options(wsse=security)
```

or, if the 'Nonce' and 'Create' elements are needed, they can be generated and set as follows:

```
#!python
from suds.wsse import *
security = Security()
token = UsernameToken('myusername', 'mypassword')
token.setnonce()
token.setcreated()
security.tokens.append(token)
client.set_options(wsse=security)
```

but, if you want to manually set the 'Nonce' and/or 'Created', you may do as follows:

```
#!python
from suds.wsse import *
security = Security()
token = UsernameToken('myusername', 'mypassword')
token.setnonce('MyNonceString...')
token.setcreated(datetime.now())
security.tokens.append(token)
client.set_options(wsse=security)
```

CHAPTER 11

Multi-Document 'Docuemnt/Literal'

In most cases, services defined using the document/literal SOAP binding style will define a single document as the message payload. The <message/> will only have (1) `<part/>` which references an `<element/>` in the schema. In this case, suds presents a RPC view of that method by displaying the method signature as the contents (nodes) of the document. Eg:

```
#!xml
<schema>
...
<xs:element name="Foo" type = "tns:Foo"/>
<xs:complexType name="Foo">
<xs:sequence>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="age" type="xs:int"/>
</xs:sequence>
</xs:complexType>
...
</schema>

<definitions>
...
<message name="FooMessage">
<part name="parameters" element="Foo">
</message>
...
</definitions>
```

Suds will report the method 'foo' signature as:

```
foo(xs:string name, xs:int age,)
```

This provides an RPC feel to the document/literal soap binding style.

Now, if the wsdl defines:

```
#!xml
<schema>
...
<xs:element name="Foo" type = "tns:Foo"/>
<xs:element name="Bar" type = "xs:string"/>
<xs:complexType name="Foo">
<xs:sequence>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="age" type="xs:int"/>
</xs:sequence>
</xs:complexType>
...
</schema>

<definitions>
...
<message name="FooMessage">
<part name="foo" element="Foo">
<part name="bar" element="Bar">
</message>
...
</definitions>
```

Suds will be forced to report the method 'foo' signature as:

```
foo(Foo foo, xs:int bar)
```

The message has (2) parts which defines that the message payload contains (2) documents. In this case, suds must present a `/Document/` view of the method.

CHAPTER 12

HTTP Authentication

12.1 Basic

HTTP authentication as defined by [http://www.ietf.org/rfc/rfc2617.txt RFC-2617] can be done as follows:

```
#!python
client = Client(url, username='elmer', password='fudd')
```

Authentication is provided by the (default) [suds.transport.https.HttpAuthenticated-class.html HttpAuthenticated] 'Transport' class defined in the [suds.transport.http-module.html transport.https] module that follows the challenge (http 401) / response model defined in the RFC.

'Transport' [suds.transport.http-module.html transport.http] module that provides http authentication for servers that don't follow the challenge/response model. Rather, it sets the 'Authentication:' http header on **all** http requests. This transport can be used as follows:

```
#!python
from suds.transport.http import HttpAuthenticated
t = HttpAuthenticated(username='elmer', password='fudd')
client = Client(url, transport=t)
```

OR:

```
#!python
from suds.transport.http import HttpAuthenticated
t = HttpAuthenticated()
client = Client(url, transport=t, username='elmer', password='fudd')
```

12.2 Windows (NTLM)

Suds includes a [suds.transport.https.WindowsHttpAuthenticated-class.html NTLM transport] based on urllib2. This implementation requires 'users' to install the [http://code.google.com/p/python-ntlm/ python-ntlm]. It is **not** packaged with ‘suds’.

To use this, simply do something like:

```python
#!python
from suds.transport.https import WindowsHttpAuthenticated
ntlm = WindowsHttpAuthenticated(username='xx', password='xx')
client = Client(url, transport=ntlm)
```

CHAPTER 13

Proxies

The suds default [suds.transport.HttpTransport-class.html transport] handles proxies using `urllib2.Request.set_proxy()`. The proxy options can be passed set using `Client.set_options`. The proxy options must contain a dictionary where `keys=protocols` and values are the hostname (or IP) and port of the proxy.:

```
#!python
...
d = dict(http='host:80', https='host:443', ...)
client.set_options(proxy=d)
...
```

CHAPTER 14

Message Injection INJECTION '(Diagnostics/Testing)'

The service API provides for message/reply injection.

To inject either a soap message to be sent or to inject a reply or fault to be processed as if returned by the soap server, simply specify the `__inject` keyword argument with a value of a dictionary containing either:

- 'msg' = <message string>
- 'reply' = <reply string>
- 'fault' = <fault string>

when invoking the service. Eg:

Sending a 'raw' soap message:

```
#!python
message = \
"""<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Body>
      ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>"""

print client.service.test(__inject={'msg':message})
```

Injecting a response for testing:

```
#!python
reply = \
"""<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Body>
      ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>"""
```

(continues on next page)

(continued from previous page)

```
print client.service.test(__inject={'reply':reply})
```

CHAPTER 15

Performance

Suds provides some URL caching. By default, http get(s) such as getting the WSDL and importing XSDs are cached. The caching applies to URL such as those used to get the referenced WSDLs and XSD schemas but does **not** apply to service method invocation as this would not make sense.

The default 'cache' is a [suds.cache.ObjectCache-class.html ObjectCache] with an expiration of (1) day.

This duration may be adjusted as follows:

```
#!python
cache = client.options.cache
cache.setduration(days=10)
```

OR:

```
#!python
cache.setduration(seconds=90)
```

The 'duration' my be (months, weeks, days, hours, seconds).

The default 'location' (directory) is '/tmp/suds' so 'Windows' users will need to set the 'location' to something that makes sense on windows.

The cache is an [suds.options.Options-class.html option] and can be set with any kind of [suds.cache.Cache-class.html Cache] object or may be disabled by setting the option to 'None'. So, uses may 'plug-in' any kind of cache they want.:

```
#!python
from suds.cache import Cache
class MyCache(Cache)
...
client.set_options(cache=MyCache())
```

To disable caching:

```
#!python
client.set_options(cache=None)
```

CHAPTER 16

Fixing broken schema(s)

There are many cases where the schema(s) defined both within the WSDL or imported are broken. The most common problem is failure to import the follow proper import rules. That is, references are made in one schema to named objects defined in another schema without importing it. The [suds.xsd.doctor-module.html doctor] module defines a set of classes for 'mending' broken schema(s).

16.1 Doctors

The [suds.xsd.doctor.Doctor-class.html Doctor] class provides the interface for classes that provide this service. Once defined, the 'doctor' can be specified using the schema 'doctor' as an [suds.options.Options-class.html option] when creating the Client. Or, you can use one of the stock 'doctors'

- [suds.xsd.doctor.ImportDoctor-class.html ImportDoctor] - Used to fix 'import' problems.

For example:

```
#!python
imp = Import('http://schemas.xmlsoap.org/soap/encoding/')
imp.filter.add('http://some/namespace/A')
imp.filter.add('http://some/namespace/B')
doctor = ImportDoctor(imp)
client = Client(url, doctor=doctor)
```

In this example, we've specified that the 'doctor' should examine schema(s) with a 'targetNamespace' of `http://some/namespace/A or http://some/namespace/B` and ensure that the schema for the `http://schemas.xmlsoap.org/soap/encoding/` is imported. If those schema(s) do not have an <xs:import/> for those namespaces, it is added.

For cases where the 'schemaLocation' is not bound to the 'namespace', the [suds.xsd.doctor.Import-class.html Import] can be created specifying the 'location' has follows:

```
#!python
imp = Import('http://www.w3.org/2001/XMLSchema', location='http://www.w3.org/2001/
↪XMLSchema.xsd')
```

(continues on next page)

(continued from previous page)

```
imp.filter.add('http://some/namespace/A')
imp.filter.add('http://some/namespace/B')
doctor = ImportDoctor(imp)
client = Client(url, doctor=doctor)
```

A commonly referenced schema (that is not imported) is the SOAP section 5 encoding schema. This can now be fixed as follows:

```
#!python
imp = Import('http://schemas.xmlsoap.org/soap/encoding/')
imp.filter.add('http://some/namespace/A')
doctor = ImportDoctor(imp)
client = Client(url, doctor=doctor)
```

16.2 Binding Schema Locations (URL) to Namespaces

Some WSDL(s) schemas import as: `<import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>` without `schemaLocation=""` and expect processor to use the namespace URI as the schema location for the namespace. The specifications for processing `<import/>` leave the resolution of the imported namespace to a schema to the decision of the processor (in this case suds) when `@schemaLocation` is not specified. Suds always looks within the WSDL for a schema but does not look outside unless:

- A schemaLocation is specified, or
- A static binding is specified using the following syntax:

  ```
  #!python
  from suds.xsd.sxbasic import Import
  ns = 'http://schemas.xmlsoap.org/soap/encoding/'
  location = 'http://schemas.xmlsoap.org/soap/encoding/'
  Import.bind(ns, location)
  ```

Or, the shorthand (when location is the same as the namespace URI):

```
#!python
Import.bind(ns)
```

Note: `http://schemas.xmlsoap.org/soap/encoding/'` automatically 'bound'

CHAPTER 17

Plugins

It is intended to be a general, more extensible, mechanism for users to inspect/modify suds while it is running. Today, there are two 'one-off' ways to do this:

1. bindings.Binding.replyfilter - The reply text can be inspected & modified.
2. xsd.Doctor - The doctor 'option' used to mend broken schemas.

The [toc-suds.plugin-module.html plugin] module provides a number of classes but users really only need to be concerned with a few:

- The [suds.plugin.Plugin-class.html Plugin] class which defines the interface for user plugins
- The 'Context' classes which are passed to the plugin.

The plugins are divided into (4) classes based on the 'tasks' of the soap client:

Initialization:

> The client initialization task which is when the client has digested the WSDL and associated XSD.

Document Loading:

> The document loading task. This is when the client is loading WSDL & XSD documents.

Messaging:

> The messaging task is when the client is doing soap messaging as part of method (operation) invocation.

17.1 !InitPlugin

The '!InitPlugin' currently has (1) hook:

initialized():

> Called after the client is initialized. The context contains the 'WSDL' object.

17.2 !DocumentPlugin

The '!DocumentPlugin' currently has (2) hooks:

loaded():

Called before parsing a 'WSDL' or 'XSD' document. The context contains the url & document text.

parsed():

Called after parsing a 'WSDL' or 'XSD' document. The context contains the url & document 'root'.

17.3 !MessagePlugin

The '!MessagePlugin' currently has (5) hooks:

marshalled():

Provides the plugin with the opportunity to inspect/modify the envelope 'Document' *before* it is sent.

sending():

Provides the plugin with the opportunity to inspect/modify the message 'text' *before* it is sent.

received():

Provides the plugin with the opportunity to inspect/modify the received XML 'text' *before* it is SAX parsed.

parsed():

Provides the plugin with the opportunity to inspect/modify the sax parsed DOM tree for the reply *before* it is unmarshalled.

unmarshalled():

Provides the plugin with the opportunity to inspect/modify the unmarshalled reply *before* it is returned to the caller.

General usage:

```
#!python
from suds.plugin import *

class MyPlugin(DocumentPlugin):
    ...

plugin = MyPlugin()
client = Client(url, plugins=[plugin])
```

Plugins need to override **only** those methods (hooks) of interest - not all of them. Exceptions are caught and logged.

Here is an example. Say I want to add some attributes to the document root element in the soap envelope. Currently suds does not provide a way to do this using the main API. Using a plugin much like the schema doctor, we can do this.

Say our envelope is being generated by suds as:

```
<soapenv:Envelope>
<soapenv:Body>
 <ns0:foo>
   <name>Elmer Fudd</name>
   <age>55</age>
 </ns0:foo>
</soapenv:Body>
</soapenv:Envelope>
```

But what you need is:

```
<soapenv:Envelope>
<soapenv:Body>
 <ns0:foo id="1234" version="2.0">
   <name>Elmer Fudd</name>
   <age>55</age>
 </ns0:foo>
</soapenv:Body>
</soapenv:Envelope>

#!python
from suds.plugin import MessagePlugin

class MyPlugin(MessagePlugin):
  def marshalled(self, context):
     body = context.envelope.getChild('Body')
     foo = body[0]
     foo.set('id', '12345')
     foo.set('version', '2.0')

client = Client(url, plugins=[MyPlugin()])
```

In the future, the `Binding.replyfilter` and 'doctor' **option** will likely be deprecated. The [suds.xsd.doctor.ImportDoctor-class.html ImportDoctor] has been extended to implement the `[suds.plugin.Plugin-class.html Plugin].onLoad()` API.

In doing this, we can treat the !ImportDoctor as a plugin:

```
#!python

imp = Import('http://www.w3.org/2001/XMLSchema')
imp.filter.add('http://webservices.serviceU.com/')

d = ImportDoctor(imp)
client = Client(url, plugins=[d])
```

We can also replace our `Binding.replyfilter()` with a plugin as follows:

```
#!python
def myfilter(reply):
    return reply[1:]

Binding.replyfilter = myfilter

# replace with:

class Filter(MessagePlugin):
```

(continues on next page)

(continued from previous page)

```
    def received(self, context):
        reply = context.reply
        context.reply = reply[1:]

client = Client(url, plugins=[Filter()])
```

CHAPTER 18

Technical (FYI) Notes

- XML namespaces are represented as a tuple (prefix, URI). The default namespace is `(None,None)`.
- **The suds.sax module was written becuase elementtree and other python XML packages either: have a DOM API which is**
  (in the case of elementtree) do not deal with namespaces and especially prefixes sufficiently.
- **A qualified reference is a type that is referenced in the WSDL such as `<tag type="tns:Person/>`**
  where the qualified reference is a tuple `('Person', ('tns','http://myservce/namespace'))` where the namespace is the 2nd part of the tuple. When a prefix is not supplied as in `<tag type="Person/>`, the namespace is the targetNamespace for the defining fragment. This ensures that all lookup and comparisons are fully qualified.

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index