Last modified on 10/11/11 23:48:18

OVERVIEW
LOGGING
BASIC USAGE
Simple Arguments
Complex Arguments
Complex Arguments Using Python (dict)
FAULTS
OPTIONS
ENUMERATIONS
FACTORY
SERVICES WITH MULTIPLE PORTS
WSDL WITH MULTIPLE SERVICES & MULTIPLE PORTS
SOAP HEADERS
CUSTOM SOAP HEADERS
WS-SECURITY
MULTI-DOCUMENT *Docuemnt/Literal?*
HTTP AUTHENTICATION
Basic
Windows (NTLM)
PROXIES
MESSAGE INJECTION *(Diagnostics/Testing?)*
PERFORMANCE
FIXING BROKEN SCHEMA(s)
Doctors
Binding Schema Locations (URL) to Namespaces
PLUGINS
InitPlugin
DocumentPlugin
MessagePlugin
TECHNICAL (FYI) NOTES

OVERVIEW

The *Suds* web services client is a lightweight soap-based client for python the is licensed under LGPL. 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.

The primary interface into the library is the Client object. It provides methods for configuring the library and (2) sub-namespaces defined below. When the Client is created, it processes the wsdl and referenced schema(s). From this information, it derives a representation of this information which is used to provide the user with a *service description* and for message/reply processing.

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:

```
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):

```
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.xsd.schema
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.

BASIC USAGE

Version: API[3]

The *suds* Client class provides a consolidated API for consuming web services. The object contains (2) sub-namespaces:

service
The service namespace provides a proxy for the consumed service. This object is used to invoke operations (methods) provided by the service endpoint.

factory
The 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:

```
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:

```
print client
```

```
Suds - version: 0.3.3 build: (beta) R397-20081121

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, )
         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 ouput lists that the service named `WebServiceTestBeanService` has methods such as getPercentBodyFat() and addPerson().

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:

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

You have 21% body fat.

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

You have 21% body fat.

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

```
You have 21% body fat.
```

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:

```
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:

```
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:

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

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

```
person.name = name
person.age = 35
person.phone = [phone]
```

or

```
person.phone.append(phone)
```

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

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

It's that easy.

The ability to use python *dict* to represent complex objects was **re-introduced in 0.3.8**. However, this is not the preferred method because it may lead to passing incomplete objects. Also, this approach has a significant limitation. 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*.

Complex Arguments Using Python (dict)

Note: version 0.3.8+

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*.

```
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.

```
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:

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

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

```
person['name'] = name
person['age'] = 35
person['phone'] = [phone,]
```

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

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

FAULTS

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

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

```
( 200, person ...)
```

OPTIONS

The *suds* client has many that may be used to control the behavior of the library. Some are general options and others are transport options. Although, the options objects are exposed, the preferred and supported way to set/unset options is through:

- The Client constructor
- The Client.set_options()
- The 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 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. *version 0.4+*

  - 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 RequestContext is returned Default: False.

ENUMERATIONS

Enumerations are handled as follows:

Let's say the wsdl defines the following enumeration:

```
<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:

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

FACTORY

The 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 becuase 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 quanified by it's parent's name using the dot notation as follows:

```
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 qualifed by either a namespace prefix such as:

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

Or, the name can be fully qualified by the namespace itself using the full qualification syntax as (as of 0.2.6):

```
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.

SERVICES WITH MULTIPLE PORTS

Some services are defined with multiple ports as:

```
<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:

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

```
Suds - version: 0.3.3 build: (beta) R397-20081121

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
      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 port as:

```
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* option before invoking the method as:

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

- fully qualify the method as:

  ```
  client.service.soap.getBank()
  ```

After r551 version 0.3.7, this changes some to 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:

```
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 subscripted either by name (string) or index(int).

There are many ways to do this:

- Select a default port using the *port* option before invoking the method as:

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

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

```
client.service['soap'].getBank()
```

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

```
client.service[0].getBank()
```

WSDL WITH MULTIPLE SERVICES & MULTIPLE PORTS

version: 0.3.7+

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

```
<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/OtherBLZServic
</wsdl:service>
```

And are reported by suds as:

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

```
Suds - version: 0.3.7 build: (beta) R550-20090820

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
      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:

```
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 option before invoking the method as:

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

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

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

- method qualified by *service* and *port* using indexes as:

  ```
  client.service[1][0].getBank()
  ```

- method qualified by *service* (by name) only as:

```
client.service['OtherBLZService'].getBank()
```

- method qualified by *service* (by index) only as:

```
client.service[1].getBank()
```

Note, that if a WSDL defines more then one service, you must qualify the *service* via option or by using the subscripting syntax in order to specify the *port* using the subscript syntax.

SOAP HEADERS

SOAP headers may be passed during the service invocation by using the *soapheaders* option as follows:

```
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

```
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

```
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:

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

Passing *soapheaders* by keyword (dict) is available in 0.3.4 (r442) and later.

CUSTOM SOAP HEADERS

Custom SOAP headers may be passed during the service invocation by using the *soapheaders* 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 Element or collection of Elements as follows:

```
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:

```
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 Elements are processed as *custom* headers.
2. The XML string would be escaped as <ssn:SessionID>123</ssn:SessionID> anyway.

***Notes:**

1. Passing single Elements as soap headers fixed in Ticket ~~#232~~ (r533) and will be released on 0.3.7.
2. Reusing this Element in subsequent calls fixed in Ticket ~~#233~~ (r533) and will be released on 0.3.7.

WS-SECURITY

As of r452 / 0.3.4 (beta) to provide basic ws-security with `UsernameToken` with *clear-text* password (no digest).

```
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:

```
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:

```
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)
```

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:

```
<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:

```
<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.

HTTP AUTHENTICATION

Basic

As of version 0.3.3 and *newer*, *basic* HTTP authentication as defined by RFC-2617 can be done as follows:

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

Authentication is provided by the (default) HttpAuthenticated *Transport* class defined in the transport.https module that follows the challenge (http 401) / response model defined in the RFC.

As of r537, *0.3.7* beta, a new *Transport* was added in the 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:

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

Or

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

For version: **0.3.3 and older ONLY**:

Revision 63+ (and release 0.1.8+) includes the migration from httplib to urllib2 in the suds default transport which enables users to leverage all of the authentication features provided by urllib2. For example basic HTTP authentication could be implemented as follows:

```
myurl = 'http://localhost:7080/webservices/WebServiceTestBean?wsdl'
client = Client(myurl)

import urllib2
baseurl = 'http://localhost:7080/'
username = 'myuser'
password = 'mypassword'
passman = urllib2.HTTPPasswordMgrWithDefaultRealm()
passman.add_password(None, baseurl, username, password)
authhandler = urllib2.HTTPBasicAuthHandler(passman)

client.options.transport.urlopener  = urllib2.build_opener(authhandler)
```

The suds default HTTP transport uses urllib2.urlopen(), basic http authentication is handled automatically if you create the transport's urlopener correctly and set the urlopener.

Windows (NTLM)

As of 0.3.8, suds includes a NTLM transport based on urllib2. This implementation requires *users* to install the python-ntlm. It is not packaged with *suds*.

To use this, simply do something like:

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

PROXIES

The suds default 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.

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

```
...
```

MESSAGE 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:

```
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:

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

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

PERFORMANCE

As of 0.3.5 r473, 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.

In 0.3.9, FileCache was replaced with ObjectCache.

The default *cache* is a ObjectCache with an expiration of (1) day.

This duration may be adjusted as follows:

```
cache = client.options.cache
cache.setduration(days=10)
```

OR

```
 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 option and can be set with any kind of Cache object or may be disabled by setting the option to *None*. So, uses may *plug-in* any kind of cache they want.

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

To disable caching:

```
client.set_options(cache=None)
```

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 doctor module defines a set of classes for *mending* broken schema(s).

Doctors

The Doctor class provides the interface for classes that provide this service. Once defined, the *doctor* can be specified using the schema *doctor* as an option when creating the Client. Or, you can use one of the stock *doctors*

- ImportDoctor - Used to fix *import* problems.

For example:

```
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 Import can be created specifying the *location* has follows:

```
imp = Import('http://www.w3.org/2001/XMLSchema', location='http://www.w3.org/2001/XMLSchem
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:

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

note: Available in r512+ and 0.3.6 *beta*.

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 descession 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:

of (r420).

mechanism for
ways to do this:

be concerned

DL and

D documents.

f method

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*.

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:

```
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>
```

```
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 ImportDoctor has been extended to implement the Plugin.onLoad() API.

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

```
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:

```
def myfilter(reply):
  return reply[1:]

Binding.replyfilter = myfilter

# replace with:

class Filter(MessagePlugin):
    def received(self, context):
        reply = context.reply
        context.reply = reply[1:]

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

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 very unfriendly or: (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 lookups and comparisons are fully qualified.