Mastering Oracle+Python, Part 7: Service-Oriented Python Architecture
by Przemyslaw Piotrowski

Getting Python into the SOA business as a consumer as well as a provider of different kinds of Web Services

Published December 2011

See series TOC

Service-oriented architecture (SOA) plays a key role in today's business strategies. Mixing and matching enterprise components has become the standard requirement for all mission-critical enterprise applications, ensuring smooth service orchestration at various layers of corporate architectures. Python could be a tool of choice for quickly bridging these services together, using one of the freely available open source libraries.

With Python, handling Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) is as sleek as working with built-in language intrinsics. Over the last couple of years a number of modules for working with Web Services emerged, making Python a head-to-head player in the SOA space where it rivals large programming stacks like Java EE or .NET.

In this installment we will look into handling SOAP and REST communication using a new library for consuming SOAP called *suds*, deploy a simple WSGI-compliant Web Service using just the standard library, and see how Oracle Application Express can swimmingly talk to Python.

Consuming SOAP

Over time numerous Python modules were written for handling SOAP communication, including ZSI, SOAPpy and soaplib. In this tutorial, however, we are going to make use of a new, lightweight library called suds. It has been targeted at efficient consumption of Web Services and provides a comfortable way of working with WSDL endpoints.

It all starts with the instantiation of a Client object that issues a Web Service call underneath. The response is then handled by suds and results in either a plain XML format or parsed Python object representation. The type of the return object depends on the service's definition and is converted by suds automatically.

Here we will access two different Web Services: one that will return a plain XML response, and the second a parsed representation. Let's see how Oracle Corporation (NQ: ORCL) is doing right now.

Listing 1. orcl_quote.py: accessing Oracle's stock information from WSDL endpoint.

```
import suds
from xml.etree import ElementTree
quote_ws = suds.client.Client("http://www.webservicex.net/stockquote.asmx?WSDL")
orcl_quote_resp = quote_ws.service.GetQuote("ORCL")
orcl_quote_xml = ElementTree.fromstring(orcl_quote_resp)
orcl_last = orcl_quote_xml.findtext("Stock/Last")
orcl_datetime = orcl_quote_xml.findtext("Stock/Date") + "@" + orcl_quote_xml.findtext("Stock/Time")
print "ORCL stock value = %s at %s" % (orcl_last, orcl_datetime
```

If you are behind a firewall, change the first line of code to something like this, with the appropriate proxy name and port substituted:

```
quote_ws = suds.client.Client("http://www.webservicex.net/stockquote.asmx?WSDL",
     proxy = dict(http = "http://myproxy:myproxyport"))
```

As a result we get output similar to this:

```
[oracle@xe ~]$ python orcl_quote.py
ORCL stock value = 32.96 at 11/15/2011@4:00pm
```

If we looked closer at the response object, we would discover that behind the scenes suds made use of Python's SAX XML library to provide representation of the received XML object.

```
>>> type(orcl_quote_resp)
<class 'suds.sax.text.Text'>
>>> print orcl_quote_resp
<StockQuotes><Stock><Symbol>ORCL</Symbol><Last>32.96</Last><Date>11/15/2011</Date><Time>4:00pm</Time>
<Change>0.00</Change><Open>N/A</Open><High>N/A</High><Low>N/A</Low><Volume>0</Volume><MktCap>166.3B</MktCap>
<PreviousClose>32.96</PreviousClose><PercentageChange>0.00%</PercentageChange><AnnRange>24.72 -
36.50</AnnRange><Earns>1.758</Earns><P-E>18.75</P-E><Name>Oracle Corporatio</Name></Stock></StockQuotes>
```

This is where xml.ElementTree comes to help, providing easy way to extract required information with findtext() method, in form of orcl_quote_xml.findtext("Stock/Last") call.

The situation changes slightly when we get proper SOAP response. This gets transformed into a Python object transparently by suds.

Listing 2. redwood.py checking current weather conditions at Oracle HQ

```
import suds
weather_ws = suds.client.Client("http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL")
redwood = weather_ws.service.GetCityWeatherByZIP("94065")
print redwood
```

Consequently, this script returns a Python object with formatted string representation as below:

```
[oracle@xe ~]$ python redwood.py
(WeatherReturn){
   Success = True
   ResponseText = "City Found"
   State = "CA"
```

```
   City = "Redwood City"
   WeatherStationCity = "Hayward"
   WeatherID = 11
   Description = "Clear"
   Temperature = "N/A"
   RelativeHumidity = "N/A"
   Wind = "NE5"
   Pressure = "30.17S"
   Visibility = None
   WindChill = None
   Remarks = None
 }
```

Scrutinizing the object further, we would have found out that what's under the redwood variable is now a Python object instance, with attributes accessible as illustrated on the string representation above. This means we can now reference redwood.Pressure or redwood.Wind and receive something close to 30.17S or NE5, respectively.

All available services and their ports, together with a text representation of their bindings and methods, are exposed through the client.wsdl.service attribute:

```
>>> weather_ws.wsdl.services
[(Service){
   name = "Weather"
   qname = "(Weather, http://ws.cdyne.com/WeatherWS/)"
   ports[] =
       (Port){
       name = "WeatherSoap"
       qname = "(WeatherSoap, http://ws.cdyne.com/WeatherWS/)"
       _Port__service = (Service)...
       binding =
              (Binding){
              name = "WeatherSoap"
              qname = "(WeatherSoap, http://ws.cdyne.com/WeatherWS/)"
              operations =
              {
                     GetCityForecastByZIP =
                     (Operation){
                            name = "GetCityForecastByZIP"
                            soap =
...
```

The suds module itself is capable of handling complex types through suds.factory namespace and can handle multiple WSDL ports, custom headers within SOAP request, HTTP authentication, and a subset of WS-security.

Time to REST

Another Web Service standard that's now omnipresent is Representational State Transfer (REST). With much less overhead than WSDL and SOAP, REST works the same way a person interacts with Internet resources, accessing Uniform Resource Locators (URL) and Uniform Resource Identifiers (URI) to get desired resource content.

To use REST you no external libraries or dedicated frameworks are required; modules within the Python Language Library are sufficient for all your RESTful needs. To quickly check what's happening within Oracle Corporation, we can simply peek at its Twitter feed. With Python this is really trivial to implement. See Listing 3 for a basic code behind recent status check.

Listing 3. twitter.py: looking at Twitter status feed of a given user

```
import json
import urllib2
import sys

twitter_api = "https://api.twitter.com/1/statuses/user_timeline.json?screen_name=%s&count=5"
endpoint = urllib2.urlopen(twitter_api % sys.argv[1])
tweets = json.loads(endpoint.read())

for tweet in tweets:
  print "%s\n%s\n" % (tweet["created_at"], tweet["text"])
```

If you are behind a firewall, insert a line similar to the following at the start of the script, with the appropriate proxy name and port substituted:

```
urllib2.install_opener(urllib2.build_opener(urllib2.ProxyHandler({'https':'myproxy:myproxyport'})))
```

Running the twitter.py will output the five most recent tweets with their posting date:

```
[oracle@xe ~]$ python twitter.py Oracle
Tue Nov 15 20:30:09 +0000 2011
Have a need for speed? Find out why #Oracle #Exalogic Elastic #Cloud puts your enterprise on the fast track:
http://t.co/AE5btu15

Tue Nov 15 16:00:09 +0000 2011
Webcast today at 1pm ET: Storage Strategies for Accelerating Database Test & Development: Register @
http://t.co/7WKGVXB4

Mon Nov 14 21:45:12 +0000 2011
Register now for 11/17 Webcast: Privileged User Access Control with #Oracle Database 11g. Details @
http://t.co/QCcwxJvk

Mon Nov 14 20:00:42 +0000 2011
Are your customers loyal? Would you like them to be? Join us for our upcoming Webcast 11/17. Register @
http://t.co/48ZNUSCS

Mon Nov 14 16:00:56 +0000 2011
```

```
Storage Strategies for Accelerating Database Test & Development: Increase the speed, efficiency of test
environment http://t.co/AtxrD8aT
```

Now, as we experienced consuming Web Services, let's move to the trickier part where we provide a RESTful API on top of the HR (Human Resources) schema of Oracle Database 11*g* Express Edition.

The example in Listing 4 will make use of json, urlparse and wsgiref modules, in addition to the one we are already well familiar with: cx_Oracle. Refraining from using any of the available Web frameworks, our aim is to implement simple JSON-based REST service on top of Python's standard library, using the widely adopted Web Server Gateway Interface (WSGI). This service responds to requests with full employee information returned as a JSON representation of application/json content type. The make_server procedure within the wsgiref.simple_server module handles HTTP requests on port 8001 until KeyboardInterrupt exception is raised (invoked by pressing CTRL+C).

Listing 4. restserver.py: Employee information JSON Web Service

```
import json
import cx_Oracle
import urlparse
from wsgiref.simple_server import make_server

def hr_web_service(environ, start_response):
  start_response('200 OK', [('Content-Type', 'application/json')])
  resp = ""
  url = urlparse.urlparse(environ['PATH_INFO'])
  parameters = urlparse.parse_qs(environ['QUERY_STRING'])
  if url.path=="/emp":
    empid = parameters['id']
    with cx_Oracle.connect('hr/hr') as db:
      cursor = db.cursor()
      cursor.execute("select * from employees e where e.employee_id=:empid", empid)
      rows = []
      for row in cursor:
        rowdict = {}
        for pos, col in enumerate(cursor.description):
          rowdict[col[0]] = str(row[pos])
        rows += [rowdict]
    return json.dumps(rows)
  return "Invalid request."

if __name__=="__main__":
  ws = make_server('', 8001, hr_web_service)
  ws.serve_forever()
```

Running restserver.py and directing your browser at http://localhost:8001/emp?id=110 will transform a Python dictionary object about employee 110 and output the JSON-encoded representation:

```
[{"PHONE_NUMBER": "515.123.4567", "SALARY": "24000.0", "FIRST_NAME": "Steven", "LAST_NAME": "King", "JOB_ID":
"AD_PRES", "HIRE_DATE": "2003-06-17 00:00:00", "COMMISSION_PCT": "None", "EMPLOYEE_ID": "100", "MANAGER_ID":
"None", "EMAIL": "SKING", "DEPARTMENT_ID": "90"}]
```

RESTful APIs are much more lightweight than their SOAP counterparts and as of today, represent the major trend in consuming and providing Web Services in corporate and Internet environments. REST relies on HTTP itself, shifting the focus from payload type to just endpoints. SOAP and WSDL, even if extremely powerful, bring a lot of overhead and are trickier to work with. Whatever you choose, Python allows for comfortable work in a "pythonic" manner - coherent, efficient, and with little code.

Answering Oracle Application Express Calls

Oracle Application Express 4.0 supports both RESTful and SOAP Web Services. With this ability in mind, we can leverage another Oracle-Python pattern based on the Application Express-WSGI bridge.

Let's make use of restserver.py from Listing 4 to provide a service called from Application Express. If you plan to run the Python server in a remote location you need to enable remote APEX calls to that machine, Listing 5 shows PL/SQL code that needs to be run as SYSDBA to enable calls to any machine (you might need to consult your network administrator about handling external network calls).

Listing 5. Granting connect privileges to all hosts for user APEX_040000

```
DECLARE
  ACL_PATH  VARCHAR2(4000);
  ACL_ID    RAW(16);
BEGIN

  SELECT ACL INTO ACL_PATH FROM DBA_NETWORK_ACLS
   WHERE HOST = '*' AND LOWER_PORT IS NULL AND UPPER_PORT IS NULL;

  SELECT SYS_OP_R2O(extractValue(P.RES, '/Resource/XMLRef')) INTO ACL_ID
    FROM XDB.XDB$ACL A, PATH_VIEW P
   WHERE extractValue(P.RES, '/Resource/XMLRef') = REF(A) AND
         EQUALS_PATH(P.RES, ACL_PATH) = 1;

  DBMS_XDBZ.ValidateACL(ACL_ID);
   IF DBMS_NETWORK_ACL_ADMIN.CHECK_PRIVILEGE(ACL_PATH, 'APEX_040000',
     'connect') IS NULL THEN
      DBMS_NETWORK_ACL_ADMIN.ADD_PRIVILEGE(ACL_PATH, 'APEX_040000', TRUE, 'connect');
  END IF;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
  DBMS_NETWORK_ACL_ADMIN.CREATE_ACL('python_apis.xml',
    'ACL - Mastering Oracle-Python, Part 7','APEX_040000', TRUE, 'connect');
  DBMS_NETWORK_ACL_ADMIN.ASSIGN_ACL('python_apis.xml','*');
END;
/

COMMIT;
```

(For details refer to this Oracle documentation.)

Now let's log into the APEX workspace and from the Application Builder choose **Create > Application Type: Database > From Scratch > Name: PYTHONWS (schema: HR) > Add Page (Blank) >** then select **Create** twice. We now have an application called PYTHONWS with a single blank

page named “Page 1”.

Before we can actually use the Web Service we need to first define a reference to it. For this purpose we need to point browser to **Application Builder > Shared Components > Web Service References > Create > REST > Name: EMPLOYEE_WEBSRV, URL: http://localhost:81/emp** (or a server when you’ll be running the Python service) **> REST Input Parameters: Add Parameter “id” of type “string” > REST Output Parameters: Output format: Text, Name: json, Path: 1 > Create**. At this point we should have a valid reference that we can easily validate through the “Test REST Web Reference” page, which is accessible from the Test column icon when the Web Service References are shown in list view in APEX.

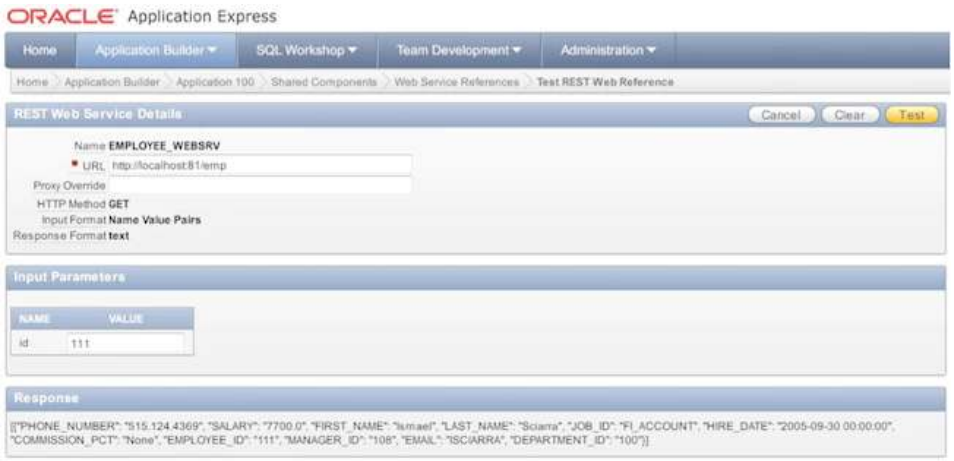

Figure 1 Validating restserver.py through APEX REST Web Reference

With a properly defined service we can now add this functionality to the page. On Page 1 create new region choosing **Form > Form on Web Service > Web Service Reference: EMPLOYEE_WEBSRV > Page Number: 1 > P1_ID > P1_JSON > Create**. This basic steps are sufficient to present a page with input field P1_ID returning response from Web Service into P1_JSON field through the Web Service Request process.

Conclusion

This article covered a number of important items related to consuming SOAP and REST Web Services using both open source modules and Python’s standard library itself. We saw how easy it is to plug into a Twitter stream and enable communication between Oracle Application Express and Python Web Services. Once again, Python’s dynamic nature and rapid development speed have proven it to be a reliable tool in SOA enablement. .

Przemyslaw Piotrowski is an information technology specialist working with emerging technologies and dynamic, agile development environments. Having a strong IT background that includes administration, development and design, he finds many paths of software interoperability.