Soapbox BSD Documentation

Release 0.0.1

AuthorName

Jan 12, 2022

CONTENTS

Contents:

CHAPTER

ONE

TUTORIAL

This tutorial assumes some understanding of XML, XSD, WSDL and SOAP.

1.1 Introduction

The main purpose of this library is neat implementation of SOAP protocol, but xsd package can used for any XML as it gives means of mapping XML to object. The object description generally is similar to Django database models - the static fields that define instance fields. The main difference would be that type is passed as first parameter, rather then being a field e.g

```
# in Django
Django: tail_number = models.CharField()

# in soapbox
soapbox: tail_number = xsd.Element(xsd.String)
```

xsd.Element reflects the nature of the field, elements are fields that will be wrapped with tag, other options are xsd.Attribute, xsd.Ref and xsd.ListElement. For more detail see xsd.Element pydoc.As SOAP, WSDL and XSD files are also XMLs the xsd package was also used to describe them. The descriptions are located in xsdspec.py, soap.py and wsdl.py. soap package also provides dispatcher and client Stub.

Other elements included in this tool are translators, that can generate python code from formal description or formal description from code. Related files: py2xsd.py, xsd2py.py, wsdl2py.py, py2wsdl.py.

utils.py is mostly jinja2 helper functions. jinja2 is templating engine used for code generation.

1.2 1. Working with XML

The main building blocks are xsd.ComplexType, xsd.Element, xsd.Attribute and simple types defined in xsd package. xsd.ComplexType is a parent to extend to define own type. Main methods for types are xml - translates object into XML, and parsexml builds object from XML.

```
#Example 1. Rendering object to XML.
from soapbox import xsd

class Airport(xsd.ComplexType):
    type = xsd.Element(xsd.String)
    code = xsd.Element(xsd.String)

```

(continues on next page)

(continued from previous page)

```
airport = Airport()
airport.type = "IATA"
airport.code = "WAW"
print airport.xml("takeoff_airport")
```

Note that xml method takes one parameter - root tag name.

```
#Example 2. Parsing XML to object.
from soapbox import xsd
class Airport(xsd.ComplexType):
     type = xsd.Element(xsd.String)
     code = xsd.Element(xsd.String)

XML = """<takeoff_airport>
  <type>IATA</type>
  <code>WAW</code>
</takeoff_airport>"""

airpport = Airport.parsexml(XML)
print "Type:", airport.type#prints Type: IATA
print "Code:", airport.code#Code: WAW
```

```
#Example 3. Nested ComplexTypes with attributes.
from datetime import datetime
from soapbox import xsd
class Airport(xsd.ComplexType):
     type = xsd.Element(xsd.String)
     code = xsd.Element(xsd.String)

class Flight(xsd.ComplexType):
    tail_number = xsd.Attribute(xsd.String)
    type = xsd.Attribute(xsd.Integer, use=xsd.Use.OPTIONAL)
    takeoff_airport = xsd.Element(Airport)
    takeoff_datetime = xsd.Element(xsd.DateTime, minOccurs=0)
    landing_airport = xsd.Element(Airport)
    landing_datetime = xsd.Element(xsd.DateTime, minOccurs=0)

flight = Flight(tail_number="LN-KKA")#Constructor handles field inititailization.
flight.takeoff_airport = Airport(type="IATA", code="WAW")
flight.landing_airport = Airport(type="ICAO", code="EGLL")

print flight.xml("flight")
#datetime field types will accept, datetime object or string,
#that parses correctly to such object.
flight.takeoff_datetime = datetime.now()
print flight.xml("flight")
```

will produce

```
<flight tail_number="LN-KKA">
  <takeoff_airport>
    <type>IATA</type>
```

(continues on next page)

(continued from previous page)

```
    <code>WAW</code>
  </takeoff_airport>
  <takeoff_datetime>2011-05-06T11:11:23</takeoff_datetime>
  <landing_airport>
    <type>ICAO</type>
    <code>EGLL</code>
  </landing_airport>
</flight>
```

1.3 2. Schema

xsd.Schema is an object that aggregates all informations stored in XSD file. There two main use cases for this object. It can be used to generate XSD file or it can be generated from such file. For detail field description see: xsd.Schema pydoc. Schema instance is required for validation and because SOAP webservice performs validation is required for service configuration too: See documentation Defining webservice.

1.3.1 2.1 Generating code from XSD file

py2xsd.py generates Python representation of XML from XSD file. Example: {{{xsd2py.py examplesops.xsd}}}

will generate:

```
from soapbox import xsd

class Pilot(xsd.String):
    enumeration = [ "CAPTAIN",  "FIRST_OFFICER", ]

class Airport(xsd.ComplexType):
    INHERITANCE = None
    INDICATOR = xsd.Sequence
    code_type = xsd.Element(xsd.String( enumeration =
    [ "ICAO", "IATA", "FAA",]) )
    code = xsd.Element(xsd.String)


class Weight(xsd.ComplexType):
    INHERITANCE = None
    INDICATOR = xsd.Sequence
    value = xsd.Element(xsd.Integer)
    unit = xsd.Element(xsd.String( enumeration =
    [ "kg", "lb",]) )


class Ops(xsd.ComplexType):
    INHERITANCE = None
    INDICATOR = xsd.Sequence
    aircraft = xsd.Element(xsd.String)
    flight_number = xsd.Element(xsd.String)
    type = xsd.Element(xsd.String( enumeration =
    [ "COMMERCIAL", "INCOMPLETE", "ENGINE_RUN_UP", "TEST", "TRAINING", "FERRY",
```

(continues on next page)

(continued from previous page)

```
"POSITIONING", "LINE_TRAINING",]) )
    takeoff_airport = xsd.Element(Airport)
    takeoff_gate_datetime = xsd.Element(xsd.DateTime, minOccurs=0)
    takeoff_datetime = xsd.Element(xsd.DateTime)
    takeoff_fuel = xsd.Element(Weight, minOccurs=0)
    takeoff_gross_weight = xsd.Element(Weight, minOccurs=0)
    takeoff_pilot = xsd.Element(Pilot, minOccurs=0)
    landing_airport = xsd.Element(Airport)
    landing_gate_datetime = xsd.Element(xsd.DateTime, minOccurs=0)
    landing_datetime = xsd.Element(xsd.DateTime)
    landing_fuel = xsd.Element(Weight, minOccurs=0)
    landing_pilot = xsd.Element(Pilot, minOccurs=0)
    destination_airport = xsd.Element(Airport, minOccurs=0)
    captain_code = xsd.Element(xsd.String, minOccurs=0)
    first_officer_code = xsd.Element(xsd.String, minOccurs=0)
    V2 = xsd.Element(xsd.Integer, minOccurs=0)
    Vref = xsd.Element(xsd.Integer, minOccurs=0)
    Vapp = xsd.Element(xsd.Integer, minOccurs=0)


class Status(xsd.ComplexType):
    INHERITANCE = None
    INDICATOR = xsd.Sequence
    action = xsd.Element(xsd.String( enumeration =
    [ "INSERTED", "UPDATED", "EXISTS",]) )
    id = xsd.Element(xsd.Long)

Schema = xsd.Schema(
    targetNamespace = "http://flightdataservices.com/ops.xsd",
    elementFormDefault = "unqualified",
    simpleTypes = [ Pilot,],
    attributeGroups = [],
    groups = [],
    complexTypes = [ Airport, Weight, Ops, Status,],
    elements = {  "status":xsd.Element(Status), "ops":xsd.Element(Ops),})
```

Let redirect output to the python file. {{{xsd2py.py examplesops.xsd > tmpops.py}}}. Now calling {{{py2xsd.py tmpops.py}}} will generate equivalent XSD from Python code. xsd2py script expects schema instance to be defined in global scope called "Schema", in way similar to one in generated code.

1.4 3. Web service

When WSDL file is provided server or client code can be generated using wsdl2py script. If not, advised would be to write code first a then use browser to request specification. Accessing URL <your webservice context>?wsdl with browser will give current WSDL with XSD embaded.

1.4.1 3.1 Generating code from WSDL file

wsdl2py can generate either client or server code. For server use -s, client -c flag. Server example: {{{wsdl2py.py -s Specificationsops.wsdl}}}

```
# ...XSD part truncated...
PutOps_method = xsd.Method(function = PutOps,
    soapAction = "http://polaris.flightdataservices.com/ws/ops/PutOps",
    input = "ops",#Pointer to Schema.elements
    output = "status",#Pointer to Schema.elements
    operationName = "PutOps")

SERVICE = soap.Service(
    targetNamespace = "http://flightdataservices.com/ops.wsdl",
    location = "http://polaris.flightdataservices.com/ws/ops",
    schema = Schema,
    methods = [PutOps_method, ])
```

Generated code includes: methods descriptions, service description, dispatcher and Django ulrs.py binding.

xsd.Method describes one method for service(that can consist from more then one method). Methods give dispatcher informations required for method distinction - soapAction and operationName, and function to call on incoming SOAP message. For detail field meaning see xsd.Method pydoc.

SERVICE aggregates all informations required for WSDL generation and correct dispatching. get_django_dispatch returns a function binded to SERVICE that pointed from urls.py will call appropriate function on incoming SOAP message. The called function, in this example PutOps, is expected to return object from XSD that could be translated to correct and valid response - for this example this would be Status instance.

URLs binding it is commented out, paste this code into your urls.py and change <fill the module path> to point file where to code was generated.

1.4.2 3.2 Client

Client can be generated with flag -c: {{{wsdl2py.py -c examplesops.wsdl}}}

Generated code:

```
# ...XSD Part truncated ...
PutOps_method = xsd.Method(
   soapAction = "http://polaris.flightdataservices.com/ws/ops/PutOps",
   input = "ops",#Pointer to Schema.elements
   output = "status",#Pointer to Schema.elements
   operationName = "PutOps")

SERVICE = soap.Service(
    targetNamespace = "http://flightdataservices.com/ops.wsdl",
```

(continues on next page)

(continued from previous page)

```
    location = "http://polaris.flightdataservices.com/ws/ops",
    schema = Schema,
    methods = [PutOps_method, ])


class ServiceStub(soap.Stub):
    SERVICE = SERVICE

    def PutOps(self, ops):
        return self.call("PutOps", ops)
```

ServiceStub is a proxy object that defines methods available on remote webservice. Calling one of those method, in the example there is only one PutOps, will produce SOAP call to remote server defined in SERVICE. The methods will return appropriate object from XSD description or raise an exception on any problems.

For more real example: See examplesclient.py

1.4.3 3.3. Building Webservice

The build a webservice we need to define few things:

- Classes that would be send via SOAP
- Schema instance that aggregates all classes with name space etc.,
- Web service functions and all related informations
- Service instance to put everything together
- Binding to URL

Lets build the stock web service that will give a stock price for provided company code and datetime.

3.3.1 Stack classes

```
class GetStockPrice(xsd.ComplexType):
    company = xsd.Element(xsd.String, minOccurs=1)
    datetime = xsd.Element(xsd.DateTime)

class StockPrice(xsd.ComplexType):
    price = xsd.Element(xsd.Integer)

Schema = xsd.Schema(
      #Should be unique URL, can be any string.
      targetNamespace = "http://code.google.com/p/soapbox/stock.xsd",
      #Register all complex types to schema.
      complexTypes = [GetStockPrice, StockPrice],
      elements = {"getStockPrice":xsd.Element(GetStockPrice),
                  "stockPrice":xsd.Element(StockPrice)}
)
```

Note the elements in schema, for this version it is required to create an element of specific type and use it string element name as input/output in Service definitions. WSDL specifications allows also direct use of type, which is not covered yet.

3.3.2 Metod definition

```
def get_stock_price(request, gsp):
    print gsp.company
    return StockPrice(price=139)

get_stock_price_method = xsd.Method(
    function = get_stock_price,
    soapAction = "http://code.google.com/p/soapbox/stock/get_stock_price",
    input = "getStockPrice",
    output = "stockPrice",
    operationName = "GetStockPrice")
```

3.3.3 Puting all together

```
SERVICE = soap.Service(
    #WSDL targetNamespce
    targetNamespace = "http://code.google.com/p/soapbox/stock.wsdl",
    #The url were request should be send.
    location = "http://127.0.0.1:8000/stock",
    schema = Schema,
    methods = [get_stock_price_method])
```

and wsgi.py

```
from wsgiref.simple_server import make_server
from soapbox import soap_dispatch
from service_gen import SERVICE

dispatcher = soap_dispatch.SOAPDispatcher(SERVICE)
app = soap_dispatch.WsgiSoapApplication({'/ChargePoint/services/chargePointService':␣
↪dispatcher})

httpd = make_server('', 8000, app)
print("Serving HTTP on port 8000...")
httpd.serve_forever()
```

Now requesting http://127.0.0.1:8000/stock?wsdl will give service specification and SOAP messages like:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:stoc=
↪"http://code.google.com/p/soapbox/stock.xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <stoc:getStockPrice>
         <company>Google</company>
         <datetime>2010-08-20T21:39:59</datetime>
      </stoc:getStockPrice>
   </soapenv:Body>
</soapenv:Envelope>
```

can be sent to http://127.0.0.1:8000/stock.

The full working example can be found in examples/stock.

CHAPTER

TWO

MIDDLEWARES

2.1 Middlewares Overview

The soapbox librairie implements a version of the Rack protocol. As a result, a soapbox dispatcher can have middlewares that may inspect, analyze, or modify the application environment, request, and response before and/or after the method call.

2.1.1 Middlewares Architecture

Think of a soapbox dispatcher as an onion. Each layer of the onion is a middleware. When you invoke the dispatcher dispatch() method, the outer-most middleware layer is invoked first. When ready, that middleware layer is responsible for optionally invoking the next middleware layer that it surrounds. This process steps deeper into the onion - through each middleware layer - until the service method is invoked. This stepped process is possible because each middleware layer are callable. When you add new middleware to the dispatcher, the added middleware will become a new outer layer and surround the previous outer middleware layer (if available) or the service method call itself.

2.1.2 Dispatcher Reference

The purpose of a middleware is to inspect, analyze, or modify the application environment, request, and response before and/or after the service methood is invoked. It is easy for each middleware to obtain references to the primary dispatcher, its environment, its request, and its response:

```
def my_middleware(request, next_call):
    # the dispatcher
    request.dispatcher

    # the wsgi environment
    request.environ

    # the service method to be invoked
    request.method

    # the raw http content
    request.http_content

    # the parsed soap body
    request.soap_body

```

(continues on next page)

(continued from previous page)

```
    # the parsed soap header
    request.soap_header
```

Changes made to the environment, request, and response objects will propagate immediately throughout the application and its other middleware layers.

2.1.3 Next Middleware Reference

Each middleware layer also has a reference to the next inner middleware layer to call with next_call. It is each middleware's responsibility to optionally call the next middleware. Doing so will allow the request to complete its full lifecycle. If a middleware layer chooses not to call the next inner middleware layer, further inner middleware and the service method itself will not be run.

```
def my_middleware(request, next_call):
    # Optionally call the next middleware
    return next_call(request)
```

2.2 How to Use Middleware

On the dispatcher instanciation, use the *middlewares* parameter to give a list of middleware, the first middleware in the list will be called first, it is the outer onion. This is also possible to add middlewares by modifying the list *dispatcher.middlewares*.

2.2.1 Example Middleware

This example middleware will log the client ip address.

```
logger = logging.getLogger(__name__)
def get_client_address(request, next_call):
    # retrieve ip address
    try:
        ip = request.environ['HTTP_X_FORWARDED_FOR'].split(',')[-1].strip()
    except KeyError:
        ip = environ['REMOTE_ADDR']
    # log it
    logger.info("Received request from %s" % str(ip))
    # call next middleware
    return next_call(request)
```

2.2.2 Add Middleware

```
dispatcher = SOAPDispatcher(service, middlewares=[
    get_client_address,
]

# or after instanciation

# add an outer middleware
dispatcher.middleware.insert(0, get_client_address)
# add an inside middleware
dispatcher.middlewate.append(get_client_address)
```

When the example dispatcher above is invoked, the client ip address will be logged.

2.3 How to Write Middleware

Middleware must be a callable accepting 2 parameters *request* and *next_call* with these exact names. The callable must return a soapbox response object. I encourage you to look at soapbox built-in middleware for working examples (eg. soapbox.middlewares.ExceptionToSoapFault or soapbox.middlewares.ExceptionLogger).

This example is the most simple implementation of middleware.

```
def my_middleware(request, next_call):
    return next_call(request)

class MyMiddlewate:
    def __call__(self, request, next_call):
        return next_call(request)
```

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search