Web Services Made Easy Documentation

Release 0.6

Christophe de Vienne

November 29, 2014

Contents

CHAPTER 1

Introduction

Web Services Made Easy (WSME) simplifies the writing of REST web services by providing simple yet powerful typing, removing the need to directly manipulate the request and the response objects.

WSME can work standalone or on top of your favorite Python web (micro)framework, so you can use both your preferred way of routing your REST requests and most of the features of WSME that rely on the typing system like:

- Alternate protocols, including those supporting batch-calls
- Easy documentation through a Sphinx extension

WSME is originally a rewrite of TGWebServices with a focus on extensibility, framework-independance and better type handling.

1.1 How Easy ?

Here is a standalone wsgi example:

```
from wsme import WSRoot, expose

class MyService(WSRoot):
    @expose(unicode, unicode)  # First parameter is the return type,
                               # then the function argument types
    def hello(self, who=u'World'):
        return u"Hello {0} !".format(who)

ws = MyService(protocols=['restjson', 'restxml', 'soap'])
application = ws.wsgiapp()
```

With this published at the /ws path of your application, you can access your hello function in various protocols:

URL	Returns
http://<server>/ws/hello.json?who=you	"Hello you !"
http://<server>/ws/hello.xml	<result>Hello World !</result>
http://<server>/ws/api.wsdl	A WSDL description for any SOAP client.

1.2 Main features

- Very simple API.
- Supports user-defined simple and complex types.
- Multi-protocol : REST+Json, REST+XML, SOAP, ExtDirect and more to come.

- Extensible : easy to add more protocols or more base types.
- Framework independence : adapters are provided to easily integrate your API in any web framework, for example a wsgi container, Pecan, TurboGears, Flask, cornice...
- Very few runtime dependencies: webob, simplegeneric. Optionnaly lxml and simplejson if you need better performances.
- Integration in Sphinx for making clean documentation with wsmeext.sphinxext.

1.3 Install

pip install WSME

or, if you do not have pip on your system or virtualenv

easy_install WSME

1.4 Changes

- Read the Changelog

1.5 Getting Help

- Read the WSME Documentation.
- Questions about WSME should go to the python-wsme mailinglist.

1.6 Contribute

Report issues WSME issue tracker

Source code git clone https://github.com/stackforge/wsme/

Gerrit https://review.openstack.org/#/q/project:stackforge/wsme,n,z/

Contents

2.1 Getting Started

For now here is just a working example. You can find it in the examples directory of the source distribution.

```
# coding=utf8
"""
A mini-demo of what wsme can do.

To run it::

    python setup.py develop

Then::

    python demo.py
"""

from wsme import WSRoot, expose, validate
from wsme.types import File

import bottle

from six import u

import logging


class Person(object):
    id = int
    firstname = unicode
    lastname = unicode

    hobbies = [unicode]

    def __repr__(self):
        return "Person(%s, %s %s, %s)" % (
            self.id,
            self.firstname, self.lastname,
            self.hobbies
        )


class DemoRoot(WSRoot):
    @expose(int)
    @validate(int, int)
```

```
    def multiply(self, a, b):
        return a * b

    @expose(File)
    @validate(File)
    def echofile(self, afile):
        return afile

    @expose(unicode)
    def helloworld(self):
        return u"Здраво, свете (<- Hello World in Serbian !)"

    @expose(Person)
    def getperson(self):
        p = Person()
        p.id = 12
        p.firstname = u'Ross'
        p.lastname = u'Geler'
        p.hobbies = []
        print p
        return p

    @expose([Person])
    def listpersons(self):
        p = Person()
        p.id = 12
        p.firstname = u('Ross')
        p.lastname = u('Geler')
        r = [p]
        p = Person()
        p.id = 13
        p.firstname = u('Rachel')
        p.lastname = u('Green')
        r.append(p)
        print r
        return r

    @expose(Person)
    @validate(Person)
    def setperson(self, person):
        return person

    @expose([Person])
    @validate([Person])
    def setpersons(self, persons):
        print persons
        return persons


root = DemoRoot(webpath='/ws')

root.addprotocol('soap',
        tns='http://example.com/demo',
        typenamespace='http://example.com/demo/types',
        baseURL='http://127.0.0.1:8080/ws/',
)

root.addprotocol('restjson')

bottle.mount('/ws/', root.wsgiapp())

logging.basicConfig(level=logging.DEBUG)
bottle.run()
```

When running this example, the following soap client can interrogate the web services:

```
from suds.client import Client

url = 'http://127.0.0.1:8080/ws/api.wsdl'

client = Client(url, cache=None)

print client

print client.service.multiply(4, 5)
print client.service.helloworld()
print client.service.getperson()
p = client.service.listpersons()
print repr(p)
p = client.service.setpersons(p)
print repr(p)

p = client.factory.create('ns0:Person')
p.id = 4
print p

a = client.factory.create('ns0:Person_Array')
print a

a = client.service.setpersons(a)
print repr(a)

a.item.append(p)
print repr(a)

a = client.service.setpersons(a)
print repr(a)
```

2.2 API

2.2.1 Public API

wsme – Essentials

class wsme.signature(\[return_type\[, arg0_type\[, arg1_type, ...\]\]\], body=None, status=None)

Decorator that specify the argument types of an exposed function.

Parameters

- return_type – Type of the value returned by the function
- argN – Type of the Nth argument
- body – If the function takes a final argument that is supposed to be the request body by itself, its type.
- status – HTTP return status code of the function.
- ignore_extra_args – Allow extra/unknow arguments (default to False)

Most of the time this decorator is not supposed to be used directly, unless you are not using WSME on top of another framework.

If an adapter is used, it will provide either a specialised version of this decororator, either a new decorator named @wsexpose that takes the same parameters (it will in addition expose the function, hence its name).

class wsme.types.Base(**kw)
Base type for complex types

class wsme.wsattr(datatype, mandatory=False, name=None, default=Unset, readonly=False)
Complex type attribute definition.

Example:

```
class MyComplexType(wsme.types.Base):
    optionalvalue = int
    mandatoryvalue = wsattr(int, mandatory=True)
    named_value = wsattr(int, name='named.value')
```

After inspection, the non-wsattr attributes will be replaced, and the above class will be equivalent to:

```
class MyComplexType(wsme.types.Base):
    optionalvalue = wsattr(int)
    mandatoryvalue = wsattr(int, mandatory=True)
```

class wsme.wsproperty(datatype, fget, fset=None, mandatory=False, doc=None, name=None)
A specialised property to define typed-property on complex types. Example:

```
class MyComplexType(wsme.types.Base):
    def get_aint(self):
        return self._aint

    def set_aint(self, value):
        assert avalue < 10  # Dummy input validation
        self._aint = value

    aint = wsproperty(int, get_aint, set_aint, mandatory=True)
```

wsme.Unset
Default value of the complex type attributes.

class wsme.WSRoot(protocols=[], webpath='', transaction=None, scan_api=<function scan_api at 0x7f2a68f42230>)
Root controller for webservices.

Parameters

- protocols – A list of protocols to enable (see addprotocol())
- webpath – The web path where the webservice is published.
- transaction (A transaction-like object or True.) – If specified, a transaction will be created and handled on a per-call base.

  This option can be enabled along with repoze.tm2 (it will only make it void).

  If True, the default transaction module will be imported and used.

wsgiapp()
Returns a wsgi application

addprotocol(protocol, **options)
Enable a new protocol on the controller.

Parameters protocol – A registered protocol name or an instance of a protocol.

getapi()
Returns the api description.

Return type list of (path, FunctionDefinition)

2.2.2 Internals

wsme.types – Types

wsme.api – API related api

class wsme.api.FunctionArgument(name, datatype, mandatory, default)
An argument definition of an api entry

name = None
argument name

datatype = None
Data type

mandatory = None
True if the argument is mandatory

default = None
Default value if argument is omitted

class wsme.api.FunctionDefinition(func)
An api entry definition

name = None
Function name

doc = None
Function documentation

return_type = None
Return type

arguments = None
The function arguments (list of FunctionArgument)

body_type = None
If the body carry the datas of a single argument, its type

status_code = None
Status code

ignore_extra_args = None
True if extra arguments should be ignored, NOT inserted in the kwargs of the function and not raise UnknownArgument exceptions

pass_request = None
name of the function argument to pass the host request object. Should be set by using the wsme.types.HostRequest type in the function @:function:‘signature‘

extra_options = None
Dictionnary of protocol-specific options.

static get(func)
Returns the FunctionDefinition of a method.

get_arg(name)
Returns a FunctionArgument from its name

wsme.rest.args – REST protocol argument handling

2.3 Types

Three kinds of data types can be used as input or output by WSME.

2.3.1 Native types

The native types are a fixed set of standard Python types that different protocols map to their own basic types.

The native types are :

- type bytes
  A pure-ascii string (wsme.types.bytes which is str in Python 2 and bytes in Python 3).
- type text
  A unicode string (wsme.types.text which is unicode in Python 2 and str in Python 3).
- type int
  An integer (int)
- type float
  A float (float)
- type bool
  A boolean (bool)
- type Decimal
  A fixed-width decimal (decimal.Decimal)
- type date
  A date (datetime.date)
- type datetime
  A date and time (datetime.datetime)
- type time
  A time (datetime.time)
- Arrays – This is a special case. When stating a list datatype, always state its content type as the unique element of a list. Example:

```
class SomeWebService(object):
    @expose([str])
    def getlist(self):
        return ['a', 'b', 'c']
```

- Dictionaries – Statically typed mappings are allowed. When exposing a dictionary datatype, you can specify the key and value types, with a restriction on the key value that must be a 'pod' type. Example:

```
class SomeType(object):
    amap = {str: SomeOthertype}
```

There are other types that are supported out of the box. See the Pre-defined user types.

2.3.2 User types

User types allow you to define new, almost-native types.

The idea is that you may have Python data that should be transported as base types by the different protocols, but needs conversion to/from these base types, or needs to validate data integrity.

To define a user type, you just have to inherit from wsme.types.UserType and instantiate your new class. This instance will be your new type and can be used as @wsme.expose or @wsme.validate parameters.

Note that protocols can choose to specifically handle a user type or a base class of user types. This is case with the two pre-defined user types, wsme.types.Enum and wsme.types.binary.

Pre-defined user types

WSME provides some pre-defined user types:

- binary – for transporting binary data as base64 strings.
- Enum – enforce that the values belongs to a pre-defined list of values.

These types are good examples of how to define user types. Have a look at their source code!

Here is a little example that combines binary and Enum:

```
ImageKind = Enum(str, 'jpeg', 'gif')

class Image(object):
    name = unicode
    kind = ImageKind
    data = binary
```

wsme.types.binary
: The wsme.types.BinaryType instance to use when you need to transfer base64 encoded data.

class wsme.types.BinaryType
: A user type that use base64 strings to carry binary data.

class wsme.types.Enum(basetype, *values, **kw)
: A simple enumeration type. Can be based on any non-complex type.

  Parameters

  - basetype – The actual data type
  - values – A set of possible values

  If nullable, 'None' should be added the values set.

  Example:

  ```
  Gender = Enum(str, 'male', 'female')
  Specie = Enum(str, 'cat', 'dog')
  ```

2.3.3 Complex types

Complex types are structured types. They are defined as simple Python classes and will be mapped to adequate structured types in the various protocols.

A base class for structured types is provided, wsme.types.Base, but is not mandatory. The only thing it adds is a default constructor.

The attributes that are set at the class level will be used by WSME to discover the structure. These attributes can be:

- A datatype – Any native, user or complex type.
- A wsattr – This allows you to add more information about the attribute, for example if it is mandatory.
- A wsproperty – A special typed property. Works like standard property with additional properties like wsattr.

Attributes having a leading '_' in their name will be ignored, as well as the attributes that are not in the above list. This means the type can have methods, they will not get in the way.

Example

```
Gender = wsme.types.Enum(str, 'male', 'female')
Title = wsme.types.Enum(str, 'M', 'Mrs')

class Person(wsme.types.Base):
    lastname = wsme.types.wsattr(unicode, mandatory=True)
    firstname = wsme.types.wsattr(unicode, mandatory=True)

    age = int
    gender = Gender
    title = Title

    hobbies = [unicode]
```

Rules

A few things you should know about complex types:

- The class must have a default constructor – Since instances of the type will be created by the protocols when used as input types, they must be instantiable without any argument.
- Complex types are registered automatically (and thus inspected) as soon a they are used in expose or validate, even if they are nested in another complex type.

  If for some reason you need to control when type is inspected, you can use wsme.types.register_type().
- The datatype attributes will be replaced.

  When using the 'short' way of defining attributes, ie setting a simple data type, they will be replaced by a wsattr instance.

  So, when you write:

  ```
  class Person(object):
      name = unicode
  ```

  After type registration the class will actually be equivalent to:

  ```
  class Person(object):
      name = wsattr(unicode)
  ```

  You can still access the datatype by accessing the attribute on the class, along with the other wsattr properties:

  ```
  class Person(object):
      name = unicode

  register_type(Person)

  assert Person.name.datatype is unicode
  assert Person.name.key == "name"
  assert Person.name.mandatory is False
  ```
- The default value of instance attributes is Unset.

  ```
  class Person(object):
      name = wsattr(unicode)
  ```

```
p = Person()
assert p.name is Unset
```

This allows the protocol to make a clear distinction between null values that will be transmitted, and unset values that will not be transmitted.

For input values, it allows the code to know if the values were, or were not, sent by the caller.

- When 2 complex types refer to each other, their names can be used as datatypes to avoid adding attributes afterwards:

```
class A(object):
    b = wsattr('B')

class B(object):
    a = wsattr(A)
```

Predefined Types

- type File
  A complex type that represents a file.

  In the particular case of protocol accepting form encoded data as input, File can be loaded from a form file field.

  Data samples:

  Json

  ```
  {
      "content": null,
      "contenttype": null,
      "filename": null
  }
  ```

  XML

  ```
  <value>
    <filename nil="true" />
    <contenttype nil="true" />
    <content nil="true" />
  </value>
  ```

  SOAP

  ```
  <Element 'value' at 0x7f2a66474510>
  ```

  ExtDirect

  ```
  {
      "content": null,
      "contenttype": null,
      "filename": null
  }
  ```

  content
  : Type binary
    File content

  filename
  : Type unicode
    The file name

contenttype
: Type unicode
Mime type of the content

2.4 Functions

WSME is based on the idea that most of the time the input and output of web services are actually strictly typed. It uses this idea to ease the implementation of the actual functions by handling those input/output. It also proposes alternate protocols on top of a proper REST api.

This chapter explains in detail how to 'sign' a function with WSME.

2.4.1 The decorators

Depending on the framework you are using, you will have to use either a @wsme.signature decorator or a @wsme.wsexpose decorator.

@signature

The base @wsme.signature decorator defines the return and argument types of the function, and if needed a few more options.

The Flask and Cornice adapters both propose a specific version of it, which also wrap the function so that it becomes suitable for the host framework.

In any case, the use of @wsme.signature has the same meaning: tell WSME what is the signature of the function.

@wsexpose

The native Rest implementation, and the TG and Pecan adapters add a @wsme.wsexpose decorator.

It does what @wsme.signature does, and exposes the function in the routing system of the host framework.

This decorator is generally used in an object-dispatch routing context.

Note: Since both decorators play the same role, the rest of this document will alway use @signature.

2.4.2 Signing a function

Signing a function is just a matter of decorating it with @signature:

```
@signature(int, int, int)
def multiply(a, b):
    return a * b
```

In this trivial example, we tell WSME that the 'multiply' function returns an integer, and takes two integer parameters.

WSME will match the argument types by order to determine the exact type of each named argument. This is important since most of the web service protocols don't provide strict argument ordering but only named parameters.

Optional arguments

Defining an argument as optional is done by providing a default value:

```
@signature(int, int, int):
def increment(value, delta=1):
    return value + delta
```

In this example, the caller may omit the 'delta' argument, and no 'MissingArgument' error will be raised.

Additionally, this argument will be documented as optional by the sphinx extension.

Body argument

When defining a Rest CRUD API, we generally have a URL to which we POST data.

For example:

```
@signature(Author, Author)
def update_author(data):
    # ...
    return data
```

Such a function will take at least one parameter, 'data', that is a structured type. With the default way of handling parameters, the body of the request would look like this:

```
{
    "data":
    {
        "id": 1,
        "name": "Pierre-Joseph"
    }
}
```

If you think (and you should) that it has one extra level of nesting, the 'body' argument is here for you:

```
@signature(Author, body=Author)
def update_author(data):
    # ...
    return data
```

With this syntax, we can now post a simpler body:

```
{
    "id": 1,
    "name": "Pierre-Joseph"
}
```

Note that this does not prevent the function from having multiple parameters; it just requires the body argument to be the last:

```
@signature(Author, bool, body=Author)
def update_author(force_update=False, data=None):
    # ...
    return data
```

In this case, the other arguments can be passed in the URL, in addition to the body parameter. For example, a POST on /author/SOMEID?force_update=true.

Status code

The default status codes returned by WSME are 200, 400 (if the client sends invalid inputs) and 500 (for server-side errors).

Since a proper Rest API should use different return codes (201, etc), one can use the 'status_code=' option of @signature to do so.

```
@signature(Author, body=Author, status_code=201)
def create_author(data):
    # ...
    return data
```

Of course this code will only be used if no error occurs.

In case the function needs to change the status code on a per-request basis, it can return a wsme.Response object, allowing it to override the status code:

```
@signature(Author, body=Author, status_code=202)
def update_author(data):
    # ...
    response = Response(data)
    if transaction_finished_and_successful:
        response.status_code = 200
    return response
```

Extra arguments

The default behavior of WSME is to reject requests that give extra/unknown arguments. In some (rare) cases, this is undesirable.

Adding 'ignore_extra_args=True' to @signature changes this behavior.

Note: If using this option seems to solve your problem, please think twice before using it!

Accessing the request

Most of the time direct access to the request object should not be needed, but in some cases it is.

On frameworks that propose a global access to the current request it is not an issue, but on frameworks like pyramid it is not the way to go.

To handle this use case, WSME has a special type, HostRequest:

```
from wsme.types import HostRequest

@signature(Author, HostRequest, body=Author)
def create_author(request, newauthor):
    # ...
    return newauthor
```

In this example, the request object of the host framework will be passed as the request parameter of the create_author function.

2.5 Protocols

In this document the same webservice example will be used to illustrate the different protocols. Its source code is in the last chapter (The example).

2.5.1 REST

Note: This chapter applies for all adapters, not just the native REST implementation.

The two REST protocols share common characterics.

Each function corresponds to distinct webpath that starts with the root webpath, followed by the controllers names if any, and finally the function name.

The example's exposed functions will be mapped to the following paths:

- /ws/persons/create
- /ws/persons/get
- /ws/persons/list
- /ws/persons/update
- /ws/persons/destroy

In addition to this trivial function mapping, a method option can be given to the expose decorator. In such a case, the function name can be omitted by the caller, and the dispatch will look at the HTTP method used in the request to select the correct function.

The function parameters can be transmitted in two ways (if using the HTTP method to select the function, one way or the other may be usable) :

1. As a GET query string or POST form parameters.

   Simple types are straight forward :

   /ws/person/get?id=5

   Complex types can be transmitted this way:

   /ws/person/update?p.id=1&p.name=Ross&p.hobbies[0]=Dinausaurs&p.hobbies[1]=Rachel

2. In a Json or XML encoded POST body (see below)

The result will be returned Json or XML encoded (see below).

In case of error, a 400 or 500 status code is returned, and the response body contains details about the error (see below).

2.5.2 REST+Json

name 'restjson'

Implements a REST+Json protocol.

This protocol is selected if:

- The request content-type is either 'text/javascript' or 'application/json'
- The request 'Accept' header contains 'text/javascript' or 'application/json'
- A trailing '.json' is added to the path
- A 'wsmeproto=restjson' is added in the query string

Options

nest_result Nest the encoded result in a result param of an object. For example, a result of 2 would be {'result': 2}

Types

Type	Json type
str	String
unicode	String
int	Number
float	Number
bool	Boolean
Decimal	String
date	String (YYYY-MM-DD)
time	String (hh:mm:ss)
datetime	String (YYYY-MM-DDThh:mm:ss)
Arrays	Array
None	null
Complex types	Object

Return

The Json encoded result when the response code is 200, or a Json object with error properties ('faulcode', 'faultstring' and 'debuginfo' if available) on error.

For example, the '/ws/person/get' result looks like:

```
{
    'id': 2
    'fistname': 'Monica',
    'lastname': 'Geller',
    'age': 28,
    'hobbies': [
        'Food',
        'Cleaning'
    ]
}
```

And in case of error:

```
{
    'faultcode': 'Client',
    'faultstring': 'id is missing'
}
```

2.5.3 REST+XML

name 'restxml'

This protocol is selected if

- The request content-type is 'text/xml'
- The request 'Accept' header contains 'text/xml'
- A trailing '.xml' is added to the path
- A 'wsmeproto=restxml' is added in the query string

Types

Type	XML example
str	<value>a string</value>
unicode	<value>a string</value>
int	<value>5</value>
float	<value>3.14</value>
bool	<value>true</value>
Decimal	<value>5.46</value>
date	<value>2010-04-27</value>
time	<value>12:54:18</value>
datetime	<value>2010-04-27T12:54:18</value>
Arrays	<value> <item>Dinausaurs<item> <item>Rachel<item> </value>
None	<value nil="true"/>
Complex types	<value> <id>1</id> <fistname>Ross</fistname> </value>

Return

A xml tree with a top 'result' element.

```
<result>
    <id>1</id>
    <firstname>Ross</firstname>
    <lastname>Geller</lastname>
</result>
```

Errors

A xml tree with a top 'error' element, having 'faultcode', 'faultstring' and 'debuginfo' subelements:

```
<error>
  <faultcode>Client</faultcode>
  <faultstring>id is missing</faultstring>
</error>
```

2.5.4 SOAP

name 'soap'

Implements the SOAP protocol.

A wsdl definition of the webservice is available at the 'api.wsdl' subpath. (/ws/api.wsdl in our example).

The protocol is selected if the request matches one of the following condition:

- The Content-Type is 'application/soap+xml'
- A header 'Soapaction' is present

Options

tns Type namespace

2.5.5 ExtDirect

name extdirect

Implements the Ext Direct protocol.

The provider definition is made available at the /extdirect/api.js subpath.

The router url is /extdirect/router[/subnamespace].

Options

namespace Base namespace of the api. Used for the provider definition.

params_notation Default notation for function call parameters. Can be overriden for individual functions by adding the extdirect_params_notation extra option to @expose.

The possible notations are :

- 'named' – The function will take only one object parameter in which each property will be one of the parameters.
- 'positional' – The function will take as many parameters as the function has, and their position will determine which parameter they are.

expose extra options

extdirect_params_notation Override the params_notation for a particular function.

2.5.6 The example

In this document the same webservice example will be used to illustrate the different protocols:

```
class Person(object):
    id = int
    lastname = unicode
    firstname = unicode
    age = int

    hobbies = [unicode]

    def __init__(self, id=None, lastname=None, firstname=None, age=None,
                 hobbies=None):
        if id:
            self.id = id
        if lastname:
            self.lastname = lastname
        if firstname:
            self.firstname = firstname
        if age:
            self.age = age
        if hobbies:
            self.hobbies = hobbies

persons = {
    1: Person(1, "Geller", "Ross", 30, ["Dinosaurs", "Rachel"]),
    2: Person(2, "Geller", "Monica", 28, ["Food", "Cleaning"])
}

class PersonController(object):
    @expose(Person)
    @validate(int)
    def get(self, id):
        return persons[id]

    @expose([Person])
    def list(self):
        return persons.values()

    @expose(Person)
    @validate(Person)
    def update(self, p):
        if p.id is Unset:
            raise ClientSideError("id is missing")
        persons[p.id] = p
        return p

    @expose(Person)
    @validate(Person)
    def create(self, p):
        if p.id is not Unset:
            raise ClientSideError("I don't want an id")
        p.id = max(persons.keys()) + 1
        persons[p.id] = p
        return p

    @expose()
    @validate(int)
    def destroy(self, id):
        if id not in persons:
            raise ClientSideError("Unknown ID")


class WS(WSRoot):
    person = PersonController()
```

```
root = WS(webpath='ws')
```

2.6 Integrating with a Framework

2.6.1 General considerations

Using WSME within another framework providing its own REST capabilities is generally done by using a specific decorator to declare the function signature, in addition to the framework's own way of declaring exposed functions.

This decorator can have two different names depending on the adapter.

@wsexpose This decorator will declare the function signature and take care of calling the adequate decorators of the framework.

Generally this decorator is provided for frameworks that use object-dispatch controllers, such as Pecan and Turbogears 1.x.

@signature This decorator only sets the function signature and returns a function that can be used by the host framework as a REST request target.

Generally this decorator is provided for frameworks that expect functions taking a request object as a single parameter and returning a response object. This is the case for Cornice and Flask.

If you want to enable additional protocols, you will need to mount a WSRoot instance somewhere in the application, generally /ws. This subpath will then handle the additional protocols. In a future version, a WSGI middleware will probably play this role.

Note: Not all the adapters are at the same level of maturity.

2.6.2 WSGI Application

The wsme.WSRoot.wsgiapp() function of WSRoot returns a WSGI application.

Example

The following example assumes the REST protocol will be entirely handled by WSME, which is the case if you write a WSME standalone application.

```
from wsme import WSRoot, expose


class MyRoot(WSRoot):
    @expose(unicode)
    def helloworld(self):
        return u"Hello World !"

root = MyRoot(protocols=['restjson'])
application = root.wsgiapp()
```

2.6.3 Cornice

> " Cornice provides helpers to build & document REST-ish Web Services with Pyramid, with decent default behaviors. It takes care of following the HTTP specification in an automated way where possible."

wsmeext.cornice – Cornice adapter

wsmeext.cornice.signature()
Declare the parameters of a function and returns a function suitable for cornice (ie that takes a request and returns a response).

Example

```
from cornice import Service
from wsmeext.cornice import signature
import wsme.types

hello = Service(name='hello', path='/', description="Simplest app")

class Info(wsme.types.Base):
    message = wsme.types.text


@hello.get()
@signature(Info)
def get_info():
    """Returns Hello in JSON or XML."""
    return Info(message='Hello World')


@hello.post()
@signature(None, Info)
def set_info(info):
    print("Got a message: %s" % info.message)
```

2.6.4 Flask

"Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions. And before you ask: It's BSD licensed! "

> **Warning:** Flask support is limited to function signature handling. It does not support additional protocols. This is a temporary limitation, if you have needs on that matter please tell us at python-wsme@googlegroups.com.

wsmeext.flask – Flask adapter

wsmeext.flask.signature(return_type, *arg_types, **options)
See @signature() for parameters documentation.

Can be used on a function before routing it with flask.

Example

```
from wsmeext.flask import signature

@app.route('/multiply')
@signature(int, int, int)
def multiply(a, b):
    return a * b
```

2.6.5 Pecan

> "Pecan was created to fill a void in the Python web-framework world – a very lightweight framework that provides object-dispatch style routing. Pecan does not aim to be a "full stack" framework, and therefore includes no out of the box support for things like sessions or databases. Pecan instead focuses on HTTP itself."

> **Warning:** A pecan application is not able to mount another WSGI application on a subpath. For that reason, additional protocols are not supported for now, until WSME provides a middleware that can do the same as a mounted WSRoot.

wsmeext.pecan – Pecan adapter

wsmeext.pecan.wsexpose(return_type, *arg_types, **options)

See @signature() for parameters documentation.

Can be used on any function of a pecan RestController instead of the expose decorator from Pecan.

Configuration

WSME can be configured through the application configation, by adding a 'wsme' configuration entry in config.py:

```
wsme = {
    'debug': True
}
```

Valid configuration variables are :

- 'debug': Whether or not to include exception tracebacks in the returned server-side errors.

Example

The example from the Pecan documentation becomes:

```
from wsmeext.pecan import wsexpose

class BooksController(RestController):
    @wsexpose(Book, int, int)
    def get(self, author_id, id):
        # ..

    @wsexpose(Book, int, int, body=Book)
    def put(self, author_id, id, book):
        # ..

class AuthorsController(RestController):
        books = BooksController()
```

2.6.6 Turbogears 1.x

The TG adapters have an api very similar to TGWebServices. Migrating from it should be straightforward (a little howto migrate would not hurt though, and it will be written as soon as possible).

wsmeext.tg11 – TG 1.1 adapter

wsmeext.tg11.wsexpose(return_type, *arg_types, **options)
See @signature() for parameters documentation.

Can be used on any function of a controller instead of the expose decorator from TG.

wsmeext.tg11.wsvalidate(*arg_types)
Set the argument types of an exposed function. This decorator is provided so that WSME is an almost drop-in replacement for TGWebServices. If starting from scratch you can use wsexpose() only

wsmeext.tg11.adapt(wsroot)
Returns a TG1 controller instance that publish a wsme.WSRoot. It can then be mounted on a TG1 controller.

Because the adapt function modifies the cherrypy filters of the controller the 'webpath' of the WSRoot instance must be consistent with the path it will be mounted on.

wsmeext.tg15 – TG 1.5 adapter

This adapter has the exact same api as wsmeext.tg11.

Example

In a freshly quickstarted tg1 application (let's say, wsmedemo), you can add REST-ish functions anywhere in your controller tree. Here directly on the root, in controllers.py:

```
# ...

# For tg 1.5, import from wsmeext.tg15 instead :
from wsmeext.tg11 import wsexpose, WSRoot

class Root(controllers.RootController):
    # Having a WSRoot on /ws is only required to enable additional
    # protocols. For REST-only services, it can be ignored.
    ws = adapt(
        WSRoot(webpath='/ws', protocols=['soap'])
    )

    @wsexpose(int, int, int)
    def multiply(self, a, b):
        return a * b
```

2.6.7 Other frameworks

Bottle

No adapter is provided yet but it should not be hard to write one, by taking example on the cornice adapter.

This example only show how to mount a WSRoot inside a bottle application.

```
import bottle
import wsme

class MyRoot(wsme.WSRoot):
    @wsme.expose(unicode)
    def helloworld(self):
        return u"Hello World !"
```

```
root = MyRoot(webpath='/ws', protocols=['restjson'])

bottle.mount('/ws', root.wsgiapp())
bottle.run()
```

Pyramid

The recommended way of using WSME inside Pyramid is to use Cornice.

2.7 Document your API

Web services without a proper documentation are usually useless.

To make it easy to document your own API, WSME provides a Sphinx extension.

2.7.1 Install the extension

Here we consider that you already quick-started a sphinx project.

1. In your conf.py file, add 'ext' to your extensions, and optionally set the enabled protocols.

   ```
   extensions = ['ext']

   wsme_protocols = ['restjson', 'restxml', 'extdirect']
   ```

2. Copy toggle.js and toggle.css in your _static directory.

2.7.2 The wsme domain

The extension will add a new Sphinx domain providing a few directives.

Config values

wsme_protocols
: A list of strings that are WSME protocol names. If provided by an additional package (for example WSME-Soap or WSME-ExtDirect), that package must be installed.

  The types and services generated documentation will include code samples for each of these protocols.

wsme_root
: A string that is the full name of the service root controller. It will be used to determinate the relative path of the other controllers when they are autodocumented, and calculate the complete webpath of the other controllers.

wsme_webpath
: A string that is the webpath where the wsme_root is mounted.

Directives

.. root:: <WSRoot full path>
: Define the service root controller for this documentation source file. To set it globally, see wsme_root.

  A webpath option allows override of wsme_webpath.

Example:

```
.. wsme:root:: myapp.controllers.MyWSRoot
   :webpath: /api
```

.. service:: name/space/ServiceName
Declare a service.

.. type:: MyComplexType
Equivalent to the py:class directive to document a complex type

.. attribute:: aname
Equivalent to the py:attribute directive to document a complex type attribute. It takes an additional :type: field.

Example

Source	Result
.. wsme:root:: wsmeext.sphinxext.SampleService :webpath: /api .. wsme:type:: MyType .. wsme:attribute:: test :type: int .. wsme:service:: name/space/SampleService .. wsme:function:: doit	type MyType test Type int service name/space/SampleService function getType Returns a MyType

Autodoc directives

Theses directives scan your code to generate the documentation from the docstrings and your API types and controllers.

.. autotype:: myapp.MyType
Generate the myapp.MyType documentation.

.. autoattribute:: myapp.MyType.aname
Generate the myapp.MyType.aname documentation.

.. autoservice:: myapp.MyService
Generate the myapp.MyService documentation.

.. autofunction:: myapp.MyService.myfunction
Generate the myapp.MyService.myfunction documentation.

2.7.3 Full Example

Python source

```
class SampleType(object):
    """A Sample Type"""

    #: A Int
    aint = int
```

```
    def __init__(self, aint=None):
        if aint:
            self.aint = aint

    @classmethod
    def sample(cls):
        return cls(10)


class SampleService(wsme.WSRoot):
    @wsme.expose(SampleType)
    @wsme.validate(SampleType, int, str)
    def change_aint(data, aint, dummy='useless'):
        """
        :param aint: The new value

        :return: The data object with its aint field value changed.
        """
        data.aint = aint
        return data
```

Documentation source

```
.. default-domain:: wsmeext

.. type:: int

    An integer

.. autotype:: wsmeext.sphinxext.SampleType
    :members:

.. autoservice:: wsmeext.sphinxext.SampleService
    :members:
```

Result

type SampleType

A Sample Type

Data samples:

Json

```
{
   "aint": 10
}
```

XML

```
<value>
  <aint>10</aint>
</value>
```

SOAP

```
<Element 'value' at 0x7f2a66ecac30>
```

ExtDirect

```
{
   "aint": 10
}
```

aint

Type int

A Int

service /

SampleService.change_aint(data, aint, dummy='useless')

Parameters aint – The new value

Returns The data object with its aint field value changed.

2.8 TODO

WSME is a work in progress. Here is a list of things that should be done :

- Use gevents for batch-calls
- Implement new protocols :
  - json-rpc
  - xml-rpc
- Implement adapters for other frameworks :
  - TurboGears 2
  - Pylons
  - CherryPy
  - Flask
  - others ?
- Add unittests for adapters
- Address the authentication subject (which should be handled by some other wsgi framework/middleware, but a little integration could help).

2.9 Changes

2.9.1 0.6.2 (next)

- Flask adapter complex types now supports flask.ext.restful
- Allow disabling complex types auto-register
- Documentation edits
- Various documentation build fixes
- Fix passing Dict and Array based UserType as params

2.9.2 0.6.1 (2014-05-02)

- Fix error: variable 'kw' referenced before assignment
- Fix default handling for zero values
- Fixing spelling mistakes
- A proper check of UuidType
- pecan: cleanup, use global vars and staticmethod
- args_from_args() to work with an instance of UserType

2.9.3 0.6 (2014-02-06)

- Add 'readonly' parameter to wsattr
- Fix typos in documents and comments
- Support dynamic types
- Support building wheels (PEP-427)
- Fix a typo in the types documentation
- Add IntegerType and some classes for validation
- Use assertRaises() for negative tests
- Remove the duplicated error message from Enum
- Drop description from 403 flask test case
- Fix SyntaxWarning under Python 3

2.9.4 0.5b6 (2013-10-16)

- Add improved support for HTTP response codes in cornice apps.
- Handle mandatory attributes
- Fix error code returned when None is used in an Enum
- Handle list and dict for body type in REST protocol
- Fix Sphinx for Python 3
- Add custom error code to ClientSideError
- Return a ClientSideError if unable to convert data
- Validate body when using Pecan

2.9.5 0.5b5 (2013-09-16)

More packaging fixes.

2.9.6 0.5b4 (2013-09-11)

Fixes some release-related files for the stackforge release process. No user-facing bug fixes or features over what 0.5b3 provides.

2.9.7 0.5b3 (2013-09-04)

The project moved to stackforge. Mind the new URLs for the repository, bug report etc (see the documentation).

- Allow non-default status code return with the pecan adapter (Angus Salked).
- Fix returning objects with object attributes set to None on rest-json & ExtDirect.
- Allow error details to be set on the Response object (experimental !).
- Fix: Content-Type header is not set anymore when the return type is None on the pecan adapter.
- Support unicode message in ClientSideError (Mehdi Abaakouk).
- Use pbr instead of d2to1 (Julien Danjou).
- Python 3.3 support (Julien Danjou).
- Pecan adapter: returned status can now be set on exceptions (Vitaly Kostenko).
- TG adapters: returned status can be set on exceptions (Ryan Petrello).
- six >= 1.4.0 support (Julien Danjou).
- Require ordereddict from pypi for python < 2.6 (Ryan Petrello).
- Make the code PEP8 compliant (Ryan Petrello).

2.9.8 0.5b2 (2013-04-18)

- Changed the way datas of complex types are stored. In previous versions, an attribute was added to the type for each attribute, its name being the attribute name prefixed with '_'.

  Starting with this version, a single attribute _wsme_dataholder is added to the instance.

  The motivation behind this change is to avoid adding too many attributes to the object.
- Add a special type 'HostRequest' that allow a function to ask for the host framework request object in its arguments.
- Pecan adapter: Debug mode (which returns the exception tracebacks to the client) can be enabled by the pecan application configuration.
- New adapter: wsmeext.flask, for the Flask framework.
- Fix: the cornice adapter was not usable.
- Fix: Submodules of wsmeext were missing in the packages.
- Fix: The demo app was still depending on the WSME-Soap package (which has been merged into WSME in 0.5b1).
- Fix: A function with only on 'body' parameter would fail when being called.
- Fix: Missing arguments were poorly reported by the frameworks adapters.

2.9.9 0.5b1 (2013-01-30)

- Introduce a new kind of adapters that rely on the framework routing. Adapters are provided for Pecan, TurboGears and cornice.
- Reorganised the rest protocol implementation to ease the implementation of adapters that rely only on the host framework routing system.
- The default rest @expose decorator does not wrap the decorated function anymore. If needed to expose a same function several times, a parameter multiple_expose=True has been introduced.
- Remove the wsme.release module

- Fix == operator on ArrayType
- Adapted the wsme.sphinxext module to work with the function exposed by the wsme.pecan adapter.
- Allow promotion of int to float on float attributes (Doug Hellman)
- Add a samples_slot option to the .. autotype directive to choose where the data samples whould be inserted (Doug Hellman).
- Add sample() to ArrayType and DictType (Doug Hellman).
- New syntax for object arrays as GET parameters, without brackets. Ex: ?o.f1=a&o.f1=b&o.f2=c&o.f2=d is an array of two objects: [{'f1': 'a', 'f2': 'c'}, {'f1': 'b', 'f2': 'd'}].
- @signature (and its @wsexpose frontends) has a new parameter: ignore_extra_args.
- Fix boolean as input type support in the soap implementation (Craig McDaniel).
- Fix empty/nil strings distinction in soap (Craig McDaniel).
- Improved unittests code coverage.
- Ported the soap implementation to python 3.
- Moved non-core features (adapters, sphinx extension) to the wsmeext module.
- Change the GET parameter name for passing the request body as a parameter is now from 'body' to '__body__'
- The soap, extdirect and sqlalchemy packages have been merged into the main package.
- Changed the documentation theme to "Cloud".

2.9.10 0.4 (2012-10-15)

- Automatically converts unicode strings to/from ascii bytes.
- Use d2to1 to simplify setup.py.
- Implements the SPORE specification.
- Fixed a few things in the documentation

2.9.11 0.4b1 (2012-09-14)

- Now supports Python 3.2
- String types handling is clearer.
- New wsme.types.File type.
- Supports cross-referenced types.
- Various bugfixes.
- Tests code coverage is now over 95%.
- RESTful protocol can now use the http method.
- UserTypes can now be given a name that will be used in the documentation.
- Complex types can inherit wsme.types.Base. They will have a default constructor and be registered automatically.
- Removed the wsme.wsgi.adapt function if favor of wsme.WSRoot.wsgiapp()

Extensions

wsme-soap

- Function names now starts with a lowercase letter.
- Fixed issues with arrays (issue #3).
- Fixed empty array handling.

wsme-sqlalchemy This new extension makes it easy to create webservices on top of a SQLAlchemy set of mapped classes.

wsme-extdirect

- Implements server-side DataStore (wsmeext.extdirect.datastore.DataStoreController).
- Add Store and Model javascript definition auto-generation
- Add Store server-side based on SQLAlchemy mapped classes (wsmeext.extdirect.sadatastore.SADataStoreController).

2.9.12 0.3 (2012-04-20)

- Initial Sphinx integration.

2.9.13 0.3b2 (2012-03-29)

- Fixed issues with the TG1 adapter.
- Now handle dict and UserType types as GET/POST params.
- Better handling of application/x-www-form-urlencoded encoded POSTs in rest protocols.
- wsattr now takes a 'default' parameter that will be returned instead of 'Unset' if no value has been set.

2.9.14 0.3b1 (2012-01-19)

- Per-call database transaction handling.
- Unset is now imported in the wsme module
- Attributes of complex types can now have a different name in the public api and in the implementation.
- Complex arguments can now be sent as GET/POST params in the rest protocols.
- The restjson protocol do not nest the results in an object anymore.
- Improved the documentation
- Fix array attributes validation.
- Fix date|time parsing errors.
- Fix Unset values validation.
- Fix registering of complex types inheriting form already registered complex types.
- Fix user types, str and None values encoding/decoding.

2.9.15 0.2.0 (2011-10-29)

- Added batch-calls abilities.
- Introduce a UnsetType and a Unset constant so that non-mandatory attributes can remain unset (which is different from null).
- Fix: If a complex type was only used as an input type, it was not registered.
- Add support for user types.
- Add an Enum type (which is a user type).
- The 'binary' type is now a user type.
- Complex types:
  - Fix inspection of complex types with inheritance.
  - Fix inspection of self-referencing complex types.
  - wsattr is now a python Descriptor, which makes it possible to retrieve the attribute definition on a class while manipulating values on the instance.
  - Add strong type validation on assignment (made possible by the use of Descriptors).
- ExtDirect:
  - Implements batch calls
  - Fix None values conversion
  - Fix transaction result : 'action' and 'method' were missing.

2.9.16 0.1.1 (2011-10-20)

- Changed the internal API by introducing a CallContext object. It makes it easier to implement some protocols that have a transaction or call id that has to be returned. It will also make it possible to implement batch-calls in a later version.
- More test coverage.
- Fix a problem with array attribute types not being registered.
- Fix the mandatory / default detection on function arguments.
- Fix issues with the SOAP protocol implementation which should now work properly with a suds client.
- Fix issues with the ExtDirect protocol implementation.

2.9.17 0.1.0 (2011-10-14)

- Protocol insertion order now influence the protocol selection
- Move the soap protocol implementation in a separate lib, WSME-Soap
- Introduce a new protocol ExtDirect in the WSME-ExtDirect lib.

2.9.18 0.1.0a4 (2011-10-12)

- Change the way framework adapters works. Now the adapter modules have a simple adapt function that adapt a wsme.WSRoot instance. This way a same root can be integrated in several framework.
- Protocol lookup now use entry points in the group [wsme.protocols].

2.9.19 0.1.0a3 (2011-10-11)

- Add specialised WSRoot classes for easy integration as a WSGI Application (wsme.wsgi.WSRoot) or a TurboGears 1.x controller (wsme.tg1.WSRoot).
- Improve the documentation.
- More unit tests and code-coverage.

2.9.20 0.1.0a2 (2011-10-07)

- Added support for arrays in all the protocols

2.9.21 0.1.0a1 (2011-10-04)

Initial public release.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

T

U

W