You remember //lbows//, right? Err… that PHP-based integration framework, that was– oh, forget it.

Anyways, I got tired of it. I'm currently moving towards Python as I like the leanness and clear structure of it, although I have to say, that PHP's currently got a far better SOAP server support (in the Zend framework).

Nonetheless, I'm still quite fond of lbows' way of delivering one object in various RPC-call styles (mainly XMLRPC, SOAP and JSONRPC). After finding the [[http://twistedmatrix.com|Twisted Framework]], a Internet-centric framework, that can build full featured servers out of a very small amount of code, I tried to mimic lbows' way.

It's quite simple, really. Let's take a simple python class with a public method:

```
class Backend:
def say_hello():
return "Hello"
```

You can use twisted's XMLRPC or SOAP proxies to generate a XMLRPC or SOAP server respectively. For this you simply let your class inherit from the proxy class. Let's take XMLRPC:

```
from twisted.web.xmlrpc import XMLRPC
```

class Backend(xmlrpc.XMLRPC):
def say_hello():
return "Hello"

Now you can easily let twisted create the server for you:

```
from twisted.web import resource, server
from twisted.internet import reactor
```

root = resource.Resource()
root.putChild('RPC2', Backend)
reactor.listenTCP(
8080,
server.Site(root)

)

reactor.run()

And you got a XMLRPC-server listening on port 8080. That's it. Quite impressing, right?

However, let's recall lbows and its way of taking one single class and deliver several methods.

For this to work, we need to create some kind of XMLRPC-Proxy-class, that takes our backend class and returns a XMLRPC-object suitable for twisted.

I created this thing here for XMLRPC:

```
from twisted.web.xmlrpc import XMLRPC
from twisted.web import xmlrpc
```

class XMLRPCResponseManager(xmlrpc.XMLRPC):
"""XML-RPC-Support for the backend methods"""

_backend = None
"""Reference to the real backend object"""

_procedures = None
"""Introspected functions"""

def __init__(self, backend):
"""Constructor for XMLRPCResponseManager"""

XMLRPC.__init__(self)

self._backend = backend
self._procedures = []

for method in dir(self._backend):
if not method.startswith("_"):
self._procedures.append(method)

def _getFunction(self, procedurePath):
"""Overridden for dynamic drop-in support
see :py:func:xmlrpc._getFunction

"""
try:
if procedurePath.startswith("system."):
return getattr(
self.subHandlers["system"],
procedurePath.replace(
"system.",
"xmlrpc_"
)
)
return getattr(self._backend, procedurePath)
except KeyError, e:
raise xmlrpc.NoSuchFunction(
self.NOT_FOUND,
"procedure %s not found" % procedurePath
)

def _listFunctions(self):
"""Overridden for dynamic drop-in support

see :py:func:xmlrpc._listFunctions
"""
return self._procedures

When constructing, the class takes an instance of our backend class, introspects it and thus knows what methods it can deliver via XMLRPC.

Now we just instantiate the backend, supply it to our XMLResponseManager and give that to twisted:

```
root = resource.Resource()
```

backend = Backend()

xMLRPCResponse = XMLRPCResponseManager(backend)
xmlrpc.addIntrospection(xMLRPCResponse)

root.putChild('RPC2', xMLRPCResponse)

reactor.listenTCP(
8080,
server.Site(root),
)

reactor.run()

We can do the same for SOAP. twisted's SOAP object has some other methods to be overridden though:

```
from twisted.web import soap
```

class SOAPResponseManager(soap.SOAPPublisher):
"""SOAP-Support for the backend methods"""

_backend = None
"""Reference to the real backend object"""

_procedures = None
"""Introspected functions"""

def __init__(self, backend):
"""Constructor for SOAPResponseManager"""

SOAPPublisher.__init__(self)

self._backend = backend

def lookupFunction(self, functionName):
"""Overridden for dynamic drop-in support
see :py:func:twisted.web.soap.SOAPPublisher.lookupFunction
"""
try:
return getattr(self._backend, functionName)
except KeyError, e:
return None

Now we can add this to our main method so it delivers XMLRPC and SOAP simultaneously:

```
root.putChild('SOAP', SOAPResponseManager(backend))
```

To additionally support JSON-RPC, we can use [[http://code.google.com/p/twisted-jsonrpc/|TxJsonRPC]], a twisted based implementation of JSON-RPC.

That ResponseManager is nearly identical to the XMLRPC one:

```
from txjsonrpc.web import jsonrpc
```

class JSONRPCResponseManager(jsonrpc.JSONRPC):
"""JSON-RPC-Support for the backend methods"""

_backend = None
"""Reference to the real backend object"""

_procedures = None
"""Introspected functions"""

def __init__(self, backend):
"""Constructor for JSONRPCResponseManager"""

jsonrpc.JSONRPC.__init__(self)

self._backend = backend
self._procedures = []

for method in dir(self._backend):
if not method.startswith("_"):
self._procedures.append(method)

def _getFunction(self, procedurePath):
"""Overridden for dynamic drop-in support
see :py:func:xmlrpc._getFunction
"""
try:
if procedurePath.startswith("system."):
return getattr(
self.subHandlers["system"],
procedurePath.replace(
"system.",

```
"xmlrpc_"
)
)
return getattr(self._backend, procedurePath)
except KeyError, e:
raise jsonrpc.NoSuchFunction(
self.NOT_FOUND,
"procedure %s not found" % procedurePath
)

def _listFunctions(self):
"""Overridden for dynamic drop-in support

see :py:func:xmlrpc._listFunctions
"""
return self._procedures
```

Add this to the main method:

```
root.putChild('JSONRPC', JSONRPCResponseManager(backend))
```

Now /RPC2 delivers XMLRPC, /SOAP delivers SOAP and /JSONRPC delivers JSONRPC. All of the same backend class. In around 200 lines of code.

THAT is the power of python.

The things missing in this implementation in contrast to lbows are:

* no WSDL generation (python really **lacks** some good SOAP support!)
* lbows supports multiple plugins and an advanced delivery method. This method is just for one service at a time
* in its latest versions lbows supported some kind of internal firewall that filtered out specific requests from specific sources
* no automatic documentation and no REST handler

But I guess, that could also be done in short time and without a large amount of code.