xmlrpc.server — An XML-RPC server

Purpose: Implements an XML-RPC server.

The `xmlrpc.server` module contains classes for creating cross-platform, language-independent servers using the XML-RPC protocol. Client libraries exist for many other languages besides Python, making XML-RPC an easy choice for building RPC-style services.

> **Note**
>
> All of the examples provided here include a client module as well to interact with the demonstration server. To run the examples, use two separate shell windows, one for the server and one for the client.

A Simple Server

This simple server example exposes a single function that takes the name of a directory and returns the contents. The first step is to create the `SimpleXMLRPCServer` instance and tell it where to listen for incoming requests ('localhost' port 9000 in this case). Then a function is defined to be part of the service, and registered so the server knows how to call it. The final step is to put the server into an infinite loop receiving and responding to requests.

> **Warning**
>
> This implementation has obvious security implications. Do not run it on a server on the open Internet or in any environment where security might be an issue.

```
# xmlrpc_function.py

from xmlrpc.server import SimpleXMLRPCServer
import logging
import os

# Set up logging
logging.basicConfig(level=logging.INFO)

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


# Expose a function
def list_contents(dir_name):
    logging.info('list_contents(%s)', dir_name)
    return os.listdir(dir_name)


server.register_function(list_contents)

# Start the server
try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

The server can be accessed at the URL `http://localhost:9000` using `xmlrpc.client`. This client code illustrates how to call the `list_contents()` service from Python.

```
# xmlrpc_function_client.py
```

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(proxy.list_contents('/tmp'))
```

The ServerProxy is connected to the server using its base URL, and then methods are called directly on the proxy. Each method invoked on the proxy is translated into a request to the server. The arguments are formatted using XML, and then sent to the server in a POST message. The server unpacks the XML and determines which function to call based on the method name invoked from the client. The arguments are passed to the function, and the return value is translated back to XML to be returned to the client.

Starting the server gives the following output.

```
$ python3 xmlrpc_function.py

Use Control-C to exit
```

Running the client in a second window shows the contents of the `/tmp` directory.

```
$ python3 xmlrpc_function_client.py

['com.apple.launchd.aoGXonn8nV', 'com.apple.launchd.ilryIaQugf',
'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']
```

After the request is finished, log output appears in the server window.

```
$ python3 xmlrpc_function.py

Use Control-C to exit
INFO:root:list_contents(/tmp)
127.0.0.1 - - [18/Jun/2016 19:54:54] "POST /RPC2 HTTP/1.1" 200 -
```

The first line of output is from the `logging.info()` call inside `list_contents()`. The second line is from the server logging the request because `logRequests` is `True`.

Alternate API Names

Sometimes the function names used inside a module or library are not the names that should be used in the external API. Names may change because a platform-specific implementation is loaded, the service API is built dynamically based on a configuration file, or real functions can be replaced with stubs for testing. To register a function with an alternate name, pass the name as the second argument to `register_function()`.

```
# xmlrpc_alternate_name.py

from xmlrpc.server import SimpleXMLRPCServer
import os

server = SimpleXMLRPCServer(('localhost', 9000))


def list_contents(dir_name):
    "Expose a function with an alternate name"
    return os.listdir(dir_name)


server.register_function(list_contents, 'dir')

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

The client should now use the name `dir()` instead of `list_contents()`.

```
# xmlrpc_alternate_name_client.py
```

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print('dir():', proxy.dir('/tmp'))
try:
    print('\nlist_contents():', proxy.list_contents('/tmp'))
except xmlrpc.client.Fault as err:
    print('\nERROR:', err)
```

Calling `list_contents()` results in an error, since the server no longer has a handler registered by that name.

```
$ python3 xmlrpc_alternate_name_client.py

dir(): ['com.apple.launchd.aoGXonn8nV',
'com.apple.launchd.ilryIaQugf', 'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']

ERROR: <Fault 1: '<class \'Exception\'>:method "list_contents"
is not supported'>
```

Dotted API Names

Individual functions can be registered with names that are not normally legal for Python identifiers. For example, a period (`.`) can be included in the names to separate the namespace in the service. The next example extends the "directory" service to add "create" and "remove" calls. All of the functions are registered using the prefix "`dir.`" so that the same server can provide other services using a different prefix. One other difference in this example is that some of the functions return `None`, so the server has to be told to translate the `None` values to a nil value.

```
# xmlrpc_dotted_name.py
```

```
from xmlrpc.server import SimpleXMLRPCServer
import os

server = SimpleXMLRPCServer(('localhost', 9000), allow_none=True)

server.register_function(os.listdir, 'dir.list')
server.register_function(os.mkdir, 'dir.create')
server.register_function(os.rmdir, 'dir.remove')

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

To call the service functions in the client, simply refer to them with the dotted name.

```
# xmlrpc_dotted_name_client.py

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print('BEFORE       :', 'EXAMPLE' in proxy.dir.list('/tmp'))
print('CREATE       :', proxy.dir.create('/tmp/EXAMPLE'))
print('SHOULD EXIST :', 'EXAMPLE' in proxy.dir.list('/tmp'))
print('REMOVE       :', proxy.dir.remove('/tmp/EXAMPLE'))
print('AFTER        :', 'EXAMPLE' in proxy.dir.list('/tmp'))
```

Assuming there is no `/tmp/EXAMPLE` file on the current system, the output for the sample client script is as follows.

```
$ python3 xmlrpc_dotted_name_client.py

BEFORE       : False
CREATE       : None
SHOULD EXIST : True
REMOVE       : None
```

```
AFTER        : False
```

Arbitrary API Names

Another interesting feature is the ability to register functions with names that are otherwise invalid Python object attribute names. This example service registers a function with the name “`multiply args`”.

```
# xmlrpc_arbitrary_name.py

from xmlrpc.server import SimpleXMLRPCServer

server = SimpleXMLRPCServer(('localhost', 9000))


def my_function(a, b):
    return a * b


server.register_function(my_function, 'multiply args')

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Since the registered name contains a space, dot notation cannot be used to access it directly from the proxy. Using `getattr()` does work, however.

```
# xmlrpc_arbitrary_name_client.py

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(getattr(proxy, 'multiply args')(5, 5))
```

Avoid creating services with names like this, though. This example is provided not necessarily because it is a good idea, but because existing services with arbitrary names exist, and new programs may need to be able to call them.

```
$ python3 xmlrpc_arbitrary_name_client.py

25
```

Exposing Methods of Objects

The earlier sections talked about techniques for establishing APIs using good naming conventions and namespacing. Another way to incorporate namespacing into an API is to use instances of classes and expose their methods. The first example can be recreated using an instance with a single method.

```
# xmlrpc_instance.py

from xmlrpc.server import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


class DirectoryService:
    def list(self, dir_name):
        return os.listdir(dir_name)


server.register_instance(DirectoryService())
```

```
try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

A client can call the method directly.

```
# xmlrpc_instance_client.py

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(proxy.list('/tmp'))
```

The output shows the contents of the directory.

```
$ python3 xmlrpc_instance_client.py

['com.apple.launchd.aoGXonn8nV', 'com.apple.launchd.ilryIaQugf',
'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']
```

The "`dir.`" prefix for the service has been lost, though. It can be restored by defining a class to set up a service tree that can be invoked from clients.

```
# xmlrpc_instance_dotted_names.py

from xmlrpc.server import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


class ServiceRoot:
    pass


class DirectoryService:

    def list(self, dir_name):
        return os.listdir(dir_name)


root = ServiceRoot()
root.dir = DirectoryService()

server.register_instance(root, allow_dotted_names=True)

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

By registering the instance of `ServiceRoot` with `allow_dotted_names` enabled, the server has permission to walk the tree of objects when a request comes in to find the named method using `getattr()`.

```
# xmlrpc_instance_dotted_names_client.py

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
```

```
print(proxy.dir.list('/tmp'))
```

The output of `dir.list()` is the same as with the previous implementations.

```
$ python3 xmlrpc_instance_dotted_names_client.py

['com.apple.launchd.aoGXonn8nV', 'com.apple.launchd.ilryIaQugf',
'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']
```

Dispatching Calls

By default, `register_instance()` finds all callable attributes of the instance with names not starting with an underscore ("_") and registers them with their name. To be more careful about the exposed methods, custom dispatching logic can be used.

```
# xmlrpc_instance_with_prefix.py

from xmlrpc.server import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


def expose(f):
    "Decorator to set exposed flag on a function."
    f.exposed = True
    return f


def is_exposed(f):
    "Test whether another function should be publicly exposed."
    return getattr(f, 'exposed', False)


class MyService:
    PREFIX = 'prefix'

    def _dispatch(self, method, params):
        # Remove our prefix from the method name
        if not method.startswith(self.PREFIX + '.'):
            raise Exception(
                'method "{}" is not supported'.format(method)
            )

        method_name = method.partition('.')[2]
        func = getattr(self, method_name)
        if not is_exposed(func):
            raise Exception(
                'method "{}" is not supported'.format(method)
            )

        return func(*params)

    @expose
    def public(self):
        return 'This is public'

    def private(self):
        return 'This is private'


server.register_instance(MyService())

try:
```

```
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

The `public()` method of `MyService` is marked as exposed to the XML-RPC service while `private()` is not. The `_dispatch()` method is invoked when the client tries to access a function that is part of `MyService`. It first enforces the use of a prefix ("`prefix.`" in this case, but any string can be used). Then it requires the function to have an attribute called exposed with a true value. The exposed flag is set on a function using a decorator for convenience. The following example includes a few sample client calls.

```
# xmlrpc_instance_with_prefix_client.py

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print('public():', proxy.prefix.public())
try:
    print('private():', proxy.prefix.private())
except Exception as err:
    print('\nERROR:', err)
try:
    print('public() without prefix:', proxy.public())
except Exception as err:
    print('\nERROR:', err)
```

The resulting output, with the expected error messages trapped and reported, follows.

```
$ python3 xmlrpc_instance_with_prefix_client.py

public(): This is public

ERROR: <Fault 1: '<class \'Exception\'>:method "prefix.private" is
not supported'>

ERROR: <Fault 1: '<class \'Exception\'>:method "public" is not
supported'>
```

There are several other ways to override the dispatching mechanism, including subclassing directly from `SimpleXMLRPCServer`. Refer to the docstrings in the module for more details.

Introspection API

As with many network services, it is possible to query an XML-RPC server to ask it what methods it supports and learn how to use them. `SimpleXMLRPCServer` includes a set of public methods for performing this introspection. By default they are turned off, but can be enabled with `register_introspection_functions()`. Support for `system.listMethods()` and `system.methodHelp()` can be added to a service by defining `_listMethods()` and `_methodHelp()` on the service class.

```
# xmlrpc_introspection.py

from xmlrpc.server import (SimpleXMLRPCServer,
                           list_public_methods)
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)
server.register_introspection_functions()


class DirectoryService:

    def _listMethods(self):
        return list_public_methods(self)

    def _methodHelp(self, method):
        f = getattr(self, method)
```

```
        return inspect.getdoc(f)

    def list(self, dir_name):
        """list(dir_name) => [<filenames>]

        Returns a list containing the contents of
        the named directory.

        """
        return os.listdir(dir_name)


server.register_instance(DirectoryService())

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

In this case, the convenience function `list_public_methods()` scans an instance to return the names of callable attributes that do not start with underscore (_). Redefine `_listMethods()` to apply whatever rules are desired. Similarly, for this basic example `_methodHelp()` returns the docstring of the function, but could be written to build a help string from another source.

This client queries the server and reports on all of the publicly callable methods.

```
# xmlrpc_introspection_client.py

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
for method_name in proxy.system.listMethods():
    print('=' * 60)
    print(method_name)
    print('-' * 60)
    print(proxy.system.methodHelp(method_name))
    print()
```

The system methods are included in the results.

```
$ python3 xmlrpc_introspection_client.py

============================================================
list
------------------------------------------------------------
list(dir_name) => [<filenames>]

Returns a list containing the contents of
the named directory.

============================================================
system.listMethods
------------------------------------------------------------
system.listMethods() => ['add', 'subtract', 'multiple']

Returns a list of the methods supported by the server.

============================================================
system.methodHelp
------------------------------------------------------------
system.methodHelp('add') => "Adds two integers together"

Returns a string containing documentation for the specified method.

============================================================
system.methodSignature
------------------------------------------------------------
system.methodSignature('add') => [double, int, int]

Returns a list describing the signature of the method. In the
above example, the add method takes two integers as arguments
```

```
above example, the add method takes two integers as arguments
and returns a double result.

This server does NOT support system.methodSignature.
```

See also

- Standard library documentation for xmlrpc.server
- `xmlrpc.client` – XML-RPC client.
- XML-RPC How To – Describes how to use XML-RPC to implement clients and servers in a variety of languages.

xmlrpc.client — Client Library for XML-RPC

Email

Quick Links

A Simple Server
Alternate API Names
Dotted API Names
Arbitrary API Names
Exposing Methods of Objects
Dispatching Calls
Introspection API

This page was last updated 2016-12-29.

Navigation

xmlrpc.client — Client Library for XML-RPC
Email

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example