

21.26. `xmlrpc.client` — XML-RPC client access

Source code: [Lib/xmlrpc/client.py](#)

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Warning: The `xmlrpc.client` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

Changed in version 3.5: For HTTPS URIs, `xmlrpc.client` now performs all the necessary certificate and hostname checks by default.

```
class xmlrpc.client.ServerProxy(uri, transport=None, encoding=None,
verbose=False, allow_none=False, use_datetime=False, use_builtin_types=False, *,
context=None)
```

Changed in version 3.3: The `use_builtin_types` flag was added.

A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal `HTTP Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If `allow_none` is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see <http://ontosys.com/xml-rpc/extensions.php> for a description. The `use_builtin_types` flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default. `datetime.datetime`, `bytes` and `bytearray` objects may be passed to calls. The obsolete `use_datetime` flag is similar to `use_builtin_types` but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be `ssl.SSLContext` and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
boolean	<code>bool</code>
int, i1, i2, i4, i8 or biginteger	<code>int</code> in range from -2147483648 to 2147483647. Values get the <code><int></code> tag.
double or float	<code>float</code> . Values get the <code><double></code> tag.
string	<code>str</code>
array	<code>list</code> or <code>tuple</code> containing conformable elements. Arrays are returned as <code>lists</code> .
struct	<code>dict</code> . Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <code>__dict__</code> attribute is transmitted.
<code>dateTime.iso8601</code>	<code>DateTime</code> or <code>datetime.datetime</code> . Returned type depends on values of <code>use_builtin_types</code> and <code>use_datetime</code> flags.
base64	<code>Binary</code> , <code>bytes</code> or <code>bytearray</code> . Returned type depends on the value of the <code>use_builtin_types</code> flag.
nil	The <code>None</code> constant. Passing is allowed only if <code>allow_none</code> is true.
bigdecimal	<code>decimal.Decimal</code> . Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special `Fault` instance, used to signal XML-RPC server errors, or `ProtocolError` used to signal an error in the HTTP/HTTPS transport layer. Both `Fault` and `ProtocolError` derive from a base class called `Error`. Note that the `xmlrpc` client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use `bytes` or `bytearray` classes or the `Binary` wrapper class described below.

`Server` is retained as an alias for `ServerProxy` for backwards compatibility. New code should use `ServerProxy`.

Changed in version 3.5: Added the `context` argument.

Changed in version 3.6: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See <http://ws.apache.org/xmlrpc/types.html> for a description.

See also:

XML-RPC HOWTO

A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection

Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification

The official specification.

Unofficial XML-RPC Errata

Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

21.26.1. ServerProxy Objects

A [ServerProxy](#) instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a [Fault](#) or [ProtocolError](#) object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.listMethods()`

This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.methodSignature(name)`

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply “string, array”. If it expects three integers and returns a string, its signature is “string, int, int, int”.

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.methodHelp(name)`

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Changed in version 3.5: Instances of [ServerProxy](#) support the [context manager](#) protocol for closing the underlying transport.

A working example follows. The server code:

```

from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()

```

The client code for the preceding server:

```

import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))

```

21.26.2. DateTime Objects

`class xmlrpc.client.DateTime`

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a `datetime.datetime` instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this `DateTime` item to the *out* stream object.

It also supports certain of Python's built-in operators through rich comparison and `__repr__()` methods.

A working example follows. The server code:

```

import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))

```

```
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.26.3. Binary Objects

`class xmlrpc.client.`**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a **Binary** object is provided by an attribute:

data

The binary data encapsulated by the **Binary** instance. The data is provided as a **bytes** object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 **bytes** object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```

from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()

```

The client gets the image and saves it to a file:

```

import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)

```

21.26.4. Fault Objects

`class xmlrpc.client.Fault`

A **Fault** object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a **Fault** by returning a complex type object. The server code:

```

from xmlrpc.server import SimpleXMLRPCServer

# A marshallng error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

```

```
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.26.5. ProtocolError Objects

`class xmlrpc.client.ProtocolError`

A `ProtocolError` object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url

The URI or URL that triggered the error.

errcode

The error code.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a `ProtocolError` by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC request
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
```



```
print("A protocol error occurred")
print("URL: %s" % err.url)
print("HTTP/HTTPS headers: %s" % err.headers)
print("Error code: %d" % err.errcode)
print("Error message: %s" % err.errmsg)
```

21.26.6. MultiCall Objects

The `MultiCall` object provides a way to encapsulate multiple calls to a remote server into a single request [1].

`class xmlrpc.client.MultiCall(server)`

Create an object used to boxcar method calls. `server` is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the `MultiCall` object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a `generator`; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.26.7. Convenience Functions

`xmlrpc.client.dumps(params, methodname=None, methodresponse=None, encoding=None, allow_none=False)`

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

`xmlrpc.client.loads(data, use_datetime=False, use_builtin_types=False)`

Convert an XML-RPC request or response into Python objects, a (params, methodname). *params* is a tuple of argument; *methodname* is a string, or None if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Changed in version 3.3: The *use_builtin_types* flag was added.

21.26.8. Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:
```

```

print(proxy)

try:
    print(proxy.examples.getStateName(41))
except Error as v:
    print("ERROR", v)

```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```

import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport)
print(server.examples.getStateName(41))

```

21.26.9. Example of Client and Server Usage

See [SimpleXMLRPCServer Example](#).

Footnotes

[1] This approach has been first presented in [a discussion on xmlrpc.com](#).