json-rpc Documentation

Release 1.10.8

Kirill Pavlov

Nov 11, 2017

Contents

Source code https://github.com/pavlov99/json-rpc

Issue tracker https://github.com/pavlov99/json-rpc/issues

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

CHAPTER 1

Features

- Supports JSON-RPC2.0 and JSON-RPC1.0
- Implementation is complete and 100% tested
- Does not depend on transport realisation, no external dependencies
- It comes with request manager and optional Django support
- Compatible with Python 2.6, 2.7, 3.x >= 3.2, PyPy

CHAPTER 2

Contents

2.1 Quickstart

2.1.1 Installation

Requirements **Python 2.6, 2.7**, **Python 3.x >= 3.2** or **PyPy**

To install the latest released version of package:

```
pip install json-rpc
```

2.1.2 Integration

Package is transport agnostic, integration depends on you framework. As an example we have server with Werkzeug and client with requests.

Server

```
from werkzeug.wrappers import Request, Response
from werkzeug.serving import run_simple

from jsonrpc import JSONRPCResponseManager, dispatcher


@dispatcher.add_method
def foobar(**kwargs):
    return kwargs["foo"] + kwargs["bar"]


@Request.application
def application(request):
    # Dispatcher is dictionary {<method_name>: callable}
    dispatcher["echo"] = lambda s: s
```

```
    dispatcher["add"] = lambda a, b: a + b

    response = JSONRPCResponseManager.handle(
        request.data, dispatcher)
    return Response(response.json, mimetype='application/json')


if __name__ == '__main__':
    run_simple('localhost', 4000, application)
```

Client

```
import requests
import json


def main():
    url = "http://localhost:4000/jsonrpc"
    headers = {'content-type': 'application/json'}

    # Example echo method
    payload = {
        "method": "echo",
        "params": ["echome!"],
        "jsonrpc": "2.0",
        "id": 0,
    }
    response = requests.post(
        url, data=json.dumps(payload), headers=headers).json()

    assert response["result"] == "echome!"
    assert response["jsonrpc"]
    assert response["id"] == 0

if __name__ == "__main__":
    main()
```

Package ensures that request and response messages have correct format. Besides that it provides *jsonrpc.manager.JSONRPCResponseManager* which handles server common cases, such as incorrect message format or invalid method parameters. Futher topics describe how to add methods to manager, how to handle custom exceptions and optional Django integration.

2.2 Method dispatcher

Dispatcher is used to add methods (functions) to the server.

For usage examples see *Dispatcher.add_method()*

class jsonrpc.dispatcher.**Dispatcher**(*prototype=None*)
Dictionary like object which maps method_name to method.

__init__(*prototype=None*)
Build method dispatcher.

Parameters **prototype** (*object or dict, optional*) – Initial method mapping.

Examples

Init object with method dictionary.

```
>>> Dispatcher({"sum": lambda a, b: a + b})
None
```

add_method(*f*, *name=None*)
Add a method to the dispatcher.

Parameters

- **f** (*callable*) – Callable to be added.
- **name** (*str, optional*) – Name to register (the default is function **f** name)

Notes

When used as a decorator keeps callable object unmodified.

Examples

Use as method

```
>>> d = Dispatcher()
>>> d.add_method(lambda a, b: a + b, name="sum")
<function __main__.<lambda>>
```

Or use as decorator

```
>>> d = Dispatcher()
>>> @d.add_method
    def mymethod(*args, **kwargs):
        print(args, kwargs)
```

build_method_map(*prototype*, *prefix=''*)
Add prototype methods to the dispatcher.

Parameters

- **prototype** (*object or dict*) – Initial method mapping. If given prototype is a dictionary then all callable objects will be added to dispatcher. If given prototype is an object then all public methods will be used.
- **prefix** (*string, optional*) – Prefix of methods

2.3 Exceptions

According to specification, error code should be in response message. Http server should respond with status code 200, even if there is an error.

2.3.1 JSON-RPC Errors

Note: Error is an object which represent any kind of erros in JSON-RPC specification. It is not python Exception and could not be raised.

Errors (Error messages) are members of *JSONRPCError* class. Any custom error messages should be inherited from it. The class is responsible for specification following and creates response string based on error's attributes.

JSON-RPC has several predefined errors, each of them has reserved code, which should not be used for custom errors.

Code	Message	Meaning
-32700	Parse error	Invalid JSON was received by the server.An error occurred on the server while parsing the JSON text.
-32600	Invalid Request	The JSON sent is not a valid Request object.
-32601	Method not found	The method does not exist / is not available.
-32602	Invalid params	Invalid method parameter(s).
-32603	Internal error	Internal JSON-RPC error.
-32000 to -32099	Server error	Reserved for implementation-defined server-errors.

JSONRPCResponseManager handles common errors. If you do not plan to implement own manager, you do not need to write custom errors. To controll error messages and codes, json-rpc has exceptions, covered in next paragraph.

2.3.2 JSON-RPC Exceptions

Note: Exception here a json-rpc library object and not related to specification. They are inherited from python Exception and could be raised.

JSON-RPC manager handles dispatcher method's exceptions, anything you raise would be catched. There are two ways to generate error message in manager:

First is to simply raise exception in your method. Manager will catch it and return *JSONRPCServerError* message with description. Advantage of this mehtod is that everything is already implemented, just add method to dispatcher and manager will do the job.

If you need custom message code or error management, you might need to raise exception, inherited from *JSONRPCDispatchException*. Make sure, your exception class has error code.

New in version 1.9.0: Fix *Invalid params* error false generated if method raises TypeError. Now in this case manager introspects the code and returns proper exception.

2.4 Integration with Django

Note: Django backend is optionaly supported. Library itself does not depend on Django.

Django integration is similar project to project. Starting from version 1.8.4 json-rpc support it and provides convenient way of integration. To add json-rpc to Django project follow steps.

2.4.1 Create api instance

If you want to use default (global) object, skip this step. In most cases it is enougth to start with it, even if you plan to add another version later. Default api is located here:

```
from jsonrpc.backend.django import api
```

If you would like to use different api versions (not, you could name methods differently) or use cudtom dispatcher, use

```
from jsonrpc.backend.django import JSONRPCAPI
api = JSONRPCAPI(dispatcher=<my_dispatcher>)
```

Later on we assume that you use default api instance

2.4.2 Add api urls to the project

In your urls.py file add

```
urlpatterns = patterns(
    ...
    url(r'^api/jsonrpc$', include(api.urls)),
)
```

2.4.3 Add methods to api

```
@api.dispatcher.add_method
def my_method(request, *args, **kwargs):
    return args, kwargs
```

Note: first argument of each method should be request. In this case it is possible to get user and control access to data

2.4.4 Make requests to api

To use api, send *POST* request to api address. Make sure your message has correct format. Also json-rpc generates method's map. It is available at *<api_url>/map* url.

2.5 Integration with Flask

Note: Flask backend is optionaly supported. Library itself does not depend on Flask.

2.5.1 Create api instance

If you want to use default (global) object, skip this step. In most cases it is enough to start with it, even if you plan to add another version later. Default api is located here:

```
from jsonrpc.backend.flask import api
```

If you would like to use different api versions (not, you could name methods differently) or use custom dispatcher, use

```
from jsonrpc.backend.flask import JSONRPCAPI
api = JSONRPCAPI(dispatcher=<my_dispatcher>)
```

Later on we assume that you use default api instance.

2.5.2 Add api endpoint to the project

You have to options to add new endpoint to your Flask application.

First - register as a blueprint. In this case, as small bonus, you got a /map handler, which prints all registered methods.

```
from flask import Flask

from jsonrpc.backend.flask import api

app = Flask(__name__)
app.register_blueprint(api.as_blueprint())
```

Second - register as a usual view.

```
from flask import Flask

from jsonrpc.backend.flask import api

app = Flask(__name__)
app.add_url_rule('/', 'api', api.as_view(), methods=['POST'])
```

2.5.3 Add methods to api

```
@api.dispatcher.add_method
def my_method(*args, **kwargs):
    return args, kwargs
```

2.5.4 Make requests to api

To use api, send *POST* request to api address. Make sure your message has correct format.

2.6 jsonrpc Package

2.6.1 JSONRPC

JSON-RPC wrappers for version 1.0 and 2.0.

Objects diring init operation try to choose JSON-RPC 2.0 and in case of error JSON-RPC 1.0. from_json methods could decide what format is it by presence of 'jsonrpc' attribute.

class jsonrpc.jsonrpc.**JSONRPCRequest**
Bases: jsonrpc.utils.JSONSerializable

JSONRPC Request.

classmethod from_data(*data*)

classmethod from_json(*json_str*)

2.6.2 Exceptions

JSON-RPC Exceptions.

exception jsonrpc.exceptions.**JSONRPCDispatchException**(*code=None*, *message=None*, *data=None*, **args*, ***kwargs*)
Bases: *jsonrpc.exceptions.JSONRPCException*

JSON-RPC Dispatch Exception.

Should be thrown in dispatch methods.

class jsonrpc.exceptions.**JSONRPCError**(*code=None*, *message=None*, *data=None*)
Bases: object

Error for JSON-RPC communication.

When a rpc call encounters an error, the Response Object MUST contain the error member with a value that is a Object with the following members:

Parameters

- **code** (*int*) – A Number that indicates the error type that occurred. This MUST be an integer. The error codes from and including -32768 to -32000 are reserved for pre-defined errors. Any code within this range, but not defined explicitly below is reserved for future use. The error codes are nearly the same as those suggested for XML-RPC at the following url: http://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php
- **message** (*str*) – A String providing a short description of the error. The message SHOULD be limited to a concise single sentence.
- **data** (*int or str or dict or list, optional*) – A Primitive or Structured value that contains additional information about the error. This may be omitted. The value of this member is defined by the Server (e.g. detailed error information, nested errors etc.).

code

data

static deserialize(*s*, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)
Deserialize s (a str or unicode instance containing a JSON document) to a Python object.

If s is a str instance and is encoded with an ASCII based encoding other than utf-8 (e.g. latin-1) then an appropriate encoding name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to unicode first.

object_hook is an optional function that will be called with the result of any object literal decode (a dict). The return value of object_hook will be used instead of the dict. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of object_pairs_hook will be used instead of the dict. This feature can be used to implement custom decoders that rely on the order that the key

and value pairs are decoded (for example, collections.OrderedDict will remember the order of insertion). If `object_hook` is also defined, the `object_pairs_hook` takes priority.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

`parse_constant`, if specified, will be called with one of the following strings: -Infinity, Infinity, NaN, null, true, false. This can be used to raise an exception if invalid JSON numbers are encountered.

To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg; otherwise `JSONDecoder` is used.

classmethod `from_json` (*json_str*)

`json`

`message`

static `serialize` (*obj*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *sort_keys=False*, ***kw*)

Serialize `obj` to a JSON formatted `str`.

If `skipkeys` is true then `dict` keys that are not basic types (`str`, `unicode`, `int`, `long`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`.

If `ensure_ascii` is false, all non-ASCII characters are not escaped, and the return value may be a `unicode` instance. See `dump` for details.

If `check_circular` is false, then the circular reference check for container types will be skipped and a circular reference will result in an `OverflowError` (or worse).

If `allow_nan` is false, then it will be a `ValueError` to serialize out of range `float` values (`nan`, `inf`, `-inf`) in strict compliance of the JSON specification, instead of using the JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`).

If `indent` is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. `None` is the most compact representation. Since the default item separator is `', '`, the output might include trailing whitespace when `indent` is specified. You can use `separators=(',', ': ')` to avoid this.

If `separators` is an `(item_separator, dict_separator)` tuple then it will be used instead of the default `(', ', ': ')` separators. `(',', ':')` is the most compact JSON representation.

`encoding` is the character encoding for str instances, default is UTF-8.

`default(obj)` is a function that should return a serializable version of obj or raise TypeError. The default simply raises TypeError.

If *sort_keys* is `True` (default: `False`), then the output of dictionaries will be sorted by key.

To use a custom `JSONEncoder` subclass (e.g. one that overrides the `.default()` method to serialize additional types), specify it with the `cls` kwarg; otherwise `JSONEncoder` is used.

exception `jsonrpc.exceptions.`**`JSONRPCException`**

Bases: `exceptions.Exception`

JSON-RPC Exception.

class jsonrpc.exceptions.**JSONRPCInternalError**(*code=None*, *message=None*, *data=None*)
Bases: *jsonrpc.exceptions.JSONRPCError*

Internal error.

Internal JSON-RPC error.

CODE = -32603

MESSAGE = 'Internal error'

class jsonrpc.exceptions.**JSONRPCInvalidParams**(*code=None*, *message=None*, *data=None*)
Bases: *jsonrpc.exceptions.JSONRPCError*

Invalid params.

Invalid method parameter(s).

CODE = -32602

MESSAGE = 'Invalid params'

class jsonrpc.exceptions.**JSONRPCInvalidRequest**(*code=None*, *message=None*, *data=None*)
Bases: *jsonrpc.exceptions.JSONRPCError*

Invalid Request.

The JSON sent is not a valid Request object.

CODE = -32600

MESSAGE = 'Invalid Request'

exception jsonrpc.exceptions.**JSONRPCInvalidRequestException**
Bases: *jsonrpc.exceptions.JSONRPCException*

Request is not valid.

class jsonrpc.exceptions.**JSONRPCMethodNotFound**(*code=None*, *message=None*, *data=None*)
Bases: *jsonrpc.exceptions.JSONRPCError*

Method not found.

The method does not exist / is not available.

CODE = -32601

MESSAGE = 'Method not found'

class jsonrpc.exceptions.**JSONRPCParseError**(*code=None*, *message=None*, *data=None*)
Bases: *jsonrpc.exceptions.JSONRPCError*

Parse Error.

Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.

CODE = -32700

MESSAGE = 'Parse error'

class jsonrpc.exceptions.**JSONRPCServerError**(*code=None*, *message=None*, *data=None*)
Bases: *jsonrpc.exceptions.JSONRPCError*

Server error.

Reserved for implementation-defined server-errors.

CODE = -32000

MESSAGE = 'Server error'

2.6.3 Manager

class jsonrpc.manager.**JSONRPCResponseManager**
Bases: object

JSON-RPC response manager.

Method brings syntactic sugar into library. Given dispatcher it handles request (both single and batch) and handles errors. Request could be handled in parallel, it is server responsibility.

Parameters

- **request_str** (*str*) – json string. Will be converted into JSONRPC20Request, JSONRPC20BatchRequest or JSONRPC10Request
- **dispather** (*dict*) – dict<function_name:function>.

RESPONSE_CLASS_MAP = {'2.0': <class 'jsonrpc.jsonrpc2.JSONRPC20Response'>, '1.0': <class 'jsonrpc.jsonrpc1.JSON

classmethod handle(*request_str*, *dispatcher*)

classmethod handle_request(*request*, *dispatcher*)
Handle request data.

At this moment request has correct jsonrpc format.

Parameters

- **request** (*dict*) – data parsed from request_str.
- **dispatcher** (jsonrpc.dispatcher.Dispatcher) –

2.6.4 jsonrpc.backend.django module

Python Module Index

j

Index

Symbols

A

B

C

D

F

H

J

M

R

S