typedjsonrpc Documentation

Release unknown

Palantir

August 13, 2015

Contents

1 typedjsonrpc **3**
- 1.1 Using typedjsonrpc . . . 3

2 Modules **7**
- 2.1 Errors . . . 7
- 2.2 Method Info . . . 7
- 2.3 Parameter Checker . . . 8
- 2.4 Registry . . . 9
- 2.5 Server . . . 10

3 Indices and tables **11**

Python Module Index **13**

Contents:

CHAPTER 1

typedjsonrpc

typedjsonrpc is a decorator-based JSON-RPC library for Python that exposes parameter and return types. It is influenced by Flask JSON-RPC but has some key differences:

typedjsonrpc...

- allows return type checking
- focuses on easy debugging

These docs are also available on Read the Docs.

1.1 Using typedjsonrpc

1.1.1 Installation

Clone the repository and install typedjsonrpc:

```
$ pip install git+ssh://git@github.com/palantir/typedjsonrpc.git
```

1.1.2 Project setup

To include typedjsonrpc in your project, use:

```
from typedjsonrpc.registry import Registry
from typedjsonrpc.server import Server

registry = Registry()
server = Server(registry)
```

The registry will keep track of methods that are available for JSON-RPC. Whenever you annotate a method, it will be added to the registry. You can always use the method `rpc.describe()` to get a description of all available methods. `Server` is a WSGI compatible app that handles requests. `Server` also has a development mode that can be run using `server.run(host, port)`.

1.1.3 Example usage

Annotate your methods to make them accessible and provide type information:

```
@registry.method(returns=int, a=int, b=int)
def add(a, b):
    return a + b

@registry.method(returns=str, a=str, b=str)
def concat(a, b):
    return a + b
```

The return type *has* to be declared using the `returns` keyword. For methods that don't return anything, you can use either `type(None)` or just `None`:

```
@registry.method(returns=type(None), a=str)
def foo(a):
    print(a)

@registry.method(returns=None, a=int)
def bar(a):
    print(5 * a)
```

You can use any of the basic JSON types:

JSON type	Python type
string	basestring (Python 2), str (Python 3)
number	int, float
null	None
boolean	bool
array	list
object	dict

Your functions may also accept `*args` and `**kwargs`, but you cannot declare their types. So the correct way to use these would be:

```
@registry.method(a=str)
def foo(a, *args, **kwargs):
    return a + str(args) + str(kwargs)
```

To check that everything is running properly, try (assuming `add` is declared in your main module):

```
$ curl -XPOST http://<host>:<port>/api -d @- <<EOF
{
    "jsonrpc": "2.0",
    "method": "__main__.add",
    "params": {
        "a": 5,
        "b": 7
    },
    "id": "foo"
}
EOF

{
    "jsonrpc": "2.0",
    "id": "foo",
    "result": 12
}
```

Passing any non-integer arguments into `add` will raise a `InvalidParamsError`.

1.1.4 Batching

You can send a list of JSON-RPC request objects as one request and will receive a list of JSON-RPC response objects in return. These response objects can be mapped back to the request objects using the `id`. Here's an example of calling the `add` method with two sets of parameters:

```
$ curl -XPOST http://<host>:<port>/api -d @- <<EOF
[
    {
        "jsonrpc": "2.0",
        "method": "__main__.add",
        "params": {
            "a": 5,
            "b": 7
        },
        "id": "foo"
    }, {
        "jsonrpc": "2.0",
        "method": "__main__.add",
        "params": {
            "a": 42,
            "b": 1337
        },
        "id": "bar"
    }
]
EOF

[
    {
        "jsonrpc": "2.0",
        "id": "foo",
        "result": 12
    }, {
        "jsonrpc": "2.0",
        "id": "bar",
        "result": 1379
    }
]
```

1.1.5 Debugging

If you create the registry with the parameter `debug=True`, you'll be able to use werkzeug's debugger. In that case, if there is an error during execution - e.g. you tried to use a string as one of the parameters for `add` - the response will contain an error object with a `debug_url`:

```
$ curl -XPOST http://<host>:<port>/api -d @- <<EOF
{
    "jsonrpc": "2.0",
    "method": "__main__.add",
    "params": {
        "a": 42,
        "b": "hello"
    },
    "id": "bar"
}
EOF
```

```
{
    "jsonrpc": "2.0",
    "id": "bar",
    "error": {
        "message": "Invalid params",
        "code": -32602,
        "data": {
            "message": "Value 'hello' for parameter 'b' is not of expected type <type 'int'>.",
            "debug_url": "/debug/1234567890"
        }
    }
}
```

This tells you to find the traceback interpreter at `<host>:<port>/debug/1234567890`.

1.1.6 Customizing type serialization

If you would like to serialize custom types, you can set the `json_encoder` and `json_decoder` attributes on `Server` to your own custom `json.JSONEncoder` and `json.JSONDecoder`. By default, we use the default encoder and decoder.

1.1.7 Adding hooks before the first request

You can add functions to run before the first request is called. This can be useful for some special setup you need for your WSGI app. For example, you can register a function to print debugging information before your first request:

```
import datetime

from typedjsonrpc.registry import Registry
from typedjsonrpc.server import Server


registry = Registry()
server = Server()

def print_time():
    now = datetime.datetime.now()
    print("Handling first request at: {}".format(now))

server.register_before_first_request(print_time)
```

CHAPTER 2

Modules

2.1 Errors

Error classes for typedjsonrpc.

exception typedjsonrpc.errors.**Error**(*data=None*)
Base class for all errors.

as_error_object()
Turns the error into an error object.

exception typedjsonrpc.errors.**InternalError**(*data=None*)
Internal JSON-RPC error.

static from_error(*exc*, *debug_url=None*)
Wraps another Exception in an InternalError.

Return type *InternalError*

exception typedjsonrpc.errors.**InvalidParamsError**(*data=None*)
Invalid method parameter(s).

exception typedjsonrpc.errors.**InvalidRequestError**(*data=None*)
The JSON sent is not a valid request object.

exception typedjsonrpc.errors.**InvalidReturnTypeError**(*data=None*)
Return type does not match expected type.

exception typedjsonrpc.errors.**MethodNotFoundError**(*data=None*)
The method does not exist.

exception typedjsonrpc.errors.**ParseError**(*data=None*)
Invalid JSON was received by the server / JSON could not be parsed.

exception typedjsonrpc.errors.**ServerError**(*data=None*)
Something else went wrong.

2.2 Method Info

Data structures for wrapping methods and information about them.

class typedjsonrpc.method_info.**MethodInfo**
An object wrapping a method and information about it.

Attribute name Name of the function

Attribute method The function being described

Attribute signature A description of the types this method takes as parameters and returns

describe()
Describes the method.

Returns Description

Return type dict[str, object]

description
Returns the docstring for this method.

Return type str

params
The parameters for this method in a JSON-compatible format

Return type list[dict[str, str]]

returns
The return type for this method in a JSON-compatible format.

This handles the special case of `None` which allows `type(None)` also.

Return type str or None

2.3 Parameter Checker

Logic for checking parameter declarations and parameter types.

typedjsonrpc.parameter_checker.**check_return_type**(*value*, *expected_type*)
Checks that the given return value has the correct type.

Parameters

- **value** (*any*) – Value returned by the method
- **expected_type** (*type*) – Expected return type

typedjsonrpc.parameter_checker.**check_type_declaration**(*parameter_names*, *parameter_types*)
Checks that exactly the given parameter names have declared types.

Parameters

- **parameter_names** (*list[str]*) – The names of the parameters in the method declaration
- **parameter_types** (*dict[str, type]*) – Parameter type by name

typedjsonrpc.parameter_checker.**check_types**(*parameters*, *parameter_types*)
Checks that the given parameters have the correct types.

Parameters

- **parameters** (*dict[str, object]*) – List of (name, value) pairs of the given parameters
- **parameter_types** (*dict[str,type]*) – Parameter type by name.

typedjsonrpc.parameter_checker.**validate_params_match**(*method*, *parameters*)
Validates that the given parameters are exactly the method's declared parameters.

Parameters

- **method** (*function*) – The method to be called
- **parameters** (*dict[str, object] | list[object]*) – The parameters to use in the call

2.4 Registry

Logic for storing and calling jsonrpc methods.

class typedjsonrpc.registry.**Registry**(*debug=False*)

The registry for storing and calling jsonrpc methods.

Attribute debug Debug option which enables recording of tracebacks

Attribute tracebacks Tracebacks for debugging

describe()

Returns a description of all the methods in the registry.

Returns Description

Return type dict[str, object]

dispatch(*request*)

Takes a request and dispatches its data to a jsonrpc method.

Parameters **request** (*werkzeug.wrappers.Request*) – a werkzeug request with json data

Returns json output of the corresponding method

Return type str

json_decoder

The JSON decoder class to use. Defaults to `json.JSONDecoder`

alias of `JSONDecoder`

json_encoder

The JSON encoder class to use. Defaults to `json.JSONEncoder`

alias of `JSONEncoder`

method(*returns*, ***parameter_types*)

Syntactic sugar for registering a method

Example:

```
>>> registry = Registry()
>>> @registry.method(returns=int, x=int, y=int)
... def add(x, y):
...     return x + y
```

Parameters

- **returns** (*type*) – The method's return type
- **parameter_types** (*dict[str,type]*) – The types of the method's parameters

register(*name*, *method*, *method_signature=None*)

Registers a method with a given name and signature.

Parameters

- **name** (*str*) – The name used to register the method
- **method** (*function*) – The method to register
- **method_signature** (*MethodSignature or None*) – The method signature for the given function

2.5 Server

Contains the Werkzeug server for debugging and WSGI compatibility.

class typedjsonrpc.server.**Server**(*registry*, *endpoint='/api'*)
A basic WSGI-compatible server for typedjsonrpc endpoints.

Attribute registry The registry for this server

register_before_first_request(*func*)
Registers a function to be called once before the first served request.

Parameters **func** (*() -> object*) – Function called

run(*host*, *port*, ***options*)
For debugging purposes, you can run this as a standalone server

wsgi_app(*environ*, *start_response*)
A basic WSGI app

class typedjsonrpc.server.**DebuggedJsonRpcApplication**(*app*, ***kwargs*)
A JSON-RPC-specific debugged application.

This differs from DebuggedApplication since the normal debugger assumes you are hitting the endpoint from a web browser.

A returned response will be JSON of the form: {"traceback_id": <id>} which you can use to hit the endpoint http://<host>:<port>/debug/<traceback_id>.

NOTE: This should never be used in production because the user gets shell access in debug mode.

debug_application(*environ*, *start_response*)
Run the application and preserve the traceback frames.

Parameters

- **environ** (*dict[str, object]*) – The environment which is passed into the wsgi application
- **start_response** (*(str, list[(str, str)]) -> None*) – The start_response function of the wsgi application

Return type generator[str]

handle_debug(*environ*, *start_response*, *traceback_id*)
Handles the debug endpoint for inspecting previous errors.

Parameters

- **environ** (*dict[str, object]*) – The environment which is passed into the wsgi application
- **start_response** (*(str, list[(str, str)]) -> NoneType*) – The start_response function of the wsgi application
- **traceback_id** (*int*) – The id of the traceback to inspect

CHAPTER 3

Indices and tables

- genindex
- modindex

t

typedjsonrpc.errors, 7
typedjsonrpc.method_info, 7
typedjsonrpc.parameter_checker, 8
typedjsonrpc.registry, 9
typedjsonrpc.server, 10

A

as_error_object() (typedjsonrpc.errors.Error method), 7

C

check_return_type() (in module typedjsonrpc.parameter_checker), 8
check_type_declaration() (in module typedjsonrpc.parameter_checker), 8
check_types() (in module typedjsonrpc.parameter_checker), 8

D

debug_application() (typedjsonrpc.server.DebuggedJsonRpcApplication method), 10
DebuggedJsonRpcApplication (class in typedjsonrpc.server), 10
describe() (typedjsonrpc.method_info.MethodInfo method), 8
describe() (typedjsonrpc.registry.Registry method), 9
description (typedjsonrpc.method_info.MethodInfo attribute), 8
dispatch() (typedjsonrpc.registry.Registry method), 9

E

Error, 7

F

from_error() (typedjsonrpc.errors.InternalError static method), 7

H

handle_debug() (typedjsonrpc.server.DebuggedJsonRpcApplication method), 10

I

InternalError, 7
InvalidParamsError, 7
InvalidRequestError, 7
InvalidReturnTypeError, 7

J

json_decoder (typedjsonrpc.registry.Registry attribute), 9
json_encoder (typedjsonrpc.registry.Registry attribute), 9

M

method() (typedjsonrpc.registry.Registry method), 9
MethodInfo (class in typedjsonrpc.method_info), 7
MethodNotFoundError, 7

P

params (typedjsonrpc.method_info.MethodInfo attribute), 8
ParseError, 7

R

register() (typedjsonrpc.registry.Registry method), 9
register_before_first_request() (typedjsonrpc.server.Server method), 10
Registry (class in typedjsonrpc.registry), 9
returns (typedjsonrpc.method_info.MethodInfo attribute), 8
run() (typedjsonrpc.server.Server method), 10

S

Server (class in typedjsonrpc.server), 10
ServerError, 7

T

typedjsonrpc.errors (module), 7
typedjsonrpc.method_info (module), 7
typedjsonrpc.parameter_checker (module), 8
typedjsonrpc.registry (module), 9
typedjsonrpc.server (module), 10

V

validate_params_match() (in module typedjsonrpc.parameter_checker), 8

W

wsgi_app() (typedjsonrpc.server.Server method), 10