

---

**pjrpc**

***Release 0.1.1***

**Nov 09, 2019**



---

## Contents

---

<b>1</b>	<b>Extra requirements</b>	<b>3</b>
<b>2</b>	<b>The User Guide</b>	<b>5</b>
2.1	Installation . . . . .	5
2.2	Quick start . . . . .	5
2.3	Client . . . . .	8
2.4	Server . . . . .	10
2.5	Validation . . . . .	13
2.6	Errors . . . . .	14
2.7	Extending . . . . .	17
2.8	pytest . . . . .	17
2.9	Server testing . . . . .	18
<b>3</b>	<b>The API Documentation</b>	<b>21</b>
3.1	Developer Interface . . . . .	21
<b>4</b>	<b>Development</b>	<b>33</b>
4.1	Development . . . . .	33
<b>5</b>	<b>Indices and tables</b>	<b>35</b>
	<b>Python Module Index</b>	<b>37</b>
	<b>Index</b>	<b>39</b>



pjrpc is a extensible **JSON-RPC** client/server library that provides a very intuitive interface, may be easily extended and integrated in your project without writing a lot of boilerplate code.

Features:

- *intuitive interface*
- *extensibility*
- *synchronous and asynchronous client backends*
- *popular frameworks integration* (aiohttp, flask, kombu, aio\_pika)
- *builtin parameter validation*
- *pytest integration*



# CHAPTER 1

---

## Extra requirements

---

- aiohttp
- aio\_pika
- flask
- jsonschema
- kombu
- pydantic
- requests





## 2.1 Installation

This part of the documentation covers the installation of *pjrpc* library.

### 2.1.1 Installation using pip

To install *pjrpc*, run:

```
$ pip install pjrpc
```

### 2.1.2 Installation from source code

You can clone the repository:

```
$ git clone git@github.com:dapper91/pjrpc.git
```

Then install it:

```
$ cd pjrpc
$ pip install .
```

## 2.2 Quick start

The way of using *pjrpc* clients is very simple and intuitive. Methods may be called by name, using proxy object or by sending handmade *pjrpc.Request* class object. Of course, the request class can be easily inherited and adapted to your needs. Notification requests also supported.

```
import pjrpc
from pjrpc.client.backend import requests as pjrpc_client

client = pjrpc_client.Client('http://localhost/api/v1')

response: pjrpc.Response = client.send(pjrpc.Request('sum', params=[1, 2]))
print(f"1 + 2 = {response.result}")

result = client('sum', a=1, b=2)
print(f"1 + 2 = {result}")

result = client.proxy.sum(1, 2)
print(f"1 + 2 = {result}")

client.notify('tick')
```

Asynchronous client api looks pretty much the same:

```
import pjrpc
from pjrpc.client.backend import aiohttp as pjrpc_client

client = pjrpc_client.Client('http://localhost/api/v1')

response = await client.send(pjrpc.Request('sum', params=[1, 2], id=1))
print(f"1 + 2 = {response.result}")

result = await client('sum', a=1, b=2)
print(f"1 + 2 = {result}")

result = await client.proxy.sum(1, 2)
print(f"1 + 2 = {result}")

await client.notify('tick')
```

pjrpc supports popular backend frameworks like [aiohttp](#), [flask](#) and message brokers like [kombu](#) and [aio\\_pika](#).

Running of aiohttp based JSON-RPC server is a very simple process. Just define methods, add them to the registry and run the server:

```
import uuid

from aiohttp import web

import pjrpc.server
from pjrpc.server.integration import aiohttp

methods = pjrpc.server.MethodRegistry()

@methods.add(context='request')
async def add_user(request: web.Request, user: dict):
    user_id = uuid.uuid4().hex
    request.app['users'][user_id] = user

    return {'id': user_id, **user}
```

(continues on next page)

(continued from previous page)

```

app = aiohttp.Application('/api/v1')
app.dispatcher.add_methods(methods)
app['users'] = {}

if __name__ == "__main__":
    web.run_app(app, host='localhost', port=8080)

```

Very often besides dumb method parameters validation you need to implement more “deep” validation and provide comprehensive errors description to your clients. Fortunately `pjrpc` has builtin parameter validation based on `pydantic` library which uses python type annotation based validation. Look at the following example. All you need to annotate method parameters (or describe more complex type if necessary), that's it. `pjrpc` will be validating method parameters and returning informative errors to clients:

```

import enum
import uuid
from typing import List

import pydantic
from aiohttp import web

import pjrpc.server
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.integration import aiohttp

methods = pjrpc.server.MethodRegistry()
validator = validators.PydanticValidator()

class ContactType(enum.Enum):
    PHONE = 'phone'
    EMAIL = 'email'

class Contact(pydantic.BaseModel):
    type: ContactType
    value: str

class User(pydantic.BaseModel):
    name: str
    surname: str
    age: int
    contacts: List[Contact]

@methods.add(context='request')
@validator.validate
async def add_user(request: web.Request, user: User):
    user_id = uuid.uuid4()
    request.app['users'][user_id] = user

    return {'id': user_id, **user.dict()}

class JSONEncoder(pjrpc.common.JSONEncoder):

```

(continues on next page)

(continued from previous page)

```

def default(self, o):
    if isinstance(o, uuid.UUID):
        return o.hex
    if isinstance(o, enum.Enum):
        return o.value

    return super().default(o)

app = aiohttp.Application('/api/v1', json_encoder=JSONEncoder)
app.dispatcher.add_methods(methods)
app['users'] = {}

if __name__ == "__main__":
    web.run_app(app, host='localhost', port=8080)

```

## 2.3 Client

pjrpc client provides three main method invocation approaches:

- using handmade `pjrpc.Request` class object

```

client = Client('http://server/api/v1')

response: pjrpc.Response = client.send(Request('sum', params=[1, 2], id=1))
print(f"1 + 2 = {response.result}")

```

- using `__call__` method

```

client = Client('http://server/api/v1')

result = client('sum', a=1, b=2)
print(f"1 + 2 = {result}")

```

- using proxy object

```

client = Client('http://server/api/v1')

result = client.proxy.sum(1, 2)
print(f"1 + 2 = {result}")

```

```

client = Client('http://server/api/v1')

result = client.proxy.sum(a=1, b=2)
print(f"1 + 2 = {result}")

```

Requests without `id` in JSON-RPC semantics called notifications. To send a notification to the server you need to send a request without `id`:

```

client = Client('http://server/api/v1')

response: pjrpc.Response = client.send(Request('sum', params=[1, 2]))

```

or use a special method `pjrpc.client.AbstractClient.notify()`

```
client = Client('http://server/api/v1')
client.notify('tick')
```

Asynchronous client api looks pretty much the same:

```
client = Client('http://server/api/v1')

result = await client.proxy.sum(1, 2)
print(f"1 + 2 = {result}")
```

### 2.3.1 Batch requests

Batch requests also supported. There are several approaches of sending batch requests:

- using handmade `pjrpc.Request` class object. The result is a `pjrpc.BatchResponse` instance you can iterate over to get all the results or get each one by the index:

```
client = Client('http://server/api/v1')

batch_response = client.send(BatchRequest(
    pjrpc.Request('sum', [2, 2], id=1),
    pjrpc.Request('sub', [2, 2], id=2),
    pjrpc.Request('div', [2, 2], id=3),
    pjrpc.Request('mult', [2, 2], id=4),
))
print(f"2 + 2 = {batch_response[0].result}")
print(f"2 - 2 = {batch_response[1].result}")
print(f"2 / 2 = {batch_response[2].result}")
print(f"2 * 2 = {batch_response[3].result}")
```

- using `__call__` method chain:

```
client = Client('http://server/api/v1')

result = client.batch('sum', 2, 2)('sub', 2, 2)('div', 2, 2)('mult', 2, 2).call()
print(f"2 + 2 = {result[0]}")
print(f"2 - 2 = {result[1]}")
print(f"2 / 2 = {result[2]}")
print(f"2 * 2 = {result[3]}")
```

- using subscription operator:

```
client = Client('http://server/api/v1')

result = client.batch[
    ('sum', 2, 2),
    ('sub', 2, 2),
    dict(method='div', a=2, b=2),
    dict(method='mult', a=2, b=2),
]
print(f"2 + 2 = {result[0]}")
print(f"2 - 2 = {result[1]}")
print(f"2 / 2 = {result[2]}")
print(f"2 * 2 = {result[3]}")
```

- using proxy chain call:

```
client = Client('http://server/api/v1')

result = client.batch.proxy.sum(2, 2).sub(2, 2).div(2, 2).mult(2, 2).call()
print(f"2 + 2 = {result[0]}")
print(f"2 - 2 = {result[1]}")
print(f"2 / 2 = {result[2]}")
print(f"2 * 2 = {result[3]}")
```

Which one to use is up to you but be aware that if any of the requests returns an error the result of the other ones will be lost. In such case the first approach can be used to iterate over all the responses and get the results of the succeeded ones like this:

```
import pjrpc
from pjrpc.client.backend import requests as pjrpc_client

client = pjrpc_client.Client('http://localhost/api/v1')

batch_response = client.send(pjrpc.BatchRequest(
    pjrpc.Request('sum', [2, 2], id=1),
    pjrpc.Request('sub', [2, 2], id=2),
    pjrpc.Request('div', [2, 2], id=3),
    pjrpc.Request('mult', [2, 2], id=4),
))

for response in batch_response:
    if response.is_success:
        print(response.result)
    else:
        print(response.error)
```

### 2.3.2 Id generators

The library request id generator also can be customized. There are four generators implemented in the library see *pjrpc.common.generators*. You can implement your own one and pass it to a client by *id\_gen* parameter.

## 2.4 Server

pjrpc supports popular backend frameworks like *aiohttp*, *flask* and message brokers like *kombu* and *aio\_pika*.

Running of aiohttp based JSON-RPC server is a very simple process. Just define methods, add them to the registry and run the server:

```
import uuid

from aiohttp import web

import pjrpc.server
from pjrpc.server.integration import aiohttp

methods = pjrpc.server.MethodRegistry()
```

(continues on next page)

(continued from previous page)

```

@methods.add(context='request')
async def add_user(request: web.Request, user: dict):
    user_id = uuid.uuid4().hex
    request.app['users'][user_id] = user

    return {'id': user_id, **user}

app = aiohttp.Application('/api/v1')
app.dispatcher.add_methods(methods)
app['users'] = {}

if __name__ == "__main__":
    web.run_app(app, host='localhost', port=8080)

```

## 2.4.1 Class-based view

pjrpc has a support of class-based method handlers:

```

import uuid

from aiohttp import web

import pjrpc.server
from pjrpc.server.integration import aiohttp

methods = pjrpc.server.MethodRegistry()

@methods.view(context='request', prefix='user')
class UserView(pjrpc.server.View):

    def __init__(self, request: web.Request):
        super().__init__()

        self._users = request.app['users']

    async def add(self, user: dict):
        user_id = uuid.uuid4().hex
        self._users[user_id] = user

        return {'id': user_id, **user}

    async def get(self, user_id: str):
        user = self._users.get(user_id)
        if not user:
            pjrpc.exc.JsonRpcError(code=1, message='not found')

        return user

app = aiohttp.Application('/api/v1')
app.dispatcher.add_methods(methods)
app['users'] = {}

```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    web.run_app(app, host='localhost', port=8080)
```

## 2.4.2 API versioning

API versioning is a framework dependant feature but `pjrpc` has a full support for that. Look at the following example illustrating how `aiohttp` JSON-RPC versioning is simple:

```
import uuid

from aiohttp import web

import pjrpc.server
from pjrpc.server.integration import aiohttp

methods_v1 = pjrpc.server.MethodRegistry()

@methods_v1.add(context='request')
async def add_user(request: web.Request, user: dict):
    user_id = uuid.uuid4().hex
    request.config_dict['users'][user_id] = user

    return {'id': user_id, **user}

methods_v2 = pjrpc.server.MethodRegistry()

@methods_v2.add(context='request')
async def add_user(request: web.Request, user: dict):
    user_id = uuid.uuid4().hex
    request.config_dict['users'][user_id] = user

    return {'id': user_id, **user}

app = web.Application()
app['users'] = {}

app_v1 = aiohttp.Application()
app_v1.dispatcher.add_methods(methods_v1)
app.add_subapp('/api/v1', app_v1)

app_v2 = aiohttp.Application()
app_v2.dispatcher.add_methods(methods_v2)
app.add_subapp('/api/v2', app_v2)

if __name__ == "__main__":
    web.run_app(app, host='localhost', port=8080)
```



## 2.5 Validation

Very often besides dumb method parameters validation you need to implement more “deep” validation and provide comprehensive errors description to your clients. Fortunately `pjrpc` has builtin parameter validation based on `pydantic` library which uses python type annotation based validation. Look at the following example. All you need to annotate method parameters (or describe more complex type if necessary), that's it. `pjrpc` will be validating method parameters and returning informative errors to clients:

```
import enum
import uuid
from typing import List

import pydantic
from aiohttp import web

import pjrpc.server
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.integration import aiohttp

methods = pjrpc.server.MethodRegistry()
validator = validators.PydanticValidator()

class ContactType(enum.Enum):
    PHONE = 'phone'
    EMAIL = 'email'

class Contact(pydantic.BaseModel):
    type: ContactType
    value: str

class User(pydantic.BaseModel):
    name: str
    surname: str
    age: int
    contacts: List[Contact]

@methods.add(context='request')
@validator.validate
async def add_user(request: web.Request, user: User):
    user_id = uuid.uuid4()
    request.app['users'][user_id] = user

    return {'id': user_id, **user.dict()}

class JSONEncoder(pjrpc.common.JSONEncoder):

    def default(self, o):
        if isinstance(o, uuid.UUID):
            return o.hex
        if isinstance(o, enum.Enum):
            return o.value
```

(continues on next page)

(continued from previous page)

```

        return super().default(o)

app = aiohttp.Application('/api/v1', json_encoder=JSONEncoder)
app.dispatcher.add_methods(methods)
app['users'] = {}

if __name__ == "__main__":
    web.run_app(app, host='localhost', port=8080)

```

The library also support builtin `pjrpc.server.validators.jsonschema` validator. In case you like any other validation library/framework like `cerberus` you can easily implement it and integrate into your project.

## 2.6 Errors

### 2.6.1 Errors handling

`pjrpc` implements all the errors listed in [protocol specification](#):

code	message	meaning
-32700	Parse error	Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.
-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.

Errors can be found in `pjrpc.common.exceptions` module. Having said that error handling is very simple and “pythonic-way”:

```

import pjrpc
from pjrpc.client.backend import requests as pjrpc_client

client = pjrpc_client.Client('http://localhost/api/v1')

try:
    result = client.proxy.sum(1, 2)
except pjrpc.MethodNotFound as e:
    print(e)

```

### 2.6.2 Custom errors

Default error list may be easily extended. All you need to create an error class inherited from `pjrpc.exc.JsonRpcError` and define an error code and a description message. `pjrpc` will be automatically deserializing

custom errors for you:

```
import pjrpc
from pjrpc.client.backend import requests as pjrpc_client

class UserNotFound(pjrpc.exc.JsonRpcError):
    code = 1
    message = 'user not found'

client = pjrpc_client.Client('http://localhost/api/v1')

try:
    result = client.proxy.get_user(user_id=1)
except UserNotFound as e:
    print(e)
```

### 2.6.3 Server side

On the server side everything is pretty straightforward:

```
import uuid

import flask

import pjrpc
from pjrpc.server import MethodRegistry
from pjrpc.server.integration import flask as integration

app = flask.Flask(__name__)

methods = pjrpc.server.MethodRegistry()

class UserNotFound(pjrpc.exc.JsonRpcError):
    code = 1
    message = 'user not found'

@methods.add
def add_user(user: dict):
    user_id = uuid.uuid4().hex
    flask.current_app.users[user_id] = user

    return {'id': user_id, **user}

def get_user(self, user_id: str):
    user = flask.current_app.users.get(user_id)
    if not user:
        raise UserNotFound(data=user_id)

    return user

json_rpc = integration.JsonRPC('/api/v1')
json_rpc.dispatcher.add_methods(methods)
```

(continues on next page)

(continued from previous page)

```

app.users = {}

json_rpc.init_app(app)

if __name__ == "__main__":
    app.run(port=80)

```

## 2.6.4 Independent clients errors

Having multiple JSON-RPC services with overlapping error codes is a “real-world” case everyone has ever dialed with. To handle such situation client has an *error\_cls* argument to set a base error class for a particular client:

```

import pjrpc
from pjrpc.client.backend import requests as jrpc_client

class ErrorV1(pjrpc.exc.JsonRpcError):
    @classmethod
    def get_error_cls(cls, code, default):
        return next(iter((c for c in cls.__subclasses__() if getattr(c, 'code', None) == code)), default)

class PermissionDenied(ErrorV1):
    code = 1
    message = 'permission denied'

class ErrorV2(pjrpc.exc.JsonRpcError):
    @classmethod
    def get_error_cls(cls, code, default):
        return next(iter((c for c in cls.__subclasses__() if getattr(c, 'code', None) == code)), default)

class ResourceNotFound(ErrorV2):
    code = 1
    message = 'resource not found'

client_v1 = jrpc_client.Client('http://localhost:8080/api/v1', error_cls=ErrorV1)
client_v2 = jrpc_client.Client('http://localhost:8080/api/v2', error_cls=ErrorV2)

try:
    response: pjrpc.Response = client_v1.proxy.add_user(user={})
except PermissionDenied as e:
    print(e)

try:
    response: pjrpc.Response = client_v2.proxy.add_user(user={})
except ResourceNotFound as e:
    print(e)

```

The above snippet illustrates two clients receiving the same error code however each one has its own semantic and therefore its own exception class. Nevertheless clients raise their own exceptions for the same error code.

## 2.7 Extending

TBD

## 2.8 pytest

pjrpc implements pytest plugin that simplifies JSON-RPC requests mocking. Look at the following test example:

```
import pytest
from unittest import mock

import pjrpc
from pjrpc.client.integrations.pytest import PjRpcAsyncMocker
from pjrpc.client.backend import aiohttp as aiohttp_client

async def test_using_fixture(pjrpc_aiohttp_mock):
    client = aiohttp_client.Client('http://localhost/api/v1')

    pjrpc_aiohttp_mock.add('http://localhost/api/v1', 'sum', result=2)
    result = await client.proxy.sum(1, 1)
    assert result == 2

    pjrpc_aiohttp_mock.replace(
        'http://localhost/api/v1', 'sum', error=pjrpc.exc.JsonRpcError(code=1,
↪message='error', data='oops')
    )
    with pytest.raises(pjrpc.exc.JsonRpcError) as exc_info:
        await client.proxy.sum(a=1, b=1)

    assert exc_info.type is pjrpc.exc.JsonRpcError
    assert exc_info.value.code == 1
    assert exc_info.value.message == 'error'
    assert exc_info.value.data == 'oops'

    sum_calls = pjrpc_aiohttp_mock.calls[('http://localhost/api/v1', '2.0', 'sum')]
    assert sum_calls.call_count == 2
    assert sum_calls.mock_calls == [mock.call(1, 1), mock.call(a=1, b=1)]

async def test_using_resource_manager():
    client = aiohttp_client.Client('http://localhost/api/v1')

    with PjRpcAsyncMocker() as mocker:
        mocker.add('http://localhost/api/v1', 'div', result=2)
        result = await client.proxy.div(4, 2)
        assert result == 2
        assert mocker.calls[('http://localhost/api/v1', '2.0', 'div')].mock_calls ==
↪[mock.call(4, 2)]
```

## 2.9 Server testing

For testing server-side code you should use framework-dependant utils and fixtures. Since `pjrpc` may be easily extended you are free from writing JSON-RPC protocol related code.

### 2.9.1 aiohttp

Testing `aiohttp` server code is very straightforward:

```
import uuid

from aiohttp import web

import pjrpc.server
from pjrpc.server.integration import aiohttp
from pjrpc.client.backend import aiohttp as aiohttp_client

methods = pjrpc.server.MethodRegistry()

@methods.add
async def sum(request: web.Request, a, b):
    return a + b

app = aiohttp.Application('/api/v1')
app.dispatcher.add_methods(methods)

async def test_sum(aiohttp_client, loop):
    session = await aiohttp_client(app)
    client = aiohttp_client.Client('http://localhost/api/v1', session=session)

    result = await client.sum(a=1, b=1)
    assert result == 2
```

### 2.9.2 flask

For flask it stays the same:

```
import uuid

import flask

from pjrpc.server.integration import flask as integration
from pjrpc.client.backend import requests as pjrpc_client

methods = pjrpc.server.MethodRegistry()

@methods.add
def sum(request: web.Request, a, b):
    return a + b

app = flask.Flask(__name__)
json_rpc = integration.JsonRPC('/api/v1')
json_rpc.dispatcher.add_methods(methods)
json_rpc.init_app(app)
```

(continues on next page)

(continued from previous page)

```
def test_sum():
    with app.test_client() as c:
        client = pjrpc_client.Client('http://localhost/api/v1', session=c)
        result = await client.sum(a=1, b=1)
        assert result == 2
```





## 3.1 Developer Interface

Extensible **JSON-RPC** client/server library.

### 3.1.1 Common

Client and server common functions, types and classes that implements JSON-RPC protocol itself and agnostic to any transport protocol layer (http, socket, amqp) and server-side implementation.

**class** `pjrpc.common.Request` (*method, params=None, id=None*)  
JSON-RPC version 2.0 request.

**Parameters**

- **method** – method name
- **params** – method parameters
- **id** – request identifier

**classmethod** `from_json` (*json\_data*)  
Deserializes a request from json data.

**Parameters** `json_data` – data the request to be deserialized from

**Returns** request object

**id**  
Request identifier.

**is\_notification**  
Returns `True` if the request is a notification e.g. *id* is `None`.

**method**  
Request method name.

**params**

Request method parameters.

**to\_json()**

Serializes the request to json data.

**Returns** json data

**class** `pjrpc.common.Response` (*id, result=UNSET, error=UNSET*)

JSON-RPC version 2.0 response.

**Parameters**

- **id** – response identifier
- **result** – response result
- **error** – response error

**error**

Response error. If the response has succeeded returns `pjrpc.common.UNSET`.

**classmethod** `from_json` (*json\_data, error\_cls=<class 'pjrpc.common.exceptions.JsonRpcError'>*)

Deserializes a response from json data.

**Parameters**

- **json\_data** – data the response to be deserialized from
- **error\_cls** – error class

**Returns** response object

**id**

Response identifier.

**is\_error**

Returns `True` if the response has not succeeded.

**is\_success**

Returns `True` if the response has succeeded.

**related**

Returns the request related response object if the response has been received from the server otherwise returns `None`.

**result**

Response result. If the response has not succeeded raises an exception deserialized from the *error* filed.

**to\_json()**

Serializes the response to json data.

**Returns** json data

**class** `pjrpc.common.BatchRequest` (*\*requests, strict=True*)

JSON-RPC 2.0 batch request.

**Parameters**

- **requests** – requests to be added to the batch
- **strict** – if `True` checks response identifier uniqueness

**append** (*request*)

Appends a request to the batch.

**extend** (*requests*)

Extends a batch with *requests*.

**classmethod from\_json** (*data*)

Deserializes a batch request from json data.

**Parameters** *data* – data the request to be deserialized from

**Returns** batch request object

**to\_json** ()

Serializes the request to json data.

**Returns** json data

**class** pjrpc.common.**BatchResponse** (\**responses*, *strict=True*)

JSON-RPC 2.0 batch response.

**Parameters**

- **responses** – responses to be added to the batch
- **strict** – if *True* checks response identifier uniqueness

**append** (*response*)

Appends a response to the batch.

**extend** (*responses*)

Extends the batch with the *responses*.

**classmethod from\_json** (*data*)

Deserializes a batch response from json data.

**Parameters** *data* – data the response to be deserialized from

**Returns** batch response object

**has\_error**

Returns *True* if any response has an error.

**related**

Returns the request related response object if the response has been received from the server otherwise returns *None*.

**result**

Returns the batch result as a tuple. If any response of the batch has an error raises an exception of the first errored response.

**to\_json** ()

Serializes the batch response to json data.

**Returns** json data

**class** pjrpc.common.**JSONEncoder** (\*, *skipkeys=False*, *ensure\_ascii=True*, *check\_circular=True*,  
*allow\_nan=True*, *sort\_keys=False*, *indent=None*, *separators=None*, *default=None*)

Library default JSON encoder. Encodes request, response and error objects to be json serializable. All custom encoders should be inherited from it.

**default** (*o*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a *TypeError*).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

## Exceptions

Definition of package exceptions and JSON-RPC protocol errors.

**exception** `pjrpc.common.exceptions.BaseError`  
Base package error. All package errors are inherited from it.

**exception** `pjrpc.common.exceptions.ClientError` (`code=None`, `message=None`,  
`data=UNSET`)  
Raised when client sent an incorrect request.

**exception** `pjrpc.common.exceptions.DeserializationError`  
Request/response deserializatoin error. Raised when request/response json has incorrect format.

**exception** `pjrpc.common.exceptions.IdentityError`  
Raised when a batch requests/responses identifiers are not unique.

**exception** `pjrpc.common.exceptions.InternalError` (`code=None`, `message=None`,  
`data=UNSET`)  
Internal JSON-RPC error.

**exception** `pjrpc.common.exceptions.InvalidParamsError` (`code=None`, `message=None`,  
`data=UNSET`)  
Invalid method parameter(s).

**exception** `pjrpc.common.exceptions.InvalidRequestError` (`code=None`, `message=None`,  
`data=UNSET`)  
The JSON sent is not a valid request object.

**exception** `pjrpc.common.exceptions.JsonRpcError` (`code=None`, `message=None`,  
`data=UNSET`)  
JSON-RPC protocol error. For more information see [Error object](#). All protocol errors are inherited from it.

### Parameters

- **code** – number that indicates the error type
- **message** – short description of the error
- **data** – value that contains additional information about the error. May be omitted.

**classmethod** `from_json(json_data)`  
Deserializes an error from json data. If data format is not correct `ValueError` is raised.

**Parameters** `json_data` – json data the error to be deserialized from

**Returns** deserialized error

**Raises** `ValueError` if format is incorrect

**to\_json()**  
Serializes the error to a dict.

**Returns** serialized error

```
class pjrpc.common.exceptions.JsonRpcErrorMeta
    pjrpc.common.exceptions.JsonRpcError metaclass. Builds a mapping from an error code number
    to an error class inherited from a pjrpc.common.exceptions.JsonRpcError.

exception pjrpc.common.exceptions.MethodNotFoundError (code=None, message=None,
                                                         data=UNSET)
    The method does not exist / is not available.

exception pjrpc.common.exceptions.ParseError (code=None, message=None,
                                                data=UNSET)
    Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.

exception pjrpc.common.exceptions.ServerError (code=None, message=None,
                                                data=UNSET)
    Reserved for implementation-defined server-errors. Codes from -32000 to -32099.
```

## Identifier generators

Builtin request id generators. Implements several identifier types and generation strategies.

```
pjrpc.common.generators.randint (a, b)
    Random integer id generator. Returns random integers between a and b.

pjrpc.common.generators.random (length=8, chars='0123456789abcdefghijklmnopqrstuvwxyz')
    Random string id generator. Returns random strings of length length using alphabet chars.

pjrpc.common.generators.sequential (start=1, step=1)
    Sequential id generator. Returns consecutive values starting from start with step step.

pjrpc.common.generators.uuid()
    UUID id generator. Returns random UUIDs.
```

## 3.1.2 Client

JSON-RPC client.

```
class pjrpc.client.AbstractClient (request_class=<class 'pjrpc.common.v20.Request'>, re-
                                response_class=<class 'pjrpc.common.v20.Response'>,
                                batch_request_class=<class
                                'pjrpc.common.v20.BatchRequest'>,
                                batch_response_class=<class
                                'pjrpc.common.v20.BatchResponse'>, id_gen=<function
                                sequential>, json_loader=<function loads>,
                                json_dumper=<function dumps>, json_encoder=<class
                                'pjrpc.common.common.JSONEncoder'>,
                                json_decoder=None, error_cls=<class
                                'pjrpc.common.exceptions.JsonRpcError'>, strict=True,
                                request_args=None)
```

Abstract JSON-RPC client.

### Parameters

- **request\_class** – request class
- **response\_class** – response class
- **batch\_request\_class** – batch request class

- **batch\_response\_class** – batch response class
- **id\_gen** – identifier generator
- **json\_loader** – json loader
- **json\_dumper** – json dumper
- **json\_encoder** – json encoder
- **json\_decoder** – json decoder
- **error\_cls** – JSON-RPC error base class
- **strict** – if `True` checks that a request and a response identifiers match

**class Proxy** (*client*)

Proxy object. Provides syntactic sugar to make method call using dot notation.

**Parameters** **client** – JSON-RPC client instance

**batch**

Client batch wrapper.

**call** (*method, \*args, \*\*kwargs*)

Makes JSON-RPC call.

**Parameters**

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments

**Returns** response result

**notify** (*method, \*args, \*\*kwargs*)

Makes a notification request

**Parameters**

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments

**proxy**

Client proxy object.

**send** (*request, \*\*kwargs*)

Sends a JSON-RPC request.

**Parameters**

- **request** – request instance
- **kwargs** – additional client request argument

**Returns** response instance

```

class pjrpc.client.AbstractAsyncClient (request_class=<class
                                     'pjrpc.common.v20.Request'>,
                                     response_class=<class
                                     'pjrpc.common.v20.Response'>,
                                     batch_request_class=<class
                                     'pjrpc.common.v20.BatchRequest'>,
                                     batch_response_class=<class
                                     'pjrpc.common.v20.BatchResponse'>,
                                     id_gen=<function sequential>,
                                     json_loader=<function loads>,
                                     json_dumper=<function dumps>,
                                     json_encoder=<class
                                     'pjrpc.common.common.JSONEncoder'>,
                                     json_decoder=None,
                                     error_cls=<class
                                     'pjrpc.common.exceptions.JsonRpcError'>,
                                     strict=True, request_args=None)

```

Abstract asynchronous JSON-RPC client.

#### **batch**

Client batch wrapper.

**call** (*method*, \**args*, \*\**kwargs*)

Makes JSON-RPC call.

#### **Parameters**

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments

**Returns** response result

**notify** (*method*, \**args*, \*\**kwargs*)

Makes a notification request

#### **Parameters**

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments

**send** (*request*, \*\**kwargs*)

Sends a JSON-RPC request.

#### **Parameters**

- **request** – request instance
- **kwargs** – additional client request argument

**Returns** response instance

## **Backends**

```

class pjrpc.client.backend.requests.Client (url, session=None, **kwargs)

```

[Requests](#) library client backend.

#### **Parameters**

- **url** – base url to be used as JSON-RPC endpoint.
- **session** – custom session to be used instead of `requests.Session`
- **kwargs** – parameters to be passed to `pjrpc.client.AbstractClient`

**close()**

Closes the current http session.

## Integrations

### 3.1.3 Server

JSON-RPC server package.

```
class pjrpc.server.Dispatcher(*, request_class=<class 'pjrpc.common.v20.Request'>,
                             response_class=<class 'pjrpc.common.v20.Response'>,
                             batch_request=<class 'pjrpc.common.v20.BatchRequest'>,
                             batch_response=<class 'pjrpc.common.v20.BatchResponse'>,
                             json_loader=<function loads>, json_dumper=<function dumps>,
                             json_encoder=None, json_decoder=None, error_handler=None)
```

Method dispatcher.

#### Parameters

- **request\_class** – JSON-RPC request class
- **response\_class** – JSON-RPC response class
- **batch\_request** – JSON-RPC batch request class
- **batch\_response** – JSON-RPC batch response class
- **json\_loader** – request json loader
- **json\_dumper** – response json dumper
- **json\_encoder** – response json encoder
- **json\_decoder** – request json decoder
- **error\_handler** – error handling function

**add** (*method*, *name=None*, *context=None*)

Adds method to the registry.

#### Parameters

- **method** – method
- **name** – method name
- **context** – application context name

#### Returns

**add\_methods** (*\*methods*)

Adds methods to the registry.

**Parameters** **methods** – method list. Each method may be an instance of `pjrpc.server.MethodRegistry`, `pjrpc.server.Method` or plain function

**dispatch** (*request\_text*, *context=None*)

Deserializes request, dispatches it to the required method and serializes the result.



**Parameters**

- **request\_text** – request text representation
- **context** – application context (if supported)

**Returns** response text representation

**view** (*view*)

Adds class based view to the registry.

**Parameters** **view** – view to be added

```
class pjrpc.server.AsyncDispatcher(*, request_class=<class
    'pjrpc.common.v20.Request'>, response_class=<class
    'pjrpc.common.v20.Response'>, batch_request=<class
    'pjrpc.common.v20.BatchRequest'>,
    batch_response=<class 'pjrpc.common.v20.BatchResponse'>,
    json_loader=<function loads>, json_dumper=<function
    dumps>, json_encoder=None, json_decoder=None,
    error_handler=None)
```

Asynchronous method dispatcher.

**dispatch** (*request\_text*, *context=None*)

Deserializes request, dispatches it to the required method and serializes the result.

**Parameters**

- **request\_text** – request text representation
- **context** – application context (if supported)

**Returns** response text representation

```
class pjrpc.server.Method(method, name=None, context=None)
```

JSON-RPC method wrapper. Stores method and some metainformation.

**Parameters**

- **method** – method
- **name** – method name
- **context** – context name

```
class pjrpc.server.MethodRegistry(prefix=None)
```

Method registry.

**Parameters** **prefix** – method name prefix to be used for naming containing methods

```
add(maybe_method=None, name=None, context=None)
```

Decorator adding decorated method to the registry.

**Parameters**

- **maybe\_method** – method or *None*
- **name** – method name to be used instead of `__name__` attribute
- **context** – parameter name to be used as an application context

**Returns** decorated method or decorator

```
add_methods(*methods)
```

Adds methods to the registry.

**Parameters methods** – methods to be added. Each one can be an instance of *pjrpc.server.Method* or plain method

**get** (*item*)

Returns a method from the registry by name.

**Parameters item** – method name

**Returns** found method or *None*

**merge** (*other*)

Merges two registries.

**Parameters other** – registry to be merged in the current one

**view** (*maybe\_view=None, context=None, prefix=None*)

Methods view decorator.

**Parameters**

- **maybe\_view** – view class instance or *None*
- **context** – application context name
- **prefix** – view methods prefix

**Returns** decorator or decorated view

**class** `pjrpc.server.View`

Class based method handler.

## Integrations

### aiohttp

### flask

### kombu

### aio\_pika

### httpserver

Standard python http server JSON-RPC integration.

**class** `pjrpc.server.integration.httpserver.JsonRpcHandler` (*request, client\_address, server*)

JSON-RPC handler.

**do\_POST** ()

Handles JSON-RPC request.

**class** `pjrpc.server.integration.httpserver.JsonRpcServer` (*server\_address, RequestHandler, Class=<class 'pjrpc.server.integration.httpserver.JsonRpcHandler', bind\_and\_activate=True, \*\*kwargs*)

`http.server.HTTPServer` based JSON-RPC server.

**Parameters**

- **path** – JSON-RPC handler base path
- **kwargs** – arguments to be passed to the dispatcher *pjrpc.server.Dispatcher*

**dispatcher**

JSON-RPC method dispatcher.

**werkzeug****Validators**

JSON-RPC method parameters validators.

**class** *pjrpc.server.validators.BaseValidator*

Base method parameters validator. Uses *inspect.signature()* for validation.

**bind** (*signature, params*)

Binds parameters to method. :param signature: method to bind parameters to :param params: parameters to be bound

**Raises** *ValidationError* is parameters binding failed

**Returns** bound parameters

**signature**

Returns method signature.

**Parameters**

- **method** – method to get signature of
- **exclude** – parameters to be excluded

**Returns** signature

**validate** (*maybe\_method=None, \*\*kwargs*)

Decorator marks a method the parameters of which to be validated when calling it using JSON-RPC protocol.

**Parameters**

- **maybe\_method** – method the parameters of which to be validated or *None* if called as *@validate(...)*
- **kwargs** – validator arguments

**validate\_method** (*method, params, exclude=(), \*\*kwargs*)

Validates params against method signature.

**Parameters**

- **method** – method to validate parameters against
- **params** – parameters to be validated
- **exclude** – parameter names to be excluded from validation
- **kwargs** – additional validator arguments

**Raises** *pjrpc.server.validators.ValidationError*

**Returns** bound method parameters

**exception** `pjrpc.server.validators.ValidationError`  
Method parameters validation error. Raised when parameters validation failed.

**jsonschema**

**pydantic**

### 4.1 Development

Install pre-commit hooks:

```
$ pre-commit install
```

For more information see [pre-commit](#)

You can run code check manually:

```
$ pre-commit run --all-file
```



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





### p

- `pjrpc`, [21](#)
- `pjrpc.client`, [25](#)
- `pjrpc.client.backend.requests`, [27](#)
- `pjrpc.common`, [21](#)
- `pjrpc.common.exceptions`, [24](#)
- `pjrpc.common.generators`, [25](#)
- `pjrpc.server`, [28](#)
- `pjrpc.server.integration.httpserver`, [30](#)
- `pjrpc.server.validators`, [31](#)



## A

AbstractAsyncClient (class in *pjrpc.client*), 26  
AbstractClient (class in *pjrpc.client*), 25  
AbstractClient.Proxy (class in *pjrpc.client*), 26  
add() (*pjrpc.server.Dispatcher* method), 28  
add() (*pjrpc.server.MethodRegistry* method), 29  
add\_methods() (*pjrpc.server.Dispatcher* method), 28  
add\_methods() (*pjrpc.server.MethodRegistry* method), 29  
append() (*pjrpc.common.BatchRequest* method), 22  
append() (*pjrpc.common.BatchResponse* method), 23  
AsyncDispatcher (class in *pjrpc.server*), 29

## B

BaseError, 24  
BaseValidator (class in *pjrpc.server.validators*), 31  
batch (*pjrpc.client.AbstractAsyncClient* attribute), 27  
batch (*pjrpc.client.AbstractClient* attribute), 26  
BatchRequest (class in *pjrpc.common*), 22  
BatchResponse (class in *pjrpc.common*), 23  
bind() (*pjrpc.server.validators.BaseValidator* method), 31

## C

call() (*pjrpc.client.AbstractAsyncClient* method), 27  
call() (*pjrpc.client.AbstractClient* method), 26  
Client (class in *pjrpc.client.backend.requests*), 27  
ClientError, 24  
close() (*pjrpc.client.backend.requests.Client* method), 28

## D

default() (*pjrpc.common.JSONEncoder* method), 23  
DeserializationError, 24  
dispatch() (*pjrpc.server.AsyncDispatcher* method), 29  
dispatch() (*pjrpc.server.Dispatcher* method), 28  
Dispatcher (class in *pjrpc.server*), 28  
dispatcher (*pjrpc.server.integration.httpserver.JsonRpcServer* attribute), 31

do\_POST() (*pjrpc.server.integration.httpserver.JsonRpcHandler* method), 30

## E

error (*pjrpc.common.Response* attribute), 22  
extend() (*pjrpc.common.BatchRequest* method), 22  
extend() (*pjrpc.common.BatchResponse* method), 23

## F

from\_json() (*pjrpc.common.BatchRequest* class method), 23  
from\_json() (*pjrpc.common.BatchResponse* class method), 23  
from\_json() (*pjrpc.common.exceptions.JsonRpcError* class method), 24  
from\_json() (*pjrpc.common.Request* class method), 21  
from\_json() (*pjrpc.common.Response* class method), 22

## G

get() (*pjrpc.server.MethodRegistry* method), 30

## H

has\_error (*pjrpc.common.BatchResponse* attribute), 23

## I

id (*pjrpc.common.Request* attribute), 21  
id (*pjrpc.common.Response* attribute), 22  
IdentityError, 24  
InternalError, 24  
InvalidParamsError, 24  
InvalidRequestError, 24  
is\_error (*pjrpc.common.Response* attribute), 22  
is\_notification (*pjrpc.common.Request* attribute), 21  
is\_success (*pjrpc.common.Response* attribute), 22

## J

JSONEncoder (class in *pjrpc.common*), 23  
 JsonRpcError, 24  
 JsonRpcErrorMeta (class  
     *pjrpc.common.exceptions*), 25  
 JsonRpcHandler (class  
     *pjrpc.server.integration.httpserver*), 30  
 JsonRpcServer (class  
     *pjrpc.server.integration.httpserver*), 30

## M

merge() (*pjrpc.server.MethodRegistry* method), 30  
 Method (class in *pjrpc.server*), 29  
 method (*pjrpc.common.Request* attribute), 21  
 MethodNotFoundError, 25  
 MethodRegistry (class in *pjrpc.server*), 29

## N

notify() (*pjrpc.client.AbstractAsyncClient* method),  
     27  
 notify() (*pjrpc.client.AbstractClient* method), 26

## P

params (*pjrpc.common.Request* attribute), 21  
 ParseError, 25  
 pjrpc (module), 21  
 pjrpc.client (module), 25  
 pjrpc.client.backend.requests (module), 27  
 pjrpc.common (module), 21  
 pjrpc.common.exceptions (module), 24  
 pjrpc.common.generators (module), 25  
 pjrpc.server (module), 28  
 pjrpc.server.integration.httpserver  
     (module), 30  
 pjrpc.server.validators (module), 31  
 proxy (*pjrpc.client.AbstractClient* attribute), 26

## R

randint() (in module *pjrpc.common.generators*), 25  
 random() (in module *pjrpc.common.generators*), 25  
 related (*pjrpc.common.BatchResponse* attribute), 23  
 related (*pjrpc.common.Response* attribute), 22  
 Request (class in *pjrpc.common*), 21  
 Response (class in *pjrpc.common*), 22  
 result (*pjrpc.common.BatchResponse* attribute), 23  
 result (*pjrpc.common.Response* attribute), 22

## S

send() (*pjrpc.client.AbstractAsyncClient* method), 27  
 send() (*pjrpc.client.AbstractClient* method), 26  
 sequential() (in module *pjrpc.common.generators*),  
     25  
 ServerError, 25

signature (*pjrpc.server.validators.BaseValidator* at-  
     tribute), 31

## T

in to\_json() (*pjrpc.common.BatchRequest* method), 23  
 in to\_json() (*pjrpc.common.BatchResponse* method), 23  
 to\_json() (*pjrpc.common.exceptions.JsonRpcError*  
     method), 24  
 in to\_json() (*pjrpc.common.Request* method), 22  
 to\_json() (*pjrpc.common.Response* method), 22

## U

uuid() (in module *pjrpc.common.generators*), 25

## V

validate() (*pjrpc.server.validators.BaseValidator*  
     method), 31  
 validate\_method()  
     (*pjrpc.server.validators.BaseValidator*  
     method), 31  
 ValidationError, 31  
 View (class in *pjrpc.server*), 30  
 view() (*pjrpc.server.Dispatcher* method), 29  
 view() (*pjrpc.server.MethodRegistry* method), 30