
pjrpc

Release 1.0.0

Mar 14, 2020

Contents

1	Extra requirements	3
2	The User Guide	5
2.1	Installation	5
2.2	Quick start	5
2.3	Client	11
2.4	Server	14
2.5	Validation	16
2.6	Errors	17
2.7	Extending	20
2.8	Testing	21
2.9	Tracing	23
2.10	Examples	25
3	The API Documentation	47
3.1	Developer Interface	47
4	Development	65
4.1	Development	65
5	Indices and tables	67
	Python Module Index	69
	Index	71

`pjrpc` is an extensible **JSON-RPC** client/server library with an intuitive interface that can 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

2.2.1 Client requests

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.common.Request* class object. Notification requests can be made using *pjrpc.client.AbstractClient.notify()* method or by sending a *pjrpc.common.Request* object without id.

```
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], id=1))
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')
```

2.2.2 Batch requests

Batch requests also supported. You can build *pjrpc.common.BatchRequest* request by your hand and then send it to the server. The result is a *pjrpc.common.BatchResponse* instance you can iterate over to get all the results or get each one by index:

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

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

batch_response = await client.batch.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),
))
print(f"2 + 2 = {batch_response[0].result}")
```

(continues on next page)

(continued from previous page)

```
print(f"2 - 2 = {batch_response[1].result}")
print(f"2 / 2 = {batch_response[2].result}")
print(f"2 * 2 = {batch_response[3].result}")
```

There are also several alternative approaches which are a syntactic sugar for the first one (note that the result is not a *pjrpc.common.BatchResponse* object anymore but a tuple of “plain” method invocation results):

- using chain call notation:

```
result = await 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:

```
result = await client.batch[
    ('sum', 2, 2),
    ('sub', 2, 2),
    ('div', 2, 2),
    ('mult', 2, 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:

```
result = await 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.batch.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)
```

(continues on next page)

(continued from previous page)

```
else:
    print(response.error)
```

Batch notifications:

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

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

client.batch.notify('tick').notify('tack').notify('tick').notify('tack').call()
```

2.2.3 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()

@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}

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

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

2.2.4 Parameter validation

Very often besides dumb method parameters validation it is necessary to implement more “deep” validation and provide comprehensive errors description to clients. Fortunately `pjrpc` has builtin parameter validation based on `pydantic` library which uses python type annotation for validation. Look at the following example: all you need to annotate method parameters (or describe more complex types beforehand if necessary). `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

        return super().default(o)

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

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

```

2.2.5 Error handling

pjrpc implements all the errors listed in [protocol specification](#) which can be found in `pjrpc.common.exceptions` module so 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)
```

Default error list can be easily extended. All you need to create an error class inherited from `pjrpc.common.exceptions.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)
```

On the server side everything is also 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
```

(continues on next page)

(continued from previous page)

```

    flask.current_app.users[user_id] = user

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

@methods.add
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)

app.users = {}

json_rpc.init_app(app)

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

```

2.3 Client

pjrpc client provides three main method invocation approaches:

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

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

batch_response = client.batch.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),
    ('div', 2, 2),
    ('mult', 2, 2),
]
```

(continues on next page)

(continued from previous page)

```
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.batch.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)
```

Notifications also supported:

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

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

client.batch.notify('tick').notify('tack').notify('tick').notify('tack').call()
```

2.3.2 Id generators

The library request id generator can also be customized. There are four generator types 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()

@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}

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

if __name__ == "__main__":
    web.run_app(jsonrpc_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
```

(continues on next page)

(continued from previous page)

```

        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

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

if __name__ == "__main__":
    web.run_app(jsonrpc_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'] = {}

```

(continues on next page)

(continued from previous page)

```

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')

```

(continues on next page)

(continued from previous page)

```

@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

        return super().default(o)

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

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

```

The library also supports `pjrpc.server.validators.jsonschema` validator. In case you like any other validation library/framework it can be easily integrated in `pjrpc` library.

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 can be easily extended. All you need to create an error class inherited from *pjrpc.common.exceptions.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 also 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):
```

(continues on next page)

(continued from previous page)

```

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)

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'

```

(continues on next page)

(continued from previous page)

```

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

pjrpc can be easily extended without writing a lot of boilerplate code. The following example illustrate an JSON-RPC server implementation based on [http.server](#) standard python library module:

```

import uuid
import http.server
import socketserver

import pjrpc
import pjrpc.server

class JsonRpcHandler(http.server.BaseHTTPRequestHandler):
    def do_POST(self):
        content_type = self.headers.get('Content-Type')
        if content_type != 'application/json':
            self.send_response(http.HTTPStatus.UNSUPPORTED_MEDIA_TYPE)
            return

        try:
            content_length = int(self.headers.get('Content-Length', -1))
            request_text = self.rfile.read(content_length).decode()
        except UnicodeDecodeError:
            self.send_response(http.HTTPStatus.BAD_REQUEST)
            return

        response_text = self.server.dispatcher.dispatch(request_text, context=self)
        if response_text is None:
            self.send_response(http.HTTPStatus.OK)
        else:
            self.send_response(http.HTTPStatus.OK)
            self.send_header("Content-type", "application/json")
            self.end_headers()

            self.wfile.write(response_text.encode())

```

(continues on next page)

(continued from previous page)

```

class JsonRequestServer(http.server.HTTPServer):
    def __init__(self, server_address, RequestHandlerClass=JsonRpcHandler, bind_and_
    ↪activate=True, **kwargs):
        super().__init__(server_address, RequestHandlerClass, bind_and_activate)
        self._dispatcher = pjrpc.server.Dispatcher(**kwargs)

    @property
    def dispatcher(self):
        return self._dispatcher

methods = pjrpc.server.MethodRegistry()

@methods.add(context='request')
def add_user(request: http.server.BaseHTTPRequestHandler, user: dict):
    user_id = uuid.uuid4().hex
    request.server.users[user_id] = user

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

class ThreadingJsonRpcServer(socketserver.ThreadingMixIn, JsonRequestServer):
    users = {}

with ThreadingJsonRpcServer(("localhost", 8080)) as server:
    server.dispatcher.add_methods(methods)

    server.serve_forever()

```

2.8 Testing

2.8.1 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 PjRpcAiohttpMocker
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(

```

(continues on next page)

(continued from previous page)

```

        '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'

    localhost_calls = pjrpc_aiohttp_mockers.calls['http://localhost/api/v1']
    assert localhost_calls[('2.0', 'sum')].call_count == 2
    assert localhost_calls[('2.0', 'sum')].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 PjRpcAiohttpMocker() as mocker:
        mocker.add('http://localhost/api/v1', 'div', result=2)
        result = await client.proxy.div(4, 2)
        assert result == 2

    localhost_calls = mocker.calls['http://localhost/api/v1']
    assert localhost_calls[('2.0', 'div')].mock_calls == [mock.call(4, 2)]

```

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

2.8.2 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

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

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

```

(continues on next page)

(continued from previous page)

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

2.8.3 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)

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
```

2.9 Tracing

pjrpc supports client and server metrics collection. If you familiar with [aiohttp](#) library it won't take a lot of time to comprehend the metrics collection process, because pjrpc inspired by it and uses the same patterns.

2.9.1 client

The following example illustrate opentracing integration. All you need is just inherit a special class `pjrpc.client.Tracer` and implement required methods:

```
import opentracing
from opentracing import tags
from pjrpc.client import tracer
from pjrpc.client.backend import requests as pjrpc_client

class ClientTracer(tracer.Tracer):

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```

    super().__init__()
    self._tracer = opentracing.global_tracer()

    async def on_request_begin(self, trace_context, request):
        span = self._tracer.start_active_span(f'jsonrpc.{request.method}').span
        span.set_tag(tags.COMPONENT, 'pjrpc.client')
        span.set_tag(tags.SPAN_KIND, tags.SPAN_KIND_RPC_CLIENT)

    async def on_request_end(self, trace_context, request, response):
        span = self._tracer.active_span
        span.set_tag(tags.ERROR, response.is_error)
        if response.is_error:
            span.set_tag('jsonrpc.error_code', response.error.code)
            span.set_tag('jsonrpc.error_message', response.error.message)

        span.finish()

    async def on_error(self, trace_context, request, error):
        span = self._tracer.active_span
        span.set_tag(tags.ERROR, True)
        span.finish()

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

result = client.proxy.sum(1, 2)

```

2.9.2 server

On the server side you need to implement simple functions (middlewares) and pass them to the JSON-RPC application. The following example illustrate prometheus metrics collection:

```

import asyncio

import prometheus_client
from aiohttp import web

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

method_latency_hist = prometheus_client.Histogram('method_latency', 'Method latency',
    ↳labelnames=['method'])
method_active_count = prometheus_client.Gauge('method_active_count', 'Method active',
    ↳count', labelnames=['method'])

async def metrics(request):
    return web.Response(body=prometheus_client.generate_latest())

http_app = web.Application()
http_app.add_routes([web.get('/metrics', metrics)])

```

(continues on next page)

(continued from previous page)

```

methods = pjrpc.server.MethodRegistry()

@methods.add(context='context')
async def method(context):
    print("method started")
    await asyncio.sleep(1)
    print("method finished")

async def latency_metric_middleware(request, context, handler):
    with method_latency_hist.labels(method=request.method).time():
        return await handler(request, context)

async def active_count_metric_middleware(request, context, handler):
    with method_active_count.labels(method=request.method).track_inprogress():
        return await handler(request, context)

jsonrpc_app = aiohttp.Application(
    '/api/v1', app=http_app, middlewares=(
        latency_metric_middleware,
        active_count_metric_middleware,
    ),
)
jsonrpc_app.dispatcher.add_methods(methods)

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

```

2.10 Examples

2.10.1 aio_pika client

```

import asyncio

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

async def main():
    client = pjrpc_client.Client('amqp://guest:guest@localhost:5672/v1', 'jsonrpc')
    await client.connect()

    response: pjrpc.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}")

```

(continues on next page)

(continued from previous page)

```

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

    await client.notify('tick')

if __name__ == "__main__":
    asyncio.run(main())

```

2.10.2 aio_pika server

```

import asyncio
import uuid

import aio_pika

import pjrpc
from pjrpc.server.integration import aio_pika as integration

methods = pjrpc.server.MethodRegistry()

@methods.add(context='message')
def add_user(message: aio_pika.IncomingMessage, user: dict):
    user_id = uuid.uuid4().hex

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

executor = integration.Executor('amqp://guest:guest@localhost:5672/v1', queue_name=
    ↪ 'jsonrpc')
executor.dispatcher.add_methods(methods)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()

    loop.run_until_complete(executor.start())
    try:
        loop.run_forever()
    finally:
        loop.run_until_complete(executor.shutdown())

```

2.10.3 aiohttp class-based handler

```

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.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

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

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

```

2.10.4 aiohttp client

```

import asyncio

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

async def main():
    async with pjrpc_client.Client('http://localhost/api/v1') as client:
        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')

asyncio.run(main())

```

2.10.5 aiohttp client batch request

```

import asyncio

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

async def main():
    async with pjrpc_client.Client('http://localhost:8080/api/v1') as client:

        batch_response = await client.batch.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),
            ),
        )
        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}")

        result = await client.batch('sum', 2, 2)('sub', 2, 2)('div', 2, 2)('mult', 2, 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]}")

        result = await client.batch[
            ('sum', 2, 2),
            ('sub', 2, 2),
            ('div', 2, 2),
            ('mult', 2, 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]}")

        result = await 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]}")

        await client.batch.notify('tick').notify('tack').call()

asyncio.run(main())

```


2.10.6 aiohttp pytest integration

```
import pytest
from unittest import mock

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

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

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

    pjrpc_aiohttp_mocker.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'

    localhost_calls = pjrpc_aiohttp_mocker.calls['http://localhost/api/v1']
    assert localhost_calls[('2.0', 'sum')].call_count == 2
    assert localhost_calls[('2.0', 'sum')].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 PjRpcAiohttpMocker() as mocker:
        mocker.add('http://localhost/api/v1', 'div', result=2)
        result = await client.proxy.div(4, 2)
        assert result == 2

        localhost_calls = mocker.calls['http://localhost/api/v1']
        assert localhost_calls[('2.0', 'div')].mock_calls == [mock.call(4, 2)]
```

2.10.7 aiohttp server

```
import uuid

from aiohttp import web

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

(continues on next page)

(continued from previous page)

```

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}

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

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

```

2.10.8 aiohttp versioning

```

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_v1(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_v2(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)

```

(continues on next page)

(continued from previous page)

```

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)

```

2.10.9 client prometheus metrics

```

import time

import prometheus_client as prom_cli
from pjrpc.client import tracer
from pjrpc.client.backend import requests as pjrpc_client

method_latency_hist = prom_cli.Histogram('method_latency', 'Method latency',
    ↳labelnames=['method'])
method_call_total = prom_cli.Counter('method_call_total', 'Method call count',
    ↳labelnames=['method'])
method_errors_total = prom_cli.Counter('method_errors_total', 'Method errors count',
    ↳labelnames=['method', 'code'])

class PrometheusTracer(tracer.Tracer):
    def on_request_begin(self, trace_context, request):
        trace_context.started_at = time.time()
        method_call_total.labels(request.method).inc()

    def on_request_end(self, trace_context, request, response):
        method_latency_hist.labels(request.method).observe(time.time() - trace_
    ↳context.started_at)
        if response.is_error:
            method_call_total.labels(request.method, response.error.code).inc()

    def on_error(self, trace_context, request, error):
        method_latency_hist.labels(request.method).observe(time.time() - trace_
    ↳context.started_at)

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

result = client.proxy.sum(1, 2)

```

2.10.10 client tracing

```

import opentracing
from opentracing import tags

```

(continues on next page)

(continued from previous page)

```

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

class ClientTracer(tracer.Tracer):

    def __init__(self):
        super().__init__()
        self._tracer = opentracing.global_tracer()

    def on_request_begin(self, trace_context, request):
        span = self._tracer.start_active_span(f'jsonrpc.{request.method}').span
        span.set_tag(tags.COMPONENT, 'pjrpc.client')
        span.set_tag(tags.SPAN_KIND, tags.SPAN_KIND_RPC_CLIENT)

    def on_request_end(self, trace_context, request, response):
        span = self._tracer.active_span
        span.set_tag(tags.ERROR, response.is_error)
        if response.is_error:
            span.set_tag('jsonrpc.error_code', response.error.code)
            span.set_tag('jsonrpc.error_message', response.error.message)

        span.finish()

    def on_error(self, trace_context, request, error):
        span = self._tracer.active_span
        span.set_tag(tags.ERROR, True)
        span.finish()

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

result = client.proxy.sum(1, 2)

```

2.10.11 flask class-based handler

```

import uuid

import flask

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

app = flask.Flask(__name__)

methods = pjrpc.server.MethodRegistry()

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

```

(continues on next page)

(continued from previous page)

```

def __init__(self):
    super().__init__()

    self._users = flask.current_app.users

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

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

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

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

app.users = {}

json_rpc.init_app(app)

if __name__ == "__main__":
    app.run(port=8080)

```

2.10.12 flask server

```

import uuid

import flask

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

app = flask.Flask(__name__)

methods = pjrpc.server.MethodRegistry()

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

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

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

app.users = {}

```

(continues on next page)

(continued from previous page)

```
json_rpc.init_app(app)

if __name__ == "__main__":
    app.run(port=8080)
```

2.10.13 flask versioning

```
import uuid

import flask

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

methods_v1 = pjrpc.server.MethodRegistry()

@methods_v1.add
def add_user_v1(user: dict):
    user_id = uuid.uuid4().hex
    flask.current_app.users[user_id] = user

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

methods_v2 = pjrpc.server.MethodRegistry()

@methods_v2.add
def add_user_v2(user: dict):
    user_id = uuid.uuid4().hex
    flask.current_app.users[user_id] = user

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

app_v1 = flask.blueprints.Blueprint('v1', __name__)

json_rpc = integration.JsonRPC('/api/v1')
json_rpc.dispatcher.add_methods(methods_v1)
json_rpc.init_app(app_v1)

app_v2 = flask.blueprints.Blueprint('v2', __name__)

json_rpc = integration.JsonRPC('/api/v2')
json_rpc.dispatcher.add_methods(methods_v2)
json_rpc.init_app(app_v2)

app = flask.Flask(__name__)
app.register_blueprint(app_v1)
app.register_blueprint(app_v2)
```

(continues on next page)

(continued from previous page)

```

app.users = {}

if __name__ == "__main__":
    app.run(port=8080)

```

2.10.14 httpserver

```

import uuid
import http.server
import socketserver

import pjrpc
import pjrpc.server

class JsonRpcHandler(http.server.BaseHTTPRequestHandler):
    """
    JSON-RPC handler.
    """

    def do_POST(self):
        """
        Handles JSON-RPC request.
        """

        content_type = self.headers.get('Content-Type')
        if content_type != 'application/json':
            self.send_response(http.HTTPStatus.UNSUPPORTED_MEDIA_TYPE)
            return

        try:
            content_length = int(self.headers.get('Content-Length', -1))
            request_text = self.rfile.read(content_length).decode()
        except UnicodeDecodeError:
            self.send_response(http.HTTPStatus.BAD_REQUEST)
            return

        response_text = self.server.dispatcher.dispatch(request_text, context=self)
        if response_text is None:
            self.send_response(http.HTTPStatus.OK)
        else:
            self.send_response(http.HTTPStatus.OK)
            self.send_header("Content-type", "application/json")
            self.end_headers()

            self.wfile.write(response_text.encode())

class JsonRpcServer(http.server.HTTPServer):
    """
    :py:class: `http.server.HTTPServer` based JSON-RPC server.

    :param path: JSON-RPC handler base path
    :param kwargs: arguments to be passed to the dispatcher :py:class: `pjrpc.server.
    ↳ Dispatcher`

```

(continues on next page)

(continued from previous page)

```

"""

def __init__(self, server_address, RequestHandlerClass=JsonRpcHandler, bind_and_
→activate=True, **kwargs):
    super().__init__(server_address, RequestHandlerClass, bind_and_activate)
    self._dispatcher = pjrpc.server.Dispatcher(**kwargs)

@property
def dispatcher(self):
    """
    JSON-RPC method dispatcher.
    """

    return self._dispatcher

methods = pjrpc.server.MethodRegistry()

@methods.add(context='request')
def add_user(request: http.server.BaseHTTPRequestHandler, user: dict):
    user_id = uuid.uuid4().hex
    request.server.users[user_id] = user

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

class ThreadingJsonRpcServer(socketserver.ThreadingMixIn, JsonRpcServer):
    users = {}

with ThreadingJsonRpcServer(("localhost", 8080)) as server:
    server.dispatcher.add_methods(methods)

    server.serve_forever()

```

2.10.15 jsonschema validator

```

import uuid

from aiohttp import web

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

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

contact_schema = {
    'type': 'object',
    'properties': {
        'type': {
            'type': 'string',

```

(continues on next page)

(continued from previous page)

```

        'enum': ['phone', 'email'],
    },
    'value': {'type': 'string'},
},
'required': ['type', 'value'],
}

user_schema = {
    'type': 'object',
    'properties': {
        'name': {'type': 'string'},
        'surname': {'type': 'string'},
        'age': {'type': 'integer'},
        'contacts': {
            'type': 'array',
            'items': contact_schema,
        },
    },
    'required': ['name', 'surname', 'age', 'contacts'],
}

params_schema = {
    'type': 'object',
    'properties': {
        'user': user_schema,
    },
    'required': ['user'],
}

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

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

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

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

```

2.10.16 kombu client

```

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

client = pjrpc_client.Client('amqp://guest:guest@localhost:5672/v1', 'jsonrpc')

```

(continues on next page)

(continued from previous page)

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

2.10.17 kombu server

```
import uuid

import kombu

import pjrpc
from pjrpc.server.integration import kombu as integration

methods = pjrpc.server.MethodRegistry()

@methods.add(context='message')
def add_user(message: kombu.Message, user: dict):
    user_id = uuid.uuid4().hex

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

executor = integration.Executor('amqp://guest:guest@localhost:5672/v1', queue_name=
    ↪ 'jsonrpc')
executor.dispatcher.add_methods(methods)

if __name__ == "__main__":
    executor.run()
```

2.10.18 middlewares

```
from aiohttp import web

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

methods = pjrpc.server.MethodRegistry()

@methods.add(context='request')
async def method(request):
    print("method")
```

(continues on next page)

(continued from previous page)

```

async def middleware1(request, context, handler):
    print("middleware1 started")
    result = await handler(request, context)
    print("middleware1 finished")

    return result

async def middleware2(request, context, handler):
    print("middleware2 started")
    result = await handler(request, context)
    print("middleware2 finished")

    return result

jsonrpc_app = aiohttp.Application(
    '/api/v1', middlewares=(
        middleware1,
        middleware2,
    ),
)
jsonrpc_app.dispatcher.add_methods(methods)

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

```

2.10.19 multiple clients

```

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'

```

(continues on next page)

(continued from previous page)

```

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)

```

2.10.20 pydantic validator

```

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()}

```

(continues on next page)

(continued from previous page)

```

class JSONEncoder(pjrpc.server.JSONEncoder):

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

        return super().default(o)

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

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

```

2.10.21 requests client

```

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], id=1))
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')

```

2.10.22 requests pytest

```

import pytest
from unittest import mock

import pjrpc
from pjrpc.client.integrations.pytest import PjRpcRequestsMocker
from pjrpc.client.backend import requests as requests_client

def test_using_fixture(pjrpc_requests_mock):
    client = requests_client.Client('http://localhost/api/v1')

```

(continues on next page)

(continued from previous page)

```

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

pjrpc_requests_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:
    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'

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

client = requests_client.Client('http://localhost/api/v2')
with pytest.raises(ConnectionRefusedError):
    client.proxy.sum(1, 1)

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

    with PjRpcRequestsMocker() as mocker:
        mocker.add('http://localhost/api/v1', 'mult', result=4)
        mocker.add('http://localhost/api/v1', 'div', callback=lambda a, b: a/b)

        result = client.batch.proxy.div(4, 2).mult(2, 2).call()
        assert result == (2, 4)

        localhost_calls = mocker.calls['http://localhost/api/v1']
        assert localhost_calls[('2.0', 'div')].mock_calls == [mock.call(4, 2)]
        assert localhost_calls[('2.0', 'mult')].mock_calls == [mock.call(2, 2)]

        with pytest.raises(pjrpc.exc.MethodNotFoundError):
            client.proxy.sub(4, 2)

```

2.10.23 sentry

```

import sentry_sdk
from aiohttp import web

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

methods = pjrpc.server.MethodRegistry()

@methods.add(context='request')

```

(continues on next page)

(continued from previous page)

```

async def method(request):
    print("method")

async def sentry_middleware(request, context, handler):
    try:
        return await handler(request, context)
    except pjrpc.exceptions.JsonRpcError as e:
        sentry_sdk.capture_exception(e)
        raise

jsonrpc_app = aiohttp.Application(
    '/api/v1', middlewares=(
        sentry_middleware,
    ),
)
jsonrpc_app.dispatcher.add_methods(methods)

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

```

2.10.24 server prometheus metrics

```

import asyncio

import prometheus_client
from aiohttp import web

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

method_latency_hist = prometheus_client.Histogram('method_latency', 'Method latency',
    ↳labelnames=['method'])
method_active_count = prometheus_client.Gauge('method_active_count', 'Method active',
    ↳count', labelnames=['method'])

async def metrics(request):
    return web.Response(body=prometheus_client.generate_latest())

http_app = web.Application()
http_app.add_routes([web.get('/metrics', metrics)])

methods = pjrpc.server.MethodRegistry()

@methods.add(context='context')
async def method(context):
    print("method started")
    await asyncio.sleep(1)
    print("method finished")

```

(continues on next page)

(continued from previous page)

```

async def latency_metric_middleware(request, context, handler):
    with method_latency_hist.labels(method=request.method).time():
        return await handler(request, context)

async def active_count_metric_middleware(request, context, handler):
    with method_active_count.labels(method=request.method).track_inprogress():
        return await handler(request, context)

jsonrpc_app = aiohttp.Application(
    '/api/v1', app=http_app, middlewares=(
        latency_metric_middleware,
        active_count_metric_middleware,
    ),
)
jsonrpc_app.dispatcher.add_methods(methods)

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

```

2.10.25 server tracing

```

import asyncio

import opentracing
from opentracing import tags
from aiohttp import web

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

@web.middleware
async def http_tracing_middleware(request, handler):
    """
    aiohttp server tracer.
    """

    tracer = opentracing.global_tracer()
    try:
        span_ctx = tracer.extract(format=opentracing.Format.HTTP_HEADERS,
    ↪ carrier=request.headers)
    except (opentracing.InvalidCarrierException, opentracing.
    ↪ SpanContextCorruptedException):
        span_ctx = None

    span = tracer.start_span(f'http.{request.method}', child_of=span_ctx)
    span.set_tag(tags.COMPONENT, 'aiohttp.server')
    span.set_tag(tags.SPAN_KIND, tags.SPAN_KIND_RPC_SERVER)
    span.set_tag(tags.PEER_ADDRESS, request.remote)
    span.set_tag(tags.HTTP_URL, str(request.url))
    span.set_tag(tags.HTTP_METHOD, request.method)

    with tracer.scope_manager.activate(span, finish_on_close=True):
        response: web.Response = await handler(request)

```

(continues on next page)

(continued from previous page)

```

        span.set_tag(tags.HTTP_STATUS_CODE, response.status)
        span.set_tag(tags.ERROR, response.status >= 400)

    return response

http_app = web.Application(
    middlewares=(
        http_tracing_middleware,
    ),
)

methods = pjrpc.server.MethodRegistry()

@methods.add(context='context')
async def method(context):
    print("method started")
    await asyncio.sleep(1)
    print("method finished")

async def jsonrpc_tracing_middleware(request, context, handler):
    tracer = opentracing.global_tracer()
    span = tracer.start_span(f'jsonrpc.{request.method}')

    span.set_tag(tags.COMPONENT, 'pjrpc')
    span.set_tag(tags.SPAN_KIND, tags.SPAN_KIND_RPC_SERVER)
    span.set_tag('jsonrpc.version', request.version)
    span.set_tag('jsonrpc.id', request.id)
    span.set_tag('jsonrpc.method', request.method)

    with tracer.scope_manager.activate(span, finish_on_close=True):
        response = await handler(request, context)
        if response.is_error:
            span.set_tag('jsonrpc.error_code', response.error.code)
            span.set_tag('jsonrpc.error_message', response.error.message)
            span.set_tag(tags.ERROR, True)
        else:
            span.set_tag(tags.ERROR, False)

    return response

jsonrpc_app = aiohttp.Application(
    '/api/v1', app=http_app, middlewares=(
        jsonrpc_tracing_middleware,
    ),
)

jsonrpc_app.dispatcher.add_methods(methods)

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

```

2.10.26 werkzeug server

```
import uuid

import werkzeug

import pjrpc.server
from pjrpc.server.integration import werkzeug as integration

methods = pjrpc.server.MethodRegistry()

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

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

app = integration.JsonRPC('/api/v1')
app.dispatcher.add_methods(methods)
app.users = {}

if __name__ == '__main__':
    werkzeug.serving.run_simple('127.0.0.1', 8080, app)
```

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 `pjsonrpc.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

Raises `pjsonrpc.common.exception.DeserializationError` if format is incorrect

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

Raises `pjrpc.common.exception.DeserializationError` if format is incorrect

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* field.

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

is_notification

Returns `True` if all the request in the batch are notifications.

to_json ()

Serializes the request to json data.

Returns json data

class `pjrpc.common.BatchResponse` (**responses, error=UNSET, 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.

error

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

extend (*responses*)

Extends the batch with the *responses*.

classmethod from_json (*json_data, error_cls=<class 'pjrpc.common.exceptions.JsonRpcError'>*)

Deserializes a batch response from json data.

Parameters

- **json_data** – data the response to be deserialized from
- **error_cls** – error class

Returns batch response object

has_error

Returns `True` if any response has an error.

is_error

Returns `True` if the request has not succeeded. Note that it is not the same as `pjrpc.common.BatchResponse.has_error`. `is_error` indicates that the batch request failed at all, while `has_error` indicates that one of the requests in the batch failed.

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

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 a 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 or missing.

```
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 JSON-RPC 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 `pjrpc.common.exception.DeserializationError` 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'>, error_cls=<class
                                'pjrpc.common.exceptions.JsonRpcError'>,
                                id_gen=<function sequential>, json_loader=<function
                                loads>, json_dumper=<function
                                dumps>, json_encoder=<class
                                'pjrpc.common.common.JSONEncoder'>,
                                json_decoder=None, strict=True, request_args=None,
                                tracers=())
```

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, _trace_ctx=namespace(), **kwargs)
```

Makes JSON-RPC call.

Parameters

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments
- **_trace_ctx** – tracers request context

Returns response result

```
notify (method, *args, _trace_ctx=namespace(), **kwargs)
```

Makes a notification request

Parameters

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments
- **_trace_ctx** – tracers request context

proxy

Clint proxy object.

send (*request*, *_trace_ctx=namespace()*, ***kwargs*)
Sends a JSON-RPC request.

Parameters

- **request** – request instance
- **kwargs** – additional client request argument
- **_trace_ctx** – tracers request context

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'>,
                                     error_cls=<class 'pjrpc.common.exceptions.JsonRpcError'>,
                                     id_gen=<function sequential>,
                                     json_loader=<function loads>,
                                     json_dumper=<function dumps>,
                                     json_encoder=<class 'pjrpc.common.common.JSONEncoder'>,
                                     json_decoder=None,
                                     strict=True,
                                     request_args=None,
                                     tracers=())
```

Abstract asynchronous JSON-RPC client.

batch

Client batch wrapper.

call (*method*, **args*, *_trace_ctx=namespace()*, ***kwargs*)
Makes JSON-RPC call.

Parameters

- **method** – method name
- **args** – method positional arguments
- **kwargs** – method named arguments
- **_trace_ctx** – tracers request context

Returns response result

send (*request*, *_trace_ctx=namespace()*, ***kwargs*)
Sends a JSON-RPC request.

Parameters

- **request** – request instance
- **kwargs** – additional client request argument
- **_trace_ctx** – tracers request context

Returns response instance

class pjrpc.client.**LoggingTracer** (*logger=<RootLogger root (WARNING)>, level=10*)
JSON-RPC client logging tracer.

on_batch_begin (*trace_context, batch_request*)
Handler called before JSON-RPC batch request begins.

Parameters

- **trace_context** – request trace context
- **batch_request** – JSON-RPC request

on_batch_end (*trace_context, batch_request, batch_response*)
Handler called after JSON-RPC batch request ends.

Parameters

- **trace_context** – request trace context
- **batch_request** – JSON-RPC request
- **batch_response** – JSON-RPC response

on_error (*trace_context, request, error*)
Handler called when JSON-RPC request failed.

Parameters

- **trace_context** – request trace context
- **request** – JSON-RPC request
- **error** – raised exception

on_request_begin (*trace_context, request*)
Handler called before JSON-RPC request begins.

Parameters

- **trace_context** – request trace context
- **request** – JSON-RPC request

on_request_end (*trace_context, request, response*)
Handler called after JSON-RPC request ends.

Parameters

- **trace_context** – request trace context
- **request** – JSON-RPC request
- **response** – JSON-RPC response

class pjrpc.client.**Tracer**
JSON-RPC client tracer.

on_batch_begin (*trace_context, batch_request*)
Handler called before JSON-RPC batch request begins.

Parameters

- **trace_context** – request trace context
- **batch_request** – JSON-RPC request

on_batch_end (*trace_context, batch_request, batch_response*)
 Handler called after JSON-RPC batch request ends.

Parameters

- **trace_context** – request trace context
- **batch_request** – JSON-RPC request
- **batch_response** – JSON-RPC response

on_error (*trace_context, request, error*)
 Handler called when JSON-RPC request failed.

Parameters

- **trace_context** – request trace context
- **request** – JSON-RPC request
- **error** – raised exception

on_request_begin (*trace_context, request*)
 Handler called before JSON-RPC request begins.

Parameters

- **trace_context** – request trace context
- **request** – JSON-RPC request

on_request_end (*trace_context, request, response*)
 Handler called after JSON-RPC request ends.

Parameters

- **trace_context** – request trace context
- **request** – JSON-RPC request
- **response** – JSON-RPC response

Backends

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

Parameters

- **url** – 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.

class `pjrpc.client.backend.aiohttp.Client` (*url, session_args=None, session=None, **kwargs*)
 Aiohttp library client backend.

Parameters

- **url** – url to be used as JSON-RPC endpoint
- **session_args** – additional `aiohttp.ClientSession` arguments
- **session** – custom session to be used instead of `aiohttp.ClientSession`

close()

Closes current http session.

```
class pjrpc.client.backend.kombu.Client (broker_url, queue_name=None, conn_args=None,  
                                         exchange_name=None, exchange_args=None,  
                                         routing_key=None, result_queue_name=None,  
                                         result_queue_args=None, **kwargs)
```

`kombu` based JSON-RPC client. Note: the client is not thread-safe.

Parameters

- **broker_url** – broker connection url
- **conn_args** – broker connection arguments.
- **queue_name** – queue name to publish requests to
- **exchange_name** – exchange to publish requests to. If `None` default exchange is used
- **exchange_args** – exchange arguments
- **routing_key** – reply message routing key. If `None` queue name is used
- **result_queue_name** – result queue name. If `None` random exclusive queue is declared for each request
- **conn_args** – additional connection arguments
- **kwargs** – parameters to be passed to `pjrpc.client.AbstractClient`

close()

Closes the current broker connection.

```
class pjrpc.client.backend.aio_pika.Client (broker_url, queue_name=None,  
                                         conn_args=None, exchange_name=None,  
                                         exchange_args=None, routing_key=None,  
                                         result_queue_name=None, re-  
                                         sult_queue_args=None, **kwargs)
```

`aio_pika` based JSON-RPC client.

Parameters

- **broker_url** – broker connection url
- **conn_args** – broker connection arguments.
- **queue_name** – queue name to publish requests to
- **exchange_name** – exchange to publish requests to. If `None` default exchange is used
- **exchange_args** – exchange arguments
- **routing_key** – reply message routing key. If `None` queue name is used
- **result_queue_name** – result queue name. If `None` random exclusive queue is declared for each request
- **conn_args** – additional connection arguments
- **kwargs** – parameters to be passed to `pjrpc.client.AbstractClient`

close()
Closes current broker connection.

connect()
Opens a connection to the broker.

Integrations

3.1.3 Server

JSON-RPC server package.

```
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,
                                   middlewares=())
```

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.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, middlewares=())
```

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

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.JSONEncoder (*, skipkeys=False, ensure_ascii=True, check_circular=True,
                                allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)
```

Server JSON encoder. All custom server 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)
```

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

JSON-RPC method wrapper. Stores method itself 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

aiohttp JSON-RPC server integration.

class `pjrpc.server.integration.aiohttp.Application` (*path="", app=None, **kwargs*)

`aiohttp` based JSON-RPC server.

Parameters

- **path** – JSON-RPC handler base path
- **app_args** – arguments to be passed to `aiohttp.web.Application`
- **kwargs** – arguments to be passed to the dispatcher `pjrpc.server.AsyncDispatcher`

app

aiohttp application.

dispatcher

JSON-RPC method dispatcher.

rpc_handle (*http_request*)

Handles JSON-RPC request.

Parameters **http_request** – `aiohttp.web.Response`

Returns `aiohttp.web.Request`

flask

Flask JSON-RPC extension.

class `pjrpc.server.integration.flask.JsonRPC` (*path*, ***kwargs*)

Flask framework JSON-RPC extension class.

Parameters

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

dispatcher

JSON-RPC method dispatcher.

init_app (*app*)

Initializes flask application with JSON-RPC extension.

Parameters **app** – flask application instance

kombu

kombu JSON-RPC server integration.

class `pjrpc.server.integration.kombu.Executor` (*broker_url*, *queue_name*,
conn_args=None, *queue_args=None*,
publish_args=None, *prefetch_count=0*,
***kwargs*)

`kombu` based JSON-RPC server.

Parameters

- **broker_url** – broker connection url
- **queue_name** – requests queue name
- **conn_args** – additional connection arguments
- **queue_args** – queue arguments
- **publish_args** – message publish additional arguments
- **prefetch_count** – worker prefetch count
- **kwargs** – dispatcher additional arguments

dispatcher

JSON-RPC method dispatcher.

aio_pika

class `pjrpc.server.integration.aio_pika.Executor` (*broker_url*, *queue_name*,
prefetch_count=0, ***kwargs*)

aio_pika based JSON-RPC server.

Parameters

- **broker_url** – broker connection url
- **queue_name** – requests queue name
- **prefetch_count** – worker prefetch count
- **kwargs** – dispatcher additional arguments

dispatcher

JSON-RPC method dispatcher.

shutdown()

Stops executor.

start(queue_args=None)

Starts executor.

Parameters *queue_args* – queue arguments

werkzeug

class `pjrpc.server.integration.werkzeug.JsonRPC` (*path=""*, ***kwargs*)

werkzeug server JSON-RPC integration.

Parameters

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

dispatcher

JSON-RPC method dispatcher.

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

class `pjrpc.server.validators.jsonschema.JsonSchemaValidator` (***kwargs*)

Parameters validator based on `jsonschema` library.

Parameters **kwargs** – default jsonschema validator arguments

validate_method (*method, params, exclude=(), **kwargs*)

Validates params against method using `pydantic` validator.

Parameters

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

Raises `pjrpc.server.validators.ValidationError`

pydantic

class `pjrpc.server.validators.pydantic.PydanticValidator` (*coerce=True, **con-fig_args*)

Parameters validator based on `pydantic` library. Uses python type annotations for parameters validation.

Parameters **coerce** – if `True` returns converted (coerced) parameters according to parameter type annotation otherwise returns parameters as is

build_validation_schema

Builds pydantic model based validation schema from method signature.

Parameters **signature** – method signature to build schema for

Returns validation schema

validate_method (*method, params, exclude=(), **kwargs*)

Validates params against method using `pydantic` validator.

Parameters

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

Returns coerced parameters if *coerce* flag is `True` otherwise parameters as is

Raises `ValidationError`

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`, [47](#)
- `pjrpc.client`, [51](#)
- `pjrpc.client.backend.aio_pika`, [56](#)
- `pjrpc.client.backend.aihttp`, [55](#)
- `pjrpc.client.backend.kombu`, [56](#)
- `pjrpc.client.backend.requests`, [55](#)
- `pjrpc.common`, [47](#)
- `pjrpc.common.exceptions`, [50](#)
- `pjrpc.common.generators`, [51](#)
- `pjrpc.server`, [57](#)
- `pjrpc.server.integration.aio_pika`, [61](#)
- `pjrpc.server.integration.aihttp`, [59](#)
- `pjrpc.server.integration.flask`, [60](#)
- `pjrpc.server.integration.kombu`, [60](#)
- `pjrpc.server.integration.werkzeug`, [61](#)
- `pjrpc.server.validators`, [61](#)
- `pjrpc.server.validators.jsonschema`, [62](#)
- `pjrpc.server.validators.pydantic`, [62](#)

A

AbstractAsyncClient (class in *pjrpc.client*), 53
 AbstractClient (class in *pjrpc.client*), 51
 AbstractClient.Proxy (class in *pjrpc.client*), 52
 add() (*pjrpc.server.Dispatcher* method), 57
 add() (*pjrpc.server.MethodRegistry* method), 59
 add_methods() (*pjrpc.server.Dispatcher* method), 58
 add_methods() (*pjrpc.server.MethodRegistry* method), 59
 app (*pjrpc.server.integration.aiohttp.Application* attribute), 59
 append() (*pjrpc.common.BatchRequest* method), 48
 append() (*pjrpc.common.BatchResponse* method), 49
 Application (class in *pjrpc.server.integration.aiohttp*), 59
 AsyncDispatcher (class in *pjrpc.server*), 57

B

BaseError, 50
 BaseValidator (class in *pjrpc.server.validators*), 61
 batch (*pjrpc.client.AbstractAsyncClient* attribute), 53
 batch (*pjrpc.client.AbstractClient* attribute), 52
 BatchRequest (class in *pjrpc.common*), 48
 BatchResponse (class in *pjrpc.common*), 49
 bind() (*pjrpc.server.validators.BaseValidator* method), 61
 build_validation_schema
 (*pjrpc.server.validators.pydantic.PydanticValidator*
 attribute), 63

C

call() (*pjrpc.client.AbstractAsyncClient* method), 53
 call() (*pjrpc.client.AbstractClient* method), 52
 Client (class in *pjrpc.client.backend.aio_pika*), 56
 Client (class in *pjrpc.client.backend.aiohttp*), 55
 Client (class in *pjrpc.client.backend.kombu*), 56
 Client (class in *pjrpc.client.backend.requests*), 55
 ClientError, 50

close() (*pjrpc.client.backend.aio_pika.Client*
 method), 56
 close() (*pjrpc.client.backend.aiohttp.Client* method),
 56
 close() (*pjrpc.client.backend.kombu.Client* method),
 56
 close() (*pjrpc.client.backend.requests.Client* method),
 55
 connect() (*pjrpc.client.backend.aio_pika.Client*
 method), 57

D

default() (*pjrpc.common.JSONEncoder* method), 50
 default() (*pjrpc.server.JSONEncoder* method), 58
 DeserializationError, 50
 dispatch() (*pjrpc.server.AsyncDispatcher* method),
 57
 dispatch() (*pjrpc.server.Dispatcher* method), 58
 Dispatcher (class in *pjrpc.server*), 57
 dispatcher (*pjrpc.server.integration.aio_pika.Executor*
 attribute), 61
 dispatcher (*pjrpc.server.integration.aiohttp.Application*
 attribute), 60
 dispatcher (*pjrpc.server.integration.flask.JsonRPC*
 attribute), 60
 dispatcher (*pjrpc.server.integration.kombu.Executor*
 attribute), 60
 dispatcher (*pjrpc.server.integration.werkzeug.JsonRPC*
 attribute), 61

E

error (*pjrpc.common.BatchResponse* attribute), 49
 error (*pjrpc.common.Response* attribute), 48
 Executor (class in *pjrpc.server.integration.aio_pika*),
 61
 Executor (class in *pjrpc.server.integration.kombu*), 60
 extend() (*pjrpc.common.BatchRequest* method), 49
 extend() (*pjrpc.common.BatchResponse* method), 49

F

`from_json()` (*pjrpc.common.BatchRequest class method*), 49
`from_json()` (*pjrpc.common.BatchResponse class method*), 49
`from_json()` (*pjrpc.common.exceptions.JsonRpcError class method*), 51
`from_json()` (*pjrpc.common.Request class method*), 47
`from_json()` (*pjrpc.common.Response class method*), 48

G

`get()` (*pjrpc.server.MethodRegistry method*), 59

H

`has_error` (*pjrpc.common.BatchResponse attribute*), 49

I

`id` (*pjrpc.common.Request attribute*), 47
`id` (*pjrpc.common.Response attribute*), 48
`IdentityError`, 50
`init_app()` (*pjrpc.server.integration.flask.JsonRPC method*), 60
`InternalError`, 50
`InvalidParamsError`, 50
`InvalidRequestError`, 50
`is_error` (*pjrpc.common.BatchResponse attribute*), 49
`is_error` (*pjrpc.common.Response attribute*), 48
`is_notification` (*pjrpc.common.BatchRequest attribute*), 49
`is_notification` (*pjrpc.common.Request attribute*), 47
`is_success` (*pjrpc.common.BatchResponse attribute*), 49
`is_success` (*pjrpc.common.Response attribute*), 48

J

`JSONEncoder` (*class in pjrpc.common*), 50
`JSONEncoder` (*class in pjrpc.server*), 58
`JsonRPC` (*class in pjrpc.server.integration.flask*), 60
`JsonRPC` (*class in pjrpc.server.integration.werkzeug*), 61
`JsonRpcError`, 50
`JsonRpcErrorMeta` (*class in pjrpc.common.exceptions*), 51
`JsonSchemaValidator` (*class in pjrpc.server.validators.jsonschema*), 62

L

`LoggingTracer` (*class in pjrpc.client*), 54

M

`merge()` (*pjrpc.server.MethodRegistry method*), 59
`Method` (*class in pjrpc.server*), 58
`method` (*pjrpc.common.Request attribute*), 47
`MethodNotFoundError`, 51
`MethodRegistry` (*class in pjrpc.server*), 58

N

`notify()` (*pjrpc.client.AbstractClient method*), 52

O

`on_batch_begin()` (*pjrpc.client.LoggingTracer method*), 54
`on_batch_begin()` (*pjrpc.client.Tracer method*), 54
`on_batch_end()` (*pjrpc.client.LoggingTracer method*), 54
`on_batch_end()` (*pjrpc.client.Tracer method*), 55
`on_error()` (*pjrpc.client.LoggingTracer method*), 54
`on_error()` (*pjrpc.client.Tracer method*), 55
`on_request_begin()` (*pjrpc.client.LoggingTracer method*), 54
`on_request_begin()` (*pjrpc.client.Tracer method*), 55
`on_request_end()` (*pjrpc.client.LoggingTracer method*), 54
`on_request_end()` (*pjrpc.client.Tracer method*), 55

P

`params` (*pjrpc.common.Request attribute*), 48
`ParseError`, 51
`pjrpc` (*module*), 47
`pjrpc.client` (*module*), 51
`pjrpc.client.backend.aio_pika` (*module*), 56
`pjrpc.client.backend.aiohttp` (*module*), 55
`pjrpc.client.backend.kombu` (*module*), 56
`pjrpc.client.backend.requests` (*module*), 55
`pjrpc.common` (*module*), 47
`pjrpc.common.exceptions` (*module*), 50
`pjrpc.common.generators` (*module*), 51
`pjrpc.server` (*module*), 57
`pjrpc.server.integration.aio_pika` (*module*), 61
`pjrpc.server.integration.aiohttp` (*module*), 59
`pjrpc.server.integration.flask` (*module*), 60
`pjrpc.server.integration.kombu` (*module*), 60
`pjrpc.server.integration.werkzeug` (*module*), 61
`pjrpc.server.validators` (*module*), 61
`pjrpc.server.validators.jsonschema` (*module*), 62

pjrpc.server.validators.pydantic (module), 62
 proxy (pjrpc.client.AbstractClient attribute), 53
 PydanticValidator (class in pjrpc.server.validators.pydantic), 62
 validate_method() (pjrpc.server.validators.pydantic.PydanticValidator method), 63
 ValidationError, 62
 View (class in pjrpc.server), 59
 view() (pjrpc.server.Dispatcher method), 58
 view() (pjrpc.server.MethodRegistry method), 59

R

randint() (in module pjrpc.common.generators), 51
 random() (in module pjrpc.common.generators), 51
 related (pjrpc.common.BatchResponse attribute), 49
 related (pjrpc.common.Response attribute), 48
 Request (class in pjrpc.common), 47
 Response (class in pjrpc.common), 48
 result (pjrpc.common.BatchResponse attribute), 49
 result (pjrpc.common.Response attribute), 48
 rpc_handle() (pjrpc.server.integration.aiohttp.Application method), 60

S

send() (pjrpc.client.AbstractAsyncClient method), 53
 send() (pjrpc.client.AbstractClient method), 53
 sequential() (in module pjrpc.common.generators), 51
 ServerError, 51
 shutdown() (pjrpc.server.integration.aio_pika.Executor method), 61
 signature (pjrpc.server.validators.BaseValidator attribute), 61
 start() (pjrpc.server.integration.aio_pika.Executor method), 61

T

to_json() (pjrpc.common.BatchRequest method), 49
 to_json() (pjrpc.common.BatchResponse method), 50
 to_json() (pjrpc.common.exceptions.JsonRpcError method), 51
 to_json() (pjrpc.common.Request method), 48
 to_json() (pjrpc.common.Response method), 48
 Tracer (class in pjrpc.client), 54

U

uuid() (in module pjrpc.common.generators), 51

V

validate() (pjrpc.server.validators.BaseValidator method), 62
 validate_method() (pjrpc.server.validators.BaseValidator method), 62
 validate_method() (pjrpc.server.validators.jsonschema.JsonSchemaValidator method), 62