pjrpc

Release 1.3.4

Sep 10, 2021

Contents

`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:

- *framework/library agnostic*
- *intuitive interface*
- *extensibility*
- *synchronous and asynchronous client backends*
- *popular frameworks integration* (aiohttp, flask, kombu, aio_pika)
- *builtin parameter validation*
- *pytest integration*
- *openapi schema generation support*
- *openrpc schema generation support*
- *web ui support (SwaggerUI, RapiDoc, ReDoc)*

CHAPTER 1

Extra requirements

- aiohttp
- aio_pika
- flask
- jsonschema
- kombu
- pydantic
- requests
- httpx
- openapi-ui-bundles
- starlette
- django

CHAPTER 2

The User Guide

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

`pjrpc` client interface 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.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)
```

2.2.5 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)
```

2.2.6 OpenAPI specification

`pjrpc` has built-in OpenAPI and OpenRPC specification generation support and integrated web UI as an extra dependency. Three UI types are supported:

- SwaggerUI (https://swagger.io/tools/swagger-ui/)
- RapiDoc (https://mrin9.github.io/RapiDoc/)
- ReDoc (https://github.com/Redocly/redoc)

Web UI extra dependency can be installed using the following code:

```
$ pip install pjrpc[openapi-ui-bundles]
```

The following example illustrates how to configure OpenAPI specification generation and Swagger UI web tool with basic auth:

```
import uuid
from typing import Any, Optional

import flask
import flask_httpauth
import pydantic
import flask_cors
from werkzeug import security

import pjrpc.server.specs.extractors.pydantic
from pjrpc.server.integration import flask as integration
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.specs import extractors, openapi as specs
```

(continues on next page)

(continued from previous page)

```
app = flask.Flask('myapp')
flask_cors.CORS(app, resources={"/myapp/api/v1/*": {"origins": "*"}})

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

auth = flask_httpauth.HTTPBasicAuth()
credentials = {"admin": security.generate_password_hash("admin")}


@auth.verify_password
def verify_password(username: str, password: str) -> Optional[str]:
    if username in credentials and security.check_password_hash(credentials.
↪get(username), password):
        return username


class AuthenticatedJsonRPC(integration.JsonRPC):
    @auth.login_required
    def _rpc_handle(self, dispatcher: pjrpc.server.Dispatcher) -> flask.Response:
        return super()._rpc_handle(dispatcher)


class JSONEncoder(pjrpc.JSONEncoder):
    def default(self, o: Any) -> Any:
        if isinstance(o, pydantic.BaseModel):
            return o.dict()
        if isinstance(o, uuid.UUID):
            return str(o)

        return super().default(o)


class UserIn(pydantic.BaseModel):
    """
    User registration data.
    """

    name: str
    surname: str
    age: int


class UserOut(UserIn):
    """
    Registered user data.
    """

    id: uuid.UUID


class AlreadyExistsError(pjrpc.exc.JsonRpcError):
    """
    User already registered error.
    """

    code = 2001
```

(continues on next page)

(continued from previous page)

```
    message = "user already exists"


class NotFoundError(pjrpc.exc.JsonRpcError):
    """
    User not found error.
    """

    code = 2002
    message = "user not found"


@specs.annotate(
    tags=['users'],
    errors=[AlreadyExistsError],
    examples=[
        specs.MethodExample(
            summary="Simple example",
            params=dict(
                user={
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
            result={
                'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                'name': 'Alex',
                'surname': 'Smith',
                'age': 25,
            },
        ),
    ],
)
@methods.add
@validator.validate
def add_user(user: UserIn) -> UserOut:
    """
    Creates a user.

    :param object user: user data
    :return object: registered user
    :raise AlreadyExistsError: user already exists
    """

    for existing_user in flask.current_app.users_db.values():
        if user.name == existing_user.name:
            raise AlreadyExistsError()

    user_id = uuid.uuid4().hex
    flask.current_app.users_db[user_id] = user

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
```

(continues on next page)

(continued from previous page)

```
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result={
                 'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                 'name': 'Alex',
                 'surname': 'Smith',
                 'age': 25,
            },
        ),
    ],
)
@methods.add
@validator.validate
def get_user(user_id: uuid.UUID) -> UserOut:
    """
    Returns a user.

    :param object user_id: user id
    :return object: registered user
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.get(user_id)
    if not user:
        raise NotFoundError()

    return UserOut(**user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result=None,
        ),
    ],
)
@methods.add
@validator.validate
def delete_user(user_id: uuid.UUID) -> None:
    """
    Deletes a user.

    :param object user_id: user id
    :raise NotFoundError: user not found
    """
```

(continues on next page)

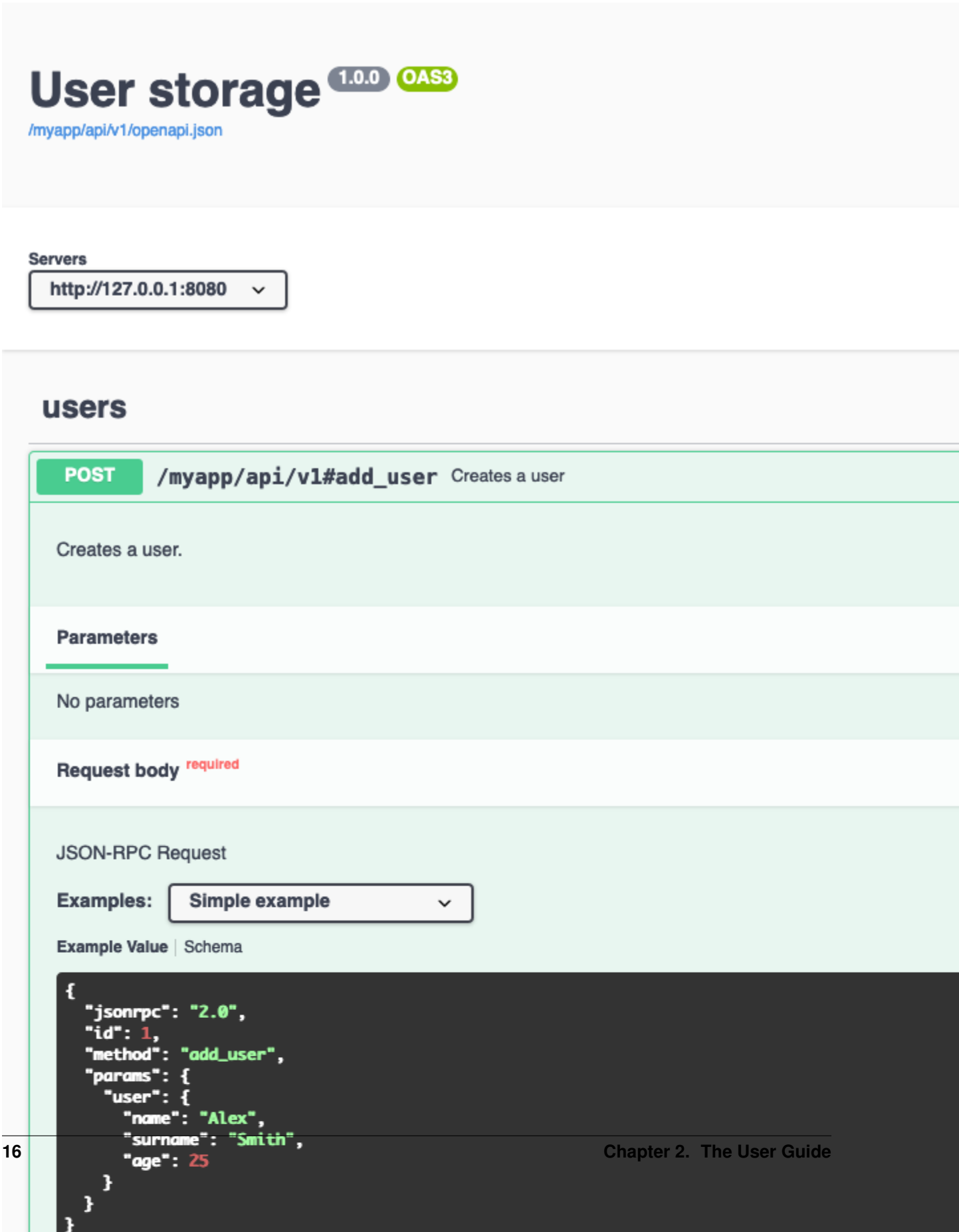
(continued from previous page)

```
    user = flask.current_app.users_db.pop(user_id, None)
    if not user:
        raise NotFoundError()


json_rpc = AuthenticatedJsonRPC(
    '/api/v1',
    json_encoder=JSONEncoder,
    spec=specs.OpenAPI(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                url='http://127.0.0.1:8080',
            ),
        ],
        security_schemes=dict(
            basicAuth=specs.SecurityScheme(
                type=specs.SecuritySchemeType.HTTP,
                scheme='basic',
            ),
        ),
        security=[
            dict(basicAuth=[])
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
        ui=specs.SwaggerUI(),
        # ui=specs.RapiDoc(),
        # ui=specs.ReDoc(),
    ),
)
json_rpc.dispatcher.add_methods(methods)

app.users_db = {}

myapp = flask.Blueprint('myapp', __name__, url_prefix='/myapp')
json_rpc.init_app(myapp)

app.register_blueprint(myapp)

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

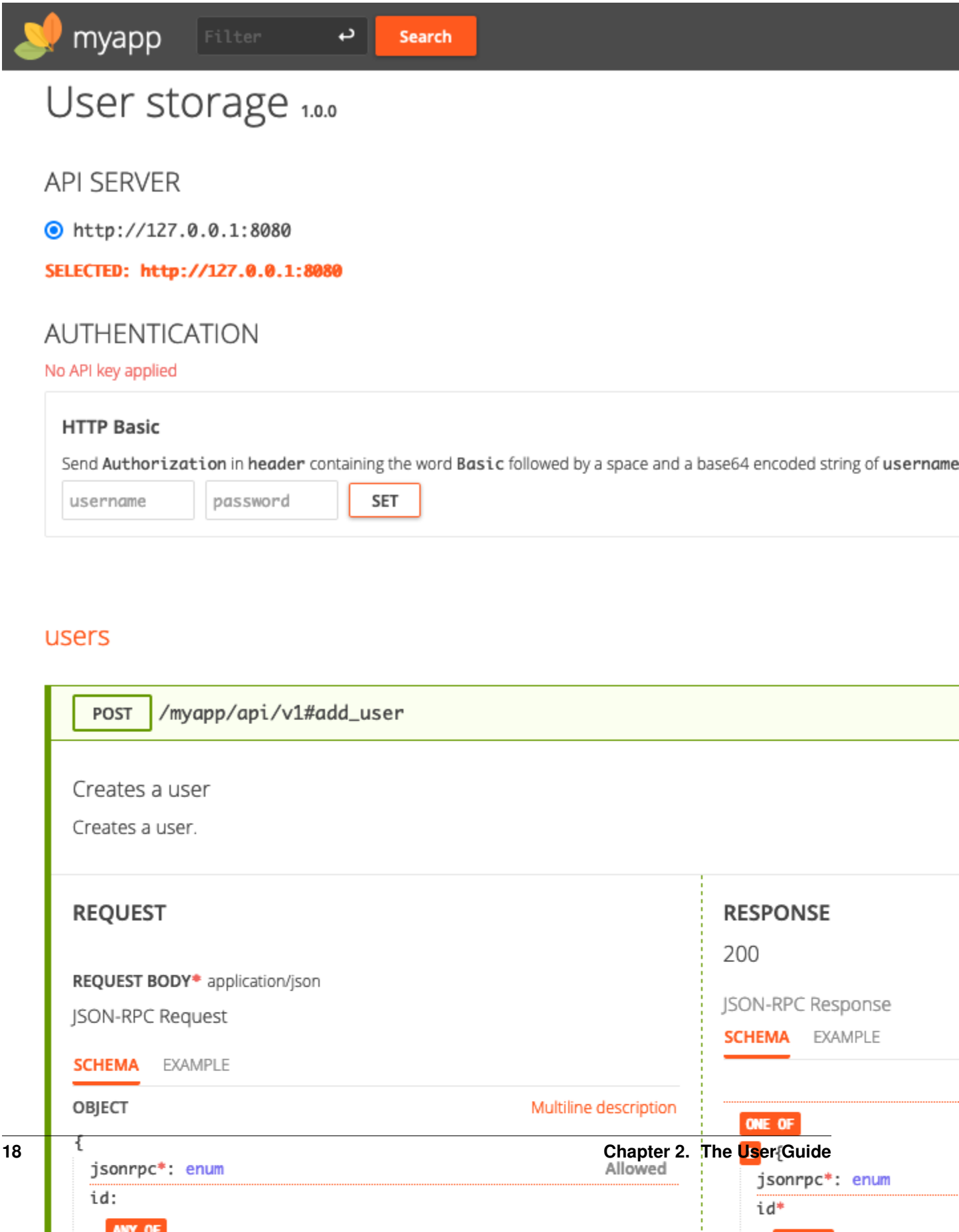
Specification is available on http://localhost:8080/myapp/api/v1/openapi.json

Web UI is running on http://localhost:8080/myapp/api/v1/ui/

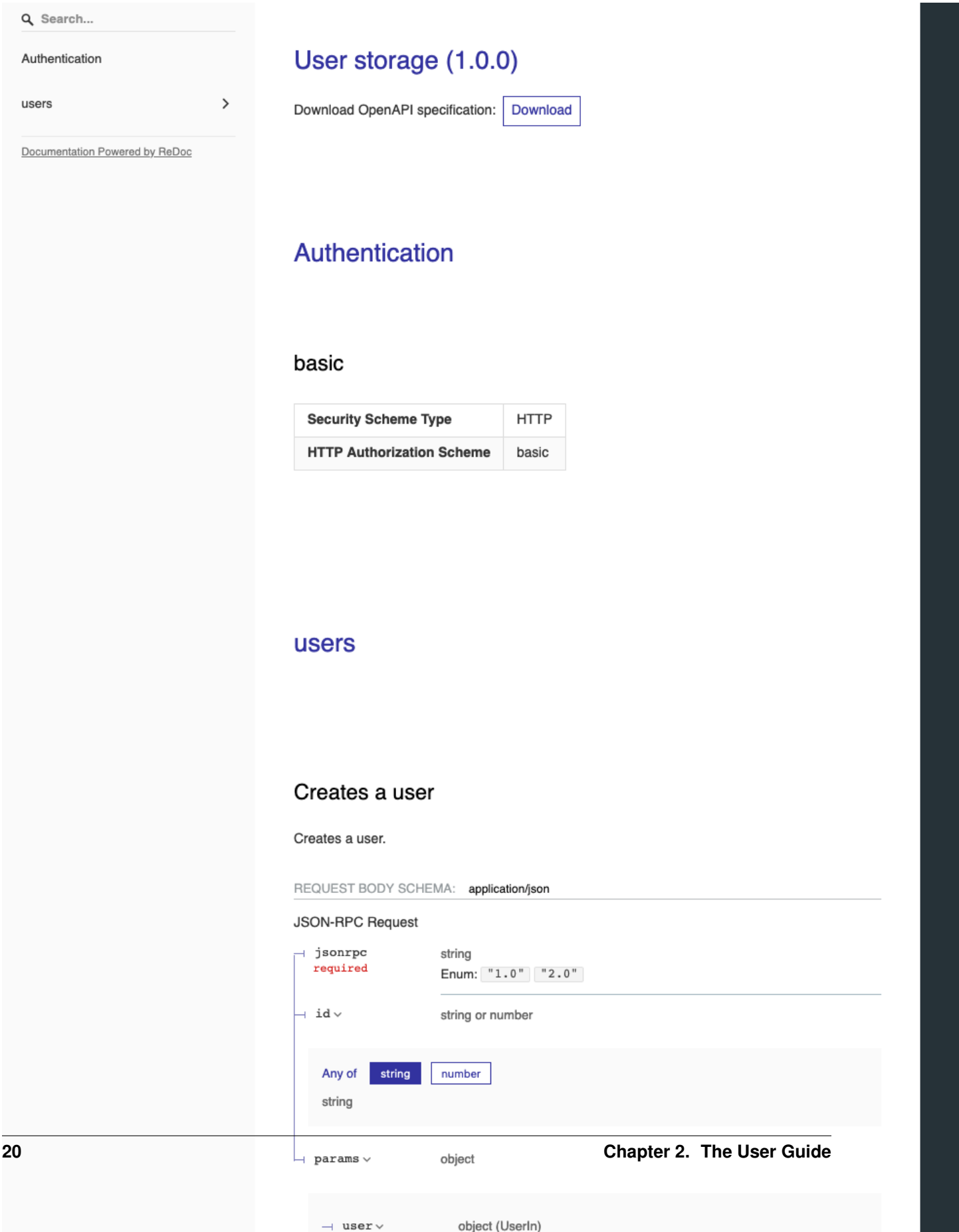
Swagger UI:


User storage
1.0.0
OAS3
/myapp/api/v1/openapi.json
Servers
http://127.0.0.1:8080
users
POST
/myapp/api/v1#add_user
Creates a user
Creates a user.
Parameters
No parameters
Request body
required
JSON-RPC Request
Examples:
Simple example
Example Value
Schema
{
"jsonrpc": "2.0",
"id": 1,
"method": "add_user",
"params": {
"user": {
"name": "Alex",
"surname": "Smith",
"age": 25
}
}
}

RapiDoc:


myapp
Filter
Search
User storage 1.0.0
API SERVER
http://127.0.0.1:8080
SELECTED: http://127.0.0.1:8080
AUTHENTICATION
No API key applied
HTTP Basic
Send Authorization in header containing the word Basic followed by a space and a base64 encoded string of username
username
password
SET
users
POST /myapp/api/v1#add_user
Creates a user
Creates a user.
REQUEST
REQUEST BODY* application/json
JSON-RPC Request
SCHEMA EXAMPLE
OBJECT
Multiline description
{
jsonrpc*: enum
Allowed
id:
RESPONSE
200
JSON-RPC Response
SCHEMA EXAMPLE
ONE OF
jsonrpc*: enum
id*

ReDoc:


Search...
Authentication
users
Documentation Powered by ReDoc
User storage (1.0.0)
Download OpenAPI specification:
Download
Authentication
basic
Security Scheme Type
HTTP
HTTP Authorization Scheme
basic
users
Creates a user
Creates a user.
REQUEST BODY SCHEMA: application/json
JSON-RPC Request
jsonrpc
required
string
Enum: "1.0" "2.0"
id
string or number
Any of
string
number
string
params
object
user
object (UserIn)

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),
]
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')
```

(continues on next page)

(continued from previous page)

```
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}
```

(continues on next page)

(continued from previous page)

```
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.

Class-based method view can be added to the registry using *pjrpc.server.MethodRegistry.view()* decorator. Class should implement *__method__* method returning a list of methods to be exposed or inherit it from *pjrpc.server.ViewMixin* which exposes all public ones.

```
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.ViewMixin):

    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)
```

2.4.2 API versioning

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

```
import uuid

from aiohttp import web

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

methods_v1 = pjrpc.server.MethodRegistry()


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

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


methods_v2 = pjrpc.server.MethodRegistry()


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

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


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

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


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

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

2.5 Validation

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

```
import enum
import uuid
from typing import List

import pydantic
from aiohttp import web

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

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


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


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


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


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

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


class JSONEncoder(pjrpc.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)
```

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'
```

(continues on next page)

(continued from previous page)

```
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):
    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'


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 theirs 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())


class JsonRpcServer(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, JsonRpcServer):
    users = {}
```

(continues on next page)

(continued from previous page)

```
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_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
```

(continues on next page)

(continued from previous page)

```
        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 pjrpc_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 = pjrpc_aiohttp_client.Client('http://localhost/api/v1', session=session)

    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')
```

(continues on next page)

(continued from previous page)

```
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):
        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=(
```

(continues on next page)

(continued from previous page)

```
        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)])


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,
```

(continues on next page)

(continued from previous page)

```
    ),
)
jsonrpc_app.dispatcher.add_methods(methods)

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

2.10 Specification:

`pjrpc` has built-in OpenAPI and OpenRPC specification generation support implemented by *pjrpc.server.specs.openapi.OpenAPI* and *pjrpc.server.specs.openrpc.OpenRPC* respectively. To enable schema generation you should pass specification generator instance to the JSON-RPC application.

```
json_rpc = integration.JsonRPC(
    '/api/v1',
    spec=specs.OpenAPI(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                url='http://127.0.0.1:8080',
            ),
        ],
        security_schemes=dict(
            basic=specs.SecurityScheme(
                type=specs.SecuritySchemeType.HTTP,
                scheme='basic',
            ),
        ),
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
        ui=specs.SwaggerUI(),
    ),
)
```

OpenAPI specification will be available on `/api/v1/openapi.json` path. Path suffix can be overridden by passing `path` parameter to a specification generator.

For more information about the specification see OpenAPI Specification.

OpenRPC specification generation looks pretty the same:

```
json_rpc = integration.JsonRPC(
    '/api/v1',
    spec=specs.OpenRPC(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                name='test',
                url='http://127.0.0.1:8080/api/v1/',
                summary='test server',
            ),
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
    ),
)
```

OpenRPC specification will be available on `/api/v1/openrpc.json` path.

Method description, tags, errors, examples, parameters and return value schemas can be provided by hand using *pjrpc.server.specs.openapi.annotate()* decorator or automatically extracted using schema extractor. `pjrpc` provides two schema extractors: *pjrpc.server.specs.extractors.pydantic.PydanticSchemaExtractor* and `pjrpc.server.specs.extractors.docstring.DocstringSchemaExtractor`. They uses pydantic models or python docstrings for method summary, description, errors, examples and schema extraction respectively. You can implement your own schema extractor inheriting it from *pjrpc.server.specs.extractors.BaseSchemaExtractor* and implementing abstract methods.

```
@specs.annotate(
    tags=['users'],
    errors=[AlreadyExistsError],
    examples=[
        specs.MethodExample(
            summary="Simple example",
            params=dict(
                user={
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
            result={
                'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                'name': 'Alex',
                'surname': 'Smith',
                'age': 25,
            },
        ),
    ],
)
@methods.add
@validator.validate
def add_user(user: UserIn) -> UserOut:
    """
    Creates a user.

    :param object user: user data
    :return object: registered user
    :raise AlreadyExistsError: user already exists
    """

    for existing_user in flask.current_app.users_db.values():
        if user.name == existing_user.name:
            raise AlreadyExistsError()

    user_id = uuid.uuid4().hex
    flask.current_app.users_db[user_id] = user

    return UserOut(id=user_id, **user.dict())
```

2.11 Web UI

`pjrpc` supports integrated web UI as an extra dependency. Three UI types are supported:

- SwaggerUI (https://swagger.io/tools/swagger-ui/)
- RapiDoc (https://mrin9.github.io/RapiDoc/)
- ReDoc (https://github.com/Redocly/redoc)

Web UI extra dependency can be installed using the following code:

```
$ pip install pjrpc[openapi-ui-bundles]
```

To enable Web UI pass *pjrpc.server.specs.openapi.SwaggerUI*, *pjrpc.server.specs.openapi.RapiDoc* or *pjrpc.server.specs.openapi.ReDoc* to a specification generator as a `ui` parameter. Web UI will be available at `/ui/` path. It can be overridden by passing `ui_path` parameter to the specification generator.

```
json_rpc = AuthenticatedJsonRPC(
    '/api/v1',
    json_encoder=JSONEncoder,
    spec=specs.OpenAPI(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                url='http://127.0.0.1:8080',
            ),
        ],
        security_schemes=dict(
            basicAuth=specs.SecurityScheme(
                type=specs.SecuritySchemeType.HTTP,
                scheme='basic',
            ),
        ),
        security=[
            dict(basicAuth=[])
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
        ui=specs.SwaggerUI(),
    ),
)
```

The following example illustrates how to configure specification generation and Swagger UI web tool with basic auth using flask web framework:

```
import uuid
from typing import Any, Optional

import flask
import flask_httpauth
import pydantic
import flask_cors
from werkzeug import security

import pjrpc.server.specs.extractors.pydantic
from pjrpc.server.integration import flask as integration
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.specs import extractors, openapi as specs


app = flask.Flask('myapp')
flask_cors.CORS(app, resources={"/myapp/api/v1/*": {"origins": "*"}})
```

(continues on next page)

(continued from previous page)

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

auth = flask_httpauth.HTTPBasicAuth()
credentials = {"admin": security.generate_password_hash("admin")}


@auth.verify_password
def verify_password(username: str, password: str) -> Optional[str]:
    if username in credentials and security.check_password_hash(credentials.
↪get(username), password):
        return username


class AuthenticatedJsonRPC(integration.JsonRPC):
    @auth.login_required
    def _rpc_handle(self, dispatcher: pjrpc.server.Dispatcher) -> flask.Response:
        return super()._rpc_handle(dispatcher)


class JSONEncoder(pjrpc.JSONEncoder):
    def default(self, o: Any) -> Any:
        if isinstance(o, pydantic.BaseModel):
            return o.dict()
        if isinstance(o, uuid.UUID):
            return str(o)

        return super().default(o)


class UserIn(pydantic.BaseModel):
    """
    User registration data.
    """

    name: str
    surname: str
    age: int


class UserOut(UserIn):
    """
    Registered user data.
    """

    id: uuid.UUID


class AlreadyExistsError(pjrpc.exc.JsonRpcError):
    """
    User already registered error.
    """

    code = 2001
    message = "user already exists"
```

(continues on next page)

(continued from previous page)

```
class NotFoundError(pjrpc.exc.JsonRpcError):
    """
    User not found error.
    """

    code = 2002
    message = "user not found"


@specs.annotate(
    tags=['users'],
    errors=[AlreadyExistsError],
    examples=[
        specs.MethodExample(
            summary="Simple example",
            params=dict(
                user={
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
            result={
                'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                'name': 'Alex',
                'surname': 'Smith',
                'age': 25,
            },
        ),
    ],
)
@methods.add
@validator.validate
def add_user(user: UserIn) -> UserOut:
    """
    Creates a user.

    :param object user: user data
    :return object: registered user
    :raise AlreadyExistsError: user already exists
    """

    for existing_user in flask.current_app.users_db.values():
        if user.name == existing_user.name:
            raise AlreadyExistsError()

    user_id = uuid.uuid4().hex
    flask.current_app.users_db[user_id] = user

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
```

(continues on next page)

(continued from previous page)

```
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result={
                 'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                 'name': 'Alex',
                 'surname': 'Smith',
                 'age': 25,
            },
        ),
    ],
)
@methods.add
@validator.validate
def get_user(user_id: uuid.UUID) -> UserOut:
    """
    Returns a user.

    :param object user_id: user id
    :return object: registered user
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.get(user_id)
    if not user:
        raise NotFoundError()

    return UserOut(**user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result=None,
        ),
    ],
)
@methods.add
@validator.validate
def delete_user(user_id: uuid.UUID) -> None:
    """
    Deletes a user.

    :param object user_id: user id
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.pop(user_id, None)
    if not user:
```

(continues on next page)

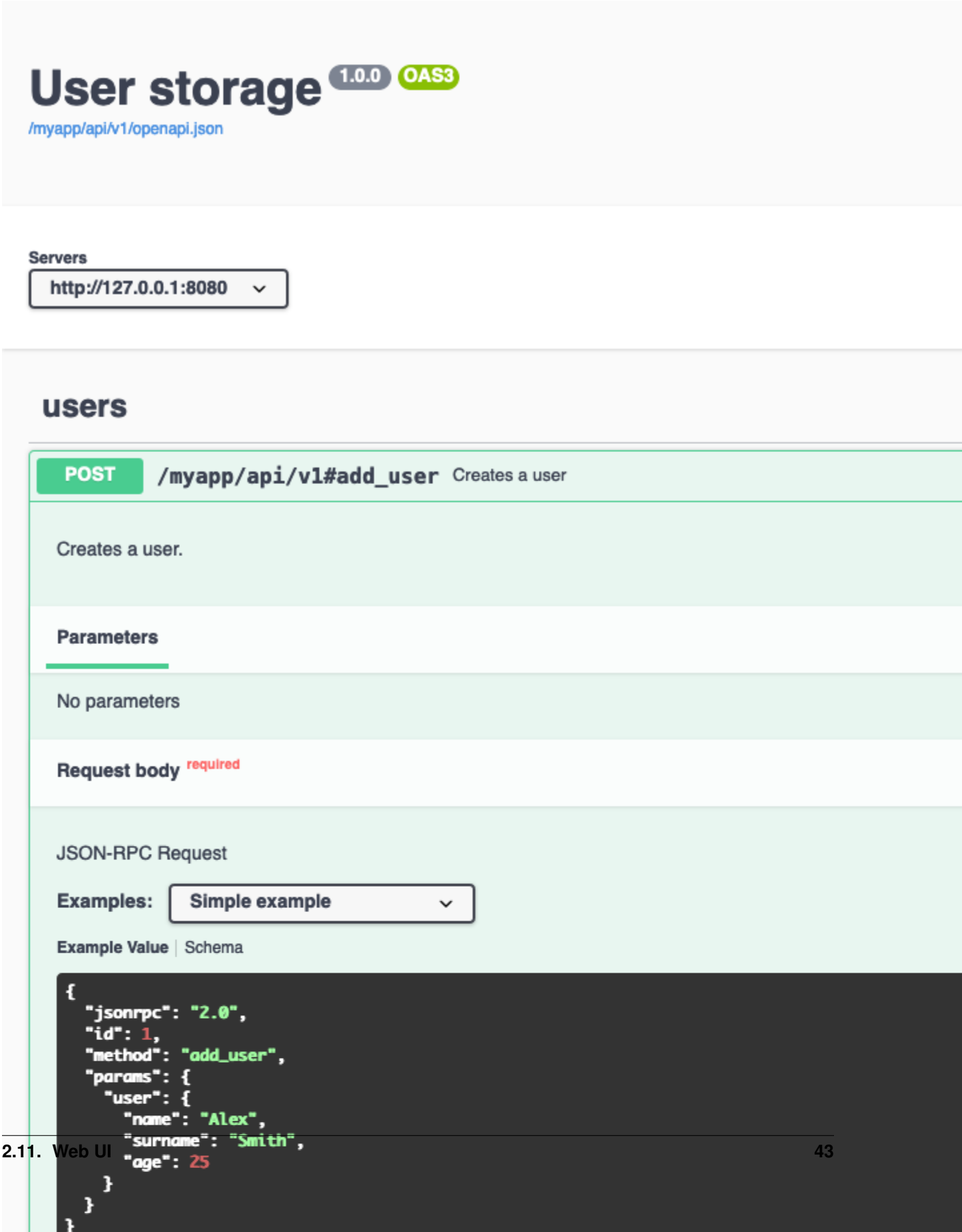
(continued from previous page)

```
        raise NotFoundError()


json_rpc = AuthenticatedJsonRPC(
    '/api/v1',
    json_encoder=JSONEncoder,
    spec=specs.OpenAPI(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                url='http://127.0.0.1:8080',
            ),
        ],
        security_schemes=dict(
            basicAuth=specs.SecurityScheme(
                type=specs.SecuritySchemeType.HTTP,
                scheme='basic',
            ),
        ),
        security=[
            dict(basicAuth=[])
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
        ui=specs.SwaggerUI(),
        # ui=specs.RapiDoc(),
        # ui=specs.ReDoc(),
    ),
)
json_rpc.dispatcher.add_methods(methods)

app.users_db = {}

myapp = flask.Blueprint('myapp', __name__, url_prefix='/myapp')
json_rpc.init_app(myapp)

app.register_blueprint(myapp)

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

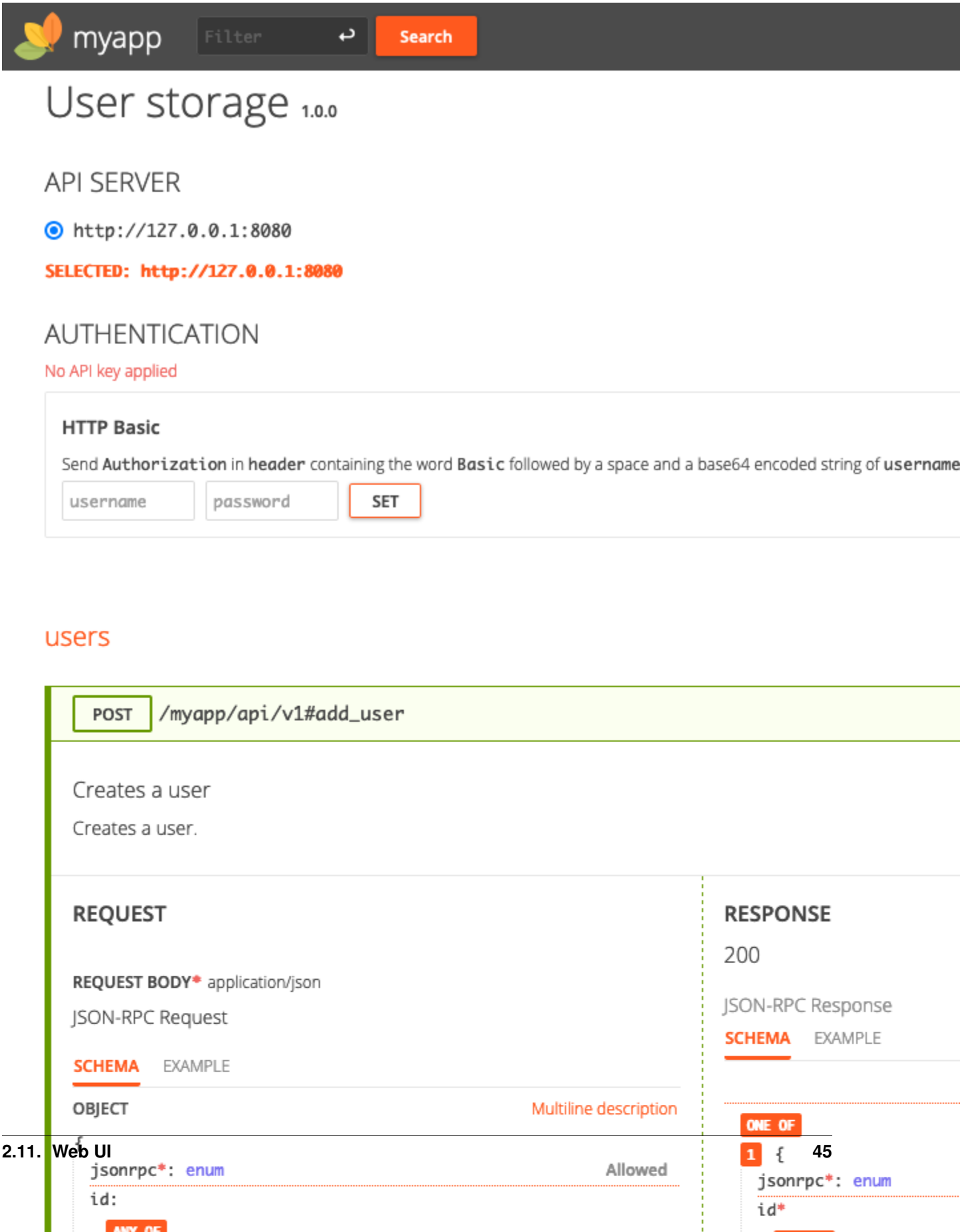
Specification is available on http://localhost:8080/myapp/api/v1/openapi.json

Web UI is running on http://localhost:8080/myapp/api/v1/ui/

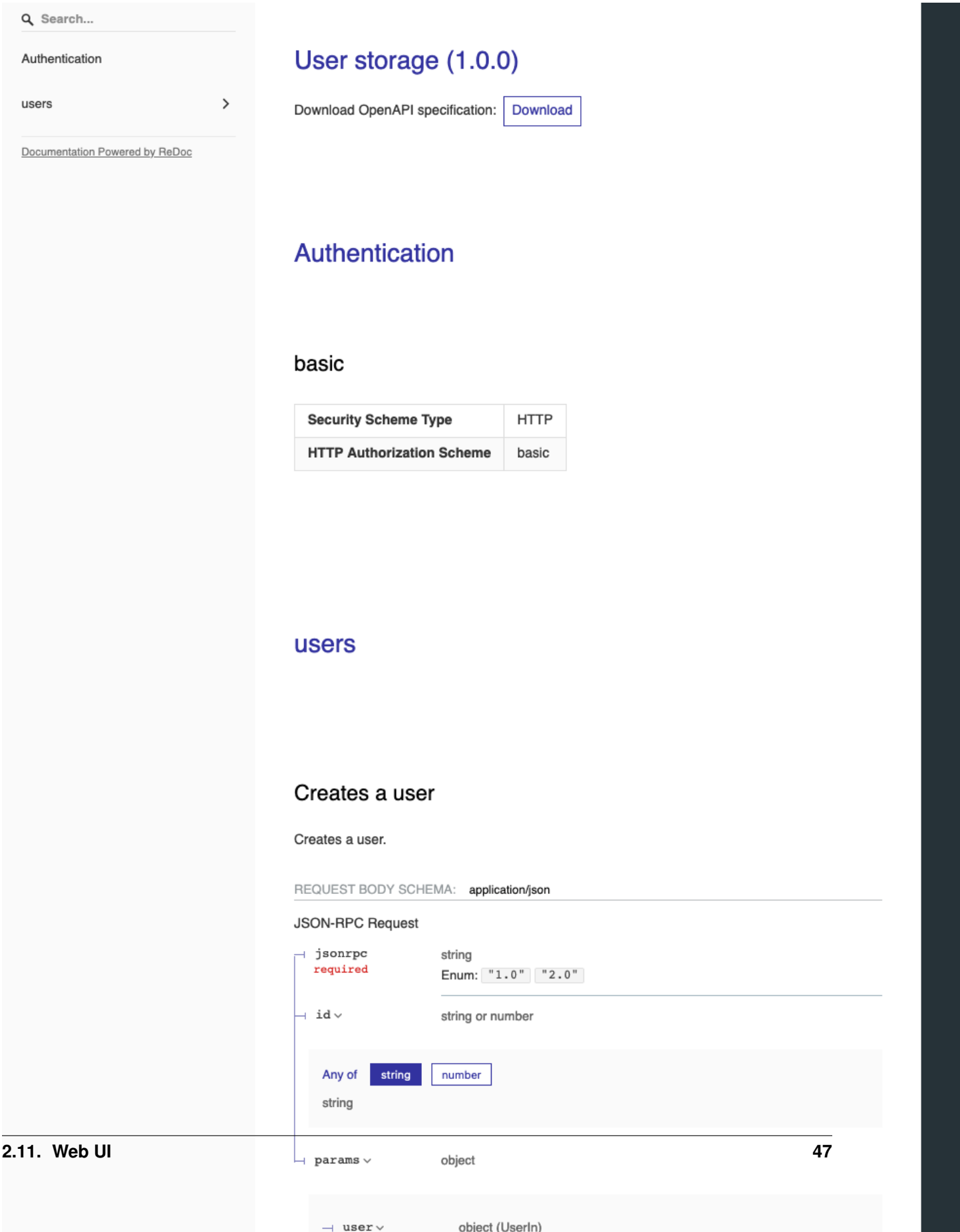
2.11.1 Swagger UI:



2.11.2 RapiDoc:



2.11.3 ReDoc:


Search...
Authentication
users
Documentation Powered by ReDoc
User storage (1.0.0)
Download OpenAPI specification:
Download
Authentication
basic
Security Scheme Type
HTTP
HTTP Authorization Scheme
basic
users
Creates a user
Creates a user.
REQUEST BODY SCHEMA: application/json
JSON-RPC Request
jsonrpc
required
string
Enum: "1.0" "2.0"
id
string or number
Any of
string
number
string
params
object
user
object (UserIn)

2.12 Examples

2.12.1 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}")

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

    await client.notify('tick')


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

2.12.2 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)
```

(continues on next page)

(continued from previous page)

```
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())
```

2.12.3 aiohttp class-based handler

```
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.ViewMixin):

    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)
```

2.12.4 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())
```

2.12.5 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).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]}")
```

(continues on next page)

(continued from previous page)

```
        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())
```

2.12.6 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)]
```

(continues on next page)

(continued from previous page)

```
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)]
```

2.12.7 aiohttp 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)
```

2.12.8 aiohttp versioning

```
import uuid

from aiohttp import web

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

methods_v1 = pjrpc.server.MethodRegistry()
```

(continues on next page)

(continued from previous page)

```
@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.app)


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

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

2.12.9 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()
```

(continues on next page)

(continued from previous page)

```
    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)
```

2.12.10 client tracing

```
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):
        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=(
```

(continues on next page)

(continued from previous page)

```
        ClientTracer(),
    ),
)

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

2.12.11 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.ViewMixin):

    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)
```

2.12.12 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 = {}

json_rpc.init_app(app)

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

2.12.13 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()
```

(continues on next page)

(continued from previous page)

```
@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)
app.users = {}


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

2.12.14 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)
```

(continues on next page)

(continued from previous page)

```
            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`
    """

    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 = {}
```

(continues on next page)

(continued from previous page)

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

    server.serve_forever()
```

2.12.15 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',
            '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'],
}
```

(continues on next page)

(continued from previous page)

```
@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)
```

2.12.16 kombu client

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


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


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')
```

2.12.17 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
```

(continues on next page)

(continued from previous page)

```
    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()
```

2.12.18 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")


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)
```

2.12.19 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'


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)
```

2.12.20 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
```

(continues on next page)

(continued from previous page)

```
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.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)
```

2.12.21 requests client

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


```

(continues on next page)

(continued from previous page)

```
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')
```

2.12.22 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_mocker):
    client = requests_client.Client('http://localhost/api/v1')

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

    pjrpc_requests_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:
        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_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)]

    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')
```

(continues on next page)

(continued from previous page)

```
    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)
```

2.12.23 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')
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)
```

2.12.24 server prometheus metrics

```
import asyncio
from typing import Any, Callable
```

(continues on next page)

(continued from previous page)

```
import prometheus_client as pc
from aiohttp import web

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

method_error_count = pc.Counter('method_error_count', 'Method error count',␣
↪labelnames=['method', 'code'])
method_latency_hist = pc.Histogram('method_latency', 'Method latency', labelnames=[
↪'method'])
method_active_count = pc.Gauge('method_active_count', 'Method active count',␣
↪labelnames=['method'])


async def metrics(request):
    return web.Response(body=pc.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: web.Request):
    print("method started")
    await asyncio.sleep(1)
    print("method finished")


async def latency_metric_middleware(request: pjrpc.Request, context: web.Request,␣
↪handler: Callable) -> Any:
    with method_latency_hist.labels(method=request.method).time():
        return await handler(request, context)


async def active_count_metric_middleware(request: pjrpc.Request, context: web.Request,
↪ handler: Callable) -> Any:
    with method_active_count.labels(method=request.method).track_inprogress():
        return await handler(request, context)


async def any_error_handler(
    request: pjrpc.Request, context: web.Request, error: pjrpc.exceptions.
↪JsonRpcError,
) -> pjrpc.exceptions.JsonRpcError:
    method_error_count.labels(method=request.method, code=error.code).inc()

    return error


async def validation_error_handler(
    request: pjrpc.Request, context: web.Request, error: pjrpc.exceptions.
↪JsonRpcError,
) -> pjrpc.exceptions.JsonRpcError:
```

(continues on next page)

(continued from previous page)

```
    print("validation error occurred")

    return error


jsonrpc_app = aiohttp.Application(
    '/api/v1',
    app=http_app,
    middlewares=(
        latency_metric_middleware,
        active_count_metric_middleware,
    ),
    error_handlers={
        -32602: [validation_error_handler],
        None: [any_error_handler],
    },
)
jsonrpc_app.dispatcher.add_methods(methods)

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

2.12.25 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)
```

(continues on next page)

(continued from previous page)

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

2.12.26 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)
```

2.12.27 flask OpenAPI specification

```
import uuid
from typing import Any, Optional

import flask
import flask_httpauth
import pydantic
import flask_cors
from werkzeug import security

import pjrpc.server.specs.extractors.pydantic
from pjrpc.server.integration import flask as integration
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.specs import extractors, openapi as specs


app = flask.Flask('myapp')
flask_cors.CORS(app, resources={"/myapp/api/v1/*": {"origins": "*"}})

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

auth = flask_httpauth.HTTPBasicAuth()
credentials = {"admin": security.generate_password_hash("admin")}
```

(continues on next page)

(continued from previous page)

```
@auth.verify_password
def verify_password(username: str, password: str) -> Optional[str]:
    if username in credentials and security.check_password_hash(credentials.
↪get(username), password):
        return username


class AuthenticatedJsonRPC(integration.JsonRPC):
    @auth.login_required
    def _rpc_handle(self, dispatcher: pjrpc.server.Dispatcher) -> flask.Response:
        return super()._rpc_handle(dispatcher)


class JSONEncoder(pjrpc.JSONEncoder):
    def default(self, o: Any) -> Any:
        if isinstance(o, pydantic.BaseModel):
            return o.dict()
        if isinstance(o, uuid.UUID):
            return str(o)

        return super().default(o)


class UserIn(pydantic.BaseModel):
    """
    User registration data.
    """

    name: str
    surname: str
    age: int


class UserOut(UserIn):
    """
    Registered user data.
    """

    id: uuid.UUID


class AlreadyExistsError(pjrpc.exc.JsonRpcError):
    """
    User already registered error.
    """

    code = 2001
    message = "user already exists"


class NotFoundError(pjrpc.exc.JsonRpcError):
    """
    User not found error.
    """

    code = 2002
    message = "user not found"
```

(continues on next page)

(continued from previous page)

```
@specs.annotate(
    tags=['users'],
    errors=[AlreadyExistsError],
    examples=[
        specs.MethodExample(
            summary="Simple example",
            params=dict(
                user={
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
            result={
                'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                'name': 'Alex',
                'surname': 'Smith',
                'age': 25,
            },
        ),
    ],
)
@methods.add
@validator.validate
def add_user(user: UserIn) -> UserOut:
    """
    Creates a user.

    :param object user: user data
    :return object: registered user
    :raise AlreadyExistsError: user already exists
    """

    for existing_user in flask.current_app.users_db.values():
        if user.name == existing_user.name:
            raise AlreadyExistsError()

    user_id = uuid.uuid4().hex
    flask.current_app.users_db[user_id] = user

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result={
                 'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                 'name': 'Alex',
```

(continues on next page)

(continued from previous page)

```
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
        ],
)
@methods.add
@validator.validate
def get_user(user_id: uuid.UUID) -> UserOut:
    """
    Returns a user.

    :param object user_id: user id
    :return object: registered user
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.get(user_id.hex)
    if not user:
        raise NotFoundError()

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result=None,
        ),
    ],
)
@methods.add
@validator.validate
def delete_user(user_id: uuid.UUID) -> None:
    """
    Deletes a user.

    :param object user_id: user id
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.pop(user_id.hex, None)
    if not user:
        raise NotFoundError()


json_rpc = AuthenticatedJsonRPC(
    '/api/v1',
    json_encoder=JSONEncoder,
    spec=specs.OpenAPI(
        info=specs.Info(version="1.0.0", title="User storage"),
```

(continues on next page)

(continued from previous page)

```
        servers=[
            specs.Server(
                url='http://127.0.0.1:8080',
            ),
        ],
        security_schemes=dict(
            basicAuth=specs.SecurityScheme(
                type=specs.SecuritySchemeType.HTTP,
                scheme='basic',
            ),
        ),
        security=[
            dict(basicAuth=[]),
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
        ui=specs.SwaggerUI(),
        # ui=specs.RapiDoc(),
        # ui=specs.ReDoc(),
    ),
)
json_rpc.dispatcher.add_methods(methods)

app.users_db = {}

myapp = flask.Blueprint('myapp', __name__, url_prefix='/myapp')
json_rpc.init_app(myapp)

app.register_blueprint(myapp)

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

2.12.28 aiohttp OpenAPI specification

```
import uuid
from typing import Any

import pydantic
from aiohttp import helpers, web

import pjrpc.server.specs.extractors.pydantic
from pjrpc.server.integration import aiohttp as integration
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.specs import extractors, openapi as specs


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

credentials = {"admin": "admin"}


class JSONEncoder(pjrpc.JSONEncoder):
    def default(self, o: Any) -> Any:
        if isinstance(o, pydantic.BaseModel):
```

(continues on next page)

(continued from previous page)

```
            return o.dict()
        if isinstance(o, uuid.UUID):
            return str(o)

        return super().default(o)


class AuthenticatedJsonRPC(integration.Application):
    async def _rpc_handle(self, http_request: web.Request, dispatcher: pjrpc.server.
↪Dispatcher) -> web.Response:
        try:
            auth = helpers.BasicAuth.decode(http_request.headers.get('Authorization',
↪''))
        except ValueError:
            raise web.HTTPUnauthorized

        if credentials.get(auth.login) != auth.password:
            raise web.HTTPUnauthorized

        return await super()._rpc_handle(http_request=http_request,␣
↪dispatcher=dispatcher)


class UserIn(pydantic.BaseModel):
    """
    User registration data.
    """

    name: str
    surname: str
    age: int


class UserOut(UserIn):
    """
    Registered user data.
    """

    id: uuid.UUID


class AlreadyExistsError(pjrpc.exc.JsonRpcError):
    """
    User already registered error.
    """

    code = 2001
    message = "user already exists"


class NotFoundError(pjrpc.exc.JsonRpcError):
    """
    User not found error.
    """

    code = 2002
    message = "user not found"
```

(continues on next page)

(continued from previous page)

```
@specs.annotate(
    tags=['users'],
    errors=[AlreadyExistsError],
    examples=[
        specs.MethodExample(
            summary="Simple example",
            params=dict(
                user={
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
            result={
                'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                'name': 'Alex',
                'surname': 'Smith',
                'age': 25,
            },
        ),
    ],
)
@methods.add(context='request')
@validator.validate
def add_user(request: web.Request, user: UserIn) -> UserOut:
    """
    Creates a user.

    :param request: http request
    :param object user: user data
    :return object: registered user
    :raise AlreadyExistsError: user already exists
    """

    for existing_user in request.config_dict['users'].values():
        if user.name == existing_user.name:
            raise AlreadyExistsError()

    user_id = uuid.uuid4().hex
    request.config_dict['users'][user_id] = user

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result={
                 'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
```

(continues on next page)

(continued from previous page)

```
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
        ],
)
@methods.add(context='request')
@validator.validate
def get_user(request: web.Request, user_id: uuid.UUID) -> UserOut:
    """
    Returns a user.

    :param request: http request
    :param object user_id: user id
    :return object: registered user
    :raise NotFoundError: user not found
    """

    user = request.config_dict['users'].get(user_id.hex)
    if not user:
        raise NotFoundError()

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            summary='Simple example',
            params=dict(
                user_id='c47726c6-a232-45f1-944f-60b98966ff1b',
            ),
            result=None,
        ),
    ],
)
@methods.add(context='request')
@validator.validate
def delete_user(request: web.Request, user_id: uuid.UUID) -> None:
    """
    Deletes a user.

    :param request: http request
    :param object user_id: user id
    :raise NotFoundError: user not found
    """

    user = request.config_dict['users'].pop(user_id.hex, None)
    if not user:
        raise NotFoundError()


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

(continues on next page)

(continued from previous page)

```
jsonrpc_app = AuthenticatedJsonRPC(
    '/api/v1',
    json_encoder=JSONEncoder,
    spec=specs.OpenAPI(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                url='http://127.0.0.1:8080',
            ),
        ],
        security_schemes=dict(
            basicAuth=specs.SecurityScheme(
                type=specs.SecuritySchemeType.HTTP,
                scheme='basic',
            ),
        ),
        security=[
            dict(basicAuth=[]),
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
        ui=specs.SwaggerUI(),
        # ui=specs.RapiDoc(),
        # ui=specs.ReDoc(),
    ),
)
jsonrpc_app.dispatcher.add_methods(methods)
app.add_subapp('/myapp', jsonrpc_app.app)

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

2.12.29 flask OpenRPC specification

```
import uuid
from typing import Any

import flask
import pydantic
from flask_cors import CORS

import pjrpc.server.specs.extractors.pydantic
import pjrpc.server.specs.extractors.docstring
from pjrpc.server.integration import flask as integration
from pjrpc.server.validators import pydantic as validators
from pjrpc.server.specs import extractors, openrpc as specs

app = flask.Flask(__name__)
CORS(app, resources={r"/api/v1/*": {"origins": "*"}})

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


class JsonEncoder(pjrpc.JSONEncoder):
```

(continues on next page)

(continued from previous page)

```
    def default(self, o: Any) -> Any:
        if isinstance(o, pydantic.BaseModel):
            return o.dict()
        if isinstance(o, uuid.UUID):
            return str(o)

        return super().default(o)


class UserIn(pydantic.BaseModel):
    """
    User registration data.
    """

    name: str
    surname: str
    age: int


class UserOut(UserIn):
    """
    Registered user data.
    """

    id: uuid.UUID


class AlreadyExistsError(pjrpc.exc.JsonRpcError):
    """
    User already registered error.
    """

    code = 2001
    message = "user already exists"


class NotFoundError(pjrpc.exc.JsonRpcError):
    """
    User not found error.
    """

    code = 2002
    message = "user not found"


@specs.annotate(
    errors=[AlreadyExistsError],
    tags=['users'],
    examples=[
        specs.MethodExample(
            name='Simple user',
            params=[
                specs.ExampleObject(
                    name='user',
                    value={
                        'name': 'Alex',
                        'surname': 'Smith',
```

(continues on next page)

(continued from previous page)

```
                        'age': 25,
                    },
                ),
            ],
            result=specs.ExampleObject(
                name='result',
                value={
                    'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                    'name': 'Alex',
                    'surname': 'Smith',
                    'age': 25,
                },
            ),
        ),
    ],
)
@methods.add
@validator.validate
def add_user(user: UserIn) -> UserOut:
    """
    Adds a new user.

    :param object user: user data
    :return object: registered user
    :raise AlreadyExistsError: user already exists
    """

    for existing_user in flask.current_app.users_db.values():
        if user.name == existing_user.name:
            raise AlreadyExistsError()

    user_id = uuid.uuid4().hex
    flask.current_app.users_db[user_id] = user

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            name='Simple example',
            params=[
                specs.ExampleObject(
                    name='user',
                    value={
                        'user_id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                    },
                ),
            ],
            result=specs.ExampleObject(
                name="result",
                value={
                    'id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                    'name': 'Alex',
                    'surname': 'Smith',
```

(continues on next page)

(continued from previous page)

```
                    'age': 25,
                },
            ),
        ),
    ],
)
@methods.add
@validator.validate
def get_user(user_id: uuid.UUID) -> UserOut:
    """
    Returns a user.

    :param object user_id: user id
    :return object: registered user
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.get(user_id.hex)
    if not user:
        raise NotFoundError()

    return UserOut(id=user_id, **user.dict())


@specs.annotate(
    tags=['users'],
    errors=[NotFoundError],
    examples=[
        specs.MethodExample(
            name='Simple example',
            summary='Simple example',
            params=[
                specs.ExampleObject(
                    name='user',
                    value={
                        'user_id': 'c47726c6-a232-45f1-944f-60b98966ff1b',
                    },
                ),
            ],
            result=specs.ExampleObject(
                name="result",
                value=None,
            ),
        ),
    ],
)
@methods.add
@validator.validate
def delete_user(user_id: uuid.UUID) -> None:
    """
    Deletes a user.

    :param object user_id: user id
    :raise NotFoundError: user not found
    """

    user = flask.current_app.users_db.pop(user_id.hex, None)
```

(continues on next page)

(continued from previous page)

```
    if not user:
        raise NotFoundError()


json_rpc = integration.JsonRPC(
    '/api/v1',
    json_encoder=JsonEncoder,
    spec=specs.OpenRPC(
        info=specs.Info(version="1.0.0", title="User storage"),
        servers=[
            specs.Server(
                name='test',
                url='http://127.0.0.1:8080/api/v1/',
                summary='test server',
            ),
        ],
        schema_extractor=extractors.pydantic.PydanticSchemaExtractor(),
    ),
)
json_rpc.dispatcher.add_methods(methods)

app.users_db = {}

json_rpc.init_app(app)

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

CHAPTER 3

The API Documentation

3.1 Developer Interface

Extensible JSON-RPC client/server library.

3.1.1 Common

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

class pjrpc.common.**Request**(*method: str*, *params: Union[list, dict, None] = None*, *id: Union[int, str, None] = None*)

JSON-RPC version 2.0 request.

Parameters

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

classmethod from_json(*json_data: Union[str, int, float, dict, bool, list, tuple, set, None]*) → pjrpc.common.v20.Request

Deserializes a request from json data.

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

Returns request object

Raises *pjrpc.common.exceptions.DeserializationError* if format is incorrect

id

Request identifier.

method

Request method name.

params
Request method parameters.

to_json() → Union[str, int, float, dict, bool, list, tuple, set, None]
Serializes the request to json data.

Returns json data

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

class pjrpc.common.**Response**(*id: Union[int, str, None], result: Union[pjrpc.common.common.UnsetType, Any] = UNSET, error: Union[pjrpc.common.common.UnsetType, pjrpc.common.exceptions.JsonRpcError] = UNSET*)
JSON-RPC version 2.0 response.

Parameters

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

classmethod from_json(*json_data: Union[str, int, float, dict, bool, list, tuple, set, None], error_cls: Type[pjrpc.common.exceptions.JsonRpcError] = <class 'pjrpc.common.exceptions.JsonRpcError'>*) → pjrpc.common.v20.Response
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.exceptions.DeserializationError* if format is incorrect

id
Response identifier.

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

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

is_success
Returns `True` if the response has succeeded.

is_error
Returns `True` if the response has not succeeded.

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

to_json() → Union[str, int, float, dict, bool, list, tuple, set, None]
Serializes the response to json data.

Returns json data

class pjrpc.common.**BatchRequest**(**requests*, *strict: bool = True*)

JSON-RPC 2.0 batch request.

Parameters

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

classmethod from_json(*data: Union[str, int, float, dict, bool, list, tuple, set, None]*) → pjrpc.common.v20.BatchRequest

Deserializes a batch request from json data.

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

Returns batch request object

append(*request: pjrpc.common.v20.Request*) → None

Appends a request to the batch.

extend(*requests: Iterable[pjrpc.common.v20.Request]*) → None

Extends a batch with *requests*.

to_json() → Union[str, int, float, dict, bool, list, tuple, set, None]

Serializes the request to json data.

Returns json data

is_notification

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

class pjrpc.common.**BatchResponse**(**responses*, *error: Union[pjrpc.common.common.UnsetType, pjrpc.common.exceptions.JsonRpcError] = UNSET*, *strict: bool = True*)

JSON-RPC 2.0 batch response.

Parameters

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

classmethod from_json(*json_data: Union[str, int, float, dict, bool, list, tuple, set, None]*, *error_cls: Type[pjrpc.common.exceptions.JsonRpcError] = <class 'pjrpc.common.exceptions.JsonRpcError'>*) → pjrpc.common.v20.BatchResponse

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

error

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

is_success

Returns `True` if the response has succeeded.

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.

has_error
Returns `True` if any response has an error.

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.

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

append(*response: pjrpc.common.v20.Response*) → None
Appends a response to the batch.

extend(*responses: Iterable[pjrpc.common.v20.Response]*) → None
Extends the batch with the *responses*.

to_json() → Union[str, int, float, dict, bool, list, tuple, set, None]
Serializes the batch response to json data.

Returns json data

class pjrpc.common.**UnsetType**
Sentinel object. Used to distinct unset (missing) values from `None` ones.

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: Any*) → Any
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.**IdentityError**
Raised when a batch requests/responses identifiers are not unique or missing.

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

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.**JsonRpcError**(*code: Optional[int] = None*, *message: Optional[str] = None*, *data: Union[pjrpc.common.common.UnsetType, Any] = 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: Union[str, int, float, dict, bool, list, tuple, set, None]*) → pjrpc.common.exceptions.JsonRpcError

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.exceptions.DeserializationError* if format is incorrect

to_json() → Union[str, int, float, dict, bool, list, tuple, set, None]

Serializes the error to a dict.

Returns serialized error

exception pjrpc.common.exceptions.**ClientError**(*code: Optional[int] = None*, *message: Optional[str] = None*, *data: Union[pjrpc.common.common.UnsetType, Any] = UNSET*)

Raised when a client sent an incorrect request.

exception pjrpc.common.exceptions.**ParseError**(*code: Optional[int] = None*, *message: Optional[str] = None*, *data: Union[pjrpc.common.common.UnsetType, Any] = UNSET*)

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

exception pjrpc.common.exceptions.**InvalidRequestError**(*code: Optional[int] = None*, *message: Optional[str] = None*, *data: Union[pjrpc.common.common.UnsetType, Any] = UNSET*)

The JSON sent is not a valid request object.

exception pjrpc.common.exceptions.**MethodNotFoundError**(*code: Optional[int] = None*, *message: Optional[str] = None*, *data: Union[pjrpc.common.common.UnsetType, Any] = UNSET*)

The method does not exist / is not available.

exception pjrpc.common.exceptions.**InvalidParamsError**(*code: Optional[int] = None, message: Optional[str] = None, data: Union[pjrpc.common.common.UnsetType, Any] = UNSET*)

Invalid method parameter(s).

exception pjrpc.common.exceptions.**InternalError**(*code: Optional[int] = None, message: Optional[str] = None, data: Union[pjrpc.common.common.UnsetType, Any] = UNSET*)

Internal JSON-RPC error.

exception pjrpc.common.exceptions.**ServerError**(*code: Optional[int] = None, message: Optional[str] = None, data: Union[pjrpc.common.common.UnsetType, Any] = 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.**sequential**(*start: int = 1*, *step: int = 1*) → Generator[int, None, None]

Sequential id generator. Returns consecutive values starting from *start* with step *step*.

pjrpc.common.generators.**randint**(*a: int*, *b: int*) → Generator[int, None, None]

Random integer id generator. Returns random integers between *a* and *b*.

pjrpc.common.generators.**random**(*length: int = 8*, *chars: str = '0123456789abcdefghijklmnopqrstuvwxyz'*) → Generator[str, None, None]

Random string id generator. Returns random strings of length *length* using alphabet *chars*.

pjrpc.common.generators.**uuid**() → Generator[uuid.UUID, None, None]

UUID id generator. Returns random UUIDs.

3.1.2 Client

JSON-RPC client.

class pjrpc.client.**AbstractClient**(*request_class: Type[pjrpc.common.v20.Request] = <class 'pjrpc.common.v20.Request'>, response_class: Type[pjrpc.common.v20.Response] = <class 'pjrpc.common.v20.Response'>, batch_request_class: Type[pjrpc.common.v20.BatchRequest] = <class 'pjrpc.common.v20.BatchRequest'>, batch_response_class: Type[pjrpc.common.v20.BatchResponse] = <class 'pjrpc.common.v20.BatchResponse'>, error_cls: Type[pjrpc.common.exceptions.JsonRpcError] = <class 'pjrpc.common.exceptions.JsonRpcError'>, id_gen_impl: Callable[[...], Generator[Union[int, str], None, None]] = <function sequential>, json_loader: Callable = <function loads>, json_dumper: Callable = <function dumps>, json_encoder: Type[pjrpc.common.common.JSONEncoder] = <class 'pjrpc.common.common.JSONEncoder'>, json_decoder: Optional[json.decoder.JSONDecoder] = None, strict: bool = True, request_args: Optional[Dict[str, Any]] = None, tracers: Iterable[pjrpc.client.tracer.Tracer] = ()*)

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_impl** – 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: pjrpc.client.client.AbstractClient*)

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

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

proxy

Clint proxy object.

batch

Client batch wrapper.

notify(*method: str*, **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

call(*method: str*, **args*, *_trace_ctx: types.SimpleNamespace = namespace()*, ***kwargs*) → Optional[pjrpc.common.v20.Response]
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: pjrpc.common.v20.Request*, *_trace_ctx: types.SimpleNamespace = namespace()*, ***kwargs*) → Optional[pjrpc.common.v20.Response]
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: Type[pjrpc.common.v20.Request] = <class 'pjrpc.common.v20.Request'>, response_class: Type[pjrpc.common.v20.Response] = <class 'pjrpc.common.v20.Response'>, batch_request_class: Type[pjrpc.common.v20.BatchRequest] = <class 'pjrpc.common.v20.BatchRequest'>, batch_response_class: Type[pjrpc.common.v20.BatchResponse] = <class 'pjrpc.common.v20.BatchResponse'>, error_cls: Type[pjrpc.common.exceptions.JsonRpcError] = <class 'pjrpc.common.exceptions.JsonRpcError'>, id_gen_impl: Callable[[...], Generator[Union[int, str], None, None]] = <function sequential>, json_loader: Callable = <function loads>, json_dumper: Callable = <function dumps>, json_encoder: Type[pjrpc.common.common.JSONEncoder] = <class 'pjrpc.common.common.JSONEncoder'>, json_decoder: Optional[json.decoder.JSONDecoder] = None, strict: bool = True, request_args: Optional[Dict[str, Any]] = None, tracers: Iterable[pjrpc.client.tracer.Tracer] = ()*)

Abstract asynchronous JSON-RPC client.

batch
Client batch wrapper.

call(*method: str*, **args*, *_trace_ctx: types.SimpleNamespace = namespace()*, ***kwargs*) → Any
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: pjrpc.common.v20.Request*, *_trace_ctx: types.SimpleNamespace = namespace()*, ***kwargs*) → Optional[pjrpc.common.v20.Response]
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: logging.Logger = <RootLogger root (WARNING)>*, *level: int = 10*)
JSON-RPC client logging tracer.

on_request_begin(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*) → None
Handler called before JSON-RPC request begins.

Parameters

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

on_request_end(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*, *response: pjrpc.common.v20.Response*) → None
Handler called after JSON-RPC request ends.

Parameters

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

on_error(*trace_context: types.SimpleNamespace, request: Union[pjrpc.common.v20.Request, pjrpc.common.v20.BatchRequest], error: BaseException*) → None
Handler called when JSON-RPC request failed.

Parameters

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

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

on_request_begin(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*) → None
Handler called before JSON-RPC request begins.

Parameters

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

on_request_end(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*, *response: pjrpc.common.v20.Response*) → None
Handler called after JSON-RPC request ends.

Parameters

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

on_error(*trace_context: types.SimpleNamespace, request: Union[pjrpc.common.v20.Request, pjrpc.common.v20.BatchRequest], error: BaseException*) → None
Handler called when JSON-RPC request failed.

Parameters

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

Backends

class pjrpc.client.backend.requests.**Client**(*url: str*, *session: Optional[requests.sessions.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() → None
Closes the current http session.

class pjrpc.client.backend.aiohttp.**Client**(*url: str*, *session_args: Optional[Dict[str, Any]] = None*, *session: Optional[aiohttp.client.ClientSession] = 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() → None
Closes current http session.

class pjrpc.client.backend.kombu.**Client**(*broker_url: str*, *queue_name: Optional[str] = None*, *conn_args: Optional[Dict[str, Any]] = None*, *exchange_name: Optional[str] = None*, *exchange_args: Optional[Dict[str, Any]] = None*, *routing_key: Optional[str] = None*, *result_queue_name: Optional[str] = None*, *result_queue_args: Optional[Dict[str, Any]] = 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() → None
Closes the current broker connection.

class pjrpc.client.backend.aio_pika.**Client**(*broker_url: str*, *queue_name: Optional[str] = None*, *conn_args: Optional[Dict[str, Any]] = None*, *exchange_name: Optional[str] = None*, *exchange_args: Optional[Dict[str, Any]] = None*, *routing_key: Optional[str] = None*, *result_queue_name: Optional[str] = None*, *result_queue_args: Optional[Dict[str, Any]] = 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() → None
Closes current broker connection.

connect() → None
Opens a connection to the broker.

Tracer

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

on_request_begin(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*) → None
Handler called before JSON-RPC request begins.

Parameters

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

on_request_end(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*, *response: pjrpc.common.v20.Response*) → None
Handler called after JSON-RPC request ends.

Parameters

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

on_error(*trace_context: types.SimpleNamespace, request: Union[pjrpc.common.v20.Request, pjrpc.common.v20.BatchRequest], error: BaseException*) → None
Handler called when JSON-RPC request failed.

Parameters

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

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

on_request_begin(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*) → None
Handler called before JSON-RPC request begins.

Parameters

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

on_request_end(*trace_context: types.SimpleNamespace*, *request: pjrpc.common.v20.Request*, *response: pjrpc.common.v20.Response*) → None
Handler called after JSON-RPC request ends.

Parameters

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

on_error(*trace_context: types.SimpleNamespace, request: Union[pjrpc.common.v20.Request, pjrpc.common.v20.BatchRequest], error: BaseException*) → None
Handler called when JSON-RPC request failed.

Parameters

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

Integrations

3.1.3 Server

JSON-RPC server package.

class pjrpc.server.**AsyncDispatcher**(**, request_class: Type[pjrpc.common.v20.Request] = <class 'pjrpc.common.v20.Request'>, response_class: Type[pjrpc.common.v20.Response] = <class 'pjrpc.common.v20.Response'>, batch_request: Type[pjrpc.common.v20.BatchRequest] = <class 'pjrpc.common.v20.BatchRequest'>, batch_response: Type[pjrpc.common.v20.BatchResponse] = <class 'pjrpc.common.v20.BatchResponse'>, json_loader: Callable = <function loads>, json_dumper: Callable = <function dumps>, json_encoder: Type[pjrpc.server.dispatcher.JSONEncoder] = <class 'pjrpc.server.dispatcher.JSONEncoder'>, json_decoder: Optional[Type[json.decoder.JSONDecoder]] = None, middlewares: Iterable[Callable] = (), error_handlers: Dict[Union[None, int, Exception], List[Callable]] = {}*)
Asynchronous method dispatcher.

dispatch(*request_text: str*, *context: Optional[Any] = None*) → Optional[str]
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: Type[pjrpc.common.v20.Request] = <class 'pjrpc.common.v20.Request'>, response_class: Type[pjrpc.common.v20.Response] = <class 'pjrpc.common.v20.Response'>, batch_request: Type[pjrpc.common.v20.BatchRequest] = <class 'pjrpc.common.v20.BatchRequest'>, batch_response: Type[pjrpc.common.v20.BatchResponse] = <class 'pjrpc.common.v20.BatchResponse'>, json_loader: Callable = <function loads>, json_dumper: Callable = <function dumps>, json_encoder: Type[pjrpc.server.dispatcher.JSONEncoder] = <class 'pjrpc.server.dispatcher.JSONEncoder'>, json_decoder: Optional[Type[json.decoder.JSONDecoder]] = None, middlewares: Iterable[Callable] = (), error_handlers: Dict[Union[None, int, Exception], List[Callable]] = {}*)

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
- **middlewares** – request middlewares
- **error_handlers** – request error handlers

add(*method: Callable*, *name: Optional[str] = None*, *context: Optional[Any] = None*) → None

Adds method to the registry.

Parameters

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

add_methods(**methods*) → None

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

view(*view: Type[pjrpc.server.dispatcher.ViewMixin]*) → None

Adds class based view to the registry.

Parameters **view** – view to be added

dispatch(*request_text: str*, *context: Optional[Any] = None*) → Optional[str]

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.**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: Any*) → Any

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: Callable*, *name: Optional[str] = None*, *context: Optional[Any] = 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: Optional[str] = None*)

Method registry.

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

get(*item: str*) → Optional[pjrpc.server.dispatcher.Method]

Returns a method from the registry by name.

Parameters **item** – method name

Returns found method or *None*

add(*maybe_method: Optional[Callable] = None*, *name: Optional[str] = None*, *context: Optional[Any] = None*) → Callable

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*) → None

Adds methods to the registry.

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

view(*maybe_view: Optional[Type[pjrpc.server.dispatcher.ViewMixin]] = None*, *context: Optional[Any] = None*, *prefix: Optional[str] = None*) → Union[pjrpc.server.dispatcher.ViewMixin, Callable]
Methods view decorator.

Parameters

- **maybe_view** – view class instance or *None*
- **context** – application context name
- **prefix** – view methods prefix

Returns decorator or decorated view

merge(*other: pjrpc.server.dispatcher.MethodRegistry*) → None
Merges two registries.

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

class pjrpc.server.**ViewMixin**
Simple class based method handler mixin. Exposes all public methods.

Integrations

aiohttp

aiohttp JSON-RPC server integration.

class pjrpc.server.integration.aiohttp.**Application**(*path: str = ''*, *spec: Optional[pjrpc.server.specs.Specification] = None*, *app: Optional[aiohttp.web_app.Application] = 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.

endpoints
JSON-RPC application registered endpoints.

add_endpoint(*prefix: str*, *subapp: Optional[aiohttp.web_app.Application] = None*, ***kwargs*) → pjrpc.server.dispatcher.Dispatcher
Adds additional endpoint.

Parameters

- **prefix** – endpoint prefix

- **subapp** – aiohttp subapp the endpoint will be served on
- **kwargs** – arguments to be passed to the dispatcher *pjrpc.server.Dispatcher*

Returns dispatcher

flask

Flask JSON-RPC extension.

class pjrpc.server.integration.flask.**JsonRPC**(*path: str*, *spec: Optional[pjrpc.server.specs.Specification] = None*, ***kwargs*)

Flask framework JSON-RPC extension class.

Parameters

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

dispatcher
JSON-RPC method dispatcher.

endpoints
JSON-RPC application registered endpoints.

add_endpoint(*prefix: str*, *blueprint: Optional[flask.blueprints.Blueprint] = None*, ***kwargs*) → pjrpc.server.dispatcher.Dispatcher
Adds additional endpoint.

Parameters

- **prefix** – endpoint prefix
- **blueprint** – flask blueprint the endpoint will be served on
- **kwargs** – arguments to be passed to the dispatcher *pjrpc.server.Dispatcher*

Returns dispatcher

init_app(*app: Union[flask.app.Flask, flask.blueprints.Blueprint]*) → None
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: str*, *queue_name: str*, *conn_args: Optional[Dict[str, Any]] = None*, *queue_args: Optional[Dict[str, Any]] = None*, *publish_args: Optional[Dict[str, Any]] = None*, *prefetch_count: int = 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: str*, *queue_name: str*, *prefetch_count: int = 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() → None
Stops executor.

start(*queue_args: Optional[Dict[str, Any]] = None*) → None
Starts executor.

Parameters **queue_args** – queue arguments

werkzeug

class pjrpc.server.integration.werkzeug.**JsonRPC**(*path: str = ''*, ***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.

validate(*maybe_method: Optional[Callable] = None*, ***kwargs*) → Callable
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: Callable, params: Union[list, dict, None], exclude: Iterable[str] = ()*, ***kwargs*) → Dict[str, Any]
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

bind(*signature: inspect.Signature, params: Union[list, dict, None]*) → inspect.BoundArguments
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

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: Callable, params: Union[list, dict, None], exclude: Iterable[str] = (), **kwargs*) → Dict[str, Any]

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: bool = True, **config_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

validate_method(*method: Callable, params: Union[list, dict, None], exclude: Iterable[str] = (), **kwargs*) → Dict[str, Any]

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

build_validation_schema

Builds pydantic model based validation schema from method signature.

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

Returns validation schema

Specification

class pjrpc.server.specs.**JSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Schema JSON encoder.

default(*o: Any*) → Any

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.specs.**BaseUI**

Base UI.

get_static_folder() → str

Returns ui statics folder.

get_index_page(*spec_url: str*) → str

Returns ui index webpage.

Parameters **spec_url** – specification url.

class pjrpc.server.specs.**Specification**(*path: str = '/spec.json'*, *ui: Optional[pjrpc.server.specs.BaseUI] = None*, *ui_path: Optional[str] = None*)

JSON-RPC specification.

Parameters

- **path** – specification url path suffix
- **ui** – specification ui instance
- **ui_path** – specification ui url path suffix

path

Returns specification url path.

ui

Returns ui instance.

ui_path

Returns specification ui url path.

schema(*path: str*, *methods: Iterable[pjrpc.server.dispatcher.Method] = ()*, *methods_map: Dict[str, Iterable[pjrpc.server.dispatcher.Method]] = {}*) → dict

Returns specification schema.

Parameters

- **path** – methods endpoint path
- **methods** – methods list the specification is generated for
- **methods_map** – methods map the specification is generated for. Each item is a mapping from a prefix to methods on which the methods will be served

extractors

class pjrpc.server.specs.extractors.**Schema**(*schema: Dict[str, Any], required: bool = True, summary: str = UNSET, description: str = UNSET, deprecated: bool = UNSET, definitions: Dict[str, Any] = UNSET*)

Method parameter/result schema.

class pjrpc.server.specs.extractors.**Example**(*params: Dict[str, Any], result: Any, version: str = '2.0', summary: str = UNSET, description: str = UNSET*)

Method usage example.

class pjrpc.server.specs.extractors.**Tag**(*name: str, description: str = UNSET, externalDocs: str = UNSET*)

A list of method tags.

class pjrpc.server.specs.extractors.**Error**(*code: int, message: str, data: Dict[str, Any] = UNSET*)

Defines an application level error.

class pjrpc.server.specs.extractors.**BaseSchemaExtractor**

Base method schema extractor.

extract_params_schema(*method: Callable, exclude: Iterable[str] = ()*) → Dict[str, pjrpc.server.specs.extractors.Schema]

Extracts method parameters schema.

extract_result_schema(*method: Callable*) → pjrpc.server.specs.extractors.Schema

Extracts method result schema.

extract_description(*method: Callable*) → Union[pjrpc.common.common.UnsetType, str]

Extracts method description.

extract_summary(*method: Callable*) → Union[pjrpc.common.common.UnsetType, str]

Extracts method summary.

extract_errors_schema(*method: Callable*) → Union[pjrpc.common.common.UnsetType, List[pjrpc.server.specs.extractors.Error]]

Extracts method errors schema.

extract_tags(*method: Callable*) → Union[pjrpc.common.common.UnsetType, List[pjrpc.server.specs.extractors.Tag]]

Extracts method tags.

extract_examples(*method: Callable*) → Union[pjrpc.common.common.UnsetType, List[pjrpc.server.specs.extractors.Example]]

Extracts method usage examples.

extract_deprecation_status(*method: Callable*) → Union[pjrpc.common.common.UnsetType, bool]

Extracts method deprecation status.

class pjrpc.server.specs.extractors.pydantic.**PydanticSchemaExtractor**(*ref_template: str = '#/components/schemas/{model}'*)

Pydantic method specification extractor.

extract_params_schema(*method: Callable, exclude: Iterable[str] = ()*) → Dict[str, pjrpc.server.specs.extractors.Schema]

Extracts method parameters schema.

extract_result_schema(*method: Callable*) → pjrpc.server.specs.extractors.Schema
Extracts method result schema.

schemas

OpenAPI Specification generator. See https://swagger.io/specification/.

class pjrpc.server.specs.openapi.**Contact**(*name: str = UNSET*, *url: str = UNSET*, *email: str = UNSET*)
Contact information for the exposed API.

Parameters

- **name** – the identifying name of the contact person/organization
- **url** – the URL pointing to the contact information
- **email** – the email address of the contact person/organization

class pjrpc.server.specs.openapi.**License**(*name: str*, *url: str = UNSET*)
License information for the exposed API.

Parameters

- **name** – the license name used for the API
- **url** – a URL to the license used for the API

class pjrpc.server.specs.openapi.**Info**(*title: str*, *version: str*, *description: str = UNSET*, *contact: pjrpc.server.specs.openapi.Contact = UNSET*, *license: pjrpc.server.specs.openapi.License = UNSET*, *termsOfService: str = UNSET*)
Metadata about the API.

Parameters

- **title** – the title of the application
- **version** – the version of the OpenAPI document
- **description** – a short description of the application
- **contact** – the contact information for the exposed API
- **license** – the license information for the exposed API
- **termsOfService** – a URL to the Terms of Service for the API

class pjrpc.server.specs.openapi.**ServerVariable**(*default: str*, *enum: List[str] = UNSET*, *description: str = UNSET*)
An object representing a Server Variable for server URL template substitution.

Parameters

- **default** – the default value to use for substitution, which SHALL be sent if an alternate value is not supplied
- **enum** – an enumeration of string values to be used if the substitution options are from a limited set
- **description** – an optional description for the server variable

class pjrpc.server.specs.openapi.**Server**(*url: str*, *description: str = UNSET*, *variables: Dict[str, pjrpc.server.specs.openapi.ServerVariable] = UNSET*)

Connectivity information of a target server.

Parameters

- **url** – a URL to the target host
- **description** – an optional string describing the host designated by the URL

class pjrpc.server.specs.openapi.**ExternalDocumentation**(*url: str*, *description: str = UNSET*)

Allows referencing an external resource for extended documentation.

Parameters

- **url** – a short description of the target documentation.
- **description** – the URL for the target documentation

class pjrpc.server.specs.openapi.**Tag**(*name: str*, *description: str = UNSET*, *externalDocs: pjrpc.server.specs.openapi.ExternalDocumentation = UNSET*)

A list of tags for API documentation control. Tags can be used for logical grouping of methods by resources or any other qualifier.

Parameters

- **name** – the name of the tag
- **externalDocs** – additional external documentation for this tag
- **description** – a short description for the tag

class pjrpc.server.specs.openapi.**SecuritySchemeType**

The type of the security scheme.

APIKEY = 'apiKey'

HTTP = 'http'

OAUTH2 = 'oauth2'

OPENID_CONNECT = 'openIdConnect'

class pjrpc.server.specs.openapi.**ApiKeyLocation**

The location of the API key.

QUERY = 'query'

HEADER = 'header'

COOKIE = 'cookie'

class pjrpc.server.specs.openapi.**OAuthFlow**(*authorizationUrl: str, tokenUrl: str, scopes: Dict[str, str], refreshUrl: str = UNSET*)

Configuration details for a supported OAuth Flow.

Parameters

- **authorizationUrl** – the authorization URL to be used for this flow
- **tokenUrl** – the token URL to be used for this flow
- **refreshUrl** – the URL to be used for obtaining refresh tokens

- **scopes** – the available scopes for the OAuth2 security scheme

class pjrpc.server.specs.openapi.**OAuthFlows**(*implicit: pjrpc.server.specs.openapi.OAuthFlow = UNSET, password: pjrpc.server.specs.openapi.OAuthFlow = UNSET, clientCredentials: pjrpc.server.specs.openapi.OAuthFlow = UNSET, authorizationCode: pjrpc.server.specs.openapi.OAuthFlow = UNSET*)

Configuration of the supported OAuth Flows.

Parameters

- **implicit** – configuration for the OAuth Implicit flow
- **password** – configuration for the OAuth Resource Owner Password flow
- **clientCredentials** – configuration for the OAuth Client Credentials flow
- **authorizationCode** – configuration for the OAuth Authorization Code flow

class pjrpc.server.specs.openapi.**SecurityScheme**(*type: pjrpc.server.specs.openapi.SecuritySchemeType, scheme: str, name: str = UNSET, location: pjrpc.server.specs.openapi.ApiKeyLocation = UNSET, bearerFormat: str = UNSET, flows: pjrpc.server.specs.openapi.OAuthFlows = UNSET, openIdConnectUrl: str = UNSET, description: str = UNSET*)

Defines a security scheme that can be used by the operations.

Parameters

- **type** – the type of the security scheme
- **name** – the name of the header, query or cookie parameter to be used
- **location** – the location of the API key
- **scheme** – the name of the HTTP Authorization scheme to be used in the Authorization header
- **bearerFormat** – a hint to the client to identify how the bearer token is formatted
- **flows** – an object containing configuration information for the flow types supported
- **openIdConnectUrl** –
- **description** – a short description for security scheme

class pjrpc.server.specs.openapi.**Components**(*securitySchemes: Dict[str, pjrpc.server.specs.openapi.SecurityScheme] = UNSET, schemas: Dict[str, Dict[str, Any]] = <factory>*)

Holds a set of reusable objects for different aspects of the OAS.

Parameters

- **securitySchemes** – an object to hold reusable Security Scheme Objects
- **schemas** – the definition of input and output data types

class pjrpc.server.specs.openapi.**Error**(*code: int*, *message: str*, *data: Dict[str, Any] = UNSET*)

Defines an application level error.

Parameters

- **code** – a Number that indicates the error type that occurred
- **message** – a String providing a short description of the error
- **data** – a Primitive or Structured value that contains additional information about the error

class pjrpc.server.specs.openapi.**MethodExample**(*params: Dict[str, Any]*, *result: Any*, *version: str = '2.0'*, *summary: str = UNSET*, *description: str = UNSET*)

Method usage example.

Parameters

- **params** – example parameters
- **result** – example result
- **name** – name for the example pairing
- **summary** – short description for the example pairing
- **description** – a verbose explanation of the example pairing

class pjrpc.server.specs.openapi.**ExampleObject**(*value: Any*, *summary: str = UNSET*, *description: str = UNSET*, *externalValue: str = UNSET*)

Method usage example.

Parameters

- **value** – embedded literal example
- **summary** – short description for the example.
- **description** – long description for the example
- **externalValue** – a URL that points to the literal example

class pjrpc.server.specs.openapi.**MediaType**(*schema: Dict[str, Any]*, *examples: Dict[str, pjrpc.server.specs.openapi.ExampleObject] = UNSET*)

Each Media Type Object provides schema and examples for the media type identified by its key.

Parameters

- **schema** – the schema defining the content.
- **example** – example of the media type

class pjrpc.server.specs.openapi.**Response**(*description: str*, *content: Dict[str, pjrpc.server.specs.openapi.MediaType] = UNSET*)

A container for the expected responses of an operation.

Parameters

- **description** – a short description of the response
- **content** – a map containing descriptions of potential response payloads

class pjrpc.server.specs.openapi.**RequestBody**(*content: Dict[str, pjrpc.server.specs.openapi.MediaType], required: bool = UNSET, description: str = UNSET*)

Describes a single request body.

Parameters

- **content** – the content of the request body
- **required** – determines if the request body is required in the request
- **description** – a brief description of the request body

class pjrpc.server.specs.openapi.**ParameterLocation**

The location of the parameter.

QUERY = 'query'

HEADER = 'header'

PATH = 'path'

COOKIE = 'cookie'

class pjrpc.server.specs.openapi.**StyleType**

Describes how the parameter value will be serialized depending on the type of the parameter value.

MATRIX = 'matrix'

LABEL = 'label'

FORM = 'form'

SIMPLE = 'simple'

SPACE_DELIMITED = 'spaceDelimited'

PIPE_DELIMITED = 'pipeDelimited'

DEEP_OBJECT = 'deepObject'

class pjrpc.server.specs.openapi.**Parameter**(*name: str, location: pjrpc.server.specs.openapi.ParameterLocation, description: str = UNSET, required: bool = UNSET, deprecated: bool = UNSET, allowEmptyValue: bool = UNSET, style: pjrpc.server.specs.openapi.StyleType = UNSET, explode: bool = UNSET, allowReserved: bool = UNSET, schema: Dict[str, Any] = UNSET, examples: Dict[str, pjrpc.server.specs.openapi.ExampleObject] = UNSET, content: Dict[str, pjrpc.server.specs.openapi.MediaType] = UNSET*)

Describes a single operation parameter.

Parameters

- **name** – the name of the parameter
- **location** – the location of the parameter
- **description** – a brief description of the parameter
- **required** – determines whether this parameter is mandatory

- **deprecated** – a parameter is deprecated and SHOULD be transitioned out of usage
- **allowEmptyValue** – the ability to pass empty-valued parameters
- **style** – describes how the parameter value will be serialized depending on the type of the parameter value
- **explode** – when this is true, parameter values of type array or object generate separate parameters for each value of the array or key-value pair of the map
- **allowReserved** – determines whether the parameter value SHOULD allow reserved characters, as defined by RFC3986 :/?#[]@!$&'()*+,;= to be included without percent-encoding
- **schema** – the schema defining the type used for the parameter.
- **examples** – examples of the parameter's potential value
- **content** – a map containing the representations for the parameter

class pjrpc.server.specs.openapi.**Operation**(*responses: Dict[str, pjrpc.server.specs.openapi.Response], requestBody: pjrpc.server.specs.openapi.RequestBody = UNSET, tags: List[str] = UNSET, summary: str = UNSET, description: str = UNSET, externalDocs: pjrpc.server.specs.openapi.ExternalDocumentation = UNSET, deprecated: bool = UNSET, servers: List[pjrpc.server.specs.openapi.Server] = UNSET, security: List[Dict[str, List[str]]] = UNSET, parameters: List[pjrpc.server.specs.openapi.Parameter] = UNSET*)

Describes a single API operation on a path.

Parameters

- **tags** – a list of tags for API documentation control
- **summary** – a short summary of what the operation does
- **description** – a verbose explanation of the operation behavior
- **externalDocs** – additional external documentation for this operation
- **requestBody** – the request body applicable for this operation
- **responses** – the list of possible responses as they are returned from executing this operation
- **deprecated** – declares this operation to be deprecated
- **servers** – an alternative server array to service this operation
- **security** – a declaration of which security mechanisms can be used for this operation

class pjrpc.server.specs.openapi.**Path**(*get: pjrpc.server.specs.openapi.Operation = UNSET, put: pjrpc.server.specs.openapi.Operation = UNSET, post: pjrpc.server.specs.openapi.Operation = UNSET, delete: pjrpc.server.specs.openapi.Operation = UNSET, options: pjrpc.server.specs.openapi.Operation = UNSET, head: pjrpc.server.specs.openapi.Operation = UNSET, patch: pjrpc.server.specs.openapi.Operation = UNSET, trace: pjrpc.server.specs.openapi.Operation = UNSET, summary: str = UNSET, description: str = UNSET, servers: List[pjrpc.server.specs.openapi.Server] = UNSET*)

Describes the interface for the given method name.

Parameters

- **summary** – an optional, string summary, intended to apply to all operations in this path
- **description** – an optional, string description, intended to apply to all operations in this path
- **servers** – an alternative server array to service all operations in this path

pjrpc.server.specs.openapi.**annotate**(*params_schema: Dict[str, pjrpc.server.specs.extractors.Schema] = UNSET, result_schema: pjrpc.server.specs.extractors.Schema = UNSET, errors: List[Union[pjrpc.server.specs.openapi.Error, Type[pjrpc.common.exceptions.JsonRpcError]]] = UNSET, examples: List[pjrpc.server.specs.openapi.MethodExample] = UNSET, tags: List[str] = UNSET, summary: str = UNSET, description: str = UNSET, external_docs: pjrpc.server.specs.openapi.ExternalDocumentation = UNSET, deprecated: bool = UNSET, security: List[Dict[str, List[str]]] = UNSET, parameters: List[pjrpc.server.specs.openapi.Parameter] = UNSET*)

Adds Open Api specification annotation to the method.

Parameters

- **params_schema** – method parameters JSON schema
- **result_schema** – method result JSON schema
- **errors** – method errors
- **examples** – method usage examples
- **tags** – a list of tags for method documentation control
- **summary** – a short summary of what the method does
- **description** – a verbose explanation of the method behavior
- **external_docs** – an external resource for extended documentation
- **deprecated** – declares this method to be deprecated
- **security** – a declaration of which security mechanisms can be used for the method
- **parameters** – a list of parameters that are applicable for the method

class pjrpc.server.specs.openapi.**OpenAPI**(*info: pjrpc.server.specs.openapi.Info*, *path: str = '/openapi.json'*, *servers: List[pjrpc.server.specs.openapi.Server] = UNSET*, *external_docs: Optional[pjrpc.server.specs.openapi.ExternalDocumentation] = UNSET*, *tags: List[pjrpc.server.specs.openapi.Tag] = UNSET*, *security: List[Dict[str, List[str]]] = UNSET*, *security_schemes: Dict[str, pjrpc.server.specs.openapi.SecurityScheme] = UNSET*, *openapi: str = '3.0.0'*, *schema_extractor: Optional[pjrpc.server.specs.extractors.BaseSchemaExtractor] = None*, *ui: Optional[pjrpc.server.specs.BaseUI] = None*, *ui_path: str = '/ui/'*)

OpenAPI Specification.

Parameters

- **info** – provides metadata about the API
- **servers** – an array of Server Objects, which provide connectivity information to a target server
- **external_docs** – additional external documentation
- **openapi** – the semantic version number of the OpenAPI Specification version that the OpenAPI document uses
- **tags** – a list of tags used by the specification with additional metadata
- **security** – a declaration of which security mechanisms can be used across the API
- **schema_extractor** – method specification extractor
- **path** – specification url path
- **security_schemes** – an object to hold reusable Security Scheme Objects
- **ui** – web ui instance
- **ui_path** – wet ui path

schema(*path: str*, *methods: Iterable[pjrpc.server.dispatcher.Method] = ()*, *methods_map: Dict[str, Iterable[pjrpc.server.dispatcher.Method]] = {}*) → dict

Returns specification schema.

Parameters

- **path** – methods endpoint path
- **methods** – methods list the specification is generated for
- **methods_map** – methods map the specification is generated for. Each item is a mapping from a prefix to methods on which the methods will be served

class pjrpc.server.specs.openapi.**SwaggerUI**(***configs*)

Swagger UI.

Parameters **config** – documentation configurations (see https://github.com/swagger-api/swagger-ui/blob/master/docs/usage/configuration.md).

get_static_folder() → str

Returns ui statics folder.

class pjrpc.server.specs.openapi.**RapiDoc**(***configs*)
RapiDoc UI.

Parameters **config** – documentation configurations (see https://mrin9.github.io/RapiDoc/api.html). Be aware that configuration parameters should be in snake case, for example: parameter *heading-text* should be passed as *heading_text*)

get_static_folder() → str
Returns ui statics folder.

class pjrpc.server.specs.openapi.**ReDoc**(***configs*)
ReDoc UI.

Parameters **config** – documentation configurations (see https://github.com/Redocly/redoc#configuration). Be aware that configuration parameters should be in snake case, for example: parameter *heading-text* should be passed as *heading_text*)

get_static_folder() → str
Returns ui statics folder.

OpenRPC specification generator. See https://spec.open-rpc.org/.

class pjrpc.server.specs.openrpc.**Contact**(*name: str = UNSET*, *url: str = UNSET*, *email: str = UNSET*)
Contact information for the exposed API.

Parameters

- **name** – the identifying name of the contact person/organization
- **url** – the URL pointing to the contact information
- **email** – the email address of the contact person/organization

class pjrpc.server.specs.openrpc.**License**(*name: str*, *url: str = UNSET*)
License information for the exposed API.

Parameters

- **name** – the license name used for the API
- **url** – a URL to the license used for the API

class pjrpc.server.specs.openrpc.**Info**(*title: str*, *version: str*, *description: str = UNSET*, *contact: pjrpc.server.specs.openrpc.Contact = UNSET*, *license: pjrpc.server.specs.openrpc.License = UNSET*, *termsOfService: str = UNSET*)
Metadata about the API.

Parameters

- **title** – the title of the application
- **version** – the version of the OpenRPC document
- **description** – a verbose description of the application
- **contact** – the contact information for the exposed API
- **license** – the license information for the exposed API
- **termsOfService** – a URL to the Terms of Service for the API

class pjrpc.server.specs.openrpc.**Server**(*name: str*, *url: str*, *summary: str = UNSET*, *description: str = UNSET*)
Connectivity information of a target server.

Parameters

- **name** – a name to be used as the canonical name for the server.
- **url** – a URL to the target host
- **summary** – a short summary of what the server is
- **description** – an optional string describing the host designated by the URL

class pjrpc.server.specs.openrpc.**ExternalDocumentation**(*url: str*, *description: str = UNSET*)

Allows referencing an external resource for extended documentation.

Parameters

- **url** – A verbose explanation of the target documentation
- **description** – The URL for the target documentation. Value MUST be in the format of a URL

class pjrpc.server.specs.openrpc.**Tag**(*name: str*, *summary: str = UNSET*, *description: str = UNSET*, *externalDocs: pjrpc.server.specs.openrpc.ExternalDocumentation = UNSET*)

A list of tags for API documentation control. Tags can be used for logical grouping of methods by resources or any other qualifier.

Parameters

- **name** – the name of the tag
- **summary** – a short summary of the tag
- **description** – a verbose explanation for the tag
- **externalDocs** – additional external documentation for this tag

class pjrpc.server.specs.openrpc.**ExampleObject**(*value: Union[str, int, float, dict, bool, list, tuple, set, None]*, *name: str*, *summary: str = UNSET*, *description: str = UNSET*)

The ExampleObject object is an object the defines an example.

Parameters

- **value** – embedded literal example
- **name** – canonical name of the example
- **summary** – short description for the example
- **description** – a verbose explanation of the example

class pjrpc.server.specs.openrpc.**MethodExample**(*name: str*, *params: List[pjrpc.server.specs.openrpc.ExampleObject]*, *result: pjrpc.server.specs.openrpc.ExampleObject*, *summary: str = UNSET*, *description: str = UNSET*)

The example Pairing object consists of a set of example params and result.

Parameters

- **params** – example parameters
- **result** – example result

- **name** – name for the example pairing
- **summary** – short description for the example pairing
- **description** – a verbose explanation of the example pairing

class pjrpc.server.specs.openrpc.**ContentDescriptor**(*name: str, schema: Dict[str, Any], summary: str = UNSET, description: str = UNSET, required: bool = UNSET, deprecated: bool = UNSET*)

Content Descriptors are objects that describe content. They are reusable ways of describing either parameters or result.

Parameters

- **name** – name of the content that is being described
- **schema** – schema that describes the content. The Schema Objects MUST follow the specifications outline in the JSON Schema Specification 7 (https://json-schema.org/draft-07/json-schema-release-notes.html)
- **summary** – a short summary of the content that is being described
- **description** – a verbose explanation of the content descriptor behavior
- **required** – determines if the content is a required field
- **deprecated** – specifies that the content is deprecated and SHOULD be transitioned out of usage

class pjrpc.server.specs.openrpc.**Error**(*code: int, message: str, data: Dict[str, Any] = UNSET*)

Defines an application level error.

Parameters

- **code** – a Number that indicates the error type that occurred
- **message** – a String providing a short description of the error
- **data** – a Primitive or Structured value that contains additional information about the error

class pjrpc.server.specs.openrpc.**ParamStructure**

The expected format of the parameters.

BY_NAME = 'by-name'

BY_POSITION = 'by-position'

EITHER = 'either'

class pjrpc.server.specs.openrpc.**MethodInfo**(*name: str, params: List[Union[pjrpc.server.specs.openrpc.ContentDescriptor, dict]], result: Union[pjrpc.server.specs.openrpc.ContentDescriptor, dict], errors: List[pjrpc.server.specs.openrpc.Error] = UNSET, paramStructure: pjrpc.server.specs.openrpc.ParamStructure = UNSET, examples: List[pjrpc.server.specs.openrpc.MethodExample] = UNSET, summary: str = UNSET, description: str = UNSET, tags: List[pjrpc.server.specs.openrpc.Tag] = UNSET, deprecated: bool = UNSET, externalDocs: pjrpc.server.specs.openrpc.ExternalDocumentation = UNSET, servers: List[pjrpc.server.specs.openrpc.Server] = UNSET*)

Describes the interface for the given method name.

Parameters

- **name** – the canonical name for the method
- **params** – a list of parameters that are applicable for this method
- **result** – the description of the result returned by the method
- **errors** – a list of custom application defined errors that MAY be returned
- **examples** – method usage examples
- **summary** – a short summary of what the method does
- **description** – a verbose explanation of the method behavior
- **tags** – a list of tags for API documentation control
- **deprecated** – declares this method to be deprecated
- **paramStructure** – the expected format of the parameters
- **externalDocs** – additional external documentation for this method
- **servers** – an alternative servers array to service this method

class pjrpc.server.specs.openrpc.**Components**(*schemas: Dict[str, Any] = <factory>*)

Set of reusable objects for different aspects of the OpenRPC.

Parameters **schemas** – reusable Schema Objects

pjrpc.server.specs.openrpc.**annotate**(*params_schema: List[pjrpc.server.specs.openrpc.ContentDescriptor] = UNSET, result_schema: pjrpc.server.specs.openrpc.ContentDescriptor = UNSET, errors: List[Union[pjrpc.server.specs.openrpc.Error, Type[pjrpc.common.exceptions.JsonRpcError]]] = UNSET, examples: List[pjrpc.server.specs.openrpc.MethodExample] = UNSET, summary: str = UNSET, description: str = UNSET, tags: List[Union[pjrpc.server.specs.openrpc.Tag, str]] = UNSET, deprecated: bool = UNSET*)

Adds JSON-RPC method to the API specification.

Parameters

- **params_schema** – a list of parameters that are applicable for this method
- **result_schema** – the description of the result returned by the method
- **errors** – a list of custom application defined errors that MAY be returned
- **examples** – method usage example
- **summary** – a short summary of what the method does
- **description** – a verbose explanation of the method behavior
- **tags** – a list of tags for API documentation control
- **deprecated** – declares this method to be deprecated

class pjrpc.server.specs.openrpc.**OpenRPC**(*info: pjrpc.server.specs.openrpc.Info*, *path: str = '/openrpc.json'*, *servers: List[pjrpc.server.specs.openrpc.Server] = UNSET*, *external_docs: Optional[pjrpc.server.specs.openrpc.ExternalDocumentation] = UNSET*, *openrpc: str = '1.0.0'*, *schema_extractor: Optional[pjrpc.server.specs.extractors.BaseSchemaExtractor] = None*)

OpenRPC Specification.

Parameters

- **info** – specification information
- **path** – specification url path
- **servers** – connectivity information
- **external_docs** – additional external documentation
- **openrpc** – the semantic version number of the OpenRPC Specification version that the OpenRPC document uses
- **schema_extractor** – method specification extractor

schema(*path: str*, *methods: Iterable[pjrpc.server.dispatcher.Method] = ()*, *methods_map: Dict[str, Iterable[pjrpc.server.dispatcher.Method]] = {}*) → dict

Returns specification schema.

Parameters

- **path** – methods endpoint path
- **methods** – methods list the specification is generated for
- **methods_map** – methods map the specification is generated for. Each item is a mapping from a prefix to methods on which the methods will be served

CHAPTER 4

Development

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

Python Module Index

p

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V