# django-modern-rpc

***Release 2.1.0***

**Antoine Lorence**

**Jun 05, 2026**

# CONTENTS

RPC (Remote Procedure Call) is a network protocol used to call functions on another system or web server through HTTP POST requests. It has been around for decades and is one of the predecessors of modern Web API protocols (REST, GraphQL, etc.).

While newer alternatives exist, there are still use-cases where XML-RPC or JSON-RPC servers are needed. **Django-modern-rpc** helps you set up such a server as part of your Django project. It provides a simple and pythonic API to expose global functions to be called from other websites or programs.

Version 2.0 brings significant improvements to the library, including a more intuitive API, better error handling, reworked authentication system, improved type annotations, and more flexible configuration options.

CHAPTER

ONE

# GETTING STARTED

**Important**

django-modern-rpc requires Python 3.10+ and Django 4.2+. If you need to install it in environment with older Python / Django versions, fallback to a previous release. See *Changelog* for more information.

Installing the library and configuring a Django project to use it can be achieved in a few minutes. Follow *Quickstart* for a very basic setup process. Later, when you need to configure your project more precisely, follow the other topics in the menu.

## 1.1 Quickstart

### 1.1.1 Installation

Install `django-modern-rpc` in your environment

```
pip install django-modern-rpc
```

```
poetry add django-modern-rpc
```

```
uv add django-modern-rpc
```

### 1.1.2 Basic setup

Create an `RpcServer` instance and register your first procedure

Listing 1: myproject/myapp/rpc.py

```
from modernrpc.server import RpcServer

server = RpcServer()

@server.register_procedure
def add(a: int, b: int) -> int:
    """Add two numbers and return the result.
    :param a: First number
    :param b: Second number
    :return: Sum of a and b
```

(continues on next page)

(continued from previous page)

```
    """
    return a + b
```

Remote procedures are Python functions decorated with the server's `register_procedure` decorator. Both server and procedure registration can be customized. See *Procedures registration*.

### 1.1.3 Serve the procedures

To execute a remote procedure, clients will send a *POST* request to a single url in your project. Declare the route to this view in your project's `urls.py`

Listing 2: myproject/myproject/urls.py

```
from django.urls import path
from myapp.rpc import server

urlpatterns = [
    # ... other url patterns
    path('rpc/', server.view),  # Synchronous view
]
```

The server's view is already configured with CSRF exemption and POST-only restrictions.

#### Async Support

For Django projects using ASGI and async views, you can use the async version of the view:

Listing 3: myproject/myproject/urls.py (with async support)

```
from django.urls import path
from myapp.rpc import server

urlpatterns = [
    # ... other url patterns
    path('rpc/', server.async_view),  # Asynchronous view
]
```

The async view provides the same functionality as the synchronous view but can be used in an async context, allowing your Django application to handle other requests while waiting for RPC operations to complete.

### 1.1.4 Test the server

Start your project using `python manage.py runserver` and call your procedure using a JSON-RPC or XML-RPC client, or directly with your favourite HTTP client

Listing 4: JSON-RPC example

```
~$ curl -X POST localhost:8000/rpc/ -H "Content-Type: application/json" -d '{"id": 1,
↪"method": "system.listMethods", "jsonrpc": "2.0"}'
{"id": 1, "jsonrpc": "2.0", "result": ["add", "system.listMethods", "system.methodHelp",
↪"system.methodSignature"]}

~$ curl -X POST localhost:8000/rpc/ -H "Content-Type: application/json" -d '{"id": 2,
```

(continues on next page)

(continued from previous page)

```
↪"method": "add", "params": [5, 9], "jsonrpc": "2.0"}'
{"id": 2, "jsonrpc": "2.0", "result": 14}
```

Listing 5: XML-RPC example

```
from xmlrpc.client import ServerProxy

with ServerProxy("http://localhost:8000/rpc/") as proxy:
    proxy.system.listMethods()
    proxy.add(5, 9)

# ['add', 'system.listMethods', 'system.methodHelp', 'system.methodSignature', 'system.
↪multicall']
# 14
```

## 1.2 Server / Namespaces

### 1.2.1 Server

The `RpcServer` class is the central component that handles remote procedure calls.

At least one instance must be created.

Listing 6: myproject/myapp/rpc.py

```
from modernrpc.server import RpcServer

server = RpcServer()
```

A server instance exposes a view that must be added to the Django routing system

Listing 7: myproject/myproject/urls.py

```
from django.urls import path
from myapp.rpc import server

urlpatterns = [
    # ... other url patterns
    path('rpc/', server.view),  # Synchronous view
]
```

Then, all requests to `http://yourwebsite/rpc/` will be routed to the server's view. They will be inspected and parsed to be interpreted as RPC calls.

If a *POST* request has a correct *Content-Type* header and a well-formed body, it will be handled by the right XML-RPC or JSON-RPC backend. The result of procedure calls will be encapsulated into a well-formed response, according to the request protocol (XML or JSON-RPC)

**Note**

You are free to choose the path that will be used to handle your remote procedure calls, but `/rpc` or `/RPC2` are the most commonly used paths (example in xmlrpc.server docs)

### Protocol restriction

Using the `supported_protocol` argument, you can configure a given server to handle only JSON-RPC or XML-RPC requests. This can be used to set up protocol-specific servers.

Default: `supported_protocol = Protocol.ALL`

Listing 8: myapp/rpc.py

```
from modernrpc import Protocol, RpcServer

# Create protocol-specific servers
json_server = RpcServer(supported_protocol=Protocol.JSON_RPC)
xml_server = RpcServer(supported_protocol=Protocol.XML_RPC)
```

Listing 9: urls.py

```
from django.urls import path
from myapp.rpc import json_server, xml_server

urlpatterns = [
    path('json-rpc/', json_server.view),
    path('xml-rpc/', xml_server.view),
]
```

### System procedures

By default, 3 introspection procedures (+ 1 multicall procedure, only for XML-RPC requests) are registered by a server, under the namespace `system`. See *Introspection procedures* for history and protocol specific details, as well as *Introspection procedures & multicall* for JSON-RPC specific implementation.

`modernrpc.system_procedures.__system_list_methods`(*_ctx*)

Returns a list of all procedures exposed by the server

**Parameters**
**_ctx** (*RpcRequestContext*)

`modernrpc.system_procedures.__system_method_signature`(*method_name*, *_ctx*)

Returns an array describing the possible signatures for the given procedure.

Currently, this procedure only returns one possible signature, so the result is a list of 1 list.

The inner list contains:

- Return type as first element
- Types of arguments from element 1 to N

**Parameters**

- **method_name** (*str*) – Name of the procedure
- **_ctx** (*RpcRequestContext*) – Request context for this call

**Returns**
An array of arrays describing types of return values and method arguments

`modernrpc.system_procedures.__system_method_help`(*method_name*, *_ctx*)

Returns the documentation of the given procedure.

**Parameters**

- **method_name** (*str*) – Name of the procedure
- **_ctx** (*RpcRequestContext*) – Request context for this call

**Returns**

Documentation text for the procedure

modernrpc.system_procedures.**__system_multicall**(*calls*, *_ctx*)

Call multiple procedure at once.

**Parameters**

- **calls** (*list*) – An array of struct like {"methodName": string, "params": [..., ...]}
- **_ctx** (*RpcRequestContext*) – Request context for this call

**Returns**

An array containing the result of each procedure call

**Note**

The *system.multicall* builtin method is registered as a synchronous version by default. In this version, each procedure is executed sequentially. If you want to opt in for the asynchronous version (procedures executed concurrently using `asyncio.gather()`), set `settings.MODERNRPC_XMLRPC_ASYNC_MULTICALL = True`.

Using the `register_system_procedures` argument, you can completely disable their automatic registration.

Default: `register_system_procedures = True`

Listing 10: myproject/myapp/rpc.py

```
from modernrpc.server import RpcServer

server = RpcServer(register_system_procedures=False)

# server won't register the system.* procedures
```

**Warning**

Disabling the system procedure registration may prevent some clients (in particular, XML-RPC ones) from sending requests to your server. Use at your own risk.

### Authentication

Listing 11: myapp/rpc.py

```
from myapp.auth import my_auth_callback
from modernrpc.server import RpcServer

# Create a server with authentication
server = RpcServer(auth=my_auth_callback)

@server.register_procedure
```

(continues on next page)

(continued from previous page)

```
def multiply(a, b):
    return a * b
```

All procedures registered in the server will use the auth callback configured in the server.

> **Note**
>
> Configured authentication callback can be overridden at namespace or procedure level.

For more information about authentication, see *Authentication*.

#### GET requests redirection

By default, when the server receives a request with a non-POST method, a "Method Not Allowed (405)" response is returned. Sometimes, you may need to allow GET requests to return a page, for example to display some documentation about the RPC server.

For that use case, the server can be configured with the `redirect_get_request_to` argument. It accepts any value accepted by `django.shortcuts.redirect` function (see official redirect() docs). When configured, a permanent redirection to the corresponding location will be returned on GET requests.

### 1.2.2 Namespace

Namespaces allow you to organize related RPC procedures under a common prefix. This is useful for:

- Grouping related procedures together
- Avoiding name conflicts between procedures
- Providing a clearer API structure

#### Creating a namespace

To create a namespace, instantiate the `RpcNamespace` class:

Listing 12: myapp/math.py

```
from modernrpc import RpcNamespace

# Create a namespace for math procedures
math = RpcNamespace()
```

#### Registering procedures

You can register procedures to a namespace using the `register_procedure` method, similar to how you would with an `RpcServer`:

Listing 13: myapp/math.py

```
@math.register_procedure
def add(a, b):
    return a + b

@math.register_procedure
```

(continues on next page)

(continued from previous page)

```
def subtract(a, b):
    return a - b
```

See *Customize registration* for a list of available customization options.

### Registering a namespace to a server

To make the procedures in a namespace available through your RPC server, register the namespace to the server:

Listing 14: myapp/rpc.py

```
from modernrpc.server import RpcServer
from myapp.math import math

server = RpcServer()
server.register_namespace(math, "math")
```

This will make the procedures available under the prefix "math.", so clients can call them as *math.add* and *math.subtract*.

If you don't provide a name when registering a namespace, the procedures will be registered without a prefix:

Listing 15: myapp/rpc.py

```
# Register without a prefix
server.register_namespace(math)

# Procedures are available as "add" and "subtract"
```

### Authentication

Namespaces can have their own authentication settings that override the server's settings:

Listing 16: myapp/math.py

```
# Create a namespace with authentication
secure_math = RpcNamespace(auth=my_auth_callback)

@secure_math.register_procedure
def multiply(a, b):
    return a * b
```

All procedures registered in the namespace will use the auth callback configured in the namespace.

> **Note**
>
> Configured authentication callback can be overridden at procedure level.

For more information about authentication, see *Authentication*.

### 1.2.3 Multiple servers definition

You can create multiple servers.

Listing 17: myapp/rpc.py

```
from modernrpc.server import RpcServer

# Create multiple server instances
api_v1 = RpcServer()
api_v2 = RpcServer()
```

Listing 18: urls.py

```
from django.urls import path
from myapp.rpc import api_v1, api_v2

urlpatterns = [
   path('api/v1/', api_v1.view),
   path('api/v2/', api_v2.view),
]
```

When multiple servers are defined, each server can register its own procedures. This is useful to have different functions performing the same task under the same name, but from a different path (a.k.a multiple API versions).

If needed, a procedure can be registered into multiple servers. This can be used to avoid code duplication when a specific procedure should run the same code for different paths.

### 1.2.4 Sync and async views

RpcServer exposes two HTTP entry points:

- view: a regular (synchronous) Django view callable
- async_view: an asynchronous Django view (a coroutine function)

They are feature-equivalent. Routing, serializers/deserializers selection, authentication/authorization, error handling, introspection methods and namespaces all behave exactly the same in both cases. The only difference is the execution model of the view itself.

When should you use async_view?

- If your deployment stack is ASGI-based (e.g., Daphne, Uvicorn, Hypercorn) and your RPC procedures contain async code, async_view lets Django run the request handler as a coroutine. Multiple in-flight RPC calls can then await I/O concurrently, which can improve throughput and tail latency under I/O-bound workloads.
- If your procedures are all synchronous or you run under a purely WSGI stack, using view is perfectly fine; async_view will not provide benefits in that scenario.

#### Behavior details

- **Async procedures**: Procedures declared with async def will run concurrently within the same event loop when invoked through async_view, allowing overlapping awaits on I/O operations. Each individual request still executes one procedure at a time, but the server can progress multiple requests concurrently.
- **Sync procedures**: Synchronous procedures are still supported by async_view. Django will execute them in a threadpool (just like calling a sync view from an async context), so they work transparently, though without the concurrency advantages of native async code.
- API parity: Configuration and registration are identical; there is nothing special to do when registering procedures or namespaces for async_view.

Listing 19: urls.py

```
from django.urls import path
from modernrpc.server import RpcServer

rpc = RpcServer()
# register procedures, namespaces, auth, etc. on rpc as usual

urlpatterns = [
    # Sync endpoint
    path("rpc/", rpc.view),
    # Async endpoint (same API, coroutine-based view)
    path("async_rpc/", rpc.async_view),
]
```

#### Notes

- Django has supported asynchronous views since 3.1; modernrpc supports Django 4.2+. async_view will work on all supported Django versions when running under an ASGI server.
- You can expose both endpoints simultaneously (as shown above). Clients can choose either; functionality is identical.

## 1.3 Procedures registration

### 1.3.1 Introduction

In Django-modern-rpc v2, you first create an RPC server instance, then use its `register_procedure` decorator to expose functions:

Listing 20: myproject/myapp/rpc.py

```
from modernrpc.server import RpcServer

# Create a server instance
server = RpcServer()
```

Listing 21: myproject/myapp/remote_procedures.py

```
from myapp.rpc import server

@server.register_procedure
```

(continues on next page)

(continued from previous page)

```
def add(a, b):
    return a + b
```

Alternatively, a procedure can be registered into a namespace. See *Namespace* for more info.

### 1.3.2 Customize registration

> **Note**
>
> All the arguments documented here can be used similarly in server or namespace registration.

By default, the `@server_or_namespace.register_procedure` decorator will register the procedure to be available to both XML-RPC and JSON-RPC calls. The "methodName" of the procedure will be the function name.

You can customize this behavior by adding arguments to the decorator:

#### Procedure name

Use `name` to override the exposed procedure's "methodName". This is useful to configure a camelCase procedure while keeping the function with a snake_case name. When no namespace is used, this can also be used to declare procedures with a dotted methodName.

Default: `name = None`

```
from myapp.rpc import server

@server.register_procedure(name='addNumbers')
def add_numbers(a, b):
    return a + b
```

#### Protocol availability

When a procedure should be exposed only to a specific protocol, set the `protocol` argument to `Protocol.JSON_RPC` or `Protocol.XML_RPC`.

Default: `protocol = Protocol.ALL`

```
from modernrpc import Protocol
from myapp.rpc import server

@server.register_procedure(protocol=Protocol.JSON_RPC)
def add(a, b):
    return a + b
```

#### Accessing the context

If you need to access some context information in your procedure, simply add an argument with a name of your choice, and declare it in decorator: `register_procedure(context_target="<arg_name>")`

```
from modernrpc import RpcRequestContext
from myapp.rpc import server
```

(continues on next page)

(continued from previous page)

```
@server.register_procedure(context_target="ctx")
def content_type_printer(ctx: RpcRequestContext):
    """Return the Content-Type of the current request.

    :param ctx: Request context (automatically injected)
    :return: Content-Type header value
    """
    # Return the Content-Type of the current request
    return ctx.request.content_type
```

The `RpcRequestContext` instance gives you access to:

- ctx.request: the Django HttpRequest
- ctx.server: the RpcServer currently handling the call
- ctx.handler: the protocol handler (JSON-RPC or XML-RPC)
- ctx.protocol: the active Protocol value
- ctx.auth_result: the value returned by the first authentication predicate that allowed the request to be executed

### 1.3.3 Multiple servers

In version 2.0, the concept of entry points has been replaced with multiple server instances. You can create multiple RPC servers, each with its own set of procedures:

Changed in version 2.0.0: In previous versions, each `RPCEntryPoint` could be defined with a name. Then, at procedure registration, it was possible to specify one or more entry points to register with. Now, if multiple servers are defined, each procedure must be registered explicitly on every server that should expose it. See *Replace multiple RPCEntryPoints*.

Listing 22: myapp/rpc.py

```
from modernrpc.server import RpcServer

# Create multiple server instances
api_v1 = RpcServer()
api_v2 = RpcServer()
```

Then register procedures with the appropriate server:

Listing 23: myapp/remote_procedures.py

```
from myapp.rpc import api_v1, api_v2

# This will expose the procedure only through api_v1
@api_v1.register_procedure
def add(a, b):
    return a + b

# This will expose the procedure only through api_v2
@api_v2.register_procedure
def multiply(a, b):
    return a * b
```

(continues on next page)

(continued from previous page)

```
# If you want to expose a procedure through multiple servers,
# you can register it with each server
@api_v1.register_procedure
@api_v2.register_procedure
def subtract(a, b):
    return a - b
```

## 1.4 Authentication

Changed in version 2.0: Authentication system has been completely rewritten. Please carefully read this page to understand how the new system works

### 1.4.1 Overview

Django-modern-rpc lets you protect procedures using small authentication predicates. A predicate is a callable that receives the Django HttpRequest and returns a truthy value when the caller is authorized, or a falsy value otherwise.

Key points:

- You can configure authentication at three levels: server, namespace, and procedure.
- Predicates are evaluated in their definition order with OR semantics: the first predicate that returns a truthy value authorizes the call; if all return falsy, the call fails with AuthenticationError.
- The truthy value returned by the successful predicate is stored into RpcRequestContext.auth_result so it's available to your procedure (see Accessing the authentication result below).

Changed in version 2.0: In the previous versions, all predicates had to validate the incoming request to allow a procedure to be called

### 1.4.2 Configuration levels

#### 1) Server level (default for all procedures)

Pass auth=<predicate or sequence of predicates> to RpcServer to define a default for all procedures registered on this server, unless overridden by a namespace or a procedure.

Example:

```
from django.http.request import HttpRequest
from modernrpc import RpcServer

def is_staff(request: HttpRequest):
    # Example using Django auth
    return request.user if (request.user.is_authenticated and request.user.is_staff)␣
↪else None

server = RpcServer(auth=is_staff)
```

#### 2) Namespace level (default within a namespace)

RpcNamespace accepts the same auth parameter. Procedures registered on the namespace inherit that default unless they override it.

```
from django.http.request import HttpRequest
from modernrpc import RpcNamespace

def has_api_key(request: HttpRequest):
    return request.headers.get("X-API-Key") == "secret" or None

api = RpcNamespace(auth=has_api_key)

@api.register_procedure
def ping():
    return "pong"

server.register_namespace(api, "api")
```

#### 3) Procedure level (highest precedence)

You can override auth at registration time for a given function. This takes precedence over namespace and server.

```
from django.http.request import HttpRequest

def is_superuser(request: HttpRequest):
    return request.user if (request.user.is_authenticated and request.user.is_superuser)␣
↪else None

@server.register_procedure(name="admin.reset", auth=is_superuser)
def reset_counters():
    ...
```

### 1.4.3 Multiple predicates and evaluation order

Auth can be a single callable or any iterable (list/tuple) of callables. They are called in the order they are provided. The first truthy result short-circuits evaluation and authorizes the request; if none return a truthy value, an `AuthenticationError` is raised.

```
from django.http.request import HttpRequest

def via_token(request: HttpRequest):
    token = request.headers.get("Authorization", "").removeprefix("Bearer ")
    return {"token": token} if token == "valid" else None

def via_session(request: HttpRequest):
    return request.user if request.user.is_authenticated else None

server = RpcServer(auth=[via_session, via_token])
```

In the example above, `via_session` is tried first, then `via_token` if needed.

### 1.4.4 Accessing the authentication result in procedures

When a predicate returns a truthy value, that value is stored in `RpcRequestContext.auth_result`. To access it from within a procedure, ask the server to inject the context into a named parameter using `context_target` at registration, then read `context.auth_result`.

```python
from django.http.request import HttpRequest
from modernrpc import RpcServer, RpcRequestContext

server = RpcServer()

def has_api_key(request: HttpRequest):
    return request.headers.get("X-API-Key") or None

@server.register_procedure(name="echo.secure", context_target="ctx", auth=has_api_key)
def echo_secure(message: str, ctx: RpcRequestContext):
    # ctx is a modernrpc.core.RpcRequestContext
    api_key = ctx.auth_result  # value returned by has_api_key
    return {"message": message, "clear-text-api-key": api_key}
```

### 1.4.5 Notes and best practices

- Predicates should be side-effect free and fast; they are called on every request of protected procedures.
- Return a meaningful truthy object (e.g., the authenticated user, a claims dict, or a token string) to make it usable in your procedures via `ctx.auth_result`.
- Precedence: procedure auth > namespace auth > server auth.

### 1.4.6 Utilities

Since the authentication system has been rewritten from scratch in v2, the decorators previously available to retrieve Basic Auth information from the request and control the permissions of the corresponding user have been removed.

A new module, `modernrpc.auth`, contains some utility functions to help you read authentication data from the request (Basic Auth, Bearer token, etc.).

`modernrpc.auth.`**`extract_header`**(*request*, *header_name*)

Extract a header from a request object or raise a ValueError when it is not found

**Parameters**

- **request** (*HttpRequest*)
- **header_name** (*str*)

**Return type**
str

`modernrpc.auth.`**`extract_generic_token`**(*request*, *header_name*, *auth_type*)

Extract a generic token from a request object and raise a ValueError when it is not found

**Parameters**

- **request** (*HttpRequest*) – HTTP request containing the headers
- **header_name** (*str*) – Header to lookup for authentication data (ex: Authorization)
- **auth_type** (*str*) – Type of authentication information to extract (ex: Bearer, Basic, etc.)

**Returns**
Authentication data

**Return type**
str

modernrpc.auth.**extract_http_basic_auth**(*request*)

Extract HTTP Basic Auth credentials from a request object. Return a tuple with username and password

**Parameters**
**request** (*HttpRequest*)

**Return type**
tuple[str, str]

modernrpc.auth.**extract_bearer_token**(*request: HttpRequest*) → str

Extract a Bearer token from a request object. Return the token.

**Parameters**
**request** (*HttpRequest*) – HTTP request containing the headers

**Returns**
Token string

**Return type**
str

## 1.5 Error handling

### 1.5.1 Introduction

Error handling in RPC protocols is a challenging topic. Any error in remote procedure call processing must be returned to the sender in a valid response with a code and a message. In addition, such a response must be returned with an HTTP status 200.

To implement this behavior, django-modern-rpc uses Python exception mechanism to:

- Raise a pre-defined exception when processing request or serializing response to provide information about the actual error to the sender
- Catch any exception raised from procedures and convert it to a standardized error response.

### 1.5.2 Builtin exceptions

Here is a list of exceptions raised by django-modern-rpc

| Exception | Code | Message |
|---|---|---|
| RPCParseError | -32700 | Parse error, unable to read the request: [...] |
| RPCInsecureRequest | -32700 | Security error: [...] |
| RPCInvalidRequest | -32600 | Invalid request: [...] |
| RPCMethodNotFound | -32601 | Method not found: [...] |
| RPCInvalidParams | -32602 | Invalid parameters: [...] |
| RPCInternalError | -32603 | Internal error: [...] |
| RPCMarshallingError | -32603 | Unable to serialize result data: [...]. Original exception: [...] |

### 1.5.3 Custom exceptions

When any exception is raised from a remote procedure, the client will get a default Internal Error as response. If you want to return a custom error code and message instead, simply define a custom exception. Create an `RPCException` sub-classes and set a *faultCode* to `RPC_CUSTOM_ERROR_BASE + N` with `N` a unique number.

Here is an example:

```
from modernrpc.exceptions import RPCException, RPC_CUSTOM_ERROR_BASE

class MyException1(RPCException):
    def __init__(self, message):
        super().__init__(RPC_CUSTOM_ERROR_BASE + 1, message)

class MyException2(RPCException):
    def __init__(self, message):
        super().__init__(RPC_CUSTOM_ERROR_BASE + 2, message)
```

Such exceptions raised from your remote procedure will be properly returned to client.

### 1.5.4 Customize error handling

#### Overview

In addition to the built-in exceptions described above, django-modern-rpc lets you plug an error handler at the server level. The handler is invoked as soon as any exception is raised from the library or your procedure and before the error response is built and returned to the sender.

The handler receives the original Python exception and a `RpcRequestContext` object. It must have the following signature:

```
from modernrpc import RpcRequestContext

def my_error_handler(exc: BaseException, ctx: RpcRequestContext) -> None:
    ...
```

See *Accessing the context* for detailed documentation about `RpcRequestContext` object.

To register the handler in your server, use the `error_handler` argument:

Listing 24: urls.py

```
from django.urls import path
from modernrpc.server import RpcServer

server = RpcServer(error_handler=my_error_handler)
```

#### Use cases

Below are a few practical examples.

Listing 25: Send any caught exception to Sentry or similar monitoring tool

```
import sentry_sdk

def send_to_sentry(exc: BaseException, ctx: RpcRequestContext) -> None:
    sentry_sdk.capture_exception(exc)

server = RpcServer(error_handler=send_to_sentry)
```

Listing 26: Log any exception as warning using python logging module

```
import logging

err_logger = logging.getLogger("myproject.errors")

def logging_handler(exc: BaseException, ctx: RpcRequestContext) -> None:
    err_logger.warning("RPC error on %s: %s", ctx.request.path, exc, exc_info=True)

server = RpcServer(error_handler=logging_handler)
```

Listing 27: Transform a specific exception into another one

```
from modernrpc.exceptions import RPCInvalidParams

def transform_handler(exc: BaseException, ctx: RpcRequestContext) -> None:
    if isinstance(exc, ZeroDivisionError):
        # Change or update the exception used to build the returned error response
        raise RPCInvalidParams("You cannot divide by Zero") from exc
    # Default case, does nothing. Original exception is kept as base for returned error␣
↪response

server = RpcServer(error_handler=transform_handler)
```

## 1.6 Backends

Added in version 2.0.

Django-modern-rpc can be configured to use different backends to deserialize incoming RPC requests and serialize response data into the right RPC format. Using a custom backend can help implement non-standard behaviors (for example, allowing serialization of additional types) or improve performance.

To compare backend performance, django-modern-rpc provides a benchmark test suite that is automatically run on GitHub Actions for each commit.

### 1.6.1 Terminology

#### Deserializer

In each backend, the Deserializer class parses the incoming request body (XML or JSON) using its underlying library and converts it into a valid Python representation. When the body is not syntactically correct, the deserialization process will usually raise an `RPCParseError`.

The result will be passed to the Unmarshaller (see below).

#### Unmarshaller

The Unmarshaller class takes a Python dictionary built from the incoming request body and extracts important information, such as the procedure name to call and the provided arguments.

When the format (XML-RPC or JSON-RPC) of the parsed request is invalid, the Unmarshaller will usually raise an `RPCInvalidRequest`.

#### Marshaller

The Marshaller class builds a high-level object representing the result of a procedure call (either success or error). This object is then passed to the Serializer, which converts it to a string representation ready to be returned in an `HttpResponse` instance.

When the procedure response contains invalid data, the Marshaller will usually raise an `RPCMarshallingError`

#### Serializer

The Serializer uses its underlying library to serialize the response built by the Marshaller into a valid string representation.

When the serialization process fails, an `RPCMarshallingError` may be raised.

### 1.6.2 Configuration

For each protocol (XML-RPC, JSON-RPC), a different class can be configured for deserialization and serialization.

The default values are:

Listing 28: myproject/settings.py

```
MODERNRPC_XML_DESERIALIZER = {
  "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcDeserializer",
  "kwargs": {}
}
MODERNRPC_XML_SERIALIZER = {
  "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcSerializer",
  "kwargs": {}
}
MODERNRPC_JSON_DESERIALIZER = {
  "class": "modernrpc.jsonrpc.backends.json.PythonJsonDeserializer",
  "kwargs": {}
}
MODERNRPC_JSON_SERIALIZER = {
  "class": "modernrpc.jsonrpc.backends.json.PythonJsonSerializer",
  "kwargs": {}
}
```

Each class can be configured using the `kwargs` dictionary. The valid parameters depend on the selected backend. Some arguments are common across many backends, while others apply only to specific ones. See below for a detailed explanation of each backend's configuration.

Some backends may be configured to use a different Unmarshaller and/or Marshaller class. When this is possible, use `unmarshaller_klass` / `unmarshaller_kwargs` and `marshaller_klass` / `marshaller_kwargs` in `kwargs`.

### 1.6.3 XML-RPC backends

#### xmlrpc (python builtin)

This is the most basic backend that depends on Python's builtin xmlrpc module. It is used by default for both deserialization and serialization of XML-RPC requests and responses.

### Pros / Cons

No additional dependency required (stdlib)
Secure parsing via defusedxml protections enabled by default
Has better performance than other backends for both serialization and deserialization

Can fail to parse some requests, particularly when there are extra spaces around certain values such as booleans, strings, or dates

/ The underlying xmlrpc library can be permissive and accept some non-standard formatting. However, this backend will raise `RPCInvalidRequest` if the method name cannot be determined (for example, when the root tag is not `methodCall`).

### Configuration

#### Unmarshaller / Deserializer

- `load_kwargs`: passed to `xmlrpc.client.loads`. See the xmlrpc.client.loads() documentation for the list of valid keyword arguments

Listing 29: myproject/settings.py

```
MODERNRPC_XML_DESERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcDeserializer",
    "kwargs": {
        "load_kwargs": {"use_datetime": False, "use_builtin_types": False}
    }
}
```

The Unmarshaller class cannot be changed or configured at the moment.

#### Marshaller / Serializer

- `dump_kwargs`: passed to `xmlrpc.client.dumps`. See the xmlrpc.client.dumps() documentation for the list of valid keyword arguments.

Listing 30: myproject/settings.py

```
MODERNRPC_XML_SERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcSerializer",
    "kwargs": {
        "dump_kwargs": {"allow_none": False}
    }
}
```

The Marshaller class cannot be changed or configured at the moment.

### etree (ElementTree)

Uses Python's standard library `xml.etree.ElementTree` (through defusedxml wrappers) to parse and build XML-RPC messages. Can be used as both serializer and deserializer.

### Pros / Cons

no additional dependency required (stdlib)
secure parsing via defusedxml protections enabled by default

may be slower and less feature-rich than lxml for large or complex XML

### Configuration

### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.xmlrpc.backends.marshalling.EtreeElementUnmarshaller`.
- `unmarshaller_kwargs`: keyword arguments passed to the Unmarshaller. Supported option:
  - `allow_none` (default: `True`): whether to allow parsing of `<nil/>`/`None` values.
- `element_type_klass`: dotted path to the XML Element class used to specialize the generic unmarshaller. Defaults to `xml.etree.ElementTree.Element`.
- `load_kwargs`: passed to `defusedxml.ElementTree.XML`. By default, `forbid_dtd` is set to `True` for safety.

Example:

Listing 31: myproject/settings.py

```
MODERNRPC_XML_DESERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.etree.EtreeDeserializer",
    "kwargs": {
        "unmarshaller_kwargs": {"allow_none": False},
        "element_type_klass": "xml.etree.ElementTree.Element",
        "load_kwargs": {"forbid_dtd": True},
    }
}
```

### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.xmlrpc.backends.marshalling.EtreeElementMarshaller`.
- `marshaller_kwargs`: keyword arguments passed to the Marshaller. Supported options:
  - `allow_none` (default: `True`): whether to allow serialization of `None` values.
  - `element_factory` (default: `xml.etree.ElementTree.Element`): the function used to build a new Element
  - `sub_element_factory` (default: `xml.etree.ElementTree.SubElement`): the function used to build a new SubElement
- `element_type_klass`: dotted path to the XML Element class used to specialize the generic marshaller. Defaults to `xml.etree.ElementTree.Element`.
- `dump_kwargs`: passed to `defusedxml.ElementTree.tostring`.

Example:

Listing 32: myproject/settings.py

```
MODERNRPC_XML_SERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.etree.EtreeSerializer",
    "kwargs": {
        "marshaller_kwargs": {"allow_none": True},
        "element_type_klass": "xml.etree.ElementTree.Element",
        "dump_kwargs": {"short_empty_elements": False},
    }
}
```

### lxml

Uses third-party lxml library (`lxml.etree`). Can be used as both serializer and deserializer.

To use this backend, `lxml` must be installed in the current environment. An extra dependency can be used for that:

```
pip install django-modern-rpc[lxml]
```

```
poetry add django-modern-rpc[lxml]
```

```
uv add django-modern-rpc[lxml]
```

### Pros / Cons

very fast and robust XML processing
richer XML feature set compared to stdlib etree

requires an additional dependency

### Configuration

#### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.xmlrpc.backends.marshalling.EtreeElementUnmarshaller`.
- `unmarshaller_kwargs`: keyword arguments passed to the Unmarshaller. Supported option:
  - `allow_none` (default: `True`)
- `element_type_klass`: dotted path to the XML element class used to specialize the generic unmarshaller. Defaults to `lxml.etree._Element`.
- `load_parser_kwargs`: keyword arguments passed to `lxml.etree.XMLParser`. Secure defaults are applied: `resolve_entities=False`, `no_network=True`, `dtd_validation=False`, `load_dtd=False`, `huge_tree=False`.
- `load_kwargs`: additional arguments forwarded to `lxml.etree.fromstring` (the constructed parser is injected by the backend).

Example:

Listing 33: myproject/settings.py

```
MODERNRPC_XML_DESERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.lxml.LxmlDeserializer",
    "kwargs": {
        "unmarshaller_kwargs": {"allow_none": True},
        "load_parser_kwargs": {"huge_tree": False},
    }
}
```

#### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.xmlrpc.backends.marshalling.EtreeElementMarshaller`.
- `marshaller_kwargs`: keyword arguments passed to the Marshaller. Supported options:
  - `allow_none` (default: `True`): whether to allow serialization of `None` values.
  - `element_factory` (default: `lxml.etree.Element`): the function used to build a new Element
  - `sub_element_factory` (default: `lxml.etree.SubElement`): the function used to build a new SubElement
- `element_type_klass`: dotted path to the XML Element class used to specialize the generic marshaller. Defaults to `lxml.etree._Element`.
- `dump_kwargs`: passed to `lxml.etree.tostring`.

Example:

Listing 34: myproject/settings.py

```
MODERNRPC_XML_SERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.lxml.LxmlSerializer",
    "kwargs": {
        "marshaller_kwargs": {"allow_none": True},
```

(continues on next page)

(continued from previous page)

```
        "dump_kwargs": {"pretty_print": True},
    }
}
```

### xmltodict

Uses third-party xmltodict library. Can be used as both serializer and deserializer.

To use this backend, `xmltodict` must be installed in the current environment. It can be done with the included extra dependency `xmltodict`:

```
pip install django-modern-rpc[xmltodict]
```

```
poetry add django-modern-rpc[xmltodict]
```

```
uv add django-modern-rpc[xmltodict]
```

### Pros / Cons

Provides more control over XML parsing and serialization

requires an additional dependency

### Configuration

### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.xmlrpc.backends.xmltodict.Unmarshaller`.
- `unmarshaller_kwargs`: keyword arguments passed to the Unmarshaller. Currently, default one doesn't support any argument
- `load_kwargs`: passed to `xmltodict.parse`. See the xmltodict docs.

Example:

Listing 35: myproject/settings.py

```
MODERNRPC_XML_DESERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmltodict.XmlToDictDeserializer",
    "kwargs": {
        "load_kwargs": {"disable_entities": False},
    }
}
```

Note: For security reasons, this backend first parses XML with `defusedxml.ElementTree` before converting it with `xmltodict`. Any XML parsing errors or security violations will be reported accordingly.

#### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.xmlrpc.backends.xmltodict.Marshaller`.
- `marshaller_kwargs`: keyword arguments passed to the Marshaller. Supported options:
  - `allow_none` (default: `True`): whether to allow serialization of `None` values.
- `dump_kwargs`: passed to `xmltodict.unparse`. See the xmltodict docs.

Note: In this backend, the Marshaller/Unmarshaller classes are fixed and cannot be changed via settings.

Example:

Listing 36: myproject/settings.py

```
MODERNRPC_XML_SERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmltodict.XmlToDictSerializer",
    "kwargs": {
        "dump_kwargs": {"pretty": True, "indent": "  ", "short_empty_elements": True},
        "marshaller_kwargs": {"allow_none": False},
    }
}
```

### 1.6.4 JSON-RPC backends

#### Common Unmarshaller / Marshaller

Unlike XML-RPC backends, JSON-RPC ones are simpler. Basically, a JSON deserializer will convert a string payload into a structured python data (list, dict, etc.). Then, most of the work will be handled by the Unmarshaller class to build a valid `JsonRpcRequest` instance.

On the other hand, the Marshaller will convert a `JsonRpcResult` (success or error) to a valid python dict and pass the result to the serializer to build the corresponding JSON payload as a string.

Currently, the default JSON-RPC Unmarshaller / Marshaller classes are shared by all backends.

#### Unmarshaller configuration

Supported kwargs:

- `validate_version` (Default: `True`): whether to enforce check for `"jsonrpc": "2.0"` in the incoming request.

#### Marshaller configuration

Common Marshaller does not support any argument for now.

#### json (python builtin)

This is the most basic backend that depends on Python's built-in `json` module. It is used by default for both deserialization and serialization of JSON-RPC requests and responses.

By default, this backend use common *Unmarshaller* and *Marshaller* classes. In addition, it is configured to use `DjangoJSONEncoder` when serializing data, allowing `date`, `time` and `datetime` instances to be serialized transparently.

### Pros / Cons

no dependency required

not the most performant backend available

### Configuration

#### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Unmarshaller`.
- `unmarshaller_kwargs`: see *Unmarshaller configuration*
- `load_kwargs`: passed to `json.loads`. See the json.loads() documentation for the list of valid keyword arguments

Listing 37: myproject/settings.py

```
MODERNRPC_JSON_DESERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.json.PythonJsonDeserializer",
    "kwargs": {
        "load_kwargs": {},
        "unmarshaller_klass": "modernrpc.jsonrpc.backends.marshalling.Unmarshaller",
        "unmarshaller_kwargs": {"validate_version": False},
    }
}
```

#### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Marshaller`.
- `marshaller_kwargs`: see *Marshaller configuration*
- `dump_kwargs`: passed to `json.dumps`. See the json.dumps() documentation for the list of valid keyword arguments.

Note: By default, `json.dumps` encoder is overridden to use DjangoJSONEncoder through the *cls* argument, primarily to allow serializing `date`, `time` and `datetime` objects.

Listing 38: myproject/settings.py

```
MODERNRPC_JSON_SERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.json.PythonJsonSerializer",
    "kwargs": {
        "marshaller_klass": "myproject.core.marshaller.CustomMarshaller",
        "marshaller_kwargs": {"config1": "foo"},
        "dump_kwargs": {"indent": 2, "sort_keys": False},
    }
}
```

### orjson

Uses third party orjson library. Can be used as both serializer and deserializer.

To use this backend, *orjson* must be installed in the current environment. An extra dependency can be used for that:

```
pip install django-modern-rpc[orjson]
```

```
poetry add django-modern-rpc[orjson]
```

```
uv add django-modern-rpc[orjson]
```

### Pros / Cons

Extremely fast

requires an additional dependency

### Configuration

#### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Unmarshaller`.
- `unmarshaller_kwargs`: see *Unmarshaller configuration*
- `load_kwargs`: ignored for now. `orjson.loads` does not accept any additional keyword arguments

Listing 39: myproject/settings.py

```
MODERNRPC_JSON_DESERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.orjson.OrjsonDeserializer",
    "kwargs": {
        "unmarshaller_klass": "modernrpc.jsonrpc.backends.marshalling.Unmarshaller",
        "unmarshaller_kwargs": {"validate_version": False},
    }
}
```

#### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Marshaller`.
- `marshaller_kwargs`: see *Marshaller configuration*
- `dump_kwargs`: passed to `orjson.dumps`. See the orjson.dumps() documentation for the list of valid keyword arguments.

Note: By default, since orjson is already able to serialize `date`, `time` and `datetime` objects natively, no particular encoder customization is performed by django-modern-rpc.

Listing 40: myproject/settings.py

```
MODERNRPC_JSON_SERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.orjson.OrjsonSerializer",
    "kwargs": {
        "dump_kwargs": {},
    }
}
```

### ujson

Uses third party ujson (UltraJSON) library. Can be used as both serializer and deserializer.

To use this backend, *ujson* must be installed in the current environment. An extra dependency can be used for that:

```
pip install django-modern-rpc[ujson]
```

```
poetry add django-modern-rpc[ujson]
```

```
uv add django-modern-rpc[ujson]
```

### Pros / Cons

Very fast

requires an additional dependency

### Configuration

### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Unmarshaller`.
- `unmarshaller_kwargs`: see *Unmarshaller configuration*
- `load_kwargs`: passed to `ujson.loads`. See the ujson docs for the list of valid keyword arguments

Listing 41: myproject/settings.py

```
MODERNRPC_JSON_DESERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.ujson.UjsonDeserializer",
    "kwargs": {
        "load_kwargs": {},
        "unmarshaller_klass": "modernrpc.jsonrpc.backends.marshalling.Unmarshaller",
        "unmarshaller_kwargs": {"validate_version": False},
    }
}
```

### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Marshaller`.
- `marshaller_kwargs`: see *Marshaller configuration*
- `dump_kwargs`: passed to `ujson.dumps`. See the ujson docs for the list of valid keyword arguments.

Note: `ujson.dumps` does not support the `cls` argument. By default, a custom `default` function created from Django's DjangoJSONEncoder `default` method is passed to `ujson.dumps`, primarily to allow serializing `date`, `time` and `datetime` objects.

Listing 42: myproject/settings.py

```
MODERNRPC_JSON_SERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.ujson.UjsonSerializer",
    "kwargs": {
        "marshaller_klass": "myproject.core.marshaller.CustomMarshaller",
        "marshaller_kwargs": {"config1": "foo"},
        "dump_kwargs": {"indent": 2, "sort_keys": False},
    }
}
```

### simplejson

Uses third party SimpleJSON library. Can be used for both serialization and deserialization.

To use this backend, *simplejson* must be installed in the current environment. An extra dependency can be used for that:

```
pip install django-modern-rpc[simplejson]
```

```
poetry add django-modern-rpc[simplejson]
```

```
uv add django-modern-rpc[simplejson]
```

### Pros / Cons

Easier encoder customization

requires an additional dependency
slower than some other options (including builtin json)

### Configuration

### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Unmarshaller`.
- `unmarshaller_kwargs`: see *Unmarshaller configuration*
- `load_kwargs`: passed to `simplejson.loads`. See the simplejson.loads() documentation for the list of valid keyword arguments

Listing 43: myproject/settings.py

```
MODERNRPC_JSON_DESERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.simplejson.SimpleJsonDeserializer",
    "kwargs": {
        "load_kwargs": {"use_decimal": False, "allow_nan": False},
        "unmarshaller_klass": "modernrpc.jsonrpc.backends.marshalling.Unmarshaller",
        "unmarshaller_kwargs": {"validate_version": False},
    }
}
```

### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Marshaller`.
- `marshaller_kwargs`: see *Marshaller configuration*
- `dump_kwargs`: passed to `simplejson.dumps`. See the simplejson.dumps() documentation for the list of valid keyword arguments.

Note: By default, a custom `default` function created from Django's DjangoJSONEncoder `default` method is passed to `simplejson.dumps`, primarily to allow serializing `date`, `time` and `datetime` objects.

Listing 44: myproject/settings.py

```
MODERNRPC_JSON_SERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.simplejson.SimpleJsonSerializer",
    "kwargs": {
        "marshaller_klass": "myproject.core.marshaller.CustomMarshaller",
        "marshaller_kwargs": {"config1": "foo"},
        "dump_kwargs": {"indent": 2, "sort_keys": False},
    }
}
```

### rapidjson

Uses third party python-rapidjson library. Can be used as both serializer and deserializer.

To use this backend, *python-rapidjson* must be installed in the current environment. An extra dependency can be used for that:

```
pip install django-modern-rpc[rapidjson]
```

```
poetry add django-modern-rpc[rapidjson]
```

```
uv add django-modern-rpc[rapidjson]
```

### Pros / Cons

Pretty fast

requires an additional dependency

#### Configuration

#### Unmarshaller / Deserializer

- `unmarshaller_klass`: dotted path to the Unmarshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Unmarshaller`.
- `unmarshaller_kwargs`: see *Unmarshaller configuration*
- `load_kwargs`: passed to `rapidjson.loads`. See the rapidjson.loads() documentation for the list of valid keyword arguments

Listing 45: myproject/settings.py

```
MODERNRPC_JSON_DESERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.rapidjson.RapidJsonDeserializer",
    "kwargs": {
        "load_kwargs": {},
        "unmarshaller_klass": "modernrpc.jsonrpc.backends.marshalling.Unmarshaller",
        "unmarshaller_kwargs": {"validate_version": False},
    }
}
```

#### Marshaller / Serializer

- `marshaller_klass`: dotted path to the Marshaller class. Defaults to `modernrpc.jsonrpc.backends.marshalling.Marshaller`.
- `marshaller_kwargs`: see *Marshaller configuration*
- `dump_kwargs`: passed to `rapidjson.dumps`. See the rapidjson.dumps() documentation for the list of valid keyword arguments.

Note: By default, date & time handling is set to *ISO-8601* via the `datetime_mode` argument (`rapidjson.DM_ISO8601`) to help serializing `date`, `time` and `datetime` objects.

Listing 46: myproject/settings.py

```
MODERNRPC_JSON_SERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.rapidjson.RapidJsonSerializer",
    "kwargs": {
        "dump_kwargs": {"indent": 2, "sort_keys": True},
    }
}
```

### 1.6.5 Helper functions

modernrpc.helpers.**get_builtin_date**(*date*, *date_format='%Y-%m-%dT%H:%M:%S'*, *raise_exception=False*)

Try to convert a date to a builtin instance of `datetime.datetime`. The input date can be a `str`, a `datetime.datetime`, a `xmlrpc.client.Datetime` or a `xmlrpclib.Datetime` instance. The returned object is a `datetime.datetime`.

**Parameters**

- **date** (*str | datetime | DateTime*) – The date object to convert.
- **date_format** (*str*) – If the given date is a str, it is passed to strptime for parsing

- **raise_exception** (*bool*) – If set to True, an exception will be raised if the input string cannot be parsed

**Returns**
A valid `datetime.datetime` instance

**Return type**
*datetime* | None

## 1.7 Procedures documentation

Django-modern-rpc processes the docstring attached to your RPC methods to provide rich information about your API. This article explains how documentation is generated and can be customized in version 2.0.

### 1.7.1 Documentation generation

In version 2.0, the automatic HTML documentation generation through entry points has been removed. However, the library still processes docstrings to provide rich information about your procedures, which can be accessed programmatically.

Each `ProcedureWrapper` instance has properties to access documentation:

- `text_doc`: The raw docstring text
- `html_doc`: The docstring converted to HTML
- `arguments`: A `dict` mapping argument names to their documentation
- `returns`: Documentation for the return value

You can use these properties to build your own documentation views or integrate with other documentation systems.

```
from myapp.rpc import server

# Get all registered procedures
procedures = server.procedures

# Access documentation for a specific procedure
add_procedure = procedures.get('add')
if add_procedure:
    html_documentation = add_procedure.html_doc
    args_documentation = add_procedure.arguments
    return_documentation = add_procedure.returns
```

### 1.7.2 Write documentation

The documentation is generated directly from RPC methods' docstrings. In version 2.0, you can use Python type hints and docstring formats like reStructuredText or Markdown:

```
from myapp.rpc import server

@server.register_procedure(context_target='ctx')
def content_type_printer(ctx) -> str:
    """
    Inspect request to extract the Content-Type header if present.
    This method demonstrates how a RPC method can access the request object.

```

(continues on next page)

(continued from previous page)

```
    :param ctx: Request context with access to the current request
    :return: The Content-Type string for incoming request
    """
    # Access the request from the context
    request = ctx.request

    # Return the content-type of the current request
    return request.content_type
```

If you want to use *Markdown* or *reStructuredText* syntax in your RPC method documentation, you have to install the corresponding package in your environment.

```
pip install Markdown
```

or

```
pip install docutils
```

Then, set `settings.MODERNRPC_DOC_FORMAT` to indicate which parser must be used to process your docstrings

```
# In settings.py
MODERNRPC_DOC_FORMAT = 'markdown'
```

or

```
# In settings.py
MODERNRPC_DOC_FORMAT = 'rst'
```

Added in version 2.0.0: Type hints are now supported to generate arguments and return type in documentation

## 1.8 Settings

This page lists all settings that can be used to customize django-modern-rpc's behavior. Set them inside your project's `settings.py`.

### 1.8.1 Global settings

#### MODERNRPC_DEFAULT_ENCODING

Default encoding used to parse incoming requests when no charset is set in request headers.

**Default**

`utf-8`

#### MODERNRPC_DOC_FORMAT

Configure the format of the docstring used to document your RPC methods.

Possible values are:

- `""` (empty string): no processing, the docstring is returned as-is (with newlines converted to HTML paragraphs)
- `"rst"` (also `"restructured"` / `"restructuredtext"`): parse docstrings as reStructuredText (requires `docutils`)
- `"md"` (also `"markdown"`): parse docstrings as Markdown (requires `markdown`)

The value is case-insensitive.

**Default**
"" (Empty string)

> **Note**
>
> The corresponding package is not automatically installed. You have to ensure library *markdown* or *docutils* is installed in your environment if set to a non-empty value.

#### MODERNRPC_XMLRPC_ASYNC_MULTICALL

When set to `True`, the `system.multicall` XML-RPC method will use an asynchronous implementation that executes procedures concurrently using `asyncio.gather()`. When `False` (default), procedures in a multicall are executed sequentially.

**Default**
`False`

#### MODERNRPC_HANDLERS

List of handler classes used by default in `RpcServer` instances. If you have overridden a handler class, you may need to specify its dotted path here to use it automatically.

**Default**
```
["modernrpc.jsonrpc.handler.JsonRpcHandler", "modernrpc.xmlrpc.handler.
XmlRpcHandler"]
```

### 1.8.2 Backends customization

Each protocol (XML-RPC & JSON-RPC) can be configured with a specific backend for both deserialization (parsing of incoming request) and serialization (dumping outgoing response).

For each setting, a dict is defined with `class` and `kwargs` keys to set the dotted path of the class to instantiate and a dictionary passed to the class when instantiating. Valid *kwargs* depend on the selected *class*. Refer to *Backends* to get a list of all valid arguments for each backend.

#### MODERNRPC_XML_DESERIALIZER

**Default**
```
{"class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcDeserializer",
"kwargs": {}}
```

#### MODERNRPC_XML_SERIALIZER

**Default**
```
{"class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcSerializer",
"kwargs": {}}
```

#### MODERNRPC_JSON_DESERIALIZER

**Default**
```
{"class": "modernrpc.jsonrpc.backends.json.PythonJsonDeserializer",
"kwargs": {}}
```

#### MODERNRPC_JSON_SERIALIZER

**Default**

```
{"class":  "modernrpc.jsonrpc.backends.json.PythonJsonSerializer",
"kwargs":  {}}
```

## 1.9 Security concerns

### 1.9.1 Protection against XML vulnerabilities

#### Why this matters

XML parsing is historically vulnerable to several classes of attacks such as:

- External entity expansion (XXE) leaking local files or reaching internal network resources
- Billion Laughs / quadratic blowup attacks via entity expansion causing DoS
- DTD-related exploits and network retrieval during parsing

To mitigate these risks, django-modern-rpc ships with safe defaults for all XML-RPC backends and requires defusedxml.

#### Backend hardening overview

All built-in XML-RPC backends are configured with protections against XXE/DoS:

- *etree* backend (`xml.etree.ElementTree`) - Uses `defusedxml.ElementTree` for parsing and serialization. - Forbidden constructs (DTD, entities, external references) raise defusedxml exceptions, which are translated into `RPCInsecureRequest`. - Malformed XML raises `RPCParseError`.
- *xmltodict* backend - First parses with defusedxml.ElementTree.fromstring to validate securely, then feeds the result to xmltodict. - `defusedxml.DefusedXmlException` (XXE/DTD/etc.) is mapped to `RPCInsecureRequest`; parse errors to `RPCParseError`.
- *lxml* backend - Uses a hardened `lxml.etree.XMLParser` with `resolve_entities=False`, `no_network=True`, `dtd_validation=False`, `load_dtd=False`, `huge_tree=False`. - `XMLSyntaxError` is mapped to `RPCParseError`.

#### What you will see on insecure input

- Requests containing dangerous XML features are rejected early with an `RPCInsecureRequest`.
- Well-formed but invalid XML for XML-RPC is rejected with `RPCInvalidRequest`.
- Broken XML syntax results in `RPCParseError`.

These exceptions are caught by the server and converted into safe protocol-specific responses.

### 1.9.2 Writing custom XML backends safely

If you implement a custom XML-RPC backend, follow these guidelines:

- Prefer `defusedxml` wrappers over the stdlib XML APIs:

```
from defusedxml import ElementTree as SafeET
root = SafeET.fromstring(xml_bytes)  # Safe parsing: DTD, entities, external refs are␣
↪forbidden
```

- Or, when using lxml, create a locked-down parser:

```
import lxml.etree as ET

parser = ET.XMLParser(resolve_entities=False, no_network=True, dtd_validation=False,␣
↪load_dtd=False, huge_tree=False)
root = ET.fromstring(xml_bytes, parser)
```

- Convert defusedxml.DefusedXmlException and parsing errors into modernrpc exceptions (`RPCInsecureRequest`, `RPCParseError`) so the server can return a safe error response.

### 1.9.3 Notes

- The secure defaults apply automatically; no additional configuration is required.
- Do not disable these protections in production. They exist to reduce the attack surface of XML parsing.

## 1.10 Frequently Asked Questions

### 1.10.1 Is it possible to return model instances?

No. RPC protocols only support a limited set of types: scalars (strings, integers, floats, booleans, None), collections (lists, dicts) and a few special types such as dates and binary data. See *Types support* for the full list. You have to convert model instances to serializable types yourself, for example by returning a dictionary:

```
@server.register_procedure
def get_user(user_id: int) -> dict:
    user = User.objects.get(pk=user_id)
    return {"id": user.id, "username": user.username, "email": user.email}
```

A third-party library like Pydantic may help to easily produce valid dict from high level data.

### 1.10.2 Can I register a procedure in multiple servers?

A procedure can be decorated multiple times, so it can be registered in more than one server.

See *Multiple servers*

### 1.10.3 Can I register a namespace into a namespace?

No, this is currently not supported. If you need multiple levels of nesting, you can use the `name` parameter on `register_procedure` to declare procedures with a dotted name (e.g., `"math.advanced.fft"`).

### 1.10.4 Is there a way to serve documentation on server endpoint with GET request?

The automatic HTML documentation generation through entry points has been removed in v2. However, you can configure `RpcServer` with `redirect_get_request_to` to redirect GET requests to another view (e.g., a custom documentation page). See *GET requests redirection* for details.

## 1.11 Migration guide

### 1.11.1 From 1.1 to 2.0

This section will guide you to update an existing setup of django-modern-rpc 1.0 or 1.1 to the latest v2 release.

### Update settings

#### MODERNRPC_METHODS_MODULES

In v2, the new Server / Namespace system will automatically import your decorated procedures. This setting is now useless, you can simply delete it from your settings.

#### MODERNRPC_LOG_EXCEPTIONS

This setting has been removed. In v2, exception logging can be handled through the `error_handler` callback on `RpcServer`. See *Customize error handling* for details. You can simply delete this setting from your configuration.

#### MODERNRPC_DOC_FORMAT

This setting still exists in v2. Accepted values are `""` (empty string, default), `"rst"` or `"md"` (also accepts `"markdown"`). No migration action is needed, but note that automatic HTML documentation through entry points has been removed. Docstrings are still processed for introspection (`system.methodHelp`).

#### MODERNRPC_DEFAULT_ENTRYPOINT_NAME

The `RPCEntryPoint` class is not used anymore. You can define multiple `RpcServer` instances to split your APIs. No specific name is required, and no default one is needed anymore. This setting is now useless, you can simply delete it from your settings.

#### MODERNRPC_JSON_DECODER / MODERNRPC_JSON_ENCODER

Backends are now configured individually. Configuring the encoder and decoder with default builtin `json` backend is still possible using new settings.

### Before

Listing 47: myproject/settings.py

```
MODERNRPC_JSON_DECODER = "path.to.valid.json.Decoder"
MODERNRPC_JSON_ENCODER = "path.to.valid.json.Encoder"
```

### After

Listing 48: myproject/settings.py

```
MODERNRPC_JSON_DESERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.json.PythonJsonDeserializer",
    "kwargs": {
        "load_kwargs": {"cls": path.to.valid.json.Decoder},
    }
}
MODERNRPC_JSON_SERIALIZER = {
    "class": "modernrpc.jsonrpc.backends.json.PythonJsonSerializer",
    "kwargs": {
        "dump_kwargs": {"cls": path.to.valid.json.Encoder},
    }
}
```

### MODERNRPC_XMLRPC_USE_BUILTIN_TYPES / MODERNRPC_XMLRPC_ALLOW_NONE

Backends are now configured individually. Configuring the behavior of builtin `xmlrpc` backend is still possible using new settings.

#### Before

Listing 49: myproject/settings.py

```
MODERNRPC_XMLRPC_USE_BUILTIN_TYPES = False
MODERNRPC_XMLRPC_ALLOW_NONE = False
```

#### After

Listing 50: myproject/settings.py

```
MODERNRPC_XML_DESERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcDeserializer",
    "kwargs": {
        "load_kwargs": {"use_builtin_types": False}
    }
}
MODERNRPC_XML_SERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcSerializer",
    "kwargs": {
        "dump_kwargs": {"allow_none": False}
    }
}
```

### MODERNRPC_XMLRPC_DEFAULT_ENCODING

In the previous versions, this setting was used to initialize an `xmlrpc.client.Marshaller`. In v2, this class is not directly instantiated but used through the serialization process. Encoding can still be configured.

#### Before

Listing 51: myproject/settings.py

```
MODERNRPC_XMLRPC_DEFAULT_ENCODING = "ascii"
```

#### After

Listing 52: myproject/settings.py

```
MODERNRPC_XML_SERIALIZER = {
    "class": "modernrpc.xmlrpc.backends.xmlrpc.PythonXmlRpcSerializer",
    "kwargs": {
        "dump_kwargs": {"encoding": "ascii"}
    }
}
```

### Replace a single RPCEntryPoint

#### Before

Procedure registration was possible from anywhere in the code, as soon as the module was declared in *settings.MODERNRPC_METHODS_MODULES*.

Listing 53: myapp/remote_procedures.py

```
from modernrpc.core import rpc_method

@rpc_method
def add(a, b):
    return a + b
```

Listing 54: myproject/urls.py

```
from django.urls import path
from modernrpc.views import RPCEntryPoint

urlpatterns = [
    # ... other url patterns
    path('rpc/', RPCEntryPoint.as_view()),
]
```

#### After

With v2, an `RpcServer` instance must be created, and then used to register procedures.

Listing 55: myproject/myapp/rpc.py

```
from modernrpc.server import RpcServer

server = RpcServer()


@server.register_procedure
def add(a: int, b: int) -> int:
    """Add two numbers and return the result.

    :param a: First number
    :param b: Second number
    :return: Sum of a and b
    """
    return a + b
```

Listing 56: myproject/urls.py

```
from django.urls import path
from myapp.rpc import server

urlpatterns = [
    # ... other url patterns
    path('rpc/', server.view),
```

(continues on next page)

(continued from previous page)

```
]
```

### Replace multiple RPCEntryPoints

#### Before

In v1.x, multiple endpoints were declared by instantiating `RPCEntryPoint` several times, each one bound to a named entry point. Procedures were then attached to a specific entry point using the `entry_point` argument of the `@rpc_method` decorator. A procedure with no explicit entry point was served by every endpoint.

Listing 57: myapp/remote_procedures.py

```
from modernrpc.core import rpc_method

@rpc_method(entry_point="apiV1")
def add(a, b):
    return a + b

@rpc_method(entry_point="apiV2")
def add(a, b):
    return a + b

@rpc_method
def ping():
    # No entry_point: available on every endpoint
    return "pong"
```

Listing 58: myproject/urls.py

```
from django.urls import path
from modernrpc.views import RPCEntryPoint

urlpatterns = [
    # ... other url patterns
    path('rpc/v1/', RPCEntryPoint.as_view(entry_point="apiV1")),
    path('rpc/v2/', RPCEntryPoint.as_view(entry_point="apiV2")),
]
```

#### After

With v2, the `entry_point` concept disappears. Each endpoint becomes a distinct `RpcServer` instance, and you register each procedure directly on the server(s) that must expose it. To make a procedure available on several servers (the v1.x behavior of a procedure without an `entry_point`), simply stack the registration decorators.

Listing 59: myproject/myapp/rpc.py

```
from modernrpc.server import RpcServer

api_v1 = RpcServer()
api_v2 = RpcServer()
```

(continues on next page)

(continued from previous page)

```
@api_v1.register_procedure
def add(a: int, b: int) -> int:
    """Add two numbers and return the result."""
    return a + b


@api_v2.register_procedure
def add(a: int, b: int) -> int:
    """Add two numbers and return the result (v2)."""
    return a + b


@api_v1.register_procedure
@api_v2.register_procedure
def ping() -> str:
    # Registered on both servers
    return "pong"
```

Listing 60: myproject/urls.py

```
from django.urls import path
from myapp.rpc import api_v1, api_v2

urlpatterns = [
    # ... other url patterns
    path('rpc/v1/', api_v1.view),
    path('rpc/v2/', api_v2.view),
]
```

This new process allows you to easily customize registration per procedure and per server.

### Update your authentication configuration

#### Before

Listing 61: myproject/myapp/auth.py

```
# Custom predicate used to block some procedures to known bots
def forbid_bots_access(request):
    """Return True when request has a User-Agent different from provided list"""
    if "User-Agent" not in request.headers:
        # No User-Agent provided, the request must be rejected
        return False

    forbidden_bots = [
        'Googlebot',  # Google
        'Bingbot',  # Microsoft
        'Slurp',  # Yahoo
        'DuckDuckBot',  # DuckDuckGo
        'Baiduspider',  # Baidu
        'YandexBot',  # Yandex
        'facebot',  # Facebook
```

(continues on next page)

(continued from previous page)

```
    ]

    if request.headers["User-Agent"].lower() in [ua.lower() for ua in forbidden_bots]:
        # ... forbid access
        return False

    # In all other cases, allow access
    return True
```

```
from modernrpc.core import rpc_method
from modernrpc.auth import set_authentication_predicate
from modernrpc.auth.basic import http_basic_auth_permissions_required
from myproject.myapp.auth import forbid_bots_access


@rpc_method
@http_basic_auth_permissions_required(permissions='auth.view_user')
def my_rpc_method_with_builtin_predicate(username):
    user = User.objects.get(username=username)
    return f"{user.first_name} {user.last_name}"

@rpc_method
@set_authentication_predicate(forbid_bots_access)
def my_rpc_method_with_custom_authentication(a, b):
    return a + b
```

### After

Listing 62: myproject/myapp/auth.py

```
from django.contrib.auth import authenticate
from django.http.request import HttpRequest

from modernrpc.auth import extract_http_basic_auth

# Predicate used to block some procedures to known bots
def forbid_bots_access(request: HttpRequest):
    """Return True when request has a User-Agent different from provided list"""
    if "User-Agent" not in request.headers:
        # No User-Agent provided, the request must be rejected
        return False

    forbidden_bots = [
        'Googlebot',  # Google
        'Bingbot',  # Microsoft
        'Slurp',  # Yahoo
        'DuckDuckBot',  # DuckDuckGo
        'Baiduspider',  # Baidu
        'YandexBot',  # Yandex
        'facebot',  # Facebook
    ]
```

(continues on next page)

(continued from previous page)

```
    if request.headers["User-Agent"].lower() in [ua.lower() for ua in forbidden_bots]:
        # ... forbid access
        return False

    # In all other cases, allow access
    return True


# Predicate to check for specific Django permissions
def check_view_permissions(perms: str):
    def inner(request: HttpRequest):
        # Use modernrpc helper to extract Basic Auth username & password
        username, password = extract_http_basic_auth(request)
        # Use Django auth system to authenticate the user
        user = authenticate(username=username, password=password)
        # Check for authentication (valid username & password) AND for permissions
        if not user or not user.has_perm(perms):
            return False
        # User is authenticated AND authorized
        return user

    return inner
```

```
from myproject.myapp.auth import check_view_permissions, forbid_bots_access


@server.register_procedure(auth=check_view_permissions("auth.view_user"))
def my_rpc_method_with_builtin_predicate(username: str):
    user = User.objects.get(username=username)
    return f"{user.first_name} {user.last_name}"

@server.register_procedure(auth=forbid_bots_access)
def my_rpc_method_with_custom_authentication(a, b):
    return a + b
```

## 1.12 Contribution guide

### 1.12.1 Introduction

django-modern-rpc is an open-source project maintained by a single developer on his free time. Any contribution is welcome!

If you find an issue or want to ask for a new feature, open a new ticket on the project's issue tracker: https://github.com/alorence/django-modern-rpc/issues

If you want to solve an existing issue or implement a new feature, you can fork the project, push your changes and open a pull request: https://github.com/alorence/django-modern-rpc/pulls

Before submitting, ensure:

- Tests pass locally (at least with your preferred python version, preferably with all supported versions). See *Matrix testing*.

- Linting and formatting are applied (Ruff), and mypy passes for supported code paths. See *Linting, formatting* and *Type checking*
- Documentation is updated according to your changes. See *Documentation*

This section explains everything you need to know to set up the project locally, run tests, linter, formatter, type checker and build the documentation.

#### Local environment

This project and its dependencies are managed with uv. This is the only tool needed to start hacking on the project. Uv can be used on any operating system to install a supported Python version, create a virtual environment, install dependencies and run *nox* (see *Optional tool*).

See https://docs.astral.sh/uv/getting-started/installation/ for detailed installation docs

#### Minimal setup

Setting up a basic local environment is very quick. Simply run:

```
uv sync
```

This will install project's requirements and everything you need to run tests (i.e: dependencies from groups `dev` and `tests` are always installed).

#### Optional tool

Nox is used as a task manager to:

- Run tests against multiple combinations of Python / Django versions
- Run pre-defined commands to simplify some usual tasks (like a Makefile, but cross-platform)

You can install it globally on your system:

```
uv tool install nox
nox -l
```

Or, you can run it directly using uvx (installed with uv):

```
uvx nox -l
```

In this document, we will assume `nox` is installed globally, but it will work just the same when used through `uvx`.

### 1.12.2 Running tests

#### Current environment

The project's tests use pytest. In the local environment (default Python version), run them with one of the following commands:

```
# Run pytest with default arguments
uv run pytest

# To speedup tests execution, you can use pytest-xdist plugin to parallelize on multiple␣
↪cores
uv run pytest -n auto
```

(continues on next page)

(continued from previous page)

```
# Run only a few tests, filtering by keyword
uv run pytest -k xmlrpc

# Run a specific test
uv run pytest -q tests/test_e2e.py::TestXmlRpc::test_xml_rpc_standard_call

# Display duration of 20 slowest tests
uv run pytest --durations=20
```

Alternatively, use **nox**:

```
# Run pytest with default arguments (with parallelization)
nox -s tests:current-venv

# Customize pytest args
nox -s tests:current-venv -- -k multicall
```

### Matrix testing

Use `nox` to run tests across multiple Python and Django versions supported by the project. Nox will use uv to automatically install the right python version when missing.

```
# Run all supported Python / Django version combinations
nox

# Run test suites with Python 3.12 (using tag)
nox -t py312

# Run test suites with Django 5.2 (using tag)
nox -t dj52

# Run a specific test suite
nox -s 'tests(Python 3.13 × Django 5.1)'

# Run a specific test in all supported test suites (a.k.a pass command arguments to␣
↪pytest)
nox -- -k jsonrpc
```

**Note**

`noxfile.py` define tags in the form `py<digits>` and `dj<digits>`, e.g. `py312` for Python 3.12 and `dj52` for Django 5.2

### Tests tips and pitfalls

- The test project used by pytest-django lives under `tests/project` and exposes two routes: `/rpc` (sync) and `/async_rpc` (async). End-to-end tests use these.
- Serializer/deserializer backends are driven by settings in `tests.project.settings` and parametrized by fixtures such as `all_*_serializers` and `all_*_deserializers`.
- Async tests are supported and automatically detected (`asyncio_mode=auto`).

- Parallelization is enabled by default. Disable with `-n 0` for tests that are not parallel-safe.

### 1.12.3 Tests coverage

pytest-cov and coverage are used to compute code coverage analytics. To get it use:

```
# Basic coverage (default options)
uv run pytest --cov

# Custom output type (see https://pytest-cov.readthedocs.io/en/latest/reporting.html
↪#reporting)
uv run pytest --cov --cov-report term-missing
```

Alternatively, use **nox**:

```
# Basic coverage (default options)
nox -s tests:coverage

# Custom output type (see https://pytest-cov.readthedocs.io/en/latest/reporting.html
↪#reporting)
nox -s tests:coverage -- term-missing
```

### 1.12.4 Benchmarks

To compare the performances of various backends and compare sync and async views behaviors, pytest-benchmark is used. By default, benchmarks are disabled by `addopts` in `pyproject.toml`. To run them in your current environment:

```
uv run pytest tests/benchmarks --benchmark-enable
```

Alternatively, use **nox**:

```
# Run benchmarks with all supported Python versions
nox -s benchmarks

# Run benchmarks with specific python version
nox -s benchmarks -t py312

# Display duration of 20 slowest tests
nox -s tests:duration
```

> **Note**
>
> Do not use xdist when running benchmarks (`-n auto`); pytest-benchmark does not support it.

### 1.12.5 Linting, formatting

This project uses Ruff for linting and formatting

```
# Linting
uv run ruff check .

# Linting (apply basic fixes when possible)
uv run ruff check . --fix
```

(continues on next page)

(continued from previous page)

```
# Formatting
uv run ruff format .
```

Alternatively, use **nox**:

```
# Run linting with all supported Python versions
nox -s lint

# Linting (apply basic fixes when possible)
nox -s lint:fix

# Formatting
nox -s format
```

### 1.12.6 Type checking

**Important**

The requirements to use type checker are not installed by default. Make sure you ran **uv sync --group type-checking** if you want to run it directly with **uv run**. Nox task will do it automatically

This project uses Mypy to check type hints consistency

```
uv run mypy .
```

Alternatively, use **nox**:

```
nox -s mypy
```

### 1.12.7 Documentation

Documentation is generated with Sphinx. The easiest way to build it is using `nox`:

```
nox -s docs:build
```

Generated files can be found into `dist/docs`. Another task can be used to wipe this directory

```
nox -s docs:clean
```

To simplify edition of the documentation, sphinx-autobuild plugin is available. When used, the HTML documentation is served through a minimal HTTP server from `http://localhost:8001`

```
nox -s docs:serve
```

**Important**

The requirements to build docs are not installed by default. Make sure you ran **uv sync --group docs** if you want to use them directly with **uv run**.

## 1.13 Protocols references

### 1.13.1 XML-RPC

XML-RPC protocol was first elaborated by Dave Winer (ref) in 1998. The most recent XML-RPC specification page used as reference by django-modern-rpc is http://xmlrpc.com/spec.md.

The original website describing and specifying the protocol (xmlrpc.scripting.com) now redirects to https://xmlrpc.com. An archive of this website is available at http://1998.xmlrpc.com.

All these websites were created and maintained by Dave Winer, the creator of the protocol.

#### Introspection procedures

XML-RPC initial specification does not provide anything to achieve introspection (list procedures, their signatures and related documentation), but this was proposed in an unofficial addendum.

System introspection procedures (*system.listMethods*, *system.methodHelp* and *system.methodSignature*) are now widely implemented in various XML-RPC server libraries.

#### Multicall

Multicall was first proposed by Eric Kidd on Jan. 2001. Since the original article is now gone from the internet, only archived versions are currently available.

References

- https://web.archive.org/web/20060624230303/http://www.xmlrpc.com:80/discuss/msgReader\protect\T1\textdollar1208?mode=topic

Like the 3 other system methods, this one is not part of the spec. But its behavior has been well defined by Eric Kidd. It is now implemented in most XML-RPC servers and supported by a number of clients (including Python's ServerProxy).

This method can be used to make many RPC calls at once, by sending an array of RPC payloads. The result is a list of responses, with the result for each individual request, or a corresponding fault result.

It is available only to XML-RPC clients, since the JSON-RPC protocol specifies how to call multiple RPC methods at once using batch requests.

#### Fault codes

XML-RPC initial specification does not define a list of common errors and related *faultCode*. In 2001, Dan Libby tried to specify such fault codes as an extension of the spec. The original document is now only available from Wayback Machine : https://web.archive.org/web/20240416231938/https://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php

Here is a list of defined codes and corresponding error description

Table 1: Dan Libby's proposal for standardized error codes

| | |
|---|---|
| -32700 | parse error. not well formed |
| -32701 | parse error. unsupported encoding |
| -32702 | parse error. invalid character for encoding |
| -32600 | server error. invalid xml-rpc. not conforming to spec. |
| -32601 | server error. requested method not found |
| -32602 | server error. invalid method parameters |
| -32603 | server error. internal xml-rpc error |
| -32500 | application error |
| -32400 | system error |
| -32300 | transport error |

#### Other useful links

- Eric Kidd's XML-RPC How To: https://tldp.org/HOWTO/XML-RPC-HOWTO/index.html

### 1.13.2 JSON-RPC

JSON-RPC specification is more recent. The main documentation is available at https://www.jsonrpc.org/specification

The current official standard for JSON format is RFC 8259.

#### Introspection procedures & multicall

XML-RPC introspection procedures (*system.listMethods*, *system.methodHelp* and *system.methodSignature*) are not included in the official JSON-RPC specification. But since they are perfectly compatible with the JSON-RPC protocol, the builtin procedures defined in django-modern-rpc are automatically available through JSON-RPC calls.

However, since the JSON-RPC spec explicitly defines *Batch requests* as a way to call multiple procedures from a single request, the builtin *system.multicall* is disabled when a server is called through JSON-RPC.

#### Batch requests

django-modern-rpc fully supports batch requests as defined in the spec. When a server exposes its async view, a batch request will execute procedures concurrently using `asyncio.gather()`. When a server exposes the sync view, procedures are executed sequentially.

#### Error codes

JSON-RPC original specification defines a list of official error codes based on the list created by Dan Libby as an XML-RPC extension. See https://www.jsonrpc.org/specification#error_object

Table 2: JSON-RPC standardized error codes

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

### 1.13.3 Types support

#### Default types support

Most of the time, django-modern-rpc will serialize and deserialize all common scalar and non-scalar types.

| Data type | XML-RPC | JSON-RPC | Python conversion |
|---|---|---|---|
| null / nil | ✓ (1) | ✓ | None |
| boolean | ✓ | ✓ | bool |
| integer | ✓ | ✓ | int |
| string | ✓ | ✓ | str |
| double | ✓ | ✓ | float |
| array | ✓ | ✓ | list |
| struct | ✓ | ✓ | dict |
| date / dateTime.iso8601 | ✓ (2) | (2) | datetime.datetime |
| base64 | ✓ (3) | | bytes |

### Specific cases

#### null and NoneType (1)

In the original XML-RPC specification, there is no support for *null* values. An extension has been proposed in 2001 to add this type. It is currently fully supported by all backends.

See *XML-RPC backends* for detailed documentation of *null* support in each backend.

JSON-RPC backends will transparently convert *null* value to Python *None* and vice versa.

#### Date / Datetime (2)

XML-RPC spec defines the type *dateTime.iso8601* to handle dates and datetimes. The default behavior depends on the configured backend.

See *XML-RPC backends* for detailed documentation of *dateTime.iso8601* support in each backend.

JSON-RPC backends have no specific support of dates. The default behavior of builtin backends is:

- **Deserialization (RPC method argument)**

  Dates are received as standard string. Unmarshaller will NOT try to recognize dates for high level conversion. You can use `modernrpc.helpers.get_builtin_date` to easily retrieve a proper *datetime* instance in such case.

- **Serialization (RPC method return type)**

  *datetime.datetime*, *datetime.date* and *datetime.time* objects will be automatically converted to string (format ISO 8601). This is configured per backend, either using a custom JSONEncoder based on Django's DjangoJSONEncoder or by defining a `default` callback used in serialization process.

See *JSON-RPC backends* for detailed documentation of *date* / *time* / *datetime* support in each backend.

#### base64 (3)

XML-RPC defines a <base64> type to transport arbitrary binary data. django-modern-rpc maps it to Python `bytes`:

- **Serialization (RPC method return type)**

  When a procedure returns a `bytes` (or `bytearray`) object, it is base64-encoded and wrapped in a <base64> element.

- **Deserialization (RPC method argument)**

  A <base64> value received in a request is decoded back into a `bytes` object before being passed to your procedure.

JSON-RPC has no equivalent binary type. If you need to exchange binary data over JSON-RPC, encode it yourself (for example as a base64 string) and decode it inside your procedure.

### 1.13.4 Logging

Internally, django-modern-rpc uses Python logging system. While messages are usually hidden by default Django logging configuration, you can easily show them if needed.

You only have to configure `settings.LOGGING` to handle log messages from `modernrpc` module. Here is a basic example of such a configuration:

Listing 63: settings.py

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        # Your formatters configuration...
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        # your other loggers configuration
        'modernrpc': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
    }
}
```

All information about logging configuration can be found in official Django docs.

## 1.14 Changelog

### 1.14.1 v2.1.0

**Release date: 2026-06-05**

#### Improvements

- A new JSON-RPC backend based on the third party library ujson has been added.
- It is now possible to call a procedure without the "params" argument in a multicall request.

#### Breaking Changes

- `AuthenticationError` now uses a dedicated error code (`-32098`) instead of the generic internal error code (`-32603`). Clients relying on the error code to detect authentication failures should update their checks accordingly.

### Misc

- Dropped support for Django < 4.2
- Dropped support for Python < 3.10

## 1.14.2 v2.0.0

**Release date: 2025-11-10**

### Improvements

- The new API, heavily inspired from tools like FastAPI or django-ninja, encapsulates the procedures registration in a dedicated `RpcServer` instance. This removes the need to lookup modules based on `settings.MODERNRPC_METHODS_MODULES` value, so `modernrpc` does not need to be added to *settings.INSTALLED_APPS* anymore.
- `RpcNamespace` was added to provide a better organization of procedures. Each one can be registered in a previously defined `RpcServer`.
- It is now possible to register `async` procedures. Both sync (legacy) and async procedures are served by the default view (synchronous).
- For improved performances with `async` procedures, an `async` view has been added and can be used as a replacement to expose procedures. The `async_view` can serve both sync and async procedures.
- Error handling has been improved, allowing execution of a callback function when an exception is caught and before an RPC error response is built.
- The authentication process has been improved. Multiple callbacks can be configured at server-level, namespace-level or directly on a specific remote procedure.
- It is now possible to configure different backends to deserialize XML-RPC and JSON-RPC requests and to serialize XML-RPC and JSON-RPC responses. Alternative backends may provide more features, configuration options, specific types support or better performances.

### Breaking Changes

- Complete architecture redesign: The library now uses a server-based approach instead of entry points
- Removed the `RPCEntryPoint` class-based view in favor of the new function based views provided by `RpcServer` class
- Removed automatic procedure registration via `MODERNRPC_METHODS_MODULES` setting
- Removed HTML documentation generation through entry points
- Changed the way procedures access request context, now using the `context_target` parameter
- Some settings were removed:
  - MODERNRPC_METHODS_MODULES
  - MODERNRPC_LOG_EXCEPTIONS
  - MODERNRPC_DEFAULT_ENTRYPOINT_NAME
  - MODERNRPC_JSON_DECODER
  - MODERNRPC_JSON_ENCODER
  - MODERNRPC_XMLRPC_USE_BUILTIN_TYPES
  - MODERNRPC_XMLRPC_ALLOW_NONE

  - MODERNRPC_XMLRPC_DEFAULT_ENCODING

#### Misc

- Added support for Django 5.2 and 6.0
- Added support for Python 3.14
- Dropped support for Django < 3.2
- Dropped support for Python 3.7

### 1.14.3 v1.1.0

**Release date: 2025-01-01**

#### Improvements

- JSON-RPC handler now performs request validation in a dedicated method, allowing better customization of that part. Inspired by #73, thanks to @hwalinga

#### Fixes

- JSON-RPC handler now correctly checks type of request "id" field, according to the specification (#72). Thanks to @moonburnt
- Fixed docstring parser to allow empty or multi-line arg description
- Fixed registration of methods with same names under different entry points (#74). Thanks to @hwalinga

#### Misc

- Added support for Django 5.1
- Added support for Python 3.13
- Dropped support of Django 2.1

### 1.14.4 v1.0.3

**Release date: 2024-02-29**

#### Improvements

- When package `defusedxml` is installed in the same environment, builtin `xmlrpc.client` module is patched to reduce the risk of various XML payload based attacks on the server

#### Fixes

- Fix dependency to `packaging` introduced in previous release.

#### Misc

- Added support for Django 5.0 (Thanks to @washeck)
- Reduced the verbosity of the package. Startup initialization message is now a DEBUG log instead of an INFO one (Thanks to @washeck)
- Dropped use of Black and PyLint. Use Ruff to enforce all linting rules and code formatting

### 1.14.5 v1.0.2

**Release date: 2023-11-27**

#### Fixes

- When request is received with an invalid Content-Type (or missing one), the error response is now returned with a "text/plain" Content-Type header.

#### Misc

- Dropped support of python 3.5 and 3.6
- Added support for python 3.12 and Django 4.2

### 1.14.6 v1.0.1

**Release date: 2023-01-26**

#### Fixes

- Fixed invalid argument used to initialize default handlers instances (#52). Thanks to @washeck

### 1.14.7 v1.0.0

**Release date: 2023-01-03**

After months of work, the 1.0 milestone is a major refactoring of the library. Many parts of the project have been modernized to improve readability and robustness, and a few issues were fixed.

#### Improvements

- Type hinting is now supported in RPC methods. Auto-generated documentation will use it when it is defined. Old-style "doctypes" are still supported.
- Dependency to `six` has been removed

#### Breaking Changes

- When an authentication error is raised, the returned status code is now 200 instead of 403 for consistency with batch and system.multicall requests (#35)
- Django < 2.1 and Python < 3.5 support has been dropped.

#### Other API changes

- A new `modernrpc.core.Protocol` enum has been introduced to enforce correct protocol value when needed. (#29, #30). This new class replaces `modernrpc.core.JSONRPC_PROTOCOL` and `modernrpc.core.XMLRPC_PROTOCOL` but aliases were created for backward compatibility.
- `RPCUnknownMethod` exception has been renamed to `RPCMethodNotFound`. An alias has been created for backward compatibility

#### Fixes

- Initialization process updated: exceptions are now raised on startup for invalid RPC modules. In addition, Django check system is used to notify common errors. This was requested multiple times (#2, #13, #34).
- JSON-RPC notification behavior has been fixed to respect standard. Requests without `id` are handled as notifications but requests with null `id` are considered invalid and will return an error

- Batch request behavior has been fixed when one or more results failed to be serialized
- Builtin `system.methodSignature` behavior has been updated to respect standard. It now returns a list of list and unknown types are returned as "undef" (see http://xmlrpc-c.sourceforge.net/introspection.html)

#### Misc

- Added support for Python 3.9, 3.10 and 3.11
- Added support for Django 3.2, 4.0 and 4.1
- Documentation tree was completely reworked for clarity and simplicity. A new theme (Book) is now used to improve readability. See https://django-modern-rpc.rtfd.io.
- Poetry is now used to configure project dependencies and build distributions. The new `pyproject.toml` file replaces `setup.py`, `setup.cfg`, `MANIFEST.in` and `requirements.txt` to centralize all dependencies, external tools settings (pytest, flake8, etc.) and packaging configuration
- Black is now used to automatically format code
- Mypy is now used to verify type hints consistency
- Tox configuration now includes pylama, mypy, pylint and black environments
- All tests have been rewritten to have a strong separation between unit and functional tests. Test classes were created to group tests by similarities. Many fixtures have been added, with more parameterization, resulting in about 350 tests executed covering more than 95% of the code.

### 1.14.8 v0.12.1

**Release date: 2020-06-11**

#### Fixes

- Fix `ImportError` with Django 3.1

### 1.14.9 v0.12.0

**Release date: 2019-12-05**

#### Misc

- Django 2.1, 2.2 and 3.0 are now officially supported. Thanks to @atodorov for 3.0 compatibility
- Added Python 3.7 and 3.8 support
- Dropped Python 3.3 support

#### Improvements

- To ensure compatibility with JSON-RPC 1.2, 2 more "Content-Type" values are supported by JSON-RPC Handler: "application/json-rpc" and "application/jsonrequest" (#24). Thanks to @dansan

### 1.14.10 v0.11.1

**Release date: 2018-05-13**

### Improvements

Last release introduced some undocumented breaking API changes regarding RPC registry management. Old API has been restored for backward compatibility. The following global functions are now back in the API:

- modernrpc.core.register_rpc_method()
- modernrpc.core.get_all_method_names()
- modernrpc.core.get_all_methods()
- modernrpc.core.get_method()
- modernrpc.core.reset_registry()

In addition, some improvements have been applied to unit tests, to make sure test environment is the same after each test function. In addition, some exclusion patterns have been added in .coveragerc file to increase coverage report accuracy.

## 1.14.11 v0.11.0

**Release date: 2018-04-25**

### Improvements

- Django 2.0 is now officially supported. Tox and Travis default config have been updated to integrate Django 2.0 in existing tests environments.
- Method's documentation is generated only if needed and uses Django's @cached_property decorator
- HTML documentation default template has been updated: Bootstrap 4.1.0 stable is now used, and the rendering has been improved.
- Many units tests have been improved. Some tests with many calls to LiveServer have been split into shorter ones.

### API Changes

- Class RPCRequest has been removed and replaced by method `execute_procedure(name, args, kwargs)` in `RPCHandler` class. This method contains common logic used to retrieve an RPC method, execute authentication predicates to make sure it can be run, execute the concrete method and return the result.
- HTML documentation content is not marked as "safe" anymore, using `django.utils.safestring.mark_safe()`. You have to use Django decorator `safe` in your template if you display this value.

### Settings

- The `kwargs` dict passed to RPC methods can have customized keys (#18). Set the following values:
  - `settings.MODERNRPC_KWARGS_REQUEST_KEY`
  - `settings.MODERNRPC_KWARGS_ENTRY_POINT_KEY`
  - `settings.MODERNRPC_KWARGS_PROTOCOL_KEY`
  - `settings.MODERNRPC_KWARGS_HANDLER_KEY`

to override dict keys and prevent conflicts with your own methods arguments.

## 1.14.12 v0.10.0

**Release date: 2017-12-06**

### Improvements

- Logging system / error management
  - In case of error, current exception stacktrace is now passed to logger by default. This allows special handler like `django.utils.log.AdminEmailHandler` or `raven.handlers.logging.SentryHandler` to use it to report more useful information (#13)
  - Error messages have been rewritten to be consistent across all modules and classes
  - Decrease log verbosity: some `INFO` log messages now have `DEBUG` level (startup methods registration)
- Documentation has been updated
  - Added a page to explain how to configure RPC methods documentation generation, and add a note to explicitly state that `markdown` or `docutils` package must be installed if `settings.MODERNRPC_DOC_FORMAT` is set to non-empty value (#16)
  - Added a page to list implemented system introspection methods
  - Added a bibliography page, to list all references used to write the library
- Default template for generated RPC methods documentation now uses Bootstrap 4.0.0-beta.2 (previously 4.0.0-alpha.5)

## 1.14.13 v0.9.0

**Release date: 2017-10-03**

This is a major release with many improvements, protocol support and bug fixes. This version introduces an API break, please read carefully.

### Improvements

- Class `RPCException` and its subclasses now accept an additional `data` argument (#10). This is used by JSON-RPC handler to report additional information to user in case of error. This data is ignored by XML-RPC handler.
- JSON-RPC: Batch requests are now supported (#11)
- JSON-RPC: Named parameters are now supported (#12)
- JSON-RPC: Notification calls are now supported. Missing "id" in a payload is no longer considered as invalid, but is correctly handled. No HTTP response is returned in such case, according to the standard.
- XML-RPC: exception raised when serializing data to XML are now caught as `InternalError` and a clear error message

### API Changes

- Both `modernrpc.handlers.JSONRPC` and `modernrpc.handlers.XMLRPC` constants were moved and renamed. They become respectively `modernrpc.core.JSONRPC_PROTOCOL` and `modernrpc.core.XMLRPC_PROTOCOL`
- `RPCHandler` class updated, as well as subclasses `XMLRPCHandler` and `JSONRPCHandler`. `RPCHandler.parse_request()` is now `RPCHandler.process_request()`. The new method does not return a tuple `(method_name, params)` anymore. Instead, it executes the underlying RPC method using new class `RPCRequest`. If you customized your handlers, please make sure you updated your code (if needed).

#### Fixes

- Code has been improved to prepare future compatibility with Django 2.0

### 1.14.14 v0.8.1

**Release date: 2017-10-02**

> **important**
>
> This version is a security fix. Upgrade is highly recommended

#### Security fix

- Authentication backend is correctly checked when executing method using `system.multicall()`

### 1.14.15 v0.8.0

**Release date: 2017-07-12**

#### Fixes

- Fixed invalid HTML tag rendered from RPC Method documentation. Single new lines are converted to space since they are mostly used to limit docstrings line width. See pull request #7, thanks to @adamdonahue
- Signature of `auth.set_authentication_predicate` has been fixed. It can now be used as decorator (#8). See the documentation for details. Thanks to @aplicacionamedida

### 1.14.16 v0.7.1

**Release date: 2017-06-24**

#### Fixes

- Removed useless settings variable introduced in the v0.7.0 release. Logging capabilities are now enabled by simply configuring a logger for `modernrpc.*` modules, using Django variable `LOGGING`. The documentation has been updated accordingly.

### 1.14.17 v0.7.0

**Release date: 2017-06-24**

#### Improvements

- Default logging behavior has changed. The library will not output any log anymore, unless `MODERNRPC_ENABLE_LOGGING` is set to True. See docs for details

### 1.14.18 v0.6.0

**Release date: 2017-05-13**

#### Improvements

- Django cache system was previously used to store the list of available methods in the current project. This was useless, and caused issues with some cache systems (#5). Use of cache system has been removed. The list of RPC methods is computed when the application is started and kept in memory until it is stopped.

### 1.14.19 v0.5.2

**Release date: 2017-04-18**

#### Improvements

- HTTP Basic Authentication backend: User instance is now correctly stored in current request after successful authentication (#4)
- Unit testing with Django 1.11 is now performed against release version (Beta and RC are not tested anymore)
- Various Documentation improvements

### 1.14.20 v0.5.1

**Release date: 2017-03-25**

#### Improvements

- When RPC methods are registered, if a module file contains errors, a python warning is produced. This ensures the message will be displayed even if the logging system is not configured in a project (#2)
- Python 2 strings standardization. Allows configuring an automatic conversion of incoming strings, to ensure they have the same type in RPC method, no matter what protocol was used to call it. Previously, due to different behavior between JSON and XML deserializers, strings were received as `str` when method was called via XML-RPC and as `unicode` with JSON-RPC. This standardization process is disabled by default, and can be configured for the whole project or for specific RPC methods.
- Tests are performed against Django 1.11rc1
- `modernrpc.core.register_method()` function was deprecated since version 0.4.0 and has been removed.

### 1.14.21 v0.5.0

**Release date: 2017-02-18**

#### Improvements

- Typo fixes
- JSON-RPC 2.0 standard explicitly allows requests without 'params' member. This doesn't produce error anymore.
- Setting variable `MODERNRPC_XML_USE_BUILTIN_TYPES` is now deprecated in favor of `MODERNRPC_XMLRPC_USE_BUILTIN_TYPES`
- Unit tests are now performed with python 3.6 and Django 1.11 alpha, in addition to supported environment already tested. This is a first step to full support for these environments.
- HTTP "Basic Auth" support: it is now possible to define RPC methods available only to specific users. The control can be done on various user attributes: group, permission, superuser status, etc. Authentication backend can be extended to support any method based on incoming request.

### 1.14.22 v0.4.2

**Release date: 2016-11-20**

#### Improvements

- Various performance improvements
- Better use of logging system (python builtin) to report errors & exceptions from library and RPC methods
- Rewritten docstring parser. Markdown and reStructured formatters are still supported to generate HTML documentation for RPC methods. They now have unit tests to validate their behavior.
- @rpc_method decorator can be used with or without the parenthesis (and this feature is tested)
- System methods have been documented

### 1.14.23 v0.4.1

**Release date: 2016-11-17**

#### Improvements

- Method arguments documentation keeps the same order as defined in docstring
- API change: `MODERNRPC_ENTRY_POINTS_MODULES` setting has been renamed to `MODERNRPC_METHODS_MODULES`.
- A simple warning is displayed when `MODERNRPC_METHODS_MODULES` is not set, instead of a radical `ImproperlyConfigured` exception.
- Some traces have been added to allow debugging in the module easily. It uses the builtin logging framework.

### 1.14.24 v0.4.0

**Release date: 2016-11-17**

#### API Changes

- New unified way to register methods. Documentation in progress
- XML-RPC handler will now correctly serialize and deserialize None values by default. This behavior can be configured using `MODERNRPC_XMLRPC_ALLOW_NONE` setting.

#### Fixes

- When Django uses a persistent cache (Redis, memcached, etc.), ensure the registry is up-to-date with current sources at startup

### 1.14.25 v0.3.2

**Release date: 2016-10-26**

#### Fixes

- Include missing templates in pypi distribution packages

### 1.14.26 v0.3.1

**Release date: 2016-10-26**

### Improvements

- HTML documentation automatically generated for an entry point
- `system.multicall` is now supported, only in XML-RPC
- Many tests added

## 1.14.27 v0.3.0

**Release date: 2016-10-18**

### API Changes

- Settings variables have been renamed to limit conflicts with other libraries. In the future, all settings will have the same prefix.
  - `JSONRPC_DEFAULT_DECODER` becomes `MODERNRPC_JSON_DECODER`
  - `JSONRPC_DEFAULT_ENCODER` becomes `MODERNRPC_JSON_ENCODER`

  See https://github.com/alorence/django-modern-rpc/blob/master/modernrpc/conf/default_settings.py for more details
- Many other settings added, to make the library more configurable. See https://django-modern-rpc.rtfd.io/en/latest/basic_usage/settings.html

### Improvements

- RPC methods can now declare the special `**kwargs` parameter. The dict will contain information about current context (request, entry point, protocol, etc.)
- About 12 tests added to increase coverage
- Many documentation improvements
- `system.methodHelp` is now supported

## 1.14.28 v0.2.3

**Release date: 2016-10-13**

### Fixes

- Packages `modernrpc.tests` and `testsite` were excluded from Pypi distribution (both binary and source). This action was forgotten in the last release

## 1.14.29 v0.2.2

**Release date: 2016-10-13**

### Fixes

- Packages `modernrpc.tests` and `testsite` were excluded from Pypi distribution (both binary and source)

## 1.14.30 v0.2.1

**Release date: 2016-10-12**

#### Improvements

- Project is now configured to report tests coverage. See https://coveralls.io/github/alorence/django-modern-rpc
- Some documentation has been added, to cover more features of the library. See https://django-modern-rpc.rtfd.io/
- Many unit tests added to increase coverage
- `RPCEntryPoint` class can now be configured to handle only requests from a specific protocol

### 1.14.31 v0.2.0

**Release date: 2016-10-05**

#### Improvements

- Added very basic documentation: https://django-modern-rpc.rtfd.io/
- `system.listMethods` is now supported
- `system.methodSignature` is now supported
- Error reporting has been improved. Correct error codes and messages are returned on usual failures. See module `modernrpc.exceptions` for more information.
- Many unit tests have been added to increase test coverage of the library

### 1.14.32 v0.1.0

**Release date: 2016-10-02**

This is the very first version of the library. Only a subset of planned features were implemented.

#### Current features

- Work with Python 2.7, 3.3, 3.4 (Django 1.8 only) and 3.5
- Work with Django 1.8, 1.9 and 1.10
- JSON-RPC and XML-RPC simple requests support
- Multiple entry-points with defined list of methods and supported protocols

#### Not implemented yet

- No authentication support
- Unit tests don't cover all the code
- RPC system methods utility (`listMethods`, `methodSignature`, etc.) are not yet implemented
- There is no way to provide documentation in HTML form
- The library itself doesn't have any documentation (apart from the README.md)

# PYTHON MODULE INDEX

## m

# INDEX

## Symbols



## E



## G



## M