django-modern-rpc

Release 1.0.0

Antoine Lorence

Jan 03, 2023

CONTENTS

1 Getting started **3**

1.1 Quickstart . . . 3
1.2 Procedures registration . . . 5
1.3 Entrypoints configuration . . . 7
1.4 Error handling . . . 10
1.5 Settings . . . 11
1.6 Implementation details . . . 13
1.7 Authentication . . . 17
1.8 Procedures documentation . . . 20
1.9 Changelog . . . 21
1.10 Get involved . . . 32
1.11 Setup environment . . . 32

Index **35**

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

While it is a bit outdated now, there is still use-cases were XML-RPC or JSON-RPC server must be implemented. **Django-modern-rpc** will help you setup such a server as part of your Django project.

CHAPTER

ONE

GETTING STARTED

Important: This library requires python 3.5 and Django 2.1. If you need to install it on older Python/Django versions, you may need to install an older release.

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

1.1 Quickstart

1.1.1 Installation

Install `django-modern-rpc` in your environment, using pip or equivalent tool

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

Add `modernrpc` app to `settings.INSTALLED_APPS`:

Listing 1: myproject/settings.py

```
INSTALLED_APPS = [
    # ...
    'modernrpc',
]
```

1.1.2 Declare a procedure

Remote procedures are global Python functions decorated with `@rpc_method`.

Listing 2: myapp/remote_procedures.py

```
from modernrpc.core import rpc_method

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

`@rpc_method` behavior can be customized to your needs. Read *Procedures registration* for a full list of available options.

1.1.3 Locate procedures modules

Django-modern-rpc will automatically register functions decorated with `@rpc_method`, but needs a hint to locate them. Set `settings.MODERNRPC_METHODS_MODULES` variable to indicate project's modules where remote procedures are declared.

Listing 3: myproject/settings.py

```
MODERNRPC_METHODS_MODULES = [
    'myapp.remote_procedures'
]
```

1.1.4 Create an entry point

The entrypoint is a special Django view handling RPC calls. Like any other view, it must be declared in URLConf or any app specific `urls.py`:

Listing 4: myproject/urls.py

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

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

Entry points behavior can be customized to your needs. Read *Entrypoints configuration* for full documentation.

1.1.5 Test the server

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

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

Listing 6: 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 Procedures registration

1.2.1 Introduction

Any global python function can be exposed. Simply decorate it with `modernrpc.core.rpc_method`:

Listing 7: myproject/myapp/remote_procedures.py

```
from modernrpc.core import rpc_method

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

Django-modern-rpc will automatically register procedures at startup, as long as the containing module is listed in `settings.MODERNRPC_METHODS_MODULES`:

Listing 8: settings.py

```
MODERNRPC_METHODS_MODULES = [
    "myapp.remote_procedures",
]
```

Note: Automatic registration is performed in `modernrpc.apps.ModernRpcConfig.ready()`. See Django docs for additional information.

1.2.2 Customize registration

Without any argument, `@rpc_method` decorator will register the procedure with default parameters. It will be available for all entry points, any protocol (XML-RPC or JSON-RPC) and will have the function name as procedure's "methodName".

You can also change this behavior by adding arguments:

Procedure name

Use `name` to override the exposed procedure's "methodName". This is useful to setup a dotted name, which is not allowed in python.

Default: `name = None`

```
@rpc_method(name='math.add')
def add(a, b):
    return a + b
```

Protocol availability

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

Default: `protocol = Protocol.ALL`

```
from modernrpc.core import rpc_method, Protocol

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

Note: Don't forget to import `modernrpc.core.Protocol` enum.

Entry point

If you declared multiple entry points (see *Declare multiple entry points*) and want a procedure to be exposed only from one of them, provide its name using `entry_point` argument. You can also expose a procedure to 2 or more entry points by setting a list of strings.

Default: `entry_point = modernrpc.core.ALL`

```
# This will expose the procedure to "apiV2" entry point only"
@rpc_method(entry_point="apiV2")
def add(a, b):
    return a + b

# This will expose the procedure to 2 different entry points
@rpc_method(entry_point=["apiV2", "legacy"])
def multiply(a, b):
    return a * b
```

1.2.3 Access internal information

If you need to access some environment from your RPC method, simply adds `**kwargs` in function parameters. When the function will be executed, a dict will be passed as argument, providing the following information:

- Current HTTP request (`HttpRequest` instance)
- Current protocol (JSON-RPC or XML-RPC)
- Current entry point name
- Current handler instance

See the example to see how to access these values:

```
from modernrpc.core import rpc_method, REQUEST_KEY, ENTRY_POINT_KEY, PROTOCOL_KEY,␣
↪HANDLER_KEY

@rpc_method
def content_type_printer(**kwargs):

    # Get the current request
    request = kwargs.get(REQUEST_KEY)

    # Other available objects are:
    # protocol = kwargs.get(PROTOCOL_KEY)
    # entry_point = kwargs.get(ENTRY_POINT_KEY)
    # handler = kwargs.get(HANDLER_KEY)

    # Return the Content-Type of the current request
    return request.content_type
```

1.3 Entrypoints configuration

Django-modern-rpc provides a class-based view `modernrpc.views.RPCEntryPoint` to handle remote procedure calls.

1.3.1 Usage

`RPCEntryPoint` is a standard Django view, you can declare it in your project or app's `urls.py`:

Listing 9: urls.py

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

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

Then, all requests to `http://yourwebsite/rpc/` will be routed to the view. They will be inspected and parsed to be interpreted as RPC call. The result of procedure call will be encapsulated into a response according to the request's protocol (JSON-RPC or XML-RPC).

1.3.2 Advanced configuration

You can modify the behavior of the entry point by passing arguments to `as_view()`.

Restrict supported protocol

Using `protocol` parameter, you can make sure a given entry point will handle only JSON-RPC or XML-RPC requests. This can be used to setup protocol-specific paths.

Default: `protocol = Protocol.ALL`

Listing 10: urls.py

```
from django.urls import path

from modernrpc.core import Protocol
from modernrpc.views import RPCEntryPoint

urlpatterns = [
    path('json-rpc/', RPCEntryPoint.as_view(protocol=Protocol.JSON_RPC)),
    path('xml-rpc/', RPCEntryPoint.as_view(protocol=Protocol.XML_RPC)),
]
```

Declare multiple entry points

Using `entry_point` parameter, you can declare different entry points. Later, you will be able to configure your RPC methods to be available to one or more specific entry points (see *Procedure registration - entry point*)

Default: `entry_point = ALL`

Listing 11: urls.py

```
from django.urls import path

from modernrpc.views import RPCEntryPoint

urlpatterns = [
   path('rpc/', RPCEntryPoint.as_view(entry_point='apiV1')),
   path('rpcV2/', RPCEntryPoint.as_view(entry_point='apiV2')),
]
```

HTML documentation

`RPCEntryPoint` view can be configured to display HTML documentation of your procedures when it receive a GET request. To enable the feature, simply set `enable_doc = True` in your view instance.

Default: `enable_doc = False`

Listing 12: urls.py

```
from django.urls import path

from modernrpc.views import RPCEntryPoint
```

(continues on next page)

(continued from previous page)

```
urlpatterns = [
    # ...
    path('rpc/', RPCEntryPoint.as_view(enable_doc=True)),
]
```

The view will return HTML documentation on GET requests and process remote procedure calls on POST requests.

If you want a documentation-only entry point, set `enable_rpc = False` on documentation entry point.

Default: `enable_rpc = True`

Listing 13: urls.py

```
urlpatterns = [
    # By default, RPCEntryPoint does NOT provide documentation but handle RPC requests
    path('rpc/', RPCEntryPoint.as_view()),

    # And you can configure it to display doc without handling RPC requests.
    path('rpc-doc/', RPCEntryPoint.as_view(enable_doc=True, enable_rpc=False)),
]
```

The complete documentation is available here *Procedures documentation*

1.3.3 Reference

class modernrpc.views.**RPCEntryPoint**(***kwargs*)

This is the main entry point class. It inherits standard Django View class.

get_context_data(***kwargs*)

Update context data with list of RPC methods of the current entry point. Will be used to display methods documentation page

handler_classes

Return the list of handlers to use when receiving RPC requests.

post(*request*, **args*, ***kwargs*)

Handle an XML-RPC or JSON-RPC request.

Parameters

- **request** – Incoming request
- **args** – Unused
- **kwargs** – Unused

Returns A HttpResponse containing XML-RPC or JSON-RPC response with the result of procedure call

1.4 Error handling

1.4.1 Introduction

When remote procedures are executed, errors can be caused by almost anything: invalid request payload, arguments deserialization or result serialization error, exception in method execution, etc. All errors must be handled properly for 2 reasons :

1. An error response must be returned to RPC client (with an error code and a textual message)
2. Developers should be able to detect such errors (using logs, error reporting tool like Sentry, etc.)

For that reasons, django-modern-rpc handle all errors with Python builtin exception system. This allows for very flexible error handling and allows to define custom exceptions with fixed error code and message.

1.4.2 Builtin exceptions

Hopefully, error codes for both JSON-RPC and XML-RPC are pretty similar. The following errors are fully supported in django-modern-rpc.

Code	Message
-32700	parse error. not well formed
-32600	Invalid request
-32601	Method not found
-32602	Invalid params
-32603	Internal error

Additional errors from XML-RPC specs are less relevant in modern web application, and have not been implemented.

1.4.3 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:

```
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.

1.4.4 Log errors

New in version 1.0.

By default, when an exception is caught from modernrpc code, a stacktrace of the error will be printed in the default log output. This allows developer to detect such case and fix the issue if needed.

To disable this behavior, set *MODERNRPC_LOG_EXCEPTIONS* to *False*.

1.5 Settings

Django-modern-rpc behavior can be customized by defining some values in project's `settings.py`.

1.5.1 Basic settings

MODERNRPC_METHODS_MODULES

Required Yes

Default `[]` (Empty list)

Define the list of python modules containing RPC methods. You must set this list with at least one module. At startup, the list is looked up to register all python functions decorated with `@rpc_method`.

MODERNRPC_LOG_EXCEPTIONS

Required No

Default `True`

Set to `False` if you want to disable logging on exception catching

MODERNRPC_DOC_FORMAT

Required No

Default `""` (Empty string)

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

Possible values are: `""`, `rst` or `markdown`.

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_HANDLERS

Required No

Default `['modernrpc.handlers.JSONRPCHandler', 'modernrpc.handlers.XMLRPCHandler']`

List of handler classes used by default in any `RPCEntryPoint` instance. If you defined your custom handler for any protocol, you can replace the default class used

MODERNRPC_DEFAULT_ENTRYPOINT_NAME

Required No

Default `"__default_entry_point__"`

Default name used for anonymous `RPCEntryPoint`

1.5.2 Protocol specific

You can configure how JSON-RPC handler will serialize and unserialize data:

MODERNRPC_JSON_DECODER

Required No

Default `"json.decoder.JSONDecoder"`

Decoder class used to convert JSON data to python values.

MODERNRPC_JSON_ENCODER

Required No

Default `"django.core.serializers.json.DjangoJSONEncoder"`

Encoder class used to convert python values to JSON data. Internally, modernrpc uses the default Django JSON encoder, which improves the builtin python encoder by adding support for additional types (DateTime, UUID, etc.).

MODERNRPC_XMLRPC_USE_BUILTIN_TYPES

Required No

Default `True`

Control how builtin types are handled by XML-RPC serializer and deserializer. If set to True (default), dates will be converted to `datetime.datetime` by XML-RPC deserializer. If set to False, dates will be converted to XML-RPC DateTime instances (or equivalent for Python 2).

This setting will be passed directly to ServerProxy instantiation.

MODERNRPC_XMLRPC_ALLOW_NONE

Required No

Default `True`

Control how XML-RPC serializer will handle None values. If set to True (default), None values will be converted to *<nil>*. If set to False, the serializer will raise a `TypeError` when encountering a *None* value.

MODERNRPC_XMLRPC_DEFAULT_ENCODING

Required No

Default `None`

Configure the default encoding used by XML-RPC serializer.

MODERNRPC_XML_USE_BUILTIN_TYPES

Deprecated. Define `MODERNRPC_XMLRPC_USE_BUILTIN_TYPES` instead.

1.6 Implementation details

1.6.1 XML-RPC

The most recent XML-RPC specification page used as reference for django-modern-rpc development is http://xmlrpc.com/spec.md. It is part of xmlrpc.com, a website created by Dave Winer in 2019 to propose updated tools around XML-RPC standard.

The original website (xmlrpc.scripting.com) has also been archived with a new URL: 1998.xmlrpc.com

System introspection methods

System introspection methods (listMethods, methodHelp, methodSignature) were not part of the original standard but were proposed in an unofficial addendum. Here is a list of references pages:

- http://xmlrpc-c.sourceforge.net/introspection.html
- http://scripts.incutio.com/xmlrpc/introspection.html (dead)
- http://xmlrpc.usefulinc.com/doc/reserved.html) (dead)

Multicall

Multicall was first proposed by Eric Kidd on 2001-01. Since the original article is now gone from the internet, it has been archived at https://mirrors.talideon.com/articles/multicall.html

Other useful links

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

1.6.2 JSON-RPC

Since JSON-RPC specification is more recent, available documentation is easier to find. The main specification is available at https://www.jsonrpc.org/specification

The current official standard for JSON format is RFC 8259.

1.6.3 Types support

Most of the time, django-modern-rpc will serialize and unserialize

RPC Data type	XML-RPC	JSON-RPC	Python conversion
null	✓ (1)	✓	None
string	✓	✓	str
int	✓	✓	int
float	✓	✓	float
boolean	✓	✓	bool
array	✓	✓	list
struct	✓	✓	dict
date	✓	(2)	See (2)
bas64	✓ (3)	N/A	See (3)

(1) null and NoneType

By default, both JSON-RPC and XML-RPC handlers can serialize None and deserialize null value. The XML handler will convert such values to <*nil/*> special argument, JSON one will convert to JSON null.

But some old XML-RPC clients may misunderstand the <*nil/*> value. If needed, you can disable its support by setting *MODERNRPC_XMLRPC_ALLOW_NONE* to *False*. The XML-RPC marshaller will raise an exception on None serialization or <*nil/*> deserialization.

(2) Date types

JSON transport has no specific support of dates, they are transmitted as string formatted with ISO 8601 standard. The behavior of default encoder and decoder classes is:

- Input date (RPC method argument)
  - Dates are transmitted as standard string. Decoder will NOT try to recognize dates to apply specific treatment
- Output date (RPC method return type)
  - `datetime.datetime` objects will be automatically converted to string (format ISO 8601), so JSON-RPC clients will be able to handle it as usual. This behavior is due to the use of `DjangoJSONEncoder` as default encoder.

If you need to customize behavior of JSON encoder and/or decoder, you can specify another classes in `settings.py`:

```
MODERNRPC_JSON_DECODER = 'json.decoder.JSONDecoder'
MODERNRPC_JSON_ENCODER = 'django.core.serializers.json.DjangoJSONEncoder'
```

XML-RPC transport defines a type to handle dates and date/times: `dateTime.iso8601`. Conversion is done as follow:

- Input date (RPC method argument)
  - If `settings.MODERNRPC_XMLRPC_USE_BUILTIN_TYPES = True (default)`, the date will be converted to `datetime.datetime`
  - If `settings.MODERNRPC_XMLRPC_USE_BUILTIN_TYPES = False`, the date will be converted to `xmlrpc.client.DateTime` (Python 3) or `xmlrpclib.DateTime` (Python 2)
- Output date (RPC method return type)
  - Any object of type `datetime.datetime`, `xmlrpclib.DateTime` or `xmlrpc.client.DateTime` will be converted to `dateTime.iso8601` in XML response

To simplify dates handling in your procedures, you can use *get_builtin_date()* helper to convert any input into a buildin *datetime.datetime*.

`modernrpc.helpers.`**`get_builtin_date`**(*date: Union[str, datetime.datetime, xmlrpc.client.DateTime]*, *date_format: str = '%Y-%m-%dT%H:%M:%S'*, *raise_exception: bool = False*) → Optional[datetime.datetime]

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`** – The date object to convert.
- **`date_format`** – If the given date is a str, format is passed to strptime to parse it
- **`raise_exception`** – If set to True, an exception will be raised if the input string cannot be parsed

Returns A valid `datetime.datetime` instance

base64

base64 is not specifically supported, but you should be able to serialize and unserialize base64 encoded data as string.

1.6.4 Logging

Internally, django-modern-rpc use 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 14: settings.py

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        # Your formatters configuration...
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
```

(continues on next page)

(continued from previous page)

```
    },
    'loggers': {
        # your other loggers configuration
        'modernrpc': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
    }
}
```

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

Note: Logging configuration is optional. If not configured, the errors will still be visible in logs unless you set *MODERNRPC_LOG_EXCEPTIONS* to *False*. See Logging errors

1.6.5 System methods

XML-RPC specification doesn't provide default methods to achieve introspection tasks, but some people proposed a standard for such methods. The original document is now offline, but has been retrieved from Google cache and is now hosted here.

system.listMethods

Return a list of all methods available.

system.methodSignature

Return the signature of a specific method

system.methodHelp

Return the documentation for a specific method.

system.multicall

Like 3 others, this system method is not part of the standard. But its behavior has been well defined by Eric Kidd. It is now implemented most of the XML-RPC servers and supported by 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 payload. 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 JSON-RPC protocol specify how to call multiple RPC methods at once using batch request.

1.7 Authentication

django-modern-rpc provides a mechanism to check authentication before executing a given RPC method. It implemented at request level and is always checked before executing procedure.

Changed in version 1.0.0: In previous releases, authentication failures caused the view to return a 403 status code on response to standard (single) request, while batch requests / multicall always returned a 200 status with an error message. For consistency with XML-RPC specs, an authentication failure now returns a 200 response with a proper error (`error: -32603, message: Authentication failed when calling "<method_name>"`).

1.7.1 HTTP Basic Auth

django-modern-rpc comes with a builtin support for HTTP Basic Authentication. It provides a set of decorators to directly extract user information from request and test this user against Django authentication system.

```
from modernrpc.auth.basic import (
    http_basic_auth_login_required,
    http_basic_auth_superuser_required,
    http_basic_auth_permissions_required,
    http_basic_auth_any_of_permissions_required,
    http_basic_auth_group_member_required,
    http_basic_auth_all_groups_member_required
)
from modernrpc.core import rpc_method


@rpc_method
@http_basic_auth_login_required
def logged_user_required(x):
    """Access allowed only to logged users"""
    return x

@rpc_method
@http_basic_auth_superuser_required
def logged_superuser_required(x):
    """Access allowed only to superusers"""
    return x

@rpc_method
@http_basic_auth_permissions_required(permissions='auth.delete_user')
def delete_user_perm_required(x):
    """Access allowed only to users with specified permission"""
    return x

@rpc_method
@http_basic_auth_any_of_permissions_required(permissions=['auth.add_user', 'auth.change_
↪user'])
def any_permission_required(x):
    """Access allowed only to users with at least 1 of the specified permissions"""
    return x

@rpc_method
```

(continues on next page)

(continued from previous page)

```
@http_basic_auth_permissions_required(permissions=['auth.add_user', 'auth.change_user'])
def all_permissions_required(x):
    """Access allowed only to users with all the specified permissions"""
    return x

@rpc_method
@http_basic_auth_group_member_required(groups='A')
def in_group_A_required(x):
    """Access allowed only to users contained in specified group"""
    return x

@rpc_method
@http_basic_auth_group_member_required(groups=['A', 'B'])
def in_group_A_or_B_required(x):
    """Access allowed only to users contained in at least 1 of the specified group"""
    return x

@rpc_method
@http_basic_auth_all_groups_member_required(groups=['A', 'B'])
def in_groups_A_and_B_required_alt(x):
    """Access allowed only to users contained in all the specified group"""
    return x
```

1.7.2 Custom authentication system

To provide authentication features, django-modern-rpc introduce concept of "predicate". This example will show you how to build a custom authentication system to restrict RPC method execution to clients that present a User-Agent different from a known list of 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
    ]

    req_user_agent = request.headers["User-Agent"].lower()
    for bot_user_agent in [ua.lower() for ua in forbidden_bots]:
        # If we detect the caller is one of the bots listed above...
        if bot_user_agent in req_user_agent:
            # ... forbid access
            return False
```

(continues on next page)

(continued from previous page)

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

Note: A predicate always takes a request as argument and returns a boolean value

It is associated with RPC method using `@set_authentication_predicate` decorator.

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

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

Now, the RPC method becomes unavailable to callers if User-Agent is not provided or if it has an invalid value.

In addition, you can provide arguments to your predicate using `params`:

```
@rpc_method
@set_authentication_predicate(my_predicate_with_params, params=('param_1', 42))
def my_rpc_method(a, b):
    return a + b
```

It is possible to declare multiple predicates for a single method. In such case, all predicates must return True to allow access to the method.

```
@rpc_method
@set_authentication_predicate(forbid_bots_access)
@set_authentication_predicate(my_predicate_with_params, params=('param_1', 42))
def my_rpc_method(a, b):
    return a + b
```

1.8 Procedures documentation

Django-modern-rpc can optionally process the docstring attached to your RPC methods and display it in a web page. This article will explain how generated documentation can bu used and customized.

1.8.1 Enable documentation

RPCEntryPoint class can be configured to provide HTML documentation of your RPC methods. To enable the feature, simply set `enable_doc = True` in your view instance

```
urlpatterns = [

    # Configure the RPCEntryPoint directly by passing some arguments to as_view() method
    path('rpc/', RPCEntryPoint.as_view(enable_doc=True)),
]
```

If you prefer provide documentation on a different URL than the one used to handle RPC requests, you just need to specify two different URLConf.

```
urlpatterns = [

    # By default, RPCEntryPoint does NOT provide documentation but handle RPC requests
    path('rpc/', RPCEntryPoint.as_view()),

    # And you can configure it to display doc without handling RPC requests.
    path('rpc-doc/', RPCEntryPoint.as_view(enable_doc=True, enable_rpc=False)),
]
```

1.8.2 Customize rendering

You can customize the documentation page by setting your own template. `RPCEntryPoint` inherits `django.views.generic.base.TemplateView`, so you have to set view's `template_name` attribute:

```
urlpatterns = [
    # RPCEntryPoint can be configured with arguments passed to as_view()
    path(
      'rpc/',
      RPCEntryPoint.as_view(
        enable_doc=True,
        template_name='my_app/my_custom_doc_template.html'
      )
    ),
]
```

In the template, you will get a list of `modernrpc.core.RPCMethod` instance (one per registered RPC method). Each instance of this class has some methods and properties to retrieve documentation.

By default, documentation will be rendered using HTML5 vanilla tags with default classes and ids.

Changed in version 1.0.0: In previous releases, the default template was based on Bootstrap 4 with collapse components and accordion widgets. To use this BS4 template, simply set ``template_name='modernrpc/bootstrap4/doc_index.html'`` when instantiating the view.

1.8.3 Write documentation

The documentation is generated directly from RPC methods docstring

```
@rpc_method(name="util.printContentType")
def content_type_printer(**kwargs):
    """
    Inspect request to extract the Content-Type header if present.
    This method demonstrates how a RPC method can access the request object.
    :param kwargs: Dict with current request, protocol and entry_point information.
    :return: The Content-Type string for incoming request
    """
    # The other available variables are:
    # protocol = kwargs.get(MODERNRPC_PROTOCOL_PARAM_NAME)
    # entry_point = kwargs.get(MODERNRPC_ENTRY_POINT_PARAM_NAME)

    # Get the current request
    request = kwargs.get(REQUEST_KEY)
    # Return the content-type of the current request
    return request.META.get('Content-Type', '')
```

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

New in version 1.0.0: Typehints are now supported to generate arguments and return type in documentation

1.9 Changelog

1.9.1 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` have been completely 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 have 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 have 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 where 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.

1.9.2 v0.12.1

Release date: 2020-06-11

Fixes

- Fix `ImportError` with Django 3.1

1.9.3 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

1.9.4 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.

1.9.5 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.

1.9.6 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)

1.9.7 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

1.9.8 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()`

1.9.9 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

1.9.10 v0.7.1

Release date: 2017-06-24

Fixes

- Removed useless settings variable introduced in last 0.7.0 release. Logging capabilities are now enabled by simply configuring a logger for `modernrpc.*` modules, using Django variable `LOGGING`. The documentation has been updated accordingly.

1.9.11 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

1.9.12 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.

1.9.13 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

1.9.14 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. Allow to configure 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.

1.9.15 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.

1.9.16 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

1.9.17 v0.4.1

Release date: 2016-11-17

Improvements

- Method arguments documentation keep the same order as defined in docstring
- API change: `MODERNRPC_ENTRY_POINTS_MODULES` setting have 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.

1.9.18 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 unserialize None values by default. This behavior can be configured using `MODERNRPC_XMLRPC_ALLOW_NONE` setting.

Fixes

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

1.9.19 v0.3.2

Release date: 2016-10-26

Fixes

- Include missing templates in pypi distribution packages

1.9.20 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

1.9.21 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

1.9.22 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

1.9.23 v0.2.2

Release date: 2016-10-13

Fixes

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

1.9.24 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 have 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

1.9.25 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 fail cause. See module `modernrpc.exceptions` for more information.
- Many unit tests have been added to increase test coverage of the library

1.9.26 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 doesn'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)

1.10 Get involved

You can contribute to the project in multiple ways. Developer or not, all contributions are welcome.

1.10.1 Report issues, suggest enhancements

If you find a bug, want to ask question about configuration or suggest an improvement to the project, feel free to use the issue tracker. You will need a GitHub account. Please be kind and respectful, this project is maintained on free time by a single developer.

1.10.2 Submit a pull request

If you improved something or fixed a bug by yourself in a fork, you can submit a pull request. We will be happy to review it before doing a merge. The next page, *Setup environment* will explain how to configure a development environment in order to work on project source code.

1.11 Setup environment

Since 1.0, django-modern-rpc uses `poetry` as main tool to manage project dependencies, environments and packaging. It must be installed globally on your system.

1.11.1 Install poetry

There is multiple way to install `poetry`. Refer to official documentation and choose your preferred method.

If you already use pipx, installation is very quick and clean.

```
$ pipx install poetry
```

Alternatively, you can use the official *get-poetry.py* install script.

```
$ curl -sSL https://install.python-poetry.org | python3 -
```

1.11.2 Install dependencies

Dependencies are configured in `pyproject.toml` file. In addition, `poetry.lock` file contains resolved dependency tree. To install basic development environment, simply run

```
$ poetry install
```

Note: This command will automatically create a new environment if needed. You do not need to create it manually.

This will install everything needed to develop and test the library. In addition, optional group may be specified to enable other toolset (See below)

1.11.3 Run tests

The project have a lot of unit and functional tests to avoid regressions across new releases. They are automatically run on CI/CD platform (currently GitHub Actions) on each push, pull request and before every release.

But you should run tests on you development machine before submitting a new pull request.

System interpreter

To run test with your current python interpreter, simply run `pytest` inside poetry environment.

```
poetry run pytest
```

Python / Django versions matrix

If you have multiple python versions on your system, or if you have pyenv installed, you can setup tox and perform tests against multiple python / django versions

```
$ poetry install --with tox
$ pyenv local system 3.8 3.6 3.5
$ poetry run tox
```

To speedup tests run, you can use `-p` option to parallelize environment specific tests

```
$ poetry run tox -p 4
```

Caution: Don't run too much parallel threads or you may slow down or completely freeze your machine !

1.11.4 Build docs

If you need to update documentation, first install required dependencies

```
poetry install --with docs
```

Then, *cd* into docs directory and use Makefile pre-defined commands.

To build docs with required options:

Listing 15: from docs/ directory

```
poetry run make html
```

The built files are stored inside `dist/docs` folder.

To simplify the writing process, you can run `autobuild` which automatically watch changes on files, rebuild docs and enable LiveServer on compatible browsers

Listing 16: from docs/ directory

```
poetry run make serve
```

1.11.5 Code quality

The project uses linting and formatting tools to unify source code definition and remove most of the typo and typing issues. You can run any tool directly inside poetry environment, or run them directly using tox (to unify command lines options used).

```
poetry install --with black,pylint,mypy,pylama
poetry run tox -e black,pylint,mypy,pylama
```

Important: These tools are run on GitHub Actions and will break the build on errors. Don't forget to run the before submitting a pull request.

INDEX

G

get_builtin_date() (*in module modernrpc.helpers*), 15

get_context_data() (*modernrpc.views.RPCEntryPoint method*), 9

H

handler_classes (*modernrpc.views.RPCEntryPoint attribute*), 9

P

post() (*modernrpc.views.RPCEntryPoint method*), 9

R

RPCEntryPoint (*class in modernrpc.views*), 9