django-modern-rpc Documentation

Release 0.8.0

Antoine Lorence

Jul 12, 2017

Table of Contents

CHAPTER 1

Overview

Django-modern-rpc provides a simple solution to implement a remote procedure call (RPC) server as part of your Django project. It supports all major Django and Python versions.

Project's main features are:

- Simple and pythonic API
- Python 2.7, 3.3, 3.4, 3.5 and 3.6
- Django 1.8, 1.9, 1.10 and 1.11
- XML-RPC and JSON-RPC 2.0 support (JSON-RPC 1.0 not supported)
- HTTP Basic Auth support
- Custom authentication support
- Automatic protocol detection based on request's `Content-Type` header
- High-level error management based on exceptions
- Multiple entry points, with specific methods and protocol attached
- RPC Methods documentation generated automatically, based on docstrings
- System introspection methods:
  - system.listMethods()
  - system.methodSignature()
  - system.methodHelp()
  - system.multicall() (XML-RPC only, using specification from https://mirrors.talideon.com/articles/multicall.html)

Quick start

Start using django-modern-rpc in a minute, following these simple steps.

Installation

Use pip to install the package in your environment:

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

Configuration

Add the library to your Django applications, in your `settings.py`:

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

Create an entry point

An entry point is the URL used by clients to call your RPC methods. Declare it as follow:

```
# In myproject/rpc_app/urls.py
from django.conf.urls import url

from modernrpc.views import RPCEntryPoint

urlpatterns = [
    # ... other views

    url(r'^rpc/', RPCEntryPoint.as_view()),
]
```

Entry points behavior can be customized with some arguments. Read the page *Entry point configuration* for more information.

Write and register your methods

In the next step, you will declare which global functions should be available for remote calls. Use `@rpc_method` decorator to simply indicates those methods.

```
# In myproject/rpc_app/rpc_methods.py
from modernrpc.core import rpc_method

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

`@rpc_method` behavior can be customized with some arguments. Read the page *Configure the registration* for more information.

Declare your RPC methods modules

Django-modern-rpc will automatically register functions decorated with `@rpc_method`, but needs a hint to locate them. Declare `MODERNRPC_METHODS_MODULES` in your settings to indicate one or more python module where

your RPC methods are defined.

```
MODERNRPC_METHODS_MODULES = [
    'rpc_app.rpc_methods'
]
```

That's all !

Your application is ready to receive XML-RPC or JSON-RPC calls. The entry point URL is `http://yourwebsite.com/rpc/` but you can customize it to fit your needs.

Basic usage

RPC methods registration

Django-modern-rpc will automatically register RPC methods at startup. To ensure this automatic registration is performed quickly, you must provide the list of python modules where your remote methods are declared.

In `settings.py`, add the variable `MODERNRPC_METHODS_MODULES` to define this list. In our example, the only defined RPC method is `add()`, declared in `myproject/rpc_app/rpc_methods.py`.

```
MODERNRPC_METHODS_MODULES = [
    'rpc_app.rpc_methods'
]
```

When django-modern-rpc application will be loaded, it's AppConfig.ready() method is executed. The automatic registration is performed at this step.

Decorate your RPC methods

Decorator usage is simple. You only need to add `@rpc_method` decorator before any method you want to provide via RPC calls.

```
# In myproject/rpc_app/rpc_methods.py
from modernrpc.core import rpc_method

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

Configure the registration

If you decorate your methods with `@rpc_method` without specifying argument, the registered method will be available for all entry points, for any XML-RPC or JSON-RPC call and will have the name of the corresponding function.

You can also change this behavior by setting arguments to the decorator:

`name = None` Can be used to override the external name of a RPC method. This is the only way to define dotted names for RPC methods, since python syntax does not allows such names in functions definitions. Example:

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

protocol = ALL Set the protocol argument to `modernrpc.handlers.JSONRPC` or `modernrpc.handlers.XMLRPC` to ensure a method will be available **only** via the corresponding protocol. Example:

```
@rpc_method(protocol=modernrpc.handlers.JSONRPC)
def add(a, b):
    return a + b
```

entry_point = ALL Set the entry_point argument to one or more str value to ensure the method will be available only via calls to corresponding entry point name. Fore more information, please check the documentation about *multiple entry points declaration*. Example:

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

Access request, protocol and other info from a RPC method

See *Access to current request from RPC methods*.

Entry point configuration

Django-modern-rpc provides a class to handle RPC calls called `RPCEntryPoint`. This standard Django view will return a valid response to any valid RPC call made via HTTP POST requests.

Basic declaration

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

```
# In myproject/rpc_app/urls.py
from django.conf.urls import url

from modernrpc.views import RPCEntryPoint

urlpatterns = [
    # ... other views

    url(r'^rpc/', RPCEntryPoint.as_view()),
]
```

As a result, all RPC requests made to `http://yourwebsite/rpc/` will be handled by the RPC entry point. Obviously, you can decide to handle requests from a different URL by updating the `regex` argument of `url()`. You can also declare more entry points with different URLs.

Advanced entry point configuration

You can modify the behavior of the view by passing some arguments to `as_view()`.

Limit entry point to JSON-RPC or XML-RPC only

Using `protocol` parameter, you can make sure a given entry point will only handle JSON-RPC or XML-RPC requests. This is useful, for example if you need to have different addresses to handle protocols.

```
from django.conf.urls import url

from modernrpc.handlers import JSONRPC, XMLRPC
from modernrpc.views import RPCEntryPoint

urlpatterns = [
    url(r'^json-rpc/$', RPCEntryPoint.as_view(protocol=JSONRPC)),
    url(r'^xml-rpc/$', RPCEntryPoint.as_view(protocol=XMLRPC)),
]
```

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.

```
from django.conf.urls import url

from modernrpc.views import RPCEntryPoint

urlpatterns = [
    url(r'^rpc/$', RPCEntryPoint.as_view(entry_point='apiV1')),
    url(r'^rpcV2/$', RPCEntryPoint.as_view(entry_point='apiV2')),
]
```

Class reference

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

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

dispatch(*request*, **args*, ***kwargs*)

Overrides the default dispatch method, to disable CSRF validation on POST requests. This is mandatory to ensure RPC calls wil be correctly handled

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

get_handler_classes()

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

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

Handle a XML-RPC or JSON-RPC request.

Parameters

- **request** – Incoming request
- **args** – Additional arguments
- **kwargs** – Additional named arguments

Returns A HttpResponse containing XML-RPC or JSON-RPC response, depending on the incoming request

Settings

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

Basic configuration

MODERNRPC_METHODS_MODULES

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*.

JSON Serialization and deserialization

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

MODERNRPC_JSON_DECODER

Default: `'json.decoder.JSONDecoder'`

Decoder class used to convert python data to JSON

MODERNRPC_JSON_ENCODER

Default: `'django.core.serializers.json.DjangoJSONEncoder'`

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

XML serialization and deserialization

MODERNRPC_XMLRPC_USE_BUILTIN_TYPES

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

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

Default: `None`

Configure the default encoding used by XML-RPC serializer.

MODERNRPC_XML_USE_BUILTIN_TYPES

Default: `True`

Deprecated. Define `MODERNRPC_XMLRPC_USE_BUILTIN_TYPES` instead.

Python 2 String standardization

MODERNRPC_PY2_STR_TYPE

Default: `None`

Define target type for *Global String standardization (project level)*.

MODERNRPC_PY2_STR_ENCODING

Default: `UTF-8`

Define global encoding used in *Global String standardization (project level)*.

RPC entry points configuration

MODERNRPC_HANDLERS

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

Default: `'__default_entry_point__'`

Default name used for anonymous `RPCEntryPoint`

Other available settings

MODERNRPC_DOC_FORMAT

Default: `''` (Empty String)

Configure the format of the docstring used to document your RPC methods. Possible values are: '', 'rst' or 'md'

Advanced configuration

Data types support

JSON transport supported types are limited by JSON type system described in http://www.ietf.org/rfc/rfc4627.txt.

XML-RPC specification contains explicit type information. As a result, more types are supported. They are described in http://xmlrpc.scripting.com/spec.html.

In addition, Python version used in your project may change how data types are transmitted. Since django-modern-rpc allows you to declare methods that can handle both protocols, this document describes how specific types are handled in RPC methods in all cases (JSON or XML-RPC transport with Python 2 or Python 3).

Basic types

The basic types are handled the same way with the 2 supported protocols. Those types are:

- bool
- int
- float
- string (Python 3 only, see *Strings* for information with Python 2)

As long as a RPC method arguments or return value is of one of the above types, the behavior is consistent across all Python version and protocols.

List and structures

Both JSON-RPC and XML-RPC supports *lists* and *structures*. Conversion is done as follow:

- Input data (RPC method argument)
  - *structure* is converted to Python `dict`
  - *list* is converted to Python `list`
- Output data (RPC method return type)
  - Python `dict` is converted to *structure*
  - Python `list` and `tuple` is converted to *list*

In other words, you can use those types without any issue, it works as you expect it.

Both *lists* and *structures* can contains any combinations of elements of types defined in this documents. A *struct* can contain another *struct* or a *list*, etc.

null and NoneType

By default, both JSON-RPC and XML-RPC handlers will be able to return `None` or to take a None value as argument. The XML handler will convert such values to `<nil/>` special argument. Since this type is not part of the original specification, some XML-RPC clients may misunderstand this value. If you prefer respect the original standard, simply define in your `settings.py`:

```
MODERNRPC_XMLRPC_ALLOW_NONE = False
```

As a result, the XML handler will raise a `TypeError` when trying to serialize a response containing a `None` value.

Strings

If your project runs in a Python 3 environment, the behavior is consistent for XML-RPC and JSON-RPC protocol.

In a Python 2 project, XML deserializer will transmit string values as `str` when JSON deserializer will produce `unicode` values. If this behavior is problematic in your project, you have to manually handle both cases for each string you manipulate in your RPC methods. As an alternative, django-modern-rpc can dynamically standardize incoming arguments to ensure contained strings are converted to have always the same type from method point of view.

Note: The strings standardization apply on strings arguments, but also on list and structures. The process inspects recursively all arguments to perform the conversion of string values. This can be inefficient for big structures or lists, that's why this feature is not enabled by default.

You have 2 options to configure this process:

Global String standardization (project level)

In your `settings.py`, define the variable `MODERNRPC_PY2_STR_TYPE` with type value `str` or `unicode`. This will automatically converts any incoming string argument to the specified type. In such case, you will need to also configure `settings.MODERNRPC_PY2_STR_ENCODING` with the strings encoding (default is `UTF-8`)

In `settings.py`

```
MODERNRPC_PY2_STR_TYPE = str
MODERNRPC_PY2_STR_ENCODING = 'UTF-8'
```

In `rpc_methods`

```
@rpc_method
def print_incoming_type(data):
    """Returns a string representation of input argument type"""
    if isinstance(data, unicode):
        return 'Incoming arg is a unicode object'
    elif isinstance(data, str):
        return 'Incoming arg is a str object'

    return 'Incoming arg has type {}'.format(type(data))
```

In this example, calling `print_incoming_type('abcd')` from a Python 2 project will always return `Incoming arg is a str object`, no matter which protocol were used to make the request (JSON-RPC or XML-RPC)

Method level String standardization

In the same way, if you need to have a different behavior for a specific RPC method, the equivalent of `settings.MODERNRPC_PY2_STR_TYPE` and `settings.MODERNRPC_PY2_STR_ENCODING` variables can be defined at method level:

```
@rpc_method(str_standardization=unicode, str_standardization_encoding='UTF-8')
def print_incoming_type(data):
    """Returns a string representation of input argument type"""
    if isinstance(data, unicode):
        return 'Incoming arg is a unicode object'
```

```
    elif isinstance(data, str):
        return 'Incoming arg is a str object'

    return 'Incoming arg has type {}'.format(type(data))
```

This parameters will override the global settings for a specific RPC method.

Dates

In XML-RPC

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

In JSON-RPC

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 treatments. Use
- 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'
```

Using helper to handle all cases

To simplify date handling in your RPC methods, django-modern-rpc defines a helper to convert any object type into a `datetime.datetime` instance:

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** – 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

Here is an usage example:

```
from modernrpc.helpers import get_builtin_date

@rpc_method()
def add_one_month(date):
    """Adds 31 days to the given date, and returns the result."""
    return get_builtin_date(date) + datetime.timedelta(days=31)
```

Authentication

Starting from version 0.5, django-modern-rpc supports authentication. It is possible to restrict access to any RPC method depending on conditions names "predicate".

Basics

To provide authentication features, django-modern-rpc introduces the concept of "predicate". It is a python function taking a request as argument and returning a boolean:

```
def forbid_bots_access(request):
    forbidden_bots = [
        'Googlebot',  # Google
        'Bingbot',  # Microsoft
        'Slurp',  # Yahoo
        'DuckDuckBot',  # DuckDuckGo
        'Baiduspider',  # Baidu
        'YandexBot',  # Yandex
        'facebot',  # Facebook
    ]
    incoming_UA = request.META.get('HTTP_USER_AGENT')
    if not incoming_UA:
        return False

    for bot_ua in forbidden_bots:
        # If we detect the caller is one of the bots listed above...
        if bot_ua.lower() in incoming_UA.lower():
            # ... forbid access
            return False

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

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.

If your predicate takes additional arguments, simply add them as arguments of the decorator:

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

HTTP Basic Authentication support

django-modern-rpc comes with a builtin support for HTTP Basic Auth. 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
@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
```

Error handling

RPC Error codes and pre-defined exceptions

django-modern-rpc provide exceptions to cover common errors when requests are processed. Error handling is fully described in both XML & JSON-RPC standards. Each common error have an associated *faultCode* and the response format is described, so errors can be handled correctly on the client side.

In django-modern-rpc, all errors are reported using a set of pre-defined exceptions. Thus, in JSON and XML-RPC handlers, when an exception is caught, the correct error response is returned to the view and transmitted to the client.

This simplify error management, and allow developers to simply return errors to clients from inside a RPC Method. The error codes values are defined in:

- http://www.jsonrpc.org/specification#error_object for JSON-RPC
- http://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php for XML-RPC

Pre-defined exceptions uses the following error codes:

```
RPC_PARSE_ERROR = -32700
RPC_INVALID_REQUEST = -32600
RPC_METHOD_NOT_FOUND = -32601
RPC_INVALID_PARAMS = -32602
RPC_INTERNAL_ERROR = -32603

# Used as minimal value for any custom error returned by the server
RPC_CUSTOM_ERROR_BASE = -32099
# Used as maximal value for any custom error returned by the server
RPC_CUSTOM_ERROR_MAX = -32000
```

exception modernrpc.exceptions.**RPCException**(*code*, *message*)
This is the base class of all RPC exception. Custom exceptions raised by your RPC methods should inherits from RPCException.

exception modernrpc.exceptions.**RPCInternalError**(*message*)
Raised by handlers if any standard exception is raised during the execution of the RPC method.

exception modernrpc.exceptions.**RPCInvalidParams**(*message=''*)
Raised by handlers if the RPC method's params does not match the parameters in RPC request

exception modernrpc.exceptions.**RPCInvalidRequest**(*message=''*)
Raised by handlers if incoming JSON or XML data is not a valid JSON-RPC or XML-RPC data.

exception modernrpc.exceptions.**RPCParseError**(*message=''*)
Raised by handlers if the request can't be read as valid JSOn or XML data.

exception modernrpc.exceptions.**RPCUnknownMethod**(*name*)
Raised by handlers the RPC method called is not defined for the current entry point and protocol.

Customize error handling

If you want to define customized exceptions for your application, you can create `RPCException` sub-classes and set, for each custom exception, 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(MyException1, self).__init__(RPC_CUSTOM_ERROR_BASE + 1, message)

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

Anyway, any exception raised during the RPC method execution will generate a `RPCInternalError` with an error message constructed from the underlying error. As a result, the RPC client will have a correct message describing what went wrong.

Tips and tricks

Access to current request from RPC methods

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, as proper Django 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 REQUEST_KEY, ENTRY_POINT_KEY, PROTOCOL_KEY, HANDLER_KEY
from modernrpc.core import rpc_method

@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.META.get('Content-Type', '')
```

Enable logging

Django-modern-rpc use Python logging system to report some information, warning and errors. If you need to troubleshoot issues, you can enable logging capabilities.

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

```
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. By default, logs from `modernrpc.*` modules are discarded silently. This behavior prevent the common Python 2 error message "No handlers could be found for logger XXX".

Get involved

There is many way to contribute to project development.

Report issues, suggest enhancements

If you find a bug, want to ask question about configuration or suggest an improvemnt to the project, feel free to use the issue tracker. You will need a GitHub account.

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 the merge.

Execute the unit tests

The project uses py.test with some plugins to perform unit testing. You can install most of them using `pip`. In addition, you will have to install a supported version of Django. This is not part of `requirements.txt` since the automatic tests are performed on various Django versions. To install all dependencies for unit tests execution, you can type:

```
pip install Django
pip install -r requirements.txt
```

The file `requirements.txt` contains references to the following packages:

```
flake8
pytest-django
pytest-pythonpath
pytest-cov
requests
markdown
docutils
jsonrpcclient
```

Installing `pytest-django` will trigger `pytest` and all its dependencies. In addition, `requests` and `jsonrpcclient` are used in some tests. `flake8` is used to control code quality and respect of PEP8 standards.

When all required packages are installed, you can run the tests using:

```
pytest .
```

Execute unit tests in all supported environments

Alternatively to simple `pytest` run, you may want to check if the tests run correctly in all supported configuration. To do so, you can install and run `tox`:

```
pip install tox
tox .
```

This will execute all tests under all supported Python and Django versions. Also, it will execute `flake8` to perform code style checks.

Changelog

Release 0.8.0 (2017-07-12)

- 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
- Fixed issue #8: signature of `auth.set_authentication_predicate` has been fixed so it can be used as decorator. Thanks to @aplicacionamedida

Release 0.7.1 (2017-06-24)

- 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.

Release 0.7.0 (2017-06-24)

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

Release 0.6.0 (2017-05-13)

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

Release 0.5.2 (2017-04-18)

- User instance is now correctly stored in the 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)
- Documentation has been improved

Release 0.5.1 (2017-03-25)

- When RPC methods are registered, if a module file contains errors, a python warning is produced. This ensure 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.

Release 0.5.0 (2017-02-18)

- 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.

Release 0.4.2 (2016-11-20)

- Various performance improvements
- Better use of logging framework (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 parenthesis (and this feature is tested)
- System methods have been documented

Release 0.4.1 (2016-11-17)

- 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.

Release 0.4.0 (2016-11-17)

- API change: new unified way to register methods. Documentation in progress
- API change: XMl-RPC handler will now correctly handle None values by default. This behavior can be configured using `MODERNRPC_XMLRPC_ALLOW_NONE` setting.
- Bugfix: when django use a persistent cache (Redis, memcached, etc.), ensure the registry is up-to-date with current sources at startup

Release 0.3.2 (2016-10-26)

- Include missing templates in pypi distribution packages

Release 0.3.1 (2016-10-26)

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

Release 0.3.0 (2016-10-18)

- 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:/alorence/django-modern-rpc/blob/master/modernrpc/config.py for more details
- Many other settings added, to make the library more configurable. See http://django-modern-rpc.readthedocs.io/en/latest/basic_usage/settings.html
- 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

Release 0.2.3 (2016-10-13)

- Useless tests & testsite packages have been removed from Pypi distributions (binary & source)

Release 0.2.2 (2016-10-13)

- Useless tests packages have been removed from Pypi distributions (binary & source)

Release 0.2.1 (2016-10-12)

- 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 http://django-modern-rpc.readthedocs.io/en/latest/
- Many unit tests added to increase coverage
- `RPCEntryPoint` class can now be configured to handle only requests from a specific protocol

Release 0.2.0 (2016-10-05)

- Added very basic documentation: http://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

Release 0.1.0 (2016-10-02)

- First version with very basic features:
  - Works with Python 2.7, 3.3, 3.4 (Django 1.8 only) and 3.5
  - Works with Django 1.8, 1.9 and 1.10
  - Supports JSON-RPC and XML-RPC simple requests
  - Supports multiple entry-points with defined list of methods and supported protocols
- Some important features are still **missing**:
  - No authentication support
  - Unit tests doesn't cover all the code
  - RPC system methods utility (listMethods, methodSignature, etc.) are not implemented
  - There is no way to provide documentation in HTML form
  - The library itself doesn't have any documentation (appart from README.md)

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index