django-modern-rpc Documentation

Release 0.11.1

Antoine Lorence

May 22, 2018

Table of Contents

CHAPTER 1

What is RPC

RPC is an acronym for "Remote Procedure Call". It is a client-server protocol allowing a program (a desktop software, a webserver, etc.) to execute a function on another machine, using HTTP messages as transport for requests and responses.

CHAPTER 2

What is django-modern-rpc

This library can be used to implement a XML-RPC / JSON-RPC server as part of your Django project. It provide a simple and pythonic API to expose a set of global functions to the outside world.

CHAPTER 3

Requirements

- Python 2.7, 3.3, 3.4, 3.5 or 3.6
- Django 1.8 to 2.0
- By default, no additional dependency is required
- Optionally, you may need to install `markdown` or `docutils` to support rich-text in your methods documentation.

CHAPTER 4

Main features

Multi-protocols support

The library supports both XML-RPC and JSON-RPC 2.0 protocols. Please note that JSON-RPC 1.0 is not supported. The request's `Content-Type` is used to determine how incoming RPC call will be interpreted.

Authentication

Restrict access to your RPC methods by configuring one or more predicates. They are executed before remote procedure to determine if client is allowed to run it. In addition, a set of pre-defined decorators can be used to control access based on HTTP Basic auth.

Error management

Internally, django-modern-rpc use exceptions to track errors. This help to return a correct error response to clients as well as tracking error on server side.

Other features

Multiple entry-points You can configure your project to have as many RPC entry point as you want. This allows to provide different RPC methods depending on the URL used to expose them.

Auto-generated documentation Provide a view and a default template to display a list of all available RPC methods on the server and the corresponding documentation, based on methods docstring.

System introspection methods Common system methods such as `system.listMethods()`, `system.methodSignature()` and `system.methodHelp()` are provided to both JSON-RPC and XML-RPC clients. In adition, `system.multicall()` is provided to XML-RPC client only to allow launching multiple methods in a single RPC call. JSON-RPC client doesn't need such a method since the protocol itself define how client can use batch requests to call multiple RPC methods at once.

CHAPTER 5

Quick-start

Learn how to install and configure django-modern-rpc in one minute: read the *Quick-start guide*.

5.1 Quick-start guide

Configuring django-modern-rpc is quick and simple. Follow that steps to be up and running in few minutes!

5.1.1 Installation and configuration

Use your preferred tool (pip, pipenv, pipsi, easy_install, requirements.txt file, etc.) to install package `django-modern-rpc` in your environment:

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

Add `modernrpc` app to your Django applications, in `settings.INSTALLED_APPS`:

```
# in project's settings.py
INSTALLED_APPS = [
    ...
    'modernrpc',
]
```

5.1.2 Declare a RPC Entry Point in URLConf

The entry point is a standard Django view class which mainly handle RPC calls. Like other Django views, you have to use `django.conf.urls.url()` to map URL pattern with this class. This can be done in your project's URLConf, or in any app specific one.

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

from modernrpc.views import RPCEntryPoint

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

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

5.1.3 Write and register your remote procedures

Now, you have to write your remote procedures. These are global functions decorated with `@rpc_method`.

```
# 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 to your needs. Read *Configure the registration* for full list of options.

5.1.4 Declare your RPC methods modules

Django-modern-rpc will automatically register functions decorated with `@rpc_method`, but needs a hint to locate them. Declare `settings.MODERNRPC_METHODS_MODULES` to indicate all python modules where remote procedures are defined.

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

5.1.5 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.

5.2 Standard configuration

5.2.1 Write and register remote procedures

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

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', '')
```

5.2.2 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

5.2.3 HTML Documentation generation

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.

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

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

Customize rendering

By default, documentation will be rendered using a Bootstrap 4 based template with collapse component, to display doc in a list of accordion widgets.

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 = [

    # Configure the RPCEntryPoint directly by passing some arguments to as_view()␣
↪method
    url(r'^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.

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 heaser if present.
    This method demonstrate 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'
```

5.2.4 Authentication

New in version 0.5.

django-modern-rpc supports authentication. It is possible to restrict access to any RPC method depending on conditions named "predicate".

Basics

To provide authentication features, django-modern-rpc introduce 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
```

(continues on next page)

(continued from previous page)

```
        '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.

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
```

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
```

5.2.5 Error handling and logging system

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.**AuthenticationFailed**(*method_name*)
Raised when authentication system forbade execution of a RPC Method

exception modernrpc.exceptions.**RPCException**(*code*, *message*, *data=None*)
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*, *data=None*)
Raised by handlers if any standard exception is raised during the execution of the RPC method.

exception modernrpc.exceptions.**RPCInvalidParams**(*message*, *data=None*)
Raised by handlers if the RPC method's params does not match the parameters in RPC request

exception modernrpc.exceptions.**RPCInvalidRequest**(*message*, *data=None*)
Raised by handlers if incoming JSON or XML data is not a valid JSON-RPC or XML-RPC data.

exception modernrpc.exceptions.**RPCParseError**(*message*, *data=None*)
Raised by handlers if the request can't be read as valid JSOn or XML data.

exception modernrpc.exceptions.**RPCUnknownMethod**(*name*, *data=None*)
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.

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.

New in version 0.7: 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".

5.2.6 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: `(empty)`, `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 you set `settings.MODERNRPC_DOC_FORMAT` to a non-empty value

---

5.3 Advanced topics

5.3.1 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):
```

(continues on next page)

(continued from previous page)

```
        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)
```

5.3.2 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.

5.4 Bibliography

The development of django-modern-rpc is a best effort to follow existing standards.

XML-RPC standard:

- http://xmlrpc.scripting.com/spec.html

JSON-RPC specification

- http://www.jsonrpc.org/specification
- JSON type support: http://www.ietf.org/rfc/rfc4627.txt

Specification for system introspection methods (XML-RPC)

- http://scripts.incutio.com/xmlrpc/introspection.html (originally available from http://xmlrpc.usefulinc.com/doc/reserved.html)

Description of the system.multicall specific method, by Eric Kidd:

- https://mirrors.talideon.com/articles/multicall.html (originally posted on http://www.xmlrpc.com/discuss/msgReader\protect\T1\textdollar1208 and http://directory.xmlrpc.com/services/xmlrpcextensions/)
- Python XML-RPC implementation: https://docs.python.org/3/library/xmlrpc.client.html

5.5 Get involved

There is many way to contribute to project development.

5.5.1 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.

5.5.2 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.

5.5.3 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==3.5.1
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 .
```

5.5.4 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 configurations. 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. In addition, it will execute `flake8` to perform code style checks.

5.6 Changelog

5.6.1 Release 0.11.1 (2018-05135)

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.

5.6.2 Release 0.11.0 (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 environements.
- 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.

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 a 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 anymore marked as "safe" 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.

Other updates

- Many units tests have been improved. Some tests with many calls to LiveServer have been splitted into shorter ones.

5.6.3 Release 0.10.0 (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)

5.6.4 Release 0.9.0 (2017-10-03)

This is a major release with many improvements, protocol support and bug fixes. This version introduce 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 payload is no longer considered as invalid, but is correectly handled. No HTTP response is returned in such case, according to the standard.
- XML-RPC: exception raised when serializing data to XML are now catched as `InternalError` and a clear error message

API Changes

- `modernrpc.handlers.JSONRPC` and `modernrpc.handlers.XMLRPC` constants have been moved and renamed. They become respectively `modernrpc.core.JSONRPC_PROTOCOL` and `modernrpc.core.XMLRPC_PROTOCOL`
- `RPCHandler` class updated, as well as subclases `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).

Minor updates

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

5.6.5 Release 0.8.1 (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()`

5.6.6 Release 0.8.0 (2017-07-12)

Bugfixes

- 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 so it can be used as decorator (#8). Thanks to @aplicacionamedida

5.6.7 Release 0.7.1 (2017-06-24)

Minor fix

- 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.

5.6.8 Release 0.7.0 (2017-06-24)

Improvement

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

5.6.9 Release 0.6.0 (2017-05-13)

Performance 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.

5.6.10 Release 0.5.2 (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

5.6.11 Release 0.5.1 (2017-03-25)

Improvements

- 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.

5.6.12 Release 0.5.0 (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.

5.6.13 Release 0.4.2 (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 parenthesis (and this feature is tested)
- System methods have been documented

5.6.14 Release 0.4.1 (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.

5.6.15 Release 0.4.0 (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.

Bugfix

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

5.6.16 Release 0.3.2 (2016-10-26)

Bugfix

- Include missing templates in pypi distribution packages

5.6.17 Release 0.3.1 (2016-10-26)

Improvements

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

5.6.18 Release 0.3.0 (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 http://django-modern-rpc.readthedocs.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

5.6.19 Release 0.2.3 (2016-10-13)

Minor change

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

5.6.20 Release 0.2.2 (2016-10-13)

Minor change

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

5.6.21 Release 0.2.1 (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 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

5.6.22 Release 0.2.0 (2016-10-05)

Improvements

- 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

5.6.23 Release 0.1.0 (2016-10-02)

This is the very first version of the library. Only a few 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

Missing features

- 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 (appart from README.md)

5.7 Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index