OdooRPC Documentation

Release 0.6.2

Sébastien Alix

Sep 18, 2018

Contents

CHAPTER 1

Introduction

OdooRPC is a Python package providing an easy way to pilot your Odoo servers through *RPC*.

Features supported:

- access to all data model methods (even `browse`) with an API similar to the server-side API,
- use named parameters with model methods,
- user context automatically sent providing support for internationalization,
- browse records,
- execute workflows,
- manage databases,
- reports downloading,
- JSON-RPC protocol (SSL supported),

CHAPTER 2

Quick start

How does it work? See below:

```
import odoorpc

# Prepare the connection to the server
odoo = odoorpc.ODOO('localhost', port=8069)

# Check available databases
print(odoo.db.list())

# Login
odoo.login('db_name', 'user', 'passwd')

# Current user
user = odoo.env.user
print(user.name)            # name of the user connected
print(user.company_id.name) # the name of its company

# Simple 'raw' query
user_data = odoo.execute('res.users', 'read', [user.id])
print(user_data)

# Use all methods of a model
if 'sale.order' in odoo.env:
    Order = odoo.env['sale.order']
    order_ids = Order.search([])
    for order in Order.browse(order_ids):
        print(order.name)
        products = [line.product_id.name for line in order.order_line]
        print(products)

# Update data through a record
user.name = "Brian Jones"
```

For more details and features, see the *tutorials*, the *Frequently Asked Questions (FAQ)* and the *API reference* sections.

CHAPTER 3

Contents

3.1 Download and install instructions

3.1.1 Python Package Index (PyPI)

You can install *OdooRPC* with *pip*:

```
$ pip install odoorpc
```

No dependency is required.

3.1.2 Source code

The project is hosted on GitHub. To get the last stable release (`master` branch), just type:

```
$ git clone https://github.com/OCA/odoorpc.git
```

Also, the project uses the Git Flow extension to manage its branches and releases. If you want to contribute, make sure to make your Pull Request against the *develop* branch.

3.1.3 Run tests

Unit tests depend on the standard module *unittest* (Python 2.7 and 3.x) and on a running Odoo instance. To run all unit tests from the project directory, run the following command:

```
$ python -m unittest discover -v
```

To run a specific test:

```
$ python -m unittest -v odoorpc.tests.test_init
```

To configure the connection to the server, some environment variables are available:

```
$ export ORPC_TEST_PROTOCOL=jsonrpc
$ export ORPC_TEST_HOST=localhost
$ export ORPC_TEST_PORT=8069
$ export ORPC_TEST_DB=odoorpc_test
$ export ORPC_TEST_USER=admin
$ export ORPC_TEST_PWD=admin
$ export ORPC_TEST_VERSION=10.0
$ export ORPC_TEST_SUPER_PWD=admin
$ python -m unittest discover -v
```

The database `odoorpc_test` will be created if it does not exist.

3.2 Tutorials

Note: The tutorial is based on *Odoo 10.0*, the examples must be adapted following the version of *Odoo* you are using.

3.2.1 Create a new database

To dialog with your *Odoo* server, you need an instance of the *odoorpc.ODOO* class. Let's instanciate it:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost', 'jsonrpc', 8069)
```

Two protocols are available: `jsonrpc` and `jsonrpc+ssl`. Then, create your database for the purposes of this tutorial (you need to know the *super* admin password to do this):

```
>>> odoo.db.create('super_password', 'tutorial', demo=True, lang='fr_FR', admin_
↪password='password')
```

The creation process may take some time on the server. If you get a timeout error, set a higher timeout before repeating the process:

```
>>> odoo.config['timeout'] = 300    # Set the timeout to 300 seconds
>>> odoo.db.create('super_password', 'tutorial', demo=True, lang='fr_FR', admin_
↪password='password')
```

To check available databases, use the *odoo.db* property with the *list* method:

```
>>> odoo.db.list()
['tutorial']
```

You are now ready to login to your database!

Documentation about databases management is available *here*.

Next step: Login to your new database

3.2.2 Login to your new database

Use the *login* method on a database with the account of your choice:

```
>>> odoo.login('tutorial', 'admin', 'password')
```

Note: Under the hood the login method creates a cookie, and all requests thereafter which need a user authentication are cookie-based.

Once logged in, you can check some information through the *environment*:

```
>>> odoo.env.db
'tutorial'
>>> odoo.env.context
{'lang': 'fr_FR', 'tz': 'Europe/Brussels', 'uid': 1}
>>> odoo.env.uid
1
>>> odoo.env.lang
'fr_FR'
>>> odoo.env.user.name              # name of the user
'Administrator'
>>> odoo.env.user.company_id.name   # the name of its company
'YourCompany'
```

From now, you can easily execute any kind of queries on your *Odoo* server (execute model methods, trigger workflow, download reports...).

Next step: Execute RPC queries

3.2.3 Execute RPC queries

The basic methods to execute RPC queries related to data models are *execute* and *execute_kw*. They take at least two parameters (the model and the name of the method to call) following by additional variable parameters according to the method called:

```
>>> order_data = odoo.execute('sale.order', 'read', [1], ['name'])
```

This instruction will call the `read` method of the `sale.order` model for the order ID=1, and will only returns the value of the field `name`.

However there is a more efficient way to perform methods of a model by getting a proxy of it with the *model registry*, which provides an API almost syntactically identical to the *Odoo* server side API (see *odoorpc.models.Model*), and which is able to send the user context automatically:

```
>>> User = odoo.env['res.users']
>>> User.write([1], {'name': "Dupont D."})
True
>>> odoo.env.context
{'lang': 'fr_FR', 'tz': False}
>>> Product = odoo.env['product.product']
>>> Product.name_get([3, 4])
[[3, '[SERV_COST] Audit externe''], [4, '[PROD_DEL] Commutateur, 24 ports']]
```

To stop sending the user context, use the *odoorpc.ODOO.config* property:

```
>>> odoo.config['auto_context'] = False
>>> Product.name_get([3, 4])     # Without context, lang 'en_US' by default
[[3, '[SERV_COST] External Audit'], [4, '[PROD_DEL] Switch, 24 ports']]
```

Note: The `auto_context` option only affect methods called from model proxies.

Here is another example of how to install a module (you have to be logged as an administrator to perform this task):

```
>>> Module = odoo.env['ir.module.module']
>>> module_id = Module.search([('name', '=', 'purchase')])
>>> Module.button_immediate_install(module_id)
```

Next step: Browse records

3.2.4 Browse records

A great functionality of *OdooRPC* is its ability to generate objects that are similar to records used on the server side.

Get records

To get one or more records (a recordset), you will use the *browse* method from a model proxy:

```
>>> Partner = odoo.env['res.partner']
>>> partner = Partner.browse(1)      # fetch one record, partner ID = 1
>>> partner
Recordset('res.partner', [1])
>>> partner.name
'YourCompany'
>>> for partner in Partner.browse([1, 3]):  # fetch several records
>>>     print(partner.name)
...
YourCompany
Administrator
```

From such objects, it is possible to easily explore relationships. The related records are generated on the fly:

```
>>> partner = Partner.browse(1)
>>> for child in partner.child_ids:
...     print("%s (%s)" % (child.name, child.parent_id.name))
...
Mark Davis (YourCompany)
Roger Scott (YourCompany)
```

Outside relation fields, *Python* data types are used, like `datetime.date` and `datetime.datetime`:

```
>>> Purchase = odoo.env['purchase.order']
>>> order = Purchase.browse(1)
>>> order.date_order
datetime.datetime(2016, 11, 6, 11, 23, 10)
```

A list of data types used by records fields are available *here*.

Get records corresponding to an External ID

To get a record through its external ID, use the *ref* method from the environment:

```
>>> lang_en = odoo.env.ref('base.lang_en')
>>> lang_en
Recordset('res.lang', [1])
>>> lang_en.code
'en_US'
```

Next step: Call methods from a Model or from records

3.2.5 Call methods from a Model or from records

Unlike the *Odoo* API, there is a difference between class methods (e.g.: *create*, *search*, ...) and instance methods that apply directly on existing records (*write*, *read*, ...):

```
>>> User = odoo.env['res.users']
>>> User.write([1], {'name': "Dupont D."})  # Using the class method
True
>>> user = User.browse(1)
>>> user.write({'name': "Dupont D."})       # Using the instance method
```

When a method is called directly on records, their *ids* (here *user.ids*) is simply passed as the first parameter. This also means that you are not able to call class methods such as *create* or *search* from a set of records:

```
>>> User = odoo.env['res.users']
>>> User.create({...})              # Works
>>> user = User.browse(1)
>>> user.ids
[1]
>>> user.create({...})              # Error, `create()` does not accept `ids` in␣
↪first parameter
>>> user.__class__.create({...})    # Works
```

This is a behaviour *by design*: *OdooRPC* has no way to make the difference between a *class* or an *instance* method through RPC, this is why it differs from the *Odoo* API.

Next step: Update data through records

3.2.6 Update data through records

By default when updating values of a record, the change is automatically sent to the server. Let's update the name of a partner:

```
>>> Partner = odoo.env['res.partner']
>>> partner_id = Partner.create({'name': "Contact Test"})
>>> partner = Partner.browse(partner_id)
>>> partner.name = "MyContact"
```

This is equivalent to:

```
>>> Partner.write([partner.id], {'name': "MyContact"})
```

As one update is equivalent to one RPC query, if you need to update several fields for one record it is encouraged to use the *write* method as above

```
>>> partner.write({'name': "MyContact", 'website': 'http://example.net'})    # one␣
↪RPC query
```

Or, deactivate the `auto_commit` option and commit the changes manually:

```
>>> odoo.config['auto_commit'] = False
>>> partner.name = "MyContact"
>>> partner.website = 'http://example.net'
>>> partner.env.commit()     # one RPC by record modified
```

Char, Float, Integer, Boolean, Text and Binary

As see above, it's as simple as that:

```
>>> partner.name = "New Name"
```

Selection

Same as above, except there is a check about the value assigned. For instance, the field `type` of the `res.partner` model accept values contains in `['default', 'invoice', 'delivery', 'contact', 'other']`:

```
>>> partner.type = 'delivery'    # Ok
>>> partner.type = 'foobar'      # Error!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "odoorpc/service/model/fields.py", line 148, in __set__
    value = self.check_value(value)
  File "odoorpc/service/model/fields.py", line 160, in check_value
    field_name=self.name,
ValueError: The value 'foobar' supplied doesn't match with the possible values '[
↪'contact', 'invoice', 'delivery', 'other']' for the 'type' field
```

Many2one

You can also update a `many2one` field, with either an ID or a record:

```
>>> partner.parent_id = 1                    # with an ID
>>> partner.parent_id = Partner.browse(1)    # with a record object
```

You can't put any ID or record, a check is made on the relationship to ensure data integrity:

```
>>> User = odoo.env['res.users']
>>> user = User.browse(1)
>>> partner = Partner.browse(2)
>>> partner.parent_id = user
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "odoorpc/service/model/fields.py", line 263, in __set__
    o_rel = self.check_value(o_rel)
  File "odoorpc/service/model/fields.py", line 275, in check_value
    field_name=self.name))
ValueError: Instance of 'res.users' supplied doesn't match with the relation 'res.
↪partner' of the 'parent_id' field.
```

One2many and Many2many

`one2many` and `many2many` fields can be updated by providing a list of tuple as specified in the *Odoo* documentation (link), a list of records, a list of record IDs, an empty list or `False`:

With a tuple (as documented), no magic here:

```
>>> user = odoo.env['res.users'].browse(1)
>>> user.groups_id = [(6, 0, [8, 5, 6, 4])]
```

With a recordset:

```
>>> groups = odoo.env['res.groups'].browse([8, 5, 6, 4])
>>> user.groups_id = groups
```

With a list of record IDs:

```
>>> user.groups_id = [8, 5, 6, 4]
```

The last two examples are equivalent to the first (they generate a `(6, 0, IDS)` tuple).

However, if you set an empty list or `False`, the relation between records will be removed:

```
>>> user.groups_id = []
>>> user.groups_id
Recordset('res.group', [])
>>> user.groups_id = False
>>> user.groups_id
Recordset('res.group', [])
```

Another facility provided by *OdooRPC* is adding and removing objects using *Python* operators += and −=. As usual, you can add an ID, a record, or a list of them:

With a list of records:

```
>>> groups = odoo.env['res.groups'].browse([4, 5])
Recordset('res.group', [1, 2, 3])
>>> user.groups_id += groups
>>> user.groups_id
Recordset('res.group', [1, 2, 3, 4, 5])
```

With a list of record IDs:

```
>>> user.groups_id += [4, 5]
>>> user.groups_id
Recordset('res.group', [1, 2, 3, 4, 5])
```

With an ID only:

```
>>> user.groups_id -= 4
>>> user.groups_id
Recordset('res.group', [1, 2, 3, 5])
```

With a record only:

```
>>> group = odoo.env['res.groups'].browse(5)
>>> user.groups_id -= group
>>> user.groups_id
Recordset('res.group', [1, 2, 3])
```

Reference

To update a `reference` field, you have to use either a string or a record object as below:

```
>>> IrActionServer = odoo.env['ir.actions.server']
>>> action_server = IrActionServer.browse(8)
>>> action_server.ref_object = 'res.partner,1'      # with a string with the format '
↪{relation},{id}'
>>> action_server.ref_object = Partner.browse(1)    # with a record object
```

A check is made on the relation name:

```
>>> action_server.ref_object = 'foo.bar,42'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "odoorpc/service/model/fields.py", line 370, in __set__
    value = self.check_value(value)
  File "odoorpc/service/model/fields.py", line 400, in check_value
    self._check_relation(relation)
  File "odoorpc/service/model/fields.py", line 381, in _check_relation
    field_name=self.name,
ValueError: The value 'foo.bar' supplied doesn't match with the possible values '[...]
↪' for the 'ref_object' field
```

Date and Datetime

`date` and `datetime` fields accept either string values or `datetime.date`/`datetime.datetime` objects.

With `datetime.date` and `datetime.datetime` objects:

```
>>> import datetime
>>> Purchase = odoo.env['purchase.order']
>>> order = Purchase.browse(1)
>>> order.date_order = datetime.datetime(2016, 11, 7, 11, 23, 10)
```

With formated strings:

```
>>> order.date_order = "2016-11-07"             # %Y-%m-%d
>>> order.date_order = "2016-11-07 12:31:24"    # %Y-%m-%d %H:%M:%S
```

As always, a wrong type will raise an exception:

```
>>> order.date_order = "foobar"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "odoorpc/fields.py", line 187, in setter
    value = self.check_value(value)
  File "odoorpc/fields.py", line 203, in check_value
    self.pattern))
ValueError: Value not well formatted, expecting '%Y-%m-%d %H:%M:%S' format
```

Next step: Download reports

3.2.7 Download reports

Another nice feature is the reports generation with the *report* property. The *list* method allows you to list all reports available on your *Odoo* server (classified by models), while the *download* method will retrieve a report as a file (in PDF, HTML... depending of the report).

To list available reports:

```
>>> odoo.report.list()
{u'account.invoice': [{u'name': u'Duplicates', u'report_type': u'qweb-pdf', u'report_
↪name': u'account.account_invoice_report_duplicate_main'}, {u'name': u'Invoices', u
↪'report_type': u'qweb-pdf', u'report_name': u'account.report_invoice'}], u'res.
↪partner': [{u'name': u'Aged Partner Balance', u'report_type': u'qweb-pdf', u'report_
↪name': u'account.report_agedpartnerbalance'}, {u'name': u'Due Payments', u'report_
↪type': u'qweb-pdf', u'report_name': u'account.report_overdue'}], ...}
```

To download a report:

```
>>> report = odoo.report.download('account.report_invoice', [1])
```

The method will return a file-like object, you will have to read its content in order to save it on your file-system:

```
>>> with open('invoice.pdf', 'w') as report_file:
...     report_file.write(report.read())
...
```

Next step: Save your credentials (session)

3.2.8 Save your credentials (session)

Once you are authenticated with your *ODOO* instance, you can *save* your credentials under a code name and use this one to quickly instantiate a new *ODOO* class:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost')
>>> user = odoo.login('tutorial', 'admin', 'admin')
>>> odoo.save('tutorial')
```

By default, these informations are stored in the `~/.odoorpcrc` file. You can however use another file:

```
>>> odoo.save('tutorial', '~/my_own_odoorpcrc')
```

Then, use the *odoorpc.ODOO.load()* class method:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO.load('tutorial')
```

Or, if you have saved your configuration in another file:

```
>>> odoo = odoorpc.ODOO.load('tutorial', '~/my_own_odoorpcrc')
```

You can check available sessions with *odoorpc.ODOO.list()*, and remove them with *odoorpc.ODOO.remove()*:

```
>>> odoorpc.ODOO.list()
['tutorial']
>>> odoorpc.ODOO.remove('tutorial')
>>> 'tutorial' not in odoorpc.ODOO.list()
True
```

3.3 Frequently Asked Questions (FAQ)

3.3.1 Why OdooRPC? And why migrate from OERPLib to OdooRPC?

It was a tough decision, but several reasons motivated the *OdooRPC* project:

RPC Protocol The first point is about the supported protocol, *XML-RPC* is kept in *Odoo* for compatibility reasons (and will not evolve anymore, maybe removed one day), replaced by the *JSON-RPC* one. Although these protocols are almost similar in the way we build RPC requests, some points make *JSON-RPC* a better and reliable choice like the way to handle errors raised by the *Odoo* server (access to the type of exception raised, the complete server traceback...). To keep a clean and maintainable base code, it would have been difficult to support both protocols in *OERPLib*, and it is why *OdooRPC* only support *JSON-RPC*.

Another good point with *JSON-RPC* is the ability to request all server web controllers to reproduce requests (*type='json'* ones) made by the official *Javascript* web client. As the code to make such requests is based on standard *HTTP* related Python modules, *OdooRPC* is also able to request *HTTP* web controllers (*type='http'* ones).

In fact, you could see *OdooRPC* as a high level API for *Odoo* with which you could replicate the behaviour of the official *Javascript* web client, but in *Python*.

New server API One goal of *OERPLib* was to give an API not too different from the server side API to reduce the learning gap between server-side development and client-side with an *RPC* library. With the new API which appears in *Odoo* 8.0 this is another brake (the old API has even been removed since Odoo 10.0), so the current API of *OERPLib* is not anymore consistent. As such, *OdooRPC* mimics A LOT the new API of Odoo, for more consistency (see the *tutorials*).

New brand Odoo *OpenERP* became *Odoo*, so what does *OERPLib* mean? *OEWhat*? This is obvious for old developers which start the *OpenERP* adventure since the early days, but the *OpenERP* brand is led to disappear, and it can be confusing for newcomers in the *Odoo* world. So, *OdooRPC* speaks for itself.

Maintenance cost, code cleanup *OpenERP* has evolved a lot since the version 5.0 (2009), making *OERPLib* hard to maintain (write tests for all versions before each *OERPLib* and *OpenERP* release is very time consuming). All the compatibility code for *OpenERP* 5.0 to 7.0 was dropped for *OdooRPC*, making the project more maintainable. *Odoo* is now a more mature product, and *OdooRPC* should suffer less about compatibility issues from one release to another.

As *OdooRPC* has not the same constraints concerning *Python* environments where it could be running on, it is able to work on *Python* 2.7 to 3.X.

OdooRPC is turned towards the future, so you are encouraged to use or migrate on it for projects based on *Odoo* >= 8.0. It is more reliable, better covered by unit tests, and almost identical to the server side new API.

3.3.2 Connect to an Odoo Online (SaaS) instance

First, you have to connect on your *Odoo* instance, and set a password for your user account in order to active the *RPC* interface.

Then, just use the `jsonrpc+ssl` protocol with the port 443:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('foobar.my.odoo.com', protocol='jsonrpc+ssl', port=443)
>>> odoo.version
'8.saas~5'
```

3.3.3 Update a record with an *on_change* method

OdooRPC does not provide helpers for such methods currently. A call to an `on_change` method intend to be executed from a view and there is no support for that (not yet?) such as fill a form, validate it, etc...

But you can emulate an `on_change` by writing your own function, for instance:

```
def on_change(record, method, args=None, kwargs=None):
    """Update `record` with the result of the on_change `method`"""
    res = record._odoo.execute_kw(record._name, method, args, kwargs)
    for k, v in res['value'].iteritems():
        setattr(record, k, v)
```

And call it on a record with the desired method and its parameters:

```
>>> order = odoo.get('sale.order').browse(42)
>>> on_change(order, 'product_id_change', args=[ARGS], kwargs={KWARGS})
```

3.3.4 Some model methods does not accept the *context* parameter

The `context` parameter is sent automatically for each call to a *Model* method. But on the side of the *Odoo* server, some methods have no `context` parameter, and *OdooRPC* has no way to guess it, which results in an nasty exception. So you have to disable temporarily this behaviour by yourself by setting the `auto_context` option to `False`:

```
>>> odoo.config['auto_context'] = False  # 'get()' method of 'ir.sequence' does not␣
↪support the context parameter
>>> next_seq = odoo.get('ir.sequence').get('stock.lot.serial')
>>> odoo.config['auto_context'] = True  # Restore the configuration
```

3.3.5 Change the behaviour of a script according to the version of Odoo

You can compare versions of *Odoo* servers with the *v* function applied on the *ODOO.version* property, for instance:

```
import odoorpc
from odoorpc.tools import v

for session in odoorpc.ODOO.list():
    odoo = odoorpc.ODOO.load(session)
    if v(odoo.version) < v('10.0'):
        pass  # do some stuff
    else:
        pass  # do something else
```

3.4 Reference

3.4.1 Browse object fields

The table below presents the Python types returned by *OdooRPC* for each *Odoo* fields used by `Recordset` objects (see the `browse` method):

Odoo fields	Python types used in *OdooRPC*
fields.Binary	unicode or str
fields.Boolean	bool
fields.Char	unicode or str
fields.Date	datetime.date
fields.Datetime	datetime.datetime
fields.Float	float
fields.Integer	integer
fields.Selection	unicode or str
fields.Text	unicode or str
fields.Html	unicode or str
fields.Many2one	`Recordset`
fields.One2many	`Recordset`
fields.Many2many	`Recordset`
fields.Reference	`Recordset`

3.4.2 odoorpc

The *odoorpc* module defines the `ODOO` class.

The `ODOO` class is the entry point to manage *Odoo* servers. You can use this one to write *Python* programs that performs a variety of automated jobs that communicate with a *Odoo* server.

Here's a sample session using this module:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost', port=8069)  # connect to localhost, default port
>>> odoo.login('my_database', 'admin', 'admin')
```

3.4.3 odoorpc.ODOO

class odoorpc.**ODOO**(*host='localhost'*, *protocol='jsonrpc'*, *port=8069*, *timeout=120*, *version=None*, *opener=None*)

Return a new instance of the `ODOO` class. *JSON-RPC* protocol is used to make requests, and the respective values for the *protocol* parameter are `jsonrpc` (default) and `jsonrpc+ssl`.

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost', protocol='jsonrpc', port=8069)
```

OdooRPC will try by default to detect the server version in order to adapt its requests if necessary. However, it is possible to force the version to use with the *version* parameter:

```
>>> odoo = odoorpc.ODOO('localhost', version='10.0')
```

You can also define a custom URL opener to handle HTTP requests. A use case is to manage a basic HTTP authentication in front of *Odoo*:

```
>>> import urllib.request
>>> import odoorpc
>>> pwd_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm()
>>> pwd_mgr.add_password(None, "http://example.net", "userName", "passWord")
>>> auth_handler = urllib.request.HTTPBasicAuthHandler(pwd_mgr)
>>> opener = urllib.request.build_opener(auth_handler)
>>> odoo = odoorpc.ODOO('example.net', port=80, opener=opener)
```

Python 2:

Raise *odoorpc.error.InternalError*

Raise *ValueError* (wrong protocol, port value, timeout value)

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.InternalError*

Raise *ValueError* (wrong protocol, port value, timeout value)

Raise *urllib.error.URLError* (connection error)

config

Dictionary of available configuration options.

```
>>> odoo.config
{'auto_commit': True, 'auto_context': True, 'timeout': 120}
```

- `auto_commit`: if set to *True* (default), each time a value is set on a record field a RPC request is sent to the server to update the record (see `odoorpc.env.Environment.commit()`).
- `auto_context`: if set to *True* (default), the user context will be sent automatically to every call of a *model* method (default: *True*):

```
>>> odoo.env.context['lang'] = 'fr_FR'
>>> Product = odoo.env['product.product']
>>> Product.name_get([2])    # Context sent by default ('lang': 'fr_FR' here)
[[2, 'Surveillance sur site']]
>>> odoo.config['auto_context'] = False
>>> Product.name_get([2])    # No context sent, 'en_US' used
[[2, 'On Site Monitoring']]
```

- `timeout`: set the maximum timeout in seconds for a RPC request (default: *120*):

```
>>> odoo.config['timeout'] = 300
```

db

The database management service. See the *odoorpc.db.DB* class.

env

The environment which wraps data to manage records such as the user context and the registry to access data model proxies.

```
>>> Partner = odoo.env['res.partner']
>>> Partner
Model('res.partner')
```

See the *odoorpc.env.Environment* class.

exec_workflow(*model*, *record_id*, *signal*)

Execute the workflow *signal* on the instance having the ID *record_id* of *model*.

Python 2:

Raise *odoorpc.error.RPCError*

Raise *odoorpc.error.InternalError* (if not logged)

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError*

Raise *odoorpc.error.InternalError* (if not logged)

Raise *urllib.error.URLError* (connection error)

execute(*model*, *method*, **args*)

Execute the *method* of *model*. **args* parameters varies according to the *method* used.

```
>>> odoo.execute('res.partner', 'read', [1], ['name'])
[{'id': 1, 'name': 'YourCompany'}]
```

Python 2:

Returns the result returned by the *method* called

Raise *odoorpc.error.RPCError*

Raise *odoorpc.error.InternalError* (if not logged)

Raise *urllib2.URLError* (connection error)

Python 3:

Returns the result returned by the *method* called

Raise *odoorpc.error.RPCError*

Raise *odoorpc.error.InternalError* (if not logged)

Raise *urllib.error.URLError* (connection error)

execute_kw(*model*, *method*, *args=None*, *kwargs=None*)

Execute the *method* of *model*. *args* is a list of parameters (in the right order), and *kwargs* a dictionary (named parameters). Both varies according to the *method* used.

```
>>> odoo.execute_kw('res.partner', 'read', [[1]], {'fields': ['name']})
[{'id': 1, 'name': 'YourCompany'}]
```

Python 2:

Returns the result returned by the *method* called

Raise *odoorpc.error.RPCError*

Raise *odoorpc.error.InternalError* (if not logged)

Raise *urllib2.URLError* (connection error)

Python 3:

Returns the result returned by the *method* called

Raise *odoorpc.error.RPCError*

Raise *odoorpc.error.InternalError* (if not logged)

Raise *urllib.error.URLError* (connection error)

host
Hostname of IP address of the the server.

http(*url*, *data=None*, *headers=None*)
Low level method to execute raw HTTP queries.

Note: For low level JSON-RPC queries, see the more convenient *odoorpc.ODOO.json()* method instead.

You have to know the names of each POST parameter required by the URL, and set them in the *data* string/buffer. The *data* argument must be built by yourself, following the expected URL parameters (with `urllib.urlencode()` function for simple parameters, or multipart/form-data structure to handle file upload).

E.g., the HTTP raw query to get the company logo on *Odoo 10.0*:

```
>>> response = odoo.http('web/binary/company_logo')
>>> binary_data = response.read()
```

Python 2:

Returns *urllib.addinfourl*

Raise *urllib2.HTTPError*

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *http.client.HTTPResponse*

Raise *urllib.error.HTTPError*

Raise *urllib.error.URLError* (connection error)

json(*url*, *params*)
Low level method to execute JSON queries. It basically performs a request and raises an *odoorpc.error.RPCError* exception if the response contains an error.

You have to know the names of each parameter required by the function called, and set them in the *params* dictionary.

Here an authentication request:

```
>>> data = odoo.json(
...     '/web/session/authenticate',
...     {'db': 'db_name', 'login': 'admin', 'password': 'admin'})
>>> from pprint import pprint
>>> pprint(data)
{'id': 645674382,
```

(continues on next page)

(continued from previous page)

```
 'jsonrpc': '2.0',
 'result': {'db': 'db_name',
            'session_id': 'fa740abcb91784b8f4750c5c5b14da3fcc782d11',
            'uid': 1,
            'user_context': {'lang': 'en_US',
                             'tz': 'Europe/Brussels',
                             'uid': 1},
            'username': 'admin'}}
```

And a call to the `read` method of the `res.users` model:

```
>>> data = odoo.json(
...     '/web/dataset/call',
...     {'model': 'res.users', 'method': 'read',
...      'args': [[1], ['name']]})
>>> from pprint import pprint
>>> pprint(data)
{'id': ...,
 'jsonrpc': '2.0',
 'result': [{'id': 1, 'name': 'Administrator'}]}
```

Python 2:

Returns a dictionary (JSON response)

Raise *odoorpc.error.RPCError*

Raise *urllib2.HTTPError* (if *params* is not a dictionary)

Raise *urllib2.URLError* (connection error)

Python 3:

Returns a dictionary (JSON response)

Raise *odoorpc.error.RPCError*

Raise *urllib.error.HTTPError* (if *params* is not a dictionary)

Raise *urllib.error.URLError* (connection error)

classmethod list(*rc_file='~/.odoorpcrc'*)

Return a list of all stored sessions available in the *rc_file* file:

```
>>> import odoorpc
>>> odoorpc.ODOO.list()
['foo', 'bar']
```

Use the *save* and *load* methods to manage such sessions.

Python 2:

Raise *IOError*

Python 3:

Raise *PermissionError*

Raise *FileNotFoundError*

classmethod load(*name*, *rc_file='~/.odoorpcrc'*)

Return a connected *ODOO* session identified by *name*:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO.load('foo')
```

Such sessions are stored with the *save* method.

Python 2:

Raise *odoorpc.error.RPCError*

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError*

Raise *urllib.error.URLError* (connection error)

login(*db*, *login='admin'*, *password='admin'*)

Log in as the given *user* with the password *passwd* on the database *db*.

```
>>> odoo.login('db_name', 'admin', 'admin')
>>> odoo.env.user.name
'Administrator'
```

Python 2:

Raise *odoorpc.error.RPCError*

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError*

Raise *urllib.error.URLError* (connection error)

logout()

Log out the user.

```
>>> odoo.logout()
True
```

Python 2:

Returns *True* if the operation succeed, *False* if no user was logged

Raise *odoorpc.error.RPCError*

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *True* if the operation succeed, *False* if no user was logged

Raise *odoorpc.error.RPCError*

Raise *urllib.error.URLError* (connection error)

port

The port used.

protocol

The protocol used.

classmethod remove(*name*, *rc_file='~/.odoorpcrc'*)

Remove the session identified by *name* from the *rc_file* file:

```
>>> import odoorpc
>>> odoorpc.ODOO.remove('foo')
True
```

Python 2:

Raise *ValueError* (if the session does not exist)

Raise *IOError*

Python 3:

Raise *ValueError* (if the session does not exist)

Raise *PermissionError*

Raise *FileNotFoundError*

report
The report management service. See the *odoorpc.report.Report* class.

save(*name*, *rc_file='~/.odoorpcrc'*)
Save the current *ODOO* instance (a *session*) inside *rc_file* (`~/.odoorpcrc` by default). This session will be identified by *name*:

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost', port=8069)
>>> odoo.login('db_name', 'admin', 'admin')
>>> odoo.save('foo')
```

Use the *list* class method to list all stored sessions, and the *load* class method to retrieve an already-connected *ODOO* instance.

Python 2:

Raise *odoorpc.error.InternalError* (if not logged)

Raise *IOError*

Python 3:

Raise *odoorpc.error.InternalError* (if not logged)

Raise *PermissionError*

Raise *FileNotFoundError*

version
The version of the server.

```
>>> odoo.version
'10.0'
```

3.4.4 odoorpc.db

Provide the *DB* class to manage the server databases.

class odoorpc.db.**DB**(*odoo*)
The *DB* class represents the database management service. It provides functionalities such as list, create, drop, dump and restore databases.

Note: This service have to be used through the *odoorpc.ODOO.db* property.

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost')
>>> odoo.db
<odoorpc.db.DB object at 0x...>
```

change_password(*password*, *new_password*)

Change the administrator password by *new_password*.

```
>>> odoo.db.change_password('super_admin_passwd', 'new_admin_passwd')
```

The super administrator password is required to perform this method.

Python 2:

Raise *odoorpc.error.RPCError* (access denied)

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError* (access denied)

Raise *urllib.error.URLError* (connection error)

create(*password*, *db*, *demo=False*, *lang='en_US'*, *admin_password='admin'*)

Request the server to create a new database named *db* which will have *admin_password* as administrator password and localized with the *lang* parameter. You have to set the flag *demo* to *True* in order to insert demonstration data.

```
>>> odoo.db.create('super_admin_passwd', 'prod', False, 'fr_FR', 'my_admin_
↪passwd')
```

If you get a timeout error, increase this one before performing the request:

```
>>> timeout_backup = odoo.config['timeout']
>>> odoo.config['timeout'] = 600    # Timeout set to 10 minutes
>>> odoo.db.create('super_admin_passwd', 'prod', False, 'fr_FR', 'my_admin_
↪passwd')
>>> odoo.config['timeout'] = timeout_backup
```

The super administrator password is required to perform this method.

Python 2:

Raise *odoorpc.error.RPCError* (access denied)

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError* (access denied)

Raise *urllib.error.URLError* (connection error)

drop(*password*, *db*)

Drop the *db* database. Returns *True* if the database was removed, *False* otherwise (database did not exist):

```
>>> odoo.db.drop('super_admin_passwd', 'test')
True
```

The super administrator password is required to perform this method.

Python 2:

Returns *True* or *False*

Raise *odoorpc.error.RPCError* (access denied)

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *True* or *False*

Raise *odoorpc.error.RPCError* (access denied)

Raise *urllib.error.URLError* (connection error)

dump (*password*, *db*, *format_='zip'*)

Backup the *db* database. Returns the dump as a binary ZIP file containing the SQL dump file alongside the filestore directory (if any).

```
>>> dump = odoo.db.dump('super_admin_passwd', 'prod')
```

If you get a timeout error, increase this one before performing the request:

```
>>> timeout_backup = odoo.config['timeout']
>>> odoo.config['timeout'] = 600    # Timeout set to 10 minutes
>>> dump = odoo.db.dump('super_admin_passwd', 'prod')
>>> odoo.config['timeout'] = timeout_backup
```

Write it on the file system:

```
>>> with open('dump.zip', 'wb') as dump_zip:
...     dump_zip.write(dump.read())
...
```

You can manipulate the file with the *zipfile* module for instance:

```
>>> import zipfile
>>> zipfile.ZipFile('dump.zip').namelist()
['dump.sql',
'filestore/ef/ef2c882a36dbe90fc1e7e28d816ad1ac1464cfbb',
'filestore/dc/dcf00aacce882bbfd117c0277e514f829b4c5bf0',
 ...]
```

The super administrator password is required to perform this method.

Python 2:

Returns *io.BytesIO*

Raise *odoorpc.error.RPCError* (access denied / wrong database)

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *io.BytesIO*

Raise *odoorpc.error.RPCError* (access denied / wrong database)

Raise *urllib.error.URLError* (connection error)

duplicate(*password*, *db*, *new_db*)
Duplicate *db' as 'new_db*.

```
>>> odoo.db.duplicate('super_admin_passwd', 'prod', 'test')
```

The super administrator password is required to perform this method.

Python 2:

Raise *odoorpc.error.RPCError* (access denied / wrong database)

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError* (access denied / wrong database)

Raise *urllib.error.URLError* (connection error)

list()
Return the list of the databases:

```
>>> odoo.db.list()
['prod', 'test']
```

Python 2:

Returns *list* of database names

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *list* of database names

Raise *urllib.error.URLError* (connection error)

restore(*password*, *db*, *dump*, *copy=False*)
Restore the *dump* database into the new *db* database. The *dump* file object can be obtained with the *dump* method. If *copy* is set to *True*, the restored database will have a new UUID.

```
>>> odoo.db.restore('super_admin_passwd', 'test', dump_file)
```

If you get a timeout error, increase this one before performing the request:

```
>>> timeout_backup = odoo.config['timeout']
>>> odoo.config['timeout'] = 7200   # Timeout set to 2 hours
>>> odoo.db.restore('super_admin_passwd', 'test', dump_file)
>>> odoo.config['timeout'] = timeout_backup
```

The super administrator password is required to perform this method.

Python 2:

Raise *odoorpc.error.RPCError* (access denied / database already exists)

Raise *odoorpc.error.InternalError* (dump file closed)

Raise *urllib2.URLError* (connection error)

Python 3:

Raise *odoorpc.error.RPCError* (access denied / database already exists)

Raise *odoorpc.error.InternalError* (dump file closed)

Raise *urllib.error.URLError* (connection error)

3.4.5 odoorpc.report

This module provide the *Report* class to list available reports and to generate/download them.

class odoorpc.report.**Report**(*odoo*)

The *Report* class represents the report management service. It provides methods to list and download available reports from the server.

Note: This service have to be used through the *odoorpc.ODOO.report* property.

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost', port=8069)
>>> odoo.login('odoorpc_test', 'admin', 'password')
>>> odoo.report
<odoorpc.report.Report object at 0x7f82fe7a1d50>
```

download(*name*, *ids*, *datas=None*, *context=None*)

Download a report from the server and return it as a remote file. For instance, to download the "Quotation / Order" report of sale orders identified by the IDs `[2, 3]`:

```
>>> report = odoo.report.download('sale.report_saleorder', [2, 3])
```

Write it on the file system:

```
>>> with open('sale_orders.pdf', 'wb') as report_file:
...     report_file.write(report.read())
...
```

Python 2:

Returns *io.BytesIO*

Raise *odoorpc.error.RPCError* (wrong parameters)

Raise *ValueError* (received invalid data)

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *io.BytesIO*

Raise *odoorpc.error.RPCError* (wrong parameters)

Raise *ValueError* (received invalid data)

Raise *urllib.error.URLError* (connection error)

list()

List available reports from the server by returning a dictionary with reports classified by data model:

```
>>> odoo.report.list()['account.invoice']
[{'name': u'Duplicates',
  'report_name': u'account.account_invoice_report_duplicate_main',
  'report_type': u'qweb-pdf'},
 {'name': 'Invoices',
```

(continues on next page)

(continued from previous page)

```
  'report_type': 'qweb-pdf',
  'report_name': 'account.report_invoice'}]
```

Python 2:

Returns *list* of dictionaries

Raise *urllib2.URLError* (connection error)

Python 3:

Returns *list* of dictionaries

Raise *urllib.error.URLError* (connection error)

3.4.6 odoorpc.models

Provide the *Model* class which allow to access dynamically to all methods proposed by a data model.

class odoorpc.models.**Model**

Base class for all data model proxies.

Note: All model proxies (based on this class) are generated by an *environment* (see the *odoorpc.ODOO.env* property).

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost', port=8069)
>>> odoo.login('db_name', 'admin', 'password')
>>> User = odoo.env['res.users']
>>> User
Model('res.users')
```

Use this data model proxy to call any method:

```
>>> User.name_get([1])  # Use any methods from the model class
[[1, 'Administrator']]
```

Get a recordset:

```
>>> user = User.browse(1)
>>> user.name
'Administrator'
```

And call any method from it, it will be automatically applied on the current record:

```
>>> user.name_get()      # No IDs in parameter, the method is applied on the␣
↪current recordset
[[1, 'Administrator']]
```

Warning: Excepted the *browse* method, method calls are purely dynamic. As long as you know the signature of the model method targeted, you will be able to use it (see the *tutorial*).

__eq__(*other*)

x.__eq__(y) <==> x==y

__getattr__(*method*)
Provide a dynamic access to a RPC *instance* method (which applies on the current recordset).

```
>>> Partner = odoo.env['res.partner']
>>> Partner.write([1], {'name': 'YourCompany'}) # Class method
True
>>> partner = Partner.browse(1)
>>> partner.write({'name': 'YourCompany'})      # Instance method
True
```

__getitem__(*key*)
If *key* is an integer or a slice, return the corresponding record selection as a recordset.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

__iter__()
Return an iterator over *self*.

__ne__(*other*)
x.__ne__(y) <==> x!=y

__repr__() <==> *repr(x)*

classmethod browse(*ids*)
Browse one or several records (if *ids* is a list of IDs).

```
>>> odoo.env['res.partner'].browse(1)
Recordset('res.partner', [1])
```

```
>>> [partner.name for partner in odoo.env['res.partner'].browse([1, 3])]
['YourCompany', 'Administrator']
```

A list of data types returned by such record fields are available *here*.

Returns a *Model* instance (recordset)

Raise *odoorpc.error.RPCError*

id
ID of the record (or the first ID of a recordset).

ids
IDs of the recorset.

with_context(**args*, ***kwargs*)
Return an instance equivalent to *self* attached to an environment based on *self.env* with another context. The context is taken from *self.env* or from the positional argument if given, and modified by *kwargs*.

Thus, the following two examples are equivalent:

```
>>> Product = odoo.env['product.product']
>>> product = Product.browse(1)
>>> product.with_context(lang='fr_FR')
Recordset('product.product', [1])
```

```
>>> context = product.env.context
>>> product.with_context(context, lang='fr_FR')
Recordset('product.product', [1])
```

This method is very convenient to update translations:

```
>>> product_en = Product.browse(1)
>>> product_en.env.lang
'en_US'
>>> product_en.name = "My product"  # Update the english translation
>>> product_fr = product_en.with_context(lang='fr_FR')
>>> product_fr.env.lang
'fr_FR'
>>> product_fr.name = "Mon produit" # Update the french translation
```

with_env(*env*)
Return an instance equivalent to *self* attached to *env*.

3.4.7 odoorpc.env

Supply the *Environment* class to manage records more efficiently.

class odoorpc.env.**Environment**(*odoo*, *db*, *uid*, *context*)
An environment wraps data like the user ID, context or current database name, and provides an access to data model proxies.

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost')
>>> odoo.login('db_name', 'admin', 'password')
>>> odoo.env
Environment(db='db_name', uid=1, context={'lang': 'fr_FR', 'tz': 'Europe/Brussels
↪', 'uid': 1})
```

__contains__(*model*)
Check if the given *model* exists on the server.

```
>>> 'res.partner' in odoo.env
True
```

Returns *True* or *False*

__getitem__(*model*)
Return the model class corresponding to *model*.

```
>>> Partner = odoo.env['res.partner']
>>> Partner
Model('res.partner')
```

Returns a *odoorpc.models.Model* class

context
The context of the user connected.

```
>>> odoo.env.context
{'lang': 'en_US', 'tz': 'Europe/Brussels', 'uid': 1}
```

db
The database currently used.

```
>>> odoo.env.db
'db_name'
```

lang
Return the current language code.

```
>>> odoo.env.lang
'en_US'
```

ref (*xml_id*)
Return the record corresponding to the given *xml_id* (also called external ID). Raise an *RPCError* if no record is found.

```
>>> odoo.env.ref('base.lang_en')
Recordset('res.lang', [1])
```

Returns a *odoorpc.models.Model* instance (recordset)

Raise *odoorpc.error.RPCError*

registry
The data model registry. It is a mapping between a model name and its corresponding proxy used to generate records. As soon as a model is needed the proxy is added to the registry. This way the model proxy is ready for a further use (avoiding costly *RPC* queries when browsing records through relations).

```
>>> odoo.env.registry
{}
>>> odoo.env.user.company_id.name    # 'res.users' and 'res.company' Model␣
↪proxies will be fetched
'YourCompany'
>>> from pprint import pprint
>>> pprint(odoo.env.registry)
{'res.company': Model('res.company'), 'res.users': Model('res.users')}
```

If you need to regenerate the model proxy, simply delete it from the registry:

```
>>> del odoo.env.registry['res.company']
```

To delete all model proxies:

```
>>> odoo.env.registry.clear()
>>> odoo.env.registry
{}
```

uid
The user ID currently logged.

```
>>> odoo.env.uid
1
```

user
Return the current user (as a record).

```
>>> user = odoo.env.user
>>> user
Recordset('res.users', [1])
>>> user.name
'Administrator'
```

Returns a *odoorpc.models.Model* instance

Raise *odoorpc.error.RPCError*

3.4.8 odoorpc.rpc

This module provides *Connector* classes to communicate with an *Odoo* server with the *JSON-RPC* protocol or through simple HTTP requests.

Web controllers of *Odoo* expose two kinds of methods: *json* and *http*. These methods can be accessed from the connectors of this module.

class odoorpc.rpc.**Connector**(*host*, *port=8069*, *timeout=120*, *version=None*)

Connector base class defining the interface used to interact with a server.

ssl

Return *True* if SSL is activated.

timeout

Return the timeout.

class odoorpc.rpc.**ConnectorJSONRPC**(*host*, *port=8069*, *timeout=120*, *version=None*, *deserialize=True*, *opener=None*)

Connector class using the *JSON-RPC* protocol.

```
>>> from odoorpc import rpc
>>> cnt = rpc.ConnectorJSONRPC('localhost', port=8069)
```

Open a user session:

```
>>> cnt.proxy_json.web.session.authenticate(db='db_name', login='admin', password=
↪'password')
{'id': 51373612,
 'jsonrpc': '2.0',
 'result': {'company_id': 1,
            'currencies': {'1': {'digits': [69, 2],
                                 'position': 'after',
                                 'symbol': '\u20ac'},
                           '3': {'digits': [69, 2],
                                 'position': 'before',
                                 'symbol': '$'}},
            'db': 'db_name',
            'is_admin': True,
            'is_superuser': True,
            'name': 'Administrator',
            'partner_id': 3,
            'server_version': '10.0',
            'server_version_info': [10, 0, 0, 'final', 0, ''],
            'session_id': '6dd7a34f16c1c67b38bfec413cca4962d5c01d53',
            'uid': 1,
            'user_companies': False,
            'user_context': {'lang': 'en_US',
                             'tz': 'Europe/Brussels',
                             'uid': 1},
            'username': 'admin',
            'web.base.url': 'http://localhost:8069',
            'web_tours': []}}
```

Read data of a partner:

```
>>> cnt.proxy_json.web.dataset.call(model='res.partner', method='read',␣
↪args=[[1]])
{'jsonrpc': '2.0', 'id': 454236230,
 'result': [{'id': 1, 'comment': False, 'ean13': False, 'property_account_position
↪': False, ...}]}
```

You can send requests this way too:

```
>>> cnt.proxy_json['/web/dataset/call'](model='res.partner', method='read',␣
↪args=[[1]])
{'jsonrpc': '2.0', 'id': 328686288,
 'result': [{'id': 1, 'comment': False, 'ean13': False, 'property_account_position
↪': False, ...}]}
```

Or like this:

```
>>> cnt.proxy_json['web']['dataset']['call'](model='res.partner', method='read',␣
↪args=[[1]])
{'jsonrpc': '2.0', 'id': 102320639,
 'result': [{'id': 1, 'comment': False, 'ean13': False, 'property_account_position
↪': False, ...}]}
```

proxy_http
Return the HTTP proxy.

proxy_json
Return the JSON proxy.

timeout
Return the timeout.

class odoorpc.rpc.**ConnectorJSONRPCSSL**(*host*, *port=8069*, *timeout=120*, *version=None*, *deserialize=True*, *opener=None*)
Connector class using the *JSON-RPC* protocol over *SSL*.

```
>>> from odoorpc import rpc
>>> cnt = rpc.ConnectorJSONRPCSSL('localhost', port=8069)
```

3.4.9 odoorpc.session

This module contains some helper functions used to save and load sessions in *OdooRPC*.

odoorpc.session.**get**(*name*, *rc_file='~/.odoorpcrc'*)
Return the session configuration identified by *name* from the *rc_file* file.

```
>>> import odoorpc
>>> from pprint import pprint as pp
>>> pp(odoorpc.session.get('foo'))
{'database': 'db_name',
 'host': 'localhost',
 'passwd': 'password',
 'port': 8069,
 'protocol': 'jsonrpc',
 'timeout': 120,
 'type': 'ODOO',
 'user': 'admin'}
```

Raise *ValueError* (wrong session name)

odoorpc.session.**get_all**(*rc_file='~/.odoorpcrc'*)
Return all session configurations from the *rc_file* file.

```
>>> import odoorpc
>>> from pprint import pprint as pp
>>> pp(odoorpc.session.get_all())
{'foo': {'database': 'db_name',
         'host': 'localhost',
         'passwd': 'password',
         'port': 8069,
         'protocol': 'jsonrpc',
         'timeout': 120,
         'type': 'ODOO',
         'user': 'admin'},
 ...}
```

odoorpc.session.**remove**(*name*, *rc_file='~/.odoorpcrc'*)
Remove the session configuration identified by *name* from the *rc_file* file.

```
>>> import odoorpc
>>> odoorpc.session.remove('foo')
```

Raise *ValueError* (wrong session name)

odoorpc.session.**save**(*name*, *data*, *rc_file='~/.odoorpcrc'*)
Save the *data* session configuration under the name *name* in the *rc_file* file.

```
>>> import odoorpc
>>> odoorpc.session.save(
...     'foo',
...     {'type': 'ODOO', 'host': 'localhost', 'protocol': 'jsonrpc',
...      'port': 8069, 'timeout': 120, 'database': 'db_name'
...      'user': 'admin', 'passwd': 'password'})
```

3.4.10 odoorpc.tools

This module contains the `Config` class which manage the configuration related to an instance of *ODOO*, and some useful helper functions used internally in *OdooRPC*.

class odoorpc.tools.**Config**(*odoo*, *options*)
Class which manage the configuration of an *ODOO* instance.

Note: This class have to be used through the *odoorpc.ODOO.config* property.

```
>>> import odoorpc
>>> odoo = odoorpc.ODOO('localhost')
>>> type(odoo.config)
<class 'odoorpc.tools.Config'>
```

odoorpc.tools.**clean_version**(*version*)
Clean a version string.

```
>>> from odoorpc.tools import clean_version
>>> clean_version('7.0alpha-20121206-000102')
'7.0'
```

Returns a cleaner version string

odoorpc.tools.**get_encodings**(*hint_encoding='utf-8'*)

Used to try different encoding. Function copied from Odoo 11.0 (odoo.loglevels.get_encodings). This piece of code is licensed under the LGPL-v3 and so it is compatible with the LGPL-v3 license of OdooRPC:

```
- https://github.com/odoo/odoo/blob/11.0/LICENSE
- https://github.com/odoo/odoo/blob/11.0/COPYRIGHT
```

odoorpc.tools.**v**(*version*)

Convert a version string to a tuple. The tuple can be use to compare versions between them.

```
>>> from odoorpc.tools import v
>>> v('7.0')
[7, 0]
>>> v('6.1')
[6, 1]
>>> v('7.0') < v('6.1')
False
```

Returns the version as tuple

3.4.11 odoorpc.error

This module contains all exceptions raised by *OdooRPC* when an error occurred.

exception odoorpc.error.**Error**

Base class for exception.

exception odoorpc.error.**InternalError**

Exception raised for errors occurring during an internal operation.

exception odoorpc.error.**RPCError**(*message*, *info=False*)

Exception raised for errors related to RPC queries. Error details (like the *Odoo* server traceback) are available through the *info* attribute:

```
>>> from pprint import pprint as pp
>>> try:
...     odoo.execute('res.users', 'wrong_method')
... except odoorpc.error.RPCError as exc:
...     pp(exc.info)
...
{'code': 200,
 'data': {'arguments': ["type object 'res.users' has no attribute 'wrong_method'
↪"],
          'debug': 'Traceback (most recent call last):\n  File ...',
          'exception_type': 'internal_error',
          'message': "'res.users' object has no attribute 'wrong_method'",
          'name': 'exceptions.AttributeError'}
 'message': 'Odoo Server Error'}
```

CHAPTER 4

Supported Odoo server versions

OdooRPC has been tested on *Odoo* 8.0, 9.0, 10.0 and 11.0. It should work on next versions if *Odoo* keeps a stable API.

CHAPTER 5

Supported Python versions

OdooRPC support Python 2.7, 3.4, 3.5 and 3.6.

CHAPTER 6

License

This software is made available under the *LGPL v3* license.

CHAPTER 7

Bug Tracker

Bugs are tracked on GitHub Issues. In case of trouble, please check there if your issue has already been reported. If you spotted it first, help us smash it by providing detailed and welcomed feedback.

CHAPTER 8

Credits

8.1 Contributors

- Sébastien Alix <sebastien.alix@osiell.com>

Do not contact contributors directly about support or help with technical issues.

8.2 Maintainer

This package is maintained by the OCA.

OCA, or the Odoo Community Association, is a nonprofit organization whose mission is to support the collaborative development of Odoo features and promote its widespread use.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

Symbols

B

C

D

E

G

H

I

J

L

M

O

P

R

S

T

U

V

W