OpenERP OpenLib Documentation

Release 0.2.4

Thibaut DIRLIK

April 27, 2013

CONTENTS

You can download OpenLib on its github page : http://github.com/WE2BS/openerp-openlib

Note: This document refers to version 0.2.4

CHAPTER

ONE

ORM EXTENSION

OpenLib provides an extension to the basic OpenERP ORM. Its main goal is to simplify everyday call to the OpenERP API. For example, you don't have to pass the `cr`, `uid` or `context` variables anymore. The ORM Extension is not intrusive, you can enable it on your objects if you want, but it's not mandatory at all.

1.1 OpenLib ORM Extension

1.1.1 Introduction

To use the OpenLib ORM Extension, you must import ExtendedOsv and Q classes:

```
from openlib.orm import ExtendedOsv, Q
```

If you want your objects to natively support the extension, make them inherit from ExtendedOsv:

```
class MyObject(osv.osv, ExtendedOsv):
    ...
```

Because OpenERP native objects does not inherit from `ExtendedOsv`, you can't directly call the new methods on these objects pools. You will have to pass throught an object which inherits from `ExtendedOsv`.

1.1.2 The ExtendedOsv class

class `openlib.orm.`**`ExtendedOsv`**

Every object which inherit from this class can use the following methods. These methods support a *django-like style* and doesn't require you to pass them `cr`, `uid` or `context` variables. These variables are recovered from the *execution stack*. This means that you **must** have variables named *cr*, *uid*, and *context* (optional) when you call these methods. Generally, these variables are passed by OpenERP.

Note: All the methods described below supports `_cr`, `_uid` and `_context` arguments to override the ones found automatically in the python stack. We use _ at the begin of arguments for methods which support *django-like searching* by arguments to avoid conflicts.

---

find

ExtendedOsv.**find**([*q=None*, *_object=None*, *_offset=0*, *_limit=None*, *_order=None*, *_count=None*, ***kwargs*])

This methods is an equivalent to the builtin `search()` method but let you use a django-like syntax or `Q` objects instead of the polish notation used in `search()`.

Parameters

- **q** – A `Q` object (the query).
- **kwargs** – *Search keywords* if you don't use `Q`.

Returns A list of integers, corresponding to ids found.

Note: If you specify one of the `_limit`, `_offset`, `_order` or `_count` arguments, they will be passed to `search()`.

Examples

Find partners with name='Agrolait':

```
partners_ids = self.find(name='Agrolait', _object='res.partners')
```

Find partners with name='Agrolait' or 'AsusTek':

```
partners_ids = self.find(Q(name='Agrolait') | Q(name='AsusTek'), _object='res.partners')
```

In the case you are using `find()` on an object which inherit `ExtendedOsv`, you can omit the *_object* argument:

```
objects_ids = self.find(name='OK')
```

filter

ExtendedOsv.**filter**([*value=None*, *_object=None*, ***kwargs*])

This method is a kind of search-and-browse. It uses `find()` to search ids and then return the result of a `browse()` call so you can iterate over the results.

Parameters

- **value** – Can be a `Q` object or a list of ids.
- **kwargs** – *Search keywords* if you don't specify *value*.

Returns A list of objects as returned by `browse()`.

If you specify a list of ids, `find()` is not called. The corresponding objects are immediatly returned.

Examples

Iterate over partners whose name starts with 'A':

```
for partner in self.filter(name__startswith='A', _object='res.partner'):
    ...
```

Almost same with a `Q` object:

```
for partner in self.filter(Q(name__startswith='A') | Q(name__startswith='B'), _object='res.partner'):
    ...
```

Iterate over a list of ids of one of our objects:

```
for obj in self.filter([1, 2, 3]):
    ...
```

get

ExtendedOsv.**get**([*value=None*, *_object=None*, ***kwargs*])

This method act like `filter()` but returns only one object. *value* can be one of the following :

- An `integer`, then the object corresponding to this id is returned
- A `string`, then the object with this XMLID is returned
- A `Q` object, return the first object corresponding to the criteria.
- `None`, then the first object corresponding to the *search keywords* is returned

Parameters

- **value** – The search criteria (see above)
- **kwargs** – If *value* is None, *search keywords*

Returns An object as returned by `browse()` or `None`.

Examples

Returns the group whose XMLID is 'group_employee':

```
group = self.get('base.group_employee', _object='res.groups')
```

Returns the user with the id 1:

```
admin = self.get(1, _object='res.users')
```

Returns the first partner whose name is 'Agrolait':

```
partner = self.get(name='Agrolait', _object='res.partner')
```

get_pools

ExtendedOsv.**get_pools**(**args*)

An equivalent of `sel.pool.get` which supports more than one argument.

Returns A list of pool objects for each pool name passed as argument.

Example

```
partner_pool, config_pool = self.get_pools('res.partner', 'openlib.config')
```

xmlid_to_id

ExtendedOsv.**xmlid_to_id**(*cr*, *uid*, *xmlid*, *context=None*)

This method returns the database ID corresponding the `xmlid` passed, or `None`.

Note: This method does not uses automatic detection of `cr`, `uid` and `context`.

1.1.3 Query Objects

class `openlib.orm.Q`

This class let you create complex search query easily. It uses *django-like keyword arguments* to define search criteria. These objects can be combined with `&` or `|` and prefixed with − to negate them :

```
criteria = Q(name='Peter', age=12) | Q(name='Paul')
```

This example will be translated into this SQL request :

```
(name='Peter' AND age=12) OR name='Paul'
```

Prefixing `Q` objects with a minus sign will negate them:

```
criteria = -Q(name='Paul')
```

Which means *name IS NOT Paul*. You can create complex search expressions like this one :

```
criteria = (Q(name='Paul') | Q(name='Pierre)) & Q(age=12)) | -Q(age=12)
```

For a detailed description the keywords arguments, read *Keywords arguments format*.

1.1.4 Keywords arguments format

With OpenLib, `Q` objects and `ExtendedOsv` class methods supports keyword argument formatting to specify you search criteria. The simple form of the keyword argument is :

```
name='value'
```

Where *name* is the name of a column. But you can specify a lookup method using this syntax :

```
column__lookup='value'
```

Where *lookup* can be one of the following values :

- `exact` - The default, same as not specifying a lookup method.
- `iexact` - Same as *exact*, but case insensitive.
- `like` - Performes an SQL LIKE with the value.
- `ilike` - Same as *like* but case insensitive.
- `gt` - Greater than, same as '>'.
- `lt` - Lesser than, same as '<'.
- `ge` - Geather than or equal, same as '>='.
- `le` - Lesser than or equal, same as '<='.
- `startswith` / `istartswith` - A shortcut to `LIKE 'Value%'`. The value is *like-protected* (special chars like `%` or _ are escaped).
- `endswith` / `iendswith` - A shortcut to `LIKE '%Value'`. Value is like-protected.
- `contains` / `icontains` - A shortcut to `LIKE '%Value%'`. Value is like-protected.

The column name can be separated with '__' to represent a relation:

```
Q(partner__address__country__code='Fr')
```

Warning: If you have a column which have the same name that a lookup method, you must repeat it (xxx__exact__exact).

Examples

Using Q objects:

```
self.filter(Q(name__startswith='P') | Q(age__gt=12))
```

Using relation without Q objects:

```
self.find(address__city='Paris', _object='res.partners')
```

CHAPTER

TWO

GITHUB AUTOMATIC BUG REPORTS

OpenLib integrates very well with GitHub and supports automatic bug reporting. This means that each time an exception is raised in your code, OpenLib will check your github project and reports the bug it hasn't been reported.

Of course, this won't report any logical bugs (Like workflow errors, or "nothing happens" bugs), but code-related bug will be reported, without any intervention from the user.

2.1 OpenLib Github Integration

OpenLib provides an easy way to automatically report bugs which happen in your modules on github. You have to configure the github repository you want to report bugs on, and it will work.

2.1.1 Github module configuration

For the example, we will imagine you are writing a module named `example`, hosted in a github repository named `openerp-example` by the organization `orga`. You just have to add three variables to your module's `__init__.py`:

```
GITHUB_ENABLED = True
GITHUB_REPO = 'openerp-example'
GITHUB_USER = 'orga'
```

Note: Setting `GITHUB_ENABLED` to `False` will disable github bug reporting. Remember to unset it during developpement.

2.1.2 Define the functions you want to watch

OpenLib can't watch all your module's method. You must tell it the one you want to watch. To do this, you just have to import the `report_bugs()` function for `openlib.github`:

```
from openlib.github import report_bugs

class MyObject(osv.osv):

    @report_bugs
    def on_change_product(...):
        ...
```

Each time an exception is raised in this method, OpenLib will check if it has already been reported. If it's not the case, a new issue will be opened on the github project you specified in `__init__.py`.

Note: Using this decorator on a lot of functions won't cause any performance problem. The only overhead is a `try .. except` block around your method call, you won't see the difference.

2.1.3 Define the account used to report bugs

To be able to report bugs, you must have a GitHub account. This configuration is done by database, and must be set by the administrator into the menu *Administration->Customization->Variables*. There are two variables, named `GITHUB_USER` and `GITHUB_TOKEN` you must fill.

You can find you token on your github account settings : *Account settings->Account admin->API Token*.

Note: OpenLib provides an installation wizard which does that automatically.

CHAPTER

THREE

GLOBAL CONFIGURATION

OpenLib let you define global variables (database-wide) easily.

3.1 OpenLib Global Configuration

Sometimes, you need to store data not attached to a specific object, a kind of *Global variable*. OpenLib let you do this with `openlib.config` object. This is a simple table with 3 columns, `module`, `key` and `value`.

This object implements the `ExtendedOsv` interface, so it can be manipulated easily. Data are stored as charfield and have maximum size of 255 characters. You can store pickled object, if you want.

3.1.1 Access a global variable

OpenLib uses this object internally to store Github credentials, for example, if you want to get the github login:

```
login = self.pool.get('openlib.config').get(module='openlib.github', key='GITHUB_USER').value
```

This is the *normal* way, but `openlib.config` provides a method which returns `None` if the key is not defined:

```
login = self.pool.get('openlib.config').get_value('openlib.github', 'GITHUB_USER')
```

Note: The second way it the safest, because it won't raise an `AtributeError` if the key is not defined.

3.1.2 Define a global variable

With an XML file

You can easily create yours variables thanks to an XML file :

```
<?xml version="1.0" encoding="utf-8"?>
<openerp>
    <data>
        <record id="config_github_user" model="openlib.config">
            <field name="module">openlig.github</field>
            <field name="key">GITHUB_USER</field>
            <field name="help">GitHub user account used to report bugs.</field>
        </record>
        <record id="config_github_token" model="openlib.config">
```

```
            <field name="module">openlig.github</field>
            <field name="key">GITHUB_TOKEN</field>
            <field name="help">GitHub token associated to the account. Check your account settings.</
        </record>
    </data>
</openerp>
```

You can provide a default value, just by adding :

```
<field name="value">default_value</field>
```

Into the record.

With Python code

You can also update/create a configuration variable with Python. Like with when you access the variable, you have two methods to do this : The *normal* way, and the shorter and recommended way :

Using write (normal way):

```
self.pool.get('openlib.config').write(cr, uid, config_id, {'value' : 'XXXXX'}, context=context)
```

Using this method implies that you already know the ID of the global variable object. If it does not exists, you have to create it with the `create()` method. To make your life simpler, OpenLib provides a `set_value` method:

```
self.pool.get('openlib.config').set_value('openlib.github', 'GITHUB_USER', 'XXXXX')
```

This method will create the entry if it doesn't exist, and update it if it does.

CHAPTER

FOUR

MISC TOOLS

Others tools provided by OpenLib.

4.1 OpenLib Tools

This module contains functions that could be useful.

4.1.1 Date and time tools

All these functions uses the OpenERP default timestamps by default as format.

`openlib.tools.`**`to_date`**(*date_string*, *format=DEFAULT_SERVER_DATE_FORMAT*)
Converts the *date_string* passed as an argument to a `datetime.date` object.

`openlib.tools.`**`to_time`**(*time_string*, *format=DEFAULT_SERVER_TIME_FORMAT*)
Converts the *time_string* argument to a `datetime.time` object.

`openlib.tools.`**`to_datetime`**(*datetime_string*, *format=DEFAULT_SERVER_DATETIME_FORMAT*)
Converts the *datetime_string* argument to a `datetime.datetime` object.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

o