From V7 to V8: The New API

Raphael Collet (rco@odoo.com)

odoo

Goal

Make the model API

- more object-oriented
- more Pythonic
- less cluttered

From V7 to V8

- compatibility: V7 and V8 APIs are interoperable

Agenda

- Recordsets
- Methods
- Environment
- Fields
- Migrating Modules

Recordsets

Programming with Recordsets

A **recordset** is:

- an ordered collection of records
- one concept to replace
  - browse records,
  - browse lists,
  - browse nulls.
- an instance of the model's class

The recordset as a collection

It implements sequence and set operations:

```
partners = env['res.partner'].search([])

for partner in partners:
    assert partner in partners
    print partner.name

if len(partners) >= 5:
    fifth = partners[4]

extremes = partners[:10] + partners[-10:]

union = partners1 | partners2
intersection = partners1 & partners2
difference = partners1 - partners2
```

The recordset as a record

It behaves just like former browse records:

```
print partner.name
print partner['name']
print partner.parent_id.company_id.name
```

Except that updates are written to database:

```
partner.name = 'Agrolait'
partner.email = 'info@agrolait.com'
partner.parent_id = ...                    # another record
```

The recordset as a record

If `len(partners) > 1`, do it on the **first** record:

```
print partners.name                    # name of first partner
print partners[0].name

partners.name = 'Agrolait'             # assign first partner
partners[0].name = 'Agrolait'
```

If `len(partners) == 0`, return the null value of the field:

```
print partners.name                    # False
print partners.parent_id               # empty recordset

partners.name = 'Foo'                  # ERROR
```

The recordset as an instance

Methods of the model's class can be invoked on recordsets:

```
# calling convention: leave out cr, uid, ids, context

@api.multi
def write(self, values):
    result = super(C, self).write(values)

    # search returns a recordset instead of a list of ids
    domain = [('id', 'in', self.ids), ('parent_id', '=', False)]
    roots = self.search(domain)

    # modify all records in roots
    roots.write({'modified': True})

    return result
```

The missing parameters are hidden inside the recordset.

Methods

Method decorators

Decorators enable support of **both** old and new API:

```
from odoo import Model, api

class stuff(Model):

    @api.model
    def create(self, values):
        # self is a recordset, but its content is unused
        ...
```

This method definition is equivalent to:

```
class stuff(Model):

    def create(self, cr, uid, values, context=None):
        # self is not a recordset
        ...
```

Method decorators

```
from odoo import Model, api

class stuff(Model):

    @api.multi
    def write(self, values):
        # self is a recordset and its content is used
        # update self.ids
        ...
```

This method definition is equivalent to:

```
class stuff(Model):

    def multi(self, cr, uid, ids, values, context=None):
        # self is not a recordset
        ...
```

Method decorators

One-by-one or "autoloop" decorator:

```
from odoo import Model, api

class stuff(Model):

    @api.one
    def cancel(self):
        self.state = 'cancel'
```

When invoked, the method is applied on every record:

```
recs.cancel()                    # [rec.cancel() for rec in recs]
```

Method decorators

Methods that return a recordset instead of ids:

```
from odoo import Model, api

class stuff(Model):

    @api.multi
    @api.returns('res.partner')
    def root_partner(self):
        p = self.partner_id
        while p.parent_id:
            p = p.parent_id
        return p
```

When called with the old API, it returns ids:

```
roots = recs.root_partner()

root_ids = model.root_partner(cr, uid, ids, context=None)
```

odoo

Environment: cr, uid, context

The environment object

Encapsulates cr, uid, context:

```
# recs.env encapsulates cr, uid, context
recs.env.cr                         # shortcut: recs._cr
recs.env.uid                        # shortcut: recs._uid
recs.env.context                    # shortcut: recs._context

# recs.env also provides helpers
recs.env.user                       # uid as a record

recs.env.ref('base.group_user')     # resolve xml id

recs.env['res.partner']             # access to new-API model
```

The environment object

Switching environments:

```
# rebrowse recs with different parameters
env2 = recs.env(cr2, uid2, context2)
recs2 = recs.with_env(env2)

# special case: change/extend the context
recs2 = recs.with_context(context2)
recs2 = recs.with_context(lang='fr')    # kwargs extend current context

# special case: change the uid
recs2 = recs.sudo(user)
recs2 = recs.sudo()                     # uid = SUPERUSER_ID
```

Fields

Fields as descriptors

Python descriptors provide getter/setter (like `property`):

```
from odoo import Model, fields

class res_partner(Model):
    _name = 'res.partner'

    name = fields.Char(required=True)
    parent_id = fields.Many2one('res.partner', string='Parent')
```

Computed fields

Regular fields with the name of the compute method:

```
class res_partner(Model):
    ...

    display_name = fields.Char(
        string='Name', compute='_compute_display_name',
    )

    @api.one
    @api.depends('name', 'parent_id.name')
    def _compute_display_name(self):
        names = [self.parent_id.name, self.name]
        self.display_name = ' / '.join(filter(None, names))
```

Computed fields

The compute method must assign field(s) on records:

```
untaxed = fields.Float(compute='_amounts')
taxes = fields.Float(compute='_amounts')
total = fields.Float(compute='_amounts')

@api.multi
@api.depends('lines.amount', 'lines.taxes')
def _amounts(self):
    for order in self:
        order.untaxed = sum(line.amount for line in order.lines)
        order.taxes = sum(line.taxes for line in order.lines)
        order.total = order.untaxed + order.taxes
```

Computed fields

Stored computed fields are much easier now:

```
display_name = fields.Char(
    string='Name', compute='_compute_display_name', store=True,
)

@api.one
@api.depends('name', 'parent_id.name')
def _compute_display_name(self):
    ...
```

Field dependencies (`@depends`) are used for

- cache invalidation,
- recomputation,
- onchange.

Fields with inverse

On may also provide **inverse** and **search** methods:

```
class stuff(Model):
    name = fields.Char()
    loud = fields.Char(
        store=False, compute='_compute_loud',
        inverse='_inverse_loud', search='_search_loud',
    )

    @api.one
    @api.depends('name')
    def _compute_loud(self):
        self.loud = (self.name or '').upper()

    ...
```

Fields with inverse

```
class stuff(Model):
    name = fields.Char()
    loud = fields.Char(
        store=False, compute='_compute_loud',
        inverse='_inverse_loud', search='_search_loud',
    )

    ...

    @api.one
    def _inverse_loud(self):
        self.name = (self.loud or '').lower()

    def _search_loud(self, operator, value):
        if value is not False:
            value = value.lower()
        return [('name', operator, value)]
```

Onchange methods

For computed fields: **nothing to do!**

For other fields: API is similar to compute methods:

```
@api.onchange('partner_id')
def _onchange_partner(self):
    if self.partner_id:
        self.delivery_id = self.partner_id
```

The record `self` is a virtual record:

- all form values are set on `self`
- assigned values are not written to database but returned to the client

Onchange methods

A field element on a form is **automatically** decorated with `on_change="1"`:

- if it has an onchange method
- if it is a dependency of a computed field

This mechanism may be prevented by explicitly decorating a field element with `on_change="0"`.

Python constraints

Similar API, with a specific decorator:

```
@api.one
@api.constrains('lines', 'max_lines')
def _check_size(self):
    if len(self.lines) > self.max_lines:
        raise Warning(_("Too many lines in %s") % self.name)
```

The error message is provided by the exception.

odoo

Migrating Modules

Guidelines

Do the migration step-by-step:

1. migrate field definitions
    - rewrite compute methods
2. migrate methods
    - rely on interoperability
    - use decorators, they are necessary
3. rewrite onchange methods (incompatible API)
    - beware of overridden methods!

odoo

From V7 to V8: The New API

Thanks!