ADVANCED FEATURES OF THE API

RAPHAEL COLLET (RCO@ODOO.COM)

TOPICS

Less known/misunderstood API features:

- recordset operations
- environment
- record cache

ODOO 8.0: THE MODEL API

Reminder of the API essentials...

```
from openerp import models, fields, api

class Employee(models.Model):
    _name = 'employee'

    name = fields.Char(string="Name", required=True)
    manager_id = fields.Many2one('employee', string="Manager")
    active = fields.Boolean()

    @api.multi
    def fire(self):
        self.write({'active': False})
```

RECORDSETS

A recordset is a *collection* of records of a given model.
It is an instance of the model's class.

```
size = len(records)
recs1 = records[:10]
recs2 = records[20:]
recs = recs1 + recs2
recs.write({'name': 'Joe'})
```

A single record is represented as a recordset of size 1.

```
for record in records:
    assert type(record) == type(records)
    print record.name
    print len(record)
```

ENVIRONMENTS

Recordsets are attached to an *environment* and a sequence of record ids.

```
# list of record ids
records.ids

# environment with cr, uid, context
records._cr == records.env.cr
records._uid == records.env.uid
records._context == records.env.context

# access recordsets of another model
employees = records.env['employee'].search([])
```

OLD AND NEW API

A model's class has two kinds of instances: old-API and new-API ones (recordsets).

```
# registry contains old API model instances
old_model = registry['employee']

# browse() returns a recordset (new API model instance)
employees = old_model.browse(cr, uid, ids, context)

assert employees == employees.browse(ids)

# attribute _model returns the old API model instance
assert old_model == employees._model
```

METHOD DECORATORS

A wrapper to adapt the API of a *method call* to the API of the *method's definition*.

```
from openerp import models, fields, api

# method definitions
@api.model                              @api.multi
def create(self, vals):                 def write(self, vals):
    ...                                     ...

# calling methods with the old API
id = old_model.create(cr, uid, vals, context=context)
old_model.write(cr, uid, ids, vals, context=context)

# calling methods with the new API
employee = employees.create(vals)
employees.write(vals)
```

METHOD DECORATORS

Specify return model:

```
@api.model
@api.returns('res.partner')
def get_default_partner(self):
    return self.env.ref('base.main_partner')

# new API call returns a recordset
partner = records.get_default_partner()

# old API call returns a list of ids
partner_ids = old_model.get_default_partner(cr, uid, context)
```

METHOD DECORATORS

Declare constraint methods:

```
@api.constrains('name')
def check_name(self):
    for record in self:
        if len(record.name) > 80:
            raise ValueError("Name is too long: %r" % record.name)
```

The method is expected to raise an exception when the constraint is not satisfied.

METHOD DECORATORS

Declare onchange methods:

```
@api.onchange('name')
def onchange_name(self):
    if self.name:
        self.other_name = self.name.upper()
    if len(self.name or "") > 40:
        return {'warning': {
                'title': _("Warning"),
                'message': _("Name migh be a bit long"),
        }}
```

"self" is a pseudo-record that only lives in cache.

COMPUTED FIELDS

Those fields are determined by a method that assigns their value on the given records.

```
from openerp import models, fields, api

class Employee(models.Model):
    _inherit = 'employee'

    title_id = fields.Many2one('employee.title', required=True)
    fullname = fields.Char(compute='_compute_fullname')

    @api.depends('title_id.name', 'name')
    def _compute_fullname(self):
        for emp in self:
            emp.fullname = "%s %s" % (emp.title_id.name, emp.name)
```

COMPUTED FIELDS

The inverse method modifies other fields based on the computed field's value

```
fullname = fields.Char(compute='_compute_fullname',
                       inverse='_inverse_fullname')

@api.depends('title_id.name', 'name')
def _compute_fullname(self):
    for emp in self:
        emp.fullname = "%s %s" % (emp.title_id.name, emp.name)

def _inverse_fullname(self):
    for emp in self:
        emp.name = ... emp.fullname ...
```

COMPUTED FIELDS

The search method transforms the domain condition ('fullname', operator, value) into another domain.

```python
fullname = fields.Char(compute='_compute_fullname',
                       search='_search_fullname')

@api.depends('title_id.name', 'name')
def _compute_fullname(self):
    for emp in self:
        emp.fullname = "%s %s" % (emp.title_id.name, emp.name)

def _search_fullname(self, operator, value):
    return ['|', ('title_id.name', operator, value),
                 ('name', operator, value)]
```

RECORDSET OPERATIONS

Combine and compare recordsets:

```
records[42]                        # indexing or slicing (ordered)
records1 + records2                # concatenation (ordered)
records1 - records2                # difference (ordered)
records1 & records2                # set intersection (unordered)
records1 | records2                # set union (unordered)

record in records                  # test membership
records1 < records2                # set inclusion (unordered)
records1 == records2               # set equivalence (unordered)
```

RECORDSET OPERATIONS

Mapping, filtering, sorting:

```
# return a list of strings, like map() would do
records.mapped(lambda rec: rec.name)
records.mapped('name')

# return a recordset of res.partner
records.mapped(lambda rec: rec.partner_id)
records.mapped('partner_id')

# always return recordsets
records.filtered(lambda rec: rec.active)
records.sorted(key=lambda rec: rec.sequence)
```

BINDING TO ENVIRONMENT

Attach a recordset to another environment:

```
others = records.with_env(env)
assert others.ids == records.ids
assert others.env is env

# switch user
records.sudo(uid)                # integer
records.sudo(user)               # user record
records.sudo()                   # defaults to SUPERUSER_ID

# switch or extend records' context
records.with_context(context)
records.with_context(extra="stuff")
```

Beware: avoid them in loops.

THE RECORD CACHE

- Stores field values for records.

```
record.name
record._cache['name']
record.env.cache[field][record.id]
```

- Every environment has its *own* cache.
  - Cache contents depends on environment.
- Cache *prefetching* when accessing a field for the first time:
  - prefetches records browsed in the same environment;
  - prefetches fields from the same table.

HOW PREFETCHING WORKS

```
# no database access                        # (env.cache)
orders = env['sale.order'].browse(ids)      # ids

# no database access
order = orders[0]                           # ids

# prefetch order ids
partner = order.partner_id                  # data(ids), pids

# prefetch partner pids
name = partner.name                         # data(ids), data(pids)

# no database access
orders[-1].partner_id.name                  # data(ids), data(pids)
```

ISSUE #6276

Overridden computed field method:

```python
# Module 1
stuff = fields.Integer(compute='_get_stuff')

@api.one
def _get_stuff(self):
    self.stuff = self._context.get('stuff', 0)
```

```python
# Module 2
@api.one
def _get_stuff(self):
    other = self.with_context(stuff=42)
    super(Class, other)._get_stuff()
```

This raises a KeyError when getting the value...

ISSUE #6276: SOLUTION

The record "other" uses another cache than "self".

Simply copy result to the expected cache:

```python
# Module 2
@api.one
def _get_stuff(self):
    other = self.with_context(stuff=42)
    # determine other.stuff in other's cache
    super(Class, other)._get_stuff()
    # copy result from other's cache to self's cache
    self.stuff = other.stuff
```

BAD PREFETCHING

"Prefetch records browsed in the same environment."

```
# retrieve a list of (name, id)
action_ids = []
for ... in ...:
    # some query... -> name, id
    action_ids.append((name, id))

# process action_ids id by id
for name, id in action_ids:
    record = self.env[model].browse(id)
    if record.partner_id ...:
        ...
```

The second loop issues O(n) queries.

GOOD PREFETCHING

Browse all records *before* accessing their fields.

```python
# retrieve a list of (name, record)
action_records = []
for ... in ...:
    # some query... -> name, id
    record = self.env[model].browse(id)
    action_records.append((name, record))

# process action_records id by id
for name, record in action_records:
    if record.partner_id ...:
        ...
```

The second loop issues O(1) queries.

THE "ORMCACHE"

- Memoize results for a given method.
- Attached to the registry (database-specific).
- Explicitly invalidated (multi-worker invalidation).

```
from openerp.tools import ormcache

class decimal_precision(models.Model):

    @ormcache(skiparg=3)
    def precision_get(self, cr, uid, application):
        ...

    def create(self, cr, uid, vals, context=None):
        res = ...
        self.clear_caches()
        return res
```

WHAT TO ORMCACHE?

- For stuff that rarely changes between requests:
  - user permissions on models
  - XML views after inheritance
- Beware of what the cached value depends on:
  - uid, context
  - other parameters
- Cached value cannot refer to transaction-specific data:
  - records
  - environments

ORMCACHE API

V8: skiparg (kind of positional dependencies)

```
@ormcache(skiparg=3)
def precision_get(self, cr, uid, application):
    ...
```

V9: name functional dependencies (expressions)

```
@ormcache('application')
def precision_get(self, cr, uid, application):
    ...
```

odoo

THANK YOU

RAPHAEL COLLET (RCO@ODOO.COM)

Odoo
sales@odoo.com
www.odoo.com

R&D and Services office
Chaussée de Namur 40
B-1367 Grand-Rosière

Sales office
Avenue Van Nieuwenhuyse 5
B-1160 Brussels