Odoo new API guideline Documentation

Release 0.1

Nicolas Bessi

May 13, 2017

Contents

1 Record/Recordset and Model **3**
- 1.1 Model . . . 3
  - 1.1.1 Inheritance . . . 3
- 1.2 Recordset . . . 4
- 1.3 Supported Operations . . . 4
- 1.4 Useful helpers . . . 5
- 1.5 The ids Attribute . . . 5
- 1.6 Record . . . 5
  - 1.6.1 Displayed Name of Record . . . 6
  - 1.6.2 Active Record Pattern . . . 6
  - 1.6.3 Active Record Pattern Be Careful . . . 6
  - 1.6.4 Chain of Browse_null . . . 6

2 Environment **7**
- 2.1 Modifing Environment . . . 8
  - 2.1.1 Changing User . . . 8
  - 2.1.2 Accessing Current User . . . 8
  - 2.1.3 Fetching record using XML id . . . 8
- 2.2 Cleaning Environment Caches . . . 9

3 Common Actions **11**
- 3.1 Searching . . . 11
  - 3.1.1 search . . . 11
  - 3.1.2 search_read . . . 11
  - 3.1.3 search_count . . . 12
- 3.2 Browsing . . . 12
- 3.3 Writing . . . 12
  - 3.3.1 Using Active Record pattern . . . 12
  - 3.3.2 From Record . . . 12
  - 3.3.3 From RecordSet . . . 12
  - 3.3.4 Many2many One2many Behavior . . . 13
- 3.4 Copy . . . 13
  - 3.4.1 From Record . . . 13
  - 3.4.2 From RecordSet . . . 13
- 3.5 Create . . . 13
- 3.6 Dry run . . . 14

4 Using Cursor **15**

5 Using Thread **17**

6 New ids **19**

7 Fields **21**
7.1 Field inheritance . . . 21
7.2 Field types . . . 22
7.2.1 Boolean . . . 22
7.2.2 Char . . . 22
7.2.3 Text . . . 22
7.2.4 HTML . . . 22
7.2.5 Integer . . . 22
7.2.6 Float . . . 23
7.2.7 Date . . . 23
7.2.8 DateTime . . . 23
7.2.9 Binary . . . 24
7.2.10 Selection . . . 24
7.2.11 Reference . . . 24
7.2.12 Many2one . . . 24
7.2.13 One2many . . . 25
7.2.14 Many2many . . . 25
7.3 Name Conflicts . . . 25
7.4 Fields Defaults . . . 26
7.5 Computed Fields . . . 26
7.6 Inverse . . . 26
7.7 Multi Fields . . . 27
7.8 Related Field . . . 27
7.9 Property Field . . . 27
7.10 WIP copyable option . . . 27

8 Method and decorator **29**
8.1 @api.returns . . . 29
8.2 @api.one . . . 29
8.3 @api.multi . . . 30
8.4 @api.model . . . 30
8.5 @api.constrains . . . 30
8.6 @api.depends . . . 30
8.6.1 View management . . . 30
8.7 @api.onchange . . . 31
8.7.1 View management . . . 31
8.7.2 Warning and Domain . . . 31
8.8 @api.noguess . . . 31

9 Introspection **33**

10 Conventions **35**
10.1 Snake_casing or CamelCasing . . . 35
10.2 Imports . . . 35
10.2.1 Model . . . 35
10.2.2 Fields . . . 35
10.2.3 Translation . . . 35
10.2.4 API . . . 35
10.2.5 Exceptions . . . 36

10.3 Classes . . . 36
10.4 New Exceptions classes . . . 36
10.4.1 RedirectWarning . . . 36
10.4.2 AccessDenied . . . 36
10.4.3 AccessError . . . 36
10.4.4 class MissingError: . . . 36
10.4.5 DeferredException: . . . 37
10.4.6 Compatibility . . . 37
10.5 Fields . . . 37
10.6 Default or compute . . . 37
10.7 Modifing self in method . . . 37
10.8 Doing thing in dry run . . . 37
10.9 Using Cursor . . . 38
10.10 Displayed Name . . . 38
10.11 Constraints . . . 38
10.12 Qweb view or not Qweb view . . . 38
10.13 Javascript and Website related code . . . 38

11 Compatibility **39**
11.1 Access old API . . . 39
11.2 How to be polite with old code base . . . 39

12 Unittest **41**

13 YAML **43**

14 Indices and tables **45**

Overview

CHAPTER 1

Record/Recordset and Model

The new version 8.0 of OpenERP/Odoo introduce a new ORM API.

It intends to add a more coherent and concise syntax and provide a bi-directional compatibility.

The new API keeps its previous root design as Model and Record but now adds new concepts like Environment and Recordset.

Some aspects of the previous API will not change with this release, e.g. the domain syntax.

Model

A model is a representation of a business Object.

It is basically a class that define various class know-how and fields that are stored in database. All functions defined in a Model where previously callable directly by the Model.

This paradigm has changed as generally you should not access Model directly but a RecordSet see *Recordset*

To instantiate a model you must inherit an openerp.model.Model:

```
from openerp import models, fields, api, _


class MyModel(models.Model):

    _name = 'a.model'  # Model identifer used for table name

    firstname = fields.Char(string="Firstname")
```

Inheritance

The inheritance mechanism has not changed. You can use:

```
class MyModelExtended(Model):
     _inherit = 'a.model'                          # direct heritage
     _inherit = ['a.model, 'a.other.model']        # direct heritage
     _inherits = {'a.model': 'field_name'}         # polymorphic heritage
```

For more details about inheritance please have a look at

Inherit

for fields inheritance please read *Field inheritance*

Recordset

All instances of Model are at the same time instances of a RecordSet. A Recordset represents a sorted set of records of the same Model as the RecordSet.

You can call function on recordset:

```
class AModel(Model):
# ...
    def a_fun(self):
        self.do_something() # here self is a recordset a mix between class and set
        record_set = self
        record_set.do_something()

    def do_something(self):
        for record in self:
           print record
```

In this example the functions are defined at model level but when executing the code the `self` variable is in fact an instance of RecordSet containing many Records.

So the self passed in the `do_something` is a RecordSet holding a list of Records.

If you decorate a function with `@api.one` it will automagically loop on the Records of current RecordSet and self will this time be the current Record.

As described in *Record* you have now access to a pseudo Active-Record pattern

Note: If you use it on a RecordSet it will break if recordset does not contains only one item.!!

Supported Operations

RecordSet also support set operations you can add, union and intersect, ... recordset:

```
record in recset1       # include
record not in recset1   # not include
recset1 + recset2       # extend
recset1 | recset2       # union
recset1 & recset2       # intersect
recset1 - recset2       # difference
recset.copy()           # to copy recordset (not a deep copy)
```

Only the + operator preserves order

RecordSet can also be sorted:

```
sorted(recordset, key=lambda x: x.column)
```

Useful helpers

The new API provides useful helper on recordset to use them in a more functional apprach

You can filter an exisiting recordset quite easily:

```
recset.filtered(lambda record: record.company_id == user.company_id)
# or using string
recset.filtered("product_id.can_be_sold")
```

You can sort a recordset:

```
# sort records by name
recset.sorted(key=lambda r: r.name)
```

You can also use the operator module:

```
from operator import attrgetter
recset.sorted(key=attrgetter('partner_id', 'name'))
```

There is an helper to map recordsets:

```
recset.mapped(lambda record: record.price_unit - record.cost_price)

# returns a list of name
recset.mapped('name')

# returns a recordset of partners
recset.mapped('invoice_id.partner_id')
```

The ids Attribute

The ids attribute is a special attribute of RecordSet. It will be return even if there is more than one Record in RecordSet

Record

A Record mirrors a "populated instance of Model Record" fetched from database. It proposes abstraction over database using caches and query generation:

```
>>> record = self
>>> record.name
toto
>>> record.partner_id.name
partner name
```

Displayed Name of Record

With new API a notion of display name is introduced. It uses the function `name_get` under the hood.

So if you want to override the display name you should override the `display_name` field. Example

If you want to override both display name and computed relation name you should override `name_get`. Example

Active Record Pattern

One of the new features introduced by the new API is a basic support of the active record pattern. You can now write to database by setting properties:

```
record = self
record.name = 'new name'
```

This will update value on the caches and call the write function to trigger a write action on the Database.

Active Record Pattern Be Careful

Writing value using Active Record pattern must be done carefully. As each assignement will trigger a write action on the database:

```
@api.one
def dangerous_write(self):
  self.x = 1
  self.y = 2
  self.z = 4
```

On this sample each assignement will trigger a write. As the function is decorated with `@api.one` for each record in RecordSet write will be called 3 times. So if you have 10 records in recordset the number of writes will be 10*3 = 30.

This may cause some trouble on an heavy task. In that case you should do:

```
def better_write(self):
   for rec in self:
      rec.write({'x': 1, 'y': 2, 'z': 4})

# or

def better_write2(self):
   # same value on all records
   self.write({'x': 1, 'y': 2, 'z': 4})
```

Chain of Browse_null

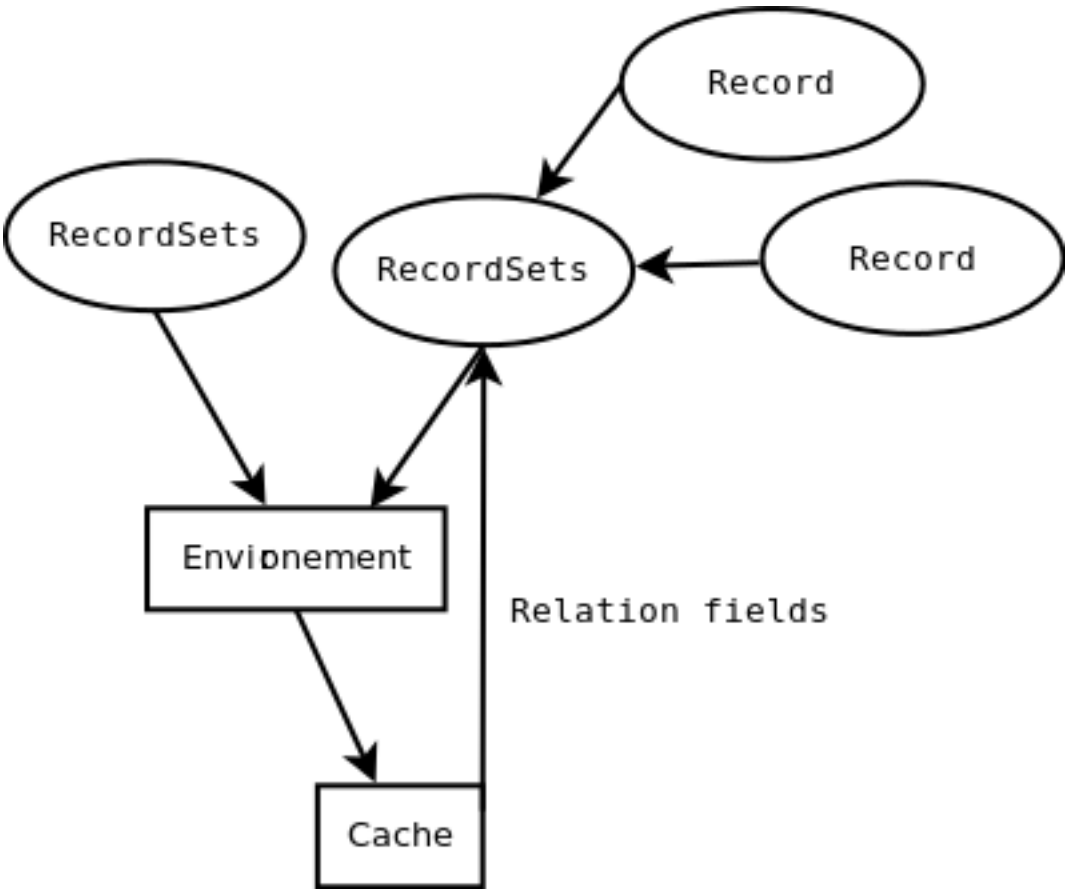
Empty relation now returns an empty RecordSet.

In the new API if you chain a relation with many empty relations, each relation will be chained and an empty RecordSet should be return at the end.

CHAPTER 2

Environment

In the new API the notion of Environment is introduced. Its main objective is to provide an encapsulation around cursor, user_id, model, and context, Recordset and caches



With this adjonction you are not anymore forced to pass the infamous function signature:

```
# before
def afun(self, cr, uid, ids, context=None):
    pass

# now
def afun(self):
    pass
```

To access the environment you may use:

```
def afun(self):
     self.env
     # or
     model.env
```

Environnement should be immutable and may not be modified in place as it also stores the caches of the RecordSet etc.

Modifing Environment

If you need to modifiy your current context you may use the with_context() function.

```
self.env['res.partner'].with_context(tz=x).create(vals)
```

Be careful not to modify current RecordSet using this functionality:

```
self = self.env['res.partner'].with_context(tz=x).browse(self.ids)
```

It will modifiy the current Records in RecordSet after a rebrowse and will generate an incoherence between caches and RecordSet.

Changing User

Environment provides an helper to switch user:

```
self.sudo(user.id)
self.sudo()    # This will use the SUPERUSER_ID by default
# or
self.env['res.partner'].sudo().create(vals)
```

Accessing Current User

```
self.env.user
```

Fetching record using XML id

```
self.env.ref('base.main_company')
```

Cleaning Environment Caches

As explained previously an Environment maintains multiple caches that are used by the Moded/Fields classes.

Sometimes you will have to do insert/write using the cursor directly. In this cases you want to invalidate the caches:

```
self.env.invalidate_all()
```

CHAPTER 3

Common Actions

Searching

Searching has not changed a lot. Sadly the domain changes announced did not meet release 8.0.

You will find main changes below.

search

Now seach function returns directly a RecordSet:

```
>>> self.search([('is_company', '=', True)])
res.partner(7, 6, 18, 12, 14, 17, 19, 8,...)
>>> self.search([('is_company', '=', True)])[0].name
'Camptocamp'
```

You can do a search using env:

```
>>> self.env['res.users'].search([('login', '=', 'admin')])
res.users(1,)
```

search_read

A `search_read` function is now available. It will do a search and return a list of dict.

Here we retrieve all partners name:

```
>>> self.search_read([], ['name'])
[{'id': 3, 'name': u'Administrator'},
 {'id': 7, 'name': u'Agrolait'},
 {'id': 43, 'name': u'Michel Fletcher'},
 ...]
```

search_count

The `search_count` function returns the count of results matching search domain:

```
>>> self.search_count([('is_company', '=', True)])
26L
```

Browsing

Browsing is the standard way to obtain Records from the database. Now browsing will return a RecordSet:

```
>>> self.browse([1, 2, 3])
res.partner(1, 2, 3)
```

More info about record *Record*

Writing

Using Active Record pattern

You can now write using Active Record pattern:

```
@api.one
def any_write(self):
  self.x = 1
  self.name = 'a'
```

More info about the subtility of the Active Record write pattern here *Record*

The classical way of writing is still available.

From Record

From Record:

```
@api.one
...
self.write({'key': value })
# or
record.write({'key': value})
```

From RecordSet

From RecordSet:

```
@api.multi
...
self.write({'key': value })
# It will write on all record.
self.line_ids.write({'key': value })
```

It will write on all Records of the relation line_ids

Many2many One2many Behavior

One2many and Many2many fields have some special behavior to be taken in account. At that time (this may change at release) using create on a multiple relation fields will not introspect to look for the relation.

```
self.line_ids.create({'name': 'Tho'})    # this will fail as order is not set
self.line_ids.create({'name': 'Tho', 'order_id': self.id})  # this will work
self.line_ids.write({'name': 'Tho'})     # this will write all related lines
```

When adding new relation records in an *@api.onchange* method, you can use the `openerp.models.BaseModel.new()` constructor. This will create a record that is not committed to the database yet, having an id of type `openerp.models.NewId`.

```
self.child_ids += self.new({'key': value})
```

Such records will be committed when the form is saved.

Copy

Note: Subject to change, still buggy !!!

From Record

From Record:

```
>>> @api.one
>>> ...
>>>     self.copy()
broken
```

From RecordSet

From RecordSet:

```
>>> @api.multi
>>> ...
>>>     self.copy()
broken
```

Create

Create has not changed, except the fact it now returns a recordset:

```
self.create({'name': 'New name'})
```

Dry run

You can do action only in caches by using the `do_in_draft` helper of Environment context manager.

CHAPTER 4

Using Cursor

Record Recordset and environment share the same cursor.

So you can access cursor using:

```
def my_fun(self):
    cursor = self._cr
    # or
    self.env.cr
```

Then you can use cursor like in previous API

CHAPTER 5

Using Thread

When using thread you have to create you own cursor and initiate a new environment for each thread. committing is done by committing the cursor:

```
with Environment.manage():  # class function
    env = Environment(cr, uid, context)
```

CHAPTER 6

New ids

When creating a record a model with computed fields, the records of the recordset will be in memory only. At that time the *id* of the record will be a dummy ids of type `openerp.models.NewId`

So if you need to use the record *id* in your code (e.g. for a sql query) you should check if it is available:

```
if isinstance(current_record.id, models.NewId):
    # do your stuff
```

CHAPTER 7

Fields

Now fields are class property:

```
from openerp import models, fields


class AModel(models.Model):

    _name = 'a_name'

    name = fields.Char(
        string="Name",                    # Optional label of the field
        compute="_compute_name_custom",   # Transform the fields in computed fields
        store=True,                       # If computed it will store the result
        select=True,                      # Force index on field
        readonly=True,                    # Field will be readonly in views
        inverse="_write_name"             # On update trigger
        required=True,                    # Mandatory field
        translate=True,                   # Translation enable
        help='blabla',                    # Help tooltip text
        company_dependent=True,           # Transform columns to ir.property
        search='_search_function'         # Custom search function mainly used with␣
↪compute
    )

   # The string key is not mandatory
   # by default it wil use the property name Capitalized

   name = fields.Char()  #  Valid definition
```

Field inheritance

One of the new features of the API is to be able to change only one attribute of the field:

```
name = fields.Char(string='New Value')
```

Field types

Boolean

Boolean type field:

```
abool = fields.Boolean()
```

Char

Store string with variable len.:

```
achar = fields.Char()
```

Specific options:

- size: data will be trimmed to specified size
- translate: field can be translated

Text

Used to store long text.:

```
atext = fields.Text()
```

Specific options:

- translate: field can be translated

HTML

Used to store HTML, provides an HTML widget.:

```
anhtml = fields.Html()
```

Specific options:

- translate: field can be translated

Integer

Store integer value. No NULL value support. If value is not set it returns 0:

```
anint = fields.Integer()
```

Float

Store float value. No NULL value support. If value is not set it returns 0.0 If digits option is set it will use numeric type:

```
afloat = fields.Float()
afloat = fields.Float(digits=(32, 32))
afloat = fields.Float(digits=lambda cr: (32, 32))
```

Specific options:

- digits: force use of numeric type on database. Parameter can be a tuple (int len, float len) or a callable that return a tuple and take a cursor as parameter

Date

Store date. The field provides some helpers:

- `context_today` returns current day date string based on tz
- `today` returns current system date string
- `from_string` returns datetime.date() from string
- `to_string` returns date string from datetime.date

:

```
>>> from openerp import fields

>>> adate = fields.Date()
>>> fields.Date.today()
'2014-06-15'
>>> fields.Date.context_today(self)
'2014-06-15'
>>> fields.Date.context_today(self, timestamp=datetime.datetime.now())
'2014-06-15'
>>> fields.Date.from_string(fields.Date.today())
datetime.datetime(2014, 6, 15, 19, 32, 17)
>>> fields.Date.to_string(datetime.datetime.today())
'2014-06-15'
```

DateTime

Store datetime. The field provide some helper:

- `context_timestamp` returns current day date string based on tz
- `now` returns current system date string
- `from_string` returns datetime.date() from string
- `to_string` returns date string from datetime.date

:

```
>>> fields.Datetime.context_timestamp(self, timestamp=datetime.datetime.now())
datetime.datetime(2014, 6, 15, 21, 26, 1, 248354, tzinfo=<DstTzInfo 'Europe/Brussels'␣
↪CEST+2:00:00 DST>)
```

```
>>> fields.Datetime.now()
'2014-06-15 19:26:13'
>>> fields.Datetime.from_string(fields.Datetime.now())
datetime.datetime(2014, 6, 15, 19, 32, 17)
>>> fields.Datetime.to_string(datetime.datetime.now())
'2014-06-15 19:26:13'
```

Binary

Store file encoded in base64 in bytea column:

```
abin = fields.Binary()
```

Selection

Store text in database but propose a selection widget. It induces no selection constraint in database. Selection must be set as a list of tuples or a callable that returns a list of tuples:

```
aselection = fields.Selection([('a', 'A')])
aselection = fields.Selection(selection=[('a', 'A')])
aselection = fields.Selection(selection='a_function_name')
```

Specific options:

- selection: a list of tuple or a callable name that take recordset as input
- size: the option size=1 is mandatory when using indexes that are integers, not strings

When extending a model, if you want to add possible values to a selection field, you may use the *selection_add* keyword argument:

```
class SomeModel(models.Model):
    _inherits = 'some.model'
    type = fields.Selection(selection_add=[('b', 'B'), ('c', 'C')])
```

Reference

Store an arbitrary reference to a model and a row:

```
aref = fields.Reference([('model_name', 'String')])
aref = fields.Reference(selection=[('model_name', 'String')])
aref = fields.Reference(selection='a_function_name')
```

Specific options:

- selection: a list of tuple or a callable name that take recordset as input

Many2one

Store a relation against a co-model:

```
arel_id = fields.Many2one('res.users')
arel_id = fields.Many2one(comodel_name='res.users')
an_other_rel_id = fields.Many2one(comodel_name='res.partner', delegate=True)
```

Specific options:

- comodel_name: name of the opposite model
- delegate: set it to `True` to make fields of the target model accessible from the current model (corresponds to `_inherits`)

One2many

Store a relation against many rows of co-model:

```
arel_ids = fields.One2many('res.users', 'rel_id')
arel_ids = fields.One2many(comodel_name='res.users', inverse_name='rel_id')
```

Specific options:

- comodel_name: name of the opposite model
- inverse_name: relational column of the opposite model

Many2many

Store a relation against many2many rows of co-model:

```
arel_ids = fields.Many2many('res.users')
arel_ids = fields.Many2many(comodel_name='res.users',
                            relation='table_name',
                            column1='col_name',
                            column2='other_col_name')
```

Specific options:

- comodel_name: name of the opposite model
- relation: relational table name
- columns1: relational table left column name
- columns2: relational table right column name

Name Conflicts

Note: fields and method name can conflict.

When you call a record as a dict it will force to look on the columns.

Fields Defaults

Default is now a keyword of a field:

You can attribute it a value or a function

```
name = fields.Char(default='A name')
# or
name = fields.Char(default=a_fun)

#...
def a_fun(self):
   return self.do_something()
```

Using a fun will force you to define function before fields definition.

Computed Fields

There is no more direct creation of fields.function.

Instead you add a `compute` kwarg. The value is the name of the function as a string or a function. This allows to have fields definition atop of class:

```
class AModel(models.Model):
    _name = 'a_name'

    computed_total = fields.Float(compute='compute_total')

    def compute_total(self):
        ...
        self.computed_total = x
```

The function can be void. It should modify record property in order to be written to the cache:

```
self.name = new_value
```

Be aware that this assignation will trigger a write into the database. If you need to do bulk change or must be careful about performance, you should do classic call to write

To provide a search function on a non stored computed field you have to add a `search` kwarg on the field. The value is the name of the function as a string or a reference to a previously defined method. The function takes the second and third member of a domain tuple and returns a domain itself

```
def search_total(self, operator, operand):
    ...
    return domain  # e.g. [('id', 'in', ids)]
```

Inverse

The inverse key allows to trigger call of the decorated function when the field is written/"created"

Multi Fields

To have one function that compute multiple values:

```
@api.multi
@api.depends('field.relation', 'an_otherfield.relation')
def _amount(self):
    for x in self:
        x.total = an_algo
        x.untaxed = an_algo
```

Related Field

There is not anymore `fields.related` fields.

Instead you just set the name argument related to your model:

```
participant_nick = fields.Char(string='Nick name',
                               related='partner_id.name')
```

The `type` kwarg is not needed anymore.

Setting the `store` kwarg will automatically store the value in database. With new API the value of the related field will be automatically updated, sweet.

```
participant_nick = fields.Char(string='Nick name',
                               store=True,
                               related='partner_id.name')
```

Note: When updating any related field not all translations of related field are translated if field is stored!!

Chained related fields modification will trigger invalidation of the cache for all elements of the chain.

Property Field

There is some use cases where value of the field must change depending of the current company.

To activate such behavior you can now use the *company_dependent* option.

A notable evolution in new API is that "property fields" are now searchable.

WIP copyable option

There is a dev running that will prevent to redefine copy by simply setting a copy option on fields:

```
copy=False  # !! WIP to prevent redefine copy
```

CHAPTER 8

Method and decorator

New decorators are just mapper around the new API. The decorator are mandatory as webclient and HTTP controller are not compliant with new API.

`api` namespace decorators will detect signature using variable name and decide to match old signature or not.

Recognized variable names are:

`cr, cursor, uid, user, user_id, id, ids, context`

@api.returns

This decorator guaranties unity of returned value. It will return a RecordSet of specified model based on original returned value:

```
@api.returns('res.partner')
def afun(self):
    ...
    return x  # a RecordSet
```

And if an old API function calls a new API function it will automatically convert it into a list of ids

All decorators inherits from this decorator to upgrade or downgrade the returned value.

@api.one

This decorator loops automatically on Records of RecordSet for you. Self is redefined as current record:

```
@api.one
def afun(self):
    self.name = 'toto'
```

Note: Caution: the returned value is put in a list. This is not always supported by the web client, e.g. on button action methods. In that case, you should use `@api.multi` to decorate your method, and probably call *self.ensure_one()* in the method definition.

@api.multi

Self will be the current RecordSet without iteration. It is the default behavior:

```
@api.multi
def afun(self):
    len(self)
```

@api.model

This decorator will convert old API calls to decorated function to new API signature. It allows to be polite when migrating code.

```
@api.model
def afun(self):
    pass
```

@api.constrains

This decorator will ensure that decorated function will be called on create, write, unlink operation. If a constraint is met the function should raise a *openerp.exceptions.Warning* with appropriate message.

@api.depends

This decorator will trigger the call to the decorated function if any of the fields specified in the decorator is altered by ORM or changed in the form:

```
@api.depends('name', 'an_other_field')
def afun(self):
    pass
```

Note: when you redefine depends you have to redefine all @api.depends, so it loses some of his interest.

View management

One of the great improvement of the new API is that the depends are automatically inserted into the form for you in a simple way. You do not have to worry about modifying views anymore.

@api.onchange

This decorator will trigger the call to the decorated function if any of the fields specified in the decorator is changed in the form:

```
@api.onchange('fieldx')
def do_stuff(self):
   if self.fieldx == x:
      self.fieldy = 'toto'
```

In previous sample *self* corresponds to the record currently edited on the form. When in on_change context all work is done in the cache. So you can alter RecordSet inside your function without being worried about altering database. That's the main difference with `@api.depends`

At function return, differences between the cache and the RecordSet will be returned to the form.

View management

One of the great improvement of the new API is that the onchange are automatically inserted into the form for you in a simple way. You do not have to worry about modifying views anymore.

Warning and Domain

To change domain or send a warning just return the usual dictionary. Be careful not to use `@api.one` in that case as it will mangle the dictionary (put it in a list, which is not supported by the web client).

@api.noguess

This decorator prevent new API decorators to alter the output of a method

CHAPTER 9

Introspection

A common pattern in OpenERP was to do Model fields introspection using `_columns` property. From 8.0 `_columns` is deprecated by `_fields` that contains list of consolidated fields instantiated using old or new API.

Conventions and code update

CHAPTER 10

Conventions

Snake_casing or CamelCasing

That was not clear. But it seems that OpenERP SA will continue to use snake case.

Imports

As discussed with Raphaël Collet. This convention should be the one to use after RC1.

Model

```
from openerp import models
```

Fields

```
from openerp import fields
```

Translation

```
from openerp import _
```

API

```
from openerp import api
```

Exceptions

```
from openerp import exceptions
```

A typical module import would be:

```
from openerp import models, fields, api, _
```

Classes

Class should be initialized like this:

```
class Toto(models.Model):
   pass


class Titi(models.TransientModel):
   pass
```

New Exceptions classes

`except_orm` exception is deprecated. We should use `openerp.exceptions.Warning` and subclasses instances

Note: Do not mix with built-in Python Warning.

RedirectWarning

Warning with a possibility to redirect the user instead of simply diplaying the warning message.

Should receive as parameters:

- **param int action_id** id of the action where to perform the redirection
- **param string button_text** text to put on the button that will trigger the redirection.

AccessDenied

Login/password error. No message, no traceback.

AccessError

Access rights error.

class MissingError:

Missing record(s)

DeferredException:

Exception object holding a traceback for asynchronous reporting.

Some RPC calls (database creation and report generation) happen with an initial request followed by multiple, polling requests. This class is used to store the possible exception occurring in the thread serving the first request, and is then sent to a polling request.

Note: Traceback is misleading, this is really a `sys.exc_info()` triplet.

Compatibility

When catching orm exception we should catch both types of exceptions:

```
try:
    pass
except (Warning, except_orm) as exc:
    pass
```

Fields

Fields should be declared using new fields API. Putting string key is better than using a long property name:

```
class AClass(models.Model):

    name = fields.Char(string="This is a really long long name")  # ok
    really_long_long_long_name = fields.Char()
```

That said the property name must be meaningful. Avoid name like 'nb' etc.

Default or compute

`compute` option should not be used as a workaround to set default. Defaut should only be used to provide property initialisation.

That said they may share the same function.

Modifing self in method

We should never alter self in a Model function. It will break the correlation with current Environment caches.

Doing thing in dry run

If you use the do_in_draft context manager of Environment it will not be committed but only be done in cache.

Using Cursor

When using cursor you should use current environment cursor:

```
self.env.cr
```

except if you need to use threads:

```
with Environment.manage():  # class function
    env = Environment(cr, uid, context)
```

Displayed Name

_name_get is deprecated.

You should define the display_name field with options:

- `compute`
- `inverse`

Constraints

Should be done using `@api.constrains` decorator in conjunction with the `@api.one` if performance allows it.

Qweb view or not Qweb view

If no advance behavior is needed on Model view, standard view (non Qweb) should be the preferred choice.

Javascript and Website related code

General guidelines should be found:

- https://doc.openerp.com/trunk/web/guidelines/
- https://doc.openerp.com/trunk/server/howto/howto_website/

CHAPTER 11

Compatibility

There is some pattern to know during the transition period to keep code base compatible with both old and new API.

Access old API

By default, using new API your are going to work on `self` that is a new RecordSet class instance. But old context and model are still available using:

```
self.pool
self._model
```

How to be polite with old code base

If your code must be used by old API code base, it should be decorated by:

- `@api.returns` to ensure adapted returned values
- `@api.model` to ensure that new signature support old API calls

CHAPTER 12

Unittest

To get access to the new API in unittest inside common.TransactionCase and others:

```
class test_partner_firstname(common.TransactionCase):

    def setUp(self):
        super(test_partner_firstname, self).setUp()
        self.user_model = self.env["res.users"]
        self.partner_model = self.env["res.partner"]
```

CHAPTER 13

YAML

To get access to the new API in Python YAML tag:

```
!python {model: account.invoice, id: account_invoice_customer0}: |
    self  # is now a new api record
    assert (self.move_id), "Move falsely created at pro-forma"
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search