Open Source RAD with OpenObject

PREAMBLE **OpenERP** is a modern Enterprise Management Software, released under the AGPL license, and featuring CRM, HR, Sales, Accounting, Manufacturing, Inventory, Project Management, ... It is based on **OpenObject**, a modular, scalable, and intuitive *Rapid Application Development (RAD)* framework written in Python.

OpenObject features a complete and modular toolbox for quickly building applications: integrated *Object-Relationship Mapping (ORM)* support, template-based *Model-View-Controller (MVC)* interfaces, a report generation system, automated internationalization, and much more.

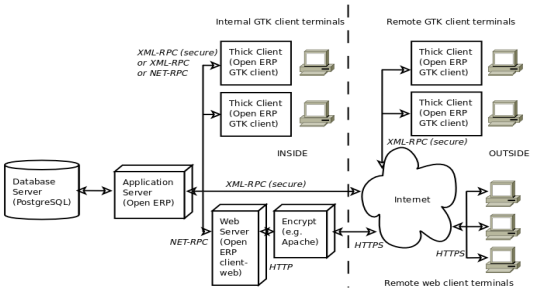
Python is a high-level dynamic programming language, ideal for *RAD,* combining power with clear syntax, and a core kept small by design.

Tip: Useful links
- Main website, with OpenERP downloads: www.openerp.com
- Functional & technical documentation: doc.openerp.com
- Community resources: www.launchpad.net/open-object
- Integration server: test,openobject.com
- Learning Python: doc.python.org
- OpenERP E-Learning platform: edu.openerp.com

Installing OpenERP

OpenERP is distributed as packages/installers for most platforms, but can of course be installed from the source on any platform.

OpenERP Architecture



OpenERP uses the well-known client-server paradigm, with different pieces of software acting as client and server depending on the desired configuration.Client software
OpenERP provides a thick desktop client (GTK+) on all platforms, and a web interface is also accessible using any modern browser.

Tip: Installation procedure
The procedure for installing OpenERP is likely to evolve (dependencies and so on), so make sure to always check the specific documentation (packaged & on website) for the latest procedures. See http://doc.openerp.com/install

Package installation

Windows	all-in-one installer, and separate installers for server, client, and webserver are on the website
Linux	*`openerp-server`* and *`openerp-client`* packages are available via corresponding package manager (e.g. Synaptic on Ubuntu)
Mac	look online for package installers for the GTK client, as well as tutorials for installing the server (e.g. devteam.taktik.be)

Installing from source

There are two alternatives: using a tarball provided on the website, or directly getting the source using Bazaar (distributed Source Version Control). You also need to install the required dependencies (PostgreSQL and a few Python libraries – see documentation on doc.openerp.com).

Compilation tip: OpenERP being Python-based, no compilation step is needed

Typical bazaar checkout procedure (on Debian-based Linux)

```
$ sudo apt-get install bzr            # install bazaar version control
$ bzr branch lp:openerp               # retrieve source installer
$ cd openerp && python ./bzr_set.py   # fetch code and perform setup
```

Database creation

After installation, run the server and the client. From the GTK client, use *File→Databases→New Database* to create a new database (default super admin password is *admin*). Each database has its own modules and config, and demo data can be included.

Building an OpenERP module: idea

CONTEXT The code samples used in this memento are taken from a hypothetical *idea* module. The purpose of this module would be to help creative minds, who often come up with ideas that cannot be pursued immediately, and are too easily forgotten if not logged somewhere. It could be used to record these ideas, sort them and rate them.

Note: Modular development
OpenObject uses modules as feature containers, to foster maintainable and robust development. Modules provide feature isolation, an appropriate level of abstraction, and obvious MVC patterns.

Composition of a module

A module may contain any of the following elements:

- **business objects**: declared as Python classes extending the osv.osv OpenObject class, the persistence of these resources is completely managed by OpenObject ;
- **data**: XML/CSV files with meta-data (views and workflows declaration), configuration data (modules parametrization) and demo data (optional but recommended for testing, e.g. sample ideas) ;
- **wizards**: stateful interactive forms used to assist users, often available as contextual actions on resources ;
- **reports**: RML (XML format), MAKO or OpenOffice report templates, to be merged with any kind of business data, and generate HTML, ODT or PDF reports.

Typical module structure

Each module is contained in its own directory within the server/bin/addons directory in the server installation.

Note: You can declare your own add-ons directory in the configuration file of OpenERP (passed to the server with the -c option) using the addons_path option.

```
addons/
 |- idea/           # The module directory
   |- demo/         # Demo and unit test population data
   |- i18n/         # Translation files
   |- report/       # Report definitions
   |- security/     # Declaration of groups and access rights
   |- view/         # Views (forms,lists), menus and actions
   |- wizard/       # Wizards definitions
   |- workflow/     # Workflow definitions
   |- __init__.py   # Python package initialization  (required)
   |- __terp__.py   # module declaration (required)
   |- idea.py       # Python classes, the module's objects
```

The __init__.py file is the Python module descriptor, because an OpenERP module is also a regular Python module.

__init__.py:

```
# Import all files & directories containing python code
import idea, wizard, report
```

The __terp__.py is the OpenERP descriptor and contains a single Python dictionary with the actual declaration of the module: its name, dependencies, description, and composition.

__terp__.py:

```
{
  'name' : 'Idea',
  'version' : '1.0',
  'author' : 'OpenERP',
  'description' : 'Ideas management module',
  'category': 'Enterprise Innovation',
  'website': 'http://www.openerp.com',
  'depends' : ['base'], # list of dependencies, conditioning startup order
  'update_xml' : [                  # data files to load at module init
    'security/groups.xml',          # always load groups first!
    'security/ir.model.access.csv', # load access rights after groups
    'workflow/workflow.xml',
    'view/views.xml',
    'wizard/wizard.xml',
    'report/report.xml',
  ],
  'demo_xml': ['demo/demo.xml'],    # demo data (for unit tests)
  'active': False,   # whether to install automatically at new DB creation
}
```

Object Service – ORM

Key component of OpenObject, the Object Service (OSV) implements a complete Object-Relational mapping layer, freeing developers from having to write basic SQL plumbing.
Business objects are declared as Python classes inheriting from the osv.osv class, which makes them part of the OpenObject Model, and magically persisted by the ORM layer.

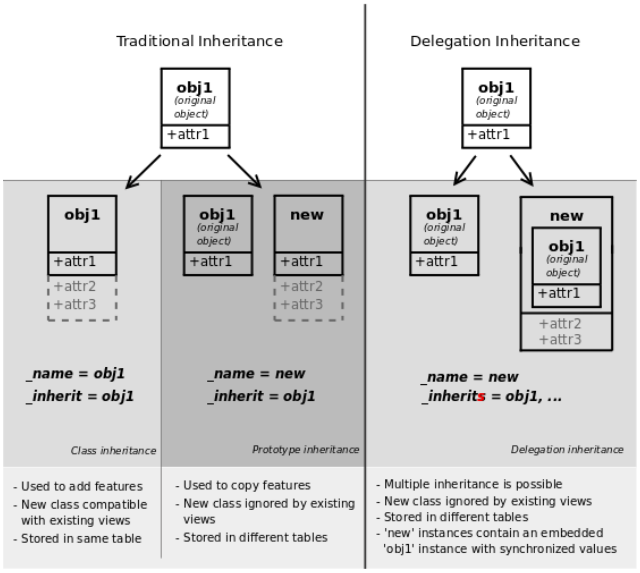
Predefined attributes are used in the Python class to specify a business object's characteristics for the ORM:

idea.py:

```
37 from osv import osv, fields
38 class idea(osv.osv)
39   _name = 'idea.idea'
40   _columns = {
41      'name': fields.char('Title', size=64, required=True, translate=True),
42      'state': fields.selection([('draft','Draft'),
43              ('confirmed','Confirmed')],'State',required=True,readonly=True),
44      # Description is read-only when not draft!
45      'description': fields.text('Description', readonly=True,
46           states={'draft': [('readonly', False)]} ),
47      'active': fields.boolean('Active'),
48      'invent_date': fields.date('Invent date'),
49      # by convention, many2one fields end with '_id'
50      'inventor_id': fields.many2one('res.partner','Inventor'),
51      'inventor_country_id': fields.related('inventor_id','country',
52                readonly=True, type='many2one',
53                relation='res.country', string='Country'),
54      # by convention, *2many fields end with '_ids'
55      'vote_ids': fields.one2many('idea.vote','idea_id','Votes'),
56      'sponsor_ids': fields.many2many('res.partner','idea_sponsor_rel',
57                                      'idea_id','sponsor_id','Sponsors'),
58      'score': fields.float('Score',digits=(2,1)),
59      'category_id' = many2one('idea.category', 'Category'),
60   }
61   _defaults = {
62      'active': lambda *a: 1,          # ideas are active by default
63      'state': lambda *a: 'draft',     # ideas are in draft state by default
64   }
65   def _check_name(self,cr,uid,ids):
66     for idea in self.browse(cr, uid, ids):
67       return 'spam' not in idea.name # Can't create ideas with spam!
68   _sql_constraints = [('name_uniq','unique(name)', 'Idea must be unique!')]
69   _constraints = [(_check_name,'Please avoid spam in ideas !', ['name'])]
70 idea() # Instantiate the class
```

Predefined osv.osv attributes for business objects	
_name (required)	business object name, in dot-notation (in module namespace)
_columns (required)	dictionary {field names → object fields declarations }
_defaults	dictionary: { field names → functions providing defaults } `_defaults['name'] = lambda cr,uid,context: 'default name'`
_auto	if *True* (default) the ORM will create the database table – set to *False* to create your own table/view within the init() method
_inherit	_name of the parent business object (for *prototype* inheritance)
_inherits	for multiple / *instance* inheritance mechanism: dictionary mapping the _name of the parent business objects to the names of the corresponding foreign key fields to use
_constraints	list of tuples defining the Python constraints, in the form (func_name, message, fields). (→68)
_sql_constraints	list of tuples defining the SQL constraints, in the form (name, sql_def, message). (→67)
_log_access	If True (default), 4 fields (create_uid, create_date, write_uid, write_date) will be used to log record-level operations, made accessible via osv's perm_read() function
_order	Name of the field used to sort the records in lists (default: 'id')
_rec_name	Alternative field to use as name, used by osv's name_get() (default: _name)
_sql	SQL code to create the table/view for this object (if _auto is False) – can be replaced by SQL execution in the init() method
_table	SQL table name to use (default: _name with dots '.' replaced by underscores '_')

Inheritance mechanisms



ORM field types

Objects may contain 3 types of fields: simple, relational, and functional. *Simple* types are integers, floats, booleans, strings, etc. *Relational* fields represent the relationships between objects (one2many, many2one, many2many). *Functional* fields are not stored in the database but calculated on-the-fly as Python functions. Relevant examples in the idea class above are indicated with the corresponding line numbers (→xx,xx)

ORM fields types	
*Common attributes supported by **all** fields (optional unless specified)*	
• string: field label **(required)** • required: *True* if mandatory • readonly: *True* if not editable • help: help tooltip • select: *1* to include in search views and optimize for list filtering (with database index)	• context: dictionary with contextual parameters (for relational fields) • change_default: True if field should be usable as condition for default values in clients • states: dynamic changes to this field's common attributes based on the state field (→42,46)
Simple fields	

boolean(...) **integer**(...) **date**(...) **datetime**(...) **time**(...)	`'active': fields.boolean('Active'),`
	`'priority': fields.integer('Priority'),`
	`'start_date': fields.date('Start Date'),`

char(string,size,translate=False,..) **text**(string, translate=False, …) *Text-based fields*	• translate: *True* if field values can be translated by users • size: maximum size for char fields (→41,45)
float(string, digits=None, ...) *Floating-point value with arbitrary precision and scale*	• digits: tuple (precision, scale) (→57) . If digits is not provided, it's a float, not a decimal type.
selection(values, string, ...) *Field allowing selection among a set of predefined values*	• values: list of values (key-label tuples) or function returning such a list **(required)** (→42)
binary(string, filters=None, ...) *Field for storing a file or binary content.*	• filters: optional filename filters `'picture': fields.binary('Picture', filters='*.png,*.gif')`

reference(string, selection, size,..)
Field with dynamic relationship to any other object, associated with an assistant widget
	• selection: model _name of allowed objects types and corresponding label (same format as values for selection fields) **(required)**
	• size: size of text column used to store it (as text: 'model_name,object_id') **(required)**
	`'contact': fields.reference('Contact',[`
	`('res.partner','Partner'),`
	`('res.partner.contact','Contact')], 128)`

Relational fields	
*Common attributes supported by **relational** fields*	• domain: optional restriction in the form of arguments for search (see search())
many2one(obj, ondelete='set null', …) (→50) *Relationship towards a parent object (using a foreign key)*	• obj: _name of destination object **(required)** • ondelete: deletion handling, e.g. 'set null', 'cascade', see PostgreSQL documentation
one2many(obj, field_id, …) (→55) *Virtual relationship towards multiple objects (inverse of many2one)*	• obj: _name of destination object **(required)** • field_id: field name of inverse many2one, i.e. corresponding foreign key **(required)**
many2many(obj, rel, field1, field2, …) (→56) *Bidirectional multiple relationship between objects*	• obj: _name of destination object **(required)** • rel: relationship table to use **(required)** • field1: name of field in rel table storing the id of the current object **(required)** • field2: name of field in rel table storing the id of the target object **(required)**

ORM fields types
Functional fields
function(fnct, arg=None, fnct_inv=None, fnct_inv_arg=None, type='float', fnct_search=None, obj=None, method=False, store=False, multi=False,…) *Functional field simulating a real field, computed rather than stored* • fnct: function to compute the field value **(required)** def fnct(self, cr, uid, ids, field_name, arg, context) returns a dictionary { ids→values } with values of type type • fnct_inv: function used to write a value in the field instead def fnct_inv(obj, cr, uid, id, name, value, fnct_inv_arg, context) • type: type of simulated field (any other type besides 'function') • fnct_search: function used to search on this field def fnct_search(obj, cr, uid, obj, name, args) returns a list of tuples arguments for search(), e.g. [('id','in',[1,3,5])] • obj: model _name of simulated field if it is a relational field • store, multi: optimization mechanisms (see usage in Performance Section)
related(f1, f2, …, type='float', …) *Shortcut field equivalent to browsing chained fields* • f1,f2,...: chained fields to reach target **(f1 required)** (→51) • type: type of target field
property(obj, type='float', view_load=None, group_name=None, …) *Dynamic attribute with specific access rights* • obj: object **(required)** • type: type of equivalent field

Tip: relational fields symmetry

- one2many ↔ many2one are symmetric
- many2many ↔ many2many are symmetric when inversed (swap field1 and field2)
- one2many ↔ many2one + many2one ↔ one2many = many2many

Special fields

The following fields have pre-defined semantics in OpenObject

name	unique name used by default for labels in forms, lists, etc. if missing, use _rec_name to specify another field to use
sequence	defines order, and allows drag & drop reordering in lists
active	archive flag for record deletion, records are hidden if *False*
parent_id	defines tree structure on records, and enables child_of operator in search criteria

Working with the ORM

Inheriting from the osv.osv class makes all the ORM methods available on business objects. These methods may be invoked on the self object within the Python class itself (see examples in the table below), or from outside the class by first obtaining an instance via the ORM's pooler system.

ORM usage sample

```
71 class idea2(osv.osv):
72   _name = 'idea.idea2'
73   _inherit = 'idea.idea'
74   def _score_calc(self,cr,uid,ids,field,arg,context):
75      res = {}
76      # This loop generates only 2 queries thanks to browse()!
77      for idea in self.browse(cr,uid,ids):
78          sum_vote = sum([v.vote for v in idea.vote_ids])
79          avg_vote = sum_vote/len(idea.vote_ids)
80          res.update(idea.id, avg_vote)
81      return res
82   _columns = {
83      # Replace static score with average of votes
84      'score':fields.function(_score_calc,type='float',method=True)
85   }
86 idea2()
```

ORM Methods on osv.osv objects	
OSV generic accessor	• self.pool('object_name') may be used to obtain a model class from anywhere
Common parameters, used by multiple methods	• cr: database connection (cursor) • uid: id of user performing the operation • ids: list of record ids, or single integer when there is only one id • context: optional dictionary of contextual parameters, such as user language e.g. { 'lang': 'en_US', ... }
create(cr, uid, values, context=None)	• values: dictionary of field values for the record

ORM Methods on osv.osv objects

Creates a new record with the specified value
Returns: id of the new record

```
idea_id = self.create(cr, uid,
     { 'name': 'Spam recipe',
       'description' : 'spam & eggs',
       'inventor_id': 45,
     })
```

search(cr, uid, args, offset=0, limit=None, order=None, context=None, count=False)

Returns: list of ids of records matching the given criteria

- args: list of tuples specifying search criteria
- offset: optional number of records to skip
- limit: optional max number of records to return
- order: optional columns to sort by (default: self._order)
- count: if *True*, returns only the number of records matching the criteria, not their ids

```
#Operators: =, !=, >, >=, <, <=, like, ilike,
#in, not in, child_of, parent_left, parent_right
#Prefix operators: '&' (default), '|', '!'
#Fetch non-spam partner shops + partner 34
ids = self.search(cr, uid,
       [ '|', ('partner_id', '!=', 34),
         '!', ('name', 'ilike', 'spam'), ],
                 order='partner_id' )
```

read(cr, user, ids, fields=None, context=None)

Returns: list of dictionaries with requested field values

- fields: optional list of field names to return (default: all fields)

```
results = self.read(cr, uid, [42,43],
                    ['name', 'inventor_id'])
print 'Inventor:', results[0]['inventor_id']
```

write(cr, uid, ids, values, context=None)

Updates records with given ids with the given values.
Returns: True

- values: dictionary of field values to update

```
self.write(cr, uid,
                  { 'name': 'spam & eggs',
                    'partner_id': 24,
                  })
```

copy(cr, uid, id, defaults,context=None)

Duplicates record with given id updating it with defaults *values.*
Returns: True

- defaults: dictionary of field values to update before saving the duplicate object

unlink(cr, uid, ids, context=None)

Deletes records with the given ids
Returns: True

```
self.unlink(cr, uid, [42,43])
```

browse(cr, uid, ids, context=None)

Fetches records as objects, allowing to use dot-notation to browse fields and relations
Returns: object or list of objects requested

```
idea = self.browse(cr, uid, 42)
print 'Idea description:', idea.description
print 'Inventor country code:',
 idea.inventor_id.address[0].country_id.code
for vote in idea.vote_ids:
  print 'Vote %2.2f' % vote.vote
```

default_get(cr, uid, fields, context=None)

Returns: a dictionary of the default values for fields (set on the object class, by the user preferences, or via the context)

- fields: list of field names

```
defs = self.default_get(cr,uid,
                        ['name','active'])
# active should be True by default
assert defs['active']
```

perm_read(cr, uid, ids, details=True)

Returns: a list of ownership dictionaries for each requested record

- details: if *True*, *_uid fields are replaced with the name of the user
- returned dictionaries contain: object id (id), creator user id (create_uid), creation date (create_date), updater user id (write_uid), update date (write_date)

```
perms = self.perm_read(cr,uid,[42,43])
print 'creator:', perms[0].get('create_uid', 'n/a')
```

fields_get(cr, uid, fields=None, context=None)

Returns a dictionary of field dictionaries, each one describing a field of the business object

- fields: list of field names

```
class idea(osv.osv):
  (...)
  _columns = {
    'name' : fields.char('Name',size=64)
    (...)
  def test_fields_get(self,cr,uid):
    assert(self.fields_get('name')['size'] == 64)
```

fields_view_get(cr, uid, view_id=None, view_type='form', context=None, toolbar=False)

Returns a dictionary describing the composition of the requested view (including inherited views and extensions)

- view_id: id of the view or None
- view_type: type of view to return if view_id is None ('form','tree', ...)
- toolbar: *True* to include contextual actions

```
def test_fields_view_get(self,cr,uid):
 idea_obj = self.pool.get('idea.idea')
 form_view = idea_obj.fields_view_get(cr,uid)
```

ORM Methods on osv.osv objects	

name_get(cr, uid, ids, context={})
Returns tuples with the text representation of requested objects for to-many relationships
	`# Ideas should be shown with invention date`
	`def name_get(self,cr,uid,ids):`
	`  res = []`
	`  for r in self.read(cr,uid,ids['name','create_date'])`
	`  res.append((r['id'], '%s (%s)' (r['name'],year))`
	`  return res`

name_search(cr, uid, name='', args=None, operator='ilike', context=None, limit=80)
 	
Returns list of object names matching the criteria, used to provide completion for to-many relationships. Equivalent of search() *on* name + name_get()
	• name: object name to search for
	• operator: operator for name criterion
	• args, limit: same as for search())
	
	`# Countries can be searched by code or name`
	`def name_search(self,cr,uid,name='',`
	`  args=[],operator='ilike',context={},`
	`  limit=80):`
	`  ids = []`
	`  if name and len(name) == 2:`
	`  ids = self.search(cr, user,`
	`  [('code', '=', name)] + args,`
	`  limit=limit, context=context)`
	`  if not ids:`
	`  ids = self.search(cr, user,`
	`  [('name', operator, name)] + args,`
	`  limit=limit, context=context)`
	`  return self.name_get(cr,uid,ids)`

export_data(cr, uid, ids, fields, context=None) *Exports* fields *for selected objects, returning a dictionary with a* datas *matrix. Used when exporting data via client menu.*	• fields: list of field names • context may contain import_comp (default: *False*) to make exported data compatible with import_data() (may prevent exporting some fields)
import_data(cr, uid, fields, data, mode='init', current_module='', noupdate=False, context=None, filename=None) *Imports given data in the given module Used when exporting data via client menu*	• fields: list of field names • data: data to import (see export_data()) • mode: 'init' or 'update' for record creation • current_module: module name • noupdate: flag for record creation • filename: optional file to store partial import state for recovery

Tip: use read() through webservice calls, but always browse() internally

Building the module interface

To construct a module, the main mechanism is to insert data records declaring the module interface components. Each module element is a regular data record: menus, views, actions, roles, access rights, etc.

Common XML structure

XML files declared in a module's update_xml attribute contain record declarations in the following form:

```
<?xml version="1.0" encoding="utf-8"?>
<openerp>
  <data>
    <record model="object_model_name" id="object_xml_id">
      <field name="field1">value1</field>
      <field name="field2">value2</field>
    </record>
    <record model="object_model_name2" id="object_xml_id2">
      <field name="field1" ref="module.object_xml_id"/>
      <field name="field2" eval="ref('module.object_xml_id')"/>
    </record>
  </data>
</openerp>
```

Each type of record (view, menu, action) support a specific set of child entities and attributes, but all share the following special attributes:

id	the unique (per module) XML identifier of this record (xml_id)
ref	used instead of element content to reference another record (works cross-module by prepending the module name)
eval	used instead of element content to provide value as a Python expression, that can use the ref() method to find the database id for a given xml_id

Tip: XML RelaxNG validation

OpenObject validates the syntax and structure of XML files, according to a RelaxNG grammar, found in server/bin/import_xml.rng.
For manual check use xmllint: xmllint –relax-ng /path/to/import_xml.rng <file>

Common CSV syntax

CSV files can also be added in update_xml, and the records will be inserted by the OSV's import_data() method, using

the CSV filename to determine the target object model. The ORM automatically reconnects relationships based on the following special column names:

id (xml_id)	column containing identifiers for relationships
many2one_field	reconnect many2one using name_search()
many2one_field:id	reconnect many2one based on object's xml_id
many2one_field.id	reconnect many2one based on object's database id
many2many_field	reconnects via name_search(), repeat for multiple values
many2many_field:id	reconnects with object's xml_id, repeat for multiple values
many2many_field.id	reconnects with object's database id, repeat for multiple values
one2many_field/field	creates one2many destination record and sets field value

ir.model.access.csv

```
"id","name","model_id:id","group_id:id","perm_read","perm_write","perm_create","perm_unlink"
"access_idea_idea","idea.idea","model_idea_idea","base.group_user",1,0,0,0
"access_idea_vote","idea.vote","model_idea_vote","base.group_user",1,0,0,0
```

Menus and actions

Actions are declared as regular records and can be triggered in 3 ways:

- by clicking on menu items linked to a specific action
- by clicking on buttons in views, if these are connected to actions
- as contextual actions on an object

Action declaration

```
<record model="ir.actions.act_window" id="action_id">
    <field name="name">action.name</field>
    <field name="view_id" ref="view_id"/>
    <field name="domain">[list of 3-tuples (max 250 characters)]</field>
    <field name="context">{context dictionary (max 250 characters)}</field>
    <field name="res_model">object.model.name</field>
    <field name="view_type">form|tree</field>
    <field name="view_mode">form,tree,calendar,graph</field>
    <field name="target">new</field>
    <field name="search_view_id" ref="search_view_id"/>
</record>
```

id	identifier of the action in table ir.actions.act_window, must be unique
name	action name **(required)**
view_id	specific view to open (if missing, highest priority view of given type is used)
domain	tuple (see search() arguments) for filtering the content of the view
context	context dictionary to pass to the view
res_model	object model on which the view to open is defined
view_type	set to *form* to open records in edit mode, set to *tree* for a tree view only
view_mode	if *view_type* is *form*, list allowed modes for viewing records (*form*, *tree*, ...)
target	set to *new* to open the view in a new window
search_view_id	identifier of the search view to replace default search form (*new in version 5.2*)

Menu declaration

The menuitem entity is a shortcut for declaring an ir.ui.menu record and connect it with a corresponding action via an ir.model.data record.

```
<menuitem id="menu_id" parent="parent_menu_id"
   name="label" action="action_id" icon="icon-code"
   groups="groupname1,groupname2" sequence="10"/>
```

id	identifier of the menuitem, must be unique
parent	id of the parent menu in the hierarchy
name	Optional menu label (default: action name)
action	identifier of action to execute, if any
icon	icon to use for this menu (e.g. *terp-graph*, *STOCK_OPEN*, see doc.opernerp.com)
groups	list of groups that can see this menu item (if missing, all groups can see it)
sequence	integer index for ordering sibling menuitems (10,20,30..)

Views and inheritance

Views form a hierarchy. Several views of the same type can be declared on the same object, and will be used depending on their priorities. By declaring an inherited view it is possible to add/remove features in a view.

Generic view declaration

```
<record model="ir.ui.view" id="view_id">
   <field name="name">view.name</field>
   <field name="model">object_name</field>
   <field name="type">form</field>  # tree,form,calendar,search,graph,gantt
   <field name="priority" eval="16"/>
      <field name="arch" type="xml">
          <!-- view content: <form>, <tree>, <graph>, … -->
```

```
        </field>
</record>
```

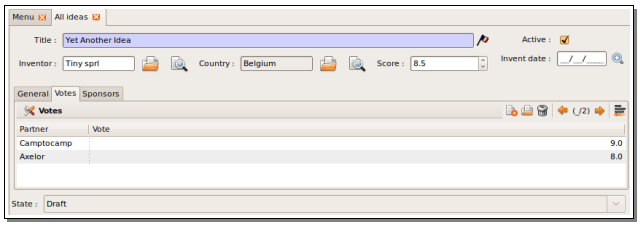
id	unique view identifier
name	view name
model	object model on which the view is defined (same as *res_model* in actions)
type	view type: *form*, *tree*, *graph*, *calendar*, *search*, *gantt* (*search* is new in 5.2)
priority	view priority, smaller is higher (default: 16)
arch	architecture of the view, see various view types below

Forms (to view/edit records)

Forms allow creation/edition or resources, and correspond to <form> elements.

Allowed elements	*all (see form elements below)*

```
<form string="Idea form">
   <group col="6" colspan="4">
      <group colspan="5" col="6">
          <field name="name" select="1" colspan="6"/>
          <field name="inventor_id" select="1"/>
          <field name="inventor_country_id" />
          <field name="score" select="2"/>
      </group>
      <group colspan="1" col="2">
          <field name="active"/><field name="invent_date"/>
      </group>
   </group>
   <notebook colspan="4">
      <page string="General">
          <separator string="Description"/>
          <field colspan="4" name="description" nolabel="1"/>
      </page>
      <page string="Votes">
          <field colspan="4" name="vote_ids" nolabel="1" select="1">
            <tree>
              <field name="partner_id"/>
              <field name="vote"/>
            </tree>
          </field>
      </page>
      <page string="Sponsors">
          <field colspan="4" name="sponsor_ids" nolabel="1" select="1"/>
      </page>
   </notebook>
   <field name="state"/>
   <button name="do_confirm" string="Confirm" icon="gtk-ok" type="object"/>
</form>
```



Form Elements

Common attributes for all elements:

- string: label of the element
- nolabel: 1 to hide the field label
- colspan: number of column on which the field must span
- rowspan: number of rows on which the field must span
- col: number of column this element must allocate to its child elements
- invisible: *1* to hide this element completely
- eval: evaluate this Python code as element content (content is string by default)
- attrs: Python map defining dynamic conditions on these attributes: readonly, invisible, required based on search tuples on other field values

field	automatic widgets depending on the corresponding field type. Attributes: • string: label of the field, also for search (overrides field name) • select: *1* to show the field in normal search, *2* for advanced only • nolabel: 1 to hide the field label • required: override required field attribute • readonly: override readonly field attribute • password: *True* to hide characters typed in this field • context: Python code declaring a context dictionary • domain: Python code declaring list of tuples for restricting values • on_change: Python method call to trigger when value is changed • groups: comma-separated list of group (id) allowed to see this field • widget: select alternative widget (*one2many_list*, *many2many*, *url*, *email*, *image*, *float_time*, *reference*, *text_wiki*, *text_html*, *progressbar*)
properties	dynamic widget showing all available properties (no attribute)
button	clickable widget associated with actions. Specific attributes: • type: type of button: *workflow* (default), *object*, or *action* • name: workflow signal, function name (without parentheses) or action to call (depending on type) • confirm: text of confirmation message when clicked • states: comma-separated list of states in which this button is shown • icon: optional icon (all GTK STOCK icons e.g. *gtk-ok*)
separator	horizontal separator line for structuring views, with optional label
newline	place-holder for completing the current line of the view
label	free-text caption or legend in the form
group	used to organise fields in groups with optional label (adds frame)
notebook, *page*	*notebook* elements are tab containers for *page* elements. Attributes: • name: label for the tab/page • position: tabs position in notebook (*inside*, *top*, *bottom*, *left*, *right*)

Dynamic views

In addition to what can be done with states and attrs attributes, functions may be called by view elements (via buttons of type object, or on_change attributes on fields) to obtain dynamic behavior. These functions may alter the view interface by returning a Python map with the following entries:

value	a dictionary of field names and their updated values
domain	a dictionary of field names and their updated domains of value
warning	a dictionary with a *title* and *message* to show a warning dialog

Lists/Trees

Lists include *field* elements, are created with type *tree*, and have a <tree> parent element.

Attributes	• colors: list of colors mapped to Python conditions • editable: *top* or *bottom* to allow in-place edit • toolbar: set to *True* to display the top level of object hierarchies as a side toolbar (example: the menu)
Allowed elements	*field, group, separator, tree, button, filter, newline*

```
158 <tree string="Idea Categories" toolbar="1" colors="blue:state==draft">
159   <field name="name"/>
160   <field name="description"/>
161 </tree>
```

Calendars

Views used to display date fields as calendar events (<calendar> parent)

Attributes	• color: name of field for color segmentation • date_start: name of field containing event start date/time • day_length: length of a calendar day in hours (default: • date_stop: name of field containing event stop date/time or • date_delay: name of field containing event duration
Allowed elements	*field (to define the label for each calendar event)*

```
162 <calendar string="Ideas" date_start="invent_date" color="inventor_id">
163       <field name="name"/>
164 </calendar>
```

Gantt Charts

Bar chart typically used to show project schedule (<gantt> parent element)

Attributes	same as <calendar>
Allowed elements	*field, level* • level elements are used to define the Gantt chart levels, with the enclosed field used as label for that drill-down level

```
165 <gantt string="Ideas" date_start="invent_date" color="inventor_id">
166   <level object="idea.idea" link="id" domain="[]">
167     <field name="inventor_id"/>
168   </level>
169 </gantt>
```

Charts (Graphs)

Views used to display statistical charts (<graph> parent element)

Tip: charts are most useful with custom views extracting ready-to-use statistics

Attributes	• type: type of chart: *bar*, *pie* (default) • orientation: *horizontal*, *vertical*
Allowed elements	*field,* with specific behavior: • first field in view is X axis, 2nd one is Y, 3rd one is Z • 2 fields required, 3rd one is optional • group attribute defines the GROUP BY field (set to 1) • operator attribute sets the aggregation operator to use for other fields when one field is grouped (+,*,**,min,max)

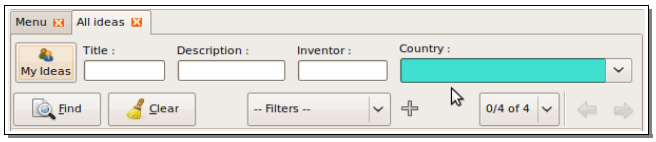
```
<graph string="Total idea score by Inventor" type="bar">
   <field name="inventor_id" />
   <field name="score" operator="+"/>
</graph>
```

Search views (new in 5.2)

Search views are used to customize the search panel on top of list views, and are declared with the *search* type, and a top-level <search> element. After defining a search view with a unique id, add it to the action opening the list view using the search_view_id field in its declaration.

Allowed elements	*field, group, separator, label, search, filter, newline, properties* • filter elements allow defining button for domain filters • adding a context attribute to fields makes widgets that alter the search context (useful for context-sensitive fields, e.g. pricelist-dependent prices)

```
<search string="Search Ideas">
  <group col="6" colspan="4">
    <filter string="My Ideas" icon="terp-partner"
        domain="[('inventor_id','=',uid)]"
        help="My own ideas"/>
    <field name="name" select="1"/>
    <field name="description" select="1"/>
    <field name="inventor_id" select="1"/>
    <!-- following context field is for illustration only -->
    <field name="inventor_country_id" select="1" widget="selection"
                    context="{'inventor_contry': self}"/>
   </group>
</search>
```



View Inheritance

Existing views should be modifying through inherited views, never directly. An inherited view references its parent view using the inherit_id field, and may add or modify existing elements in the view by referencing them through XPath expressions, specifying the appropriate position.

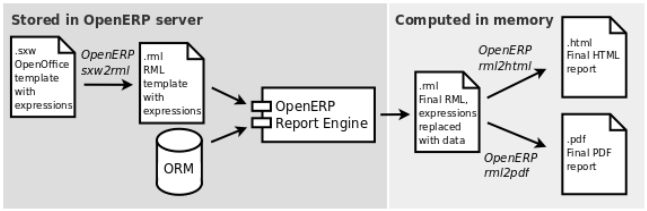
Tip: XPath reference can be found at www.w3.org/TR/xpath

position	• *inside*: placed inside match (default) • *replace*: replace match	• *before*: placed before match • *after*: placed after match

```
<!-- improved idea categories list -->
<record id="idea_category_list2" model="ir.ui.view">
  <field name="name">id.category.list2</field>
  <field name="model">ir.ui.view</field>
  <field name="inherit_id" ref="id_category_list"/>
  <field name="arch" type="xml">
    <xpath expr="/tree/field[@name='description']" position="after">
      <field name="idea_ids" string="Number of ideas"/>
    </xpath>
  </field>
</record>
```

Reports

There are several report engines in OpenERP, to produce reports from different sources and in many formats.



Special expressions used inside report templates produce dynamic data and/or modify the report structure at rendering time.
Custom report parsers may be written to support additional expressions.

Alternative Report Formats (see doc.openerp.com)

sxw2rml	OpenOffice 1.0 templates (.sxw) converted to RML with sxw2rml tool, and the RML rendered in HTML or PDF
rml	RML templates rendered directly as HTML or PDF
xml,xsl:rml	XML data + XSL:RML stylesheets to generate RML
odt2odt	OpenOffice templates (.odt) used to produce directly OpenOffice documents (.odt) (As of OpenERP 5.2)
mako	Mako template library used to produce HTML output, by embedding Python code and OpenERP expressions within any text file (As of OpenERP 5.2)

Expressions used in OpenERP report templates	
[[<content>]]	double brackets content is evaluated as a Python expression based on the following expressions

Predefined expressions:
- objects contains the list of records to print
- data comes from the wizard launching the report
- user contains the current user (as per browse())
- time gives access to Python *time* module
- repeatIn(list,'var','tag') repeats the current parent element named tag for each object in list, making the object available as var during each loop
- setTag('tag1','tag2') replaces the parent RML tag1 with tag2
- removeParentNode('tag') removes parent RML element tag
- formatLang(value, digits=2, date=False, date_time=False, grouping=True, monetary=False) can be used to format a date, time or amount according to the locale
- setLang('lang_code') sets the current language and locale for translations

Report declaration

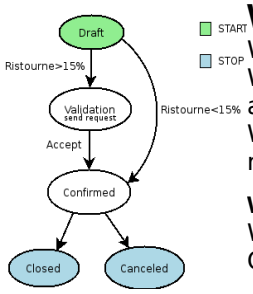
```
198 <!-- The following creates records in ir.actions.report.xml model -->
199 <report id="idea_report" string="Print Ideas" model="idea.idea"
200       name="idea.report" rml="idea/report/idea.rml" >
201 <!-- Use addons/base_report_designer/wizard/tiny_sxw2rml/tiny_sxw2rml.py
202       to generate the RML template file from a .sxw template -->
```

id	unique report identifier
name	name for the report **(required)**
string	report title **(required)**
model	object model on which the report is defined **(required)**
rml, sxw, xml, xsl	path to report template sources (starting from addons), depending on report
auto	set to *False* to use a custom parser, by subclassing report_sxw.rml_parse and declaring the report as follows: report_sxw.report_sxw(report_name, object_model,rml_path,parser=customClass)
header	set to *False* to suppress report header (default: *True*)
groups	comma-separated list of groups allowed to view this report
menu	set to *True* to link the report with the Print icon (default: *True*)
keywords	specify report type keyword (default: *client_print_multi*)

Tip: RML User Guide: www.reportlab.com/docs/rml2pdf-userguide.pdf

Example RML report extract:

```
203 <story>
204   <blockTable style="Table">
205     <tr>
206       <td><para style="Title">Idea name</para> </td>
207       <td><para style="Title">Score</para> </td>
208     </tr>
209     <tr>
210       <td><para>[[ repeatIn(objects,'o','tr') ]] [[ o.name ]]</para></td>
211       <td><para>[[ o.score ]]</para></td>
212     </tr>
213   </blockTable>
214 </story>
```



Workflows

Workflows may be associated with any object in OpenERP, and are entirely customizable.
Workflows are used to structure and manage the lifecycles of business objects and documents, and define transitions, triggers, etc. with graphical tools.
Workflows, activities (nodes or actions) and transitions (conditions) are declared as XML records, as usual. The tokens that navigate in workflows are called *workitems*.

Workflow declaration

Workflows are declared on objects that possess a state field (see the example idea class in the ORM section)

```
<record id="wkf_idea" model="workflow">
    <field name="name">idea.basic</field>
    <field name="osv">idea.idea</field>
    <field name="on_create" eval="1"/>
</record>
```

id	unique workflow record identifier
name	name for the workflow **(required)**
osv	object model on which the workflow is defined **(required)**
on_create	if *True*, a workitem is instantiated automatically for each new *osv* record

Workflow Activities (nodes)

```
<record id="act_confirmed" model="workflow.activity">
    <field name="name">confirmed</field>
    <field name="wkf_id" ref="wkf_idea"/>
    <field name="kind">function</field>
    <field name="action">action_confirmed()</field>
</record>
```

id	unique activity identifier
wkf_id	parent workflow identifier
name	activity node label
flow_start	*True* to make it a 'begin' node, receiving a workitem for each workflow instance
flow_stop	*True* to make it an 'end' node, terminating the workflow when all items reach it
join_mode	logical behavior of this node regarding incoming transitions: • *XOR*: activate on the first incoming transition (default) • *AND*: waits for all incoming transitions to become valid
split_mode	logical behavior of this node regarding outgoing transitions: • *XOR*: one valid transition necessary, send workitem on it (default) • *OR*: send workitems on all valid transitions (0 or more), sequentially • *AND*: send a workitem on all valid transitions at once (fork)
kind	type of action to perform when node is activated by a transition: • *dummy* to perform no operation when activated (default) • *function* to invoke a function determined by *action* • *subflow* to execute the subflow with *subflow_id*, invoking *action* to determine the record id of the record for which the subflow should be instantiated. If action returns no result, the workitem is deleted. • *stopall* to terminate the workflow upon activation
subflow_id	if kind *subflow*, id of the subflow to execute (use *ref* attribute or *search* with a tuple)
action	object method call, used if kind is *function* or *subflow*. This function should also update the *state* field of the object, e.g. for a *function* kind:

```
def action_confirmed(self, cr, uid, ids):
    self.write(cr, uid, ids, { 'state' : 'confirmed' })
    # … perform other tasks
    return True
```

Workflow Transitions (edges)

Conditions are evaluated in this order: role_id, signal, condition expression

```
<record id="trans_idea_draft_confirmed" model="workflow.transition">
    <field name="act_from" ref="act_draft"/>
    <field name="act_to" ref="act_confirmed"/>
    <field name="signal">button_confirm</field>
    <field name="role_id" ref="idea_manager"/>
    <field name="condition">1 == 1</field>
</record>
```

act_from, *act_to*	identifiers of the source and destination activities
signal	name of a button of type workflow that triggers this transition
role_id	reference to the role that user must have to trigger the transition (see *Roles*)
condition	Python expression that must evaluate to *True* for transition to be triggered

Tip: The Web client features a graphical workflow editor, via the *Customise→Manage Workflows* link at the bottom left in lists and forms.

Security

Access control mechanisms must be combined to achieve a coherent security policy.

Group-based access control mechanisms

Groups are created as normal records on the res.groups model, and granted menu access via **menu** definitions. However even without a menu, objects may still be accessible indirectly, so actual **object-level permissions** (*create,read,write,unlink*) must be defined for groups. They are usually inserted via CSV files inside modules. It is also possible to restrict access to **specific fields** on a view or object using the field's groups attribute.

ir.model.access.csv

```
"id","name","model_id:id","group_id:id","perm_read","perm_write","perm_create","perm_unlink"
"access_idea_idea","idea.idea","model_idea_idea","base.group_user",1,1,1,0
"access_idea_vote","idea.vote","model_idea_vote","base.group_user",1,1,1,0
```

Roles

Roles are created as normal records on the res.roles model and used only to condition workflow transitions through transitions' role_id attribute.

Wizards

Wizards describe stateful interactive sessions with the user through dynamic forms. As of OpenERP v5.0, wizards make use of the *osv_memory* in-memory persistence to allow constructing wizards from regular business objects and views.

Wizard objects (osv_memory)

In-memory objects are created by extending osv.osv_memory:

```
from osv import fields,osv
import datetime
class cleanup_wizard(osv.osv_memory):
  _name = 'idea.cleanup.wizard'
  _columns = {
    'idea_age': fields.integer('Age (in days)'),
  }
  def cleanup(self,cr,uid,ids,context={}):
    idea_obj = self.pool.get('idea.idea')
    for wiz in self.browse(cr,uid,ids):
      if wiz.idea_age <= 3:
        raise osv.except_osv('UserError','Please select a larger age')
      limit = datetime.date.today()-datetime.timedelta(days=wiz.idea_age)
      ids_to_del = idea_obj.search(cr,uid, [('create_date', '<' ,
              limit.strftime('%Y-%m-%d 00:00:00'))],context=context)
      idea_obj.unlink(cr,uid,ids_to_del)
    return {}
cleanup_wizard()
```

Views

Wizards use regular views and their buttons may use a special cancel attribute to close the wizard window when clicked.

```
<record id="wizard_idea_cleanup" model="ir.ui.view">
  <field name="name">idea.cleanup.wizard.form</field>
  <field name="model">idea.cleanup.wizard</field>
  <field name="type">form</field>
  <field name="arch" type="xml">
    <form string="Idea Cleanup Wizard">
      <label colspan="4" string="Select the age of ideas to cleanup"/>
      <field name="idea_age" string="Age (days)"/>
      <group colspan="4">
       <button string="Cancel" special="cancel" icon="gtk-cancel"/>
       <button string="Cleanup" name="cleanup" type="object" icon="gtk-ok"/>
      </group>
    </form>
  </field>
</record>
```

Wizard execution

Such wizards are launched via regular action records, with a special target field used to open the wizard view in a new window.

```
<record id="action_idea_cleanup_wizard" model="ir.actions.act_window">
  <field name="name">Cleanup</field>
  <field name="type">ir.actions.act_window</field>
  <field name="res_model">idea.cleanup.wizard</field>
  <field name="view_type">form</field>
  <field name="view_mode">form</field>
  <field name="target">new</field>
</record>
```

WebServices – XML-RPC

OpenERP is accessible through XML-RPC interfaces, for which libraries exist in many languages.

Python example

```
import xmlrpclib
```

```
# ... define HOST, PORT, DB, USER, PASS
url = 'http://%s:%d/xmlrpc/common' % (HOST,PORT)
sock = xmlrpclib.ServerProxy(url)
uid = sock.login(DB,USER,PASS)
print "Logged in as %s (uid:%d)" % (USER,uid)

# Create a new idea
url = 'http://%s:%d/xmlrpc/object' % (HOST,PORT)
sock = xmlrpclib.ServerProxy(url)
args = {
    'name' : 'Another idea',
    'description' : 'This is another idea of mine',
    'inventor_id': uid,
}
idea_id = sock.execute(DB,uid,PASS,'idea.idea','create',args)
```

PHP example

```
<?
include('xmlrpc.inc'); // Use phpxmlrpc library, available on sourceforge
// ... define $HOST, $PORT, $DB, $USER, $PASS
$client = new xmlrpc_client("http://$HOST:$PORT/xmlrpc/common");
$msg = new xmlrpcmsg("login");
$msg->addParam(new xmlrpcval($DB, "string"));
$msg->addParam(new xmlrpcval($USER, "string"));
$msg->addParam(new xmlrpcval($PASS, "string"));
resp = $client->send($msg);
uid = $resp->value()->scalarval()
echo "Logged in as $USER (uid:$uid)"

// Create a new idea
$arrayVal = array(
  'name'=>new xmlrpcval("Another Idea", "string") ,
  'description'=>new xmlrpcval("This is another idea of mine" , "string"),
  'inventor_id'=>new xmlrpcval($uid, "int"),
);
$msg = new xmlrpcmsg('execute');
$msg->addParam(new xmlrpcval($DB, "string"));
$msg->addParam(new xmlrpcval($uid, "int"));
$msg->addParam(new xmlrpcval($PASS, "string"));
$msg->addParam(new xmlrpcval("idea.idea", "string"));
$msg->addParam(new xmlrpcval("create", "string"));
$msg->addParam(new xmlrpcval($arrayVal, "struct"));
$resp = $client->send($msg);
?>
```

Internationalization

Each module can provide its own translations within the i18n directory, by having files named LANG.po where LANG is the locale code for the country and language combination (e.g. fr_FR.po). Translations will be loaded automatically by OpenERP for all enabled languages.
Developers always use English when creating a module, then export the module terms using OpenERP's gettext *POT* export feature (Administration>Translations>Export a Translation File without specifying a language) , to create the module template POT file, and then derive the translated PO files.
Many IDE's have plugins or modes for editing and merging PO/POT files.

Tip: The GNU `gettext` format (Portable Object) used by OpenERP is integrated into LaunchPad, making it an online collaborative translation platform, with automatic translation features.

```
|- idea/           # The module directory
  |- i18n/         # Translation files
    | - idea.pot   # Translation Template (exported from OpenERP)
    | - fr_FR.po   # French translation
    | - es_ES.po   # Spanish translation
    | (...)
```

Tip: By default OpenERP's POT export only extracts labels inside XML files or inside field definitions in Python code, but any Python string can be translated this way by surrounding it with the `tools.translate._` method (e.g. `_('Label')`)

Rapid Application Development

Module recorder
The base_module_record module can be used to export a set of changes in the form of a new module. It should be used for all customizations that should be carried on through migrations and updates. It has 2 modes:

• Start/Pause/Stop mode, where all operations (on business objects or user interface) are recorded until the recorder is stopped or paused.
• Date- and model-based mode where all changes performed after a given date on the given models (object types) are exported. .

Report Creator (view) and Report Designer (print) modules

The base_report_creator module can be used to automate the creation of custom statistics views, e.g. to construct dashboards. The resulting dashboards can then be exported using the base_module_record module.
The base_report_designer module can be used in conjunction with the OpenOffice plugin to provide a user-friendly interface for selecting data from OpenERP and designing report templates within OpenOffice.

Quality assessment module

When writing you module, use the base_module_quality module to test various aspects of your module: coding standards, code duplication, code efficiency, etc. (web client only). Make sure to provide a lot of demo data.

Unit tests

Unit test files are regular OpenERP XML files, with regular record elements plus an appropriate combination of function, workflow and assert elements to test the module's business logic.
The continuous integration server will automatically execute unit tests and provide feedback. Unit tests can also be used as installation checks if you reference the XML file in the update_xml section of your module descriptor.

idea_unit_test.xml

```
<record id="idea_test_1" model="idea.idea">
    <field name="name">Unit Test Idea</field>
    <field name="description">A sample idea for performing tests</field>
    <field name="invent_date">20100101</field>
</record>
<assert id="idea_test_1" model="idea.idea" severity="warning"
      string="New idea is not draft!">
    <test expr="state">draft</field>
</assert>
<workflow ref="idea_test_1" model="idea.idea" action="button_confirm"
       uid="base.user_admin"/>
<assert id="idea_test_1" model="idea.idea" severity="warning"
            string="Confirm button does not work!">
    <test expr="state == 'confirmed'"/>
</assert>
<function model="idea.idea" name="unlink">
    <value eval="ref('idea_test_1')"/>
</function>
<assert search="[('name','=','Unit Test Idea']" model="idea.idea" count="0"
   severity="warning" string="Test data is not deleted (name is unique!)"/>
```

Common attributes:
• model: target object model name
• id: *xml_id* of the record to test (assert) or to move in workflow (workflow)
• uid: optional id of user to perform operation (function or workflow)

assert — Perform test(s) and fail with given string if tests do not pass.
• string: error message in case of test failure
• severity: error severity in case of test failure (debug,info,error,warning,critical)
• search: domain of search to perform if id is not provided (each record is tested)
• count: if search is provided number of expected records (failure if not verified)
• <test> children with expr Python expression that must evaluate to True or to the text content of the element. It can use any field of the object, Python built-ins and the ref() method that returns the database id for a given *xml_id*.

function — Call method on the given model, passing the value children as arguments.
• name: name of method to call
• <value> children with Python expressions, that can use the ref() method

workflow — Send a workflow signal on a given object
• ref: *xml_id* of object to send workflow signal to
• action: name of workflow signal to send

Recurrent jobs

The ir.cron model is used to setup recurrent tasks.

```
<record id="task_id" model="ir.cron">
    <field name="name">Task title</field>
    <field name="user_id" ref="module.user_xml_id">
    <field name="interval_type">minutes|hours|days|work_days|weeks|months</field>
    <field name="interval_number" eval="<number>"/>
    <field name="numbercall" eval="<number,negative for unlimited>"/>
    <field name="doall" eval="True|False"/> <!-- Repeat missed calls? -->
    <field name="model">model.name</field>
    <field name="function">name_of_model_function_to_call</field>
    <field name="args" eval="python code for arguments tuple"/>
    <field name="priority" eval="<integer,smaller is higher>"/>
</record>
```

Performance Optimization

As Enterprise Management Software typically has to deal with large amounts of records, you may want to pay attention to the following *anti-patterns*, to obtain consistent performance:

- Do not place browse() calls inside loops, put them before and access only the browsed objects inside the loop. The ORM will optimize the number of database queries based on the *browsed* attributes.
- Avoid recursion on object hierarchies (objects with a parent_id relationship), by adding parent_left and parent_right integer fields on your object, and setting _parent_store to True in your object class. The ORM will use a *modified preorder tree traversal* to be able to perform recursive operations (e.g. child_of) with database queries in *O(1)* instead of *O(n)*
- Do not use function fields lightly, especially if you include them in tree views. To optimize function fields, two mechanisms are available:
  - multi: all fields sharing the same multi attribute value will be computed with one single call to the function, which should then return a dictionary of values in its values map
  - store: function fields with a store attribute will be stored in the database, and recomputed on demand when the relevant trigger objects are modified. The format for the trigger specification is as follows: `store = {'model': (_ref_fnct, fields, priority)}` (see example below)

```
358   def _get_idea_from_vote(self,cr,uid,ids,context={}):
359     res = {}
360     vote_ids = self.pool.get('idea.vote').browse(cr,uid,ids,context=context)
361     for v in vote_ids:
362       res[v.idea_id.id] = True  # Store the idea identifiers in a set
363     return res.keys()
364   def _compute(self,cr,uid,ids,field_name,arg,context={}):
365     res = {}
366     for idea in self.browse(cr,uid,ids,context=context):
367       vote_num = len(idea.vote_ids)
368       vote_sum = sum([v.vote for v in idea.vote_ids])
369       res[idea.id] = {
370          'vote_sum': vote_sum,
371          'vote_avg': (vote_sum/vote_num) if vote_num else 0.0,
372       }
373     return res
374   _columns = {
375    # These fields are recomputed whenever one of the votes changes
376    'vote_avg': fields.function(_compute, method=True, string='Votes Average',
377      store = {'idea.vote': (_get_idea_from_vote,['vote'],10)},multi='votes'),
378    'vote_sum': fields.function(_compute, method=True, string='Votes Sum',
379      store = {'idea.vote': (_get_idea_from_vote,['vote'],10)},multi='votes'),
380   }
```

Community / Contributing

OpenERP projects are hosted on LaunchPad(LP), where all project resources may be found: Bazaar branches, bug tracking, blueprints, roadmap, FAQs, etc. Create a free account on launchpad.net to be able to contribute.

Launchpad groups

*Group**	*Members*	*Bazaar/LP restrictions*
OpenERP Quality Team (~openerp)	OpenERP Core Team	Can merge and commit on official branches.
OpenERP Commiters (~openerp-commiter)	Selected active community members	Can mark branches to be merged into official branch. Can commit on *extra-addons* branch
OpenERP Drivers (~openerp-drivers)	Selected active community members	Can confirm bugs and set milestones on bugs / blueprints
OpenERP Community (~openerp-community)	Open group, anyone can join	Can create community branches where everyone can contribute

***Members of upper groups are also members of lower groups**

License

Published by Open Object Press, Grand Rosière, Belgium