Open Source RAD with OpenERP 7.0

PREAMBLE **OpenERP** is a modern Suite of Business Applications, released under the AGPL license, and featuring CRM, HR, Sales, Accounting, Manufacturing, Warehouse Management, Project Management, and more.
It is based on a modular, scalable, and intuitive *Rapid Application Development (RAD)* framework written in Python.

OpenERP 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.

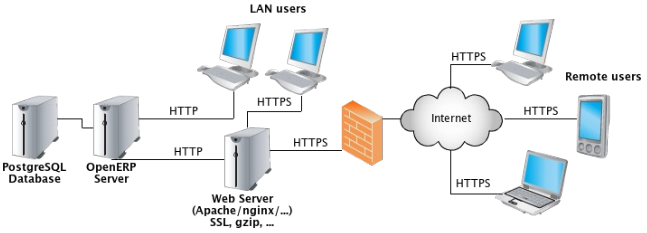
Python is a high-level dynamic programming language, ideal for *RAD,* combining power with clear syntax, and a core kept small by design.

Tip: Useful links
- Main website, with OpenERP downloads: www.openerp.com
- Functional & technical documentation: doc.openerp.com
- Community resources: www.openerp.com/community
- Continous Integration server: runbot.openerp.com
- Learning Python: doc.python.org

Installing OpenERP

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

OpenERP Architecture



OpenERP uses the well-known client-server paradigm: the client is running as a Javascript application in your browser, connecting to the server using the JSON-RPC protocol over HTTP(S). Ad-hoc clients can also be easily written and connect to the server using XML-RPC or JSON-RPC.

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/v7.0/install

Package installation

Windows	all-in-one installer
Linux	all-in-one packages available for Debian-based (.deb) and RedHat-based (.rpm) distributions
Mac	no all-in-one installer, needs to be installed from source

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 software)
$ bzr cat -d lp:~openerp-dev/openerp-tools/trunk setup.sh | sh # Get Installer
$ make init-v70                    # Install OpenERP 7.0
$ make server                      # Start OpenERP Server with embedded Web
```

Database creation

After starting the server, open http://localhost:8069 in your favorite browser. You will see the Database Manager screen where you can create a new database. Each database has its own modules and config, and can be created in demo mode to test a pre-populated database (do not use demo mode for a real database!)

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
OpenERP 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.Model class, the persistence of these resources is completely managed by OpenERP ;
- **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 addons 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)
   |- __openerp__.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 __openerp__.py is the OpenERP module manifest and contains a single Python dictionary with the declaration of the module: its name, dependencies, description, and composition.

__openerp__.py:

```
19 {
20   'name' : 'Idea',
21   'version' : '1.0',
22   'author' : 'OpenERP',
23   'description' : 'Ideas management module',
24   'category': 'Enterprise Innovation',
25   'website': 'http://www.openerp.com',
26   'depends' : ['base'], # list of dependencies, conditioning startup order
27   'data' : [                            # data files to load at module install
28     'security/groups.xml',              # always load groups first!
29     'security/ir.model.access.csv',     # load access rights after groups
30     'workflow/workflow.xml',
31     'view/views.xml',
32     'wizard/wizard.xml',
33     'report/report.xml',
34   ],
35   'demo': ['demo/demo.xml'],     # demo data (for unit tests)
36 }
```

Object-Relational Mapping Service – ORM

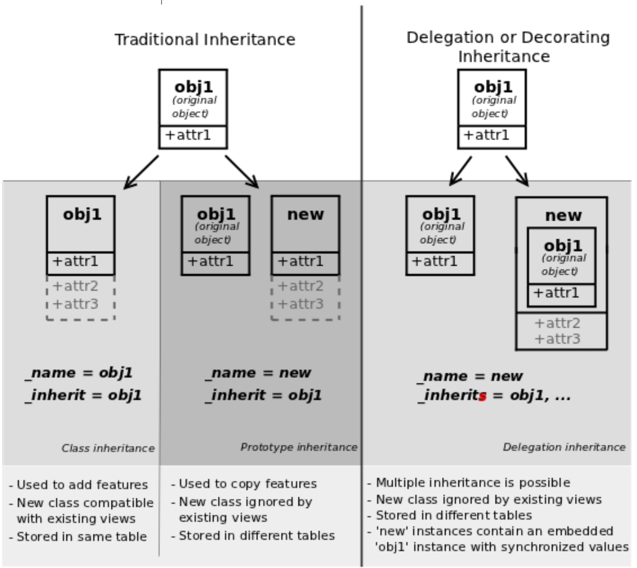
Key component of OpenERP, the ORM is a complete Object-Relational mapping layer, freeing developers from having to write basic SQL plumbing. Business objects are declared as Python classes inheriting from the osv.Model class, which makes them magically persisted by the ORM layer.

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.Model):
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': True,          # ideas are active by default
63     'state': '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         if 'spam' in idea.name: return False # Can't create ideas with spam!
68     return True
69   _sql_constraints = [('name_uniq','unique(name)', 'Ideas must be unique!')]
70   _constraints = [(_check_name, 'Please avoid spam in ideas !', ['name'])]
71
```

Predefined osv.osv attributes for business objects	
_name (required)	business object name, in dot-notation (in module namespace)
_columns (required)	dictionary {field name→field declaration }
_defaults	dictionary: {field name→literal or function providing default} `_defaults['name'] = lambda self,cr,uid,context: 'eggs'`
_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 inheritance)
_inherits	for *decoration* inheritance: dictionary mapping the _name of the parent business object(s) 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) (→70)
_sql_constraints	list of tuples defining the SQL constraints, in the form (name, sql_def, message) (→69)
_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 the 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 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 '_')



Traditional Inheritance
Delegation or Decorating Inheritance
obj1 (original object) +attr1
obj1 +attr1 +attr2 +attr3
obj1 (original object) +attr1
new +attr1 +attr2 +attr3
obj1 (original object) +attr1
new obj1 (original object) +attr1 +attr2 +attr3
_name = obj1
_inherit = obj1
_name = new
_inherit = obj1
_name = new
_inherits = obj1, ...
Class inheritance
Prototype inheritance
Delegation inheritance
- Used to add features
- New class compatible with existing views
- Stored in same table
- Used to copy features
- New class ignored by existing views
- Stored in different tables
- Multiple inheritance is possible
- New class ignored by existing views
- Stored in different tables
- 'new' instances contain an embedded 'obj1' instance with synchronized values

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)*

ORM fields types

• string: field label **(required)** • required: *True* if mandatory • readonly: *True* if not editable • help: help tooltip • select: *True* to create a database index on this column	• 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, for char/text fields • size: optional max size for char fields (→41,45)
float(string, digits=None, ...) *Decimal value*	• digits: tuple (precision, scale) (→58)
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 for selection
	`'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 (storage format is 'model_name,object_id')
	`'contact': fields.reference('Contact',[`
	`('res.partner','Partner'),`
	`('res.partner.contact','Contact')])`

Relational fields	
*Common attributes supported by **relational** fields*	• domain: optional filter 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: optional name of relationship table to use (default: auto-assigned based on model names) • field1: name of field in rel table storing the id of the current object (default: based on model) • field2: name of field in rel table storing the id of the target object (default: based on model)
Functional fields	

ORM fields types

function(fnct, arg=None, fnct_inv=None, fnct_inv_arg=None, type='float', fnct_search=None, obj=None, 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 (can be any other type except '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 if explicit)
- one2many ↔ many2one + many2one ↔ one2many = many2many

Special / Reserved field names

A few field names are reserved for pre-defined behavior in OpenERP. Some of them are created automatically by the system, and in that case any field wih that name will be ignored.

id	unique system identifier for the object
name	field whose value is used to display the record in lists, etc. if missing, set _rec_name to specify another field to use
active	toggle visibility: records with active set to *False* are hidden by default
sequence	defines order and allows drag&drop reordering if visible in list views
state	lifecycle stages for the object, used by the states attribute
parent_id	defines tree structure on records, and enables child_of operator
parent_left, parent_right	used in conjunction with _parent_store flag on object, allows faster access to tree structures (see also *Performance Optimization* section)
create_date, create_uid, write_date, write_uid	used to log creator, last updater, date of creation and last update date of the record. disabled if _log_access flag is set to *False* (created by ORM, do not add them)

Working with the ORM

Inheriting from the osv.Model 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 pool system.

ORM usage sample

```python
class idea2(osv.Model):
  _inherit = 'idea.idea'
  def _score_calc(self,cr,uid,ids,field,arg,context=None):
    res = {}
    # This loop generates only 2 queries thanks to browse()!
    for idea in self.browse(cr,uid,ids,context=context):
        sum_vote = sum([v.vote for v in idea.vote_ids])
```

```
      avg_vote = sum_vote/len(idea.vote_ids)
      res[idea.id] = avg_vote
   return res
_columns = {
  # Replace static score with average of votes
  'score':fields.function(_score_calc,type='float')
}
```

ORM Methods on osv.Model objects

OSV generic accessor

- self.pool.get('object_name') may be used to obtain a model from any other

Common parameters, used by multiple methods

- cr: database connection (cursor)
- uid: id of user performing the operation
- ids: record ids to perform the operation on
- context: optional dictionary of contextual parameters, e.g. `{ 'lang': 'en_US', ... }`

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

Returns: list of ids of records matching the given criteria

- domain: filter 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' )
```

create(cr, uid, values, context=None)

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

- values: dictionary of field values

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

read(cr, uid, 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']
```

read_group(cr, uid, domain, fields, groupby, offset=0, limit=None, orderby=None, context=None)

Returns: list of dictionaries with requested field values, grouped by given groupby field(s).

- domain: search filter (see search())
- fields: list of field names to read
- groupby: field or list of fields to group by
- offset, limit: see search()
- orderby: optional ordering for the results

```
> print self.read_group(cr,uid,[],
             ['score'], ['inventor_id'])
[{'inventor_id': (1, 'Administrator'),
  'score': 23,   # aggregated score
  'inventor_id_count': 12, # group count
 },
 {'inventor_id': (3, 'Demo'),
  'score': 13,
  'inventor_id_count': 7,
 }]
```

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, [42,43],
                 { '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 modify in the copied values when creating the duplicated object

ORM Methods on osv.Model objects

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 values are replaced with pairs (id, name_of_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)

- 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 also return context 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)
```

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

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='', domain=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
- domain, limit: same as for search())

```
# Countries can be searched by code or name
def name_search(self,cr,uid,name='',
      domain=[],operator='ilike',
      context=None,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)
```

ORM Methods on osv.Model objects	
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 prefer 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 data section 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) supports a specific set of child entities and attributes, but all share the following special attributes:

id	the unique (per module) external identifier of this record (xml_id)
ref	may be used instead of normal 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
OpenERP 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 –relaxng /path/to/import_xml.rng <file>

Common CSV syntax

CSV files can also be added in the data section 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	reconnect via name_search(), multiple values w/ commas
many2many_field:id	reconnect w/ object's xml_id, multiple values w/ commas
many2many_field.id	reconnect w/ object's database id, multiple values w/ commas
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 (visible in the side bar)

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 hierarchy 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/popup
search_view_id	identifier of the search view to replace default search form (*new in version 6.0*)

Menu declaration

The menuitem element 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" groups="groupname1,groupname2" sequence="10"/>
```

id	identifier of the menuitem, must be unique
parent	external ID (xml_id) of the parent menu in the hierarchy
name	optional menu label (default: action name)
action	identifier of action to execute, if any
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>
   <!-- types: tree,form,calendar,search,graph,gantt,kanban -->
   <field name="type">form</field>
   <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, kanban*
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" colspan="6"/>
          <field name="inventor_id"/>
          <field name="inventor_country_id" />
          <field name="score"/>
      </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">
            <tree>
              <field name="partner_id"/>
              <field name="vote"/>
            </tree>
          </field>
      </page>
      <page string="Sponsors">
          <field colspan="4" name="sponsor_ids" nolabel="1"/>
      </page>
   </notebook>
   <field name="state"/>
   <button name="do_confirm" string="Confirm" type="object"/>
</form>
```

New: the v7.0 form API

As of OpenERP 7.0 a new form view API has been introduced. It can be turned on by adding `version="7.0"` to the `<form>` element. This new form API allows mixing arbitrary XHTML code with regular OpenERP form elements. It also introduces a few specific elements to produce better-looking forms, such as `<sheet>`, `<header>`, `<footer>`, and a set of general purpose CSS classes to customize the appearance and behavior of form elements. Best practices and examples for the new form API are available in the technical documentation:
http://doc.openerp.com/trunk/developers/server/form-view-guidelines

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 for this particular view
- nolabel: *1* to hide the field label
- required: override required field attribute from Model for this view
- readonly: override readonly field attribute from Model for this view
- 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 to call when field value is changed
- groups: comma-separated list of group (id) allowed to see this field
- widget: select alternative field widget (*url, email, image, float_time, reference, html, progressbar, statusbar, handle, etc.*)

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

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 triggers 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 new 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 and Hierarchical Tree Lists

List views include *field* elements, are created with type *tree*, and have a <tree> parent element. They are used to define flat lists (editable or not) and hierarchical lists.

Attributes	• colors: list of colors or HTML color codes 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 (only for hierarchical lists, i.e. opened with actions that set the `view_type` to `"tree"` instead of `"mode"`)
Allowed elements	*field, group, separator, tree, button, filter, newline*

```
<tree string="Idea Categories" toolbar="1" colors="blue:state==draft">
  <field name="name"/>
  <field name="state"/>
</tree>
```

Kanban Boards

As of OpenERP 6.1 a new type versatile board view, in which each record is rendered as a small "kanban card". It supports drag&drop to manage the lifecycle of the kanban cards based on configurable dimensions.
Kanban views are introduced in the OpenERP 6.1 release notes and defined using the QWeb templating language, documented in the technical documentation: see http://bit.ly/18usDXt and http://doc.openerp.com/trunk/developers/web/qweb

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 [working] day in hours (default: 8) • 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)*

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

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

Charts (Graphs)

Views used to display statistics 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

Search views customize the search panel on top of other views.

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"
        domain="[('inventor_id','=',uid)]"
        help="My own ideas"/>
    <field name="name"/>
    <field name="description"/>
    <field name="inventor_id"/>
    <!-- following context field is for illustration only -->
    <field name="inventor_country_id" widget="selection"
                         context="{'inventor_country': 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, and specifying the appropriate position.

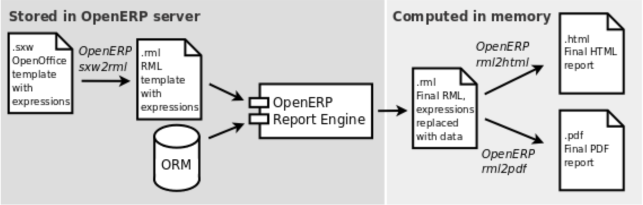
Tip: XPath reference can be found at www.w3.org/TR/xpath

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

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 (browse_record, as returned 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

```
<!-- The following creates records in ir.actions.report.xml model -->
<report id="idea_report" string="Print Ideas" model="idea.idea"
  name="idea.report" rml="idea/report/idea.rml" >
<!-- Use addons/base_report_designer/wizard/tiny_sxw2rml/tiny_sxw2rml.py
  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 display this report in the Print menu (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:

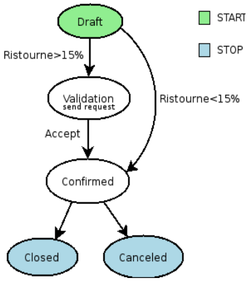
```
<story>
  <blockTable style="Table">
    <tr>
      <td><para style="Title">Idea name</para> </td>
      <td><para style="Title">Score</para> </td>
    </tr>
    <tr>
      <td><para>[[ repeatIn(objects,'o','tr') ]] [[ o.name ]]</para></td>
      <td><para>[[ o.score ]]</para></td>
    </tr>
  </blockTable>
</story>
```

Workflows

Workflows may be associated with any object in OpenERP, and are entirely customizable.

Workflows are used to structure and manage the life-cycles 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: OpenERP features a graphical workflow editor, available by switching to the Diagram view while viewing a workflow in the Settings>Technical>Workflows

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. They are constructed based on the `osv.TransientModel` class and automatically garbage-collected after use. They're defined using the same API and views as regular `osv.Model` objects.

Wizard models (TransientModel)

```
from osv import fields,osv
```

```
import datetime
class cleanup_wizard(osv.TransientModel):
  _name = 'idea.cleanup.wizard'
  _columns = {
    'idea_age': fields.integer('Age (in days)'),
  }
  def cleanup(self,cr,uid,ids,context=None):
    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 {}
```

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"/>
       <button string="Cleanup" name="cleanup" type="object"/>
      </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"),
);
```

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)

```
def _get_idea_from_vote(self,cr,uid,ids,context=None):
  res = {}
  vote_ids = self.pool.get('idea.vote').browse(cr,uid,ids,context=context)
  for v in vote_ids:
    res[v.idea_id.id] = True  # Store the idea identifiers in a set
  return res.keys()
def _compute(self,cr,uid,ids,field_name,arg,context=None):
  res = {}
  for idea in self.browse(cr,uid,ids,context=context):
    vote_num = len(idea.vote_ids)
    vote_sum = sum([v.vote for v in idea.vote_ids])
    res[idea.id] = {
       'vote_sum': vote_sum,
       'vote_avg': (vote_sum/vote_num) if vote_num else 0.0,
    }
  return res
_columns = {
 # These fields are recomputed whenever one of the votes changes
 'vote_avg': fields.function(_compute, string='Votes Average',
   store = {'idea.vote': (_get_idea_from_vote,['vote'],10)},multi='votes'),
 'vote_sum': fields.function(_compute, string='Votes Sum',
   store = {'idea.vote': (_get_idea_from_vote,['vote'],10)},multi='votes'),
}
```

Community / Contributing

OpenERP projects are hosted on Launchpad(LP), where all project resources may be found: Bazaar branches, bug tracking, blueprints, 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 Drivers (~openerp-drivers)	Selected active community members	Can confirm bugs and set milestones on bugs
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

While every precaution has been taken in the preparation of this work, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Published by Open Object Press, Grand Rosière, Belgium