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
- Continuous 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.

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 package creation 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 -o option) using the addons_path option.

```
addons/
├── idea/
│   ├── demo/
│   │   ├── l10n/
│   │   └── report/
│   │       ├── security/
│   │       │   ├── view/
│   │       │   │   ├── security/
│   │       │   │   ├── wizard/
│   │       │   │   └── workflow/
│   │       │   ├── __init__.py
│   │       │       # Python package initialization (required)
│   │       │   └── openerp/__init__.py
│   │   └── __init__.py
│   │       # Python classes, the module's objects
│   └── __init__.py
│       # The module directory
│       # Demo and unit test population data
│       # Translation files
│       # Declaration of groups and access rights
│       # Views (forms,lists), menus and actions
│       # Wizards definitions
│       # Workflow definitions
└── __openerp__.py
    # Python module descriptor, because an OpenERP module is also a regular Python module.
```

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

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.
Predefined `osv.osv` attributes for business objects

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_name</code></td>
<td>(required)</td>
</tr>
<tr>
<td><code>_columns</code></td>
<td>(required)</td>
</tr>
<tr>
<td><code>_defaults</code></td>
<td></td>
</tr>
<tr>
<td><code>auto</code></td>
<td></td>
</tr>
<tr>
<td><code>inherit</code></td>
<td></td>
</tr>
<tr>
<td><code>inherits</code></td>
<td></td>
</tr>
</tbody>
</table>

**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)

**Inheritance mechanisms**

```python
from osv import osv, fields

class idea(osv.Model):
    _name = 'idea.idea'
    _columns = [
        # 'name': fields.char('Title', size=64, required=True, translate=True),
        # 'state': fields.selection([('draft', 'Draft'), ('confirmed', 'Confirmed')], 'State', required=True, readonly=True),
        # # Description is read-only when not draft!
        # 'description': fields.text('Description', readonly=True),
        # states=('draft': [('readonly', False)]),
        # 'active': fields.boolean('Active'),
        # 'invent_date': fields.date('Invent date'),
        # # by convention, many2one fields end with '_id'
        # 'inventor_id': fields.many2one('res.partner', 'Inventor'),
        # 'inventor_country_id': fields.many2one('res.country', 'Country'),
        # # by convention, 'many2many' fields end with '_ids'
        # 'vote_ids': fields.one2many('idea.vote', 'idea_id', 'Votes'),
        # 'sponsor_ids': fields.many2many('res.partner', 'idea_sponsor_rel', 'idea_id', 'sponsor_id', 'Sponsors'),
        # 'score': fields.float('Score', digits=(2, 1)),
        # 'category_id': fields.many2one('idea.category', 'Category'),
    ]

    # defaults = {
    #     'active': True,  # ideas are active by default
    #     'state': 'draft',  # ideas are in draft state by default
    # }
    # def _check_name(self, cr, uid, ids):
    #     if idea.name in self.browse(cr, uid, ids):
    #         return False  # Can't create ideas with spam!
    #     return True
    # _sql_constraints = [('name_uniq', 'unique(name)', 'Ideas must be unique!')]
    # _constraints = []
```

Copyright © 2013 Open Object Press - All rights reserved – See license on page 12.
### ORM fields types

<table>
<thead>
<tr>
<th>ORM fields types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple fields</td>
</tr>
<tr>
<td>boolean(...) integer(...) date(...) datetime(...) time(...)</td>
</tr>
<tr>
<td>char(string.size,translate=False,...) text(string, translate=False,...)</td>
</tr>
<tr>
<td>float(string, digits=None, ...)</td>
</tr>
<tr>
<td>selection(values, string, ...)</td>
</tr>
<tr>
<td>binary(string, filters=None, ...)</td>
</tr>
<tr>
<td>reference(string, selection, size,...)</td>
</tr>
<tr>
<td>Common attributes supported by relational fields</td>
</tr>
<tr>
<td>• domain: optional filter in the form of arguments for search (see search())</td>
</tr>
<tr>
<td>many2one(obj, ondelete='set null', ...) (→50) Relationship towards a parent object (using a foreign key)</td>
</tr>
</tbody>
</table>
| • 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 |
| • 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) |
| • 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 |
| • function: function 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, arg) 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) |

#### ORM fields types

##### 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 with that name will be ignored.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>unique system identifier for the object</td>
</tr>
<tr>
<td>name</td>
<td>field whose value is used to display the record in lists, etc. if missing, set _rec_name to specify another field to use</td>
</tr>
<tr>
<td>active</td>
<td>toggle visibility: records with active set to False are hidden by default</td>
</tr>
<tr>
<td>sequence</td>
<td>defines order and allows drag&amp;drop reordering if visible in list views</td>
</tr>
<tr>
<td>state</td>
<td>lifecycle stages for the object, used by the states attribute</td>
</tr>
<tr>
<td>parent_id</td>
<td>defines tree structure on records, and enables child_of operator</td>
</tr>
<tr>
<td>parent_left, parent_right</td>
<td>used in conjunction with parent_store flag on object, allows faster access to tree structures (see also Performance Optimization section)</td>
</tr>
<tr>
<td>create_date, create_uid, write_date, write_uid</td>
<td>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)</td>
</tr>
</tbody>
</table>

### 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
72 class idea2(osv.Model):
73     inherit = 'idea.idea'
74
75     def _score_calc(self, cr, uid, ids, field, arg, context=None):
76         res = {}  
77         # This loop generates only 2 queries thanks to browse()!
78         # for idea in self.browse(cr, uid, ids, context):  
79         #    res[idea.id] = sum((v.vote for v in idea.vote_ids))  
710     sum_vote = sum((v.vote for v in idea.vote_ids))
```
### ORM Methods on `osv.Model` objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OSV generic accessor</strong></td>
<td><code>self.pool.get('object_name')</code> may be used to obtain a model from any other</td>
</tr>
</tbody>
</table>
| **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) | **Returns:** list of ids of records matching the given criteria  
**domain**: filter specifying search criteria  
**offset**: optional number of records to skip  
**order**: optional max number of records to return  
**context**: optional columns to sort by (default: self.order)  
**count**: if True, returns only the number of records matching the criteria, not their ids |  |
| **create** (cr, uid, values, context=None) | **Returns:** id of the new record  
**values**: dictionary of field values |  |
| **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) |  |
| **read_group** (cr, uid, domain, fields, groupby=None, 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**, **orderby**: see `search()` |  |
| **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 |  |
| **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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| **unlink** (cr, uid, ids, context=None) | Deletes records with the given ids  
**Returns:** True |  |
| **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 |  |
| **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 |  |
| **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)`  
**returns dictionaries contain:** object id (id), creator user id (`create_uid`), creation date (`create_date`), updater user id (`write_uid`), update date (`write_date`) |  |
| **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 |  |
| **fields_view_get** (cr, uid, view_id=None, view_type=None, 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 |  |
| **name_get** (cr, uid, ids, context=None) | **Returns:** tuples with the text representation of requested objects for to-many relationships  
**name**: object name to search for  
**operator**: operator for name criterion  
**domain**: list of name as same for search()  
**ids**: if ids is an integer, return the name of all the objects defined by the same name  
**context**: context to use to find the name |  |

---

### ORM Methods on `osv.Model` objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| **avg_vote** | `sum_vote/len(idea.vote_ids)`  
**res[idea.id] = avg_vote** |  |
| **columns** | `avg_vote = sum_vote/len(idea.vote_ids)`  
`return res`  
`_columns = {`  
# Replace static score with average of votes  
'score': fields.function(_score_calc,type='float') |  |
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.

**import_data**(cr, uid, fields, data, **mode='init', current_module=None**, **noupdate=False, context=None**, **file_name=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 |
| **noupdate** | flag for record creation |
| **file_name** | 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">value1</field>
      <field name="field2">value2</field>
    </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** | 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 |

Tables

### 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>
  <field name="domain">[]</field>
  <field name="context">context</field>
  <field name="res_model">object.model.name</field>
  <field name="view_type">type</field>
  <field name="search_view_id">ref("search_view_id")</field>
</record>
```

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.module.data** record.

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

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">model.object.name</field>
  <field name="type">type</field>
  <field name="priority">eval=16</field>
  <field name="arch">arch</field>
</record>
```
**Forms (to view/edit records)**

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

| Allowed elements | all (see form elements below) |

**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:


### 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

### 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.)

### 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 place-holder for completing the current line of the view

### newline

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:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>value</strong></td>
<td>a dictionary of field names and their new values</td>
</tr>
<tr>
<td><strong>domain</strong></td>
<td>a dictionary of field names and their updated domains of value</td>
</tr>
<tr>
<td><strong>warning</strong></td>
<td>a dictionary with a <code>title</code> and <code>message</code> to show a warning dialog</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>field</th>
<th>group</th>
<th>separator</th>
<th>tree</th>
<th>button</th>
<th>filter</th>
<th>newline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Allowed elements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>colors</strong>: list of colors or HTML color codes mapped to Python conditions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>editable</strong>: <code>top</code> or <code>bottom</code> to allow in-place edit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>toolbar</strong>: set to <code>True</code> to display the top level of object hierarchies as a side toolbar (only for hierarchical lists, i.e. opened with actions that set the <code>view_type</code> to &quot;tree&quot; instead of &quot;mode&quot;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`<field`, `group`, `separator`, `tree`, `button`, `filter`, `newline`
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.openenerp.com/trunk/developers/web/qweb

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

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
- **date_delay**: name of field containing event duration

Allowed elements
- *field* (to define the label for each calendar event)

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

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

Allowed elements
- *field* (to define the label for each calendar event)

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
- **field**, with specific behavior:
  - first field in view is X axis, 2\(^{nd}\) one is Y, 3\(^{rd}\) one is Z
  - 2 fields required, 3\(^{rd}\) 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 (+,\(*\),\(**\),\(*\min\)*,\(*\max\))

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
- **context** attribute allows fields makes widgets that alter the search context (useful for context-sensitive fields, e.g. pricelist-dependent prices)

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

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

**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
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)

```
IELDS
id unique workflow record identifier
name name for the workflow (required)
act_node object model on which the workflow is defined (required)
auto set False to use a custom parser, by subclassing report_xml and declaring the report as follows:
header set False to suppress report header (default: True)
groups comma-separated list of groups allowed to view this report
keywords specify report type keyword (default: client_print_multi

Example RML report extract:
```
```

Workflow Transitions (edges)

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

```
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.

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)

```

Copyright © 2013 Open Object Press - All rights reserved – See license on page 12.
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):
        for wiz in self:
            if wiz.idea_age <= 3:
                raise osv.except_osv('UserError', 'Please select a larger age')

Wizard views

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

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.

WebServices – XML-RPC

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

Python example

```python
from xmlrpclib import
import xmlrpclib

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

# Create a new idea
url = 'http://<HOST>:<PORT>/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

```php
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"));
$response = $client->send($msg);
$uid = $response->value(1)->scalarval();
echo "Logged in as $USER ($uid:$uid)"

// Create a new idea
$arrayVal = array(
   'name'=>'Another Idea', "string"),
   'description'=>'This is another idea of mine', "string"),
```
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` attribute.
- 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)

```python
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

<table>
<thead>
<tr>
<th>Group*</th>
<th>Members</th>
<th>Bazaar/LP restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenERP Quality Team (~openerp)</td>
<td>OpenERP Core Team</td>
<td>Can merge and commit on official branches.</td>
</tr>
<tr>
<td>OpenERP Drivers (~openerp-drivers)</td>
<td>Selected active community members</td>
<td>Can confirm bugs and set milestones on bugs</td>
</tr>
<tr>
<td>OpenERP Community (~openerp-community)</td>
<td>Open group, anyone can join</td>
<td>Can create community branches where everyone can contribute</td>
</tr>
</tbody>
</table>

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

Copyright © 2010-2013 Open Object Press. All rights reserved.

You may take electronic copy of this work and distribute it if you don't change the content. You can also print a copy to be read by yourself only.

We have contracts with different publishers in different countries to sell and distribute paper or electronic based versions of this work (translated or not) in bookstores. This helps to distribute and promote the Open ERP product. It also helps us to create incentives to pay contributors and authors with the royalties.

Due to this, grants to translate, modify or sell this work are strictly forbidden, unless OpenERP s.a. (representing Open Object Press) gives you a written authorization for this.

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