



# **Open Object Developer Book**

***Release 6.0.0***

**OpenERP SA**

2013-12-29



# CONTENTS

<b>I Forewords</b>	<b>9</b>
1 Introduction	11
2 Who is this book for ?	13
3 Content of the book	15
4 About the author(s)	17
<b>II Getting starting with OpenERP development</b>	<b>19</b>
5 Installation	21
5.1 Windows . . . . .	21
5.2 Debian/Ubuntu . . . . .	21
5.3 Sources . . . . .	21
5.4 Development version . . . . .	22
6 Configuration	23
7 Command line options	27
7.1 General Options . . . . .	27
7.2 Database related options: . . . . .	27
7.3 Internationalization options: . . . . .	28
7.4 Options from previous versions: . . . . .	28
<b>III Architecture</b>	<b>29</b>
8 MVC architecture	31
8.1 MVC Model in OpenERP . . . . .	31
8.2 MVCSQL . . . . .	32
9 Technical architecture	35
9.1 The OpenERP server . . . . .	35
9.2 Modules . . . . .	35
9.3 Clients . . . . .	36
9.4 Relational database server and ORM . . . . .	36
9.5 Models . . . . .	36
9.6 Modules . . . . .	36
9.7 Services and WSGI . . . . .	37
9.8 XML-RPC, JSON-RPC . . . . .	37
10 Module Integrations	41

<b>11 Inheritance</b>	<b>43</b>
11.1 Traditional Inheritance . . . . .	43
11.2 Inheritance by Delegation . . . . .	44
<b>IV Modules</b>	<b>45</b>
<b>12 Module development</b>	<b>47</b>
12.1 Introduction . . . . .	47
12.2 Module Structure . . . . .	47
12.3 OpenERP Module Descriptor File : <code>__openerp__.py</code> . . . . .	56
12.4 Module creation . . . . .	58
12.5 Action creation . . . . .	61
<b>13 Objects, Fields and Methods</b>	<b>63</b>
13.1 OpenERP Objects . . . . .	63
13.2 The ORM - Object-relational mapping - Models . . . . .	63
13.3 OpenERP Object Attributes . . . . .	64
13.4 Object Inheritance - <code>_inherit</code> . . . . .	65
13.5 Inheritance by Delegation - <code>_inherits</code> . . . . .	66
13.6 Fields Introduction . . . . .	66
13.7 Type of Fields . . . . .	67
13.8 ORM methods . . . . .	76
<b>14 Views and Events</b>	<b>79</b>
14.1 Introduction to Views . . . . .	79
14.2 Form views . . . . .	79
14.3 Tree views . . . . .	81
14.4 Graph views . . . . .	81
14.5 Search views . . . . .	82
14.6 Calendar Views . . . . .	85
14.7 Gantt Views . . . . .	86
14.8 Design Elements . . . . .	88
14.9 Inheritance in Views . . . . .	95
14.10 Specify the views you want to use . . . . .	97
14.11 Events . . . . .	101
<b>15 Menu and Actions</b>	<b>103</b>
15.1 Menus . . . . .	103
15.2 Actions . . . . .	104
15.3 Security . . . . .	109
<b>V Creating Wizard - (The Process)</b>	<b>113</b>
<b>16 Introduction</b>	<b>115</b>
<b>17 Wizards - Principles</b>	<b>119</b>
17.1 The list of actions . . . . .	119
17.2 The result . . . . .	120
<b>18 Specification</b>	<b>125</b>
18.1 Form . . . . .	125
18.2 Fields . . . . .	125
<b>19 Add A New Wizard</b>	<b>127</b>
<b>20 osv_memory Wizard System</b>	<b>129</b>

<b>21</b>	<b>osv_memory configuration item</b>	<b>131</b>
21.1	The basic concepts . . . . .	131
21.2	Creating a basic configuration item . . . . .	131
21.3	Customizing your configuration item . . . . .	133
21.4	res.config's public API . . . . .	136
<b>22</b>	<b>Guidelines on how to convert old-style wizard to new osv_memory style</b>	<b>137</b>
22.1	OSV Memory Wizard . . . . .	137
22.2	Steps . . . . .	137
22.3	New Wizard File : <<module_name>>_<<filename>>.py . . . .	139
22.4	Old Wizard File : wizard_product_margin.py . . . . .	139
22.5	New Wizard File : wizard/<<module_name>>_<<filename>>_view.xml . . . . .	139
22.6	Default_focus attribute . . . . .	140
22.7	In Menu Item . . . . .	141
<b>VI</b>	<b>Reports</b>	<b>143</b>
<b>23</b>	<b>OpenOffice.org reports</b>	<b>147</b>
23.1	Creating a SXW . . . . .	148
23.2	Dynamic content in OpenOffice reports . . . . .	148
23.3	SXW2RML . . . . .	150
23.4	OpenERP Server PDF Output . . . . .	151
<b>24</b>	<b>XSL:RML reports</b>	<b>153</b>
24.1	XML Template . . . . .	157
24.2	Introduction to RML . . . . .	159
24.3	XSL:RML Stylesheet . . . . .	159
<b>25</b>	<b>Reports without corporate header</b>	<b>163</b>
<b>26</b>	<b>Each report with its own corporate header</b>	<b>165</b>
<b>27</b>	<b>Bar Codes</b>	<b>167</b>
27.1	Barcodes in RML files . . . . .	167
<b>28</b>	<b>How to add a new report</b>	<b>169</b>
<b>29</b>	<b>Usual TAGS</b>	<b>171</b>
29.1	Code within [[ ]] tags is python code . . . . .	171
<b>30</b>	<b>Unicode reports</b>	<b>173</b>
30.1	The solution consists of 3 parts . . . . .	173
30.2	All these ideas are taken from the forums . . . . .	173
<b>31</b>	<b>Html Reports Using Mako Templates</b>	<b>175</b>
31.1	Mako Template . . . . .	175
31.2	Python Blocks . . . . .	176
<b>VII</b>	<b>Server Action</b>	<b>179</b>
<b>32</b>	<b>Introduction</b>	<b>181</b>
<b>33</b>	<b>Step 1: Definition of Server Action</b>	<b>183</b>
33.1	Client Action . . . . .	183
33.2	Iteration . . . . .	184
33.3	Python Code . . . . .	184
33.4	Trigger . . . . .	184
33.5	Email Action . . . . .	185

33.6 Create Object . . . . .	186
33.7 Write Object . . . . .	187
33.8 Multi Action . . . . .	187
<b>34 Step 2: Mapping Server actions to workflows</b>	<b>189</b>
<b>VIII Workflow-Business Process</b>	<b>191</b>
<b>35 Introduction</b>	<b>193</b>
35.1 Example 1: Discount On Orders . . . . .	193
35.2 Example 2: A sale order that generates an invoice and a shipping order . . . . .	196
35.3 Example 3: Account invoice basic workflow . . . . .	196
<b>36 Defining Workflow</b>	<b>197</b>
<b>37 General structure of a workflow XML file</b>	<b>199</b>
<b>38 Activity</b>	<b>201</b>
38.1 Introduction . . . . .	201
38.2 The fields . . . . .	201
38.3 Defining activities using XML files . . . . .	202
38.4 Examples . . . . .	203
<b>39 Transition</b>	<b>205</b>
39.1 Introduction . . . . .	205
39.2 The fields . . . . .	205
39.3 Defining Transitions Using XML Files . . . . .	205
<b>40 Expressions</b>	<b>207</b>
<b>41 User Role</b>	<b>209</b>
<b>42 Error handling</b>	<b>211</b>
<b>43 Creating a Workflow</b>	<b>213</b>
43.1 Define the States of your object . . . . .	213
43.2 Define the State-change Handling Methods . . . . .	213
43.3 Create your Workflow XML file . . . . .	214
43.4 Add mymod_workflow.xml to __openerp__.py . . . . .	215
43.5 Add Workflow Buttons to your View . . . . .	215
43.6 Testing . . . . .	215
43.7 Troubleshooting . . . . .	215
<b>IX Dashboard</b>	<b>217</b>
<b>X I18n - Internationalization</b>	<b>221</b>
<b>44 Introduction</b>	<b>225</b>
<b>XI Testing</b>	<b>227</b>
<b>45 Unit testing</b>	<b>229</b>
45.1 Generalities . . . . .	229
45.2 Using unit tests . . . . .	229
45.3 Assert Tag . . . . .	230
45.4 Workflow Tag . . . . .	231

45.5 Function Tag . . . . .	231
<b>46 Acceptance testing</b>	<b>233</b>
46.1 Integrity tests on migrations . . . . .	233
46.2 Workflow tests . . . . .	233
46.3 Record creation . . . . .	233
<b>XII Serialization, Migration and Upgrading</b>	<b>235</b>
<b>47 Data Serialization</b>	<b>237</b>
47.1 XML Data Serialization . . . . .	237
47.2 YAML Data Serialization . . . . .	240
47.3 Writing YAML Tests . . . . .	247
47.4 CSV Data Serialization . . . . .	250
47.5 Multiple CSV Files . . . . .	252
<b>48 Data Migration - Import / Export</b>	<b>255</b>
48.1 Data Importation . . . . .	255
48.2 Data Loading . . . . .	257
<b>49 Upgrading</b>	<b>259</b>
49.1 Upgrading Server, Modules . . . . .	259
<b>XIII API</b>	<b>263</b>
<b>50 Working with Web Services</b>	<b>265</b>
50.1 Supported Web Services Protocols . . . . .	265
50.2 Available Web Services . . . . .	265
50.3 Example : writing data through the Web Services . . . . .	266
<b>51 XML-RPC Web services</b>	<b>269</b>
51.1 Interfaces . . . . .	269
51.2 Python Example . . . . .	275
51.3 PHP Example . . . . .	277
51.4 Perl Example . . . . .	278
<b>XIV Build and deploy</b>	<b>281</b>
<b>52 Building</b>	<b>285</b>
52.1 Dependencies . . . . .	285
52.2 Source distribution . . . . .	286
52.3 Binary distribution . . . . .	286
<b>53 Deploy</b>	<b>289</b>
53.1 Package script . . . . .	289
53.2 Batch . . . . .	289
53.3 SSH server . . . . .	289
53.4 Fabric . . . . .	289
<b>XV Appendix</b>	<b>291</b>
<b>54 Conventions</b>	<b>293</b>
54.1 Guidelines . . . . .	293
54.2 Module structure and file names . . . . .	293
54.3 Naming conventions . . . . .	293

<b>55 Translations</b>	<b>295</b>
55.1 How to change the language of the user interface ? . . . . .	295
55.2 Store a translation file on the server . . . . .	295
55.3 Translate to a new language . . . . .	295
55.4 Using context Dictionary for Translations . . . . .	297
<b>56 Technical Memento</b>	<b>299</b>
<b>57 Information Repository</b>	<b>301</b>
57.1 Setting Value . . . . .	301
57.2 IR Methods . . . . .	302
<b>XVI Community Book</b>	<b>303</b>
<b>58 Collaboration</b>	<b>305</b>
58.1 Launchpad and bazaar . . . . .	305
58.2 Working with Branch . . . . .	305
58.3 Registration and Configuration . . . . .	308
58.4 Branch . . . . .	309
58.5 How to commit . . . . .	309
58.6 Answer and bug tracking and management . . . . .	310
58.7 Translation . . . . .	310
58.8 Blueprints . . . . .	310
<b>XVII Remainder of old TOC</b>	<b>313</b>
<b>59 OpenERP Web Client v6.0</b>	<b>315</b>
59.1 OpenERP Web v6.0 . . . . .	315
59.2 What is Cherrypy ? . . . . .	316
59.3 Mako Template . . . . .	319
<b>60 Other Topics</b>	<b>325</b>
60.1 RAD Tools . . . . .	325
<b>Index</b>	<b>339</b>

# **Part I**

# **Forewords**



---

**CHAPTER  
ONE**

---

# **INTRODUCTION**

OpenERP is a rich development environment. Thanks to its Python and PostgreSQL bindings, and above all, its Object Relational Mapping (ORM), you can develop any arbitrary complex module in OpenERP.



---

CHAPTER  
TWO

---

## WHO IS THIS BOOK FOR ?



---

CHAPTER  
THREE

---

# CONTENT OF THE BOOK

*Book Contents*



---

CHAPTER  
FOUR

---

## ABOUT THE AUTHOR(S)

**Note:**

---

*Leave your comment here respecting the laws. Any comments deemed inappropriate (aggressive, racist, libelous, advertising, rude, off topic ...) will be removed.*

---



## **Part II**

# **Getting starting with OpenERP development**



# INSTALLATION

## 5.1 Windows

### Windows install:

- executable location
- howto

## 5.2 Debian/Ubuntu

How to get .deb packages

## 5.3 Sources

In order to get the sources, you will need Bazaar version control to pull the source from Launchpad. Check how to get Bazaar according to your development environment. After having installed and configured Bazaar, setup your development environment by typing:

```
mkdir source;cd source
```

Get the setup script of OpenERP by typing:

```
bzr cat -d lp:~openerp-dev/openerp-tools/trunk setup.sh | sh
```

Get the current trunk version of OpenERP by typing:

```
make init-trunk
```

The makefile contains other options. For details about options, please type:

```
make
```

Some dependencies are necessary to use OpenERP. Depending on your environment, you might have to install the following packets:

```
sudo apt-get install graphviz ghostscript postgresql python-imaging python-matplotlib
```

You then have to initialise the database. This will create a new openerp role:

```
make db-setup
```

Finally, launch the OpenERP server:

```
make server
```

Testing your installation can be done on <http://localhost:8069/>

## 5.4 Development version

Location of development version + specifics if necessary to precise

# CONFIGURATION

Two configuration files are available:

- one for the client: `~/.openerprc`
- one for the server: `~/.openerp_serverrc`

Those files follow the convention used by python's ConfigParser module.

Lines beginning with “#” or ";" are comments.

The client configuration file is automatically generated upon the first start. The one of the server can automatically be created using the command:

```
openerp-server.py -s
```

If they are not found, the server and the client will start with the default configuration.

## Server Configuration File

The server configuration file `.openerp_serverrc` is used to save server startup options. Here is the list of the available options:

- interface** Address to which the server will be bound
- port** Port the server will listen on
- database** Name of the database to use
- user** Username used when connecting to the database
- translate\_in** File used to translate OpenERP to your language
- translate\_out** File used to export the language OpenERP use
- language** Use this language as the language of the server. This must be specified as an ISO country code, as specified by the W3C.
- verbose** Enable debug output
- init** init a module (use “all” for all modules)
- update** update a module (use “all” for all modules)
- upgrade** Upgrade/install/uninstall modules
- db\_name** specify the database name
- db\_user** specify the database user name
- db\_password** specify the database password
- pg\_path** specify the pg executable path
- db\_host** specify the database host
- db\_port** specify the database port

**translate\_modules** Specify modules to export. Use in combination with –i18n-export

You can create your own configuration file by specifying -s or –save on the server command line. If you would like to write an alternative configuration file, use -c <config file> or –config=<config file> Here is a basic configuration for a server:

```
[options]
verbose = False
xmlrpc = True
database = terp
update = {}
port = 8069
init = {}
interface = 127.0.0.1
reportgz = False
```

#### Full Example for Server V5.0

```
[printer]
path = none
softpath_html = none
preview = True
softpath = none

[logging]
output = stdout
logger =
verbose = True
level = error

[help]
index = http://www.openerp.com/documentation/user-manual/
context = http://www.openerp.com/scripts/context_index.php

[form]
autosave = False
toolbar = True

[support]
recipient = support@openerp.com
support_id =

[tip]
position = 0
autostart = False

[client]
lang = en_US
default_path = /home/user
filetype = {}
theme = none
toolbar = icons
form_tab_orientation = 0
form_tab = top

[survey]
position = 3

[path]
pixmaps = /usr/share/pixmaps/openerp-client/
share = /usr/share/openerp-client/

[login]
db = eo2
```

```
login = admin
protocol = http://
port = 8069
server = localhost
```



# COMMAND LINE OPTIONS

## 7.1 General Options

<b>--version</b>	show program version number and exit
<b>-h, --help</b>	show this help message and exit
<b>-c CONFIG, --config=CONFIG</b>	specify alternate config file
<b>-s, --save</b>	save configuration to <code>~/.terp_serverrc</code>
<b>-v, --verbose</b>	enable debugging
<b>--pidfile=PIDFILE</b>	file where the server pid will be stored
<b>--logfile=LOGFILE</b>	file where the server log will be stored
<b>-n INTERFACE, --interface=INTERFACE</b>	specify the TCP IP address
<b>-p PORT, --port=PORT</b>	specify the TCP port
<b>--net_interface=NETINTERFACE</b>	specify the TCP IP address for netrpc
<b>--net_port=NETPORT</b>	specify the TCP port for netrpc
<b>--no-netrpc</b>	disable netrpc
<b>--no-xmlrpc</b>	disable xmlrpc
<b>-i INIT, --init=INIT</b>	init a module (use “all” for all modules)
<b>--without-demo=WITHOUT_DEMO</b>	load demo data for a module (use “all” for all modules)
<b>-u UPDATE, --update=UPDATE</b>	update a module (use “all” for all modules)
<b>--stop-after-init</b>	stop the server after it initializes
<b>--debug</b>	enable debug mode
<b>-S, --secure</b>	launch server over https instead of http
<b>--smtp=SMTP_SERVER</b>	specify the SMTP server for sending mail

## 7.2 Database related options:

<b>-d DB_NAME, --database=DB_NAME</b>	specify the database name
<b>-r DB_USER, --db_user=DB_USER</b>	specify the database user name
<b>-w DB_PASSWORD, --db_password=DB_PASSWORD</b>	specify the database password

```
--pg_path=PG_PATH specify the pg executable path  
--db_host=DB_HOST specify the database host  
--db_port=DB_PORT specify the database port
```

## 7.3 Internationalization options:

Use these options to translate OpenERP to another language. See i18n section of the user manual. Option ‘-l’ is mandatory.

```
-l LANGUAGE, --language=LANGUAGE specify the language of the translation file. Use it with -i18n-export and -i18n-import  
--i18n-export=TRANSLATE_OUT export all sentences to be translated to a CSV file and exit  
--i18n-import=TRANSLATE_IN import a CSV file with translations and exit  
--modules=TRANSLATE_MODULES specify modules to export. Use in combination with -i18n-export
```

## 7.4 Options from previous versions:

Some options were removed in version 6. For example, `price_accuracy` is now configured through the `decimal_accuracy` screen.

# **Part III**

# **Architecture**



# MVC ARCHITECTURE

According to [Wikipedia](#), “a Model-view-controller (MVC) is an architectural pattern used in software engineering”. In complex computer applications presenting lots of data to the user, one often wishes to separate data (model) and user interface (view) concerns. Changes to the user interface does therefore not impact data management, and data can be reorganized without changing the user interface. The model-view-controller solves this problem by decoupling data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

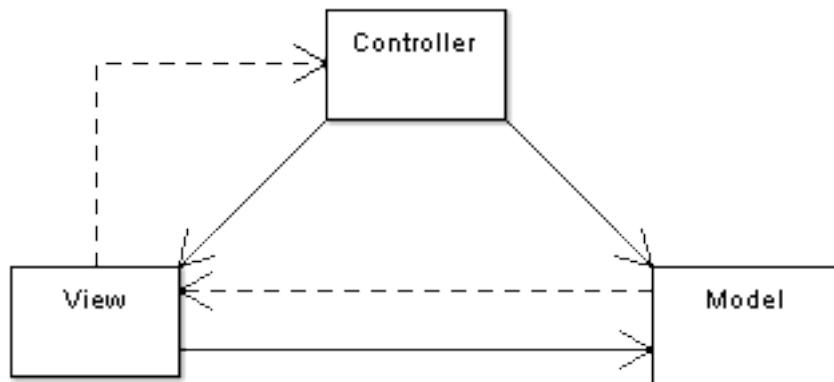


Figure 8.1: MVC Diagram

For example in the diagram above, the solid lines for the arrows starting from the controller and going to both the view and the model mean that the controller has a complete access to both the view and the model. The dashed line for the arrow going from the view to the controller means that the view has a limited access to the controller. The reasons of this design are :

- From **View to Model** : the model sends notification to the view when its data has been modified in order the view to redraw its content. The model doesn't need to know the inner workings of the view to perform this operation. However, the view needs to access the internal parts of the model.
- From **View to Controller** : the reason why the view has limited access to the controller is because the dependencies from the view to the controller need to be minimal: the controller can be replaced at any moment.

## 8.1 MVC Model in OpenERP

In OpenERP, we can apply this model-view-controller semantic with

- model : The PostgreSQL tables.
- view : views are defined in XML files in OpenERP.
- controller : The objects of OpenERP.

## 8.2 MVCSQL

### 8.2.1 Example 1

Suppose sale is a variable on a record of the sale.order object related to the ‘sale\_order’ table. You can acquire such a variable doing this.:

```
sale = self.browse(cr, uid, ID)
```

(where cr is the current row, from the database cursor, uid is the current user’s ID for security checks, and ID is the sale order’s ID or list of IDs if we want more than one)

Suppose you want to get: the country name of the first contact of a partner related to the ID sale order. You can do the following in OpenERP:

```
country_name = sale.partner_id.address[0].country_id.name
```

If you want to write the same thing in traditional SQL development, it will be in python: (we suppose cr is the cursor on the database, with psycopg)

```
cr.execute('select partner_id from sale_order where id=%d', (ID,))
partner_id = cr.fetchone()[0]
cr.execute('select country_id from res_partner_address where partner_id=%d', (partner_id,))
country_id = cr.fetchone()[0]
cr.execute('select name from res_country where id=%d', (country_id,))
del partner_id
del country_id
country_name = cr.fetchone()[0]
```

Of course you can do better if you develop smartly in SQL, using joins or subqueries. But you have to be smart and most of the time you will not be able to make such improvements:

- Maybe some parts are in others functions
- There may be a loop in different elements
- You have to use intermediate variables like country\_id

The first operation as an object call is much better for several reasons:

- It uses objects facilities and works with modules inheritances, overload, ...
- It’s simpler, more explicit and uses less code
- It’s much more efficient as you will see in the following examples
- Some fields do not directly correspond to a SQL field (e.g.: function fields in Python)

### 8.2.2 Prefetching

Suppose that later in the code, in another function, you want to access the name of the partner associated to your sale order. You can use this:

```
partner_name = sale.partner_id.name
```

And this will not generate any SQL query as it has been prefetched by the object relational mapping engine of OpenERP.

### 8.2.3 Loops and special fields

Suppose now that you want to compute the totals of 10 sales order by countries. You can do this in OpenERP within a OpenERP object:

```

def get_totals(self, cr, uid, ids):
    countries = {}
    for sale in self.browse(cr, uid, ids):
        country = sale.partner_invoice_id.country
        countries.setdefault(country, 0.0)
        countries[country] += sale.amount_untaxed
    return countries

```

And, to print them as a good way, you can add this on your object:

```

def print_totals(self, cr, uid, ids):
    result = self.get_totals(cr, uid, ids)
    for country in result.keys():
        print '[%s] %s: %.2f' (country.code, country.name, result[country])

```

The 2 functions will generate 4 SQL queries in total ! This is due to the SQL engine of OpenERP that does prefetching, works on lists and uses caching methods. The 3 queries are:

1. Reading the sale.order to get ID's of the partner's address
2. Reading the partner's address for the countries
3. Calling the amount\_untaxed function that will compute a total of the sale order lines
4. Reading the countries info (code and name)

That's great because if you run this code on 1000 sales orders, you have the guarantee to only have 4 SQL queries.

Notes:

- IDS is the list of the 10 ID's: [12,15,18,34, ...,99]
- The arguments of a function are always the same:
  - cr: the cursor database (from psycopg)
  - uid: the user id (for security checks)
- If you run this code on 5000 sales orders, you may have 8 SQL queries because as SQL queries are not allowed to take too much memory, it may have to do two separate readings.

## 8.2.4 Complex example

Here is a complete example, from the OpenERP official distribution, of the function that does bill of material explosion and computation of associated routings:

```

class mrp_bom(osv.osv):
    ...
    def _bom_find(self, cr, uid, product_id, product_uom, properties=[]):
        bom_result = False
        # Why searching on BoM without parent ?
        cr.execute('select id from mrp_bom where product_id=%d and bom_id is null
                    order by sequence', (product_id,))
        ids = map(lambda x: x[0], cr.fetchall())
        max_prop = 0
        result = False
        for bom in self.pool.get('mrp.bom').browse(cr, uid, ids):
            prop = 0
            for prop_id in bom.property_ids:
                if prop_id.id in properties:
                    prop+=1
            if (prop>max_prop) or ((max_prop==0) and not result):
                result = bom.id
                max_prop = prop
        return result

```

```

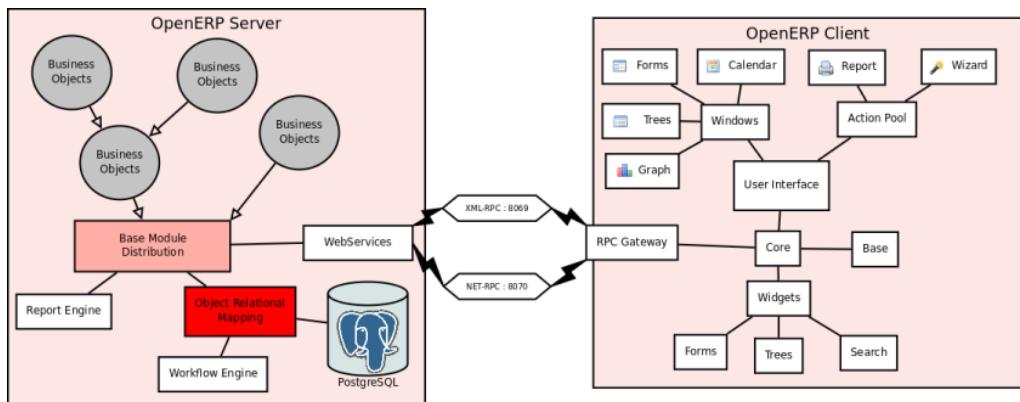
def _bom_explode(self, cr, uid, bom, factor, properties, addthis=False, level=10):
    factor = factor / (bom.product_efficiency or 1.0)
    factor = rounding(factor, bom.product_rounding)
    if factor<bom.product_rounding:
        factor = bom.product_rounding
    result = []
    result2 = []
    phantom = False
    if bom.type=='phantom' and not bom.bom_lines:
        newbom = self._bom_find(cr, uid, bom.product_id.id,
                               bom.product_uom.id, properties)
        if newbom:
            res = self._bom_explode(cr, uid, self.browse(cr, uid, [newbom])[0],
                                   factor*bom.product_qty, properties, addthis=True, level=level+10)
            result = result + res[0]
            result2 = result2 + res[1]
            phantom = True
        else:
            phantom = False
    if not phantom:
        if addthis and not bom.bom_lines:
            result.append(
            {
                'name': bom.product_id.name,
                'product_id': bom.product_id.id,
                'product_qty': bom.product_qty * factor,
                'product_uom': bom.product_uom.id,
                'product_uos_qty': bom.product_uos and
                                   bom.product_uos_qty * factor or False,
                'product_uos': bom.product_uos and bom.product_uos.id or False,
            })
        if bom.routing_id:
            for wc_use in bom.routing_id.workcenter_lines:
                wc = wc_use.workcenter_id
                d, m = divmod(factor, wc_use.workcenter_id.capacity_per_cycle)
                mult = (d + (m and 1.0 or 0.0))
                cycle = mult * wc_use.cycle_nbr
                result2.append({
                    'name': bom.routing_id.name,
                    'workcenter_id': wc.id,
                    'sequence': level+(wc_use.sequence or 0),
                    'cycle': cycle,
                    'hour': float(wc_use.hour_nbr*mult +
                                  (wc.time_start+wc.time_stop+cycle*wc.time_cycle) *
                                  (wc.time_efficiency or 1.0)),
                })
        for bom2 in bom.bom_lines:
            res = self._bom_explode(cr, uid, bom2, factor, properties,
                                   addthis=True, level=level+10)
            result = result + res[0]
            result2 = result2 + res[1]
    return result, result2

```

# TECHNICAL ARCHITECTURE

OpenERP is a multitenant, three-tier architecture. The application tier itself is written as a core, multiple additional modules can be installed to create a particular configuration of OpenERP.

The core of OpenERP and its modules are written in [Python](#). The functionality of a module is exposed through XML-RPC (and/or NET-RPC depending on the server's configuration)[#]. Modules typically make use of OpenERP's ORM to persist their data in a relational database (PostgreSQL). Modules can insert data in the database during installation by providing XML, CSV, or YML files.



## 9.1 The OpenERP server

OpenERP provides an application server on which specific business applications can be built. It is also a complete development framework, offering a range of features to write those applications. The salient features are a flexible ORM, a MVC architecture, extensible data models and views, different report engines, all tied together in a coherent, network-accessible framework.

From a developer perspective, the server acts both as a library which brings the above benefits while hiding the low-level, nitty-gritty details, and as a simple way to install, configure and run the written applications.

## 9.2 Modules

By itself, the OpenERP server is not very useful. For any enterprise, the value of OpenERP lies in its different modules. It is the role of the modules to implement any business needs. The server is only the necessary machinery to run the modules. A lot of modules already exist. Any official OpenERP release includes about 170 of them, and hundreds of modules are available through the community. Examples of modules are Account, CRM, HR, Marketing, MRP, Sale, etc.

A module is usually composed of data models, together with some initial data, views definitions (i.e. how data from specific data models should be displayed to the user), wizards (specialized screens to help the user for specific interactions), workflows definitions, and reports.

## 9.3 Clients

Clients can communicate with an OpenERP server using XML-RPC. A custom, faster protocol called NET-RPC is also provided but will shortly disappear, replaced by JSON-RPC. XML-RPC, as JSON-RPC in the future, makes it possible to write clients for OpenERP in a variety of programming languages. OpenERP S.A. develops two different clients: a desktop client, written with the widely used [GTK+](#) graphical toolkit, and a web client that should run in any modern web browser.

As the logic of OpenERP should entirely reside on the server, the client is conceptually very simple; it issues a request to the server and display the result (e.g. a list of customers) in different manners (as forms, lists, calendars, ...). Upon user actions, it will send modified data to the server.

## 9.4 Relational database server and ORM

The data tier of OpenERP is provided by a PostgreSQL relational database. While direct SQL queries can be executed from OpenERP modules, most database access to the relational database is done through the [Object-Relational Mapping](#).

The ORM is one of the salient features mentioned above. The data models are described in Python and OpenERP creates the underlying database tables. All the benefits of RDBMS (unique constraints, relational integrity, efficient querying, ...) are used when possible and completed by Python flexibility. For instance, arbitrary constraints written in Python can be added to any model. Different modular extensibility mechanisms are also afforded by OpenERP[#].

## 9.5 Models

To define data models and otherwise pursue any work with the associated data, OpenERP as many ORMs uses the concept of ‘model’. A model is the authoritative specification of how some data are structured, constrained, and manipulated. In practice, a model is written as a Python class. The class encapsulates anything there is to know about the model: the different fields composing the model, default values to be used when creating new records, constraints, and so on. It also holds the dynamic aspect of the data it controls: methods on the class can be written to implement any business needs (for instance, what to do upon user action, or upon workflow transitions).

There are two different models. One is simply called ‘model’, and the second is called ‘transient model’. The two models provide the same capabilities with a single difference: transient models are automatically cleared from the database (they can be cleaned when some limit on the number of records is reached, or when they are untouched for some time).

To describe the data model per se, OpenERP offers a range of different kind of fields. There are basic fields such as integer, or text fields. There are relational fields to implement one-to-many, many-to-one, and many-to-many relationships. There are so-called function fields, which are dynamically computed and are not necessarily available in database, and more.

Access to data is controlled by OpenERP and configured by different mechanisms. This ensures that different users can have read and/or write access to only the relevant data. Access can be controlled with respect to user groups and rules based on the value of the data themselves.

## 9.6 Modules

OpenERP supports a modular approach both from a development perspective and a deployment point of view. In essence, a module groups everything related to a single concern in one meaningful entity. It is comprised of models, views, workflows, and wizards.

## 9.7 Services and WSGI

Everything in OpenERP, and models methods in particular, are exposed via the network and a security layer. Access to the data model is in fact a ‘service’ and it is possible to expose new services. For instance, a WebDAV service and a FTP service are available.

While not mandatory, the services can make use of the [WSGI](#) stack. WSGI is a standard solution in the Python ecosystem to write HTTP servers, applications, and middleware which can be used in a mix-and-match fashion. By using WSGI, it is possible to run OpenERP in any WSGI-compliant server, but also to use OpenERP to host a WSGI application.

A striking example of this possibility is the OpenERP Web project. OpenERP Web is the server-side counter part to the web clients. It is OpenERP Web which provides the web pages to the browser and manages web sessions. OpenERP Web is a WSGI-compliant application. As such, it can be run as a stand-alone HTTP server or embedded inside OpenERP.

## 9.8 XML-RPC, JSON-RPC

The access to the models makes also use of the WSGI stack. This can be done using the XML-RPC protocol, and JSON-RPC will be added soon.

Explanation of modules:

### Server - Base distribution

We use a distributed communication mechanism inside the OpenERP server. Our engine supports most commonly distributed patterns: request/reply, publish/subscribe, monitoring, triggers/callback, ...

Different business objects can be in different computers or the same objects can be on multiple computers to perform load-balancing.

### Server - Object Relational Mapping (ORM)

This layer provides additional object functionality on top of PostgreSQL:

- Consistency: powerful validity checks,
- Work with objects (methods, references, ...)
- Row-level security (per user/group/role)
- Complex actions on a group of resources
- Inheritance

### Server - Web-Services

The web-service module offer a common interface for all web-services

- SOAP
- XML-RPC
- NET-RPC

Business objects can also be accessed via the distributed object mechanism. They can all be modified via the client interface with contextual views.

### Server - Workflow Engine

Workflows are graphs represented by business objects that describe the dynamics of the company. Workflows are also used to track processes that evolve over time.

An example of workflow used in OpenERP:

A sales order generates an invoice and a shipping order

## **Server - Report Engine**

Reports in OpenERP can be rendered in different ways:

- Custom reports: those reports can be directly created via the client interface, no programming required.  
Those reports are represented by business objects (ir.report.custom)
- High quality personalized reports using openreport: no programming required but you have to write 2 small XML files:
  - a template which indicates the data you plan to report
  - an XSL:RML stylesheet
- Hard coded reports
- OpenOffice Writer templates

Nearly all reports are produced in PDF.

## **Server - Business Objects**

Almost everything is a business object in OpenERP, they describe all data of the program (workflows, invoices, users, customized reports, ...). Business objects are described using the ORM module. They are persistent and can have multiple views (described by the user or automatically calculated).

Business objects are structured in the /module directory.

## **Client - Wizards**

Wizards are graphs of actions/windows that the user can perform during a session.

## **Client - Widgets**

Widgets are probably, although the origin of the term seems to be very difficult to trace, “WIndow gaDGETS” in the IT world, which mean they are gadgets before anything, which implement elementary features through a portable visual tool.

All common widgets are supported:

- entries
- textboxes
- floating point numbers
- dates (with calendar)
- checkboxes
- ...

And also all special widgets:

- buttons that call actions
- references widgets
  - one2one
  - many2one
  - many2many
  - one2many in list
  - ...

Widget have different appearances in different views. For example, the date widget in the search dialog represents two normal dates for a range of date (from...to...).

Some widgets may have different representations depending on the context. For example, the one2many widget can be represented as a form with multiple pages or a multi-columns list.

Events on the widgets module are processed with a callback mechanism. A callback mechanism is a process whereby an element defines the type of events he can handle and which methods should be called when this event is triggered. Once the event is triggered, the system knows that the event is bound to a specific method, and calls that method back. Hence callback.



# MODULE INTEGRATIONS

There are many different modules available for OpenERP and suited for different business models. Nearly all of these are optional (except ModulesAdminBase), making it easy to customize OpenERP to serve specific business needs. All the modules are in a directory named addons/ on the server. You simply need to copy or delete a module directory in order to either install or delete the module on the OpenERP platform.

Some modules depend on other modules. See the file addons/module/\_\_openerp\_\_.py for more information on the dependencies.

Here is an example of \_\_openerp\_\_.py:

```
{  
    "name" : "Open TERP Accounting",  
    "version" : "1.0",  
    "author" : "Bob Gates - Not So Tiny",  
    "website" : "http://www.openerp.com/",  
    "category" : "Generic Modules/Others",  
    "depends" : ["base"],  
    "description" : """A  
    Multiline  
    Description  
    """,  
    "init_xml" : ["account_workflow.xml", "account_data.xml", "account_demo.xml"],  
    "demo_xml" : ["account_demo.xml"],  
    "update_xml" : ["account_view.xml", "account_report.xml", "account_wizard.xml"],  
    "active": False,  
    "installable": True  
}
```

When initializing a module, the files in the init\_xml list are evaluated in turn and then the files in the update\_xml list are evaluated. When updating a module, only the files from the **update\_xml** list are evaluated.



# INHERITANCE

## 11.1 Traditional Inheritance

### 11.1.1 Introduction

Objects may be inherited in some custom or specific modules. It is better to inherit an object to add/modify some fields.

It is done with:

```
_inherit='object.name'
```

### 11.1.2 Extension of an object

There are two possible ways to do this kind of inheritance. Both ways result in a new class of data, which holds parent fields and behaviour as well as additional fields and behaviour, but they differ in heavy programmatical consequences.

While Example 1 creates a new subclass “custom\_material” that may be “seen” or “used” by any view or tree which handles “network.material”, this will not be the case for Example 2.

This is due to the table (other.material) the new subclass is operating on, which will never be recognized by previous “network.material” views or trees.

Example 1:

```
class custom_material(osv.osv):
    _name = 'network.material'
    _inherit = 'network.material'
    _columns = {
        'manuf_warranty': fields.boolean('Manufacturer warranty?'),
    }
    _defaults = {
        'manuf_warranty': lambda *a: False,
    }
custom_material()
```

**Tip:**

---

*Notice*

*\_name == \_inherit*

---

In this example, the ‘custom\_material’ will add a new field ‘manuf\_warranty’ to the object ‘network.material’. New instances of this class will be visible by views or trees operating on the superclasses table ‘network.material’.

This inheritance is usually called “class inheritance” in Object oriented design. The child inherits data (fields) and behavior (functions) of his parent.

Example 2:

```
class other_material(osv.osv):
    _name = 'other.material'
    _inherit = 'network.material'
    _columns = {
        'manuf_warranty': fields.boolean('Manufacturer warranty?'),
    }
    _defaults = {
        'manuf_warranty': lambda *a: False,
    }
other_material()
```

**Tip:**

*Notice*  
`_name != _inherit`

In this example, the ‘other\_material’ will hold all fields specified by ‘network.material’ and it will additionally hold a new field ‘manuf\_warranty’. All those fields will be part of the table ‘other.material’. New instances of this class will therefore never be seen by views or trees operating on the superclasses table ‘network.material’.

This type of inheritance is known as “inheritance by prototyping” (e.g. Javascript), because the newly created subclass “copies” all fields from the specified superclass (prototype). The child inherits data (fields) and behavior (functions) of his parent.

## 11.2 Inheritance by Delegation

**Syntax ::**

```
class tiny_object(osv.osv)
    _name = 'tiny.object'
    _table = 'tiny_object'
    _inherits = { 'tiny.object_a' : 'name_col_a', 'tiny.object_b' : 'name_col_b',
                 ... , 'tiny.object_n' : 'name_col_n' }
    (...)
```

The object ‘tiny.object’ inherits from all the columns and all the methods from the n objects ‘tiny.object\_a’, ..., ‘tiny.object\_n’.

To inherit from multiple tables, the technique consists in adding one column to the table tiny\_object per inherited object. This column will store a foreign key (an id from another table). The values ‘name\_col\_a’ ‘name\_col\_b’ ... ‘name\_col\_n’ are of type string and determine the title of the columns in which the foreign keys from ‘tiny.object\_a’, ..., ‘tiny.object\_n’ are stored.

This inheritance mechanism is usually called ”*instance inheritance* ” or ”*value inheritance* ”. A resource (instance) has the VALUES of its parents.

## **Part IV**

# **Modules**



# MODULE DEVELOPMENT

## 12.1 Introduction

OpenERP uses a [three-tier architecture](#). The application tier itself is written as a core and multiple additional modules that can be installed or not to create a particular configuration of OpenERP.

The core of OpenERP and its different modules are written in [Python](#). The functionality of a module is exposed through XML-RPC (and/or NET-RPC depending on the server's configuration). Modules also typically make use of OpenERP ORM to persist their data in a relational database (PostgreSQL). Modules can insert data in the database during installation by providing XML (or CSV or YML) files.

Although modules are a simple way to structure a complex application, OpenERP modules also extend the system. Modules are also called addons (they could also have been called plugins).

In a typical configuration of OpenERP, the following modules can be found:

- base: the most basic module; it is always installed and can be thought as being part of the core of OpenERP. It defines `ir.property`, `res.company`, `res.request`, `res.currency`, `res.users`, `res.partner`, and so on.
- crm: Customer & Supplier Relationship management.
- sale: Sales management.
- mrp: Manufacturing Resource Planning.

By using Python, XML files, and relying on OpenERP's ORM and its extensibility mechanisms, new modules can be written easily and quickly. OpenERP's open source nature and its numerous modules also provide a lot of examples for any new development.

## 12.2 Module Structure

### 12.2.1 The Modules

1. Introduction
2. **Files & Directories**
  - (a) `__openerp__.py`
  - (b) `__init__.py`
  - (c) **XML Files**
    - i. Actions
    - ii. Menu Entries
    - iii. Reports

- iv. Wizards
- 3. Profiles

## 12.2.2 Modules - Files and Directories

All the modules are located in the server/addons directory.

The following steps are necessary to create a new module:

- create a subdirectory in the server/addons directory
- create a module description file: **\_\_openerp\_\_.py**
- create the **Python** file containing the **objects**
- create **.xml files** that download the data (views, menu entries, demo data, ...)
- optionally create **reports**, **wizards** or **workflows**.

### The Modules - Files And Directories - XML Files

XML files located in the module directory are used to modify the structure of the database. They are used for many purposes, among which we can cite :

- initialization and demonstration data declaration,
- views declaration,
- reports declaration,
- wizards declaration,
- workflows declaration.

General structure of OpenERP XML files is more detailed in the [XML Data Serialization](#) section. Look here if you are interested in learning more about *initialization* and *demonstration data declaration* XML files. The following section are only related to XML specific to *actions*, *menu entries*, *reports*, *wizards* and *workflows* declaration.

### Python Module Descriptor File **\_\_init\_\_.py**

#### The **\_\_init\_\_.py** file

The **\_\_init\_\_.py** file is, like any Python module, executed at the start of the program. It needs to import the Python files that need to be loaded.

So, if you create a “**module.py**” file, containing the description of your objects, you have to write one line in **\_\_init\_\_.py**:

```
import module
```

### OpenERP Module Descriptor File **\_\_openerp\_\_.py**

In the created module directory, you must add a **\_\_openerp\_\_.py** file. This file, which must be in Python format, is responsible to

1. determine the *XML files that will be parsed* during the initialization of the server, and also to
2. determine the *dependencies* of the created module.

This file must contain a Python dictionary with the following values:

#### **name**

The (Plain English) name of the module.

**version**

The version of the module, on 2 digits (1.2 or 2.0).

**description**

The module description (text) including documentation on how to use your modules.

**author**

The author of the module.

**website**

The website of the module.

**license**

The license of the module (default:GPL-2).

**depends**

List of modules on which this module depends. The base module must almost always be in the dependencies because some necessary data for the views, reports, ... are in the base module.

**init**

List of .xml files to load when the server is launched with the “–init=module” argument. Filepaths must be relative to the directory where the module is. OpenERP XML File Format is detailed in this section.

**data**

List of .xml files to load when the server is launched with the “–update=module” launched. Filepaths must be relative to the directory where the module is. OpenERP XML File Format is detailed in this section.

**demo**

List of .xml files to provide demo data. Filepaths must be relative to the directory where the module is. OpenERP XML File Format is detailed in this section.

**installable**

True or False. Determines if the module is installable or not.

**images**

List of .png files to provide screenshots, used on <http://apps.openerp.com>.

**active**

True or False (default: False). Determines the modules that are installed on the database creation.

**test**

List of .yml files to provide YAML tests.

**Example**

Here is an example of \_\_openerp\_\_.py file for the product module

```
{  
    "name" : "Products & Pricelists",  
    "version" : "1.1",  
    "author" : "Open",  
    "category" : "Generic Modules/Inventory Control",  
    "depends" : ["base", "account"],  
    "init_xml" : [],  
    "demo_xml" : ["product_demo.xml"],  
    "update_xml" : ["product_data.xml", "product_report.xml", "product_wizard.xml",  
                  "product_view.xml", "pricelist_view.xml"],
```

```

    "installable": True,
    "active": True
}

```

The files that must be placed in init\_xml are the ones that relate to the workflow definition, data to load at the installation of the software and the data for the demonstrations.

The files in **update\_xml** concern: views, reports and wizards.

## Objects

All OpenERP resources are objects: menus, actions, reports, invoices, partners, ... OpenERP is based on an object relational mapping of a database to control the information. Object names are hierarchical, as in the following examples:

- account.transfer : a money transfer
- account.invoice : an invoice
- account.invoice.line : an invoice line

Generally, the first word is the name of the module: account, stock, sale.

Other advantages of an ORM;

- simpler relations : invoice.partner.address[0].city
- objects have properties and methods: invoice.pay(3400 EUR),
- inheritance, high level constraints, ...

It is easier to manipulate one object (example, a partner) than several tables (partner address, categories, events, ...)

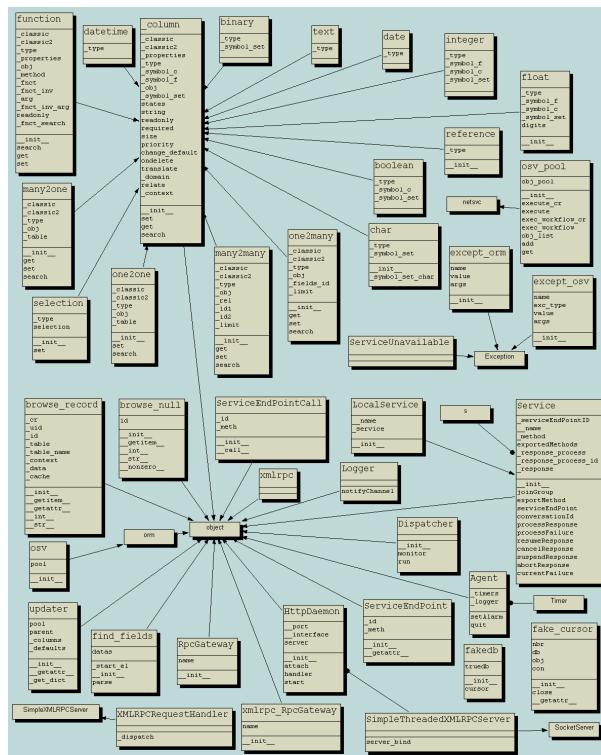


Figure 12.1: The Physical Objects Model of [OpenERP version 3.0.3]

## PostgreSQL

The ORM of OpenERP is constructed over PostgreSQL. It is thus possible to query the object used by OpenERP using the object interface or by directly using SQL statements.

But it is dangerous to write or read directly in the PostgreSQL database, as you will shortcut important steps like constraints checking or workflow modification.

**Note:**

*The Physical Database Model of OpenERP*

## Pre-Installed Data

Data can be inserted or updated into the PostgreSQL tables corresponding to the OpenERP objects using XML files. The general structure of an OpenERP XML file is as follows:

```
<?xml version="1.0"?>
<openerp>
    <data>
        <record model="model.name_1" id="id_name_1">
            <field name="field1">
                "field1 content"
            </field>
            <field name="field2">
                "field2 content"
            </field>
            ...
        </record>
        <record model="model.name_2" id="id_name_2">
            ...
        </record>
        ...
    </data>
</openerp>
```

Fields content are strings that must be encoded as *UTF-8* in XML files.

Let's review an example taken from the OpenERP source (base\_demo.xml in the base module):

```
<record model="res.company" id="main_company">
    <field name="name">Tiny sprl</field>
    <field name="partner_id" ref="main_partner"/>
    <field name="currency_id" ref="EUR"/>
</record>

<record model="res.users" id="user_admin">
    <field name="login">admin</field>
    <field name="password">admin</field>
    <field name="name">Administrator</field>
    <field name="signature">Administrator</field>
    <field name="action_id" ref="action_menu_admin"/>
    <field name="menu_id" ref="action_menu_admin"/>
    <field name="address_id" ref="main_address"/>
    <field name="groups_id" eval="[(6,0,[group_admin])]"/>
    <field name="company_id" ref="main_company"/>
</record>
```

This last record defines the admin user :

- The fields login, password, etc are straightforward.

- The **ref** attribute allows to fill relations between the records :

```
<field name="company_id" ref="main_company"/>
```

The field **company\_id** is a many-to-one relation from the user object to the company object, and **main\_company** is the id of to associate.

- The **eval** attribute allows to put some python code in the xml: here the groups\_id field is a many2many. For such a field, “[ $(6,0,[group_admin])$ ]” means : Remove all the groups associated with the current user and use the list [group\_admin] as the new associated groups (and group\_admin is the id of another record).
- The **search** attribute allows to find the record to associate when you do not know its xml id. You can thus specify a search criteria to find the wanted record. The criteria is a list of tuples of the same form than for the predefined search method. If there are several results, an arbitrary one will be chosen (the first one):

```
<field name="partner_id" search="[]" model="res.partner"/>
```

This is a classical example of the use of **search** in demo data: here we do not really care about which partner we want to use for the test, so we give an empty list. Notice the **model** attribute is currently mandatory.

## Record Tag

### Description

The addition of new data is made with the record tag. This one takes a mandatory attribute : **model**. Model is the object name where the insertion has to be done. The tag record can also take an optional attribute: **id**. If this attribute is given, a variable of this name can be used later on, in the same file, to make reference to the newly created resource ID.

A record tag may contain field tags. They indicate the record's fields value. If a field is not specified the default value will be used.

### Example

```
<record model="ir.actions.report.xml" id="10">
    <field name="model">account.invoice</field>
    <field name="name">Invoices List</field>
    <field name="report_name">account.invoice.list</field>
    <field name="report_xsl">account/report/invoice.xsl</field>
    <field name="report_xml">account/report/invoice.xml</field>
</record>
```

## Field tag

The attributes for the field tag are the following:

**name** [mandatory] the field name

**eval** [optional] python expression that indicating the value to add

**ref** reference to an id defined in this file

**model** model to be looked up in the search

**search** a query

## Function tag

A function tag can contain other function tags.

**model** [mandatory] The model to be used

**name** [mandatory] the function given name

**eval** should evaluate to the list of parameters of the method to be called, excluding cr and uid

### Example

```
<function model="ir.ui.menu" name="search" eval="[[('name', '=', 'Operations')]]"/>
```

## Getitem tag

Takes a subset of the evaluation of the last child node of the tag.

**type** [mandatory] int or list

**index** [mandatory] int or string (a key of a dictionary)

### Example

Evaluates to the first element of the list of ids returned by the function node

```
<getitem index="0" type="list">
    <function model="ir.ui.menu" name="search" eval="[[('name', '=', 'Operations')]]"/>
</getitem>
```

## i18n

### Improving Translations

#### Translating in launchpad

Translations are managed by the [Launchpad](#) Web interface. Here, you'll find the list of translatable projects.

Please read the [FAQ](#) before asking questions.

#### Translating your own module

Changed in version 5.0. Contrary to the 4.2.x version, the translations are now done by module. So, instead of an unique `i18n` folder for the whole application, each module has its own `i18n` folder. In addition, OpenERP can now deal with `.po`<sup>1</sup> files as import/export format. The translation files of the installed languages are automatically loaded when installing or updating a module. OpenERP can also generate a `.tgz` archive containing well organised `.po` files for each selected module.

## Process

### Defining the process

Through the interface and module recorder. Then, put the generated XML in your own module.

## Views

### Technical Specifications - Architecture - Views

Views are a way to represent the objects on the client side. They indicate to the client how to lay out the data coming from the objects on the screen.

There are two types of views:

- form views
- tree views

<sup>1</sup> <http://www.gnu.org/software/autoconf/manual/gettext/PO-Files.html#PO-Files>

Lists are simply a particular case of tree views.

A same object may have several views: the first defined view of a kind (*tree*, *form*, ...) will be used as the default view for this kind. That way you can have a default tree view (that will act as the view of a one2many) and a specialized view with more or less information that will appear when one double-clicks on a menu item. For example, the products have several views according to the product variants.

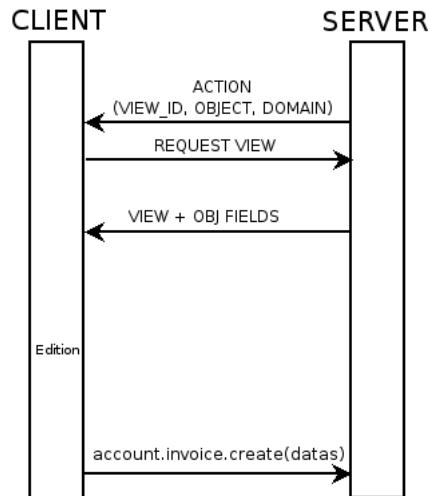
Views are described in XML.

If no view has been defined for an object, the object is able to generate a view to represent itself. This can limit the developer's work but results in less ergonomic views.

### Usage example

When you open an invoice, here is the chain of operations followed by the client:

- An action asks to open the invoice (it gives the object's data (`account.invoice`), the view, the domain (e.g. only unpaid invoices) ).
- The client asks (with XML-RPC) to the server what views are defined for the invoice object and what are the data it must show.
- The client displays the form according to the view



### To develop new objects

The design of new objects is restricted to the minimum: create the objects and optionally create the views to represent them. The PostgreSQL tables do not have to be written by hand because the objects are able to automatically create them (or adapt them in case they already exist).

### Reports

OpenERP uses a flexible and powerful reporting system. Reports are generated either in PDF or in HTML. Reports are designed on the principle of separation between the data layer and the presentation layer.

Reports are described more in details in the [Reporting chapter](#).

### Wizards

Here's an example of a .XML file that declares a wizard.

```

<?xml version="1.0"?>
<openerp>
    <data>
        <wizard string="Employee Info"
            model="hr.employee"
            name="employee.info.wizard"
            id="wizard_employee_info"/>
    </data>
</openerp>

```

A wizard is declared using a wizard tag. See “Add A New Wizard” for more information about wizard XML. also you can add wizard in menu using following xml entry

```

<?xml version="1.0"?>
</openerp>
    <data>
        <wizard string="Employee Info"
            model="hr.employee"
            name="employee.info.wizard"
            id="wizard_employee_info"/>
        <menuitem
            name="Human Resource/Employee Info"
            action="wizard_employee_info"
            type="wizard"
            id="menu_wizard_employee_info"/>
    </data>
</openerp>

```

## Workflow

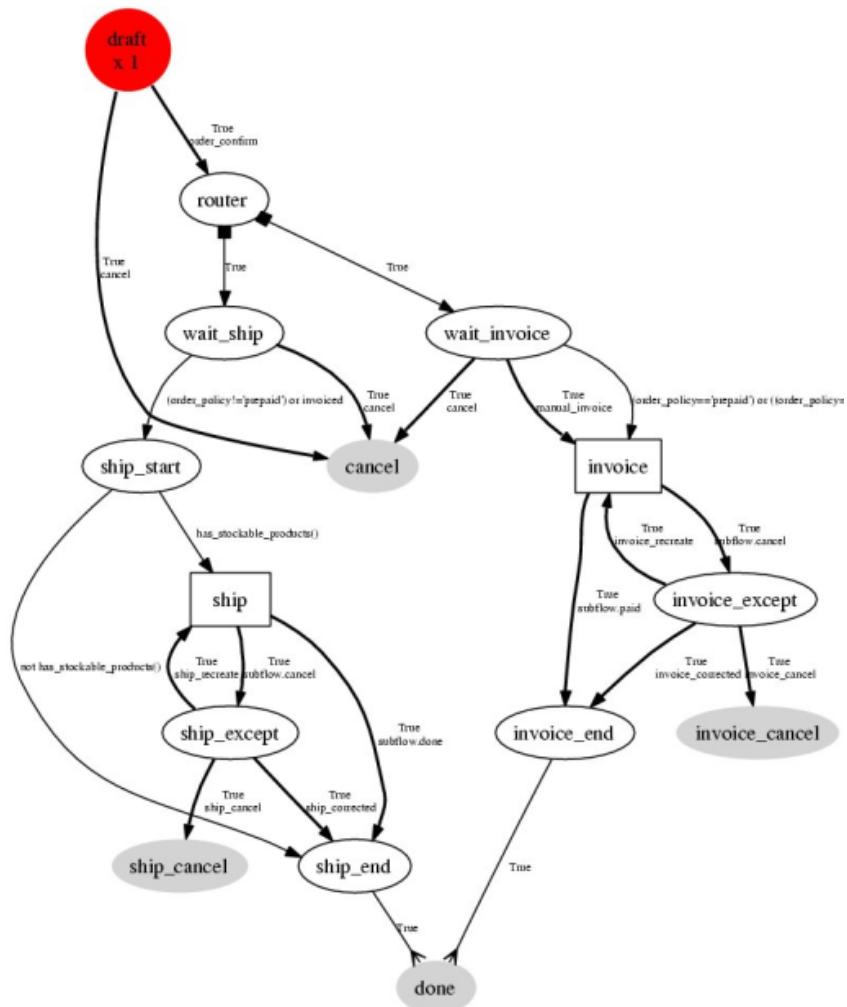
The objects and the views allow you to define new forms very simply, lists/trees and interactions between them. But that is not enough, you must define the dynamics of these objects.

A few examples:

- a confirmed sale order must generate an invoice, according to certain conditions
- a paid invoice must, only under certain conditions, start the shipping order

The workflows describe these interactions with graphs. One or several workflows may be associated to the objects. Workflows are not mandatory; some objects don't have workflows.

Below is an example workflow used for sale orders. It must generate invoices and shipments according to certain conditions.



In this graph, the nodes represent the actions to be done:

- create an invoice,
- cancel the sale order,
- generate the shipping order, ...

The arrows are the conditions;

- waiting for the order validation,
- invoice paid,
- click on the cancel button, ...

The squared nodes represent other Workflows;

- the invoice
- the shipping

## 12.3 OpenERP Module Descriptor File : `__openerp__.py`

### 12.3.1 Normal Module

In the created module directory, you must add a `__openerp__.py` file. This file, which must be in Python format, is responsible to

1. determine the XML files that will be parsed during the initialization of the server, and also to
2. determine the dependencies of the created module.

This file must contain a Python dictionary with the following values:

#### **name**

The (Plain English) name of the module.

#### **version**

The version of the module.

#### **description**

The module description (text).

#### **author**

The author of the module.

#### **website**

The website of the module.

#### **license**

The license of the module (default:GPL-2).

#### **depends**

List of modules on which this module depends. The base module must almost always be in the dependencies because some necessary data for the views, reports, ... are in the base module.

#### **init\_xml**

List of .xml files to load when the server is launched with the “–init=module” argument. Filepaths must be relative to the directory where the module is. OpenERP XML File Format is detailed in this section.

#### **update\_xml**

List of .xml files to load when the server is launched with the “–update=module” launched. Filepaths must be relative to the directory where the module is. OpenERP XML File Format is detailed in this section.

#### **installable**

True or False. Determines if the module is installable or not.

#### **active**

True or False (default: False). Determines the modules that are installed on the database creation.

### **Example**

Here is an example of \_\_openerp\_\_.py file for the *product* module:

```
{
    "name" : "Products & Pricelists",
    "version" : "1.1",
    "author" : "Open",
    "category" : "Generic Modules/Inventory Control",
    "depends" : ["base", "account"],
    "init_xml" : [],
    "demo_xml" : ["product_demo.xml"],
    "update_xml" : ["product_data.xml", "product_report.xml", "product_wizard.xml", "product_view.xml"],
    "installable": True,
```

```

    "active": True
}

```

The files that must be placed in init\_xml are the ones that relate to the workflow definition, data to load at the installation of the software and the data for the demonstrations.

The files in **update\_xml** concern: views, reports and wizards.

### 12.3.2 Profile Module

The purpose of a profile is to initialize OpenERP with a set of modules directly after the database has been created. A profile is a special kind of module that contains no code, only *dependencies on other modules*.

In order to create a profile, you only have to create a new directory in server/addons (you *should* call this folder profile\_modulename), in which you put an *empty \_\_init\_\_.py* file (as every directory Python imports must contain an *\_\_init\_\_.py* file), and a *\_\_openerp\_\_.py* whose structure is as follows :

```

{
    "name": "'Name of the Profile'",
    "version": "'Version String'",
    "author": "'Author Name'",
    "category": "Profile",
    "depends": ["List of the modules to install with the profile"],
    "demo_xml": [],
    "update_xml": [],
    "active": False,
    "installable": True,
}

```

#### Example

Here's the code of the file server/bin/addons/profile\_manufacturing/\_\_openerp\_\_.py, which corresponds to the manufacturing industry profile in OpenERP.

```

{
    "name": "Manufacturing industry profile",
    "version": "1.1",
    "author": "Open",
    "category": "Profile",
    "depends": ["mrp", "crm", "sale", "delivery"],
    "demo_xml": [],
    "update_xml": [],
    "active": False,
    "installable": True,
}

```

## 12.4 Module creation

### 12.4.1 Getting the skeleton directory

You can copy *\_\_openerp\_\_.py* and *\_\_init\_\_.py* from any other module to create a new module into a new directory.

As an example on Ubuntu:

```

$ cd ~/workspace/stable/stable_addons_5.0/
$ mkdir travel
$ sudo cp ~/workspace/stable/stable_addons_5.0/hr/__openerp__.py ~/workspace/stable/stable_addons_
sudo cp ~/workspace/stable/stable_addons_5.0/hr/__init__.py ~/workspace/stable/stable_addons_5.0/

```

You will need to give yourself permissions over that new directory if you want to be able to modify it:

```
$ sudo chown -R `whoami` travel
```

You got yourself the directory for a new module there, and a skeleton structure, but you still need to change a few things inside the module's definition...

## 12.4.2 Changing the default definition

To change the default settings of the “travel” module, get yourself into the “travel” directory and edit `__openerp__.py` (with `gedit`, for example, a simple text editor. Feel free to use another one)

```
$ cd travel
$ gedit __openerp__.py
```

The file looks like this:

```
{
    "name" : "Human Resources",
    "version" : "1.1",
    "author" : "Tiny",
    "category" : "Generic Modules/Human Resources",
    "website" : "http://www.openerp.com",
    "description": """
Module for human resource management. You can manage:
* Employees and hierarchies
* Work hours sheets
* Attendances and sign in/out system

Different reports are also provided, mainly for attendance statistics.

""",
    'author': 'Tiny',
    'website': 'http://www.openerp.com',
    'depends': ['base', 'process'],
    'init_xml': [],
    'update_xml': [
        'security/hr_security.xml',
        'security/ir.model.access.csv',
        'hr_view.xml',
        'hr_department_view.xml',
        'process/hr_process.xml'
    ],
    'demo_xml': ['hr_demo.xml', 'hr_department_demo.xml'],
    'installable': True,
    'active': False,
    'certificate': '0086710558965',
}
```

You will want to change whichever settings you feel right and get something like this:

```
{
    "name" : "Travel agency module",
    "version" : "1.1",
    "author" : "Tiny",
    "category" : "Generic Modules/Others",
    "website" : "http://www.openerp.com",
    "description": "A module to manage hotel bookings and a few other useful features.",
    "depends" : ["base"],
    "init_xml" : [],
    "update_xml" : ["travel_view.xml"],
    "active": True,
    "installable": True
}
```

Note the “active” field becomes true.

### 12.4.3 Changing the main module file

Now you need to update the travel.py script to suit the needs of your module. We suggest you follow the Flash tutorial for this or download the travel agency module from the 20 minutes tutorial page.

The documentation below is overlapping the two next step in this wiki tutorial, so just consider them as a help and head towards the next two pages first...

The travel.py file should initially look like this:

```
from osv import osv, fields

class travel_hostel(osv.osv):
    _name = 'travel.hostel'
    _inherit = 'res.partner'
    _columns = {
        'rooms_id': fields.one2many('travel.room', 'hostel_id', 'Rooms'),
        'quality': fields.char('Quality', size=16),
    }
    _defaults = {
    }
travel_hostel()
```

Ideally, you would copy that bunch of code several times to create all the entities you need (travel\_airport, travel\_room, travel\_flight). This is what will hold the database structure of your objects, but you don't really need to worry too much about the database side. Just filling this file will create the system structure for you when you install the module.

### 12.4.4 Customizing the view

You can now move on to editing the views. To do this, edit the custom\_view.xml file. It should first look like this:

```
<openerp>
<data>
    <record model="res.groups" id="group_compta_user">
        <field name="name">grcompta</field>
    </record>
    <record model="res.groups" id="group_compta_admin">
        <field name="name">grcomptaadmin</field>
    </record>
    <menuitem name="Administration" groups="admin,grcomptaadmin"
              icon="terp-stock" id="menu_admin_compta"/>
</data>
</openerp>
```

This is, as you can see, an example taken from an accounting system (French people call accounting “comptabilité”, which explains the compta bit).

Defining a view is defining the interfaces the user will get when accessing your module. Just defining a bunch of fields here should already get you started on a complete interface. However, due to the complexity of doing it right, we recommend, once again, that download the travel agency module example from this link <http://www.openerp.com/download/modules/5.0/>.

Next you should be able to create different views using other files to separate them from your basic/admin view.

## 12.5 Action creation

### 12.5.1 Linking events to action

The available type of events are:

- **client\_print\_multi** (print from a list or form)
- **client\_action\_multi** (action from a list or form)
- **tree\_but\_open** (double click on the item of a tree, like the menu)
- **tree\_but\_action** (action on the items of a tree)

To map an events to an action:

```
<record model="ir.values" id="ir_open_journal_period">
    <field name="key2">tree_but_open</field>
    <field name="model">account.journal.period</field>
    <field name="name">Open Journal</field>
    <field name="value" eval="'ir.actions.wizard,%d'%action_move_journal_line_form_select"/>
    <field name="object" eval="True"/>
</record>
```

If you double click on a journal/period (object: account.journal.period), this will open the selected wizard. (id="action\_move\_journal\_line\_form\_select").

You can use a res\_id field to allow this action only if the user click on a specific object.

```
<record model="ir.values" id="ir_open_journal_period">
    <field name="key2">tree_but_open</field>
    <field name="model">account.journal.period</field>
    <field name="name">Open Journal</field>
    <field name="value" eval="'ir.actions.wizard,%d'%action_move_journal_line_form_select"/>
    <field name="res_id" eval="3"/>
    <field name="object" eval="True"/>
</record>
```

The action will be triggered if the user clicks on the account.journal.period n°3.

When you declare wizard, report or menus, the ir.values creation is automatically made with these tags:

- <wizard... />
- <menuitem... />
- <report... />

So you usually do not need to add the mapping by yourself.



# OBJECTS, FIELDS AND METHODS

## 13.1 OpenERP Objects

All the ERP's pieces of data are accessible through "objects". As an example, there is a `res.partner` object to access the data concerning the partners, an `account.invoice` object for the data concerning the invoices, etc...

Please note that there is an object for every type of resource, and not an object per resource. We have thus a `res.partner` object to manage all the partners and not a `res.partner` object per partner. If we talk in "object oriented" terms, we could also say that there is an object per level.

The direct consequences is that all the methods of objects have a common parameter: the "ids" parameter. This specifies on which resources (for example, on which partner) the method must be applied. Precisely, this parameter contains a list of resource ids on which the method must be applied.

For example, if we have two partners with the identifiers 1 and 5, and we want to call the `res_partner` method "send\_email", we will write something like:

```
res_partner.send_email(..., [1, 5], ...)
```

We will see the exact syntax of object method calls further in this document.

In the following section, we will see how to define a new object. Then, we will check out the different methods of doing this.

For developers:

- OpenERP "objects" are usually called classes in object oriented programming.
- A OpenERP "resource" is usually called an object in OO programming, instance of a class.

It's a bit confusing when you try to program inside OpenERP, because the language used is Python, and Python is a fully object oriented language, and has objects and instances ...

Luckily, an OpenERP "resource" can be converted magically into a nice Python object using the "browse" class method (OpenERP object method).

## 13.2 The ORM - Object-relational mapping - Models

The ORM, short for Object-Relational Mapping, is a central part of OpenERP.

In OpenERP, the data model is described and manipulated through Python classes and objects. It is the ORM job to bridge the gap – as transparently as possible for the developer – between Python and the underlying relational database (PostgreSQL), which will provide the persistence we need for our objects.

## 13.3 OpenERP Object Attributes

### 13.3.1 Objects Introduction

To define a new object, you must define a new Python class then instantiate it. This class must inherit from the osv class in the osv module.

### 13.3.2 Object definition

The first line of the object definition will always be of the form:

```
class name_of_the_object(osv.osv):  
    _name = 'name.of.the.object'  
    _columns = { ... }  
    ...  
name_of_the_object()
```

An object is defined by declaring some fields with predefined names in the class. Two of them are required (\_name and \_columns), the rest are optional. The predefined fields are:

### 13.3.3 Predefined fields

**\_auto** Determines whether a corresponding PostgreSQL table must be generated automatically from the object. Setting \_auto to False can be useful in case of OpenERP objects generated from PostgreSQL views. See the “Reporting From PostgreSQL Views” section for more details.

**\_columns (required)** The object fields. See the [fields](#) section for further details.

**\_constraints** The constraints on the object. See the constraints section for details.

**\_sql\_constraints** The SQL Constraint on the object. See the SQL constraints section for further details.

**\_defaults** The default values for some of the object’s fields. See the default value section for details.

**\_inherit** The name of the osv object which the current object inherits from. See the [object inheritance section](#) (first form) for further details.

**\_inherits** The list of osv objects the object inherits from. This list must be given in a python dictionary of the form: {‘name\_of\_the\_parent\_object’: ‘name\_of\_the\_field’, ...}. See the [object inheritance section](#) (second form) for further details. Default value: {}.

**\_log\_access** Determines whether or not the write access to the resource must be logged. If true, four fields will be created in the SQL table: create\_uid, create\_date, write\_uid, write\_date. Those fields represent respectively the id of the user who created the record, the creation date of record, the id of the user who last modified the record, and the date of that last modification. This data may be obtained by using the perm\_read method.

**\_name (required)** Name of the object. Default value: None.

**\_order** Name of the fields used to sort the results of the search and read methods.

Default value: ‘id’.

Examples:

```
_order = "name"  
_order = "date_order desc"
```

**\_rec\_name** Name of the field in which the name of every resource is stored. Default value: ‘name’. Note: by default, the name\_get method simply returns the content of this field.

**\_sequence** Name of the SQL sequence that manages the ids for this object. Default value: None.

`_sql` SQL code executed upon creation of the object (only if `_auto` is True). It means this code gets executed after the table is created.

`_table` Name of the SQL table. Default value: the value of the `_name` field above with the dots (.) replaced by underscores (\_).

## 13.4 Object Inheritance - `_inherit`

### 13.4.1 Introduction

Objects may be inherited in some custom or specific modules. It is better to inherit an object to add/modify some fields.

It is done with:

```
_inherit='object.name'
```

### 13.4.2 Extension of an object

There are two possible ways to do this kind of inheritance. Both ways result in a new class of data, which holds parent fields and behaviour as well as additional fields and behaviour, but they differ in heavy programmatical consequences.

While Example 1 creates a new subclass “custom\_material” that may be “seen” or “used” by any view or tree which handles “network.material”, this will not be the case for Example 2.

This is due to the table (other.material) the new subclass is operating on, which will never be recognized by previous “network.material” views or trees.

Example 1:

```
class custom_material(osv.osv):
    _name = 'network.material'
    _inherit = 'network.material'
    _columns = {
        'manuf_warranty': fields.boolean('Manufacturer warranty?'),
    }
    _defaults = {
        'manuf_warranty': lambda *a: False,
    }
    custom_material()
```

#### Tip:

Notice

```
_name == _inherit
```

In this example, the ‘custom\_material’ will add a new field ‘manuf\_warranty’ to the object ‘network.material’. New instances of this class will be visible by views or trees operating on the superclasses table ‘network.material’.

This inheritance is usually called “class inheritance” in Object oriented design. The child inherits data (fields) and behavior (functions) of his parent.

Example 2:

```
class other_material(osv.osv):
    _name = 'other.material'
    _inherit = 'network.material'
    _columns = {
        'manuf_warranty': fields.boolean('Manufacturer warranty?'),
```

```

        }
        _defaults = {
            'manuf_warranty': lambda *a: False,
        }
    other_material()

```

**Tip:**

---

*Notice*  
`_name != _inherit`

---

In this example, the ‘other\_material’ will hold all fields specified by ‘network.material’ and it will additionally hold a new field ‘manuf\_warranty’. All those fields will be part of the table ‘other.material’. New instances of this class will therefore never been seen by views or trees operating on the superclasses table ‘network.material’.

This type of inheritance is known as “inheritance by prototyping” (e.g. Javascript), because the newly created subclass “copies” all fields from the specified superclass (prototype). The child inherits data (fields) and behavior (functions) of his parent.

## 13.5 Inheritance by Delegation - `_inherits`

**Syntax ::**

```

class tiny_object(osv.osv):
    _name = 'tiny.object'
    _table = 'tiny_object'
    _inherits = {
        'tiny.object_a': 'object_a_id',
        'tiny.object_b': 'object_b_id',
        ...
        'tiny.object_n': 'object_n_id'
    }
    (...)


```

The object ‘tiny.object’ inherits from all the columns and all the methods from the n objects ‘tiny.object\_a’, ..., ‘tiny.object\_n’.

To inherit from multiple tables, the technique consists in adding one column to the table `tiny_object` per inherited object. This column will store a foreign key (an id from another table). The values ‘`object_a_id`’ ‘`object_b_id`’ ... ‘`object_n_id`’ are of type string and determine the title of the columns in which the foreign keys from ‘`tiny.object_a`’, ..., ‘`tiny.object_n`’ are stored.

This inheritance mechanism is usually called “*instance inheritance*” or “*value inheritance*”. A resource (instance) has the VALUES of its parents.

## 13.6 Fields Introduction

Objects may contain different types of fields. Those types can be divided into three categories: simple types, relation types and functional fields. The simple types are integers, floats, booleans, strings, etc ... ; the relation types are used to represent relations between objects (one2one, one2many, many2one). Functional fields are special fields because they are not stored in the database but calculated in real time given other fields of the view.

Here’s the header of the initialization method of the class any field defined in OpenERP inherits (as you can see in `server/bin/osv/fields.py`):

```

def __init__(self, string='unknown', required=False, readonly=False,
            domain=None, context="", states=None, priority=0, change_default=False, size=None,
            ondelete="set null", translate=False, select=False, **args) :

```

There are a common set of optional parameters that are available to most field types:

**change\_default** Whether or not the user can define default values on other fields depending on the value of this field. Those default values need to be defined in the ir.values table.

**help** A description of how the field should be used: longer and more descriptive than *string*. It will appear in a tooltip when the mouse hovers over the field.

**onDelete** How to handle deletions in a related record. Allowable values are: ‘restrict’, ‘no action’, ‘cascade’, ‘set null’, and ‘set default’.

**priority** Not used?

**readonly** *True* if the user cannot edit this field, otherwise *False*.

**required** *True* if this field must have a value before the object can be saved, otherwise *False*.

**size** The size of the field in the database: number characters or digits.

**states** Lets you override other parameters for specific states of this object. Accepts a dictionary with the state names as keys and a list of name/value tuples as the values. For example: `states={'posted':[('readonly',True)]}`

**string** The field name as it should appear in a label or column header. Strings containing non-ASCII characters must use python unicode objects. For example: `'tested': fields.boolean(u'Testé')`

**translate** *True* if the *content* of this field should be translated, otherwise *False*.

There are also some optional parameters that are specific to some field types:

**context** Define a variable’s value visible in the view’s context or an on-change function. Used when searching child table of *one2many* relationship?

**domain** Domain restriction on a relational field.

Default value: [].

Example: `domain=[('field','=',value)]`

**invisible** Hide the field’s value in forms. For example, a password.

**on\_change** Default value for the *on\_change* attribute in the view. This will launch a function on the server when the field changes in the client. For example, `on_change="onchange_shop_id(shop_id)"`.

**relation** Used when a field is an id reference to another table. This is the name of the table to look in. Most commonly used with related and function field types.

**select** Default value for the *select* attribute in the view. 1 means basic search, and 2 means advanced search.

## 13.7 Type of Fields

### 13.7.1 Basic Types

**boolean** A boolean (true, false).

Syntax:

```
fields.boolean('Field Name' [, Optional Parameters]),
```

**integer** An integer.

Syntax:

```
fields.integer('Field Name' [, Optional Parameters]),
```

**float** A floating point number.

Syntax:

```
fields.float('Field Name' [, Optional Parameters]),
```

**Note:**

*The optional parameter digits defines the precision and scale of the number. The scale being the number of digits after the decimal point whereas the precision is the total number of significant digits in the number (before and after the decimal point). If the parameter digits is not present, the number will be a double precision floating point number. Warning: these floating-point numbers are inexact (not any value can be converted to its binary representation) and this can lead to rounding errors. You should always use the digits parameter for monetary amounts.*

Example:

```
'rate': fields.float(
    'Relative Change rate',
    digits=(12, 6) [,,
    Optional Parameters]),
```

**char** A string of limited length. The required size parameter determines its size.

Syntax:

```
fields.char(
    'Field Name',
    size=n [,,
    Optional Parameters]), # where ''n'' is an integer.
```

Example:

```
'city' : fields.char('City Name', size=30, required=True),
```

**text** A text field with no limit in length.

Syntax:

```
fields.text('Field Name' [, Optional Parameters]),
```

**date** A date.

Syntax:

```
fields.date('Field Name' [, Optional Parameters]),
```

**datetime** Allows to store a date and the time of day in the same field.

Syntax:

```
fields.datetime('Field Name' [, Optional Parameters]),
```

**binary** A binary chain

**selection** A field which allows the user to make a selection between various predefined values.

Syntax:

```
fields.selection([('n','Unconfirmed'), ('c','Confirmed')),
    'Field Name' [, Optional Parameters]),
```

**Note:**


---

*Format of the selection parameter: tuple of tuples of strings of the form:*

```
(('key_or_value', 'string_to_display'), ... )
```

---

**Note:**


---

*You can specify a function that will return the tuple. Example*

```
def _get_selection(self, cursor, user_id, context=None):
    return (
        ('choice1', 'This is the choice 1'),
        ('choice2', 'This is the choice 2'))

_columns = {
    'sel' : fields.selection(
        _get_selection,
        'What do you want ?')
}
```

---

*Example*

Using relation fields **many2one** with **selection**. In fields definitions add:

```
...
'my_field': fields.many2one(
    'mymodule.relation.model',
    'Title',
    selection=_sel_func),
...,
```

And then define the `_sel_func` like this (but before the fields definitions):

```
def _sel_func(self, cr, uid, context=None):
    obj = self.pool.get('mymodule.relation.model')
    ids = obj.search(cr, uid, [])
    res = obj.read(cr, uid, ids, ['name', 'id'], context)
    res = [(r['id'], r['name']) for r in res]
    return res
```

### 13.7.2 Relational Types

**one2one** A one2one field expresses a one:to:one relation between two objects. It is deprecated. Use many2one instead.

Syntax:

```
fields.one2one('other.object.name', 'Field Name')
```

**many2one** Associates this object to a parent object via this Field. For example Department an Employee belongs to would Many to one. i.e Many employees will belong to a Department

Syntax:

```
fields.many2one(
    'other.object.name',
    'Field Name',
    optional parameters)
```

Optional parameters:

- **ondelete:** What should happen when the resource this field points to is deleted.

- Predefined value: “cascade”, “set null”, “restrict”, “no action”, “set default”
- Default value: “set null”
- required: True
- readonly: True
- select: True - (creates an index on the Foreign Key field)

*Example*

```
'commercial': fields.many2one(
    'res.users',
    'Commercial',
    ondelete='cascade'),
```

## one2many TODO

Syntax:

```
fields.one2many(
    'other.object.name',
    'Field relation id',
    'Fieldname',
    optional parameter)
```

### Optional parameters:

- invisible: True/False
- states: ?
- readonly: True/False

*Example*

```
'address': fields.one2many(
    'res.partner.address',
    'partner_id',
    'Contacts'),
```

## many2many TODO

Syntax:

```
fields.many2many('other.object.name',
    'relation object',
    'actual.object.id',
    'other.object.id',
    'Field Name')
```

### Where:

- other.object.name is the other object which belongs to the relation
- relation object is the table that makes the link
- actual.object.id and other.object.id are the fields' names used in the relation table

Example:

```
'category_ids':
    fields.many2many(
        'res.partner.category',
        'res_partner_category_rel',
        'partner_id',
        'category_id',
        'Categories'),
```

To make it bidirectional (= create a field in the other object):

```
class other_object_name2(osv.osv):
    _inherit = 'other.object.name'
    _columns = {
        'other_fields': fields.many2many(
            'actual.object.name',
            'relation object',
            'actual.object.id',
            'other.object.id',
            'Other Field Name'),
    }
other_object_name2()
```

Example:

```
class res_partner_category2(osv.osv):
    _inherit = 'res.partner.category'
    _columns = {
        'partner_ids': fields.many2many(
            'res.partner',
            'res_partner_category_rel',
            'category_id',
            'partner_id',
            'Partners'),
    }
res_partner_category2()
```

**related** Sometimes you need to refer to the relation of a relation. For example, supposing you have objects: City -> State -> Country, and you need to refer to the Country from a City, you can define a field as below in the City object:

```
'country_id': fields.related(
    'state_id',
    'country_id',
    type="many2one",
    relation="res.country",
    string="Country",
    store=False)
```

**Where:**

- The first set of parameters are the chain of reference fields to follow, with the desired field at the end.
- *type* is the type of that desired field.
- Use *relation* if the desired field is still some kind of reference. *relation* is the table to look up that reference in.

### 13.7.3 Functional Fields

A functional field is a field whose value is calculated by a function (rather than being stored in the database).

**Parameters:**

```
fnct, arg=None, fnct_inv=None, fnct_inv_arg=None, type="float",
fnct_search=None, obj=None, method=False, store=False, multi=False
```

where

- *fnct* is the function or method that will compute the field value. It must have been declared before declaring the functional field.
- *fnct\_inv* is the function or method that will allow writing values in that field.

- *type* is the field type name returned by the function. It can be any field type name except function.
- *fnct\_search* allows you to define the searching behaviour on that field.
- *method* whether the field is computed by a method (of an object) or a global function
- *store* If you want to store field in database or not. Default is False.
- *multi* is a group name. All fields with the same *multi* parameter will be calculated in a single function call.

## **fnct parameter**

If *method* is True, the signature of the method must be:

```
def fnct(self, cr, uid, ids, field_name, arg, context):
```

otherwise (if it is a global function), its signature must be:

```
def fnct(cr, table, ids, field_name, arg, context):
```

Either way, it must return a dictionary of values of the form **{id'\_1\_': value'\_1\_ ', id'\_2\_ ': value'\_2\_ ',...}**.

The values of the returned dictionary must be of the type specified by the type argument in the field declaration.

If *multi* is set, then *field\_name* is replaced by *field\_names*: a list of the field names that should be calculated. Each value in the returned dictionary is also a dictionary from field name to value. For example, if the fields ‘name’, and ‘age’ are both based on the *vital\_statistics* function, then the return value of *vital\_statistics* might look like this when *ids* is [1, 2, 5]:

```
{
    1: {'name': 'Bob', 'age': 23},
    2: {'name': 'Sally', 'age': 19},
    5: {'name': 'Ed', 'age': 62}
}
```

## **fnct\_inv parameter**

If *method* is true, the signature of the method must be:

```
def fnct(self, cr, uid, ids, field_name, field_value, arg, context):
```

otherwise (if it is a global function), it should be:

```
def fnct(cr, table, ids, field_name, field_value, arg, context):
```

## **fnct\_search parameter**

If *method* is true, the signature of the method must be:

```
def fnct(self, cr, uid, obj, name, args, context):
```

otherwise (if it is a global function), it should be:

```
def fnct(cr, uid, obj, name, args, context):
```

The return value is a list containing 3-part tuples which are used in search function:

```
return [('id', 'in', [1, 3, 5])]
```

*obj* is the same as *self*, and *name* receives the field name. *args* is a list of 3-part tuples containing search criteria for this field, although the search function may be called separately for each tuple.

## Example

Suppose we create a contract object which is :

```
class hr_contract(osv.osv):
    _name = 'hr.contract'
    _description = 'Contract'
    _columns = {
        'name' : fields.char('Contract Name', size=30, required=True),
        'employee_id' : fields.many2one('hr.employee', 'Employee', required=True),
        'function' : fields.many2one('res.partner.function', 'Function'),
    }
hr_contract()
```

If we want to add a field that retrieves the function of an employee by looking its current contract, we use a functional field. The object hr\_employee is inherited this way:

```
class hr_employee(osv.osv):
    _name = "hr.employee"
    _description = "Employee"
    _inherit = "hr.employee"
    _columns = {
        'contract_ids' : fields.one2many('hr.contract', 'employee_id', 'Contracts'),
        'function' : fields.function(
            _get_cur_function_id,
            type='many2one',
            obj="res.partner.function",
            method=True,
            string='Contract Function'),
    }
hr_employee()
```

### Note:

three points

- type = 'many2one' is because the function field must create a many2one field; function is declared as a many2one in hr\_contract also.
- obj = "res.partner.function" is used to specify that the object to use for the many2one field is res.partner.function.
- We called our method \_get\_cur\_function\_id because its role is to return a dictionary whose keys are ids of employees, and whose corresponding values are ids of the function of those employees. The code of this method is:

```
def _get_cur_function_id(self, cr, uid, ids, field_name, arg, context):
    for i in ids:
        #get the id of the current function of the employee of identifier "i"
        sql_req= """
        SELECT f.id AS func_id
        FROM hr_contract c
        LEFT JOIN res_partner_function f ON (f.id = c.function)
        WHERE
            (c.employee_id = %d)
        """ % (i,)

        cr.execute(sql_req)
        sql_res = cr.dictfetchone()

        if sql_res: #The employee has one associated contract
            res[i] = sql_res['func_id']
        else:
            #res[i] must be set to False and not to None because of XML:RPC
            # "cannot marshal None unless allow_none is enabled"
```

```

        res[i] = False
    return res

```

The id of the function is retrieved using a SQL query. Note that if the query returns no result, the value of `sql_res['func_id']` will be None. We force the False value in this case value because XML:RPC (communication between the server and the client) doesn't allow to transmit this value.

### store Parameter

It will calculate the field and store the result in the table. The field will be recalculated when certain fields are changed on other objects. It uses the following syntax:

```

store = {
    'object_name': (
        function_name,
        ['field_name1', 'field_name2'],
        priority)
}

```

It will call function `function_name` when any changes are written to fields in the list `['field1','field2']` on object `'object_name'`. The function should have the following signature:

```
def function_name(self, cr, uid, ids, context=None):
```

Where `ids` will be the ids of records in the other object's table that have changed values in the watched fields. The function should return a list of ids of records in its own table that should have the field recalculated. That list will be sent as a parameter for the main function of the field.

Here's an example from the membership module:

```

'membership_state':
    fields.function(
        _membership_state,
        method=True,
        string='Current membership state',
        type='selection',
        selection=STATE,
        store={
            'account.invoice': (_get_invoice_partner, ['state'], 10),
            'membership.membership_line': (_get_partner_id, ['state'], 10),
            'res.partner': (
                lambda self, cr, uid, ids, c={}: ids,
                ['free_member'],
                10)
        },
)

```

## 13.7.4 Property Fields

### Declaring a property

A property is a special field: `fields.property`.

```

class res_partner(osv.osv):
    _name = "res.partner"
    _inherit = "res.partner"
    _columns = {
        'property_product_pricelist':
            fields.property(
                'product.pricelist',
                type='many2one',
                relation='product.pricelist',
)

```

```

        string="Sale Pricelist",
        method=True,
        view_load=True,
        group_name="Pricelists Properties"),
}

```

Then you have to create the default value in a .XML file for this property:

```

<record model="ir.property" id="property_product_pricelist">
    <field name="name">property_product_pricelist</field>
    <field name="fields_id" search="[(('model','=','res.partner'),
        ('name','=','property_product_pricelist'))]">
        <field name="value" eval="'product.pricelist,'+str(list0)"/>
    </field>
</record>

```

#### Tip:

*if the default value points to a resource from another module, you can use the ref function like this:*

```

<field name="value" eval="‘product.pricelist,’+str(ref(‘module.data_id’))”/>

```

### Putting properties in forms

To add properties in forms, just put the <properties/> tag in your form. This will automatically add all properties fields that are related to this object. The system will add properties depending on your rights. (some people will be able to change a specific property, others won't).

Properties are displayed by section, depending on the group\_name attribute. (It is rendered in the client like a separator tag).

#### How does this work ?

The fields.property class inherits from fields.function and overrides the read and write method. The type of this field is many2one, so in the form a property is represented like a many2one function.

But the value of a property is stored in the ir.property class/table as a complete record. The stored value is a field of type reference (not many2one) because each property may point to a different object. If you edit properties values (from the administration menu), these are represented like a field of type reference.

When you read a property, the program gives you the property attached to the instance of object you are reading. If this object has no value, the system will give you the default property.

The definition of a property is stored in the ir.model.fields class like any other fields. In the definition of the property, you can add groups that are allowed to change to property.

#### Using properties or normal fields

When you want to add a new feature, you will have to choose to implement it as a property or as normal field. Use a normal field when you inherit from an object and want to extend this object. Use a property when the new feature is not related to the object but to an external concept.

Here are a few tips to help you choose between a normal field or a property:

Normal fields extend the object, adding more features or data.

A property is a concept that is attached to an object and have special features:

- Different value for the same property depending on the company
- Rights management per field
- It's a link between resources (many2one)

#### Example 1: Account Receivable

The default “Account Receivable” for a specific partner is implemented as a property because:

- This is a concept related to the account chart and not to the partner, so it is an account property that is visible on a partner form. Rights have to be managed on this fields for accountants, these are not the same rights that are applied to partner objects. So you have specific rights just for this field of the partner form: only accountants may change the account receivable of a partner.
- This is a multi-company field: the same partner may have different account receivable values depending on the company the user belongs to. In a multi-company system, there is one account chart per company. The account receivable of a partner depends on the company it placed the sale order.
- The default account receivable is the same for all partners and is configured from the general property menu (in administration).

**Note:**

*One interesting thing is that properties avoid “spaghetti” code. The account module depends on the partner (base) module. But you can install the partner (base) module without the accounting module. If you add a field that points to an account in the partner object, both objects will depend on each other. It’s much more difficult to maintain and code (for instance, try to remove a table when both tables are pointing to each others.)*

### Example 2: Product Times

The product expiry module implements all delays related to products: removal date, product usetime, ... This module is very useful for food industries.

This module inherits from the product.product object and adds new fields to it:

```
class product_product(osv.osv):
    _inherit = 'product.product'
    _name = 'product.product'
    _columns = {
        'life_time': fields.integer('Product lifetime'),
        'use_time': fields.integer('Product usetime'),
        'removal_time': fields.integer('Product removal time'),
        'alert_time': fields.integer('Product alert time'),
    }
product_product()
```

This module adds simple fields to the product.product object. We did not use properties because:

- We extend a product, the life\_time field is a concept related to a product, not to another object.
- We do not need a right management per field, the different delays are managed by the same people that manage all products.

## 13.8 ORM methods

### 13.8.1 Keeping the context in ORM methods

In OpenObject, the context holds very important data such as the language in which a document must be written, whether function field needs updating or not, etc.

When calling an ORM method, you will probably already have a context - for example the framework will provide you with one as a parameter of almost every method. If you do have a context, it is very important that you always pass it through to every single method you call.

This rule also applies to writing ORM methods. You should expect to receive a context as parameter, and always pass it through to every other method you call..

### 13.8.2 ORM methods



# VIEWS AND EVENTS

## 14.1 Introduction to Views

As all program data is stored in objects, as explained in the Objects section, how are these objects exposed to the user ? We will try to answer this question in this section.

First of all, let's note that every resource type uses its own interface. For example, the screen to modify a partner's data is not the same as the one to modify an invoice.

Then, you should know that the OpenERP user interface is dynamic, which means it is not described "statically" by some code, but is dynamically built from XML descriptions of the client screens.

From now on, we will call these screen descriptions views.

A notable characteristic of these views is that they can be edited at any time (even during program execution). After modifying a displayed view you simply need to close the tab corresponding to that 'view' and re-open it for the changes to appear.

### 14.1.1 Views principles

Views describe how each object (type of resource) is displayed. More precisely, for each object, we can define one (or several) view(s) to describe which fields should be drawn and how.

There are two types of views:

1. form views
2. tree views

**Note:**

*Since OpenERP 4.1, form views can also contain graphs.*

## 14.2 Form views

The field disposition in a form view always follows the same principle. Fields are distributed on the screen following the rules below:

- By default, each field is preceded by a label, with its name.
- Fields are placed on the screen from left to right, and from top to bottom, according to the order in which they are declared in the view.
- Every screen is divided into 4 columns, each column being able to contain either a label, or an "edition" field. As every edition field is preceded (by default) by a label with its name, there will be two fields

(and their respective labels) on each line of the screen. The green and red zones on the screen-shot below illustrate those 4 columns. They designate respectively the labels and their corresponding fields.

This screenshot shows a sales order form in the OpenERP application. The 'Customer' field is highlighted with a green border, indicating it is a label column. The 'Order Reference' and 'Shop' fields are highlighted with a red border, indicating they are value columns. The rest of the fields, such as 'Invoice Address', 'Pricelist', and 'Sales order lines', are standard input fields.

Views also support more advanced placement options:

- A view field can use several columns. For example, on the screen-shot below, the zone in the blue frame is, in fact, the only field of a “one to many”. We will come back later on this note, but let’s note that it uses the whole width of the screen and not only one column.

This screenshot shows a sales order form in the OpenERP application. The 'Sales order lines' section is enclosed in a blue frame, indicating it is a multi-column view field. This allows for a more complex and dynamic display of data compared to a single column view.

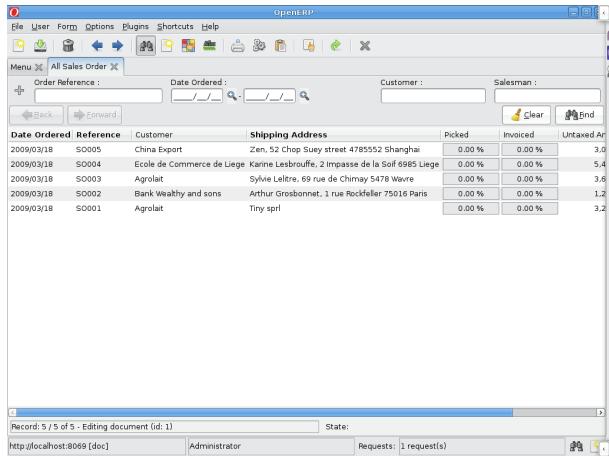
- We can also make the opposite operation: take a columns group and divide it in as many columns as desired. The surrounded green zones of the screen above are good examples. Precisely, the green framework up and on the right side takes the place of two columns, but contains 4 columns.

As we can see below in the purple zone of the screen, there is also a way to distribute the fields of an object on different tabs.

This screenshot shows a sales order form in the OpenERP application. The 'Sales order' tab is selected, and the 'Other data' tab is visible below it, demonstrating a tabbed interface for distributing fields. This allows users to switch between different sets of fields without having to scroll through them all at once.

## 14.3 Tree views

These views are used when we work in list mode (in order to visualise several resources at once) and in the search screen. These views are simpler than the form views and thus have less options.



## 14.4 Graph views

A graph is a new mode of view for all views of type form. If, for example, a sale order line must be visible as list or as graph, define it like this in the action that opens this sale order line. Do not set the view mode as “tree,form,graph” or “form,graph” - it must be “graph,tree” to show the graph first or “tree,graph” to show the list first. (This view mode is extra to your “form,tree” view and should have a separate menu item):

```
<field name="view_type">form</field>
<field name="view_mode">tree,graph</field>
```

view\_type:

```
tree = (tree with shortcuts at the left), form = (switchable view form/list)
```

view\_mode:

```
tree,graph : sequences of the views when switching
```

Then, the user will be able to switch from one view to the other. Unlike forms and trees, OpenERP is not able to automatically create a view on demand for the graph type. So, you must define a view for this graph:

```
<record model="ir.ui.view" id="view_order_line_graph">
    <field name="name">sale.order.line.graph</field>
    <field name="model">sale.order.line</field>
    <field name="type">graph</field>
    <field name="arch" type="xml">
        <graph string="Sales Order Lines">
            <field name="product_id" group="True"/>
            <field name="price_unit" operator="*"/>
        </graph>
    </field>
</record>
```

The graph view

A view of type graph is just a list of fields for the graph.

### 14.4.1 Graph tag

The default type of the graph is a pie chart - to change it to a barchart change `<graph string="Sales Order Lines">` to `<graph string="Sales Order Lines" type="bar">` You also may change the orientation.

:Example :

```
<graph string="Sales Order Lines" orientation="horizontal" type="bar">
```

### 14.4.2 Field tag

The first field is the X axis. The second one is the Y axis and the optional third one is the Z axis for 3 dimensional graphs. You can apply a few attributes to each field/axis:

- **group**: if set to true, the client will group all item of the same value for this field. For every other field, it will apply an operator
- **operator**: the operator to apply if another field is grouped. By default it is ‘+’. Allowed values are:
  - +: addition
  - \*: multiply
  - \*\*: exponent
  - min: minimum of the list
  - max: maximum of the list

#### Defining real statistics on objects

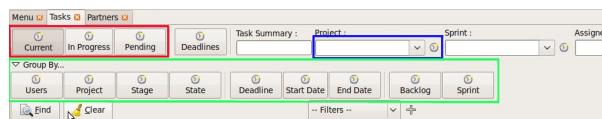
The easiest method to compute real statistics on objects is:

1. Define a statistic object which is a postgresql view
2. Create a tree view and a graph view on this object

You can get an example in all modules of the form: report\_.... Example: report\_crm.

## 14.5 Search views

Search views are a new feature of OpenERP supported as of version 6.0. It creates a customized search panel, and is declared quite similarly to a form view, except that the view type and root element change to `search` instead of `form`.



Following is the list of new elements and features supported in search views.

### 14.5.1 Group tag

Unlike form group elements, search view groups support unlimited number of widgets (fields or filters) in a row (no automatic line wrapping), and only use the following attributes:

- **expand**: turns on the expander icon on the group (1 for expanded by default, 0 for collapsed)
- **string**: label for the group

```

<group expand="1" string="Group By...">
  <filter string="Users" icon="terp-project" domain="[]" context="{'group_by':'user_id'}"/>
  <filter string="Project" icon="terp-project" domain="[]" context="{'group_by':'project_id'}"/>
  <separator orientation="vertical"/>
  <filter string="Deadline" icon="terp-project" domain="[]" context="{'group_by':'date_deadline'}/>
</group>

```

In the screenshot above the green area is an expandable group.

### 14.5.2 Filter tag

Filters are displayed as a toggle button on search panel Filter elements can add new values in the current domain or context of the search view. Filters can be added as a child element of field too, to indicate that they apply specifically to that field (in this case the button's icon will smaller)

In the picture above the red area contains filters at the top of the form while the blue area highlights a field and its child filter.

```

<filter string="Current" domain="['state','in',('open','draft'))]" help="Draft, Open and Pending">
<field name="project_id" select="1" widget="selection">
  <filter domain="['project_id.user_id','=',uid]" help="My Projects" icon="terp-project"/>
</field>

```

### 14.5.3 Group By

```
<filter string="Project" icon="terp-project" domain="[]" context="{'group_by':'project_id'}/>
```

Above filters groups records sharing the same project\_id value. Groups are loaded lazily, so the inner records are only loaded when the group is expanded. The group header lines contain the common values for all records in that group, and all numeric fields currently displayed in the view are replaced by the sum of the values in that group.

It is also possible to group on multiple values by specifying a list of fields instead of a single string. In this case nested groups will be displayed:

```
<filter string="Project" icon="terp-project" domain="[]" context="{'group_by': ['project_id', 'use'...]
```

### 14.5.4 Fields

Field elements in search views are used to get user-provided values for searches. As a result, as for group elements, they are quite different than form view's fields:

- a search field can contain filters, which generally indicate that both field and filter manage the same field and are related.

Those inner filters are rendered as smaller buttons, right next to the field, and *must not* have a string attribute.

- a search field really builds a domain composed of [(field\_name, operator, field\_value)]. This domain can be overridden in two ways:
  - @operator replaces the default operator for the field (which depends on its type)
  - @filter\_domain lets you provide a fully custom domain, which will replace the default domain creation
- a search field does not create a context by default, but you can provide an @context which will be evaluated and merged into the wider context (as with a filter element).

To get the value of the field in your @context or @filter\_domain, you can use the variable self:

```
<field name="location_id" string="Location"
      filter_domain="['|', ('location_id','ilike',self), ('location_dest_id','ilike',self)]"/>
```

or

```
<field name="journal_id" widget="selection"
      context="{'journal_id':self, 'visible_id':self, 'normal_view':False}"/>
```

### Range fields (date, datetime, time)

The range fields are composed of two input widgets (from and to) instead of just one.

This leads to peculiarities (compared to non-range search fields):

- It is not possible to override the operator of a range field via `@operator`, as the domain is built of two sections and each section uses a different operator.
- Instead of being a simple value (integer, string, float) `self` for use in `@filter_domain` and `@context` is a `dict`.

Because each input widget of a range field can be empty (and the field itself will still be valid), care must be taken when using `self`: it has two string keys "from" and "to", but any of these keys can be either missing entirely or set to the value `False`.

### 14.5.5 Actions for Search view

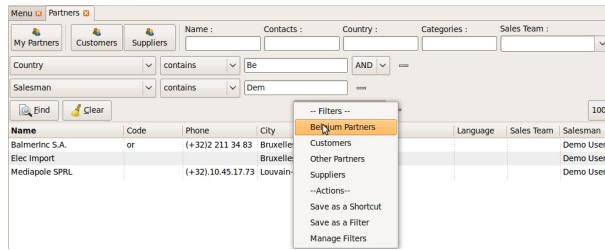
After declaring a search view, it will be used automatically for all tree views on the same model. If several search views exist for a single model, the one with the highest priority (lowest sequence) will be used. Another option is to explicitly select the search view you want to use, by setting the `search_view_id` field of the action.

In addition to being able to pass default form values in the context of the action, OpenERP 6.0 now supports passing initial values for search views too, via the context. The context keys need to match the `search_default_XXX` format. `XXX` may refer to the name of a `<field>` or `<filter>` in the search view (as the `name` attribute is not required on filters, this only works for filters that have an explicit `name` set). The value should be either the initial value for search fields, or simply a boolean value for filters, to toggle them

```
<record id="action_view_task" model="ir.actions.act_window">
  <field name="name">Tasks</field>
  <field name="res_model">project.task</field>
  <field name="view_type">form</field>
  <field name="view_mode">tree,form,calendar,gantt,graph</field>
  <field eval="False" name="filter"/>
  <field name="view_id" ref="view_task_tree2"/>
  <field name="context">{"search_default_current":1,"search_default_user_id":uid}</field>
  <field name="search_view_id" ref="view_task_search_form"/>
</record>
```

### 14.5.6 Custom Filters

As of v6.0, all search views feature custom search filters, as shown below. Users can define their own custom filters using any of the fields available on the current model, combining them with AND/OR operators. It is also possible to save any search context (the combination of all currently applied domain and context values) as a personal filter, which can be recalled at any time. Filters can also be turned into Shortcuts directly available in the User's homepage.



In above screenshot we filter Partner where Salesman = Demo user and Country = Belgium, We can save this search criteria as a Shortcut or save as Filter.

Filters are user specific and can be modified via the Manage Filters option in the filters drop-down.

## 14.6 Calendar Views

Calendar view provides timeline/schedule view for the data.

### 14.6.1 View Specification

Here is an example view:

```
<calendar color="user_id" date_delay="planned_hours" date_start="date_start" string="Tasks">
    <field name="name"/>
    <field name="project_id"/>
</calendar>
```

Here is the list of supported attributes for `calendar` tag:

**string** The title string for the view.

**date\_start** A `datetime` field to specify the starting date for the calendar item. This attribute is required.

**date\_stop** A `datetime` field to specify the end date. Ignored if `date_delay` attribute is specified.

**date\_delay** A numeric field to specify time in hours for a record. This attribute will get preference over `date_stop` and `date_stop` will be ignored.

**day\_length** An integer value to specify working day length. Default is 8 hours.

**color** A field, generally many2one, to colourise calendar/gantt items.

**mode** A string value to set default view/zoom mode. For calendar view, this can be one of following (default is `month`):

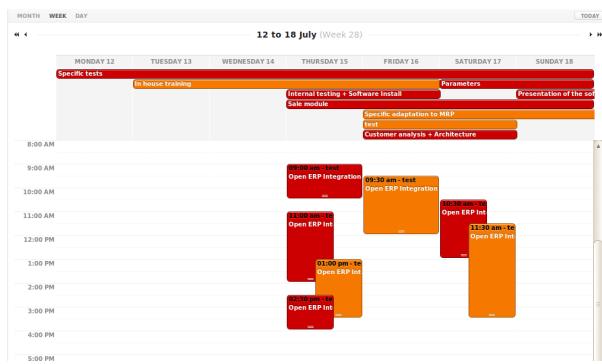
- day
- week
- month

### 14.6.2 Screenshots

Month Calendar:



Week Calendar:



## 14.7 Gantt Views

Gantt view provides timeline view for the data. Generally, it can be used to display project tasks and resource allocation.

A Gantt chart is a graphical display of all the tasks that a project is composed of. Each bar on the chart is a graphical representation of the length of time the task is planned to take.

A resource allocation summary bar is shown on top of all the grouped tasks, representing how effectively the resources are allocated among the tasks.

Color coding of the summary bar is as follows:

- *Gray* shows that the resource is not allocated to any task at that time
- *Blue* shows that the resource is fully allocated at that time.
- *Red* shows that the resource is overallocated

### 14.7.1 View Specification

Here is an example view:

```
<gantt color="user_id" date_delay="planned_hours" date_start="date_start" string="Tasks">
    <level object="project.project" link="project_id" domain="[] ">
        <field name="name"/>
    </level>
</gantt>
```

The attributes accepted by the gantt tag are similar to calendar view tag. The level tag is used to group the records by some many2one field. Currently, only one level is supported.

Here is the list of supported attributes for gantt tag:

**string** The title string for the view.

**date\_start** A datetime field to specify the starting date for the gantt item. This attribute is required.

**date\_stop** A datetime field to specify the end date. Ignored if date\_delay attribute is specified.

**date\_delay** A numeric field to specify time in hours for a record. This attribute will get preference over date\_stop and date\_stop will be ignored.

**day\_length** An integer value to specify working day length. Default is 8 hours.

**color** A field, generally many2one, to colorize calendar/gantt items.

**mode** A string value to set default view/zoom mode. For gantt view, this can be one of following (default is month):

- day
- 3days
- week
- 3weeks
- month
- 3months
- year
- 3years
- 5years

The level tag supports following attributes:

**object** An openerp object having many2one relationship with view object.

**link** The field name in current object that links to the given object.

**domain** The domain to be used to filter the given object records.

## 14.7.2 Drag and Drop

The left side pane displays list of the tasks grouped by the given level field. You can reorder or change the group of any records by dragging them.

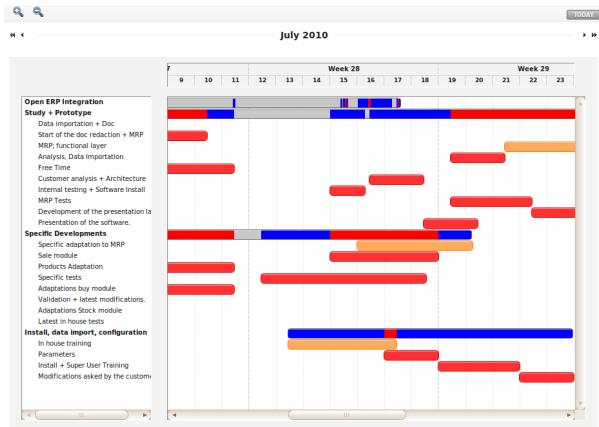
The main content pane displays horizontal bars plotted on a timeline grid. A group of bars are summarised with a top summary bar displaying resource allocation of all the underlying tasks.

You can change the task start time by dragging the tasks horizontally. While end time can be changed by dragging right end of a bar.

**Note:**

*The time is calculated considering day\_length so a bar will span more than one day if total time for a task is greater than day\_length value.*

### 14.7.3 Screenshots



## 14.8 Design Elements

The files describing the views are of the form:

### Example

```
<?xml version="1.0"?>
<openerp>
    <data>
        [view definitions]
    </data>
</openerp>
```

The view definitions contain mainly three types of tags:

- **<record>** tags with the attribute `model="ir.ui.view"`, which contain the view definitions themselves
- **<record>** tags with the attribute `model="ir.actions.act_window"`, which link actions to these views
- **<menuitem>** tags, which create entries in the menu, and link them with actions

New : You can specify groups for whom the menu is accessible using the `groups` attribute in the `menuitem` tag.

New : You can now add shortcut using the `shortcut` tag.

### Example

```
<shortcut
    name="Draft Purchase Order (Proposals)"
    model="purchase.order"
    login="demo"
    menu="m"/>
```

Note that you should add an `id` attribute on the `menuitem` which is referred by `menu` attribute.

```
<record model="ir.ui.view" id="v">
    <field name="name">sale.order.form</field>
    <field name="model">sale.order</field>
    <field name="priority" eval="2"/>
    <field name="arch" type="xml">
        <form string="Sale Order">
            .....
        </form>
    </field>
</record>
```

Default value for the priority field : 16. When not specified the system will use the lower priority.

## 14.8.1 View Types

### Tree View

You can specify the columns to include in the list, along with some details of the list's appearance. The search fields aren't specified here, they're specified by the *select* attribute in the form view fields.

```
<record id="view_location_tree2" model="ir.ui.view">
    <field name="name">stock.location.tree</field>
    <field name="model">stock.location</field>
    <field name="type">tree</field>
    <field name="priority" eval="2"/>
    <field name="arch" type="xml">
        <tree
            colors="blue:usage=='view';darkred:usage=='internal'">
            <field name="complete_name"/>
            <field name="usage"/>
            <field
                name="stock_real"
                invisible="'product_id' not in context"/>
            <field
                name="stock_virtual"
                invisible="'product_id' not in context"/>
        </tree>
    </field>
</record>
```

That example is just a flat list, but you can also display a real tree structure by specifying a *field\_parent*. The name is a bit misleading, though; the field you specify must contain a list of all **child** entries.

```
<record id="view_location_tree" model="ir.ui.view">
    <field name="name">stock.location.tree</field>
    <field name="model">stock.location</field>
    <field name="type">tree</field>
    <field name="field_parent">child_ids</field>
    <field name="arch" type="xml">
        <tree toolbar="1">
            <field icon="icon" name="name"/>
        </tree>
    </field>
</record>
```

On the *tree* element, the following attributes are supported:

**colors** Conditions for applying different colours to items in the list. The default is black.

**toolbar** Set this to 1 if you want a tree structure to list the top level entries in a separate toolbar area. When you click on an entry in the toolbar, all its descendants will be displayed in the main tree. The value is ignored for flat lists.

## 14.8.2 Grouping Elements

### Separator

Adds a separator line

#### Example

```
<separator string="Links" colspan="4"/>
```

The *string* attribute defines its label and the *colspan* attribute defines his horizontal size (in number of columns).

## Notebook

<notebook>: With notebooks you can distribute the view fields on different tabs (each one defined by a page tag). You can use the tabpos properties to set tab at: up, down, left, right.

### Example

```
<notebook colspan="4">....</notebook>
```

## Group

<group>: groups several columns and split the group in as many columns as desired.

- **colspan**: the number of columns to use
- **: the number of rows to use**
- **expand**: if we should expand the group or not
- **col**: the number of columns to provide (to its children)
- **string**: (optional) If set, a frame will be drawn around the group of fields, with a label containing the string. Otherwise, the frame will be invisible.

### Example

```
<group col="3" colspan="2">
    <field name="invoiced" select="2"/>
    <button colspan="1" name="make_invoice" states="confirmed" string="Make Invoice"
        type="object"/>
</group>
```

## Page

Defines a new notebook page for the view.

### Example

```
<page string="Order Line"> ... </page>:
```

- **string**: defines the name of the page.

## 14.8.3 Data Elements

### Field

*attributes for the “field” tag*

- **select="1"**: mark this field as being one of the search criteria for this resource’s search view. A value of 1 means that the field is included in the basic search, and a value of 2 means that it is in the advanced search.
- **colspan="4"**: the number of columns on which a field must extend.
- **readonly="1"**: set the widget as read only
- **required="1"**: the field is marked as required. If a field is marked as required, a user has to fill it the system won’t save the resource if the field is not filled. This attribute supersedes the required field value defined in the object.
- **nolabel="1"**: hides the label of the field (but the field is not hidden in the search view).
- **invisible="True"**: hides both the label and the field.

- `password="True"`: replace field values by asterisks, “\*”.
- `string=" "`: change the field label. Note that this label is also used in the search view: see `select` attribute above).
- **domain: can restrict the domain.**
  - Example: `domain=[('partner_id','=',partner_id)]`
- **widget: can change the widget.**
  - **Example: `widget="one2many_list"`**
    - \* `one2one_list`
    - \* `one2many_list`
    - \* `many2one_list`
    - \* `many2many`
    - \* `url`
    - \* `email`
    - \* `image`
    - \* `float_time`
    - \* `reference`
- **mode: sequences of the views when switching.**
  - Example: `mode="tree,graph"`
- **on\_change: define a function that is called when the content of the field changes.**
  - Example: `on_change="onchange_partner(type,partner_id)"`
  - See the *on change event* for details.

• **attrs:** Permits to define attributes of a field depends on other fields of the same window. (It can be use on page, group, ...)

- Format: “{‘attribute’:[(‘field\_name’,‘operator’,‘value’),(‘field\_name’,‘operator’,‘value’)],‘attribute2’:[(‘field\_name’,‘operator’,‘value’)],...}
- where attribute will be readonly, invisible, required
- Default value: {}.
- Example: (in `product.product`)

```
<field digits="(14, 3)" name="volume" attrs="{'readonly':[('type','=','service')]}"/>
```

- `eval`: evaluate the attribute content as if it was Python code (see *below* for example)
- `default_focus`: set to 1 to put the focus (cursor position) on this field when the form is first opened.  
There can only be one field within a view having this attribute set to 1 (**new as of 5.2**)

```
<field name="name" default_focus="1"/>
```

### Example

Here's the source code of the view of a sale order object. This is the object shown on the screen shots of the presentation.

```
<?xml version="1.0"?>
<openerp>
    <data>
        <record id="view_partner_form" model="ir.ui.view">
            <field name="name">res.partner.form</field>
```

```

<field name="model">res.partner</field>
<field name="type">form</field>
<field name="arch" type="xml">
<form string="Partners">
    <group colspan="4" col="6">
        <field name="name" select="1"/>
        <field name="ref" select="1"/>
        <field name="customer" select="1"/>
        <field domain="[( 'domain', '=', 'partner' )]" name="title"/>
        <field name="lang" select="2"/>
        <field name="supplier" select="2"/>
    </group>
    <notebook colspan="4">
        <page string="General">
            <field colspan="4" mode="form,tree" name="address"
                nolabel="1" select="1">
                <form string="Partner Contacts">
                    <field name="name" select="2"/>
                    <field domain="[( 'domain', '=', 'contact' )]" name="title"/>
                    <field name="function"/>
                    <field name="type" select="2"/>
                    <field name="street" select="2"/>
                    <field name="street2"/>
                    <newline/>
                    <field name="zip" select="2"/>
                    <field name="city" select="2"/>
                    <newline/>
                    <field completion="1" name="country_id" select="2"/>
                    <field name="state_id" select="2"/>
                    <newline/>
                    <field name="phone"/>
                    <field name="fax"/>
                    <newline/>
                    <field name="mobile"/>
                    <field name="email" select="2" widget="email"/>
                </form>
                <tree string="Partner Contacts">
                    <field name="name"/>
                    <field name="zip"/>
                    <field name="city"/>
                    <field name="country_id"/>
                    <field name="phone"/>
                    <field name="email"/>
                </tree>
            </field>
            <separator colspan="4" string="Categories"/>
            <field colspan="4" name="category_id" nolabel="1" select="2"/>
        </page>
        <page string="Sales & Purchases">
            <separator string="General Information" colspan="4"/>
            <field name="user_id" select="2"/>
            <field name="active" select="2"/>
            <field name="website" widget="url"/>
            <field name="date" select="2"/>
            <field name="parent_id"/>
            <newline/>
        </page>
        <page string="History">
            <field colspan="4" name="events" nolabel="1" widget="one2many_list"/>
        </page>
        <page string="Notes">
            <field colspan="4" name="comment" nolabel="1"/>
        </page>
    </notebook>
</form>

```

```

        </page>
    </notebook>
</form>
</field>
</record>
<menuitem
    action="action_partner_form"
    id="menu_partner_form"
    parent="base.menu_base_partner"
    sequence="2"/>
</data>
</openerp>

```

## The eval attribute

The **eval** attribute evaluates its content as if it was Python code. This allows you to define values that are not strings.

Normally, content inside `<field>` tags are always evaluated as strings.

### Example 1:

```
<field name="value">2.3</field>
```

This will evaluate to the string '`2.3`' and not the float `2.3`

### Example 2:

```
<field name="value">False</field>
```

This will evaluate to the string '`False`' and not the boolean `False`. This is especially tricky because Python's conversion rules consider any non-empty string to be `True`, so the above code will end up storing the opposite of what is desired.

If you want to evaluate the value to a float, a boolean or another type, except string, you need to use the **eval** attribute:

```
<field name="value" eval="2.3" />
<field name="value" eval="False" />
```

## Button

Adds a button to the current view. Allows the user to perform various actions on the current record.

After a button has been clicked, the record should always be reloaded.

Buttons have the following attributes:

**@type** Defines the type of action performed when the button is activated:

**workflow (default)** The button will send a workflow signal <sup>1</sup> on the current model using the `@name` of the button as workflow signal name and providing the record id as parameter (in a list).

The workflow signal may return an *action descriptor*, which should be executed. Otherwise it will return `False`.

**object** The button will execute the method of name `@name` on the current model, providing the record id as parameter (in a list). This call may return an *action descriptor*, which should be executed. Otherwise it will return `False`.

---

<sup>1</sup> via `exec_workflow` on the `object` rpc endpoint

**action** The button will trigger the execution of an action (`ir.actions.actions`). The `id` of this action is the `@name` of the button.

From there, follows the normal action-execution workflow. One extra action type is to just close the window.

```
return {'type': 'ir.actions.act_window_close'}
```

**@special** Only has one possible value currently: `cancel`, which indicates that the popup should be closed without performing any RPC call or action resolution.

**Note:**

*Only meaningful within a popup-type window (e.g. a wizard). Otherwise, is a noop.*

**Warning:**

*@special and @type are incompatible.*

**@name** The button's identifier, used to indicate which method should be called, which signal sent or which action executed.

**@confirm** A confirmation popup to display before executing the button's task. If the confirmation is dismissed the button's task *must not* be executed.

**@string** The label which should be displayed on the button <sup>2</sup>.

**@icon** Display an icon on the button, if absent the button is text-only <sup>3</sup>.

**@states, @attrs, @invisible** Standard OpenERP meaning for those view attributes

**@default\_focus** If set to a truthy value (1), automatically selects that button so it is used if RETURN is pressed while on the form.

May be ignored by the client. New in version 6.0.

**Example**

```
<button name="order_confirm" states="draft" string="Confirm Order" icon="gtk-execute"/>
<button name="_action_open_window" string="Open Margins" type="object" default_focus="1"/>
```

## Label

Adds a simple label using the `string` attribute as caption.

**Example**

```
<label string="Test"/>
```

## New Line

Force a return to the line even if all the columns of the view are not filled in.

**Example**

```
<newline/>
```

<sup>2</sup> in form view, in list view buttons have no label

<sup>3</sup> behavior in list view is undefined, as list view buttons don't have labels.

## 14.9 Inheritance in Views

When you create and inherit objects in some custom or specific modules, it is better to inherit (than to replace) from an existing view to add/modify/delete some fields and preserve the others.

### Example

```
<record model="ir.ui.view" id="view_partner_form">
    <field name="name">res.partner.form.inherit</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <notebook position="inside">
            <page string="Relations">
                <field name="relation_ids" colspan="4" nolabel="1"/>
            </page>
        </notebook>
    </field>
</record>
```

This will add a page to the notebook of the `res.partner.form` view in the base module.

The inheritance engine will parse the existing view and search for the root nodes of

```
<field name="arch" type="xml">
```

It will append or edit the content of this tag. If this tag has some attributes, it will look in the parent view for a node with matching attributes (except position).

You can use these values in the position attribute:

- inside (default): your values will be appended inside the tag
- after: add the content after the tag
- before: add the content before the tag
- replace: replace the content of the tag.

### 14.9.1 Replacing Content

```
<record model="ir.ui.view" id="view_partner_form1">
    <field name="name">res.partner.form.inherit1</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <page string="Extra Info" position="replace">
            <field name="relation_ids" colspan="4" nolabel="1"/>
        </page>
    </field>
</record>
```

Will replace the content of the Extra Info tab of the notebook with the `relation_ids` field.

The parent and the inherited views are correctly updated with `--update=all` argument like any other views.

### 14.9.2 Deleting Content

To delete a field from a form, an empty element with `position="replace"` attribute is used. Example:

```

<record model="ir.ui.view" id="view_partner_form2">
    <field name="name">res.partner.form.inherit2</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <field name="lang" position="replace"/>
    </field>
</record>

```

### 14.9.3 Inserting Content

To add a field into a form before the specified tag use position="before" attribute.

```

<record model="ir.ui.view" id="view_partner_form3">
    <field name="name">res.partner.form.inherit3</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <field name="lang" position="before">
            <field name="relation_ids"/>
        </field>
    </field>
</record>

```

Will add relation\_ids field before the lang field.

To add a field into a form after the specified tag use position="after" attribute.

```

<record model="ir.ui.view" id="view_partner_form4">
    <field name="name">res.partner.form.inherit4</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <field name="lang" position="after">
            <field name="relation_ids"/>
        </field>
    </field>
</record>

```

Will add relation\_ids field after the lang field.

### 14.9.4 Multiple Changes

To make changes in more than one location, wrap the fields in a data element.

```

<record model="ir.ui.view" id="view_partner_form5">
    <field name="name">res.partner.form.inherit5</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <data>
            <field name="lang" position="replace"/>
            <field name="website" position="after">
                <field name="lang"/>
            </field>
        </data>
    </field>
</record>

```

Will delete the lang field from its usual location, and display it after the website field.

### 14.9.5 XPath Element

Sometimes a view is too complicated to let you simply identify a target field by name. For example, the field might appear in two places. When that happens, you can use an `xpath` element to describe where your changes should be placed.

```
<record model="ir.ui.view" id="view_partner_form6">
    <field name="name">res.partner.form.inherit6</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <data>
            <xpath
                expr="//field[@name='address']/form/field[@name='email']"
                position="after">
                <field name="age"/>
            </xpath>
            <xpath
                expr="//field[@name='address']/tree/field[@name='email']"
                position="after">
                <field name="age"/>
            </xpath>
        </data>
    </field>
</record>
```

Will add the `age` field after the `email` field in both the form and tree view of the address list.

### 14.9.6 Replacing Attributes

The `attributes` position lets you change an element's attributes without completely replacing it and its children. A common example is changing the colours in a tree view.

```
<record id="mrp_production_tree_view" model="ir.ui.view">
    <field name="name">mrp.production.mycompany.tree.view</field>
    <field name="model">mrp.production</field>
    <field name="type">tree</field>
    <field name="inherit_id" ref="mrp.mrp_production_tree_view"/>
    <field name="arch" type="xml">
        <xpath expr="//tree" position="attributes">
            <attribute name="colors">blue:state=='draft'</attribute>
        </xpath>
    </field>
</record>
```

## 14.10 Specify the views you want to use

There are some cases where you would like to specify a view other than the default:

- If there are several form or tree views for an object.
- If you want to change the form or tree view used by a relational field (one2many for example).

### 14.10.1 Using the priority field

This field is available in the view definition, and is 16 by default. By default, OpenERP will display a model using the view with the highest priority (the smallest number). For example, imagine we have two views for a simple model. The model `client` with two fields : **firstname** and **lastname**. We will define two views, one which shows the firstname first, and the other one which shows the lastname first.

```

1 <!--
2     Here is the first view for the model 'client'.
3     We don't specify a priority field, which means
4     by default 16.
5 -->
6 <record model="ir.ui.view" id="client_form_view_1">
7     <field name="name">client.form.view1</field>
8     <field name="model">client</field>
9     <field name="type">form</field>
10    <field name="arch" type="xml">
11        <field name="firstname"/>
12        <field name="lastname"/>
13    </field>
14 </record>
15
16 <!--
17     A second view, which show fields in an other order.
18     We specify a priority of 15.
19 -->
20 <record model="ir.ui.view" id="client_form_view_2">
21     <field name="name">client.form.view2</field>
22     <field name="model">client</field>
23     <field name="priority" eval="15"/>
24     <field name="type">form</field>
25     <field name="arch" type="xml">
26         <field name="lastname"/>
27         <field name="firstname"/>
28     </field>
29 </record>

```

Now, each time OpenERP will have to show a form view for our object *client*, it will have the choice between two views. **It will always use the second one, because it has a higher priority !** Unless you tell it to use the first one !

## 14.10.2 Specify per-action view

To illustrate this point, we will create 2 menus which show a form view for this *client* object :

```

1 <!--
2     This action open the default view (in our case,
3     the view with the highest priority, the second one)
4 -->
5 <record
6     model="ir.actions.act_window"
7     id="client_form_action"
8     <field name="name">client.form.action</field>
9     <field name="res_model">client</field>
10    <field name="view_type">form</field>
11    <field name="view_mode">form</field>
12 </record>
13
14 <!--
15     This action open the view we specify.
16 -->
17 <record
18     model="ir.actions.act_window"
19     id="client_form_action1"
20     <field name="name">client.form.action1</field>
21     <field name="res_model">client</field>
22     <field name="view_type">form</field>
23     <field name="view_mode">form</field>

```

```

24   <field name="view_id" ref="client_form_view_1"/>
25 </record>
26
27 <menuitem id="menu_id" name="Client main menu"/>
28 <menuitem
29   id="menu_id_1"
30   name="Here we don't specify the view"
31   action="client_form_action" parent="menu_id"/>
32 <menuitem
33   id="menu_id_1"
34   name="Here we specify the view"
35   action="client_form_action1" parent="menu_id"/>

```

As you can see on line 19, we can specify a view. That means that when we open the second menu, OpenERP will use the form view `client_form_view_1`, regardless of its priority.

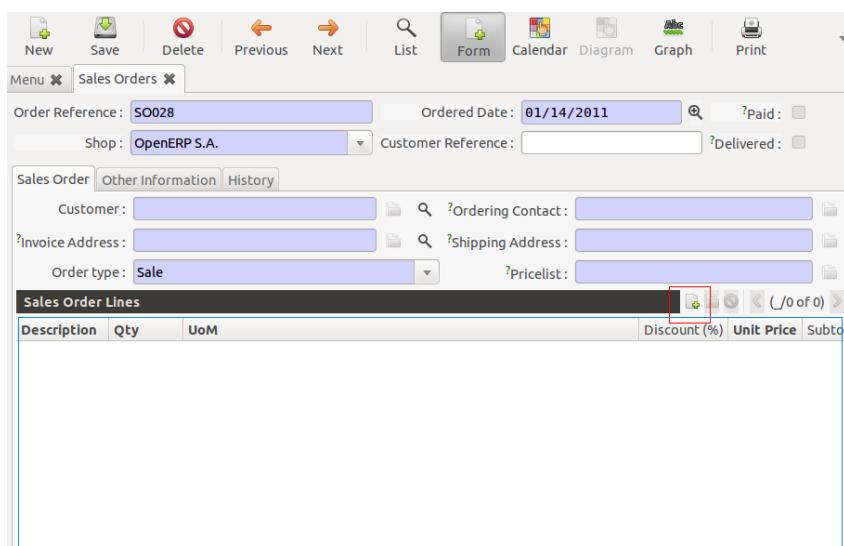
**Note:**

*Remember to use the module name (module.view\_id) in the ref attribute if you are referring to a view defined in another module.*

### 14.10.3 Specify views for related fields

#### Using the context

The `view_id` method works very well for menus/actions, but how can you specify the view to use for a one2many field, for example? When you have a one2many field, two views are used, a tree view (in blue), and a form view when you click on the add button (in red).



When you add a one2many field in a form view, you do something like this :

```
<field name="order_line" colspan="4" nolabel="1"/>
```

If you want to specify the views to use, you can add a `context` attribute, and specify a view id for each type of view supported, exactly like the action's `view_id` attribute:

```
<field name="order_line" colspan="4" nolabel="1"
       context="{'form_view_ref' : 'module.view_id', 'tree_view_ref' : 'module.view_id'}"/>
```

If you don't specify the views, OpenERP will choose one in this order :

1. It will use the <form> or <tree> view defined **inside** the field (see below)
2. Else, it will use the views with the highest priority for this object.
3. Finally, it will generate default empty views, with all fields.

**Note:**

---

*The context keys are named <view\_type>\_view\_ref.*

---

**Note:**

*By default, OpenERP will never use a view that is not defined for your object. If you have two models, with the same fields, but a different model name, OpenERP will never use the view of one for the other, even if one model inherit an other.*

*You can force this by manually specifying the view, either in the action or in the context.*

---

## Using subviews

In the case of relational fields, you can create a view directly inside a field :

```
<record model="ir.ui.view" id="some_view">
    <field name="name">some.view</field>
    <field name="type">form</field>
    <field name="model">some.model.with.one2many</field>
    <field name="arch" type="xml">
        <field name="..."/>

        <!-- <== order_line is a one2many field -->
        <field name="order_line" colspan="4" nolabel="1">
            <form>
                <field name="qty"/>
                ...
            </form>
            <tree>
                <field name="qty"/>
                ...
            </tree>
        </field>
    </field>
```

If you or another developer want to inherit from this view in another module, you need to inherit from the parent view and then modify the child fields. With child views, you'll often need to use an *XPath Element* to describe exactly where to place your new fields.

```
<record model="ir.ui.view" id="some_inherited_view">
    <field name="name">some.inherited.view</field>
    <field name="type">form</field>
    <field name="model">some.model.with.one2many</field>
    <field name="inherit_id" ref="core_module.some_view"/>
    <field name="arch" type="xml">
        <data>
            <xpath
                expr="//field[@name='order_line']/form/field[@name='qty']"
                position="after">
                <field name="size"/>
            </xpath>
            <xpath
                expr="//field[@name='order_line']/tree/field[@name='qty']"
                position="after">
                <field name="size"/>
            </xpath>
        </data>
    </field>
```

```

        </xpath>
    </data>
</field>
```

One down side of defining a subview like this is that it can't be inherited on its own, it can only be inherited with the parent view. Your views will be more flexible if you define the child views separately and then specify which child view to use as part of the one2many field.

## 14.11 Events

### 14.11.1 On Change

The on\_change attribute defines a method that is called when the content of a view field has changed.

This method takes at least arguments: cr, uid, ids, which are the three classical arguments and also the context dictionary. You can add parameters to the method. They must correspond to other fields defined in the view, and must also be defined in the XML with fields defined this way:

```
<field
    name="name_of_field"
    on_change="name_of_method(other_field_1, ..., other_field_n)"/>
```

The example below is from the sale order view.

You can use the ‘context’ keyword to access data in the context that can be used as params of the function.:

```
<field name="shop_id" on_change="onchange_shop_id(shop_id)"/>

def onchange_shop_id(self, cr, uid, ids, shop_id):

    v={}
    if shop_id:

        shop=self.pool.get('sale.shop').browse(cr,uid,shop_id)
        v['project_id']=shop.project_id.id
        if shop.pricelist_id.id:

            v['pricelist_id']=shop.pricelist_id.id

        v['payment_default_id']=shop.payment_default_id.id

    return {'value':v}
```

When editing the shop\_id form field, the onchange\_shop\_id method of the sale\_order object is called and returns a dictionary where the ‘value’ key contains a dictionary of the new value to use in the ‘project\_id’, ‘pricelist\_id’ and ‘payment\_default\_id’ fields.

Note that it is possible to change more than just the values of fields. For example, it is possible to change the value of some fields and the domain of other fields by returning a value of the form: return {‘domain’: d, ‘value’: value}

**returns** a dictionary with any mix of the following keys:

**domain** A mapping of {field: domain}.

The returned domains should be set on the fields instead of the default ones.

**value** A mapping of {field: value}, the values will be set on the corresponding fields and may trigger new onchanges or attrs changes

**warning** A dict with the keys **title** and **message**. Both are mandatory. Indicate that an error message should be displayed to the user.



# MENU AND ACTIONS

## 15.1 Menus

Here's the template of a menu item :

```
<menuitem id="menuitem_id"
          name="Position/Of/The/Menu/Item/In/The/Tree"
          action="action_id"
          icon="NAME_FROM_LIST"
          groups="groupname"
          sequence="" />
```

Where

- **id** specifies the identifier of the menu item in the menu items table. This identifier must be unique. Mandatory field.
  - **name** defines the position of the menu item in the menu hierarchy. Elements are separated by slashes (“/”). A menu item name with no slash in its text is a top level menu. Mandatory field.
  - **action** specifies the identifier of the action that must have been defined in the action table (ir.actions.act\_window). Note that this field is not mandatory : you can define menu elements without associating actions to them. This is useful when defining custom icons for menu elements that will act as folders (for example this is how custom icons for “Projects”, “Human Resources” in OpenERP are defined).
  - **icon** specifies which icon will be displayed for the menu item using the menu item. The default icon is STOCK\_OPEN.
- The available icons are : STOCK\_ABOUT, STOCK\_ADD, STOCK\_APPLY, STOCK\_BOLD, STOCK\_CANCEL, STOCK\_CDROM, STOCK\_CLEAR, STOCK\_CLOSE, STOCK\_COLOR\_PICKER, STOCK\_CONNECT, STOCK\_CONVERT, STOCK\_COPY, STOCK\_CUT, STOCK\_DELETE, STOCK\_DIALOG\_AUTHENTICATION, STOCK\_DIALOG\_ERROR, STOCK\_DIALOG\_INFO, STOCK\_DIALOG\_QUESTION, STOCK\_DIALOG\_WARNING, STOCK\_DIRECTORY, STOCK\_DISCONNECT, STOCK\_DND, STOCK\_DND\_MULTIPLE, STOCK\_EDIT, STOCK\_EXECUTE, STOCK\_FILE, STOCK\_FIND, STOCK\_FIND\_AND\_REPLACE, STOCK\_FLOPPY, STOCK\_GOTO\_BOTTOM, STOCK\_GOTO\_FIRST, STOCK\_GOTO\_LAST, STOCK\_GOTO\_TOP, STOCK\_GO\_BACK, STOCK\_GO\_DOWN, STOCK\_GO\_FORWARD, STOCK\_GO\_UP, STOCK\_HARDDISK, STOCK\_HELP, STOCK\_HOME, STOCK\_INDENT, STOCK\_INDEX, STOCK\_ITALIC, STOCK\_JUMP\_TO, STOCK\_JUSTIFY\_CENTER, STOCK\_JUSTIFY\_FILL, STOCK\_JUSTIFY\_LEFT, STOCK\_JUSTIFY\_RIGHT, STOCK\_MEDIA\_FORWARD, STOCK\_MEDIA\_NEXT, STOCK\_MEDIA\_PAUSE, STOCK\_MEDIA\_PLAY, STOCK\_MEDIA\_PREVIOUS, STOCK\_MEDIA\_RECORD, STOCK\_MEDIA\_REWIND, STOCK\_MEDIA\_STOP, STOCK\_MISSING\_IMAGE, STOCK\_NETWORK, STOCK\_NEW, STOCK\_NO, STOCK\_OK, STOCK\_OPEN, STOCK\_PASTE, STOCK\_PREFERENCES, STOCK\_PRINT, STOCK\_PRINT\_PREVIEW, STOCK\_PROPERTIES, STOCK\_QUIT, STOCK\_REFRESH, STOCK\_REDRAW,

STOCK\_REMOVE, STOCK\_REVERT\_TO\_SAVED, STOCK\_SAVE, STOCK\_SAVE\_AS, STOCK\_SELECT\_COLOR, STOCK\_SELECT\_FONT, STOCK\_SORT\_ASCENDING, STOCK\_SORT\_DESCENDING, STOCK\_SPELL\_CHECK, STOCK\_STOP, STOCK\_STRIKETHROUGH, STOCK\_UNDELETE, STOCK\_UNDERLINE, STOCK\_UNDO, STOCK\_UNINDENT, STOCK\_YES, STOCK\_ZOOM\_100, STOCK\_ZOOM\_FIT, STOCK\_ZOOM\_IN, STOCK\_ZOOM\_OUT, terp-account, terp-crm, terp-mrp, terp-product, terp-purchase, terp-sale, terp-tools, terp-administration, terp-hr, terp-partner, terp-project, terp-report, terp-stock

- **groups** specifies which group of user can see the menu item (example : groups="admin"). See section "Management of Access Rights" for more information. Multiple groups should be separated by a ',' (example: groups="admin,user")
- **sequence** is an integer that is used to sort the menu item in the menu. The higher the sequence number, the downer the menu item. This argument is not mandatory: if sequence is not specified, the menu item gets a default sequence number of 10. Menu items with the same sequence numbers are sorted by order of creation (\_order = "sequence,id").

### 15.1.1 Example

In server/bin/addons/sale/sale\_view.xml, we have, for example

```
<menuitem name="Sales Management/Sales Order/Sales Order in Progress" id="menu_action_order_tree4"
```

To change the icon of menu item :

- \* Highlight the menu with the icon you want to change.
- \* Select the "Switch to list/form" option from the "Form" menu. This will take you to the Menu editor.
- \* From here you can change the icon of the selected menu.

## 15.2 Actions

### 15.2.1 Introduction

The actions define the behavior of the system in response to the actions of the users ; login of a new user, double-click on an invoice, click on the action button, ...

There are different types of simple actions:

- Window: Opening of a new window
- **Report: The printing of a report** o Custom Report: The personalized reports o RML Report: The XSL:RML reports
- Wizard: The beginning of a Wizard
- Execute: The execution of a method on the server side
- Group: Gather some actions in one group

The actions are used for the following events;

- User connection,
- The user double-clicks on the menu,
- The user clicks on the icon 'print' or 'action'.

## 15.2.2 Example of events

In OpenERP, all the actions are described and not configured. Two examples:

- Opening of a window when double-clicking in the menu
- User connection

### Opening of the menu

When the user open the option of the menu “Operations > Partners > Partners Contact”, the next steps are done to give the user information on the action to undertake.

1. Search the action in the IR.

#### 2. Execution of the action

- (a) If the action is the type Opening the Window; it indicates to the user that a new window must be opened for a selected object and it gives you the view (form or list) and the file to use (only the pro-forma invoice).
- (b) The user asks the object and receives information necessary to trace a form; the fields description and the XML view.

### User connection

When a new user is connected to the server, the client must search the action to use for the first screen of this user. Generally, this action is: open the menu in the ‘Operations’ section.

The steps are:

1. Reading of a user file to obtain ACTION\_ID
2. Reading of the action and execution of this one

### The fields

**Action Name** The action name

**Action Type** Always ‘ir.actions.act\_window’

**View Ref** The view used for showing the object

**Model** The model of the object to post

**Type of View** The type of view (Tree/Form)

**Domain Value** The domain that decreases the visible data with this view

## 15.2.3 The view

The view describes how the edition form or the data tree/list appear on screen. The views can be of ‘Form’ or ‘Tree’ type, according to whether they represent a form for the edition or a list/tree for global data viewing.

A form can be called by an action opening in ‘Tree’ mode. The form view is generally opened from the list mode (like if the user pushes on ‘switch view’).

## 15.2.4 The domain

This parameter allows you to regulate which resources are visible in a selected view.(restriction)

For example, in the invoice case, you can define an action that opens a view that shows only invoices not paid.

The domains are written in python; list of tuples. The tuples have three elements;

- the field on which the test must be done
- the operator used for the test (<, >, =, like)
- the tested value

For example, if you want to obtain only ‘Draft’ invoice, use the following domain; [('state','=','draft')]

In the case of a simple view, the domain define the resources which are the roots of the tree. The other resources, even if they are not from a part of the domain will be posted if the user develop the branches of the tree.

## 15.2.5 Window Action

Actions are explained in more detail in the *Client Action* section. Here’s the template of an action XML record :

```
<record model="ir.actions.act_window" id="action_id_1">
    <field name="name">action.name</field>
    <field name="view_id" ref="view_id_1"/>
    <field name="domain">["list of 3-tuples (max 250 characters)"]</field>
    <field name="context">{"context dictionary (max 250 characters)"}</field>
    <field name="res_model">Open.object</field>
    <field name="view_type">form|tree</field>
    <field name="view_mode">form,tree|tree,form|form|tree</field>
    <field name="usage">menu</field>
    <field name="target">new</field>
</record>
```

### Where

- **id** is the identifier of the action in the table “ir.actions.act\_window”. It must be unique.
- **name** is the name of the action (mandatory).
- **view\_id** is the name of the view to display when the action is activated. If this field is not defined, the view of a kind (list or form) associated to the object res\_model with the highest priority field is used (if two views have the same priority, the first defined view of a kind is used).
- **domain** is a list of constraints used to refine the results of a selection, and hence to get less records displayed in the view. Constraints of the list are linked together with an AND clause : a record of the table will be displayed in the view only if all the constraints are satisfied.
- **context** is the context dictionary which will be visible in the view that will be opened when the action is activated. Context dictionaries are declared with the same syntax as Python dictionaries in the XML file. For more information about context dictionaries, see section ” The context Dictionary”.
- **res\_model** is the name of the object on which the action operates.
- **view\_type** is set to form when the action must open a new form view, and is set to tree when the action must open a new tree view.
- **view\_mode** is only considered if view\_type is form, and ignored otherwise. The four possibilities are :
  - **form,tree** : the view is first displayed as a form, the list view can be displayed by clicking the “alternate view button” ;
  - **tree,form** : the view is first displayed as a list, the form view can be displayed by clicking the “alternate view button” ;

- **form** : the view is displayed as a form and there is no way to switch to list view ;
- **tree** : the view is displayed as a list and there is no way to switch to form view.

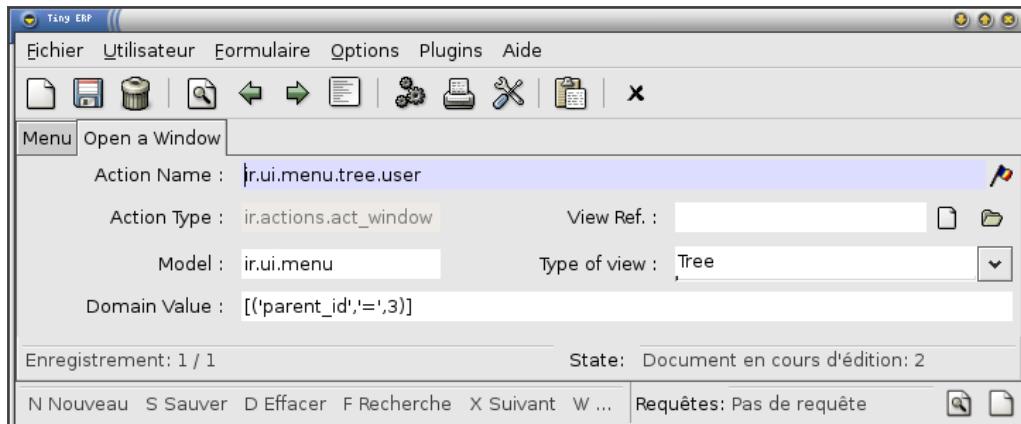
(version 5 introduced **graph** and **calendar** views)

- **usage** is used [+ \*TODO\* +]
- **target** the view will open in new window like wizard.
- **context** will be passed to the action itself and added to its global context

```
<record model="ir.actions.act_window" id="a">
    <field name="name">account.account.tree1</field>
    <field name="res_model">account.account</field>
    <field name="view_type">tree</field>
    <field name="view_mode">form,tree</field>
    <field name="view_id" ref="v"/>
    <field name="domain">[('code','=',0')]</field>
    <field name="context">{'project_id': active_id}</field>
</record>
```

They indicate at the user that he has to open a new window in a new ‘tab’.

Administration > Custom > Low Level > Base > Action > Window Actions



## Examples of actions

This action is declared in server/bin/addons/project/project\_view.xml.

```
<record model="ir.actions.act_window" id="open_view_my_project">
    <field name="name">project.project</field>
    <field name="res_model">project.project</field>
    <field name="view_type">tree</field>
    <field name="domain">[('parent_id','=',False), ('manager', '=', uid)]</field>
    <field name="view_id" ref="view_my_project" />
</record>
```

This action is declared in server/bin/addons/stock/stock\_view.xml.

```
<record model="ir.actions.act_window" id="action_picking_form">
    <field name="name">stock.picking</field>
    <field name="res_model">stock.picking</field>
    <field name="type">ir.actions.act_window</field>
    <field name="view_type">form</field>
    <field name="view_id" ref="view_picking_form"/>
    <field name="context">{'contact_display': 'partner'}</field>
</record>
```

## 15.2.6 Url Action

## 15.2.7 Wizard Action

Here's an example of a .XML file that declares a wizard.

```
<?xml version="1.0"?>
<openerp>
    <data>
        <wizard string="Employee Info"
            model="hr.employee"
            name="employee.info.wizard"
            id="wizard_employee_info"/>
    </data>
</openerp>
```

A wizard is declared using a wizard tag. See “Add A New Wizard” for more information about wizard XML.

also you can add wizard in menu using following xml entry

```
<?xml version="1.0"?>
<openerp>
    <data>
        <wizard string="Employee Info"
            model="hr.employee"
            name="employee.info.wizard"
            id="wizard_employee_info"/>
        <menuitem
            name="Human Resource/Employee Info"
            action="wizard_employee_info"
            type="wizard"
            id="menu_wizard_employee_info"/>
    </data>
</openerp>
```

## 15.2.8 Report Action

### Report declaration

Reports in OpenERP are explained in chapter “Reports Reporting”. Here's an example of a XML file that declares a RML report :

```
<?xml version="1.0"?>
<openerp>
    <data>
        <report id="sale_category_print"
            string="Sales Orders By Categories"
            model="sale.order"
            name="sale_category.print"
            rml="sale_category/report/sale_category_report.rml"
            menu="True"
            auto="False"/>
    </data>
</openerp>
```

A report is declared using a **report tag** inside a “data” block. The different arguments of a report tag are :

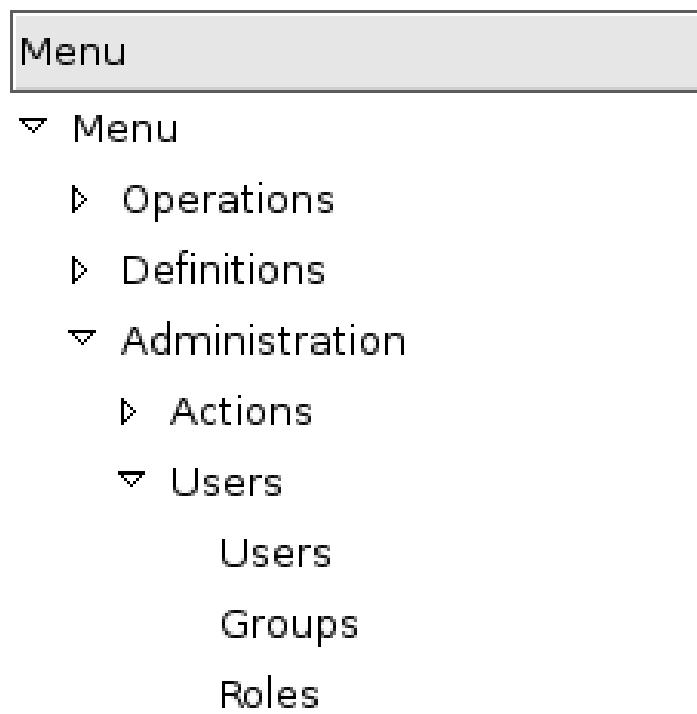
- **id** : an identifier which must be unique.
- **string** : the text of the menu that calls the report (if any, see below).
- **model** : the OpenERP object on which the report will be rendered.

- **rml** : the .RML report model. Important Note : Path is relative to addons/ directory.
- **menu** : whether the report will be able to be called directly via the client or not. Setting menu to False is useful in case of reports called by wizards.
- **auto** : determines if the .RML file must be parsed using the default parser or not. Using a custom parser allows you to define additional functions to your report.

## 15.3 Security

Three concepts are differentiated into OpenERP;

1. The users: person identified by his login/password
2. The groups: define the access rights of the resources
3. The roles: determine the roles/duties of the users



### The users

They represent physical persons. These are identified with a login and a password. A user may belong to several groups and may have several roles.

A user must have an action set up. This action is executed when the user connects to the program with his login and password. An example of action would be to open the menu at ‘Operations’.

The preferences of the user are available with the preference icon. You can, for example, through these preferences, determine the working language of this user. English is set by default.

A user can modify his own preferences while he is working with OpenERP. To do that, he clicks on this menu: User > Preferences. The OpenERP administrator can also modify some preferences of each and every user.

### The groups

The groups determine the access rights to the different resources. There are three types of right:

- The writing access: recording & creation,
- The reading access: reading of a file,

- The execution access: the buttons of workflows or wizards.

A user can belong to several groups. If he belongs to several groups, we always use the group with the highest rights for a selected resource.

### The roles

The roles define a hierarchical structure in tree. They represent the different jobs/roles inside the company. The biggest role has automatically the rights of all the inferior roles.

#### Example:

CEO

- Technical manager
  - Chief of projects
    - \* Developers
    - \* Testers
- Commercial manager
  - Salesmen
  - ...

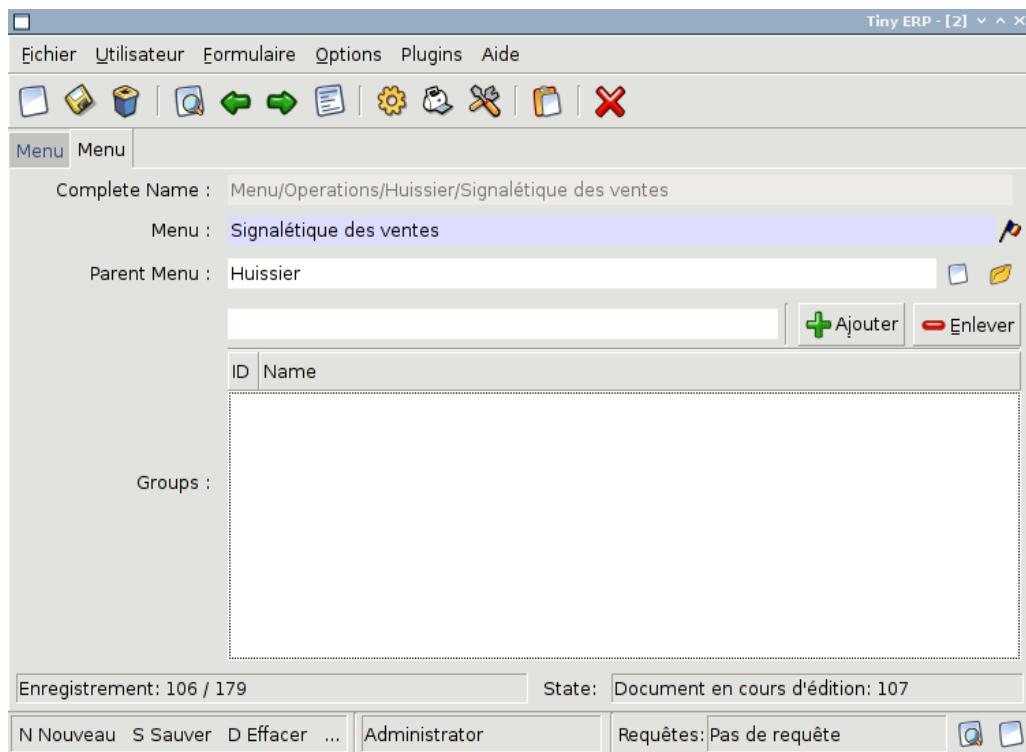
If we want to validate the test of a program (=role Testers), it may be done by a user having one of the following roles: Testers, Chief of the project, Technical manager, CEO.

The roles are used for the transition of Workflow actions into confirmation, choice or validation actions. Their implications will be detailed in the Workflow section.

### 15.3.1 Menu Access

It's easy (but risky) to grant grained access to menu based on the user's groups.

First of all, you should know that if a menu is not granted to any group then it is accessible to everybody ! If you want to grant access to some groups just go to **Menu > Administration > Security > Define access to Menu-items** and select the groups that can use this menu item.



Beware ! If the Administrator does not belong to one of the group, he will not be able to reach this menu again.



## **Part V**

### **Creating Wizard - (The Process)**



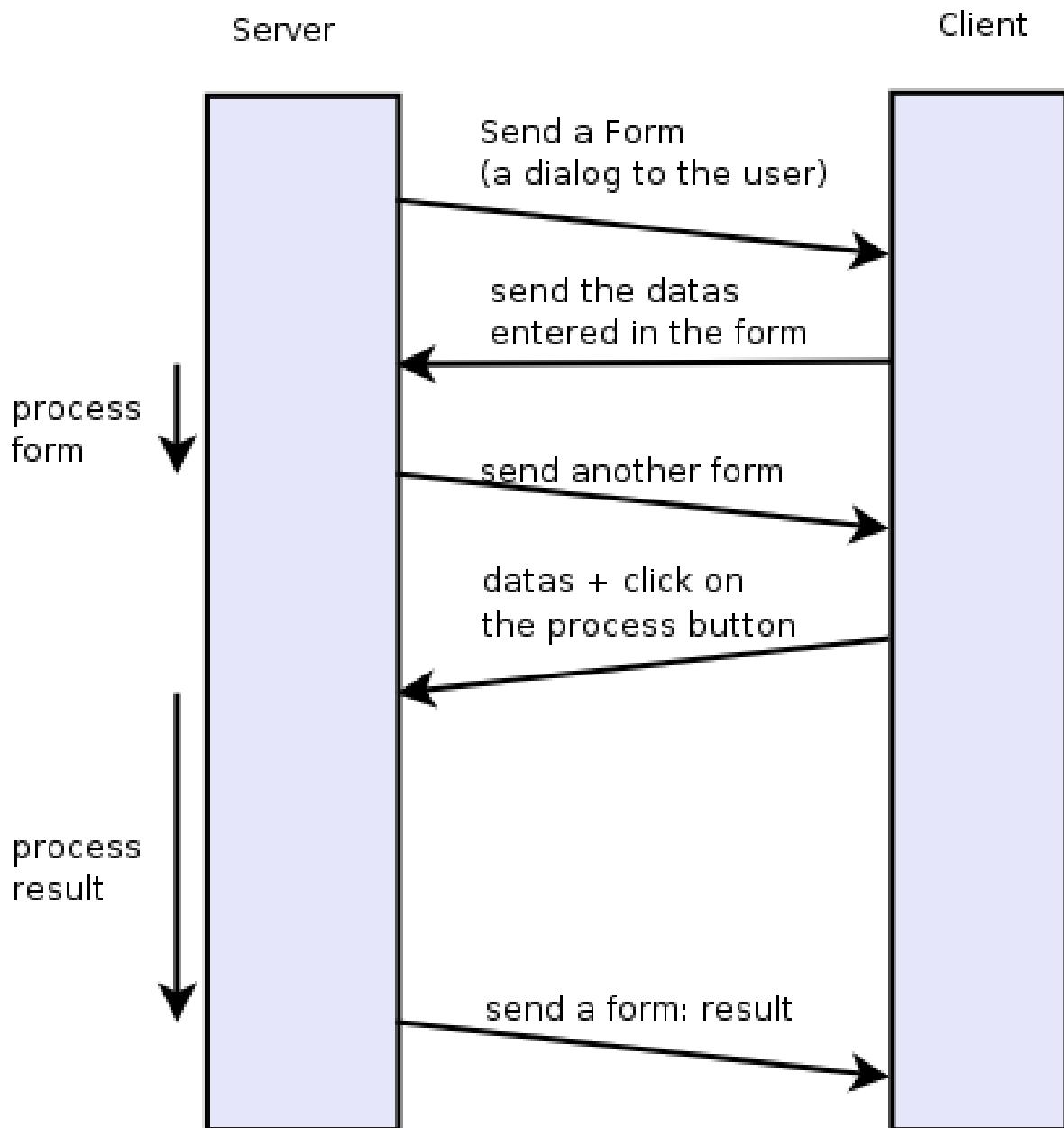
# **INTRODUCTION**

Wizards describe interaction sequences between the client and the server.

Here is, as an example, a typical process for a wizard:

1. A window is sent to the client (a form to be completed)
2. The client sends back the data from the fields which were filled in
3. The server gets the result, usually execute a function and possibly sends another window/form to the client

# A wizard process: example



Here is a screenshot of the wizard used to reconcile transactions (when you click on the gear icon in an account chart):

Tiny ERP

File User Form Options Plugins Help

New Save Delete Find Back Forward Other View Action Print Attachment Close

Menu Account Chart Account Entry Line

**Information**

**General Information**

Analytic Lines

Name : <input type="text" value="2"/>	Effective date : <input type="text" value="03/13/2007"/>
Account : <input type="text" value="560 - Compte courant"/>	Partner Ref. : <input type="text" value="ASUStek"/>
Debit : <input type="text" value="545.00"/>	Credit : <input type="text" value="0.00"/>

**Optional Information**

Currency : <input type="text"/>	Amount Currency : <input type="text" value="0.00"/>
Quantity : <input type="text" value="0.00"/>	Entry : <input type="text" value="2"/>
St. : <input type="text"/>	Litigation : <input type="checkbox"/>
Maturity date : <input type="text"/>	Creation date : <input type="text" value="03/13/2007"/>

**State**

Journal : <input type="text" value="Sales Journal"/>	Period : <input type="text" value="Mar.2007"/>
Reconcile : <input type="checkbox"/>	Active : <input checked="" type="checkbox"/>

**X Reconciliation**

**Reconciliation transactions**

# of Transaction : <input type="text" value="1"/>	Credit amount : <input type="text" value="0.00"/>	Debit amount : <input type="text" value="545.00"/>
---	---	--

**Write-Off**

Write-Off amount : <input type="text" value="545.00"/>
--

Enregistrements: 1 / 1 - Document en cours d'édition: 5)

Cancel Reconcile State:

http://localhost:8069 [terp] Administrator Requests: Pas de requête



# WIZARDS - PRINCIPLES

A wizard is a succession of steps. A step is composed of several actions;

1. send a form to the client and some buttons
2. get the form result and the button pressed from the client
3. execute some actions
4. send a new action to the client (form, print, ...)

To define a wizard, you have to create a class inheriting from **wizard.interface** and instantiate it. Each wizard must have a unique name, which can be chosen arbitrarily except for the fact it has to start with the module name (for example: account.move.line.reconcile). The wizard must define a dictionary named **states** which defines all its steps. A full example of a simple wizard can be found at <http://www.openobject.com/forum/post43900.html#43900>

Here is an example of such a class:

```
class wiz_reconcile(wizard.interface):  
    states = {  
        'init': {  
            'actions': [_trans_rec_get],  
            'result': {'type': 'form',  
                      'arch': _transaction_form,  
                      'fields': _transaction_fields,  
                      'state': [('reconcile', 'Reconcile'), ('end', 'Cancel')]}  
        },  
        'reconcile': {  
            'actions': [_trans_rec_reconcile],  
            'result': {'type': 'state', 'state': 'end'}  
        }  
    }  
wiz_reconcile('account.move.line.reconcile');
```

The ‘states’ dictionary define all the states of the wizard. In this example; **init** and **reconcile**. There is another state which is named **end** which is implicit.

A wizard always starts in the **init** state and ends in the **end** state.

A state define two things:

1. a list of actions
2. a result

## 17.1 The list of actions

Each step/state of a wizard defines a list of actions which are executed when the wizard enters the state. This list can be empty.

The function (actions) must have the following signatures:

```
def _trans_rec_get(self, uid, data, res_get=False):
```

Where:

- **self** is the pointer to the wizard object
- **uid** is the user ID of the user which is executing the wizard
- **data is a dictionary containing the following data:**
  - **ids**: the list of ids of resources selected when the user executed the wizard
  - **id**: the id highlighted when the user executed the wizard
  - **form**: a dictionary containing all the values the user completed in the preceding forms. If you change values in this dictionary, the following forms will be pre-completed.

Each action function must return a dictionary. Any entries in this dictionary will be merged with the data that is passed to the form when it's displayed.

## 17.2 The result

Here are some result examples:

Result: next step

```
'result': {'type': 'state',
            'state': 'end'}
```

Indicate that the wizard has to continue to the next state: 'end'. If this is the 'end' state, the wizard stops.

Result: new dialog for the client

```
'result': {'type': 'form',
            'arch': '_form',
            'fields': '_fields',
            'state': [('reconcile', 'Reconcile'), ('end', 'Cancel')]}]
```

The type=form indicate that this step is a dialog to the client. The dialog is composed of:

1. a form : with fields description and a form description
2. some buttons : on which the user press after completing the form

The form description (arch) is like in the views objects. Here is an example of form:

```
_form = """<?xml version="1.0"?>
<form title="Reconciliation">
    <separator string="Reconciliation transactions" colspan="4"/>
    <field name="trans_nbr"/>
    <newline/>
    <field name="credit"/>
    <field name="debit"/>
    <field name="state"/>
    <separator string="Write-Off" colspan="4"/>
    <field name="writeoff"/>
    <newline/>
    <field name="writeoff_acc_id" colspan="3"/>
</form>
"""
```

The fields description is similar to the fields described in the python ORM objects. Example:

```

_transaction_fields = {
    'trans_nbr': {'string': '# of Transaction', 'type': 'integer', 'readonly': True},
    'credit': {'string': 'Credit amount', 'type': 'float', 'readonly': True},
    'debit': {'string': 'Debit amount', 'type': 'float', 'readonly': True},
    'state': {
        'string': "Date/Period Filter",
        'type': 'selection',
        'selection': [('bydate', 'By Date'),
                      ('byperiod', 'By Period'),
                      ('all', 'By Date and Period'),
                      ('none', 'No Filter')],
        'default': lambda *a:'none'
    },
    'writeoff': {'string': 'Write-Off amount', 'type': 'float', 'readonly': True},
    'writeoff_acc_id': {'string': 'Write-Off account',
                        'type': 'many2one',
                        'relation': 'account.account'
    },
}

```

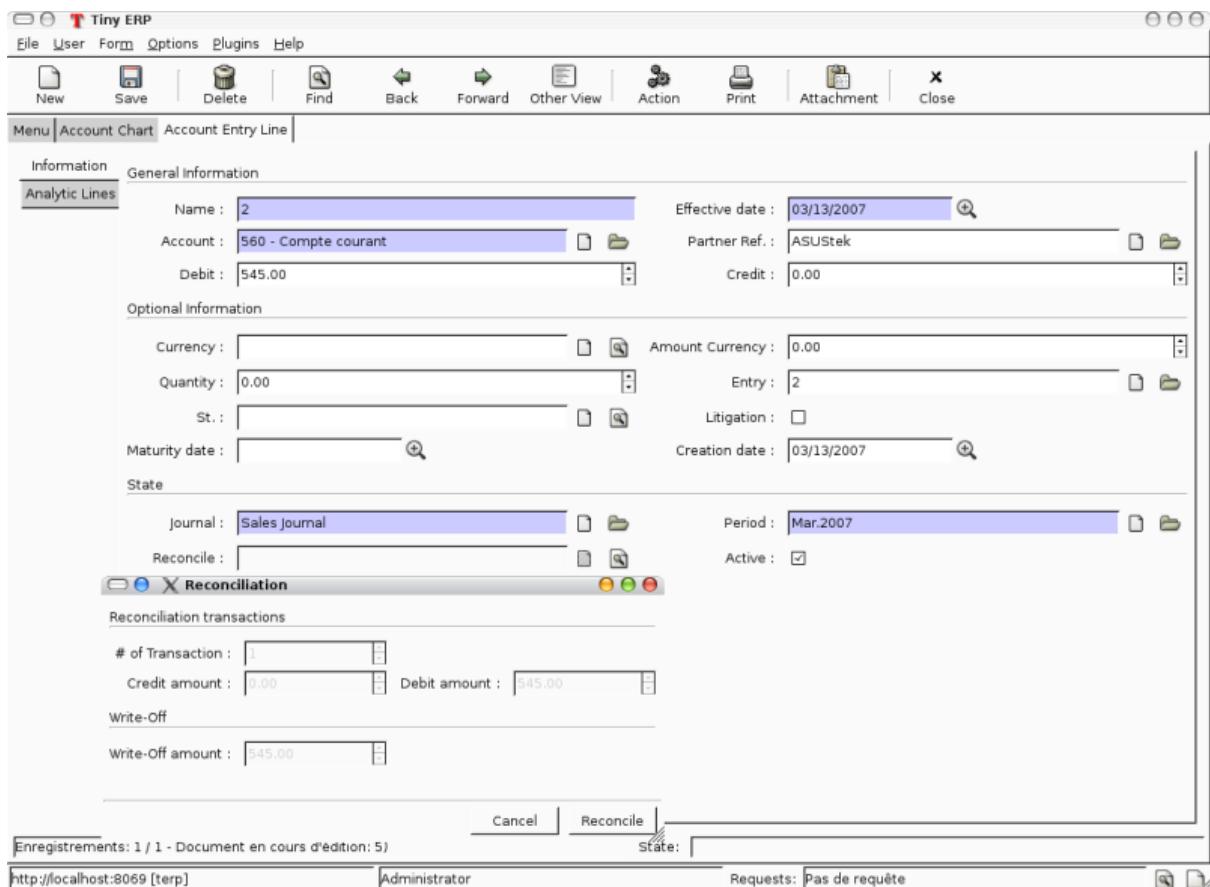
Each step/state of a wizard can have several buttons. Those are located on the bottom right of the dialog box. The list of buttons for each step of the wizard is declared in the state key of its result dictionary.

For example:

```
'state':[('end', 'Cancel', 'gtk-cancel'), ('reconcile', 'Reconcile', '', True)]
```

1. the next step name (determine which state will be next)
2. the button string (to display for the client)
3. the gtk stock item without the stock prefix (since 4.2)
4. a boolean, if true the button is set as the default action (since 4.2)

Here is a screenshot of this form:



Result: call a method to determine which state is next

```
def _check_refund(self, cr, uid, data, context):
    ...
    return datas['form']['refund_id'] and 'wait_invoice' or 'end'
    ...
    'result': {'type':'choice', 'next_state':_check_refund}
```

Result: print a report

```
def _get_invoice_id(self, uid, datas):
    ...
    return {'ids': [...]}
    ...
    'actions': [_get_invoice_id],
    'result': {'type':'print',
               'report':'account.invoice',
               'get_id_from_action': True,
               'state':'check_refund'}
```

Result: client run an action

```
def _makeInvoices(self, cr, uid, data, context):
    ...
    return {
        'domain': "[('id','in', ["+', '.join(map(str,newinv))+"])]",
        'name': 'Invoices',
        'view_type': 'form',
        'view_mode': 'tree,form',
```

```

        'res_model': 'account.invoice',
        'view_id': False,
        'context': "{'type':'out_refund'}",
        'type': 'ir.actions.act_window'
    }

    ...

    'result': {'type': 'action',
    'action': _makeInvoices,
    'state': 'end'}

```

The result of the function must be an all the fields of an ir.actions.\* Here it is an ir.action.act\_window, so the client will open a new tab for the objects account.invoice For more information about the fields used click here.

It is recommended to use the result of a read on the ir.actions object like this:

```

def _account_chart_open_window(self, cr, uid, data, context):
    mod_obj = pooler.get_pool(cr.dbname).get('ir.model.data')
    act_obj = pooler.get_pool(cr.dbname).get('ir.actions.act_window')

    result = mod_obj._get_id(cr, uid, 'account', 'action_account_tree')
    id = mod_obj.read(cr, uid, [result], ['res_id'])[0]['res_id']
    result = act_obj.read(cr, uid, [id])[0]
    result['context'] = str({'fiscalyear': data['form']['fiscalyear']})
    return result

    ...

    'result': {'type': 'action',
    'action': _account_chart_open_window,
    'state': 'end'}

```



# SPECIFICATION

## 18.1 Form

```
_form = '''<?xml version="1.0"?>
<form string="Your String">
    <field name="Field 1"/>
    <newline/>
    <field name="Field 2"/>
</form>'''
```

## 18.2 Fields

### 18.2.1 Standard

Field **type**: char, integer, boolean, float, date, datetime

```
_fields = {
    'str_field': {'string':'product name', 'type':'char', 'readonly':True},
}
```

- **string**: Field label (required)
- **type**: (required)
- **readonly**: (optional)

### 18.2.2 Relational

Field **type**: one2one, many2one, one2many, many2many

```
_fields = {
    'field_id': {'string':'Write-Off account', 'type':'many2one', 'relation':'account.account'}
}
```

- **string**: Field label (required)
- **type**: (required)
- **relation**: name of the relation object

### 18.2.3 Selection

```
Field type: selection

_fields = {
    'field_id': {
        'string': "Date/Period Filter",
        'type': 'selection',
        'selection': [ ('bydate', 'By Date'),
                      ('byperiod', 'By Period'),
                      ('all', 'By Date and Period'),
                      ('none', 'No Filter')],
        'default': lambda *a:'none'
    },
}
```

- **string:** Field label (required)
- **type:** (required)
- **selection:** key and values for the selection field

# ADD A NEW WIZARD

To create a new wizard, you must:

- **create the wizard definition in a .py file**
  - wizards are usually defined in the wizard subdirectory of their module as in server/bin/addons/module\_name/wizard/your\_wizard\_name.py
- add your wizard to the list of import statements in the \_\_init\_\_.py file of your module's wizard subdirectory.
- declare your wizard in the database

The declaration is needed to map the wizard with a key of the client; when to launch which client. To declare a new wizard, you need to add it to the module\_name\_wizard.xml file, which contains all the wizard declarations for the module. If that file does not exist, you need to create it first.

Here is an example of the account\_wizard.xml file;

```
<?xml version="1.0"?>
<openerp>
    <data>
        <delete model="ir.actions.wizard" search="[(‘wiz_name’, ‘like’, ‘account.’)]" />
        <wizard string="Reconcile Transactions" model="account.move.line"
            name="account.move.line.reconcile" />
        <wizard string="Verify Transac steptions" model="account.move.line"
            name="account.move.line.check" keyword="tree_but_action" />
        <wizard string="Verify Transactions" model="account.move.line"
            name="account.move.line.check" />
        <wizard string="Print Journal" model="account.account"
            name="account.journal" />
        <wizard string="Split Invoice" model="account.invoice"
            name="account.invoice.split" />
        <wizard string="Refund Invoice" model="account.invoice"
            name="account.invoice.refund" />
    </data>
</openerp>
```

Attributes for the wizard tag:

- **id:** Unique identifier for this wizard.
- **string:** The string which will be displayed if there are several wizards for one resource. (The user will be presented a list with the wizards' names).
- **model:** The name of the **model** where the data needed by the wizard is.
- **name:** The name of the wizard. It is used internally and should be unique.
- **replace (optional):** Whether or not the wizard should override **all** existing wizards for this model. Default value: False.
- **menu (optional):** Whether or not (True|False) to link the wizard with the 'gears' button (i.e. show the button or not). Default value: True.

- **keyword (optional): Bind the wizard to another action (print icon, gear icon, ...). Possible values for the keyword attribute are:**
  - **client\_print\_multi**: the print icon in a form
  - **client\_action\_multi**: the ‘gears’ icon in a form
  - **tree\_but\_action**: the ‘gears’ icon in a tree view (with the shortcuts on the left)
  - **tree\_but\_open**: the double click on a branch of a tree (with the shortcuts on the left). For example, this is used, to bind wizards in the menu.

### **\_\_openerp\_\_.py**

If the wizard you created is the first one of its module, you probably had to create the modulename\_wizard.xml file yourself. In that case, it should be added to the update\_xml field of the \_\_openerp\_\_.py file of the module.

Here is, for example, the \_\_openerp\_\_.py file for the account module.

```
{
    "name": "OpenERP Accounting",
    "version": "0.1",
    "depends": ["base"],
    "init_xml": ["account_workflow.xml", "account_data.xml"],
    "update_xml": ["account_view.xml", "account_report.xml", "account_wizard.xml"],
}
```

# OSV\_MEMORY WIZARD SYSTEM

To develop osv\_memory wizard, just create a normal object, But instead of inheriting from osv.osv, Inherit from osv.osv\_memory. Methods of “wizard” are in object and if the wizard is complex, You can define workflow on object. osv\_memory object is managed in memory instead of storing in postgresql.

That's all, nothing more than just changing the inherit. These wizards can be defined at any location unlike addons/modulename/modulename\_wizard.py. Historically, the \_wizard prefix is for actual (old-style) wizards, so there might be a connotation there, the “new-style” osv\_memory based “wizards” are perfectly normal objects (just used to emulate the old wizards, so they don't really match the old separations. Furthermore, osv\_memory based “wizards” tend to need more than one object (e.g. one osv\_memory object for each state of the original wizard) so the correspondence is not exactly 1:1.

So what makes them looks like ‘old’ wizards?

- In the action that opens the object, you can put

```
<field name="target">new</field>
```

It means the object will open in a new window instead of the current one.

- On a button, you can use <button special="cancel" .../> to close the window.

Example : In project.py file.

```
class config_compute_remaining(osv.osv_memory):  
    _name='config.compute.remaining'  
    def _get_remaining(self,cr, uid, ctx):  
        if 'active_id' in ctx:  
            return self.pool.get('project.task').browse(cr,uid,ctx['active_id']).remaining_hours  
        return False  
    _columns = {  
        'remaining_hours' : fields.float('Remaining Hours', digits=(16,2)),  
    }  
    _defaults = {  
        'remaining_hours': _get_remaining  
    }  
    def compute_hours(self, cr, uid, ids, context=None):  
        if 'active_id' in context:  
            remaining_hrs=self.browse(cr,uid,ids)[0].remaining_hours  
            self.pool.get('project.task').write(cr,uid,context['active_id'],  
                                              {'remaining_hours' : remaining_hrs})  
        return {  
            'type': 'ir.actions.act_window_close',  
        }  
config_compute_remaining()
```

- View is same as normal view (Note buttons).

Example :

```

<record id="view_config_compute_remaining" model="ir.ui.view">
    <field name="name">Compute Remaining Hours </field>
    <field name="model">config.compute.remaining</field>
    <field name="type">form</field>
    <field name="arch" type="xml">
        <form string="Remaining Hours">
            <separator colspan="4" string="Change Remaining Hours"/>
            <newline/>
            <field name="remaining_hours" widget="float_time"/>
            <group col="4" colspan="4">
                <button icon="gtk-cancel" special="cancel" string="Cancel"/>
                <button icon="gtk-ok" name="compute_hours" string="Update" type="object"/>
            </group>
        </form>
    </field>
</record>

```

- Action is also same as normal action (don't forget to add target attribute)

Example :

```

<record id="action_config_compute_remaining" model="ir.actions.act_window">
    <field name="name">Compute Remaining Hours</field>
    <field name="type">ir.actions.act_window</field>
    <field name="res_model">config.compute.remaining</field>
    <field name="view_type">form</field>
    <field name="view_mode">form</field>
    <field name="target">new</field>
</record>

```

# OSV\_MEMORY CONFIGURATION ITEM

Sometimes, your addon can't do with configurable defaults and needs upfront configuration settings to work correctly. In these cases, you want to provide a configuration wizard right after installation, and potentially one which can be re-run later if needed.

Up until 5.0, OpenERP had such a facility but it was hardly documented and a very manual, arduous process. A simpler, more straightforward solution has been implemented for those needs.

## 21.1 The basic concepts

The new implementation provides a base behavior `osv_memory` object from which you need to inherit. This behavior handles the flow between the configuration items of the various extensions, and inheriting from it is therefore mandatory.

There is also an inheritable view which provides a basic canvas, through mechanisms which will be explained later it's highly customizable. It's therefore strongly suggested that you should inherit from that view from yours as well.

## 21.2 Creating a basic configuration item

### 21.2.1 Your configuration model

First comes the creation of the configuration item itself. This is a normal `osv_memory` object with a few constraints:

- it has to inherit from `res.config`, which provides the basic configuration behaviors as well as the base event handlers and extension points
- it has to provide an `execute` method. This method will be called when validating the configuration form and contains the validation logic. It shouldn't return anything.

```
class my_item_config(osv.osv_memory):
    _name = 'my.model.config'
    _inherit = 'res.config' # mandatory

    _columns = {
        'my_field': fields.char('Field', size=64, required=True),
    }

    def execute(self, cr, uid, ids, context=None):
        'do whatever configuration work you need here'
my_item_config()
```

## 21.2.2 Your configuration view

Then comes the configuration form. OpenERP provides a base view which you can inherit so you don't have to deal with creating buttons and handling the progress bar (which should be displayed at the bottom left of all initial configuration dialogs). It's very strongly recommended that you use this base view.

Simply add an `inherit_id` field to a regular `ir.ui.view` and set its value to `res_config_view_base`:

```
<record id="my_config_view_form" model="ir.ui.view">
    <field name="name">my.item.config.view</field>
    <!-- this is the model defined above -->
    <field name="model">my.model.config</field>
    <field name="type">form</field>
    <field name="inherit_id" ref="base.res_config_view_base"/>
    ...
</record>
```

While this could be used as-is, it would display an empty dialog with a progress bar and two buttons which isn't of much interest. `res_config_view_base` has a special group hook which you should replace with your own content like so:

```
<field name="arch" type="xml">
    <group string="res_config_contents" position="replace">
        <!-- your content should be inserted within this, the string
            attribute of the previous group is used to easily find
            it for replacement -->
        <label colspan="4" align="0.0" string="
            Configure this item by defining its field"/>
        <field colspan="2" name="my_field"/>
    </group>
</field>
```

## 21.2.3 Opening your window

The next step is to create the `act_window` which links to the configuration model and the view:

```
<record id="my_config_window" model="ir.actions.act_window">
    <field name="name">My config window</field>
    <field name="type">ir.actions.act_window</field>
    <field name="res_model">my.model.config</field>
    <field name="view_type">form</field>
    <field name="view_id" ref="my_config_view_form"/>
    <field name="view_mode">form</field>
    <field name="target">new</field>
</record>
```

Note that the `name` field of this `act_window` will be displayed when listing the various configuration items in the Config Wizard Steps submenu (in Administration > Configuration > Configuration Wizards).

## 21.2.4 Registering your action

Finally comes actually registering the configuration item with OpenERP. This is done with an `ir.actions.todo` object, which mandates a single `action_id` field referencing the `act_window` created previously:

```
<record id="my_config_step" model="ir.actions.todo">
    <field name="action_id" ref="my_config_window"/>
</record>
```

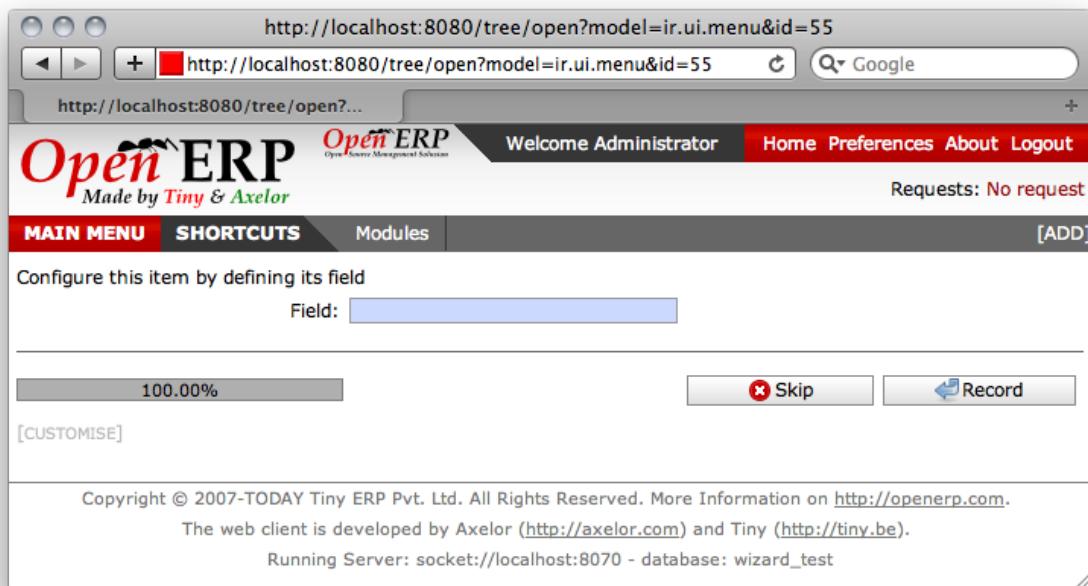
`ir.actions.todo` also has 3 optional fields:

**sequence (default: 10)** The order in which the different steps are to be executed, lowest first.

**active (default: True)** An inactive step will not be executed on the next round of configuration.

**state (default: 'open')** The current state for the configuration step, mostly used to register what happened during its execution. The possible values are 'open', 'done', 'skip' and 'cancel'.

The result at this point is the following:



## 21.3 Customizing your configuration item

While your current knowledge is certainly enough to configure your addon, a bit of good customization can be the difference between a good user experience and a great user experience.

### 21.3.1 More extensive view customization

As you might have noticed from the previous screen shot, by default your configuration window doesn't have a *title*, which isn't a problem but doesn't look very good either.

Before setting a title, a small modification to the existing view is needed though: the existing `group` needs to be wrapped in a `data` element so it's possible to customize more than one item of the parent view:

```
<record id="my_config_view_form" model="ir.ui.view">
    <field name="name">my.item.config.view</field>
    <!-- this is the model defined above -->
    <field name="model">my.model.config</field>
    <field name="type">form</field>
    <field name="inherit_id">res_config_view_base</field>
    <field name="arch" type="xml">
        <data>
            <group string="res_config_contents" position="replace">
                <!-- your content should be inserted within this, the
                    string attribute of the previous group is used to
                    easily find it for replacement
                -->
                <label colspan="4" align="0.0" string=""

```

```

        Configure this item by defining its field
    ">
        <field colspan="2" name="my_field"/>
    </group>
</data>
</field>
</record>
```

Then it becomes possible to alter the `string` attribute of the original `form` by adding the following code within the `data` element (in this case, probably before `group`):

```

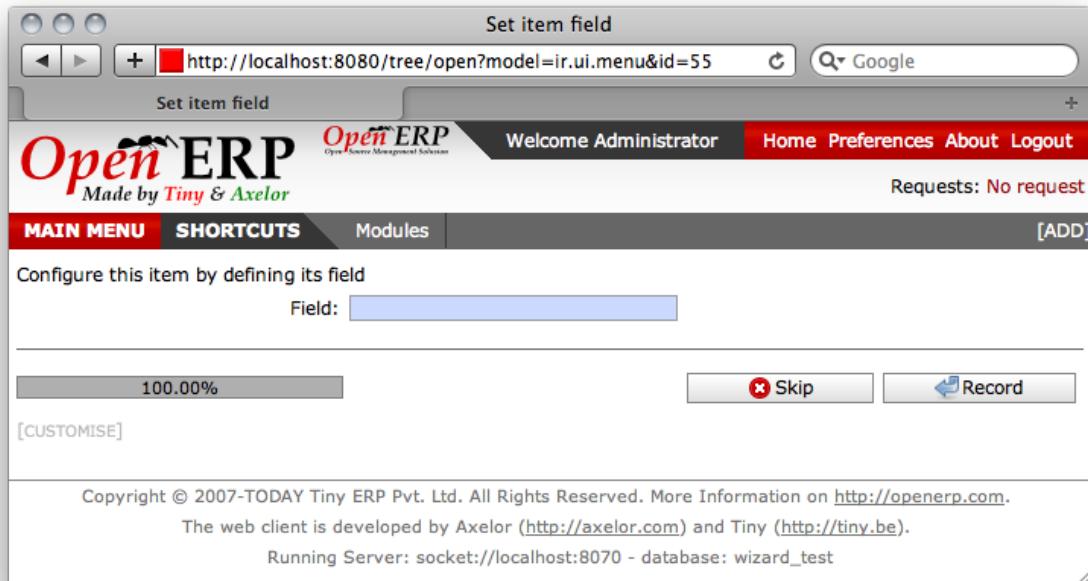
<!-- position=attributes is new and is used to alter the
     element's attributes, instead of its content -->
<form position="attributes">
    <!-- set the value of the 'string' attribute -->
    <attribute name="string">Set item field</attribute>
</form>
```

**Warning:**

*Comments in view overload*

*At this point (December 2009) OpenERP cannot handle comments at the toplevel of the view element overload. When testing or reusing these examples, remember to strip out the comments or you will get runtime errors when testing the addon.*

With this, the configuration form gets a nice title:



More interesting customizations might be to alter the buttons provided by `res_config_view_base` at the bottom of the dialog: remove a button (if the configuration action shouldn't be skipped), change the button labels, ...

Since no specific hooks are provided for these alterations, they require the use of xpath selectors (using the `xpath` element).

Removing the Skip button and changing the label of the Record button to Set, for instance, would be done by adding the following after the `group` element:

```

<!-- select the button 'action_skip' of the original template
     and replace it by nothing, removing it -->
```

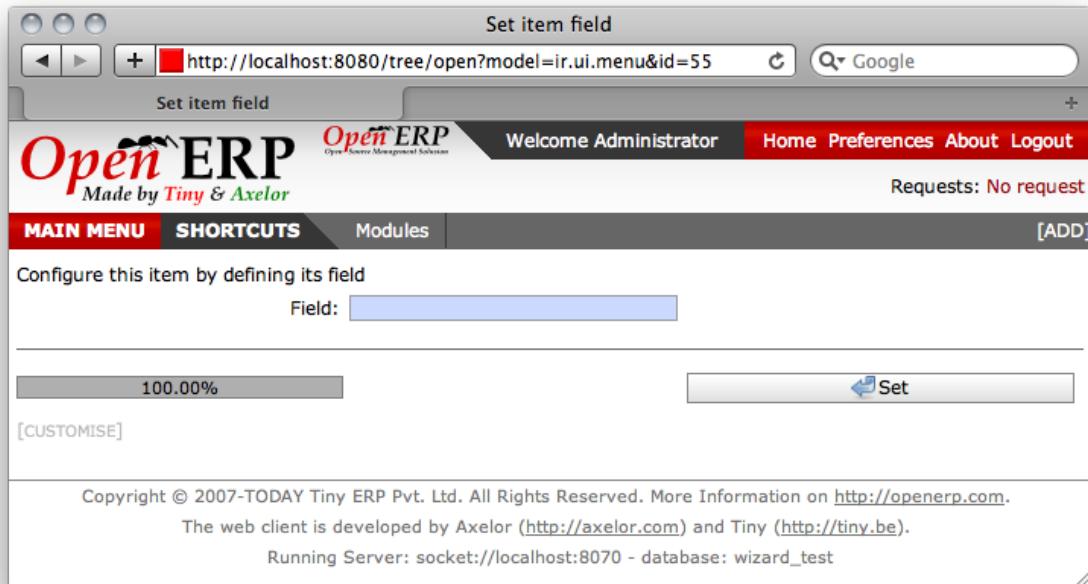
```

<xpath expr="/button[@name='action_skip']"
       position="replace"/>

<!-- select the button 'action_next' -->
<xpath expr="/button[@name='action_next']"
       position="attributes">
    <!-- and change the attribute 'string' to 'Set' -->
    <attribute name="string">Set</attribute>
</xpath>

```

and yield:



It is also possible to use this method to change the name of the button, and thus the method invoked on the object (though that isn't necessarily recommended).

### 21.3.2 Model customization

Though most of the requirements should be easy to fulfill using the provided `execute` method hook, some addon-specific requirements are a bit more complex. `res.config` should be able to provide all the hooks necessary for more complex behaviors.

#### Ignoring the next step

Ultimately, the switch to the next configuration item is done by calling the `self.next` method of `res.config`<sup>1</sup>. This is the last thing the base implementations of `action_next` and `action_skip` do. But in some cases, looping on the current view or implementing a workflow-like behavior is needed. In these cases, you can simply return a dictionary from `execute`, and `res.config` will jump to that view instead of the one returned by `self.next`.

This is what the user creation item does, for instance, to let the user create several new users in a row.

---

<sup>1</sup> This isn't completely true, as you will see when [Customizing your configuration item](#)

## Performing an action on skipping

As opposed to `action_next` which requires that `execute` be implemented by the children classes, `action_skip` comes fully implemented in `res.config`. But in the case where the child model needs to perform an action upon skipping discovery, it also provides a hook method called `cancel` which you can overload in a way similar to `execute`. Its behavior is identical to `execute`'s: not only is `next` called automatically at the end of `cancel` but it also gives the possibility of ignoring the next step.

## Alternative actions

It's also possible to either overload `action_next` and `action_skip` or, more useful, to implement more actions than these two, if more than two buttons are needed for instance.

In this case, please remember that you should always provide a way to reach `self.next` to the user, in order for him to be able to configure the rest of his addons.

# 21.4 `res.config`'s public API

All of the public API methods take the standard OpenERP set of arguments: `self, cr, uid, ids` and `context`.

## 21.4.1 `execute`

Hook method called in case the `action_next` button (default label: Record) is clicked. Should not return *anything* unless you want to display another view than the next configuration item. Returning anything other than a view dictionary will lead to undefined behaviors.

It is mandatory to overload it. Failure to do so will result in a `NotImplementedError` being raised at runtime.

The default `res.config` implementation should not be called in the overload (don't use `super`).

## 21.4.2 `cancel`

Hook method called in case the `action_skip` button (default label: Skip) is clicked. Its behavior is the same as `execute`'s, except it's not mandatory to overload it.

## 21.4.3 `next`

Method called to fetch the todo (and the corresponding action) for the next configuration item. It can be overloaded if the configuration item needs custom behavior common to all events.

If overloaded, the default `res.config` implementation must be called and its result returned in order to get and execute the next configuration item.

## 21.4.4 `action_next` and `action_skip`

Event handler for the buttons of the base view, overloading them should never be necessary but in case it's needed the default `res.config` implementation should be called (via `super`) and its result returned.

# GUIDELINES ON HOW TO CONVERT OLD-STYLE WIZARD TO NEW OSV\_MEMORY STYLE

## 22.1 OSV Memory Wizard

provide important advantages over the pre-5.0 wizard system, with support features that were difficult to implement in wizards previously, such as:

1. inheritance
2. workflows
3. complex relation fields
4. computed fields
5. all kind of views (lists, graphs, ...)

The new wizards are also easier and more intuitive to write as they make use of the same syntax as other osv objects and views.

This section will highlight the main steps usually required when porting a classical wizard to the new osv\_memory wizard system. For more details about the osv\_memory wizard see also section XXX.

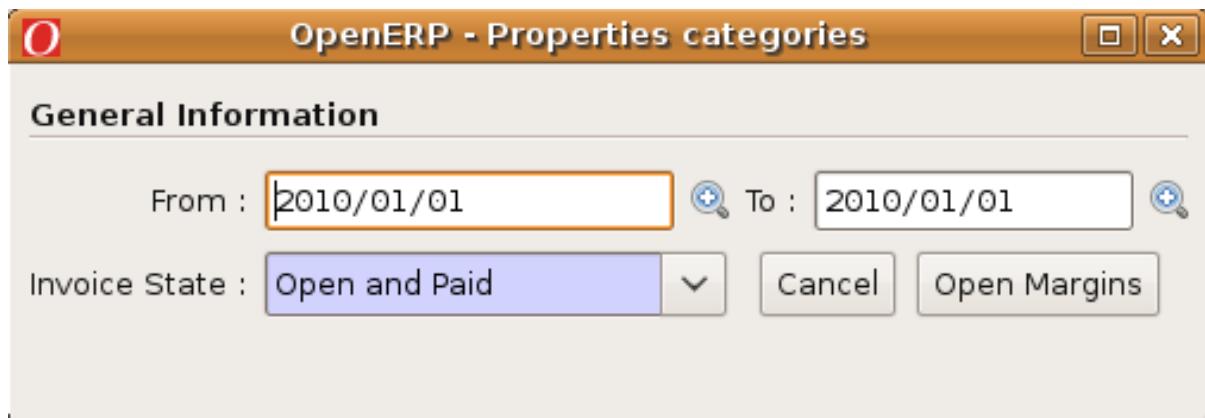
Basically the idea is to create a regular osv object to hold the data structures and the logic of the wizard, but instead of inheriting from osv.osv, you inherit from osv.osv\_memory. The methods of the old-style wizard will be moved as methods of the osv\_memory object, and the various views changed into real views defined on the model of the wizard.

If the wizard is complex, you could even define a workflow on the wizard object (see section XXX for details about workflows)

Using a very simple wizard as an example, here is a step-by-step conversion to the new osv\_memory system:

## 22.2 Steps

1. Create a new object that extends osv\_memory, including the required fields and methods:



```

def _action_open_window(self, cr, uid, data, context):
    .

.

class product_margins(wizard.interface):
    form1 = '''<?xml version="1.0"?>
<form string="View Stock of Products">
    <separator string="Select " colspan="4"/>
    <field name="from_date"/>
    <field name="to_date"/>
    <field name="invoice_state"/>
</form>'''

    form1_fields = {
        'from_date': {
            'string': 'From',
            'type': 'date',
            'default': lambda *a:time.strftime('%Y-01-01'),
        },
        'to_date': {
            'string': 'To',
            'type': 'date',
            'default': lambda *a:time.strftime('%Y-12-31'),
        },
        'invoice_state': {
            'string': 'Invoice State',
            'type': 'selection',
            'selection': [ ('paid','Paid'), ('open_paid','Open and Paid'), ('draft_open_paid','Draft Open Paid') ],
            'required': True,
            'default': lambda *a:"open_paid",
        },
    }

    states = {
        'init': {
            'actions': [],
            'result': {'type': 'form', 'arch':form1, 'fields':form1_fields, 'state': [('end', 'Cancel')]}
        },
        'open': {
            'actions': [],
            'result': {'type': 'action', 'action': _action_open_window, 'state':'end'}
        }
    }
product_margins('product.margins')

```

## 22.3 New Wizard File : <<module\_name>>\_<<filename>>.py

```
class product_margin(osv.osv_memory):
    """
    Product Margin
    """
    _name = 'product.margin'
    _description = 'Product Margin'

    def _action_open_window(self, cr, uid, ids, context):
        .
        .
        .

    _columns = {
        #TODO : import time required to get correct date
        'from_date': fields.date('From'),
        #TODO : import time required to get correct date
        'to_date': fields.date('To'),
        'invoice_state': fields.selection([
            ('paid','Paid'),
            ('open_paid','Open and Paid'),
            ('draft_open_paid','Draft, Open and Paid'),
        ],'Invoice State', select=True, required=True),
    }
    _defaults = {
        'from_date': lambda *a:time.strftime('%Y-01-01'),
        'to_date': lambda *a:time.strftime('%Y-01-01'),
        'invoice_state': lambda *a:"open_paid",
    }
product_margin()
```

Convert the views into real view records defined on the model of your wizard:

## 22.4 Old Wizard File : wizard\_product\_margin.py

```
form1 = '''<?xml version="1.0"?>
<form string="View Stock of Products">
    <separator string="Select " colspan="4"/>
    <field name="date"/>
    <field name="invoice_state"/>
</form>'''
```

## 22.5 New Wizard File : wizard/<<module\_name>>\_<<filename>>\_view.xml

```
<record id="product_margin_form_view" model="ir.ui.view">
    <field name="name">product.margin.form</field>
    <field name="model">product.margin</field>
    <field name="type">form</field>
    <field name="arch" type="xml">
        <form string="Properties categories">
            <separator colspan="4" string="General Information"/>
            <field name="from_date" />
            <field name="to_date" />
            <field name="invoice_state" />
            <group col="4" colspan="2">
                <button special="cancel" string="Cancel" type="object"/>

```

```

        <button name="_action_open_window" string="Open Margins" type="object"
      </group>
    </form>
  </field>
</record>
```

## 22.6 Default\_focus attribute

```
<button name="_action_open_window" string="Open Margins" type="object" default_focus="1"/>
```

**default\_focus="1"** is a new attribute added in 5.2. While opening wizard default control will be on the widget having this attribute. There must be only one widget on a view having this attribute = 1 otherwise it will raise exception.

Note: For all states in the old wizard, we need to create buttons in new approach.

2. To open the new wizard, you need to register an action that opens the first view on your wizard object. You will need to do the same for each view if your wizard contains several views. To make the view open in a pop-up window you can add a special target='new' field in the action:

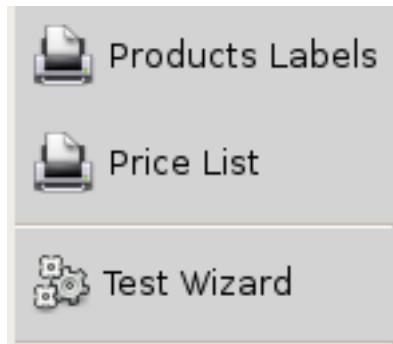
```
<act_window name="Open Margin"
  res_model="product.margin"
  src_model="product.product"
  view_mode="form"
  target="new"
  key2="client_action_multi"
  id="product_margin_act_window"/>
```

key2="client\_action\_multi" : While using it in the act\_window, wizard will be added in the

1. Action



2. Sidebar



If key2 is omitted then it will be displayed only in sidebar.

Note: The "src\_model" attribute is only required if you want to put the wizard in the side bar of an object, you can leave it out, for example if you define an action to open the second view of a wizard.

3. You can register this new action as a menuitem or in the context bar of any object by using a <menuitem> or <act\_window> record instead of the old <wizard> tag that can be removed:

## 22.7 In Menu Item

To open a wizard view via a menuitem you can use the following syntax for the menu, using the XML id of the corresponding act\_window.

```
<menuitem id="main" name="OSV Memory Wizard Test"/>
<menuitem
    action="product_margin_act_window"
    id="menu_product_act"
    parent="main" />
```

4. To open a wizard view via a button in another form you can use the following syntax for the button, using the XML id of the corresponding act\_window. This can be used to have multiple steps in your wizard:

```
<button name="% (product_margin.product_margin_act_window) d"
        string="Test Wizard" type="action" states="draft"/>
```

5. Finally, you need to cleanup the module, update the python \_\_init\_\_.py files if you have changed the python file name for the wizard, and add your new XML files in the update\_xml list in the \_\_openerp\_\_.py file.



# **Part VI**

# **Reports**



There are mainly three types of reports in OpenERP:

- OpenOffice.org reports
- RML reports
- custom reports (based on PostgreSQL views and displayed within the interface)

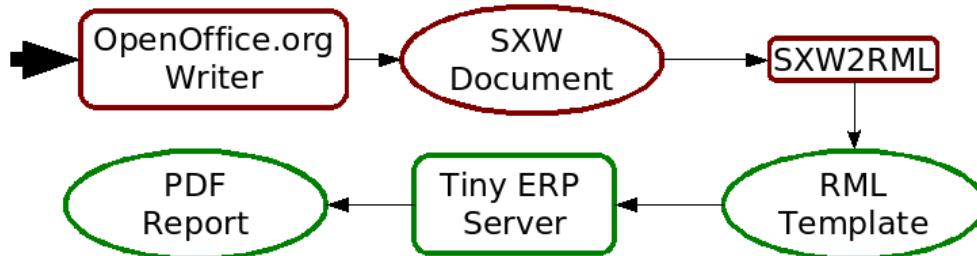
This chapter mainly describes OpenOffice.org reports, and then XSL:RML reports. Custom reports are described in section Advanced Modeling - Reporting With PostgreSQL Views.



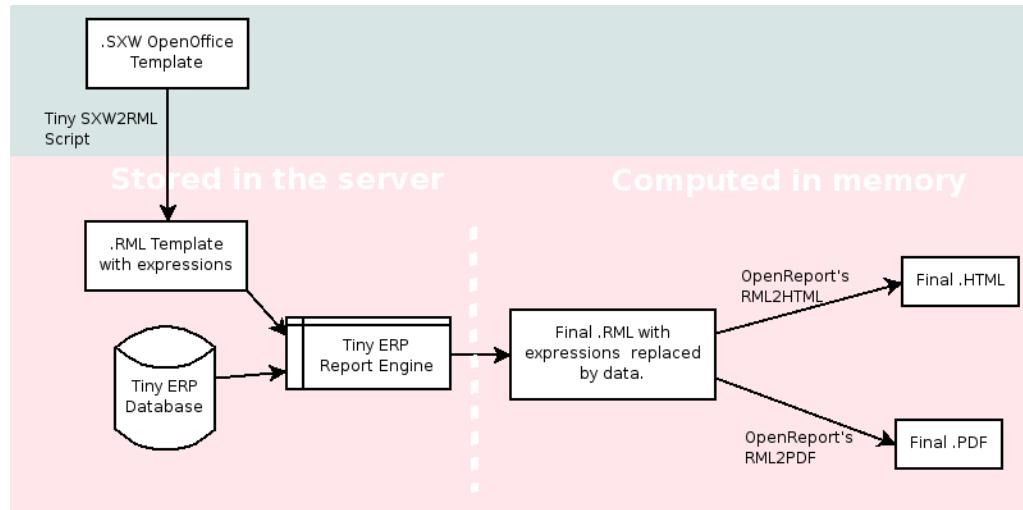
# OPENOFFICE.ORG REPORTS

## The document flow

OpenOffice.org reports are the most commonly used report formats. OpenOffice.org Writer is used (in combination with [[1]]) to generate a RML template, which in turn is used to generate a pdf printable report.



## The internal process



## The .SXW template file

- We use a .SXW file for the template, which is the OpenOffice 1.0 format. The template includes expressions in brackets or OpenOffice fields to point where the data from the OpenERP server will be filled in. This document is only used for developers, as a help-tool to easily generate the .RML file. OpenERP does not need this .SXW file to print reports.

## The .RML template

- We generate a .RML file from the .SXW file using Open SXW2RML. A .RML file is a XML format that represent a .PDF document. It can be converted to a .PDF after. We use RML for more easy processing: XML syntax seems to be more common than PDF syntax.

## The report engine

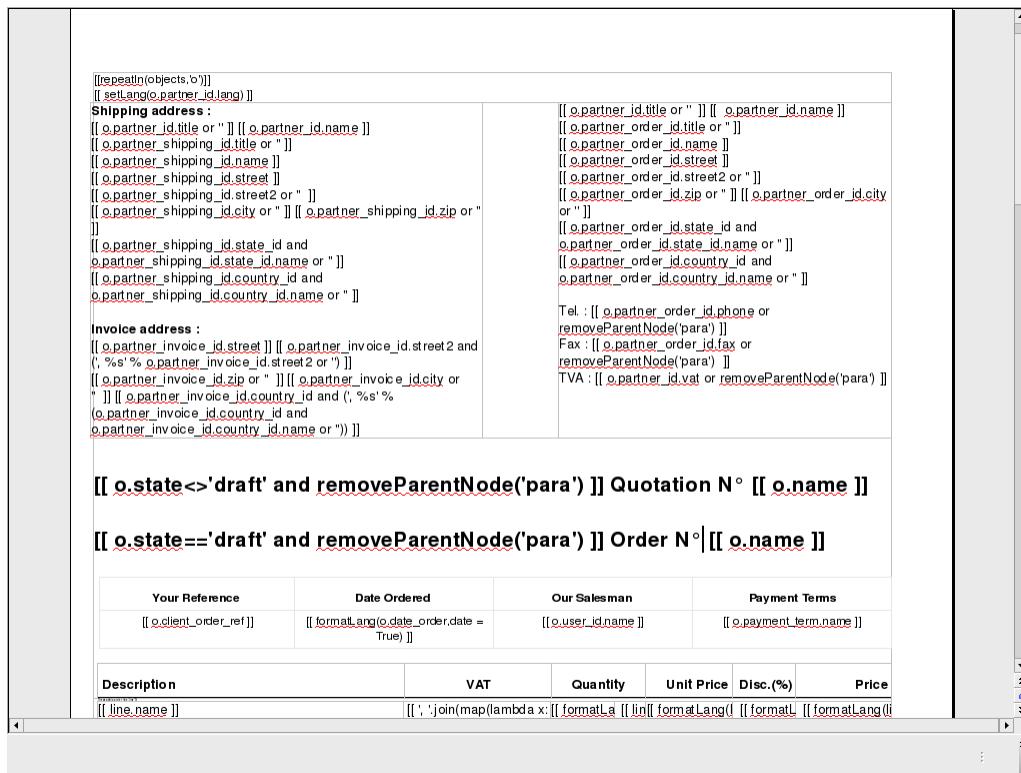
- The Open Report Engine process the .RML file inserting data from the database at each expression.
- in the .RML file will be replaced by the name of the country of the partner of the printed invoice. This report engine produce the same .RML file where all expressions have been replaced by real data.

### The final document

- Finally the .RML file is converted to PDF or HTML as needed, using OpenReport's scripts.

## 23.1 Creating a SXW

You can design reports using *OpenOffice*. Here, as an example, is the file `server/bin addons/sale/report/order.sxw`.



## 23.2 Dynamic content in OpenOffice reports

### Dynamic content

In the .SXW/.RML reports, you can put some Python code that accesses the OpenERP objects in brackets. The context of the code (the variable's values you can use) is the following:

#### Available variables

Here are Python objects/variables available:

- objects** : the list of objects to be printed (invoices for example).
- data** : comes from the wizard
- time** : the Python time module (see Python documentation for more information).
- user** : the user object launching the report.

#### Available functions

Here are Python functions you can use:

- **setLang('fr')** : change the language used in automated translation (fields...).
- **repeatIn(list, varname[, tagname])** : repeat the current part of the template (whole document, current section, current row in the table) for each object in the list. Use varname in the template's tags. Since versions 4.1.X, you can use an optional third argument that is the name of the .RML tag you want to loop on.
- **setTag('para','xpre')** : replace the enclosing RML tag (usually 'para') with an other (xpre is a preformatted paragraph), in the (converted from sxw)rml document (?)
- **removeParentNode('tr')** : removes the parent node of type 'tr', this parameter is usually used together with a conditional (see examples below)

Example of useful tags:

- `[[ repeatIn(objects,'o') ]]` : Loop on each objects selected for the print
- `[[ repeatIn(o.invoice_line,'l') ]]` : Loop on every line
- `[[ repeatIn(o.invoice_line,'l', 'td') ]]` : Loop on every line and make a new table cell for each line.
- `[[ (o.prop=='draft')and 'YES' or 'NO' ]]` : Print YES or NO according the field 'prop'
- `[[ round(o.quantity * o.price * 0.9, 2) ]]` : Operations are OK.
- `[[ '%07d' % int(o.number) ]]` : Number formatting
- `[[ reduce(lambda x, obj: x+obj.qty , list , 0 ) ]]` : Total qty of list (try "objects" as list)
- `[[ user.name ]]` : user name
- `[[ setLang(o.partner_id.lang) ]]` : Localized printings
- `[[ time.strftime("%d/%m/%Y") ]]` : Show the time in format=dd/MM/YYYY, check python doc for more about "%d", ...
- `[[ time.strftime(time.ctime()[0:10]) ]]` or `[[ time.strftime(time.ctime()[-4:]) ]]` : Prints only date.
- `[[ time.ctime() ]]` : Prints the actual date & time
- `[[ time.ctime().split()[3] ]]` : Prints only time
- `[[ o.type in ['in_invoice', 'out_invoice'] and 'Invoice' or removeParentNode('tr') ]]` : If the type is 'in\_invoice' or 'out\_invoice' then the word 'Invoice' is printed, if it's neither the first node above it of type 'tr' will be removed.

One more interesting tag: if you want to print out the creator of an entry (create\_uid) or the last one who wrote on an entry (write\_uid) you have to add something like this to the class your report refers to:

```
'create_uid': fields.many2one('res.users', 'User', readonly=1)
```

and then in your report it's like this to print out the corresponding name:

```
o.create_uid.name
```

Sometimes you might want to print out something only if a certain condition is met. You can construct it with the python logical operators "not", "and" and "or". Because every object in python has a logical value (TRUE or FALSE) you can construct something like this:

```
(o.prop=='draft') and 'YES' or 'NO' #prints YES or NO
```

It works like this: *and* is higher priority than *or*, so that expression is equivalent to this one:

```
((o.prop=='draft') and 'YES') or 'NO'
```

If *o.prop* is 'draft', then it evaluates like this:

1. *o.prop == 'draft'* is *True*.
2. *True and 'YES'* is '*YES*'. Because the left side is a "true" value, the *and* expression evaluates to the right side.

3. ‘YES’ or ‘NO’ is ‘YES’. Because the left side is a “true” value, the *or* expression short cuts and ignores the right side. It evaluates to the left side.

If *o.prop* is something else like ‘confirm’, then it evaluates like this:

1. *o.prop == ‘draft’* is *False*.
2. *False and ‘YES’* is *False*. Because the left side is a “false” value, the *and* expression short cuts and ignores the right side. It evaluates to the left side.
3. *False or ‘NO’* is ‘NO’. Because the left side is a “false” value, the *or* expression evaluates to the right side.

One can use very complex structures. To learn more, see the python manuals section on [Python’s boolean operators](#).

python function “filter” can... filter: try something like:

```
repeatIn(filter( lambda l: l.product_id.type=='service' ,o.invoice_line), 'line')
```

for printing only product with type=’service’ in a line’s section.

To display binary field image on report (to be checked)

```
[ [ setTag('para','image',{'width':'100.0','height':'80.0'}) ] ] o.image or setTag('image','para')
```

## 23.3 SXW2RML

### 23.3.1 Open Report Manual

#### About

The OpenERP’s report engine.

Open Report is a module that allows you to render high quality PDF document from an OpenOffice template (.sxw) and any relational database. It can be used as an OpenERP module or as a standalone program.

#### SXW to RML script setup - Windows users

In order to use the ‘tiny\_sxw2rml.py’ Python script you need the following packages installed:

- Python (<http://www.python.org>)
- ReportLab (<http://www.reportlab.org/>)/(Installation)
- Libxml for Python (<http://users.skynet.be/sbi/libxml-python>)

#### SXW to RML script setup - Linux (Open source) users

The **tiny\_sxw2rml.py** can be found in the **base\_report\_designer** OpenERP module at this location:

```
server/bin addons/base_report_designer/wizard/tiny_sxw2rml/tiny_sxw2rml.py
```

Ensure normalized\_oo2rml.xsl is available to tiny\_sxw2rml otherwise you will get an error like:

- failed to load external entity normalized\_oo2rml.xsl

## Running tiny\_sxw2rml

When you have all that installed just edit your report template and run the script with the following command:

```
tiny_sxw2rml.py template.sxw > template.rml
```

Note: **tiny\_sxw2rml.py** help suggests that you specify the output file with: “-o OUTPUT” but this does not seem to work as of V0.9.3

## 23.4 OpenERP Server PDF Output

### 23.4.1 Server PDF Output

#### About

To generate the pdf from the rml file, OpenERP needs a rml parser.

#### Parser

The parsers are generally put into the report folder of the module. Here is the code for the sale order report:

```
import time
from report import report_sxw

class order(report_sxw.rml_parse):
    def __init__(self, cr, uid, name, context):
        super(order, self).__init__(cr, uid, name, context)
        self.localcontext.update({
            'time': time,
        })

report_sxw.report_sxw('report.sale.order', 'sale.order',
                      'addons/sale/report/order.rml', parser=order, header=True)
```

The parser inherit from the **report\_sxw.rml\_parse** object and it add to the localcontext, the function time so it will be possible to call it in the report.

After an instance of **report\_sxw.report\_sxw** is created with the parameters:

- the name of the report
- the object name on which the report is defined
- the path to the rml file
- the parser to use for the report (by default rml\_parse)
- **the header to use from the company configuration**
  - ‘external’ (default)
  - ‘internal’
  - ‘internal landscape’
  - False - use the report’s own header

## The xml definition

To be visible from the client, the report must be declared in an xml file (generally: “module\_name”\_report.xml) that must be put in the **\_\_openerp\_\_.py** file

Here is an example for the sale order report:

```
<?xml version="1.0"?>
<openerp>
    <data>
        <report
            id="report_sale_order"
            string="Print Order"
            model="sale.order"
            name="sale.order"
            rml="sale/report/order.rml"
            auto="False"/>
            header="False"/>
    </data>
</openerp>
```

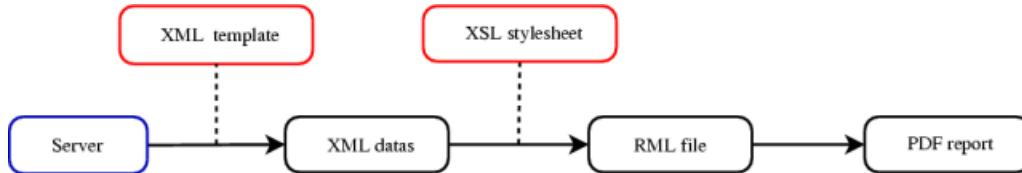
The arguments are:

- **id**: the id of the report like any xml tag in OpenERP
- **string**: the string that will be display on the Client button
- **model**: the object on which the report will run
- **name**: the name of the report without the first “report.”
- **rml**: the path to the rml file
- **auto**: boolean to specify if the server must generate a default parser or not
- **header**: allows to enable or disable the report header. To edit them for a specific company, go to: Administration -> Users -> Company's structure -> Companies. There, select and edit your company: the “Header/Footer” tab allows you to edit corporate header/footer.

# XSL:RML REPORTS

RML reports don't require programming but require two simple XML files to be written:

- a file describing the data to export (\*.xml)
- a file containing the presentation rules to apply to that data (\*.xsl)



The role of the XML template is to describe which fields of the resource have to be exported (by the server). The XSL:RML style sheet deals with the layout of the exported data as well as the “static text” of reports. Static text is referring to the text which is common to all reports of the same type (for example, the title of table columns).

## Example

Here is, as an example, the different files for the simplest report in the ERP.

Ref.	Name
pnk00	Tiny sprl
	ASUS
	Agrolait
	Banque Plein-Aux-As
	China Export
	Ditrib PC
	Ecole de Commerce de Liege
	Elec Import
	Maxtor
	Mediapole SPRL
os	Opensides sprl
	Tecsa sarl

## XML Template

```
<?xml version="1.0"?>

<ids>
<id type="fields" name="id">
    <name type="field" name="name"/>
    <ref type="field" name="ref"/>
```

```
</id>
</ids>
```

#### **XML data file (generated)**

```
<?xml version="1.0"?>

<ids>
<id>

    <name>Tiny sprl</name>
    <ref>pnk00</ref>

</id><id>

    <name>ASUS</name>
    <ref></ref>

</id><id>

    <name>Agrolait</name>
    <ref></ref>

</id><id>

    <name>Banque Plein-Aux-As</name>
    <ref></ref>

</id><id>

    <name>China Export</name>
    <ref></ref>

</id><id>

    <name>Distrib PC</name>
    <ref></ref>

</id><id>

    <name>Ecole de Commerce de Liege</name>
    <ref></ref>

</id><id>

    <name>Elec Import</name>
    <ref></ref>

</id><id>

    <name>Maxtor</name>
    <ref></ref>

</id><id>

    <name>Mediapole SPRL</name>
    <ref></ref>

</id><id>

    <name>Opensides sprl</name>
    <ref>os</ref>
```

```

</id><id>

    <name>Tecsas sarl</name>
    <ref></ref>

</id>
</ids>

```

### **XSL stylesheets**

```

<?xml version="1.0" encoding="utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/2001/XSL/Transform">

    <xsl:template match="/">

        <xsl:apply-templates select="ids"/>

    </xsl:template>

    <xsl:template match="ids">

        <document>

            <template pageSize="21cm,29.7cm">

                <pageTemplate>

                    <frame id="col1" x1="2cm" y1="2.4cm" width="8cm" height="26cm"/>
                    <frame id="col2" x1="11cm" y1="2.4cm" width="8cm" height="26cm"/>

                </pageTemplate>

            </template>

        </stylesheets>

        <blockTableStyle id="ids">

            <blockFont name="Helvetica-BoldOblique" size="12" start="0,0" stop="-1,0"/>
            <lineStyle kind="BOX" colorName="black" start="0,0" stop="-1,0"/>

            <lineStyle kind="BOX" colorName="black" start="0,0" stop="-1,-1"/>

        </blockTableStyle>

    </stylesheets>

    <story>

        <blockTable colWidths="2cm, 6cm" repeatRows="1" style="ids">

            <tr>

                <td t="1">Ref.</td>
                <td t="1">Name</td>

            </tr>
            <xsl:apply-templates select="id"/>

        </blockTable>

    </story>
</document>

```

```

</xsl:template>

<xsl:template match="id">

<tr>

<td><xsl:value-of select="ref"/></td>
<td><para><xsl:value-of select="name"/></para></td>

</tr>

</xsl:template>
</xsl:stylesheet>

```

### **Resulting RML file (generated)**

```

<?xml version="1.0"?>

<document>
  ...
<story>

  <blockTable colWidths="2cm, 6cm" repeatRows="1" style="ids">

    <tr>

      <td t="1">Ref.</td>
      <td t="1">Name</td>

    </tr>
    <tr>

      <td>pnk00</td>
      <td><para>Tiny sprl</para></td>

    </tr>
    <tr>

      <td></td>
      <td><para>ASUS</para></td>

    </tr>
    <tr>

      <td></td>
      <td><para>Agrolait</para></td>

    </tr>
    <tr>

      <td></td>
      <td><para>Banque Plein-Aux-As</para></td>

    </tr>
    <tr>

      <td></td>
      <td><para>China Export</para></td>

    </tr>
    <tr>

  </blockTable>
</story>

```

```

<td></td>
<td><para>Distrib PC</para></td>

</tr>
<tr>

<td></td>
<td><para>Ecole de Commerce de Liege</para></td>

</tr>
<tr>

<td></td>
<td><para>Elec Import</para></td>

</tr>
<tr>

<td></td>
<td><para>Maxtor</para></td>

</tr>
<tr>

<td></td>
<td><para>Mediapole SPRL</para></td>

</tr>
<tr>

<td>os</td>
<td><para>Opensides sprl</para></td>

</tr>
<tr>
<td></td>

<td><para>Tecsas sarl</para></td>

</tr>
</blockTable>

</story>

</document>

```

For more information on the formats used:

- [RML user guide](#)
- [XSL specification](#)
- [XSL tutorial](#)

All these formats are extensions of the [XML](#) specification.

## 24.1 XML Template

XML templates are simple XML files describing which fields among all available object fields are necessary for the report.

## 24.1.1 File format

Tag names can be chosen arbitrarily (it must be valid XML though). In the XSL file, you will have to use those names. Most of the time, the name of a tag will be the same as the name of the object field it refers to.

Nodes without **type** attribute are transferred identically into the XML destination file (the data file). Nodes with a type attribute will be parsed by the server and their content will be replaced by data coming from objects. In addition to the type attribute, nodes have other possible attributes. These attributes depend on the type of the node (each node type supports or needs different attributes). Most node types have a name attribute, which refers to the **name** of a field of the object on which we work.

As for the “browse” method on objects, field names in reports can use a notation similar to the notation found in object oriented programming languages. It means that “relation fields” can be used as “bridges” to fetch data from other (related) objects.

Let’s use the “account.transfer” object as an example. It contains a partner\_id field. This field is a relation field (“many to one”) pointing to the “res.partner” object. Let’s suppose that we want to create a report for transfers and in this report, we want to use the name of the recipient partner. This name could be accessed using the following expression as the name of the field:

```
partner_id.name
```

## 24.1.2 Possible types

Here is the list of available field types:

- **field**: It is the simplest type. For nodes of this type, the server replaces the node content by the value of the field whose name is given in the name attribute.
- **fields**: when this type of node is used, the server will generate a node in the XML data file for each unique value of the field whose name is given in the name attribute.

Notes:

\*\* This node type is often used with “id” as its name attribute. This has the effect of creating one node for each resource selected in the interface by the user.

\*\* The semantics of a node `<node type="fields" name="field_name">` is similar to an SQL statement of the form “`SELECT FROM object_table WHERE id in identifier_list GROUP BY field_name`” where identifier\_list is the list of ids of the resources selected by the ::user (in the interface).

- **eval**: This node type evaluate the expression given in the *expr* attribute. This expression may be any Python expression and may contain objects fields names.
- **zoom**: This node type allows to “enter” into the resource referenced by the relation field whose name is given in the name attribute. It means that its child nodes will be able to access the fields of that resource without having to prefix them with the field name that makes the link with the other object. In our example above, we could also have accessed the field name of the partner with the following:

```
<partner type="zoom" name="partner_id">  
    <name type="field" name="name"/>  
</partner>
```

In this precise case, there is of course no point in using this notation instead of the standard:

```
<name type="field" name="partner_id.name"/>
```

The **zoom** type is only useful when we want to recover several fields in the same object.

- **function**: returns the result of the call to the function whose name is given in the name attribute. This function must be part of the list of predefined functions. For the moment, the only available function is today, which returns the current date.
- **call**: calls the object method whose name is given in the name attribute with the arguments given in the args attribute. The result is stored into a dictionary of the form {‘name\_of\_variable’: value, ... } and can be accessed through child nodes. These nodes must have a value attribute which correspond to one of the keys of the dictionary returned by the method.

**Example:**

```
<cost type="call" name="compute_seller_costs" args="">

  <name value="name"/>
  <amount value="amount"/>

</cost>
```

**TODO:** documenter format methode appellée def compute\_buyer\_costs(self, cr, uid, ids, \*args):

- **attachment**: extract the first attachment of the resource whose id is taken from the field whose name is given in the name attribute, and put it as an image in the report.

**Example:** <image type="attachment" name="id"/>

**Example**

Here is an example of XML file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<transfer-list>

  <transfer type="fields" name="id">

    <name type="field" name="name"/>
    <partner_id type="field" name="partner_id.name"/>
    <date type="field" name="date"/>
    <type type="field" name="type"/>
    <reference type="field" name="reference"/>
    <amount type="field" name="amount"/>
    <change type="field" name="change"/>

  </transfer>

</transfer-list>
```

## 24.2 Introduction to RML

For more information on the RML format, please refer to the official Reportlab documentation.

- <http://www.reportlab.com/docs/rml2pdf-userguide.pdf>

## 24.3 XSL:RML Stylesheet

There are two possibilities to do a XSL style sheet for a report. Either making everything by yourself, or use our predefined templates

Either freestyle or use corporate\_defaults + rml\_template

```
import rml_template.xsl
```

```
required templates:
```

- frames?
  - stylesheet
  - story
- optional templates:

### 24.3.1 Translations

As OpenERP can be used in several languages, reports must be translatable. But in a report, everything doesn't have to be translated : only the actual text has to be translated, not the formatting codes. A field will be processed by the translation system if the XML tag which surrounds it (whatever it is) has a `t="1"` attribute. The server will translate all the fields with such attributes in the report generation process.

### 24.3.2 Useful links

- [url=http://www.reportlab.com/docs/rml2pdf-userguide.pdf](http://www.reportlab.com/docs/rml2pdf-userguide.pdf) RML UserGuide (pdf) (reportlab.com)
- <http://www.zvon.org/xxl/XSLTutorial/Output/index.html> XSL Tutorial (zvon.org)
- <http://www.zvon.org/xxl/XSLReference/Output/index.html> XSL Reference (zvon.org)
- <http://www.w3schools.com/xsl/> XSL tutorial and references (W3Schools)
- <http://www.w3.org/TR/xslt/> XSL Specification (W3C)

### 24.3.3 Example (with corporate defaults)

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" :xmlns:fo="http:////>

<xsl:import href="../../custom/corporate_defaults.xsl"/>
<xsl:import href="../../base/report/rml_template.xsl"/>
<xsl:variable name="page_format">a4_normal</xsl:variable>
<xsl:template match="/">

    <xsl:call-template name="rml"/>

</xsl:template>
<xsl:template name="stylesheet">

    </xsl:template>

<xsl:template name="story">

    <xsl:apply-templates select="transfer-list"/>

</xsl:template>
<xsl:template match="transfer-list">

    <xsl:apply-templates select="transfer"/>

</xsl:template>
<xsl:template match="transfer">

    <setNextTemplate name="other_pages"/>
    <para>

        Document: <xsl:value-of select="name"/>

    </para><para>
```

```
Type: <xsl:value-of select="type"/>

</para><para>

    Reference: <xsl:value-of select="reference"/>

</para><para>

    Partner ID: <xsl:value-of select="partner_id"/>

</para><para>

    Date: <xsl:value-of select="date"/>

</para><para>

    Amount: <xsl:value-of select="amount"/>

</para>
<xsl:if test="number(change)>0">

    <para>

        Change: <xsl:value-of select="change"/>

    </para>

</xsl:if>
<setNextTemplate name="first_page"/>
<pageBreak/>

</xsl:template>

</xsl:stylesheet>
```



# REPORTS WITHOUT CORPORATE HEADER

## Example (with corporate defaults):

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" :xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <xsl:import href="../../base/report/rml_template.xsl"/>
    <xsl:variable name="page_format">a4_normal</xsl:variable>

    <xsl:template match="/">
        <xsl:call-template name="rml"/>
    </xsl:template>

    <xsl:template name="stylesheet">
    </xsl:template>

    <xsl:template name="story">
        <xsl:apply-templates select="transfer-list"/>
    </xsl:template>

    <xsl:template match="transfer-list">
        <xsl:apply-templates select="transfer"/>
    </xsl:template>

    <xsl:template match="transfer">
        <setNextTemplate name="other_pages"/>

        <para>
            Document: <xsl:value-of select="name"/>
        </para><para>
            Type: <xsl:value-of select="type"/>
        </para><para>
            Reference: <xsl:value-of select="reference"/>
        </para><para>
            Partner ID: <xsl:value-of select="partner_id"/>
        </para><para>
            Date: <xsl:value-of select="date"/>
        </para><para>
            Amount: <xsl:value-of select="amount"/>
        </para>

        <xsl:if test="number(change)>0">
            <para>
                Change: <xsl:value-of select="change"/>
            </para>
        </xsl:if>

        <setNextTemplate name="first_page"/>
    </xsl:template>
</xsl:stylesheet>
```

```
<pageBreak/>
</xsl:template>
</xsl:stylesheet>
```

# EACH REPORT WITH ITS OWN CORPORATE HEADER

**Example (with corporate defaults):**

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" :xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <xsl:import href="../../custom/corporate_defaults.xsl"/>
  <xsl:import href="../../base/report/rml_template.xsl"/>
  <xsl:variable name="page_format">a4_normal</xsl:variable>
  .....
  </xsl:template>

</xsl:stylesheet>
```



# BAR CODES

## 27.1 Barcodes in RML files

Barcodes can be generated using the <barCode> tag in RML files. The following formats are supported:

- codabar
- code11
- code128 (default if no ‘code’ specified’)
- standard39
- standard93
- i2of5
- extended39
- extended93
- msi
- fim
- postnet
- ean13
- ean8
- usps\_4state

You can change the following attributes for rendering your barcode:

- ‘code’: ‘char’
- ‘ratio’:’float’
- ‘xdim’:’unit’
- ‘height’:’unit’
- ‘checksum’:’bool’
- ‘quiet’:’bool’

Examples:

```
<barcode code="code128" xdim="28cm" ratio="2.2">SN12345678</barcode>
```



---

CHAPTER  
**TWENTYEIGHT**

---

# **HOW TO ADD A NEW REPORT**

In 4.0.X

Administration -> Custom -> Low Level -> Base->Actions -> ir.actions.report.xml



# USUAL TAGS

## 29.1 Code within [[ ]] tags is python code

The context of the code (the variable's values you can use) is the same as that described for *Dynamic content in OpenOffice reports*.



# UNICODE REPORTS

As of OpenERP 5.0-rc3 unicode printing with ReportLab is still not available. The problem is that OpenERP uses the PDF standard fonts (14 fonts, they are not embedded in the document but the reader provides them) that are Type1 and have only Latin1 characters.

## 30.1 The solution consists of 3 parts

- Provide TrueType fonts and make them accessible for ReportLab.
- Register the TrueType fonts with ReportLab before using them in the reports.
- Replace the old fontNames in xsl and rml templates with the TrueType ones.

## 30.2 All these ideas are taken from the forums

### Free TrueType fonts

that can be used for this purpose are in the DejaVu family. [http://dejavu-fonts.org/wiki/index.php?title=Main\\_Page](http://dejavu-fonts.org/wiki/index.php?title=Main_Page)  
They can be installed

- in the ReportLab's fonts directory,
- system-wide and include that directory in rl\_config.py,
- in a subdirectory of the OpenERP installation and give that path to ReportLab during the font registration.

### In the server/bin/report/render/rml2pdf/\_init\_\_.py

```
import reportlab.rl_config
reportlab.rl_config.warnOnMissingFontGlyphs = 0

from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.ttfonts import TTFont
import reportlab

enc = 'UTF-8'

#repeat these for all the fonts needed
pdfmetrics.registerFont(TTFont('DejaVuSans', 'DejaVuSans.ttf', enc))
pdfmetrics.registerFont(TTFont('DejaVuSans-Bold', 'DejaVuSans-Bold.ttf', enc))

from reportlab.lib.fonts import addMapping

#repeat these for all the fonts needed
addMapping('DejaVuSans', 0, 0, 'DejaVuSans') #normal
addMapping('DejaVuSans-Bold', 1, 0, 'DejaVuSans') #normal
```

trml2pdf.py should be modified to load this if invoked from the command line.

### All the xsl and rml files have to be modified

A list of possible alternatives:

```
'Times-Roman',      'DejaVuSerif.ttf'  
'Times-BoldItalic', 'DejaVuSerif-BoldItalic.ttf'  
'Times-Bold',       'DejaVuSerif-Bold.ttf'  
'Times-Italic',    'DejaVuSerif-Italic.ttf'  
  
'Helvetica',        'DejaVuSans.ttf'  
'Helvetica-BoldItalic', 'DejaVuSans-BoldOblique.ttf'  
'Helvetica-Bold',   'DejaVuSans-Bold.ttf'  
'Helvetica-Italic', 'DejaVuSans-Oblique.ttf'  
  
'Courier',          'DejaVuSansMono.ttf'  
'Courier-Bold',     'DejaVuSansMono-Bold.ttf'  
'Courier-BoldItalic', 'DejaVuSansMono-BoldOblique.ttf'  
'Courier-Italic',  'DejaVuSansMono-Oblique.ttf'  
  
'Helvetica-ExtraLight', 'DejaVuSans-ExtraLight.ttf'  
  
'TimesCondensed-Roman', 'DejaVuSerifCondensed.ttf'  
'TimesCondensed-BoldItalic', 'DejaVuSerifCondensed-BoldItalic.ttf'  
'TimesCondensed-Bold',   'DejaVuSerifCondensed-Bold.ttf'  
'TimesCondensed-Italic', 'DejaVuSerifCondensed-Italic.ttf'  
  
'HelveticaCondensed', 'DejaVuSansCondensed.ttf'  
'HelveticaCondensed-BoldItalic', 'DejaVuSansCondensed-BoldOblique.ttf'  
'HelveticaCondensed-Bold',   'DejaVuSansCondensed-Bold.ttf'  
'HelveticaCondensed-Italic', 'DejaVuSansCondensed-Oblique.ttf'
```

# HTML REPORTS USING MAKO TEMPLATES

**Note:**

---

*Implemented in trunk only*

*Mako is a template library written in Python. It provides a familiar, non-XML syntax which compiles into Python modules for maximum performance.*

---

## 31.1 Mako Template

### 31.1.1 Syntax

A Mako template is parsed from a text stream containing any kind of content, XML, HTML, email text, etc.

The template can further contain Mako-specific directives which represent variable and/or expression substitutions, control structures (i.e. conditionals and loops), server-side comments, full blocks of Python code, as well as various tags that offer additional functionality. All of these constructs compile into real Python code.

This means that you can leverage the full power of Python in almost every aspect of a Mako template.

### 31.1.2 Expression Substitution

The simplest expression is just a variable substitution. The syntax for this is the \${ } construct instead of [[ ]] in rml.

eg:

```
this is x: ${x}
```

Above, the string representation of x is applied to the template's output stream where x comes

The contents within the \${} tag are evaluated by Python directly.

**Control Structures** In Mako, control structures (i.e. if/else, loops (like while and for), as well as things like try/except) are written using the % marker followed by a regular Python control expression, and are “closed” by using another % marker with the tag “end<name>”, where “<name>” is the keyword of the expression:

eg:

```
% if x==5:
    this is some output
% endif
```

## 31.2 Python Blocks

Within <% %>, you're writing a regular block of Python code. While the code can appear with an arbitrary level of preceding whitespace, it has to be consistently formatted with itself. Mako's compiler will adjust the block of Python to be consistent with the surrounding generated Python code.

**Useful links:** <http://www.makotemplates.org/docs/>

### 31.2.1 An Overview of Sale Order Example

For Complete Example of Sale\_order please Refer the module sale\_report\_html from :

<https://code.launchpad.net/~openerp-community/openobject-addons/trunk-addons-community>

```
## -*- coding: utf-8 -*-
<html>
<head>
    <%include file="mako_header.html"/>
</head>
% for o in objects:
<body>
    <table width="100" border="0" cellspacing="0" cellpadding="0">
        <tr>
            <td>
                <p><small><b>Shipping address :</b></small>
            </td>
        </tr>
        <tr>
            <td>
                <small>${ o.partner_id.title or '' } ${ o.partner_id.name }</small>
            </td>
        </tr>
        <tr>
            <td>
                <small>${ o.partner_shipping_id.state_id and o.partner_shipping_id.name or '' } ${ o.partner_shipping_id.zip or '' }</small>
            </td>
        </tr>
    </table>
    <table>
        <tr align="left">
            <th>Description</th>
            <th>VAT</th>
            <th>Quantity</th>
            <th>Unit Price</th>
            <th>Disc. (%)</th>
            <th>Price</th>
        </tr>
        % for line in o.order_line:
            <tr>
                <td>${line.name}</p>
                <td>${', '.join(map(lambda x: x.name, line.tax_id))}</td>
                <td>${line.product_uos and line.product_uos_qty or line.product_uom_qty}</td>
                <td>${line.product_uos and line.product_uos.name or line.product_uom.name}</td>
```

```

        <td>${line.price_unit}</td>
        <td>${line.discount or 0.00 }</td>
        <td>${line.price_subtotal or 0.00 }</td>
    </tr>
    % if line['notes']:
        <tr>
            <td>${line.notes}</td>
        </tr>

    % endif
    % endfor
</table>
</body>
% endfor
<%include file="mako_header.html"/>
</html>
```

You can format the report as you need using HTML.

### 31.2.2 Report with header and footer

To create reports with your company header you need to include `<%include file="mako_header.html"/>` To create reports with your company footer you need to include `<%include file="mako_footer.html"/>` These files will bring the header and footer that you have defined for your company in the database.



## **Part VII**

# **Server Action**



# **INTRODUCTION**

Server action is a new feature available since the OpenERP version 5.0 beta. This is a useful feature to fulfill customer requirements. It provides a quick and easy configuration for day to day requirements such as sending emails on confirmation of sale orders or invoice, logging operations on invoices (confirm, cancel, etc.), or running wizard/report on confirmation of sales, purchases, or invoices.



# STEP 1: DEFINITION OF SERVER ACTION

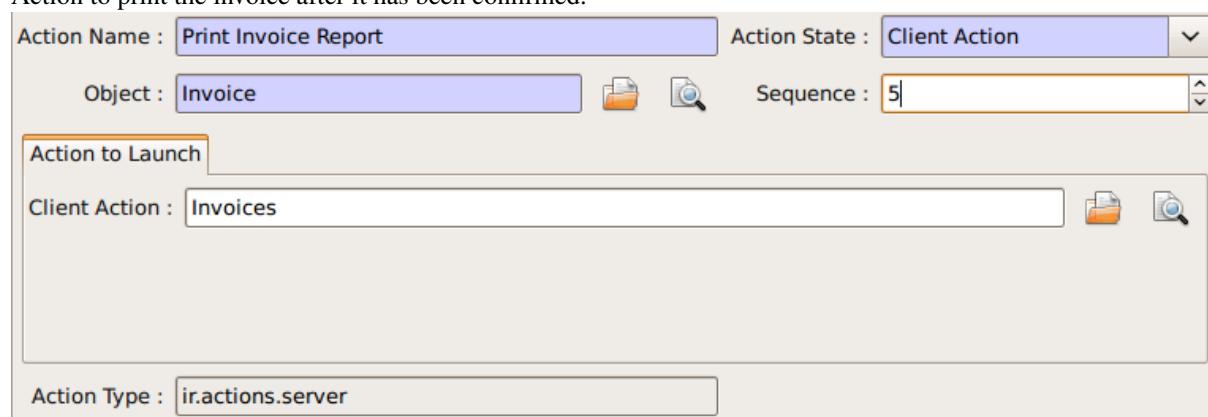
Here is the list of the different action types supplied under the Server Action.

- Client Action
- Dummy
- Iteration
- Python Code
- Trigger
- Email
- SMS
- Create Object
- Write Object
- Multi Action

Each type of action has special features and different configuration parameters. The following sections review each action type and describe how to configure them, together with a list of parameters affecting the system.

## 33.1 Client Action

This action executes on the client side. It can be used to run a wizard or report on the client side. For example, a Client Action can print an invoice after it has been confirmed and run the payment wizard. Technically we can run any client action executed on client side. This includes ir.actions.report.custom, ir.actions.report.xml, ir.actions.act\_window, ir.actions.wizard, and ir.actions.url. In the following example, we can configure a Client Action to print the invoice after it has been confirmed.



The screenshot shows the configuration of a Client Action named "Print Invoice Report". The "Action Name" field contains "Print Invoice Report". The "Action State" dropdown is set to "Client Action". The "Object" field is set to "Invoice", with a browse icon and a search icon. The "Sequence" field is set to "5". The "Action to Launch" section contains a "Client Action" field with the value "Invoices", accompanied by a browse icon and a search icon. At the bottom, the "Action Type" field is set to "ir.actions.server".

Important fields are:

**Object** the object affected by the workflow on for which we want to run the action

**Client Action** the client action, which will be executed on the client side. It must have one of the following types:

- ir.actions.report.custom
- ir.actions.report.xml
- ir.actions.act\_window
- ir.actions.wizard
- ir.actions.url

## 33.2 Iteration

Using a Python loop expression, it is possible to iterate over a server action. For example, when confirming a inward stock move, each line item must be historized. You can loop on expression `object.move_lines` and create another server action which is referred to do the historizing job.

## 33.3 Python Code

This action type is used to execute multiline python code. The returned value is the value of the variable `action`, defaulting to `{ }`. This makes sense only if you want to pop a specific window(form) specific to the context, but a return value is generally not needed.

Note: The code is executed using Python's `exec` built-in function. This function is run in a dedicated namespace containing the following identifiers: `object, time, cr, uid, ids`.

## 33.4 Trigger

Any transition of the workflow can be triggered using this action. The options you need to set are:

**Object** the object affected by the workflow on for which we want to run the action

**Workflow on** The target object on which you want to trigger the workflow.

**Trigger on** the ID of the target model record, e.g. `Invoice` if you want to trigger a change on an invoice.

**Trigger Name** the signal you have to use to initiate the transition. The drop down lists all possible triggers. Note: the list contains all possible transitions from other models also, so ensure you select the right trigger. Models are shown in brackets.

The following example shows the configuration of a trigger used to automatically confirm invoices:

Action Name : <input type="text" value="Confirm Invoice"/>	Action State : <input type="text" value="Trigger"/>
Object : <input type="text" value="Purchase order"/>	<input type="button" value="File"/> <input type="button" value="Search"/>
Sequence : <input type="text" value="5"/>	
<b>Trigger</b>	
<b>Trigger Configuration</b>	
Workflow on : <input type="text" value="Invoice"/>	<input type="button" value="File"/> <input type="button" value="Search"/>
Trigger On : <input type="text" value="Invoice"/>	<input type="button" value="File"/> <input type="button" value="Search"/>
Trigger Name : <input type="text" value="invoice_open - [ account.invoice ]"/>	<input type="button" value="File"/>
Action Type : <input type="text" value="ir.actions.server"/>	

### 33.5 Email Action

This action fulfills a common requirement for all business process, sending a confirmation by email whenever sales order, purchase order, invoice, payment or shipping of goods takes place.

Using this action does not require a dedicated email server: any existing SMTP email server and account can be used, including free email account (Gmail, Yahoo !, etc...)

#### *Server Configuration*

The OpenERP server must know how to connect to the SMTP server. This can be done from the command line when starting the server or by editing the configuration file. Here are the command line options:

```
--email-from=<sender_email@address>
--smtp=<smtp server name or IP address>
--smtp-port=<smtp server port>
--smtp-user=<smtp user name, if required>
--smtp-password=<smtp user password, if required>
--smtp-ssl=<true if the server requires SSL for sending email, else false>
```

Here is an example configuration an action which sends an email when an invoice is confirmed

Action Name : <input type="text" value="Invoice Confirmation Email !!!"/>	Action State : <input type="text" value="Email"/>
Object : <input type="text" value="Invoice"/>	<input type="button" value="File"/> <input type="button" value="Search"/>
Sequence : <input type="text" value="0"/>	
<b>Email Configuration</b>	
<b>Email Configuration</b>	
Contact : <input type="text" value="E-Mail"/>	<input type="button" value="File"/> <input type="button" value="Search"/>
<p>&lt;p&gt;</p> <p>Dear [[ object.partner_id.name ]]&lt;br/&gt;</p> <p>&lt;br/&gt;</p> <p>Message : Your Invoice have been confirmed with the following details.&lt;br/&gt;</p> <p>&lt;br/&gt;</p> <p>Invoice No : [[ object.number ]]&lt;br/&gt;</p> <p>&lt;br/&gt;</p> <p>Access all the fields related to the current object using expression in double brackets, i.e. [[ object.partner_id.name ]]</p>	

Important Fields are:

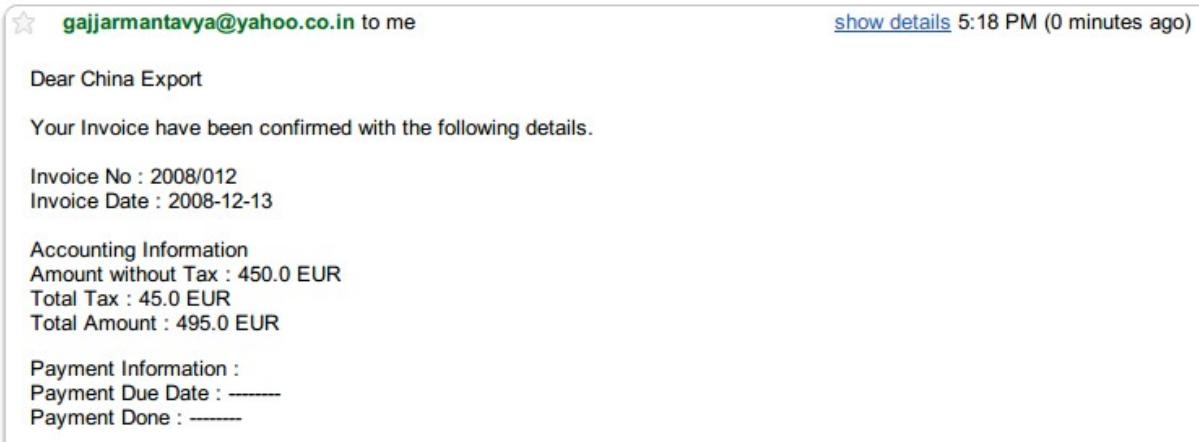
**Object** the object affected by the workflow on for which we want to run the action

**Contact** the field from which action will find the email address of the recipient of the email. The system will displays all the fields related to the object selected in the Object field.

**Message** the message template with the fields that will filled using the current object. The notation is the same as the one used RML to design reports: you can use the [[ ]] + HTML tags to design

in the HTML format. For example to get the partner name we can use [[ object.partner\_id.name ]], object refers to the current object and we can access any fields which exist in the model.

After configuring this action, whenever an invoice is confirmed, an email such as the following is sent:  
**Invoice Confirmation Email !!!**



## 33.6 Create Object

This type of action can be used to emulate the Event history feature currently available on Partners, which logs sales orders issued by a partner, on other objects which do not natively support this feature, such as invoices:

Destination	Type	Value
Description	Formula	"Confirm the Invoice with the Number : " + str(object.number)
Events	Formula	"Invoice " + str(object.number)
Partner	Formula	object.partner_id.id
User	Formula	object.partner_id.user_id.id

Create Object action configuration can be tricky, since it is currently necessary to remember the field names (or to check them out from the source code itself). There are plans to provide an expression builder inside OpenERP in the future, which will be useful to build complex expressions.

Important fields are:

**Object** the object affected by the workflow on for which we want to run the action

**Model** the target model for the object to be created. If empty, it refers to the current object and allows to select the fields from it. It is recommended to provide a model in all cases.

**Fields Mapping** Need to provide 3 values:

1. *Destination*: any of the fields from the target model
2. *Type*: the type of the mapping. Allowed values are `value` or `formula`
3. *Value*: provide the value or expression the expression. The `object` refers to the current object.

*You must select the all required fields from the target model*

**Record Id** the field in which the id of the new record is stored. This is used to refer to the same object in future operations (see below)

## 33.7 Write Object

The configuration is very similar to the Create Object actions. The following example writes ‘Additional Information’ on the same object

Destination	Type	Value
Additional Information	Formula	"Invoice Number : " + str(object.number)

Important Fields are

same as the Create Object

## 33.8 Multi Action

This action allows to execute multiple server actions on the same business operation. For instance, it can be used to print *and* send an email on confirmation of an invoice. This requires creating 3 server actions:

- Print Invoice
- Invoice Confirmation Email !!
- Multi Action

There is a fundamental restriction on this action: it can execute many actions at the server side, but only one single client action. It is therefore not possible to print a report and execute a wizard at the same time.

Action Name	Action State
Print Invoice Report	Client Action
Invoice Confirmation Email !!!	Email

Only one client action will be execute, last client action will be consider in case of multipls clinets actions

Important Fields are:

**Object** the object affected by the workflow on for which we want to run the action

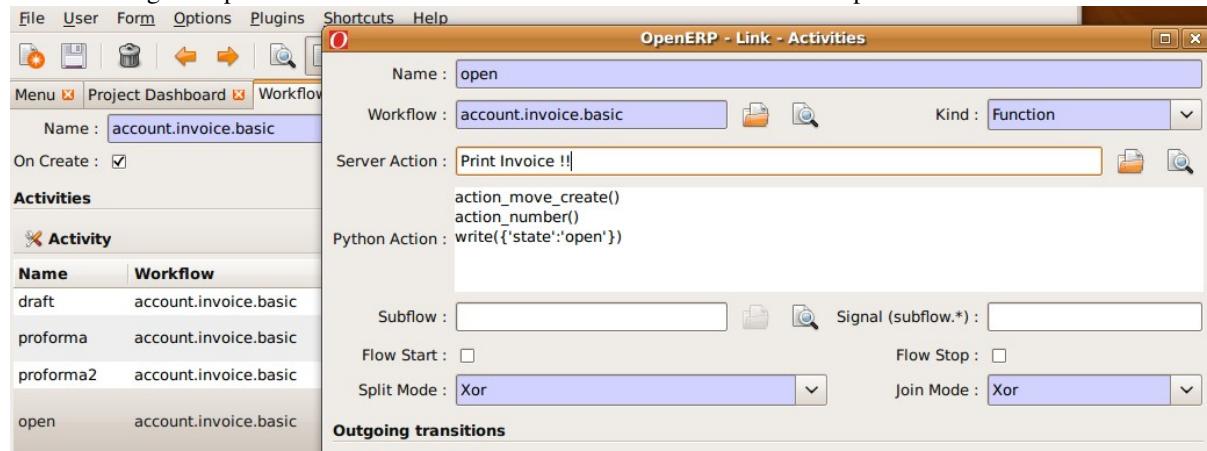
**Other Actions** the list of server action. Any number of actions can be selected, but beware of the restriction mentioned above: if you select more than one Client action, only the first will be executed.

## STEP 2: MAPPING SERVER ACTIONS TO WORKFLOWS

Server actions by themselves are useless, until a workflow stage is set up to trigger them.

Workflows can be accessed at: Administration >> Customization >> Workflow Definitions >> Workflows. Open the corresponding workflow, edit the stage at which the server action needs to be triggered. Then Select the server action in the box.

The following example shows how to associate the Print invoice action to the Open state of the Invoice workflow:





## **Part VIII**

# **Workflow-Business Process**



# INTRODUCTION

The workflow system in OpenERP is a very powerful mechanism that can describe the evolution of documents (model) in time.

Workflows are entirely customizable, they can be adapted to the flows and trade logic of almost any company. The workflow system makes OpenERP very flexible and allows it to easily support changing needs without having to program new functionality.

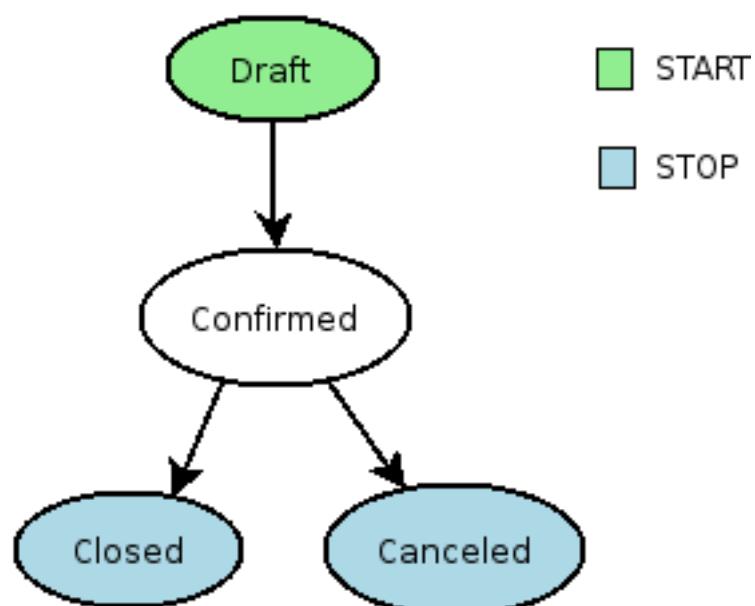
## Goals

- description of document evolution in time
- automatic trigger of actions if some conditions are met
- management of company roles and validation steps
- management of interactions between the different objects/modules
- graphical tool for visualization of document flows

To understand their utility, see the following three:

### 35.1 Example 1: Discount On Orders

The first diagram represent a very basic workflow of an order:



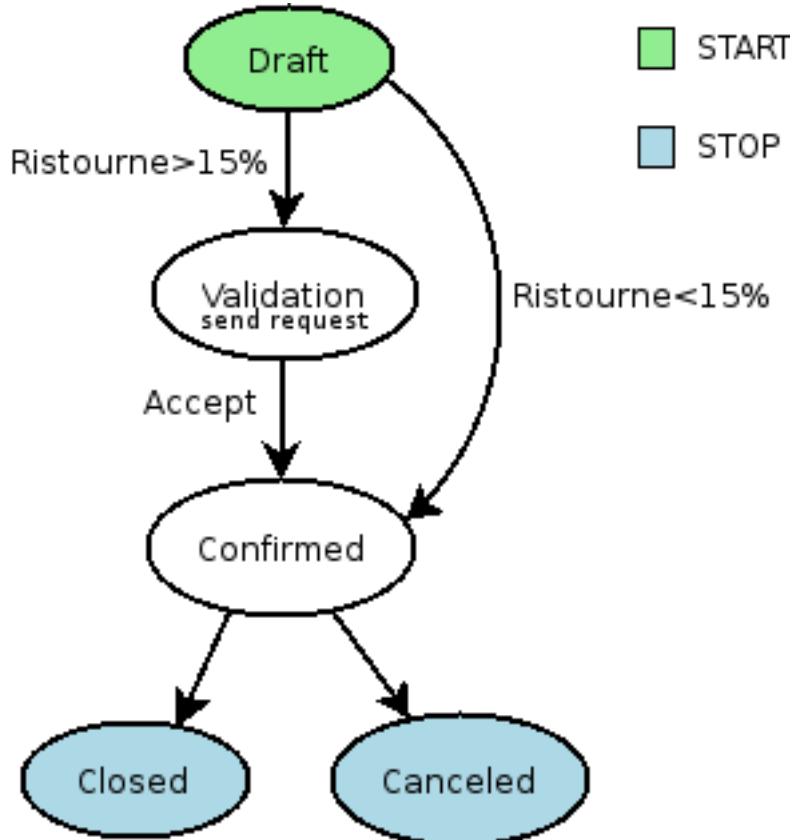
The order starts in the 'draft' state, when it is being written and has not been approved yet. When the user presses on the 'Confirm' button, the invoice is created and the order transitions to the 'CONFIRMED' state.

Then, two operations are possible:

1. the order is done (shipped)
2. the order is canceled

Let's suppose a company has a need not implemented in OpenERP. For example, their sales staff can only offer discounts of 15% or less. Every order having a discount above 15% must be approved by the sales manager.

This modification in the sales logic doesn't need any lines of Python code! A simple modification of the workflow allows us to take this new need into account and add the extra validation step.

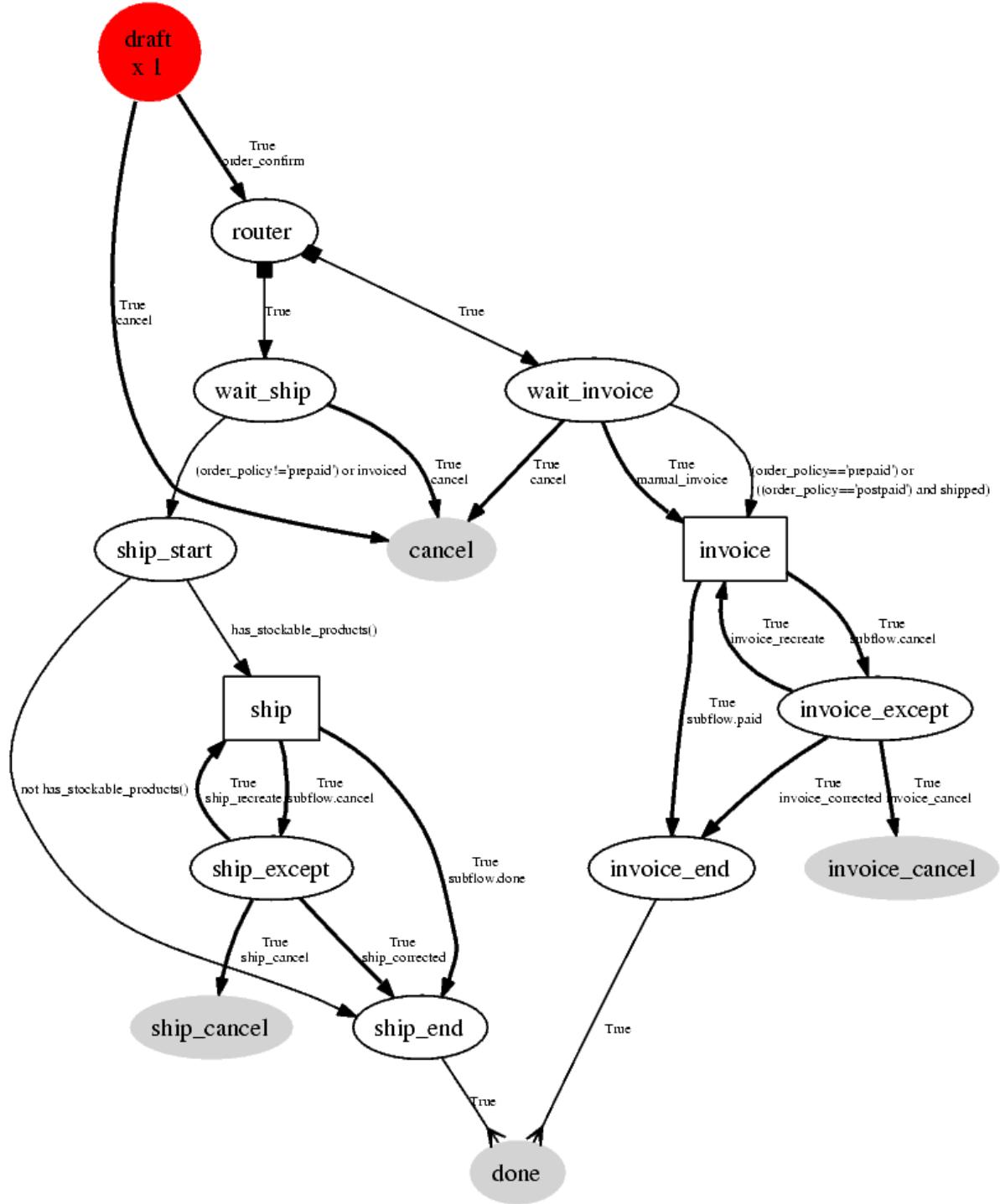


The workflow is modified as above and the orders will react as requested. We then only need to modify the order form view and add a validation button at the desired location.

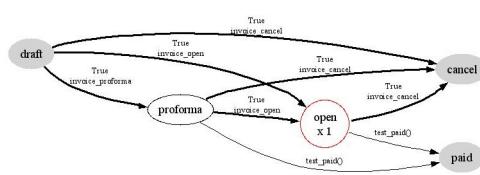
We could then further improve this workflow by sending a request to the sales manager when an order enters the 'Validation' state. Workflow nodes can execute object methods; only two lines of Python are needed to send a request asking the sales manager to validate or reject the order.



## 35.2 Example 2: A sale order that generates an invoice and a shipping order



## 35.3 Example 3: Account invoice basic workflow



# DEFINING WORKFLOW

Workflows are defined in the file `server/addons/base/ir/workflow/workflow.py`. The first three classes defined in this file are `workflow`, `wkf_activity` and `wkf_transition`. They correspond to the three types of resources necessary to describe a workflow:

- `workflow` : the workflow,
- `wkf_activity` : the activities (nodes),
- `wkf_transition` : the transitions between the activities.



# GENERAL STRUCTURE OF A WORKFLOW XML FILE

The general structure of a workflow XML file is as follows:

```
<?xml version="1.0"?>
<openerp>
<data>
<record model="workflow" id=workflow_id>

    <field name="name">workflow.name</field>
    <field name="osv">resource.model</field>
    <field name="on_create" eval='True|False' />

</record>

</data>
</openerp>
```

## Where

- **id** (here “workflow\_id”) is a workflow identifier. Each workflow must have an unique identifier.
- **name** (here “workflow.name”) is the name of the workflow. The name of the workflow must respect the OpenERP syntax of “dotted names”.
- **osv** (here “resource.model”) is the name of the object we use as a model [-(Remember an OpenERP object inherits from osv.osv, hence the ‘<field name=”osv”>’)-].
- **on\_create** is True if workflow.name must be instantiated automatically when resource.model is created, and False otherwise.

## Example

The workflow `sale.order.basic` defined in `addons/sale/sale_workflow.xml` follows exactly this model, the code of its workflow tag is:

```
<record model="workflow" id="wkf_sale">

    <field name="name">sale.order.basic</field>
    <field name="osv">sale.order</field>
    <field name="on_create" eval='True' />

</record>
```



# ACTIVITY

## 38.1 Introduction

The `wkf_activity` class represents the nodes of workflows. These nodes are the actions to be executed.

## 38.2 The fields

`split_mode`



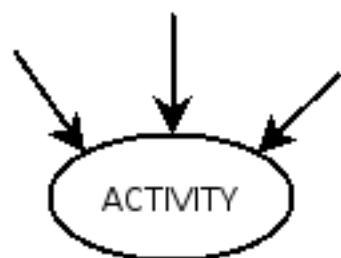
Possible values:

- **XOR:** One necessary transition, takes the first one found (default).
- **OR:** Take only valid transitions (0 or more) in sequential order.
- **AND:** All valid transitions are launched at the same time (fork).

In the OR and AND separation mode, certain workitems can be generated.

In the AND mode, the activity waits for all transitions to be valid, even if some of them are already valid. They are all triggered at the same time.

`join_mode`



Possible values:

- **XOR:** One transition necessary to continue to the destination activity (default).
- **AND:** Waits for all transition conditions to be valid to execute the destination activity.

kind

Possible values:

- **dummy**: Do nothing (default).
- **function**: Execute the function selected by an action.
- **subflow**: Execute a sub-workflow SUBFLOW\_ID. The action method must return the ID of the concerned resource by the subflow. If the action returns False, the workitem disappears.
- **stopall**:

A sub-workflow is executed when an activity is of the type SUBFLOW. This activity ends when the sub-workflow has finished. While the sub-workflow is active, the workitem of this activity is frozen.

action

The action indicates the method to execute when a workitem comes into this activity. The method must be defined in an object which belongs to this workflow and have the following signature:

```
def object_method(self, cr, uid, ids):
```

In the action though, they will be called by a statement like:

```
object_method()  
signal_send
```

This field is used to specify a signal that will be sent to the parent workflow when the activity becomes active. To do this, set the value to the name of the signal (without the `signal.` prefix).

flow\_start

Indicates if the node is a start node. When a new instance of a workflow is created, a workitem is activated for each activity marked as a `flow_start`.

**Warning:**

*As for all Boolean fields, when writing the `<field>` tag in your XML data, be sure to use the `eval` attribute and not a text node for this attribute. Read the section about the [eval attribute](#) for an explanation.*

flow\_stop

Indicates if the node is an ending node. When all the active workitems for a given instance come in the node marked by `flow_stop`, the workflow is finished.

**Warning:**

*See above in the description of the `flow_start` field.*

:: wkf\_id

The workflow this activity belongs to.

### 38.3 Defining activities using XML files

The general structure of an activity record is as follows

```
<record model="workflow.activity" id="activity_id">  
    <field name="wkf_id" ref="workflow_id"/>  
    <field name="name">activity.name</field>:  
  
    <field name="split_mode">XOR | OR | AND</field>  
    <field name="join_mode">XOR | AND</field>
```

```
<field name="kind">dummy | function | subflow | stopall</field>  
  
<field name="action">' (....)'</field>  
<field name="signal_send">' (....)'</field>  
<field name="flow_start" eval='True | False' />  
<field name="flow_stop" eval='True | False' />  
</record>
```

The first two arguments **wkf\_id** and name are mandatory.

## 38.4 Examples

There are too many possibilities of activity definition to choose from using this definition. We recommend you to have a look at the file server/addons/sale/sale\_workflow.xml for several examples of activity definitions.



# TRANSITION

## 39.1 Introduction

Workflow transitions are the conditions which need to be satisfied to move from one activity to the next. They are represented by one-way arrows joining two activities.

The conditions are of different types:

- role that the user must satisfy
- button pressed in the interface
- end of a subflow through a selected activity of subflow

The roles and signals are evaluated before the expression. If a role or a signal is false, the expression will not be evaluated.

Transition tests may not write values in objects.

## 39.2 The fields

`act_from`

Source activity. When this activity is over, the condition is tested to determine if we can start the ACT\_TO activity.

`act_to`

The destination activity.

`condition`

**Expression** to be satisfied if we want the transition done.

`signal`

When the operation of transition comes from a button pressed in the client form, signal tests the name of the pressed button.

If signal is NULL, no button is necessary to validate this transition.

`role_id`

The **role** that a user must have to validate this transition.

## 39.3 Defining Transitions Using XML Files

The general structure of a transition record is as follows

```
<record model="workflow.transition" id="transition_id">

    <field name="act_from" ref="activity_id'_1_'" />
    <field name="act_to" ref="activity_id'_2_'" />

    <field name="signal">(...)</field>
    <field name="role_id" ref="role_id'_1_'" />
    <field name="condition">(...)</field>

    <field name="trigger_model">(...)</field>
    <field name="trigger_expr_id">(...)</field>

</record>
```

Only the fields **act\_from** and **act\_to** are mandatory.

# EXPRESSIONS

Expressions are written as in Python:

- True
- 1==1
- 'hello' in ['hello','bye']

Any field from the resource the workflow refers to can be used in these expressions. For example, if you were creating a workflow for partner addresses, you could use expressions like:

- zip==1400
- phone==mobile



# **USER ROLE**

Roles can be attached to transitions. If a role is given for a transition, that transition can only be executed if the user who triggered it has the required role.

Each user can have one or several roles. Roles are defined in a tree of roles, parent roles having the rights of all their children.

Example:

CEO

- Technical manager
  - Lead developer
    - \* Developers
    - \* Testers
  - Sales manager
    - Commercials
    - ...

Let's suppose we handle our own bug database and that the action of marking a bug as valid needs the Testers role. In the example tree above, marking a bug as valid could be done by all the users having the following roles: Testers, Lead developer, Technical manager, CEO.



# **ERROR HANDLING**

As of this writing, there is no exception handling in workflows.

Workflows being made of several actions executed in batch, they can't trigger exceptions. In order to improve the execution efficiency and to release a maximum of locks, workflows commit at the end of each activity. This approach is reasonable because an activity is only started if the conditions of the transactions are satisfied.

The only problem comes from exceptions due to programming errors; in that case, only transactions belonging to the entirely completed activities are executed. Other transactions are "rolled back".



# CREATING A WORKFLOW

Steps for creating a simple state-changing workflow for a custom module called **mymod**

## 43.1 Define the States of your object

The first step is to define the States your object can be in. We do this by adding a ‘state’ field to our object, in the `_columns` collection

```
_columns = {  
    ...  
    'state': fields.selection([  
        ('new', 'New'),  
        ('assigned', 'Assigned'),  
        ('negotiation', 'Negotiation'),  
        ('won', 'Won'),  
        ('lost', 'Lost')], 'Stage', readonly=True),  
}
```

## 43.2 Define the State-change Handling Methods

Add the following additional methods to your object. These will be called by our workflow buttons.

```
def mymod_new(self, cr, uid, ids):  
    self.write(cr, uid, ids, {'state': 'new'})  
    return True  
  
def mymod_assigned(self, cr, uid, ids):  
    self.write(cr, uid, ids, {'state': 'assigned'})  
    return True  
  
def mymod_negotiation(self, cr, uid, ids):  
    self.write(cr, uid, ids, {'state': 'negotiation'})  
    return True  
  
def mymod_won(self, cr, uid, ids):  
    self.write(cr, uid, ids, {'state': 'won'})  
    return True  
  
def mymod_lost(self, cr, uid, ids):  
    self.write(cr, uid, ids, {'state': 'lost'})  
    return True
```

Obviously you would extend these methods in the future to do something more useful!

### 43.3 Create your Workflow XML file

There are three types of records we need to define in a file called mymod\_workflow.xml

#### 1. Workflow header record (only one of these)

```
<record model="workflow" id="wkf_mymod">
    <field name="name">mymod.wkf</field>
    <field name="osv">mymod.mymod</field>
    <field name="on_create" eval='True' />
</record>
```

#### 2. Workflow Activity records

These define the actions that must be executed when the workflow reaches a particular state

```
<record model="workflow.activity" id="act_new">
    <field name="wkf_id" ref="wkf_mymod" />
    <field name="flow_start" eval='True' />
    <field name="name">new</field>
    <field name="kind">function</field>
    <field name="action">mymod_new()</field>
</record>

<record model="workflow.activity" id="act_assigned">
    <field name="wkf_id" ref="wkf_mymod" />
    <field name="name">assigned</field>
    <field name="kind">function</field>
    <field name="action">mymod_assigned()</field>
</record>

<record model="workflow.activity" id="act_negotiation">
    <field name="wkf_id" ref="wkf_mymod" />
    <field name="name">negotiation</field>
    <field name="kind">function</field>
    <field name="action">mymod_negotiation()</field>
</record>

<record model="workflow.activity" id="act_won">
    <field name="wkf_id" ref="wkf_mymod" />
    <field name="name">won</field>
    <field name="kind">function</field>
    <field name="action">mymod_won()</field>
    <field name="flow_stop" eval='True' />
</record>

<record model="workflow.activity" id="act_lost">
    <field name="wkf_id" ref="wkf_mymod" />
    <field name="name">lost</field>
    <field name="kind">function</field>
    <field name="action">mymod_lost()</field>
    <field name="flow_stop" eval='True' />
</record>
```

#### 3. Workflow Transition records

These define the possible transitions between workflow states

```
<record model="workflow.transition" id="t1">
    <field name="act_from" ref="act_new" />
    <field name="act_to" ref="act_assigned" />
    <field name="signal">mymod_assigned</field>
</record>
```

```

<record model="workflow.transition" id="t2">
    <field name="act_from" ref="act_assigned" />
    <field name="act_to" ref="act_negotiation" />
    <field name="signal">mymod_negotiation</field>
</record>

<record model="workflow.transition" id="t3">
    <field name="act_from" ref="act_negotiation" />
    <field name="act_to" ref="act_won" />
    <field name="signal">mymod_won</field>
</record>

<record model="workflow.transition" id="t4">
    <field name="act_from" ref="act_negotiation" />
    <field name="act_to" ref="act_lost" />
    <field name="signal">mymod_lost</field>
</record>

```

## 43.4 Add mymod\_workflow.xml to \_\_openerp\_\_.py

Edit your module's \_\_openerp\_\_.py and add "mymod\_workflow.xml" to the update\_xml array, so that OpenERP picks it up next time your module is loaded.

## 43.5 Add Workflow Buttons to your View

The final step is to add the required buttons to mymod\_views.xml file.

Add the following at the end of the <form> section of your object's view definition:

```

<separator string="Workflow Actions" colspan="4"/>
<group colspan="4" col="3">
    <button name="mymod_assigned" string="Assigned" states="new" />
    <button name="mymod_negotiation" string="In Negotiation" states="assigned" />
    <button name="mymod_won" string="Won" states="negotiating" />
    <button name="mymod_lost" string="Lost" states="negotiating" />
</group>

```

## 43.6 Testing

Now use the Module Manager to install or update your module. If you have done everything correctly you shouldn't get any errors. You can check if your workflow is installed in the menu *Administration → Customization → Workflow Definitions*.

When you are testing, remember that the workflow will only apply to NEW records that you create.

## 43.7 Troubleshooting

If your buttons do not seem to be doing anything, one of the following two things are likely:

1. The record you are working on does not have a Workflow Instance record associated with it (it was probably created before you defined your workflow)
2. You have not set the osv field correctly in your workflow XML file



## **Part IX**

# **Dashboard**



OpenERP objects can be created from PostgreSQL views. The technique is as follows :

1. Declare your `_columns` dictionary. All fields must have the flag `readonly=True`.
2. Specify the parameter `_auto=False` to the OpenERP object, so no table corresponding to the `_columns` dictionary is created automatically.
3. Add a method `init(self, cr)` that creates a *PostgreSQL* View matching the fields declared in `_columns`.

**Example** The object `report_crm_case_user` follows this model.

```
class report_crm_case_user(osv.osv):  
    _name = "report.crm.case.user"  
    _description = "Cases by user and section"  
    _auto = False  
    _columns = {  
        'name': fields.date('Month', readonly=True),  
        'user_id': fields.many2one('res.users', 'User',  
                                   readonly=True, relate=True),  
        'section_id': fields.many2one('crm.case.section', 'Section',  
                                   readonly=True, relate=True),  
        'amount_revenue': fields.float('Est.Revenue',  
                                       readonly=True),  
        'amount_costs': fields.float('Est.Cost', readonly=True),  
        'amount_revenue_prob': fields.float('Est. Rev*Prob.',  
                                         readonly=True),  
        'nbr': fields.integer('# of Cases', readonly=True),  
        'probability': fields.float('Avg. Probability',  
                                   readonly=True),  
        'state': fields.selection(AVAILABLE_STATES, 'State',  
                                 size=16, readonly=True),  
        'delay_close': fields.integer('Delay to close',  
                                      readonly=True),  
    }  
    _order = 'name desc, user_id, section_id'  
  
    def init(self, cr):  
        cr.execute("""  
CREATE OR REPLACE VIEW report_crm_case_user AS (  
SELECT  
    min(c.id) as id,  
    SUBSTRING(c.create_date for 7)||'-01' as name,  
    c.state,  
    c.user_id,  
    c.section_id,  
    COUNT(*) AS nbr,  
    SUM(planned_revenue) AS amount_revenue,  
    SUM(planned_cost) AS amount_costs,  
    SUM(planned_revenue*probability)::decimal(16,2)  
        AS amount_revenue_prob,  
    avg(probability)::decimal(16,2) AS probability,  
    TO_CHAR(avg(date_closed-c.create_date),  
           'DD"d" HH24:MI:SS') AS delay_close  
FROM  
    crm_case c  
GROUP BY SUBSTRING(c.create_date for 7), c.state,  
         c.user_id, c.section_id  
)""")  
report_crm_case_user()
```



## **Part X**

# **I18n - Internationalization**



Explain about the multiple language application



# INTRODUCTION

The I18n contains the translation of module in different languages. The folder contains two type of files .po and .pot. The .po files are the actual translation files where as .pot is the template for the translation.

The .po files should be named according to the language code of specific language and .pot should be named according to the module name.

Example of a account.po file

```
"Project-Id-Version: OpenERP Server 5.0.0\n"
"Report-Msgid-Bugs-To: support@openerp.com\n"
"POT-Creation-Date: 2009-05-19 14:36+0000\n"
"PO-Revision-Date: 2009-05-20 10:36+0000\n"
"Last-Translator: <>\n"
"Language-Team: \n"
" MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"X-Launchpad-Export-Date: 2009-12-07 15:06+0000\n"
"X-Generator: Launchpad (build Unknown)\n"

#. module: account
#: code:addons/account/account.py:0
#, python-format
msgid "Integrity Error !"
msgstr ""
```

The above file is the template for the translation files for the account module they tell what all values of the module should be translated in the respective languages.

Example of fr\_FR.po for account module translates in French language

```
# Translation of OpenERP Server.
# This file contains the translation of the following modules:
#      * account
#
msgid ""
msgstr ""
"Project-Id-Version: OpenERP Server 5.0.0\n"
"Report-Msgid-Bugs-To: support@openerp.com\n"
"POT-Creation-Date: 2009-05-19 14:36+0000\n"
"PO-Revision-Date: 2010-01-04 11:19+0530\n"
"Last-Translator: Anup <ach@tinyerp.co.in>\n"
"Language-Team: \n"
" MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"X-Launchpad-Export-Date: 2009-12-07 15:07+0000\n"
"X-Generator: Launchpad (build Unknown)\n"
```

```
#. module: account
#: code:addons/account/account.py:0
#, python-format
msgid "Integrity Error !"
msgstr "Erreur d'Intégrité !"
```

The above file translates the values of msgid to French in msgstr. Based on the msgid present in the .pot file. fr\_FR is the language code for French language in which fr specifies the language and FR specifies the country in which the language is spoken.

## **Part XI**

# **Testing**



# UNIT TESTING

Since version 4.2 of OpenERP, the XML api provides several features to test your modules. They allow you to

- test the properties of your records, your class invariants etc.
- test your methods
- manipulate your objects to check your workflows and specific methods

This thus allows you to simulate user interaction and automatically test your modules.

## 45.1 Generalities

As you will see in the next pages, unit testing through OpenERP's XML can be done using three main tags: <assert>, <workflow> and <function>. All these tags share some common optional attributes:

**uid** allows you to do the tag interpretation through a specific User ID (you must specify the XML id of that user, for example “base.user\_demo”)

**context** allows you to specify a context dictionary (given as a Python expression) to use when applicable (for <function> notice that not all objects methods take a context attribute so it won't be automatically transmitted to them, however it applies on <value>)

These two attributes might be set on any of those tags (for <functions>, only the root <function> tag may accept it) or on the <data> tag itself. If you set a context attribute on both, they will be merged automatically.

Notice that Unit Testing tags will not be interpreted inside a <data> tag set in nouptdate.

## 45.2 Using unit tests

You can declare unit tests in all your .XML files. We suggest you to name the files like this:

- module\_name\_test.xml

If your tests are declared as demo data in the \_\_openerp\_\_.py, they will be checked at the installation of the system with demo data. Example of usage, testing the demo sale order produce a correct amount in the generated invoice.

If your tests are declared like init data, they will be checked at all installation of the software. Use it to test the consistency of the software after installation.

If your tests are declared in update sections, the tests are checked at the installation and also at all updates. Use it to tests consistencies, invariants of the module. Example: The sum of the credits must be equal to the sum of the debits for all non draft entries in the accounting module. Putting tests in update sections is very useful to check consistencies of migrations or new version upgrades.

## 45.3 Assert Tag

The assert tag allows you to define some assertions that have to be checked at boot time. Example :

```
<assert model="res.company" id="main_company" string="The main company name is Open sprl">
    <test expr="name">Open sprl</test>
</assert>
```

This assert will check that the company with id main\_company has a name equal to “Open sprl”. The expr field specifies a python expression to evaluate. The expression can access any field of the specified model and any python built-in function (such as sum, reduce etc.). The ref function, which gives the database id corresponding to a specified XML id, is also available (in the case that “ref” is also the name of an attribute of the specified model, you can use \_ref instead). The resulting value is then compared with the text contained in the test tag. If the assertion fails, it is logged as a message containing the value of the string attribute and the test tag that failed.

For more complex tests it is not always sufficient to compare a result to a string. To do that you may instead omit the tag’s content and just put an expression that must evaluate to True:

```
<assert model="res.company"
        id="main_company"
        string="The main company's currency is €" severity="warning">
    <test expr="currency_id.code == 'eur'.upper()" />
</assert>
```

The severity attribute defines the level of the assertion: debug, info, warning, error or critical. The default is error. If an assertion of too high severity fails, an exception is thrown and the parsing stops. If that happens during server initialization, the server will stop. Else the exception will be transmitted to the client. The level at which a failure will throw an exception is by default at warning, but can be specified at server launch through the --assert-exit-level argument.

As sometimes you do not know the id when you’re writing the test, you can use a search instead. So we can define another example, which will be always true:

```
<assert model="res.partner"
        search="[(‘name’, ‘=’, ‘Agrolait’)]"
        string="The name of Agrolait is :Agrolait">
    <test expr="name">Agrolait</test>
</assert>
```

When you use the search, each resulting record is tested but the assertion is counted only once. Thus if an assertion fails, the remaining records won’t be tested. In addition, if the search finds no record, nothing will be tested so the assertion will be considered successful. If you want to make sure that there are a certain number of results, you might use the count parameter:

```
<assert model="res.partner"
        search="[(‘name’, ‘=’, ‘Agrolait’)]"
        string="The name of Agrolait is :Agrolait"
        count="1">
    <test expr="name">Agrolait</test>
</assert>
```

### Example

Require the version of a module.

```
<!-- modules requirement -->
<assert model="ir.module.module"
        search="[(‘name’, ‘=’, ‘common’)]"
        severity="critical" count="1">
    <test expr="state == ‘installed’" />
    <!-- only check module version -->
    <test expr="‘.‘.join(installed_version.split(‘.’)[3:]) >= ‘2.4’" />
</assert>
```

## 45.4 Workflow Tag

The workflow tag allows you to call for a transition in a workflow by sending a signal to it. It is generally used to simulate an interaction with a user (clicking on a button...) for test purposes:

```
<workflow model="sale.order" ref="test_order_1" action="order_confirm" />
```

This is the syntax to send the signal `order_confirm` to the sale order with id `test_order_1`.

Notice that workflow tags (as all other tags) are interpreted as root which might be a problem if the signals handling needs to use some particular property of the user (typically the user's company, while root does not belong to one). In that case you might specify a user to switch to before handling the signal, through the `uid` property:

```
<workflow model="sale.order" ref="test_order_1" action="manual_invoice" uid="base.user_admin" />
```

(here we had to specify the module `base` - from which `user_admin` comes - because this tag is supposed to be placed in an xml file of the `sale` module)

In some particular cases, when you write the test, you don't know the id of the object to manipulate through the workflow. It is thus allowed to replace the `ref` attribute with a `value` child tag:

```
<workflow model="account.invoice" action="invoice_open">
  <value model="sale.order" eval="obj(ref('test_order_1')).invoice_ids[0].id" />
</workflow>
```

(notice that the `eval` part must evaluate to a valid database id)

## 45.5 Function Tag

The function tag allows to call some method of an object. The called method must have the following signature:

```
def mymethod(self, cr, uid [, ...])
```

Where

- `cr` is the database cursor
- `uid` is the user id

Most of the methods defined in Tiny respect that signature as `cr` and `uid` are required for a lot of operations, including database access.

The function tag can then be used to call that method:

```
<function model="mypackage.myclass" name="mymethod" />
```

Most of the time you will want to call your method with additional arguments. Suppose the method has the following signature:

```
def mymethod(self, cr, uid, mynumber)
```

There are two ways to call that method:

- either by using the `eval` attribute, which must be a python expression evaluating to the list of additional arguments:

```
<function model="mypackage.myclass" name="mymethod" eval="[42]" />
```

In that case you have access to all native python functions, to a function `ref()` that takes as its argument an XML id and returns the corresponding database id, and to a function `obj()` that takes a database id and returns an object with all fields loaded as well as related records.

- or by putting a child node inside the function tag:

```
<function model="mypackage.myclass" name="mymethod">
    <value eval="42" />
</function>
```

Only value and function tags have meaning as function child nodes (using other tags will give unspecified results). This means that you can use the returned result of a method call as an argument of another call. You can put as many child nodes as you want, each one being an argument of the method call (keeping them in order). You can also mix child nodes and the eval attribute. In that case the attribute will be evaluated first and child nodes will be appended to the resulting list.

# ACCEPTANCE TESTING

This document describes all tests that are made each time someone install OpenERP on a computer. You can then assume that all these tests are valid as we must launch them before publishing a new module or a release of OpenERP.

## 46.1 Integrity tests on migrations

- Sum credit = Sum debit
- Balanced account chart

... Describe all integrity tests here

## 46.2 Workflow tests

... Describe all processes tested here.

## 46.3 Record creation

More than 300 records are created, describe them here.



## **Part XII**

# **Serialization, Migration and Upgrading**



# DATA SERIALIZATION

During OpenERP installation, two steps are necessary to create and feed the database:

1. Create the SQL tables
2. Insert the different data into the tables

The creation (or modification in the case of an upgrade) of SQL tables is automated thanks to the description of objects in the server.

With OpenERP, everything except the business logic of objects is stored in the database. We find for example:

- the definitions of the reports,
- the default values for fields,
- the definition of client interfaces for each document (views),
- the relationships between menus, buttons and actions
- etc.

There must be a mechanism to describe, modify and reload these different kinds of data. OpenERP data may be specified in CSV, XML or YAML serialization files provided by modules, and loaded during module installation/upgrade in order to fill or update the database tables.

## 47.1 XML Data Serialization

Since version 4.2, OpenERP provides an XML-based data serialization format.

The basic format of an OpenERP XML file is as follows:

```
<?xml version="1.0"?>
<openerp>
    <data>
        <record model="model.name_1" id="id_name_1">
            <field name="field1">
                field1 content
            </field>
            <field name="field2">
                field2 content
            </field>
            ...
        </record>
        <record model="model.name_2" id="id_name_2">
            ...
        </record>
        ...
    </data>
</openerp>
```

Fields contents are strings that must be encoded as UTF-8 in XML files.

Let's review an example taken from the OpenERP source (base\_demo.xml in the base module):

```
<record model="res.company" id="main_company">
    <field name="name">OpenERP SA</field>
    <field name="partner_id" ref="main_partner"/>
    <field name="currency_id" ref="EUR"/>
</record>

<record model="res.users" id="user_admin">
    <field name="login">admin</field>
    <field name="password">admin</field>
    <field name="name">Administrator</field>
    <field name="signature">Administrator</field>
    <field name="action_id" ref="action_menu_admin"/>
    <field name="menu_id" ref="action_menu_admin"/>
    <field name="address_id" ref="main_address"/>
    <field name="groups_id" eval="[(6,0,[group_admin])]"/>
    <field name="company_id" ref="main_company"/>
</record>
```

This last record defines the admin user :

- The fields login, password, etc are straightforward.
- The **ref** attribute allows to fill relations between the records :

```
<field name="company_id" ref="main_company"/>
```

The “company\_id” field is a many-to-one relation from the user object to the company object, and **main\_company** is the id of to associate.

- The **eval** attribute allows to put some python code in the xml: here the groups\_id field is a many2many. For such a field, “[6,0,[group\_admin]]” means : Remove all the groups associated with the current user and use the list [group\_admin] as the new associated groups (and group\_admin is the id of another record).
- The **search** attribute allows to find the record to associate when you do not know its xml id. You can thus specify a search criteria to find the wanted record. The criteria is a list of tuples of the same form than for the predefined search method. If there are several results, an arbitrary one will be chosen (the first one):

```
<field name="partner_id" search="[]" model="res.partner"/>
```

This is a classical example of the use of **search** in demo data: here we do not really care about which partner we want to use for the test, so we give an empty list. Notice the **model** attribute is currently mandatory.

Some typical XML elements are described below.

### 47.1.1 Record Tag

The addition of new data is made with the **record** tag. This one takes a mandatory attribute : **model**. Model is the object name where the insertion has to be done. The tag record can also take an optional attribute: **id**. If this attribute is given, a variable of this name can be used later on, in the same file, to make reference to the newly created resource ID.

A **record** tag may contain field tags. They indicate the record's **fields** value. If a field is not specified the default value will be used.

## Example

```
<record model="ir.actions.report.xml" id="10">
    <field name="model">account.invoice</field>
    <field name="name">Invoices List</field>
    <field name="report_name">account.invoice.list</field>
    <field name="report_xsl">account/report/invoice.xsl</field>
    <field name="report_xml">account/report/invoice.xml</field>
</record>
```

### 47.1.2 field tag

The attributes for the field tag are the following:

- **name** o mandatory attribute indicating the field name
- **eval** o python expression that indicating the value to add
- **ref** o reference to an id defined in this file

### 47.1.3 function tag

- **model:**
- **name:**
- **eval** o should evaluate to the list of parameters of the method to be called, excluding cr and uid

## Example

```
<function
    model="ir.ui.menu"
    name="search"
    eval="[[('name', '=', 'Operations')]]"/>
```

### 47.1.4getitem tag

Takes a subset of the evaluation of the last child node of the tag.

- **type**
  - int or list
- index
- int or string (a key of a dictionary)

## Example

Evaluates to the first element of the list of ids returned by the function node:

```
<getitem index="0" type="list">
    <function
        model="ir.ui.menu"
        name="search"
        eval="[[('name', '=', 'Operations')]]"/>
</getitem>
```

## 47.2 YAML Data Serialization

YAML is a **human-readable** data serialization format that takes concepts from programming languages such as C, Perl, and **Python**, and ideas from **XML** and the data format of electronic mail. YAML stands for *YAML Ain't Markup Language* (yes, that's a recursive acronym). YAML is available as a format for OpenERP data **as of OpenERP 6.0**, featuring the following advantages:

- User friendly format as an alternative to our current XML data format.
- Same system to load data or tests, integrated in modules.
- Built in OpenERP so that you can develop complex Python tests.
- Simpler for non developers to write functional tests.

The following section compares an XML record with an equivalent YAML record.

First the XML Record using the current XML serialization format (see [previous section](#))

```
<!--
    Resource: sale.order
-->

<record id="order" model="sale.order">
    <field name="shop_id" ref="shop"/>
    <field model="product.pricelist" name="pricelist_id" search="[]"/>
    <field name="user_id" ref="base.user_root"/>
    <field model="res.partner" name="partner_id" search="[]"/>
    <field model="res.partner.address" name="partner_invoice_id" search="[]"/>
    <field model="res.partner.address" name="partner_shipping_id" search="[]"/>
    <field model="res.partner.address" name="partner_order_id" search="[]"/>
</record>

<!--
    Resource: sale.order.line
-->

<record id="line" model="sale.order.line">
    <field name="order_id" ref="order"/>
    <field name="name">New server config + material</field>
    <field name="price_unit">123</field>
</record>

<record id="line1" model="sale.order.line">
    <field name="order_id" ref="order"/>
    <field name="name">[PC1] Basic PC</field>
    <field name="price_unit">450</field>
</record>
```

### 47.2.1 YAML Record

```
#<!--
#      Resource: sale.order
#      -->

-
!record {model: sale.order, id: sale_order_so4}:
    amount_total: 3263.0
    amount_untaxed: 3263.0
    create_date: '2010-04-06 10:45:14'
    date_order: '2010-04-06'
```

```

invoice_quantity: order
name: SO001
order_line:
  - company_id: base.main_company
    name: New server config + material
    order_id: sale_order_so4
    price_unit: 123.0
  - company_id: base.main_company
    name: '[PC1] Basic PC'
    order_id: sale_order_so4
    price_unit: 450.0
order_policy: manual
partner_id: base.res_partner_agrolait
partner_invoice_id: base.main_address
partner_order_id: base.main_address
partner_shipping_id: base.main_address
picking_policy: direct
pricelist_id: product.list0
shop_id: sale.shop

```

## 47.2.2 YAML Tags

### data

- **Tag:** data
- **Compulsory attributes:** None
- **Optional attributes:** nouupdate : 0 | 1
- **Child\_tags:**

- menuitem
- record
- workflow
- delete
- act\_window
- assert
- report
- function
- ir\_set

- **Example:**

```

  !context
  nouupdate: 0

```

### record

- **Tag:** record
- **Compulsory attributes:**
  - model
- **Optional attributes:** nouupdate : 0 | 1

- **Child\_tags:**
  - field

- **Optional attributes:**
  - id
  - forcreate
  - context

- **Example:**
  - ```
!record {model: sale.order, id: order}:
    name: "[PC1] Basic PC"
    amount_total: 3263.0
    type_ids:
        - project_tt_specification
        - project_tt_development
        - project_tt_testing
    order_line:
        - name: New server config
          order_id: sale_order_so4
        - name: '[PC1] Basic PC'
          order_id: sale_order_so4
```

## field

- **Tag:** field

- **Compulsory attributes:**
  - name

- **Optional attributes:**
  - type
  - ref
  - eval
  - domain
  - search
  - model
  - use

- **Child\_tags:**
  - text node

- **Example:**
  - ```
-price_unit: 450
-product_id: product.product_product_pc1
```

## workflow

- **Tag:** workflow

- **Compulsory attributes:**
  - model

- action

- **Optional attributes:**

- uid
- ref

- **Child\_tags:**

- value

- **Example:**

```
- !workflow {action: invoice_open, model: account.invoice}:
  - eval: "obj(ref('test_order_1')).invoice_ids[0].id"
    model: sale.order
  - model: account.account
    search: [('type', '=', 'cash')]
```

## function

- **Tag:** function

- **Compulsory attributes:**

- model
- name

- **Optional attributes:**

- id
- eval

- **Child\_tags:**

- value
- function

- **Example:**

```
- !function {model: account.invoice, name: pay_and_reconcile}:
  -eval: "[obj(ref('test_order_1')).id]"
    model: sale.order
```

## value

- **Tag:** value

- **Compulsory attributes:** None

- **Optional attributes:**

- model
- search
- eval

- **Child\_tags:** None

- **Example:**

```
-eval: "[obj(ref('test_order_1')).id]"
model: sale.order
```

## menuitem

- **Tag:** menuitem
- **Compulsory attributes:** None
- **Optional attributes:**
  - id
  - name
  - parent
  - icon
  - action
  - string
  - sequence
  - groups
  - type
  - menu
- **Child\_tags:** None
- **Example:**

```
!
!menuitem {sequence: 20, id: menu_administration,
name: Administration,
icon: terp-administration}
```

## act\_window

- **Tag:** act\_window

- **Compulsory attributes:**

- id
- name
- res\_model

- **Optional attributes:**

- domain
- src\_model
- context
- view
- view\_id
- view\_type
- view\_mode
- multi

- target
- key2
- groups
- **Child\_tags:** None
- **Example:**
  -

```
!act_window {target: new,
res_model: wizard.ir.model.menu.create,
id:act_menu_create, name: Create Menu}
```

## report

- **Tag:** report
- **Compulsory attributes:**
  - string
  - model
  - name
- **Optional attributes:**
  - id
  - report
  - multi
  - menu
  - keyword
  - rml
  - sxw
  - xml
  - xsl
  - auto
  - header
  - attachment
  - attachment\_use
  - groups
- **Child\_tags:** None
- **Example:**
  -

```
!report {string: Technical guide,
auto: False, model: ir.module.module,
id: ir_module_reference_print,
rml: base/module/report/ir_module_reference.rml,
name: ir.module.reference}
```

## ir\_set

- **Tag:** ir\_set
- **Compulsory attributes:** None
- **Optional attributes:** None
- **Child\_tags:**
  - field
- **Example:**

```
–
!ir_set:
args: "[]"
name: account.seller.costs
value: tax_seller
```

## python

- **Tag:** Python
- **Compulsory attributes:**
  - model
- **Optional attributes:** None
- **Child\_tags:** None
- **Example:**

```
Python code
```

## delete

- **Tag:** delete
  - **Compulsory attributes:**
    - model
  - **Optional attributes:**
    - id
    - search
  - **Child\_tags:** None
  - **Example:**
- ```
–
!delete {model: ir.actions, search: "[ (model,like,auction.) ]"}
```

## assert

- **Tag:** assert
- **Compulsory attributes:**
  - model
- **Optional attributes:**

- id
- search
- string
- **Child\_tags:**
  - test
- **Example:**
  -

```
!assert {model: sale.order,
         id: test_order, string: order in progress}:
    - state == "progress"
```

## test

- **Tag:** test
- **Compulsory attributes:**
  - expr
- **Optional attributes:** None
- **Child\_tags:**
  - text node
- **Example:**
  - picking\_ids[0].state == "done"

## url

- **Tag:** url
- **Compulsory attributes:** -
- **Optional attributes:** -
- **Child\_tags:** -
- **Example:** -

## 47.3 Writing YAML Tests

### Note:

*Please see also section yaml-testing-guidelines*

### Write manually

- Record CRUD
- Workflow transition
- Assertions (one expression like in XML)
- Pure Python code

### Use base\_module\_record(er)

- Generate YAML file with record and workflow



- Update this YAML with assertions / Python code

**Warning:**

*Important*

As yaml is structured with indentation(like Python), each child tag(sub-tag) must be indented as compared to its parent tag.

### 47.3.1 Field Tag

- text
  - text with special characters at beginning or at end must be enclosed with double quotes.  
Ex: name: “[PC1] Basic PC”
- integer and float Ex: price\_unit: 450 Ex: amount\_total: 3263.0
- boolean active: 1
- datetime date\_start: str(time.localtime()[0] - 1) + -08-07
- selection
  - give the shortcut Ex: title: M.
- many2one
  - if its a reference to res\_id, specify the res\_id Ex: user\_id: base.user\_root
  - if its value is based on search criteria specify the model to search on and the criteria Ex:  
object\_id: !ref {model: ir.model, search: “[('model','=','crm.claim')]”}

- one2many
  - start each record in one2many field on a new line with a space and a hyphen Ex: order\_line: name: New server config order\_id: sale\_order\_so4 .....
  - name: '[PC1] Basic PC' order\_id: sale\_order\_so4 .....
- many2many
  - start each record in many2many field with a space and a hyphen Ex: type\_ids: project\_tt\_specification \*\*\*- project\_tt\_development - project\_tt\_testing

### 47.3.2 Value tag

- if the value can be evaluated(like res\_id is available), we write value tag as follows:
 

```
- !function {model: account.invoice, name: pay_and_reconcile}: - eval:  
“obj(ref('test_order_1')).amount_total” model: sale.order
```

This will fetch the ‘amount\_total’ value of a ‘sale.order’ record with res\_id ‘test\_order\_1’
- If the value is to be searched on some model based on a criteria, we write value tag as follows:
 

```
!function {model: account.invoice, name: pay_and_reconcile}: - model: account.account  
search: “[('type', '=', 'cash')]”
```

This will fetch all those account.account records whose type is equal to ‘cash’

### 47.3.3 Test Tag

- specify the test directly Ex: - picking\_ids[0].state == “done” - state == “manual”

### 47.3.4 comment

```
#<!-- Resource: sale.order -->
```

### 47.3.5 Asserts and Python code

To create an invoice, python code could be written as:

```
-  
!python {model: account.invoice}: | self.action_move_create(cr, uid, [ref("invoice1")])
```

The invoice must be in draft state:

```
-  
!assert {model: account.invoice , id: invoice1, string: “the invoice is now in Draft state”}: - state ==  
“draft”
```

To test that all account are in a tree data structure, we write the below python code:

```
-  
!python {model: account.account}: ids = self.search(cr, uid, [])  
accounts_list = self.read(cr, uid, ids['parent_id','parent_left','parent_right'])  
accounts = dict((x['id'], x) for x in accounts_list)  
log("Testing parent structure for %d accounts", len(accounts_list))  
for a in accounts_list:
```

```

if a['parent_id']: assert a['parent_left']>accounts[a['parent_id'][0]]['parent_left']
    assert a['parent_right']<accounts[a['parent_id'][0]]['parent_right']
    assert a['parent_left']<a['parent_right']

for a2 in accounts_list:
    assert not ((a2['parent_right']>a['parent_left'])and (a2['parent_left']<a['parent_left'])and
(a2['parent_right']<a['parent_right']))
    if a2['parent_id']==a['id']: assert(a2['parent_left']>a['parent_left'])and(a2['parent_right']<a['parent_right'])

```

### 47.3.6 Running tests

- Save the file with ‘.yml’ extension
- Add the yaml file under ‘demo\_xml’ in terp file
- Run the server with ‘–log-level=test’ option

## 47.4 CSV Data Serialization

Since version 4.2, OpenERP provides a Comma-Separated-Values (CSV), spreadsheet-compatible data serialization format.

The basic format of an OpenERP CSV file is as follows:

```
"id","name","model_id:id","group_id:id","perm_read","perm_write","perm_create","perm_unlink"
"access_product_uom_categ_manager","product.uom.categ manager","model_product_uom_categ","product.gr
"access_product_uom_manager","product.uom manager","model_product_uom","product.group_product_manage
"access_product_ul_manager","product.ul manager","model_product_ul","product.group_product_manager",
"access_product_category_manager","product.category manager","model_product_category","product.group
"access_product_template_manager","product.template manager","model_product_template","product.group
"access_product_product_manager","product.product manager","model_product_product","product.group_pr
"access_product_packaging_manager","product.packaging manager","model_product_packaging","product.gr
"access_product_uom_categ_user","product.uom.categ.user","model_product_uom_categ","base.group_user"
"access_product_uom_user","product.uom.user","model_product_uom","base.group_user",1,0,0,0
"access_product_ul_user","product.ul.user","model_product_ul","base.group_user",1,0,0,0
"access_product_category_user","product.category.user","model_product_category","base.group_user",1,
"access_product_template_user","product.template.user","model_product_template","base.group_user",1,
"access_product_product_user","product.product.user","model_product_product","base.group_user",1,0,0
"access_product_packaging_user","product.packaging.user","model_product_packaging","base.group_user"
```

### 47.4.1 Importing from a CSV

Instead of using .XML file, you can import .CSV files. It is simpler but the migration system does not migrate the data imported from the .CSV files. It’s like the noudate attribute in .XML files. It’s also more difficult to keep track of relations between resources and it is slower at the installation of the server.

Use this only for [demo] data that will never been upgraded from one version of OpenERP to another.

The name of the object is the name of the CSV file before the first ‘-’. You must use one file per object to import. For example, to import a file with partners (including their multiple contacts and events), the file must be named like one of the following example:

- res.partner.csv
- res.partner-tiny\_demo.csv
- res.partner-tiny.demo.csv

## 47.4.2 Structure of the CSV file

- Separator to use: ,
- Quote character for strings: " (optional if no separator is found in field values)
- Encoding to use: UTF-8
- No whitespace allowed around separators if not using quote characters
- Be sure to configure your CSV export software (e.g. spreadsheet editor) with the above parameters

## 47.4.3 Exporting demo data and import it from a module

You can import .CSV file that have been exported from the OpenERP client. This is interesting to create your own demo module. But both formats are not exactly the same, mainly due to the conversion: Structured Data -> Flat Data -> Structured Data.

- The name of the column (first line of the .CSV file) use the end user term in his own language when you export from the client. If you want to import from a module, you must convert the first column using the fields names. Example, from the partner form:

```
Name,Code,Contacts/Contact Name,Contacts/Street,Contacts/Zip
```

becomes:

```
name,ref,address/name,address/street,address/zip
```

- When you export from the OpenERP client, you can select any many2one fields and their child's relation. When you import from a module, OpenERP tries to recreate the relations between the two resources. For example, do not export something like this from a sale order form - otherwise OpenERP will not be able to import your file:

```
Order Description,Partner/Name,Partner/Payable,Partner/Address/Name
```

- To find the link for a many2one or many2many field, the server uses the name\_search function when importing. So, for a many2one field, it is better to export the field 'name' or 'code' of the related resource only. Use the more unique one. Be sure that the field you export is searchable by the name\_search function. (the 'name' column is always searchable):

```
Order Description,Partner/Code
```

- Change the title of the column for all many2many or many2one fields. It's because you export the related resource and you import a link on the resource. Example from a sale order: Partner/Code should become partner\_id and not partner\_id/code. If you kept the /code, OpenERP will try to create those entries in the database instead of finding references to existing ones.
- Many2many fields. If all the exported data contains 0 or 1 relation on each many2many fields, there will be no problem. Otherwise, the export will result in one line per many2many. The import function expects to get all many2many relations in one column, separated by a comma.

So, you have to make the transformation. For example, if the categories "Customer" and "Supplier" already exists:

```
name,category_id  
Smith, "Customer, Supplier"
```

If you want to create these two categories you can try

```
name,category_id/name  
Smith, "Customer, Supplier"
```

But this does not work as expected: a category "Customer, Supplier" is created. The solution is to create an empty line with only the second category:

```
name,category_id/name
Smith, Customer
,Supplier
```

Note the comma before “Supplier”!

- Read only fields. Do not try to import read only fields like the amount receivable or payable for a partner. Otherwise, OpenERP will not accept to import your file.
- Exporting trees. You can export and import tree structures using the parent field. You just have to take care of the import order. The parent have to be created before his child's.

#### 47.4.4 Use record id like in xml file

It's possible to define an id for each line of the csv file. This allow to define references between records:

```
id, name, parent_id:id record_one, Father, record_two, Child, record_one
```

If you do this, the line with the parent data must be before the child lines in the file.

### 47.5 Multiple CSV Files

#### 47.5.1 Importing from multiple CSV a full group of linked data

It's possible to import a lot of data, with multiple CSV files imported as a single operation. Assume we have a database with books and authors with a relation many2many between book and author.

And that you already have a file with a lot of books (like a library) and an other file with a lot of authors and a third file with the links between them.

How to import that easily in openERP ?

#### 47.5.2 Definition of an import module

You can do this in the module you have defined to manage yours books and authors. but Sometimes, the tables to import can also be in several modules.

For this example, let's say that 'book' object is defined in a module called 'library\_management' and that 'Author' object in a module called 'contact\_name'.

In this case, you can create a 'fake' module, just to import the data for all these multiples modules. Call this importation module : 'import\_my\_books'.

You create this module as others modules of OpenObject :

1. create a folder 'import\_my\_books'
2. inside, create a '\_\_init\_\_.py' file with only one line : import import\_my\_books
3. again, in the same folder, create a '\_\_openerp\_\_.py' file and in this file, write the following code :

```
# -*- encoding: utf-8 -*-
{
    'name': 'My Book Import',
    'category': 'Data Module 1',
    'init_xml': [],
    'author': 'mySelf & I',
    'depends': ['base','library_management','contact_name'],
    'version': '1.0',
    'active': False,
    'demo_xml': [],
}
```

```

'update_xml': ['contact_name.author.csv', 'library.book.csv'],
'installable': True
}

```

### 47.5.3 Creation of CSV files

For the CSV files, you'll import one after the other.

So you have to choose in which way you'll treat the many2many relation. For our example, we've choose to import all the authors, then all the books with the links to the authors.

#### 1. authors CSV file

You have to put your data in a CSV file without any link to books (because the book ids will be known only AFTERWARDS...) For example : ("contact\_name.author.csv")

```

id,last_name,first_name,type
author_1,Bradley,Marion Zimmer,Book writer
author_2,"Szu T'su",,Chinese philosopher
author_3,Zelazny,Roger,Book writer
author_4,Arleston,Scotch,Screen Writer
author_5,Magnin,Florence,Comics Drawer
...

```

#### 1. Books CSV file

Here, you can put the data about your books, but also, the links to the authors, using the same id as the column 'id' of the author CSV file. For example : ("library.book.csv" )

```

id,title,isbn,pages,date,author_ids:id
book_a,Les Cours du Chaos,1234567890123,268,1975-12-25,"author_3"
book_b,"L'art de la Guerre, en 219 volumes",1234567890124,1978-01-01,"author_2"
book_c,"new marvellous comics",1587459248579,2009-01-01,"author_5,author_4"
...

```

Five remarks :

1. the field content must be enclosed in double quotes (") if there is a double quote or a comma in the field.
2. the dates are in the format YYYY-MM-DD
3. if you have many ids in the same column, you must separate them with a comma, and, by the way, you must enclosed the content of the column between double quotes...
4. the name of the field is the same as the name of the field in the class definition AND must be followed by ':id' if the content is an ID that must be interpreted by the import module. In fact, "author\_4" will be transformed by the import module in an integer id for the database module and this numerical id will be put also in the table between author and book, not the literal ID (author\_4).
5. the encoding to be used by the CSV file is the 'UTF-8' encoding



# DATA MIGRATION - IMPORT / EXPORT

## 48.1 Data Importation

### 48.1.1 Introduction

There are different methods to import your data into OpenERP:

- Through the web-service interface
- Using CSV files through the client interface
- Building a module with .XML or .CSV files with the content
- Directly into the SQL database, using an ETL

### 48.1.2 Importing data through a module

The best way to import data in OpenERP is to build a module that integrates all the data you want to import. So, when you want to import all the data, you just have to install the module and OpenERP manages the different creation operations. When you have lots of different data to import, we sometimes create different modules.

So, let's create a new module where we will store all our data. To do this, from the addons directory, create a new module called data\_yourcompany.

- mkdir data\_yourcompany
- cd data\_yourcompany
- touch \_\_init\_\_.py

You must also create a file called \_\_openerp\_\_.py in this new module. Write the following content in this module file description.

```
{  
    'name': 'Module for Data Importation',  
    'version': '1.0',  
    'category': 'Generic Modules/Others',  
    'description': "Sample module for data importation.",  
    'author': 'Tiny',  
    'website': 'http://www.openerp.com',  
    'depends': ['base'],  
    'init_xml': [  
        'res.partner.csv',  
        'res.partner.address.csv'  
    ],  
    'update_xml': [],  
    'installable': True,  
    'active': False,  
}
```

The following module will import two different files:

- res.partner.csv : a CSV file containing records of the res.partner object
- res.partner.address.csv : a CSV file containing records of the res.partner.address object

Once this module is created, you must load data from your old application to .CSV file that will be loaded in OpenERP. OpenERP has a builtin system to manage identifications columns of the original software.

For this exercise, we will load data from another OpenERP database called old. As this database is in SQL, it's quite easy to export the data using the command line postgresql client: psql. As to get a result that looks like a .CSV file, we will use the following arguments of psql:

- -A : display records without space for the row separators
- -F , : set the separator character as ‘,’
- --pset footer : don't write the latest line that looks like “(21 rows)”

When you import a .CSV file in OpenERP, you can provide a ‘id’ column that contains a uniq identification number or string for the record. We will use this ‘id’ column to refer to the ID of the record in the original application. As to refer to this record from a many2one field, you can use ‘FIELD\_NAME:id’. OpenERP will re-create the relationship between the record using this uniq ID.

So let's start to export the partners from our database using psql:::

```
psql trunk -c "select 'partner_'||id as id, name from res_partner"  
-A -F , --pset footer > res.partner.csv
```

This creates a res.partner.csv file containing a structure that looks like this:

```
id,name  
partner_2,ASUSTek  
partner_3,Agrolait  
partner_4,Campnocamp  
partner_5,Syleam
```

By doing this, we generated data from the res.partner object, by creating a uniq identification string for each record, which is related to the old application's ID.

Now, we will export the table with addresses (or contacts) that are linked to partners through the relation field: partner\_id. We will proceed in the same way to export the data and put them into our module:

```
psql trunk -c "select 'partner_address'||id as id, name, 'partner_'||  
partner_id as \"partner_id:id\" from res_partner_address"  
-A -F , --pset footer > res.partner.address.csv
```

This should create a file called res.partner.address with the following data:

```
id,name,partner_id:id  
partner_address2,Benoit Mortier,partner_2  
partner_address3,Laurent Jacot,partner_3  
partner_address4,Laith Jubair,partner_4  
partner_address5,Fabien Pinckaers,partner_4
```

When you will install this module, OpenERP will automatically import the partners and then the address and recreate efficiently the link between the two records. When installing a module, OpenERP will test and apply the constraints for consistency of the data. So, when you install this module, it may crash, for example, because you may have different partners with the same name in the system. (due to the uniq constraint on the name of a partner). So, you have to clean your data before importing them.

If you plan to upload thousands of records through this technique, you should consider using the argument ‘-P’ when running the server.

```
openerp_server.py -P status.pickle --init=data_yourcompany
```

This method provides a faster importation of the data and, if it crashes in the middle of the import, it will continue at the same line after rerunning the server. This may preserves hours of testing when importing big files.

### 48.1.3 Using OpenERP's ETL

The next version of OpenERP will include an ETL module to allow you to easily manage complex import jobs. If you are interested in this system, you can check the complete specifications and the available prototype at this location:

```
bzr branch lp:~openerp-committer/openobject-addons/trunk-extra-addons/etl  
... to be continued ...
```

## 48.2 Data Loading

During OpenERP installation, two steps are necessary to create and feed the database:

1. Create the SQL tables
2. Insert the different data into the tables

The creation (or modification in the case of an upgrade) of SQL tables is automated thanks to the description of objects in the server.

Into OpenERP, all the logic of the application is stored in the database. We find for example:

- the definitions of the reports,
- the object default values,
- the form description of the interface client,
- the relations between the menu and the client buttons, ...

There must be a mechanism to describe, modify and reload the different data. These data are represented into a set of XML files that can possibly be loaded during start of the program in order to fill in the tables.



# UPGRADING

**Warning:**

*This section needs to be rewritten or improved. If you think you can contribute to this effort, and are already familiar with Launchpad and OpenERP's source control system, Bazaar, please have a look at:*

- the section explaining how you can download and build the current documentation on your system: building\_documentation*
- an RST primer such as [this one](#) to learn how you can start modifying the documentation content*

## 49.1 Upgrading Server, Modules

The upgrade from version to version is automatic and doesn't need any special scripting on the user's part. In fact, the server is able to automatically rebuild the database and the data from a previously installed version.

The tables are rebuilt from the current module definitions. To rebuild the tables, the server uses the definition of the objects and adds / modifies database fields as necessary.

To invoke a database upgrade after installing a new version, you need to start the server with the **--update=all** argument :

```
openerp-server.py --update=all
```

You can also only upgrade specific modules, for example:

```
openerp-server.py --update=account,base
```

The database is rebuilt according to information provided in XML files and Python Classes. You can also execute the server with **--init=all**. The server will then rebuild the database according to the existing XML files on the system, delete all existing data and return OpenERP to its basic configuration.

### 49.1.1 Detailed update operations

OpenERP has a built-in migration and upgrade system which allows updates to be nearly (or often) automatic. This system also allows to easily include custom modules.

#### Table/Object structure

When you run `openerp-server` with option `--init` or `--update`, the table structure is updated to match the new description that is in Python code. Fields that are removed from Python code are not removed from the postgresql database to avoid losing data.

So, simply running with `--update` or `--init`, will upgrade your table structure.

It's important to run `--init=module` the first time you install the module. Next time, you must use the `--update=module` argument instead of the `init` one. This is because `--init` loads resources that are loaded

only once and never upgraded (i.e., resources with no `id=""` attribute or within a `<data nouupdate="1">` tag). Resources with the `nouupdate` attribute will still be created if they do not exist at upgrade time. This can be overridden by marking a record with `forcecreate="False"`.

## Data

Some data is automatically loaded at the installation of OpenERP:

- views, actions, menus,
- workflows,
- demo data

This data is also migrated to a new version if you run `--update` or `--init`.

## Workflows

Workflows are also upgraded automatically. If some activities are removed, the documents states evolves automatically to the preceding activities. That ensure that all documents are always in valid states.

You can freely remove activities in your XML files. If workitems are in this activity, they will evolve to the preceding unlinked activity. And after the activity will be removed.

## Things to care about during development

Since version 3.0.2 of OpenERP, you can not use twice the same ‘`id="..."`’ during resource creation in your XML files, unless they are in two different modules.

Resources which don’t contain an id are created (and updated) only once; at the installation of the module or when you use the `--init` argument.

If a resource has an id and this resource is not present anymore in the next version of the XML file, OpenERP will automatically remove it from the database. If this resource is still present, OpenERP will update the modifications to this resource.

If you use a new id, the resource will be automatically created at the next update of this module.

**Use explicit id declaration !**, Example:

- `view_invoice_form`,
- `view_move_line_tree`,
- `action_invoice_form_open`, ...

It is important to put `id="..."` to all record that are important for the next version migrations. For example, do not forget to put some `id="..."` on all workflows transitions. This will allows OpenERP to know which transition has been removed and which transition is new or updated.

## Custom modules

For example, if you want to override the view of an object named ‘`invoice_form`’ in your xml file (`id="invoice_form"`). All you have to do is redefine this view in your custom module with the same id. You can prefix ids with the name of the module to reference an id defined in another module.

Example:

```
<record model="ir.ui.view" id="account.invoice_form"> ... <record>
```

This will override the invoice form view. You do not have to delete the old view, like in 3.0 versions of OpenERP.

Note that it is often better to use view inheritance instead of overwriting views.

In this migration system, you do not have to delete any resource. The migration system will detect if it is an update or a delete using `id="..."` attributes. This is important to preserve references during migrations.

## Demo data

Demo data does not have to be upgraded; because it was probably modified or deleted by users. To avoid demo data being upgraded you can put a `noupdate="1"` attribute in the `<data>` tag of your .xml data files.

### 49.1.2 Summary of update and init process

init:

modify/add/delete demo data and built-in data

update:

modify/add/delete non demo data

Examples of built-in (non demo) data:

- Menu structure,
- View definition,
- Workflow description, ...
- Everything that has an `id` attribute in the XML data declaration (if no attr `noupdate="1"` in the header)

What's going on during the update process:

**1. If you manually added data within the client:**

- the update process will not change them

**2. If you dropped data:**

- if it was demo data, the update process will do nothing
- if it was built-in data (like a view), the update process will recreate it

**3. If you modified data (either in the .XML or the client):**

- if it's demo data: nothing
- if it's built-in data, data are updated

**4. If built-in data have been deleted in the .XML file:**

- this data will be deleted in the database.



## **Part XIII**

### **API**



# WORKING WITH WEB SERVICES

Given the architecture of OpenERP, it is not possible to reliably access the database with the PostgreSQL client or through a direct connection method such as ODBC. Fortunately, OpenERP provides a very comprehensive set of web services that allow you to do everything through standard protocols.

**Note:**

*Though it is technically possible, you must be aware that this can have disastrous consequences for your data, unless you know exactly what you are doing. You are advised to shut down the OpenERP server when accessing the database to avoid caching and concurrency issues.*

---

## 50.1 Supported Web Services Protocols

The currently supported protocols are XML-RPC and Net-RPC. XML-RPC is one of the first standard for web services, and can be used in almost any language. It is a pretty verbose protocol, which may sometimes introduce a bit of latency. Net-RPC, on the other hand, is an optimized protocol particularly designed for use between applications written in Python.

Support for REST-style webservices is planned for future releases of OpenERP.

Support for the SOAP protocol is deprecated at the moment, but could maybe be revived if sufficient interest is found in the community.

## 50.2 Available Web Services

The OpenERP server provides you with the following web services.

**Note:**

*You may find out the details of each service in the corresponding class in the server sources, in bin/service/web\_services.py .*

---

**db** Provides functions to create, drop, backup and restore databases. Use with caution!

**common** Lets you log in and out of OpenERP, and provides various utility functions. You will need to call the function “login” before you can use most of the other web services.

**object** The most useful web service, as it provides access to the OpenERP Objects. Most notably, the function “execute” lets you call methods of the Objects, such as most of the ORM methods to search, read and write records. It can also be used to call any other method of the object, such as computing a price for example.

**Note:**

Here is a quick reminder of the main ORM methods:

**create({‘field’:’value’})**

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

**search([({‘arg1’:’=’,’value1’})...], offset=0, limit=1000)**

- arg1, arg2, .. ,argN: list of tuples specifying search criteria
- offset: optional number of records to skip
- limit: optional max number of records to return
- Returns: list of IDS of records matching the given criteria

**read([IDS], [‘field1’,’field2’,...])**

- fields: optional list of field names to return (default: all fields)
- Returns: the id of each record and the values of the requested field

**write([IDS], {‘field1’:’value1’,’field2’:3})**

- values: dictionary of field values to update
- Updates records with given ids with the given values
- Returns: True

**unlink([IDS])**

- Deletes records with the given ids
- Returns: True

**browse()** can’t be used through web services.

Another useful function is “exec\_workflow”, which lets you make a record progress through a workflow.

**wizard**

Provides access to the old-style wizards. Please note that the new-style wizards are based on the ORM, and as such they can be accessed though the “object” web service.

**report**

Lets you generate and retrieve reports.

## 50.3 Example : writing data through the Web Services

Here is an example process that you could follow to write data. You will find more detailed examples for XML-RPC in various programming languages in the next chapter.

1. login: call “login” in the web service “common” with the following parameters:

- database
- user name
- password

2. create a new partner: call “execute” in the web service “object” with the following parameters:

- database
- user id provided by “login” in step 1.
- the object name : ‘res.partner’
- the name of the ORM method : “create”
- some data to be recorded

The data mentioned above is a dictionary of keys and values, for example:

- name: Fabien Pinckaers
- lang: fr\_FR

But more complex data structures can also be sent - for example you could record a partner and their addresses, all in a single call to the web service. In that case, all the data is processed by the server during the same database transaction - meaning you are sure to keep a consistent state for your data - a critical requirement for all ERP applications.



# XML-RPC WEB SERVICES

XML-RPC is known as a web service. Web services are a set of tools that let one build distributed applications on top of existing web infrastructures. These applications use the Web as a kind of “transport layer” but don’t offer a direct human interface via the browser.<sup>[1]</sup> Extensible Markup Language (XML) provides a vocabulary for describing Remote Procedure Calls (RPC), which is then transmitted between computers using the HyperText Transfer Protocol (HTTP). Effectively, RPC gives developers a mechanism for defining interfaces that can be called over a network. These interfaces can be as simple as a single function call or as complex as a large API.

XML-RPC therefore allows two or more computers running different operating systems and programs written in different languages to share processing. For example, a Java application could talk with a Perl program, which in turn talks with Python application that talks with ASP, and so on. System integrators often build custom connections between different systems, creating their own formats and protocols to make communications possible, but one can often end up with a large number of poorly documented single-use protocols. The RPC approach spares programmers the trouble of having to learn about underlying protocols, networking, and various implementation details.

XML-RPC can be used with Python, Java, Perl, PHP, C, C++, Ruby, Microsoft’s .NET and many other programming languages. Implementations are widely available for platforms such as Unix, Linux, Windows and the Macintosh.

An XML-RPC call is conducted between two parties: the client (the calling process) and the server (the called process). A server is made available at a particular URL (such as <http://example.org:8080/rpcserv/>).

The above text just touches the surface of XML-RPC. I recommend O’Reilly’s “Programming Web Service with XML-RPC” for further reading. One may also wish to review the following links:

## 51.1 Interfaces

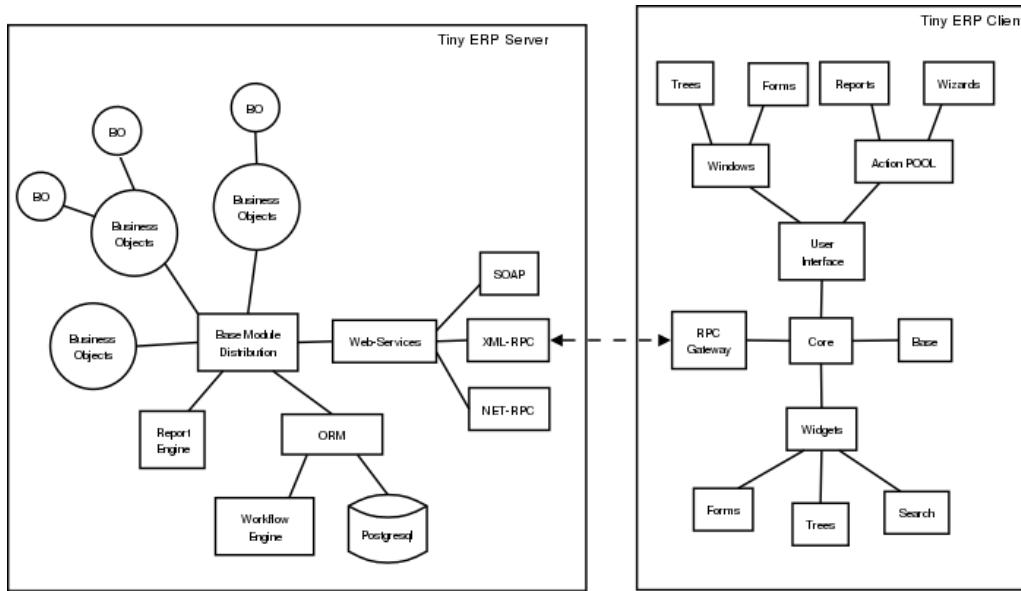
### 51.1.1 XML-RPC

#### XML-RPC Architecture

OpenERP is based on a client/server architecture. The server and the client(s) communicate using the XML-RPC protocol. XML-RPC is a very simple protocol which allows the client to do remote procedure calls. The called function, its arguments, and the result of the call are transported using HTTP and encoded using XML. For more information on XML-RPC, please see: <http://www.xml-rpc.com>.

#### Architecture

The diagram below synthesizes the client server architecture of OpenERP. OpenERP server and OpenERP clients communicate using XML-RPC.



## Client

The logic of OpenERP is configured on the server side. The client is very simple; it is only used to post data (forms, lists, trees) and to send back the result to the server. The updates and the addition of new functionality don't need the clients to be frequently upgraded. This makes OpenERP easier to maintain.

The client doesn't understand what it posts. Even actions like 'Click on the print icon' are sent to the server to ask how to react.

The client operation is very simple; when a user makes an action (save a form, open a menu, print, ...) it sends this action to the server. The server then sends the new action to execute to the client.

There are three types of action;

- Open a window (form or tree)
- Print a document
- Execute a wizard

### 51.1.2 Python

#### Access tiny-server using xml-rpc

##### Demo script

- Create a partner and their address

```
import xmlrpclib

username = 'admin' #the user
pwd = 'admin'      #the password of the user
dbname = 'terp'    #the database

# Get the uid
sock_common = xmlrpclib.ServerProxy ('http://localhost:8069/xmlrpc/common')
uid = sock_common.login(dbname, username, pwd)

#replace localhost with the address of the server
sock = xmlrpclib.ServerProxy('http://localhost:8069/xmlrpc/object')

partner = {
```

```

        'name': 'Fabien Pinckaers',
        'lang': 'fr_FR',
    }

partner_id = sock.execute(dbname, uid, pwd, 'res.partner', 'create', partner)

address = {
    'partner_id': partner_id,
    'type' : 'default',
    'street': 'Chaussée de Namur 40',
    'zip': '1367',
    'city': 'Grand-Rosière',
    'phone': '+3281813700',
    'fax': '+3281733501',
}

address_id = sock.execute(dbname, uid, pwd, 'res.partner.address', 'create', address)

```

- **Search a partner**

```

args = [('vat', '=', 'ZZZZZZ')] #query clause
ids = sock.execute(dbname, uid, pwd, 'res.partner', 'search', args)

```

- **Read partner data**

```

fields = ['name', 'active', 'vat', 'ref'] #fields to read
data = sock.execute(dbname, uid, pwd, 'res.partner', 'read', ids, fields) #ids is a list of i

```

- **Update partner data**

```

values = {'vat': 'ZZ1ZZZ'} #data to update
result = sock.execute(dbname, uid, pwd, 'res.partner', 'write', ids, values)

```

- **Delete partner**

```

# ids : list of id
result = sock.execute(dbname, uid, pwd, 'res.partner', 'unlink', ids)

```

## 51.1.3 PHP

### Access Open-server using xml-rpc

#### Download the XML-RPC framework for PHP

windows / linux: download the xml-rpc framework for php from <http://phpxmlrpc.sourceforge.net/> The latest stable release is version 2.2 released on February 25, 2007

#### Setup the XML-RPC for PHP

extract file xmlrpc-2.2.tar.gz and take the file xmlrpc.inc from lib directory place the xmlrpc.inc in the php library folder restart the apache/iis server

#### Demo script

- **Login**

```

function connect() {
    var $user = 'admin';
    var $password = 'admin';
    var $dbname = 'db_name';
    var $server_url = 'http://localhost:8069/xmlrpc/';

```

```

if(isset($_COOKIE["user_id"])) == true) {
    if($_COOKIE["user_id"]>0) {
        return $_COOKIE["user_id"];
    }
}

$sock = new xmlrpc_client($server_url.'common');
$msg = new xmlrpcreq('login');
$msg->addParam(new xmlrpcval($dbname, "string"));
$msg->addParam(new xmlrpcval($user, "string"));
$msg->addParam(new xmlrpcval($password, "string"));
$resp = $sock->send($msg);
$val = $resp->value();
$id = $val->scalarval();
setcookie("user_id",$id,time()+3600);
if($id > 0) {
    return $id;
} else{
    return -1;
}
}

```

### • Search

```

/**
 * $client = xml-rpc handler
 * $relation = name of the relation ex: res.partner
 * $attribute = name of the attribute ex:code
 * $operator = search term operator ex: ilike, =, !=
 * $key=search for
 */

function search($client,$relation,$attribute,$operator,$keys) {
    var $user = 'admin';
    var $password = 'admin';
    var $userId = -1;
    var $dbname = 'db_name';
    var $server_url = 'http://localhost:8069/xmlrpc/';

    $key = array(new xmlrpcval(array(new xmlrpcval($attribute , "string"),
        new xmlrpcval($operator,"string"),
        new xmlrpcval($keys,"string")),"array"),
    );

    if($userId<=0) {
        connect();
    }

    $msg = new xmlrpcreq('execute');
    $msg->addParam(new xmlrpcval($dbname, "string"));
    $msg->addParam(new xmlrpcval($userId, "int"));
    $msg->addParam(new xmlrpcval($password, "string"));
    $msg->addParam(new xmlrpcval($relation, "string"));
    $msg->addParam(new xmlrpcval("search", "string"));
    $msg->addParam(new xmlrpcval($key, "array"));

    $resp = $client->send($msg);
    $val = $resp->value();
    $ids = $val->scalarval();

    return $ids;
}

```

- Create

```
<?

include('xmlrpc.inc');

$arrayVal = array(
'name'=>new xmlrpcval('Fabien Pinckaers', "string") ,
'vat'=>new xmlrpcval('BE477472701' , "string")
);

$client = new xmlrpc_client("http://localhost:8069/xmlrpc/object");

$msg = new xmlrpcrequest('execute');
$msg->addParam(new xmlrpcval("dbname", "string"));
$msg->addParam(new xmlrpcval("3", "int"));
$msg->addParam(new xmlrpcval("demo", "string"));
$msg->addParam(new xmlrpcval("res.partner", "string"));
$msg->addParam(new xmlrpcval("create", "string"));
$msg->addParam(new xmlrpcval($arrayVal, "struct"));

$resp = $client->send($msg);

if ($resp->faultCode())

    echo 'Error: '.$resp->faultString();

else

    echo 'Partner '.$resp->value()->scalarval().' created !';

?>
```

- Write

```
/***
* $client = xml-rpc handler
* $relation = name of the relation ex: res.partner
* $attribute = name of the attribute ex:code
* $operator = search term operator ex: ilike, =, !=
* $id = id of the record to be updated
* $data = data to be updated
*/

function write($client,$relation,$attribute,$operator,$data,$id) {
    var $user = 'admin';
    var $password = 'admin';
    var $userId = -1;
    var $dbname = 'db_name';
    var $server_url = 'http://localhost:8069/xmlrpc/';

    $id_val = array();
    $id_val[0] = new xmlrpcval($id, "int");

    if($userId<=0) {
        connect();
    }

    $msg = new xmlrpcrequest('execute');
    $msg->addParam(new xmlrpcval($dbname, "string"));
    $msg->addParam(new xmlrpcval($userId, "int"));
    $msg->addParam(new xmlrpcval($password, "string"));
    $msg->addParam(new xmlrpcval($relation, "string"));
    $msg->addParam(new xmlrpcval("write", "string"));
```

```

$msg->addParam(new xmlrpcval($id, "array"));
$msg->addParam(new xmlrpcval($data, "struct"));

$resp = $client->send($msg);
$val = $resp->value();
$record = $val->scalarval();

return $record;

}

```

## 51.1.4 JAVA

### Access Open-server using xml-rpc

#### Download the apache XML-RPC framework for JAVA

Download the xml-rpc framework for java from <http://ws.apache.org/xmlrpc/> The latest stable release is version 3.1 released on August 12, 2007. All OpenERP errors throw exceptions because the framework allows only an int as the error code where OpenERP returns a string.

#### Demo script

- Find Databases

```

import java.net.URL;
import java.util.Vector;

import org.apache.commons.lang.StringUtils;
import org.apache.xmlrpc.XmlRpcException;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public Vector<String> getDatabaseList(String host, int port)
{
    XmlRpcClient xmlrpcDb = new XmlRpcClient();

    XmlRpcClientConfigImpl xmlrpcConfigDb = new XmlRpcClientConfigImpl();
    xmlrpcConfigDb.setEnabledForExtensions(true);
    xmlrpcConfigDb.setServerURL(new URL("http", host, port, "/xmlrpc/db"));

    xmlrpcDb.setConfig(xmlrpcConfigDb);

    try {
        //Retrieve databases
        Vector<Object> params = new Vector<Object>();
        Object result = xmlrpcDb.execute("list", params);
        Object[] a = (Object[]) result;

        Vector<String> res = new Vector<String>();
        for (int i = 0; i < a.length; i++) {
            if (a[i] instanceof String) {
                res.addElement((String)a[i]);
            }
        }
        catch (XmlRpcException e) {
            logger.warn("XmlException Error while retrieving OpenERP Databases: ",e);
            return -2;
        }
        catch (Exception e)

```

```

    {
        logger.warn("Error while retrieving OpenERP Databases: ",e);
        return -3;
    }
}

• Login

import java.net.URL;  

import org.apache.commons.lang.StringUtils;  

import org.apache.xmlrpc.XmlRpcException;  

import org.apache.xmlrpc.client.XmlRpcClient;  

import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;  

public int Connect(String host, int port, String tinydb, String login, String password)  

{
    XmlRpcClient xmlrpcLogin = new XmlRpcClient();  

    XmlRpcClientConfigImpl xmlrpcConfigLogin = new XmlRpcClientConfigImpl();  

    xmlrpcConfigLogin.setEnabledForExtensions(true);  

    xmlrpcConfigLogin.setServerURL(new URL("http",host,port,"/xmlrpc/common"));  

    xmlrpcLogin.setConfig(xmlrpcConfigLogin);  

try {  

    //Connect  

    params = new Object[] {tinydb,login,password};  

    Object id = xmlrpcLogin.execute("login", params);  

    if (id instanceof Integer)  

        return (Integer)id;  

    return -1;  

}  

catch (XmlRpcException e) {  

    logger.warn("XmlException Error while logging to OpenERP: ",e);
    return -2;
}  

catch (Exception e)
{
    logger.warn("Error while logging to OpenERP: ",e);
    return -3;
}
}
}

```

- **Search**

TODO

- **Create**

TODO

- **Write**

TODO

## 51.2 Python Example

Example of creation of a partner and their address.

```
import xmlrpclib
```

```

sock = xmlrpclib.ServerProxy('http://localhost:8069/xmlrpc/object')
uid = 1
pwd = 'demo'

partner = {
    'title': 'Monsieur',
    'name': 'Fabien Pinckaers',
    'lang': 'fr',
    'active': True,
}

partner_id = sock.execute(dbname, uid, pwd, 'res.partner', 'create', partner)

address = {
    'partner_id': partner_id,
    'type': 'default',
    'street': 'Rue du vieux chateau, 21',
    'zip': '1457',
    'city': 'Walhain',
    'phone': '(+32)10.68.94.39',
    'fax': '(+32)10.68.94.39',
}
sock.execute(dbname, uid, pwd, 'res.partner.address', 'create', address)

```

To get the UID of a user, you can use the following script:

```

sock = xmlrpclib.ServerProxy('http://localhost:8069/xmlrpc/common')
UID = sock.login('terp3', 'admin', 'admin')

```

**CRUD example:**

```

"""
:The login function is under
::      http://localhost:8069/xmlrpc/common
:For object retrieval use:
::      http://localhost:8069/xmlrpc/object
"""

import xmlrpclib

user = 'admin'
pwd = 'admin'
dbname = 'terp3'
model = 'res.partner'

sock = xmlrpclib.ServerProxy('http://localhost:8069/xmlrpc/common')
uid = sock.login(dbname, user, pwd)

sock = xmlrpclib.ServerProxy('http://localhost:8069/xmlrpc/object')

# CREATE A PARTNER
partner_data = {'name': 'Tiny', 'active': True, 'vat': 'ZZZZZ'}
partner_id = sock.execute(dbname, uid, pwd, model, 'create', partner_data)

# The relation between res.partner and res.partner.category is of type many2many
# To add categories to a partner use the following format:
partner_data = {'name': 'Provider2', 'category_id': [(6, 0, [3, 2, 1])]}
# Where [3, 2, 1] are id fields of lines in res.partner.category

# SEARCH PARTNERS
args = [('vat', '=', 'ZZZZZ'),]
ids = sock.execute(dbname, uid, pwd, model, 'search', args)

```

```

# READ PARTNER DATA
fields = ['name', 'active', 'vat', 'ref']
results = sock.execute(dbname, uid, pwd, model, 'read', ids, fields)
print results

# EDIT PARTNER DATA
values = {'vat':'ZZ1ZZ'}
results = sock.execute(dbname, uid, pwd, model, 'write', ids, values)

# DELETE PARTNER DATA
results = sock.execute(dbname, uid, pwd, model, 'unlink', ids)

```

PRINT example:

1. PRINT INVOICE
2. IDS is the invoice ID, as returned by:
3. ids = sock.execute(dbname, uid, pwd, 'account.invoice', 'search', [('number', 'ilike', invoicenumber), ('type', '=', 'out\_invoice')])

```

import time
import base64
printsock = xmlrpclib.ServerProxy('http://server:8069/xmlrpc/report')
model = 'account.invoice'
id_report = printsock.report(dbname, uid, pwd, model, ids, {'model': model, 'id': ids[0], 'report_type': 'pdf'})
time.sleep(5)
state = False
attempt = 0
while not state:
    report = printsock.report_get(dbname, uid, pwd, id_report)
    state = report['state']
    if not state:
        time.sleep(1)
        attempt += 1
    if attempt > 200:
        print 'Printing aborted, too long delay !'

    string_pdf = base64.decodestring(report['result'])
    file_pdf = open('/tmp/file.pdf', 'w')
    file_pdf.write(string_pdf)
    file_pdf.close()

```

## 51.3 PHP Example

Here is an example on how to insert a new partner using PHP. This example makes use of the phpxmlrpc library, available on sourceforge.

```

<?
include('xmlrpc.inc');

$arrayVal = array(
'name'=>new xmlrpcval('Fabien Pinckaers', "string") ,
'vat'=>new xmlrpcval("BE477472701" , "string")
);

$client = new xmlrpc_client("http://localhost:8069/xmlrpc/object");

$msg = new xmlrpcmsg('execute');
$msg->addParam(new xmlrpcval("dbname", "string"));

```

```

$msg->addParam(new xmlrpcval("3", "int"));
$msg->addParam(new xmlrpcval("demo", "string"));
$msg->addParam(new xmlrpcval("res.partner", "string"));
$msg->addParam(new xmlrpcval("create", "string"));
$msg->addParam(new xmlrpcval($arrayVal, "struct"));

$resp = $client->send($msg);

if ($resp->faultCode())
    echo 'Error: '.$resp->faultString();

else
    echo 'Partner '.$resp->value()->scalarval().' created !';

?>

```

## 51.4 Perl Example

Here is an example in Perl for creating, searching and deleting a partner.

```

#!/usr/bin/perl
# 17-02-2010
# OpenERP XML RPC communication example
# Todor Todorov <todorov@hp.com> <tttodorov@yahoo.com>

use strict;
use Frontier::Client;
use Data::Dumper;

my ($user) = 'admin';
my ($pw) = 'admin';
my ($db) = 'put_your_dbname_here';
my ($model) = 'res.partner';

#login
my $server_url = 'http://localhost:8069/xmlrpc/common';
my $server = Frontier::Client->new('url' => $server_url);
my $uid = $server->call('login', $db, $user, $pw);

print Dumper($uid);

my $server_url = 'http://localhost:8069/xmlrpc/object';
my $server = Frontier::Client->new('url' => $server_url);

print Dumper($server);

#
# CREATE A PARTNER
#
my $partner_data = {'name'=>'MyNewPartnerName',
                    'active'=> 'True',
                    'vat'=>'ZZZZZ'};
my $partner_id = $server->call('execute', $db, $uid, $pw, $model, 'create', $partner_data);

print Dumper($partner_id);

#
# SEARCH PARTNERS

```

```

#
my $query = [['vat', '=', 'ZZZZZ']];
print Dumper($query);

my $ids = $server->call('execute', $db, $uid, $pw, $model, 'search', $query);
print Dumper($ids);

#Here waiting for user input
#OpenERP interface my be checked if partner is shown there

print $/."Check OpenERP if partner is inserted. Press ENTER".$/;
<STDIN>

#
# DELETE PARTNER DATA
#
my $results = $server->call('execute', $db, $uid, $pw, $model, 'unlink', $ids);
print Dumper($results);

```

Everything done in the GTK or web client in OpenERP is through XML/RPC webservices. Start openERP GTK client using ./openerp-client.py -l debug\_rpc (or debug\_rpc\_answer) then do what you want in the GTK client and watch your client logs, you will find out the webservice signatures. By creating indents in the logs will help you to spot which webservice you want.



## **Part XIV**

# **Build and deploy**



This page describes how to build a custom version of OpenERP for Windows.



# BUILDING

## 52.1 Dependencies

The first step is to build the dependencies. To do so, grab the Windows installer branch:

```
bzr branch lp:~openerp-groupes/openerp/win-installer-trunk
```

and install the packages:

- 7z465.msi
- python-2.5.2.msi
- setuptools-0.6c9.win32-py2.5.exe
- Beaker-1.4.1.tar.gz
- Mako-0.2.4.tar.gz
- pytz-2010l.win32.exe

### 52.1.1 Server

Install the packages:

- lxml-2.1.2.win32-py2.5.exe
- PIL-1.1.6.win32-py2.5.exe
- psycopg2-2.2.2.win32-py2.5-pg9.0.1-release.exe
- PyChart-1.39.win32.exe
- pydot-1.0.2.win32.exe
- python-dateutil-1.5.tar.gz
- pywin32-212.win32-py2.5.exe
- PyYAML-3.09.win32-py2.5.exe
- ReportLab-2.2.win32-py2.5.exe

### 52.1.2 Web

Install the packages:

- Babel-0.9.4-py2.5.egg
- CherryPy-3.1.2.win32.exe
- FormEncode-1.2.2.tar.gz

- simplejson-2.0.9-py2.5-win32.egg
- xlwt-0.7.2.win32.exe

## 52.2 Source distribution

The second step is to build a source distribution on Linux.

### 52.2.1 Server

Let's assume you work on your own server branch named **6.0** and you want to build a server with the following modules:

- base\_setup
- base\_tools
- board

This implies that these modules have been linked in *bin addons* by a command similar to:

```
ln -s ~/openerp/addons/6.0/{base_setup,base_tools,board} .
```

To build the server, go to the root directory and type:

```
python setup.py sdist --format=zip
```

**You now have a new file in the dist directory, called openerp-server-M.m.P.zip where:**

- **M** is the major version, example 6
- **m** is the minor version, example 0
- **p** is the patch version, example 1

### 52.2.2 Web

To build the web client, go to the root directory and type:

```
python setup.py sdist --format=zip
```

**You now have a new file in the dist directory, called openerp-web-M.m.P.zip where:**

- **M** is the major version, example 6
- **m** is the minor version, example 0
- **p** is the patch version, example 1

## 52.3 Binary distribution

The third step is to build a binary distribution on Windows.

### 52.3.1 Server

Open a command prompt and unzip the file:

```
7z x openerp-server-M.m.P.zip -oC:\openerp
```

Go to the **win32** directory:

```
cd C:\openerp\openerp-server-M.m.P\win32
```

Generate the service exe with:

```
python setup.py py2exe
```

Go to the parent directory:

```
cd ..
```

Generate the server exe with:

```
python setup.py py2exe
```

Build the Windows installer with:

```
makensis setup.nsi
```

You now have a new file in the root directory, called `openerp-server-setup-M.m.P.exe`. This file is the installer that you can use to install a custom version of OpenERP.

### 52.3.2 Web

Open a command prompt and unzip the file:

```
7z x openerp-web-M.m.P.zip -oC:\openerp
```

Go to the **win32** directory:

```
cd C:\openerp\openerp-web-M.m.P\win32
```

Generate the service exe with:

```
python setup.py py2exe
```

Go to the parent directory:

```
cd ..
```

Generate the web exe with:

```
python setup.py py2exe
```

Build the Windows installer with:

```
makensis setup.nsi
```

You now have a new file in the root directory, called `openerp-web-setup-M.m.P.exe`. This file is the installer that you can use to install a custom version of OpenERP.



# DEPLOY

This page describes how to deploy a custom version of OpenERP on Windows.

## 53.1 Package script

The first step is to grab the package script branch:

```
bzr branch lp:~openerp-groupes/openerp/package-script
```

## 53.2 Batch

Go to the *packaging* directory of the branch and copy the file *build.bat* to the *C:\openerp* directory of your Windows machine.

## 53.3 SSH server

You need to install a SSH server on Windows. You can for example install [freeSSHd](#).

## 53.4 Fabric

You need to install the tool [Fabric](#) to run commands on Windows from Linux using SSH. Refer to your linux package manager to install it.

### 53.4.1 Configure

Go to the *packaging* directory of the branch and edit the file *fabfile.py*. Change what need to be changed.

### 53.4.2 Run

run the command:

```
fab -H host -u user server
```

where:

- *host* is the Windows host name
- *user* is the Windows user name



## **Part XV**

# **Appendice**



# CONVENTIONS

## 54.1 Guidelines

For guidelines and general recommendations with regard to the development of OpenERP modules, please refer to the *Guidelines* of the Contribution section.

## 54.2 Module structure and file names

The structure of a module should be:

```
/module/  
  /__init__.py  
  /__openerp__.py  
  /module.py  
  /module_other.py  
  /module_view.xml  
  /module_data.xml  
  /module_demo.xml  
  
  /wizard/  
  /wizard/__init__.py  
  /wizard/wizard_name.py  
  
  /report/  
  /report/  
  /report/__init__.py  
  /report/report_name.sxw  
  /report/report_name.rml  
  /report/report_name.py
```

## 54.3 Naming conventions

- **modules:** modules must be written in lower case, with underscores. The name of the module is the name of the directory
  - sale
  - sale\_commission
- **objects:** the name of an object must be of the form name\_of\_module.name1.name2.name3.... The namei part of the object name must be in lowercase.
  - sale.order

- sale.order.line
- sale.shop
- sale\_commission.commission.rate

- **fields:** field must be in lowercase, separated by underscores. Try to use commonly used names for fields: name, state, active

- many2one: must end by ‘\_id’ (eg: partner\_id, order\_line\_id)
- many2many: must end by ‘\_ids’ (eg: category\_ids)
- one2many: must end by ‘\_ids’ (eg: line\_ids)

# TRANSLATIONS

OpenERP is multilingual. You can add as many languages as you wish. Each user may work with the interface in his own language. Moreover, some resources (the text of reports, product names, etc.) may also be translated.

This section explains how to change the language of the program shown to individual users, and how to add new languages to OpenERP.

Nearly all the labels used in the interface are stored on the server. In the same way, the translations are also stored on the server. By default the English dictionary is stored on the server, so if the users want to try OpenERP in a language other than English you must store these languages definitions on the server.

However, it is not possible to store “everything” on the server. Indeed, the user gets some menus, buttons, etc... that must contain some text *even before* being connected to the server. These few words and sentences are translated using GETTEXT. The chosen language by default for these is the language of the computer from which the user connects.

The translation system of OpenERP is not limited to interface texts; it also works with reports and the “content” of some database fields. Obviously, not all the database fields need to be translated. The fields where the content is multilingual are marked thus by a flag icon.

## 55.1 How to change the language of the user interface ?

The language is a user preference. To change the language of the current user, click on the menu: User > Preferences.

An administrator may also modify the preferences of a user (including the language of the interface) in the menu: Administration > Users > Users. He merely has to choose a user and toggle on “preferences”.

## 55.2 Store a translation file on the server

To import a file having translations, use this command:

```
./openerp_server.py -i18n-import=file.csv -l LANG
```

where **LANG** is the language of the translation data in the CSV file.

Note that the translation file must be encoded in **UTF8!**

## 55.3 Translate to a new language

**Please keep in mind to use the same translation string for identical sources .** Launchpad Online Translation may give helpful hints.

More information on accelerators on this website: <http://translate.sourceforge.net/wiki/guide/translation/accelerators>

To translate or modify the translation of a language already translated, you have to:

### 55.3.1 1. Export all the sentences to translate in a CSV file

To export this file, use this command:

```
./openerp_server.py -i18n-export=file.csv -l**LANG**
```

where **LANG** is the language to which you want to translate the program.

### 55.3.2 2. Translate the last column of the file

You can make a translation for a language, which has already been translated or for a new one. If you ask for a language already translated, the sentences already translated will be written in the last column.

For example, here are the first lines of a translation file (Dutch):

| type  | name                   | res_id | src         | value             |
|-------|------------------------|--------|-------------|-------------------|
| field | “account.account,code” | 0      | Code        | Code              |
| field | “account.account,name” | 0      | Name        | Name              |
| model | “account.account,name” | 2      | Assets      | Aktiva            |
| model | “account.account,name” | 25     | Results     | Salden            |
| model | “account.account,name” | 61     | Liabilities | Verbindlichkeiten |

### 55.3.3 3. Import this file into OpenERP (as explained in the preceding section)

#### Notes

- You should perform all these tasks on an empty database, so as to avoid over-writing data.

To create a new database (named ‘terp\_test’), use these commands:

```
createdb terp_test –encoding=unicode terp_server.py –database=terp_test –init=all
```

Alternatively, you could also delete your current database with these:

```
dropdb terp createdb terp –encoding=unicode terp_server.py –init=all
```

### 55.3.4 4. Using Launchpad / Rosetta to translate modules and applications

A good starting point is here <https://launchpad.net/openobject>

#### Online

Select the module translation section and enter your translation.

#### Offline

Use this, if you want to translate some 100 terms.

It seems mandatory to follow these steps to successfully complete a translation cycle. (tested on Linux)

1. Download the <po file> from Launchpad
2. Get the message template file <pot file> from bzr branches
  - (a) keep in mind that the <pot file> might not always contain all strings, the <pot files> are updated irregularly.
  - (b) msgmerge <pot file> <po file> -o <new po file>
3. translate <new po file> using poedit, kbabel (KDE)
  - (a) some programs (like kbabel) allow using dictionaries to create rough translations.

- (b) **It is especially useful to create a complete dictionary from existing translations to reuse existing terms related to the application**
- i. In OpenERP load most/all of the modules
  - ii. Load your language
  - iii. export all modules of your language as po file and use this one as dictionary. Depending on context of the module this creates 30-80% exact translations.
- 4. the <new po file> must not contain <fuzzy> comments inserted by kbabel for rough translation**
- (a) grep -v fuzzy <new po file> > <po file>
- 5. check for correct spelling**
- (a) msgfmt <po file> -o <mo file>
- 6. check your translation for correct context**
- (a) import the <po file> (for modules)
  - (b) install the <mo file> and restart the application (for applications)
- 7. adjust the translation Online in OpenERP**
- (a) check context
  - (b) check length of strings
  - (c) export <po file>
- 8. upload <po file> to Launchpad**
- (a) keep in mind that Launchpad / Rosetta uses some tags (not sure which) in the header section of the exported <po file> to recognize the imported <po file> as valid.
  - (b) after some time (hours) you will receive a confirmation E-Mail (success / error)

## 55.4 Using context Dictionary for Translations

The context dictionary is explained in details in section “The Objects - Methods - The context Dictionary”. If an additional language is installed using the Administration menu, the context dictionary will contain an additional key : lang. For example, if you install the French language then select it for the current user, his or her context dictionary will contain the key lang to which will be associated the value *fr\_FR*.



# TECHNICAL MEMENTO

A technical reference memento is available, to be used as a quick reference guide for OpenERP developers, often nicknamed a “cheat sheet”.

-  Technical Memento

The memento is usually updated for each major version of OpenERP, and contains a global overview of OpenERP’s Application Programming Interface, including the declaration of modules, the ORM, the XML syntax, Dynamic views and Workflows. The memento is not an extensive reference, but a way to quickly find out how a certain OpenERP feature is accessed or used. Therefore each topic is only described in a few words, usually with a small example.

The examples in the technical memento all come from the example module `idea`, which allows an organisation to manage the generic *ideas* submitted by its members.

There are 2 versions of the memento. One is suited for printing in A4 landscape mode, with 3 columns of text per page, so that the whole memento is contained in less than 20 mini-pages (columns). The idea is to print and bind these pages as a reference booklet. The second version contains some more details and is formatted in A4 portrait mode, making it easier to read, but larger.

All versions of the technical memento (including previous ones) can be found at this location:  [Technical Memento](#)



# INFORMATION REPOSITORY

The information repository is a semantics tree in which the data that are not the resources are stored. We find in this structure:

1. the values by default
2. **the conditional values;**
  - the state depends on the zip code,
  - the payment method depends of the partner, ...
3. **the reactions to the events client;**
  - click on the invoice menu,
  - print an invoice,
  - action on a partner, ...

The IR has 3 methods;

- add a value in the tree
- delete a value in the tree
- obtain all the values of a selected sheet

## 57.1 Setting Value

The ir\_set tag allows you to insert new values in the “Information Repository”. This tag must contain several *field* tags with *name* and *eval* attributes.

The attributes are those defined by the access methods to the information repository. We must provide it with several attributes: *keys*, *args*, *name*, *value*, *isobject*, *replace*, *meta* and some optional fields.

Example:

```
<ir_set>
    <field name="keys" eval="['action','client_print_multi'],('res_model','account.invoice')"/>
    <field name="args" eval="[]"/>
    <field name="name">Print Invoices</field>
    <field name="value" eval="'ir.actions.report.xml,'+str(10)"/>
    <field name="isobject" eval="True"/>
    <field name="replace" eval="False"/>
</ir_set>
```

## 57.2 IR Methods

```
def ir_set(cr, uid, key, key2, name, models, value, replace=True, isobject=False, meta=None)

def ir_get(cr, uid, key, key2, models, meta=False, context={}, res_id_req=False)

def ir_del(cr, uid, id):
```

### Description of the fields

1. key:
2. key2:
3. name:
4. models:
5. value:
6. isobject:
7. replace: whether or not the action described should override an existing action or be appended to the list of actions.
8. meta:

### Using ir\_set and ir\_get

```
...
```

```
res = ir.ir_set(cr, uid, key, key2, name, models, value, replace, isobject, meta)
```

```
...
```

```
...
```

```
if not report.menu_id:
```

```
    ir.ir_set(cr, uid, 'action', 'client_print_multi', name, [(model, False)], action, False,
```

```
else:
```

```
    ir.ir_set(cr, uid, 'action', 'tree_but_open', 'MenuItem', [('ir.ui.menu', int(m_id))], act,
```

```
...
```

```
...
```

```
res = ir.ir_get(cr, uid, [('default', self._name), ('field', False)], [('user_id', str(uid))])
```

```
...
```

```
account_payable = ir.ir_get(cr, uid, [('meta', 'res.partner'), ('name', 'account.payable')], opt,
```

```
...
```

**Part XVI**

**Community Book**



# COLLABORATION

## 58.1 Launchpad and bazaar

Before you can commit on launchpad, you have to create a login. This login is needed if you intend to commit on openerp-comiter or on your own branch via bazaar. Go to: <https://launchpad.net> → log in / register on top right.

Enter your e-mail address and wait for an e-mail which will guide you through the login creation process.

You can then refer to this link : <https://help.launchpad.net/YourAccount/NewAccount>

Any contributor who is interested to become a committer must show his interest on working for openerp project and ability to do it properly as selection is based on merit. It can be by proposing bug fixes, features requested on our *bug tracker* system. You can even suggest additional modules and/or functionality on our *bug tracker* system.

You can contribute or join OpenERP team, : <https://help.launchpad.net/Teams/Joining>

Contributors are people who want to help the project improve, add functionality and improve stability. Anyone can contribute to the project by reporting bugs, proposing improvements and posting patches.

The community team is available on launchpad: <https://launchpad.net/~openerp-community>

Members of the quality and committer team are automatically members of the community.

Once you have configured your Launchpad account, get Bazaar version control to pull the source from Launchpad.

NoteThib: viré les explications sur comment installer bazaar, je propos de mettre le lien vers le site qui expliquera comment l'installer si besoin est

If you experience problems with Bazaar, please read the *bazaar-faq-link* before asking any questions.

## 58.2 Working with Branch

The combination of Bazaar branch hosting and Launchpad's teams infrastructure gives you a very powerful capability to collaborate on code. Essentially, you can push a branch into a shared space and anyone on that team can then commit to the branch.

This means that you can use Bazaar in the same way that you would use something like SVN, i.e. centrally hosting a branch that many people commit to. You have the added benefit, though, that anyone outside the team can always create their own personal branch of your team branch and, if they choose, upload it back to Launchpad.

This is the official and proposed way to contribute on OpenERP and OpenObject.

### 58.2.1 Quick Summary

To download the latest sources and create your own local branches of OpenERP, do this:

```

mkdir openerp
cd openerp
bzr branch lp:~openerp/openobject-server/trunk server
bzr branch lp:~openerp/openobject-addons/trunk addons
bzr branch lp:~openerp-committer/openobject-addons/trunk-extra-addons addons-extra
bzr branch lp:~openerp-community/openobject-addons/trunk-addons-community addons-community
bzr branch lp:~openerp/openerp-web/trunk web
bzr branch lp:~openerp/openobject-client/trunk client
bzr branch lp:~openerp-community/openobject-doc/6.1 doc

```

This will download all components of openerp (server, client, addons) and create links of modules in addons on your server so that you can use it directly. You can change the bsr\_set.py file to select what you want to download exactly. Now, you can edit the code and commit in your local branch.:

```

EDIT addons/account/account.py
cd addons
bzr ci -m "Testing Modifications"

```

Once your code is good enough and follow the [coding-guidelines-link](#), you can push your branch to launchpad. You may have to create an account on launchpad first, register your public key, and subscribe to the [openerp-community](#) team. Then, you can push your branch. Suppose you want to push your addons:

```

cd addons
bzr push lp:~openerp-community/openobject-addons/YOURLOGIN_YOURBRANCHNAME
bzr bind lp:~openerp-community/openobject-addons/YOURLOGIN_YOURBRANCHNAME

```

After having done that, your branch is public on Launchpad, in the [OpenObject](#) project, and committers can work on it, review it and propose for integration in the official branch. The last line allows you to rebind your branch to the one on launchpad, after having done this, your commit will be applied on launchpad directly (unless you use --local):

```

bzr pull      # Get modifications on your branch from others
EDIT STUFF
bzr ci      # commit your changes on your public branch

```

If your changes fix a public bug on launchpad, you can use this to mark the bug as fixed by your branch:

```
bzr ci --fixes=lp:453123    # Where 453123 is a bug ID
```

Once your branch is mature, mark it as mature in the web interface of launchpad and request for merging in the official release. Your branch will be reviewed by a committer and then the quality team to be merged into the official release.

[Read more about [OpenERP Team](#)]

### 58.2.2 Pushing a new branch

If you want to contribute on OpenERP or OpenObject, here is the proposed method:

- You create a branch on launchpad on the project that interests you. It's important that you create your branch on launchpad and not on your local system so that we can easily merge, share code between projects and centralize future developments.
- You develop your own features or bugfixes in your own branch on launchpad. Don't forget to set the status of your branch (new, experimental, development, mature, ...) so that contributors know what they can and cannot use.
- Once your code is good enough, propose your branch for merging
- Your work will be evaluated by a member of the committers team.
  - If they accept your branch for integration in the official version, they will submit to the quality team that will review and merge in the official branch.

- If the committer team refuses your branch, they will explain why so that you can review the code to better fit the guidelines (problem for future migrations, ...)

The extra-addons branch, that stores all extra modules, is directly accessible to all committers. If you are a committer, you can work directly on this branch and commit your own work. This branch does not require a validation of the quality team. You should put there your special modules for your own customers.

If you want to propose or develop new modules, we suggest creating your own branch in the [openobject-addons project](#) and develop within your branch. You can fill in a bug to request that your modules are integrated in one of the two branches:

- extra-addons : if your module touches a few companies
- addons : if your module will be useful for most of the companies

We invite all our partners and contributors to work in that way so that we can easily integrate and share the work done between the different projects.

After having done that, your branch is public on Launchpad, in the [OpenObject project](#), and committers can work on it, review it and propose for integration in the official branch. The last line allows you to rebind your branch to the one which is on launchpad, after having done this, your commit will be applied on launchpad directly (unless you use --local):

```
bzr pull      # Get modifications on your branch from others
EDIT STUFF
bzr ci      # commit your changes on your public branch
```

If your changes fix a public bug on launchpad, you can use this to mark the bug as fixed by your branch:

```
bzr ci --fixes=lp:453123    # Where 453123 is a bug ID
```

Once your branch is mature, mark it as mature in the web interface of launchpad and request for merging in the official release. Your branch will be reviewed by a committer and then the quality team to be merged in the official release.

### 58.2.3 How to commit Your Work

If you want to contribute on OpenERP or OpenObject, here is the proposed method:

- You create a branch on launchpad on the project that interests you. It's important that you create your branch on launchpad and not on your local system so that we can easily merge, share code between projects and centralize future developments.
- You develop your own features or bugfixes in your own branch on launchpad. Don't forget to set the status of your branch (new, experimental, development, mature, ...) so that contributors know what they can and cannot use.
- Once your code is good enough, propose your branch for merging
- Your work will be evaluated by a member of the committers team.
  - If they accept your branch for integration in the official version, they will submit to the quality team that will review and merge in the official branch.
  - If the committer team refuses your branch, they will explain why so that you can review the code to better fit the guidelines (problem for future migrations, ...)

The [extra-addons branch](#), that stores all extra modules, is directly accessible to all committers. If you are a committer, you can work directly on this branch and commit your own work. This branch does not require validation by the quality team. You should put there your special modules for your own customers.

If you want to propose or develop new modules, we suggest creating your own branch in the [openobject-addons project](#) and develop within your branch. You can fill in a bug to request that your modules are integrated in one of the two branches:

- [extra-addons branch](#) : if your module touches a few companies

- addons : if your module will be useful for most of the companies

We invite all our partners and contributors to work in that way so that we can easily integrate and share the work done between the different projects.

## 58.3 Registration and Configuration

Before you can commit on launchpad, you need to create a login.

Go to: <https://launchpad.net> → log in / register on top right.

You enter your e-mail address and you wait for an e-mail which will guide you through the process needed to create your login.

This login is only needed if you intend to commit on bazaar on openerp-committers or on your own branch.

You can refer to this link : <https://help.launchpad.net/YourAccount/NewAccount>

Any contributor who is interested to become a committer must show his interest on working for openerp project and ability to do it properly as selection is based on merit. It can be by proposing bug fixes, features requested on our *bug tracker* system. You can even suggest additional modules and/or functionality on our *bug tracker* system.

You can contribute or join OpenERP team, : <https://help.launchpad.net/Teams/Joining>

Contributors are people who want to help the project improve, add functionality and improve stability. Anyone can contribute on the project by reporting bugs, proposing some improvement and posting patch.

The community team is available on launchpad: <https://launchpad.net/~openerp-community>

Member of the quality and committer team are automatically members of the community.

### 58.3.1 Installing Bazaar

Get Bazaar version control to pull the source from Launchpad.

To install bazaar on any ubuntu distribution, you can edit /etc/apt/sources.list by

```
sudo gedit /etc/apt/sources.list
```

and put these lines in it:

```
(for ubuntu intrepid 8.10)
deb http://ppa.launchpad.net/bzr/ubuntu intrepid main
deb-src http://ppa.launchpad.net/bzr/ubuntu intrepid main

or (for ubuntu jaunty 9.04)
deb http://ppa.launchpad.net/bzr/ubuntu jaunty main
deb-src http://ppa.launchpad.net/bzr/ubuntu jaunty main

or (for ubuntu karmic 9.10)
deb http://ppa.launchpad.net/bzr/ubuntu karmic main
deb-src http://ppa.launchpad.net/bzr/ubuntu karmic main
```

Here, intrepid, jaunty and karmic are version names of ubuntu, replace it with your ubuntu version.

Then, do the following

```
sudo apt-get install bzr
```

To work correctly, bzr version must be at least 1.3. Check it with the command:

```
bzr --version
```

If you have an older version check this url: <http://bazaar-vcs.org/Download> On debian, in any distribution, the 1.5 version is working, you can get it on this url: [http://backports.org/debian/pool/main/b/bzr/bzr\\_1.5-1~bpo40+1\\_i386.deb](http://backports.org/debian/pool/main/b/bzr/bzr_1.5-1~bpo40+1_i386.deb)

If you experience problems with Bazaar, please read the *bazaar-faq-link* before asking any questions.

## 58.4 Branch

The combination of Bazaar branch hosting and Launchpad's teams infrastructure gives you a very powerful capability to collaborate on code. Essentially, you can push a branch into a shared space and anyone on that team can then commit to the branch.

This means that you can use Bazaar in the same way that you would use something like SVN, i.e. centrally hosting a branch that many people commit to. You have the added benefit, though, that anyone outside the team can always create their own personal branch of your team branch and, if they choose, upload it back to Launchpad.

This is the official and proposed way to contribute on OpenERP and OpenObject.

## 58.5 How to commit

If you want to contribute on OpenERP or OpenObject, here is the proposed method:

- You create a branch on launchpad on the project that interests you. It's important that you create your branch on launchpad and not on your local system so that we can easily merge, share code between projects and centralize future developments.
- You develop your own features or bugfixes in your own branch on launchpad. Don't forget to set the status of your branch (new, experimental, development, mature, ...) so that contributors know what they can and cannot use.
- Once your code is good enough, propose your branch for merging
- Your work will be evaluated by a member of the committers team.
  - If they accept your branch for integration in the official version, they will submit to the quality team that will review and merge in the official branch.
  - If the committer team refuses your branch, they will explain why so that you can review the code to better fit the guidelines (problem for future migrations, ...)

The extra-addons branch, that stores all extra modules, is directly accessible to all committers. If you are a committer, you can work directly on this branch and commit your own work. This branch does not require a validation of the quality team. You should put there your special modules for your own customers.

If you want to propose or develop new modules, we suggest creating your own branch in the [openobject-addons project](#) and develop within your branch. You can fill in a bug to request that your modules are integrated in one of the two branches:

- extra-addons : if your module touches a few companies
- addons : if your module will be useful for most of the companies

We invite all our partners and contributors to work in that way so that we can easily integrate and share the work done between the different projects.

After having done that, your branch is public on Launchpad, in the [OpenObject project](#), and committers can work on it, review it and propose for integration in the official branch. The last line allows you to rebind your branch to the one which is on launchpad, after having done this, your commit will be applied on launchpad directly (unless you use `--local`):

```
bzr pull      # Get modifications on your branch from others  
EDIT STUFF  
bzr ci      # commit your changes on your public branch
```

If your changes fix a public bug on launchpad, you can use this to mark the bug as fixed by your branch:

```
bzr ci --fixes=lp:453123    # Where 453123 is a bug ID
```

Once your branch is mature, mark it as mature in the web interface of launchpad and request for merging in the official release. Your branch will be reviewed by a committer and then the quality team to be merged in the official release.

## 58.6 Answer and bug tracking and management

We use launchpad on the openobject project to track all bugs and features request related to openerp and openobject. the bug tracker is available here:

- Bug Tracker : <https://bugs.launchpad.net/openobject>
- Ideas Tracker : <https://blueprints.launchpad.net/openobject>
- FAQ Manager : <https://answers.launchpad.net/openobject>

Every contributor can report bug and propose bugfixes for the bugs. The status of the bug is set according to the correction.

When a particular branch fixes the bug, a committer (member of the [Committer Team](#)) can set the status to “Fix Committed”. Only committers have the right to change the status to “Fix Committed.”, after they validated the proposed patch or branch that fixes the bug.

The [Quality Team](#) have a look every day to bugs in the status “Fix Committed”. They check the quality of the code and merge in the official branch if it’s OK. To limit the work of the quality team, it’s important that only committers can set the bug in the status “Fix Committed”. Once quality team finish merging, they change the status to “Fix Released”.

## 58.7 Translation

Translations are managed by the [Launchpad Web interface](#). Here, you’ll find the list of translatable projects.

Please read the [FAQ](#) before asking questions.

## 58.8 Blueprints

Blueprint is a lightweight way to manage releases of your software and to track the progress of features and ideas, from initial concept to implementation. Using Blueprint, you can encourage contributions from right across your project’s community, while targeting the best ideas to future releases.

Launchpad Blueprint helps you to plan future release with two tools:

- milestones: points in time, such as a future release or development sprint
- series goals: a statement of intention to work on the blueprint for a particular series.

Although only drivers can target blueprints to milestones and set them as series goals, anyone can propose a blueprint as a series goal. As a driver or owner, you can review proposed goals by following the Set goals link on your project’s Blueprint overview page.

By following the [Subscribe yourself](#) link on a blueprint page, you can ask Launchpad to send you email notification of any changes to the blueprint. In most cases, you'll receive notification only of changes made to the blueprint itself in Launchpad and not to any further information, such as in an external wiki.

However, if the wiki software supports email change notifications, Launchpad can even notify you of changes to the wiki page.

If you're a blueprint owner and want Launchpad to know about updates to the related wiki page, ask the wiki admin how to send email notifications. Notifications should go to [notifications@specs.launchpad.net](mailto:notifications@specs.launchpad.net).

The Blueprints for OpenERP are listed here:

- <https://blueprints.launchpad.net/openerp>
- <https://blueprints.launchpad.net/~openerp-committer>



## **Part XVII**

### **Remainder of old TOC**



# OPENERP WEB CLIENT V6.0

## 59.1 OpenERP Web v6.0

### 59.1.1 Introduction

#### Web Client – A software application that is launched via a web browser.

With the launch of OpenERP v6, a new web client has been designed and developed, which provides a more professional appearance than OpenERP v5.

We migrated the web client to CherryPy3 dropping TurboGears completely and migrated kid templates to faster Mako templates, a major step towards making the Web Client much faster and easier to deploy.

All the Kid templates were converted to faster Mako templates, i18n/l18n features have been partially reimplemented using Python Babel, CherryPy2 (TG is built on top of CP2) was replaced with CherryPy3, the latest, much better version of CherryPy Server.

This greatly reduces the pain of getting started with and deploying of OpenERP Web client.

Now the number of third party dependencies are reduced to 3-4 pure Python libraries which you can install within the local lib dir with the help of *populate.sh* script, found under the same lib directory.

- Just get the source from Launchpad, run the **populate.sh** and launch the web client...

The initial test results are very impressive.

We have seen 3-5 times speed improvement.

### 59.1.2 Against OpenERP Web 5.0

Here are the benchmark results of the latest stable 6.0 version against the stable 5.0 branch which is running over TurboGears.

The benchmark test used Apache Benchmark Tool against larger Customer Invoice Form view.

| Result of OpenERP Web 5.0 (TurboGears + Kid)                                                                    | Result of OpenERP Web 6.0 (CherryPy3 + Mako)                                                                    |
|-----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| This is ApacheBench, Version 2.3 <\$Revision: 655654 \$>                                                        | This is ApacheBench, Version 2.3 <\$Revision: 655654 \$>                                                        |
| Copyright 1996 Adam Twiss, Zeus Technology Ltd, <a href="http://www.zeustech.net/">http://www.zeustech.net/</a> | Copyright 1996 Adam Twiss, Zeus Technology Ltd, <a href="http://www.zeustech.net/">http://www.zeustech.net/</a> |
| Licensed to The Apache Software Foundation, <a href="http://www.apache.org/">http://www.apache.org/</a>         | Licensed to The Apache Software Foundation, <a href="http://www.apache.org/">http://www.apache.org/</a>         |
| Benchmarking localhost (be patient).....done                                                                    | Benchmarking localhost (be patient).....done                                                                    |
| Server Software: CherryPy/2.3.0                                                                                 | Server Software: CherryPy/3.1.2                                                                                 |
| Server Hostname: localhost                                                                                      | Server Hostname: localhost                                                                                      |
| Server Port: 8081                                                                                               | Server Port: 8080                                                                                               |
| Document Path: /form/edit?model=account.invoice&id=1                                                            | Document Path: /form/edit?model=account.invoice&id=1                                                            |
| Document Length: 79965 bytes                                                                                    | Document Length: 90394 bytes                                                                                    |
| Concurrency Level: 1                                                                                            | Concurrency Level: 1                                                                                            |

**Table 59.1 – continued from previous page**

| <b>Result of OpenERP Web 5.0 (TurboGears + Kid)</b>                    | <b>Result of OpenERP Web 6.0 (CherryPy3 + Mako)</b>                   |
|------------------------------------------------------------------------|-----------------------------------------------------------------------|
| Time taken for tests: 166.323 seconds                                  | Time taken for tests: 42.054 seconds                                  |
| Complete requests: 100                                                 | Complete requests: 100                                                |
| Failed requests: 0                                                     | Failed requests: 0                                                    |
| Write errors: 0                                                        | Write errors: 0                                                       |
| Total transferred: 8022000 bytes                                       | Total transferred: 9063400 bytes                                      |
| HTML transferred: 7996500 bytes                                        | HTML transferred: 9039400 bytes                                       |
| Requests per second: 0.60 [#/sec] (mean)                               | Requests per second: 2.38 [#/sec] (mean)                              |
| Time per request: 1663.228 [ms] (mean, across all concurrent requests) | Time per request: 420.543 [ms] (mean, across all concurrent requests) |
| Transfer rate: 47.10 [Kbytes/sec] received                             | Transfer rate: 210.47 [Kbytes/sec] received                           |
| <b>Connection Times (ms)</b>                                           | <b>Connection Times (ms)</b>                                          |
| min mean[+/-sd] median max                                             | min mean[+/-sd] median max                                            |
| Connect: 0 0 0.0 0                                                     | Connect: 0 0 0.0 0                                                    |
| Processing: 1556 1663 71.3 1663 1856                                   | Processing: 1556 1663 71.3 1663 1856                                  |
| Waiting: 1555 1662 71.3 1662 1855                                      | Waiting: 381 420 27.7 415 522                                         |
| Total: 1556 1663 71.3 1663 1856                                        | Total: 382 420 27.7 416 523                                           |
| <b>Percentage of the requests served within a certain time (ms)</b>    | <b>Percentage of the requests served within a certain time (ms)</b>   |
| 50% 1663                                                               | 50% 416                                                               |
| 66% 1681                                                               | 66% 418                                                               |
| 75% 1695                                                               | 75% 420                                                               |
| 80% 1715                                                               | 80% 424                                                               |
| 90% 1775                                                               | 90% 436                                                               |
| 95% 1801                                                               | 95% 512                                                               |
| 98% 1829                                                               | 98% 520                                                               |
| 99% 1856                                                               | 99% 523                                                               |
| 100% 1856 (longest request)                                            | 100% 523 (longest request)                                            |

### 59.1.3 Conclusion

- You can see significant performance boost in second test result.
- We observed 3-5 times speedup.
- There is still room to improve the performance further.
- Like reducing RPC calls, catching results of some computationally heavy functions.

## 59.2 What is CherryPy ?

CherryPy is a pythonic, object-oriented HTTP framework.

CherryPy allows developers to build web applications in much the same way they would build any other object-oriented Python program. This results in smaller source code developed in less time.

```
import cherrypy
class HelloWorld:
    def index(self):
        return "Hello World!"
    index.exposed = True
cherrypy.quickstart(HelloWorld())
```

**Start the application at the command prompt(after navigating to its folder): python hello.py**

Direct your browser to <http://localhost:8080>

**The rendering: Hello World!**

ctrl+c in command window to terminate the application

Statement **import cherrypy** imports the main CherryPy module.

An instance of class **HelloWorld** is the object that will be **published**.

Method **index()** is called when the root URL for the site(e.g., <http://localhost:8080>) is requested, This method returns the **contents** of the Web page(the ‘Hello World!’ string)

Statement **index.exposed = True** tells CherryPy that method **index()** will be exposed

- Only exposed methods can be called to answer a request
- Lets the user to select which methods of an object are Web accessible
- Can also place the decoration **@cherrypy.expose** immediately before the method:

```
@cherrypy.expose  
def index(self):  
    return "Hello World!"
```

Statement, **cherrypy.quickstart(HelloWorld())**

**publishes** an instance of the HelloWorld class

- And it starts the embedded webserver
- Runs until explicitly interrupted(ctrl+c)

When the application is executed, the CherryPy server is started with the default configuration

- Listening on **localhost** at port **8080**
- Defaults overriden by using a configuration file or dictionary
  - **cherrypy.config.update({'server.socket\_port':8010})**
  - Now it will run on port 8010.

Webserver receives the request for URL <http://localhost:8080>

- Searches for the best method to handle the request,starting from the **HelloWorld** instance
- CherryPy calls **HelloWorld().index()**
- Result of the call is sent back to the browser as the content of the index page for the website

### 59.2.1 Cherrypy Application Facts

Your CherryPy powered web applications are in fact stand-alone Python applications embedding their own multi-threaded web server. You can deploy them anywhere you can run Python applications. Apache is not required, but it's possible to run a CherryPy application behind it (or lighttpd, or IIS). CherryPy applications run on Windows, Linux, Mac OS X and any other platform supporting Python.

Beyond this functionality, CherryPy pretty much stays out of your way. You are free to use any kind of templating, data access etc. technology you want. CherryPy can also handle sessions, static files, cookies, file uploads and everything you would expect from a decent web framework.

### 59.2.2 Features

- A **fast**, HTTP/1.1-compliant, WSGI thread-pooled webserver. Typically, CherryPy itself takes only 1-2ms per page!
- Support for any other WSGI-enabled webserver or adapter, including Apache, IIS, lighttpd, mod\_python, FastCGI, SCGI, and mod\_wsgi
- Easy to run multiple HTTP servers (e.g. on multiple ports) at once

- A powerful configuration system for developers and deployers alike
- A flexible plugin system
- Built-in tools for caching, encoding, sessions, authorization, static content, and many more
- A native mod\_python adapter
- A complete test suite
- Swappable and customisable... everything.
- Built-in profiling, coverage, and testing support.

**Consider these examples (root is conceptual, referring to the root of the document tree), root = HelloWorld()**

```
root.onepage = OnePage()
```

```
root.otherpage = OtherPage()
```

URL `http://localhost:8080/onepage` points at the 1st object,

URL `http://localhost:8080/otherpage` points at the 2nd.

**Consider, root.some = Page()**

```
root.some.page = Page()
```

URL `http://localhost:8080/some/page` is mapped to the `root.some.page` object. If this object is exposed (or its `index` method is), it's called for that URL

In our HelloWorld example, adding the `http://.../onepage` to OnePage() mapping could be done as:

```
class OnePage():
    def index(self):
        return "one page!"
    index.exposed = True
class HelloWorld(object):
    onepage = OnePage()
    def index(self):
        return "hello world"
    index.exposed = True
cherrypy.quickstart(HelloWorld())
```

In the address bar of the browser, put `http://localhost:8080/onepage`

### 59.2.3 The Index Method

- Method `index()`, like the `index.html` file, is the default page for any internal node in the object tree
- Can take additional keyword arguments, mapped to the form variables as sent via its GET or POST methods
- It's only called for a full match on the URL

### 59.2.4 Calling Other Methods

CherryPy can also directly call methods in the published objects if it receives a URL that is directly mapped to them—e.g.,

```
class HelloWorld():
    def index(self):
        return "Hello World!"
    index.exposed = True

@cherrypy.expose
def test(self):
```

```
    return "Test Controller"
cherrypy.quickstart(HelloWorld())
```

Then request `http://localhost:8080/test`

When CherryPy receives a request for the `/test` URL, it calls the `test()` function.

- It can be a plain function, or a method of any object—any callable will do.

If CherryPy finds a full match and the last object in the match is a **callable**.

- A method, function, or any other Python object that supports the `__call__` method and the callable doesn't contain a valid `index()` method.

Then the object itself is called.

These rules are needed because classes in Python are callables (for producing instances).

CherryPy supports both the GET and POST method for forms.

## 59.3 Mako Template

Mako is a template library written in Python.

It provides a familiar, non-XML syntax which compiles into Python modules for maximum performance.

Mako's syntax and API borrows from the best ideas of many others, including Django templates, Cheetah, Myghty, and Genshi.

Conceptually, Mako is an embedded Python (i.e. Python Server Page) language, which refines the familiar ideas of componentized layout and inheritance to produce one of the most straightforward and flexible models available, while also maintaining close ties to Python calling and scoping semantics.

```
<%inherit file="base.html"/>
<%
    rows = [[v for v in range(0,10)] for row in range(0,10)]
%>







<%def name="makerow(row)">
    <tr>
        % for name in row:
            <td>${name}</td>
        % endfor
    </tr>
</%def>
```

### 59.3.1 Features

Super-simple API. For basic usage, just one class, `Template` is needed:

```
from mako.template import Template
print Template("hello ${data}!").render(data="world")
```

For filesystem management and template caching, add the `TemplateLookup` class.

Insanely Fast. An included bench suite, adapted from a suite included with Genshi, has these results for a simple three-sectioned layout:

Mako: 1.10 ms

Kid: 14.54 ms

- **Standard template features**

- control structures constructed from real Python code (i.e. loops, conditionals)
- straight Python blocks, inline or at the module-level

- **Callable blocks**

- can access variables from their enclosing scope as well as the template's request context
- can be nested arbitrarily
- can specify regular Python argument signatures
- outer-level callable blocks can be called by other templates or controller code (i.e. “method call”)
- calls to functions can define any number of sub-blocks of content which are accessible to the called function (i.e. “component-call-with-content”). This is the basis for nestable custom tags.

- **Inheritance**

- supports “multi-zoned” inheritance - define any number of areas in the base template to be overridden.
- supports “chaining” style inheritance - call next.body() to call the “inner” content.
- the full inheritance hierarchy is navigable in both directions (i.e. parent and child) from anywhere in the chain.
- inheritance is dynamic ! Specify a function instead of a filename to calculate inheritance on the fly for every request.

## 59.3.2 Examples

- **Basic Usage**

```
from mako.template import Template

mytemplate = Template("hello world!")
print mytemplate.render()
```

The text argument to **Template** is **compiled** into a Python module representation.

This module contains a function called `render_body()`, which produces the output of the template.

When `mytemplate.render()` is called, Mako sets up a runtime environment for the template and calls the `render_body()` function, capturing the output into a buffer and returning its string contents.

The code inside the `render_body()` function has access to a namespace of variables. You can specify these variables by sending them as additional keyword arguments to the `render()` method:

```
from mako.template import Template

mytemplate = Template("hello, ${name}!")
print mytemplate.render(name="openerp")
```

- **Using File-based Templates**

A **Tempalte** can also load its template source code from a file, using the `filename` keyword argument:

```
from mako.template import Template

mytemplate = Template(filename='/test.html')
print mytemplate.render()
```

- **Using TemplateLookup**

All of the examples thus far have dealt with the usage of a single **Template** object.

If the code within those templates tries to locate another template resource, it will need some way to find them, using simple URI strings.

For this need, the resolution of other templates from within a template is accomplished by the **TemplateLookup** class.

This class is constructed given a list of directories in which to search for templates, as well as keyword arguments that will be passed to the **Template** objects it creates:

```
from mako.template import Template
from mako.lookup import TemplateLookup

mylookup = TemplateLookup(directories=[''])
mytemplate = Template('<% include file="header.txt"/> Hello!', lookup=mylookup)
```

Above, we created a textual template which includes the file “header.txt”.

In order for it to have somewhere to look for “header.txt”, we passed a **TemplateLookup** object to it, which will search in the current directory for the file “header.txt”.

### 59.3.3 Syntax

- **Expression Substitution**

The simplest expression is just a variable substitution.

The syntax for this is the \${ } construct, which is inspired by Perl, Genshi, JSP EL, and others:

```
${x}
${5%5}
${7*2}
${pow(x,2) + pow(y,2)}
```

- **Controller Structures**

- Conditionals(i.e if/else)
- loops(for and while)
- as well as try/except

control structures are written using the % marker followed by a regular Python control expression, and are “closed” by using another % marker with the tag “end<name>“, where “<name>” is the keyword of the expression:

```
% if user_name == 'openerp':
    valid user
% endif

% if a > 1:
    a is positive number
% elif a == 0:
    a is 0
% else:
    a is negative number
% endif

<table>
% for a in [1,2,3,4,5]:
    <tr>
        <td>
            ${a}
        </td>
```

```

        </tr>
% endfor
</table>
```

- **Python Blocks**

Any arbitrary block of python can be dropped in using the `<% %>` tags:

```

<%
    a = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
    b = a.values()
%>
% for x in b:
    ${x}
% endfor
```

- **Module-level Blocks**

A variant on `<% %>` is the module-level code block, denoted by `<%! %>`.

Code within these tags is executed at the module level of the template, and not within the rendering function of the template.

```

<%!
    import cherrypy
    def get_user_from_session():
        return cherrypy.session['current_user']
%>
```

Therefore, this code does not have access to the template's context and is only executed when the template is loaded into memory (which can be only once per application, or more, depending on the runtime environment).

- **Mako Tags**

### `<%page>`

This tag defines general characteristics of the template, including caching arguments, and optional lists of arguments which the template expects when invoked.

Also defines caching characteristics.

```
<%page args="x, y, z='default' "/>
<%page cached="True" cache_type="memory"/>
```

### `<%include>`

just accepts a file argument and calls in the rendered result of that file:

Also accepts arguments which are available as `<%page>` arguments in the receiving template:

```
<%include file="header.mako"/>
    Welcome to OpenERP
<%include file="footer.mako"/>

<%include file="toolbar.html" args="current_section='members', username='ed' "/>
```

### `<%inherit>`

Inherit allows templates to arrange themselves in inheritance chains.

When using the `%inherit` tag, control is passed to the topmost inherited template first, which then decides how to handle calling areas of content from its inheriting templates.

```
<%inherit file="index.mako"/>
```

### `<%def>`

The %def tag defines a Python function which contains a set of content, that can be called at some other point in the template.

The %def tag is a lot more powerful than a plain Python def, as the Mako compiler provides many extra services with %def that you wouldn't normally have, such as the ability to export defs as template "methods", automatic propagation of the current Context, buffering/filtering/caching flags, and def calls with content, which enable packages of defs to be sent as arguments to other def calls (not as hard as it sounds).

```
<%def name="my_function(x)">
    this is function ${x}
<%def>
```

### **<%namespace>**

%namespace is Mako's equivalent of Python's import statement.

It allows access to all the rendering functions and metadata of other template files, plain Python modules, as well as locally defined "packages" of functions.

```
<%namespace file="test.mako" import="*"/>
```

### **<%doc>**

handles multiline comments:

```
<%doc>
    Multi line comments
    Using doc tag
</%doc>
```

For More Details visit the documentation: <http://www.makotemplates.org/docs/index.html>



# OTHER TOPICS

## 60.1 RAD Tools

### 60.1.1 DIA

The `uml_dia` module helps to develop new modules after an UML description using the DIA tool (<http://www.gnome.org/projects/dia>).

It's not a typical module in the sense that you don't have to install it on the server as another module. The contents of the module are just a python script for dia (`codegen_openerp.py`), a test dia diagram and the module generated by the test.

The module is located in the `extra-addons` branch: <https://code.launchpad.net/openobject-addons>

To use the module you need to make `codegen_openerp.py` accessible from dia, usually in your `/usr/share/dia/python` directory and make sure that it gets loaded once. To do it, just open dia and open a **Python Console** from the **Dialog Menu**, and type there "import codegen\_openerp". If everything goes alright you will have a new option in your "Export..." dialog named "PyDia Code Generation (OpenERP)" that will create a zip module from your UML diagram.

To install win Dia in windows, first install Python-2.3.5, then when you install Dia, you will have an option to install the python plug-in. After this, put the `codegen_openerp.py` file in **C:\Program Files\Dia** and you will have the export function in Dia.

For further guidance to install Dia in Windows you can refer to this link (<http://openerpdev.blogspot.com/2009/11/rad-with-openerp.html>)

If you find that the zip file is corrupt, use DiskInternals ZipRepair utility to repair the zip file before you'll be able to import it - make sure the zip file you import has the same name you saved as.

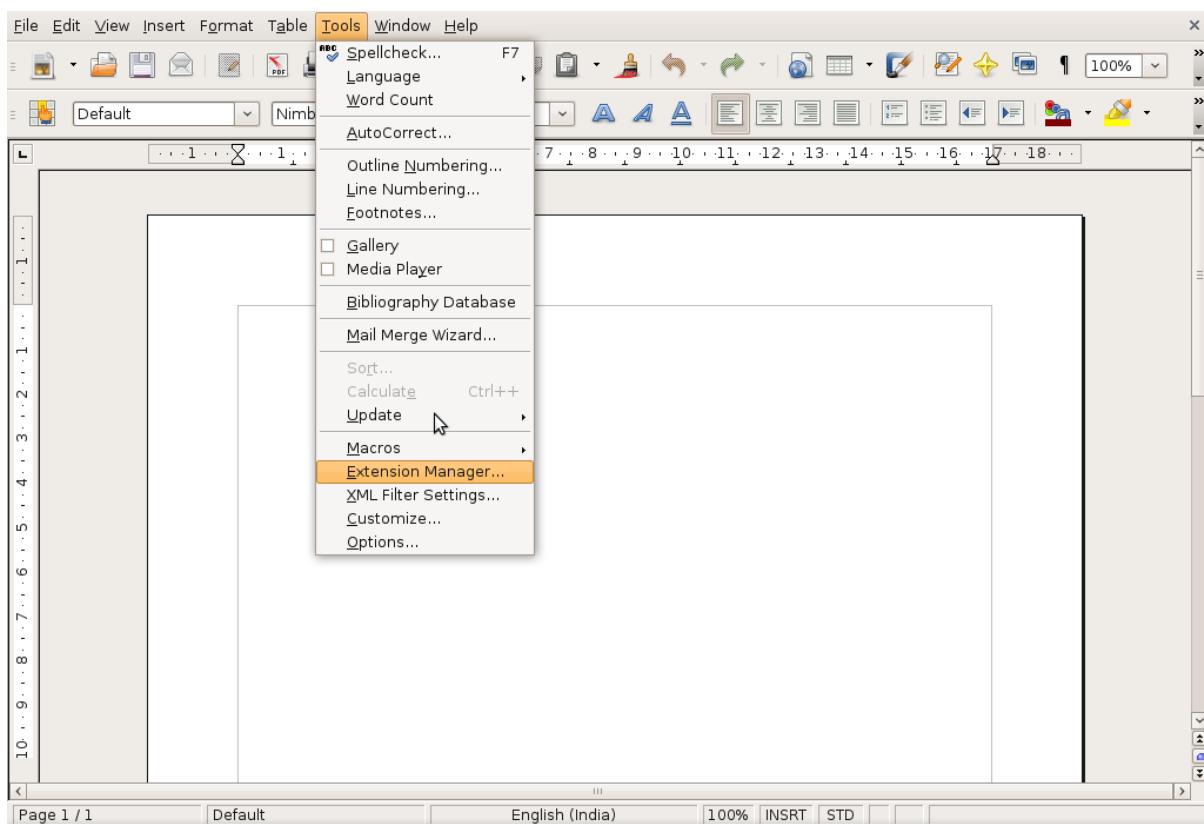
### 60.1.2 Open Office Report Designer

#### Installation

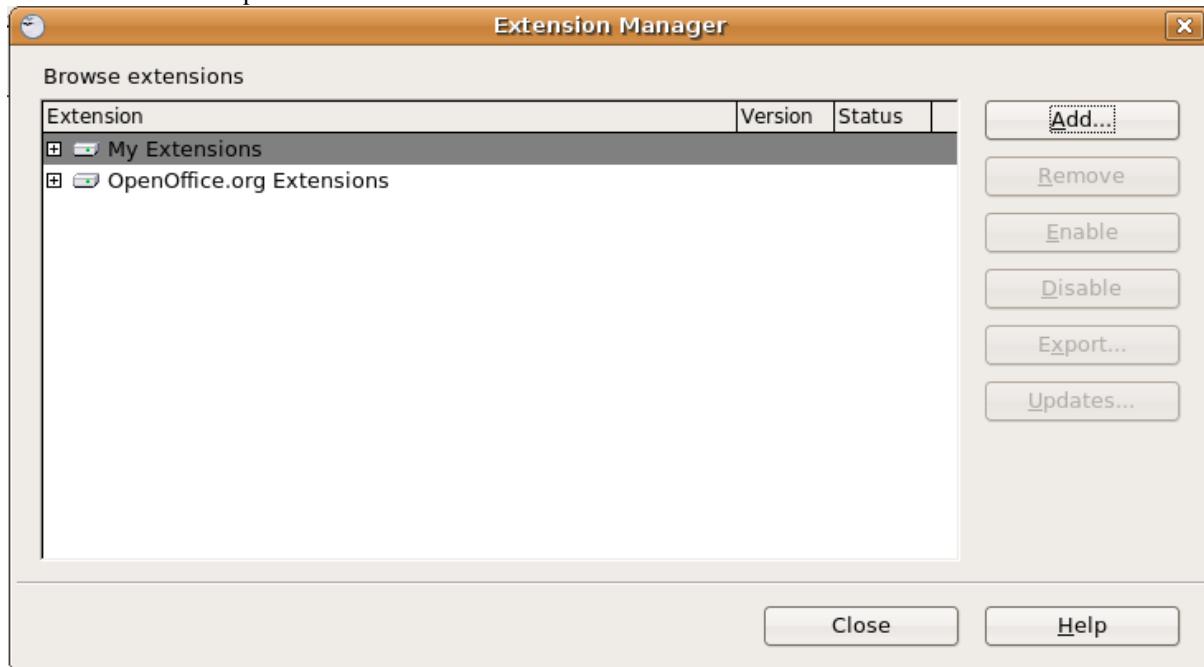
Openoffice.org Report Designer plugin is very easy to install and use. The plugin is a bundle of two files: `openoficereport.zip` and `Makefile`. We have installation procedure.

- Install using Extension Manager in Openoffice.org Writer

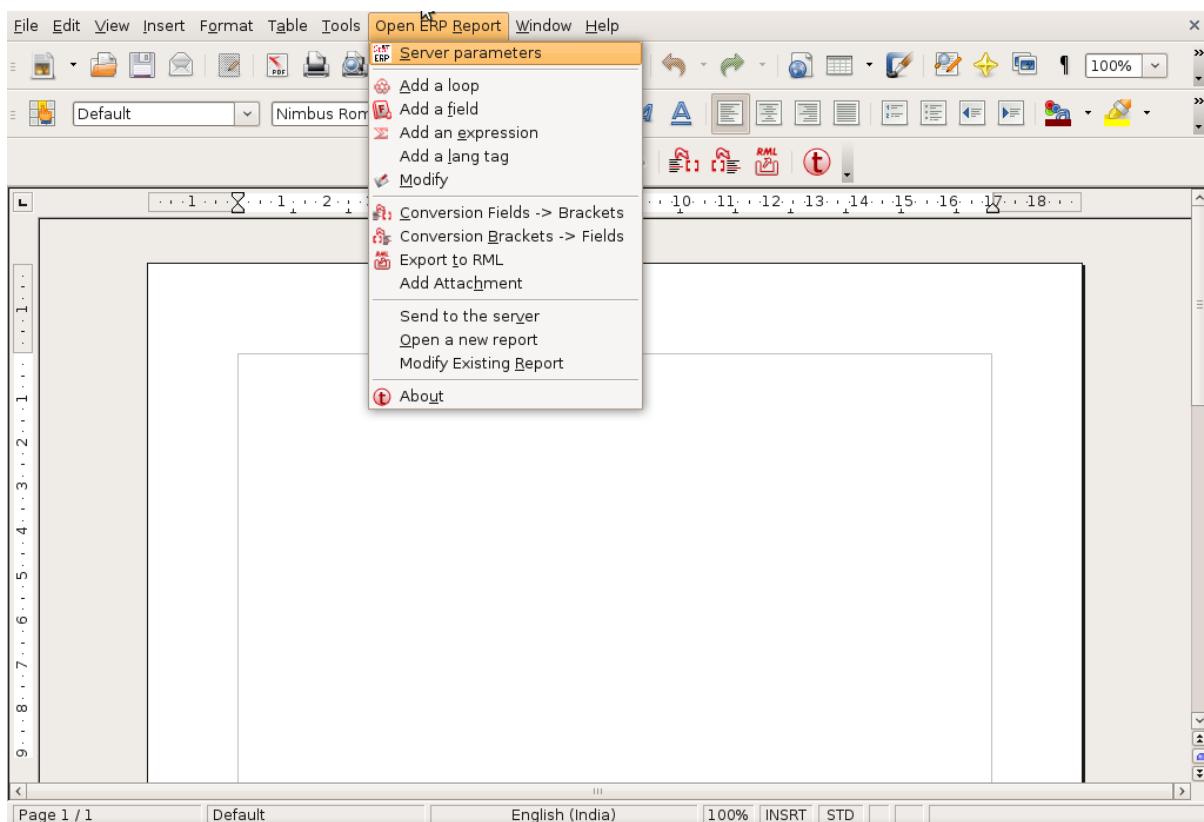
Installing by Extension Manager is interactive installation procedure, for installation you have to use Tools -> Extension Manager provided by Openoffice.org Writer.



In Extension Manager you have Add button, by clicking add button you will get opendialog box from which you will have to select ".zip" file.



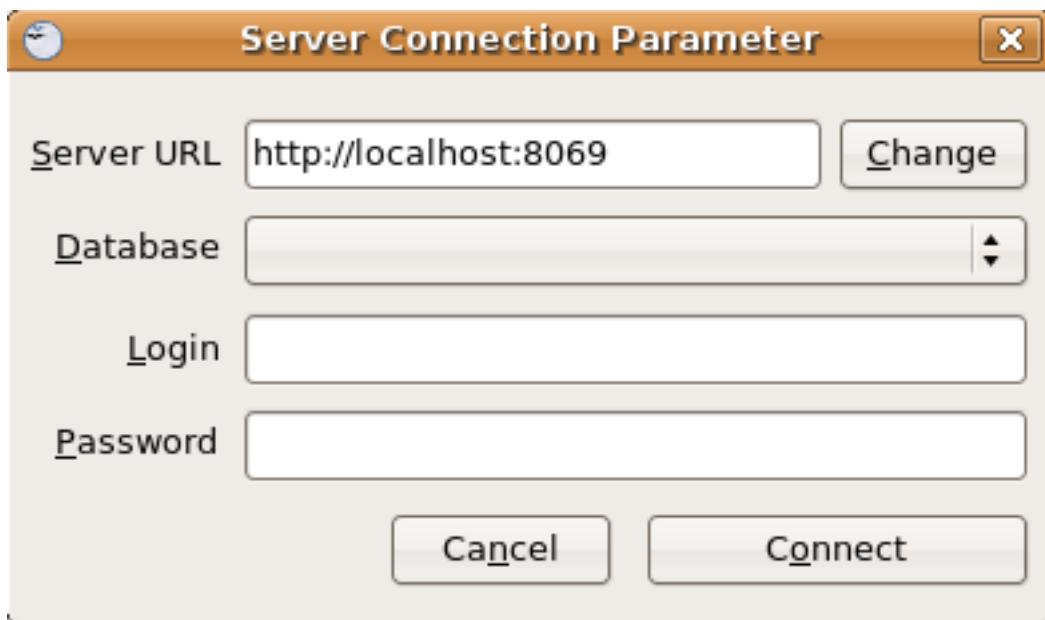
After installation you will get OpenERP Report Menu and its Toolbar in Openoffice.org Writer.



## Server Parameters

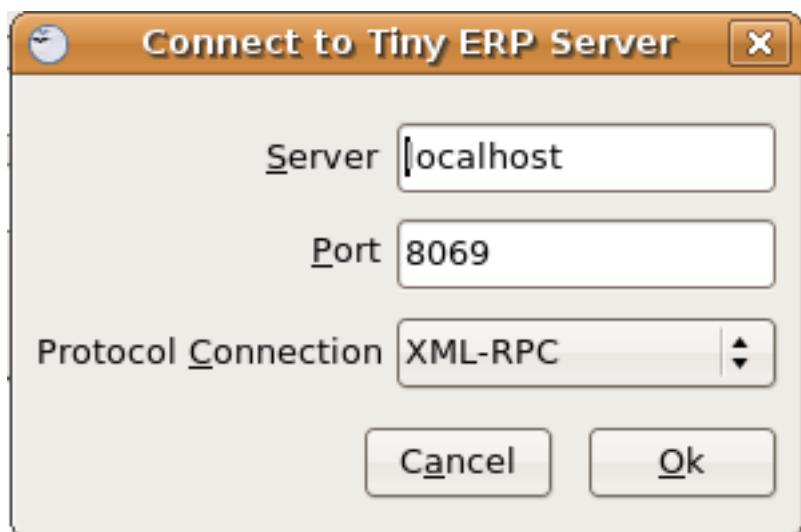
This Functionality is used to connect with OpenERP Server with different login mode. You can access that functionality by clicking on OpenERP button on toolbar or just go to Open Report > Server Parameters.

The screen will look like



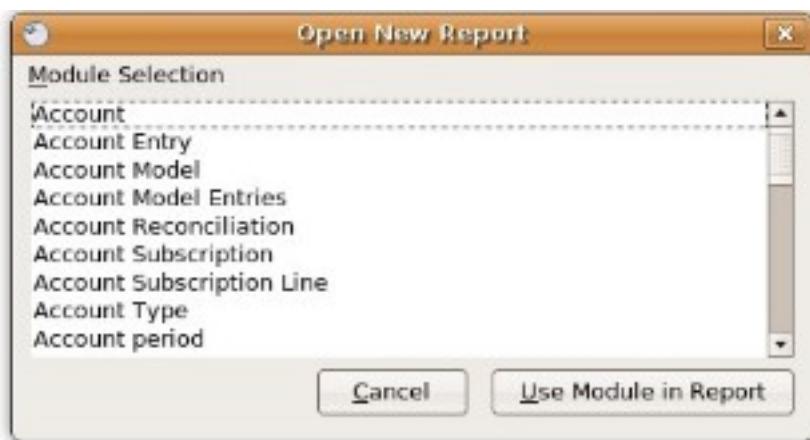
After giving proper Server URL you can select Database available in selected server and login as the given user.

There is one additional functionality of changing server parameters by clicking on 'Change' button



### Open a New report

You can open this dialog box by clicking on Open Report -> Open a new report



By using above window you can select module for which you want to create report. This is first process to create new report, so you have to select module. By clicking on 'Use Module in Report' selected module will be used to create report.

### Add a loop

This functionality is used to create repeatIn statement in Open Report. You see this dialog box by clicking on Open Report -> Add a loop from menubar or just on this button from toolbar. \* The loop can be put into a table (the lines will then be repeated) or into an OpenOffice.org section.

Objects to loop on : List of Partner

Field to loop on : objects

Variable name : partner

Displayed name : |-.partner.-|

Cancel      ok

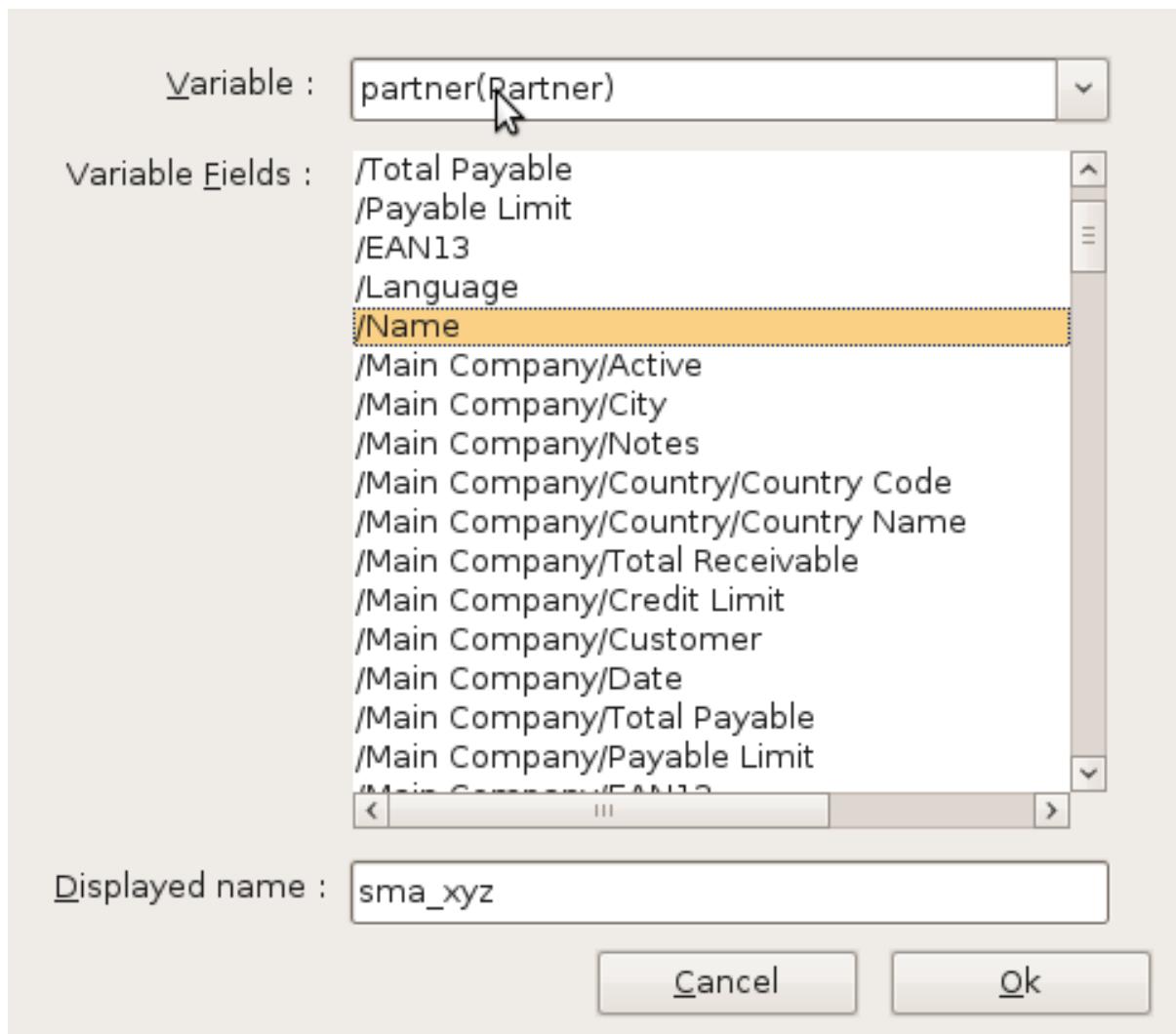
After click on 'ok' button you will get repeatIn object just like displayed below.



Above report statement is written in Input Field a special functionality available in Openoffice.org. In which main statement available in background and it will display some English type of name as here displayed <partner>.

### Add a field

This functionality is used to create field statement in OpenReport. You see this dialog box by clicking on Open Report > Add a field from menubar or just this button from toolbar. Also select the multiple field .



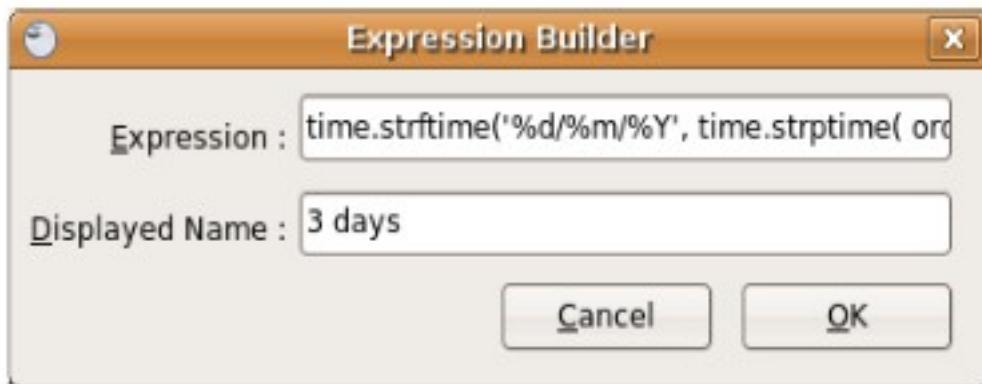
### Add an expression

This functionality is used to create expression which we can not add by using fields in Open Report. You see this dialog box by clicking on OpenReport - >Add an expression from menubar.

- Using the Expression button you can enter expressions in the Python language. These expressions can use all of the object's fields for their calculations. For example if you make a report on an order you can use the following expression:

```
'%.2f' % (amount_total * 0.9,)
```

In this example, amount\_total is a field from the order object. The result will be 90% of the total of the order, formatted to two decimal places.



After click on 'ok' button you will get expression object just like displayed below

Date	Qty
3 days	2.00 Unit

### Add lang tag

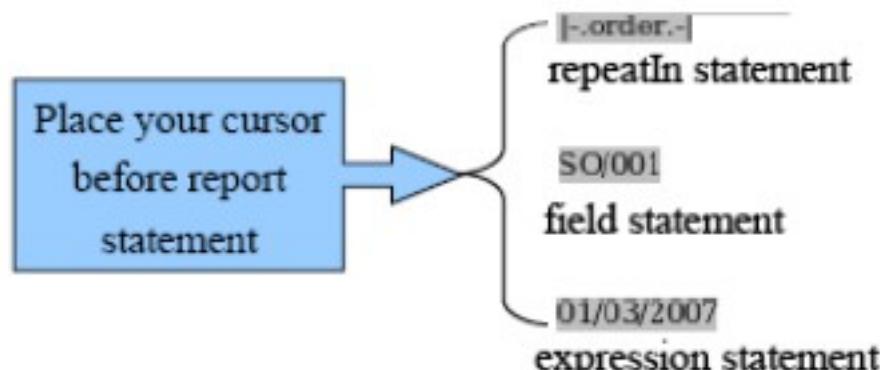
As OpenERP can be used in several languages, reports must be translatable. But in a report, everything mustn't be translated: only the actual text and not the formatting codes. A field will be processed by the translation system if the XML tag which surrounds it (whatever it is) has a `t="1"` attribute. The server will translate all the fields with such attributes in the report generation process. It creates the set Lang tag.

## Modify

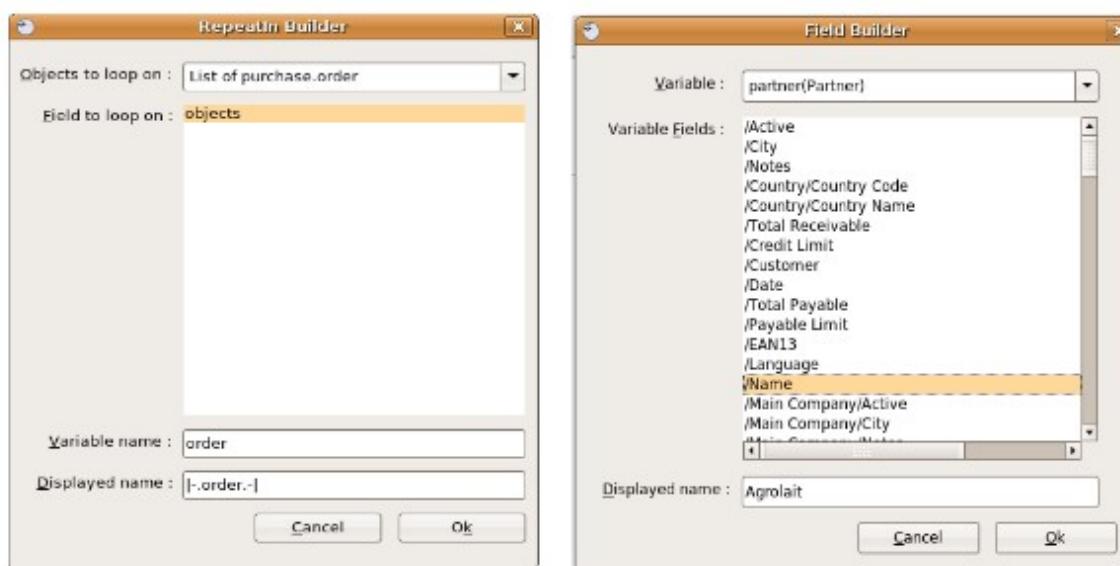
This functionality is used to modify existing repeatIn, Fields, or Expression in Open Report. \* This functionality will work with cursor you have to place your current cursor before the report statement and it will open dialog box after detecting that statement is either Expression, RepeatIn or Fields.

- You see this dialog by clicking on Open Report -> Modify from menubar.

## Modify RepeatIn



After placing your cursor at the beginning of the report statement press modify button from toolbar or click on Open Report -> Modify . It will detect the type of that statement weather its is Field, Expression or RepeatIn and generate window accordingly as displayed below give following window.



## Conversion Fields > Brackets

The purpose of this functionality is mapping old (use bracket for writing report statement) and new (use input filed for writing report statement). So whenever you want to convert your new report statement to old fashion then you can use this method. if you want to access this functionality you can click on OpenReport > Conversion Fields > Bracket from menubar.

Full Screen

1 · · · 1 · · 2 · · 3 · · 4 · · 5 · · 6 · · 7 · · 8 · · 9 · · 10 · · 11 · · 12 · · 13 · · 14 · · 15 · · 16 · · 17 · · 18 · · 19 ·

```

[[ repeatIn(objects,'o') ]]
[[ setLang(o.partner_id.lang) ][[ setLang(o.partner_id.lang) ]]

[[ o.partner_id.title or ""] [[ o.partner_id.name ]]
[[ o.address_invoice_id.title or ""] [[ o.address_invoice_id.name ]]
[[ o.address_invoice_id.street ]]
[[ o.address_invoice_id.street2 or ""]]
[[ o.address_invoice_id.zip or ""] [[ o.address_invoice_id.city or
    ""]]
[[ o.address_invoice_id.state_id and
    o.address_invoice_id.state_id.name or ""]]
[[ o.address_invoice_id.country_id and
    o.address_invoice_id.country_id.name or ""]]

Tel : [[ o.address_invoice_id.phone or removeParentNode(para)
]]
Fax : [[ o.address_invoice_id.fax or removeParentNode('para') ]]
VAT : [[ o.partner_id.vat or removeParentNode('para') ]]

Invoice [[ ((o.type == 'out_invoice' and (o.state == 'open' or o.state == 'paid')) or
removeParentNode('para')) and ""]][[ o.number ]]

PRO-FORMA [[ ((o.type == 'out_invoice' and o.state == 'proforma') or
removeParentNode('para')) and ""]]

PRO-FORMA [[ ((o.type == 'out_invoice' and o.state == 'draft') or removeParentNode('para')) and ""]]

Canceled Invoice [[ ((o.type == 'out_invoice' and o.state == 'cancel') or
removeParentNode('para')) and ""]]

Refund [[ (o.type=='out_refund' or removeParentNode('para')) and ""]][[ o.number ]]

Supplier Refund [[ (o.type=='in_refund' or removeParentNode('para')) and ""]][[ o.number ]]
```

## Conversion Brackets > Fields

This is reverse functionality in which you can change your old-format report into new format. If you want to access this functionality you can click on Open Report > Conversion Bracket > Fields from menubar.

The screenshot shows a software interface with a toolbar at the top and a main content area. In the content area, there is a list of document types under the heading 'Invoice out\_invoice 2008/001' and a detailed view of an invoice.

**Document Types:**

- PRO-FORMA out\_invoice
- PRO-FORMA out\_invoice
- Canceled Invoice out\_invoice
- Refund out\_invoice 2008/001
- Supplier Refund out\_invoice 2008/001
- Supplier Invoice out\_Invoice 2008/001

**Invoice Details:**

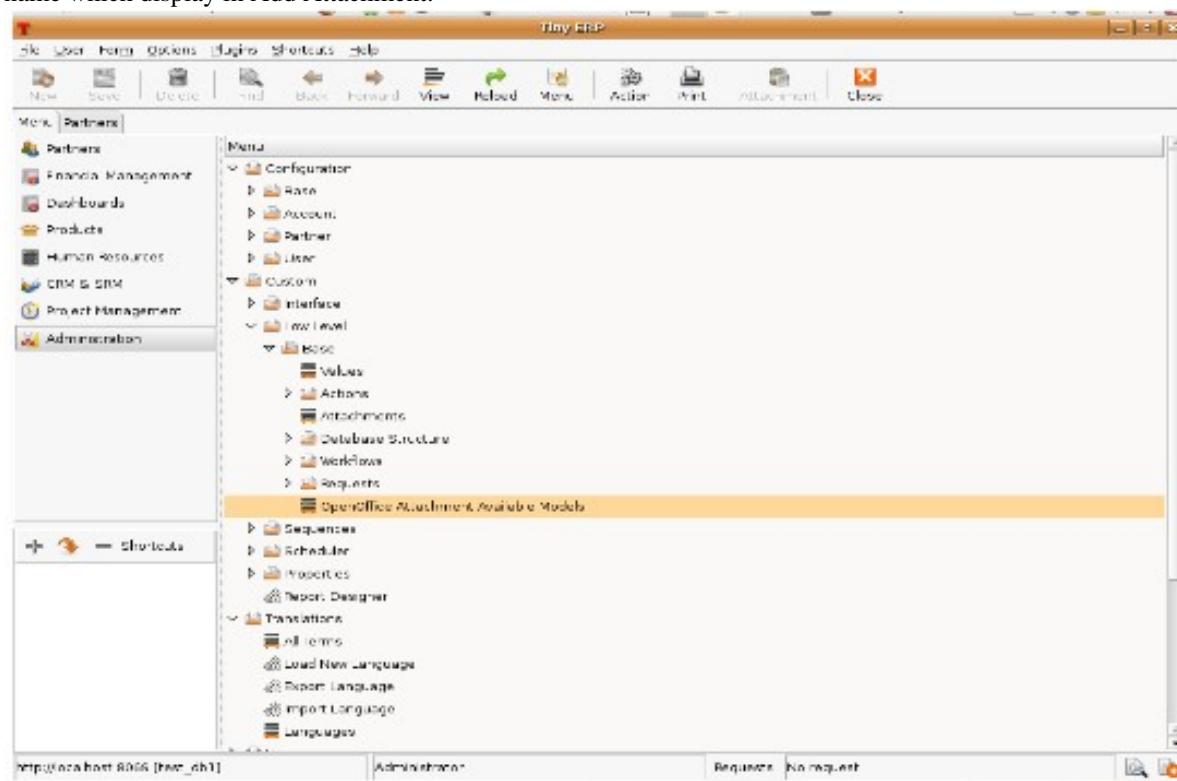
Document	Invoice Date	Partner Ref.
2008/003	2008-10-28	10001

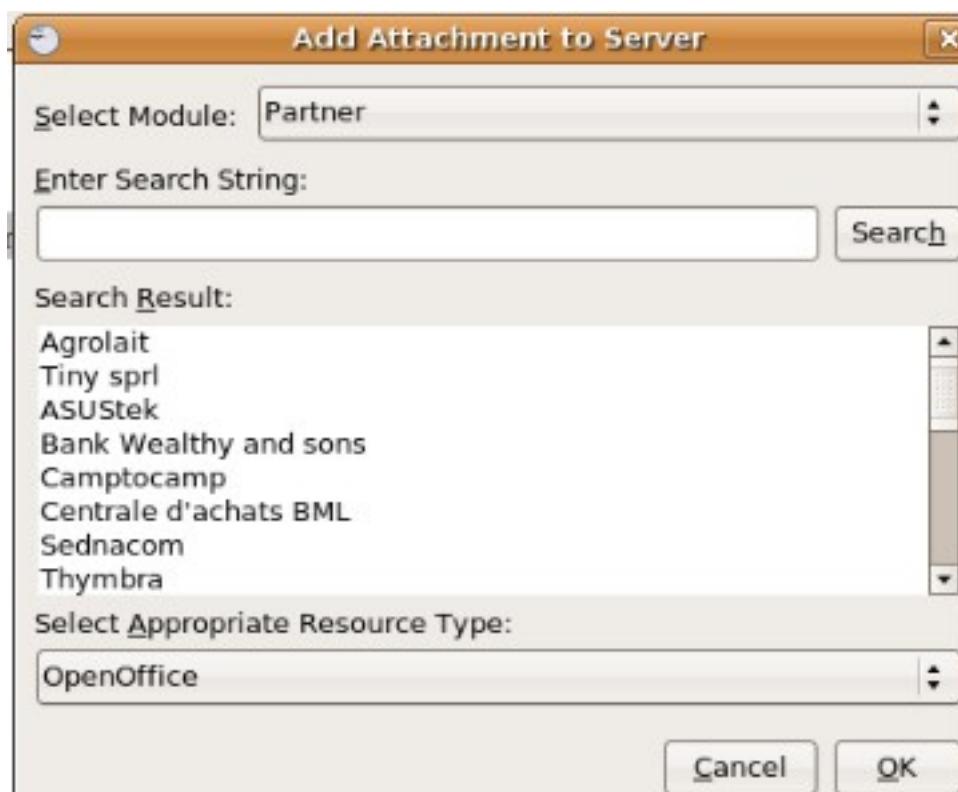
**Invoice Lines:**

Description	Taxes	Quantity	Unit Price	Disc. (%)	Price
Basic computer with Dvorak keyboard and left-handed mouse	21%	1,0	250,0	0,0	250,0 EUR

## Add Attachment

You can attach the report with record using attachment and the model which add in base\_model are displayed in list. Note: Server side Add the base\_model module this module available in trunk-extra-addons and add the model name which display in Add Attachment.



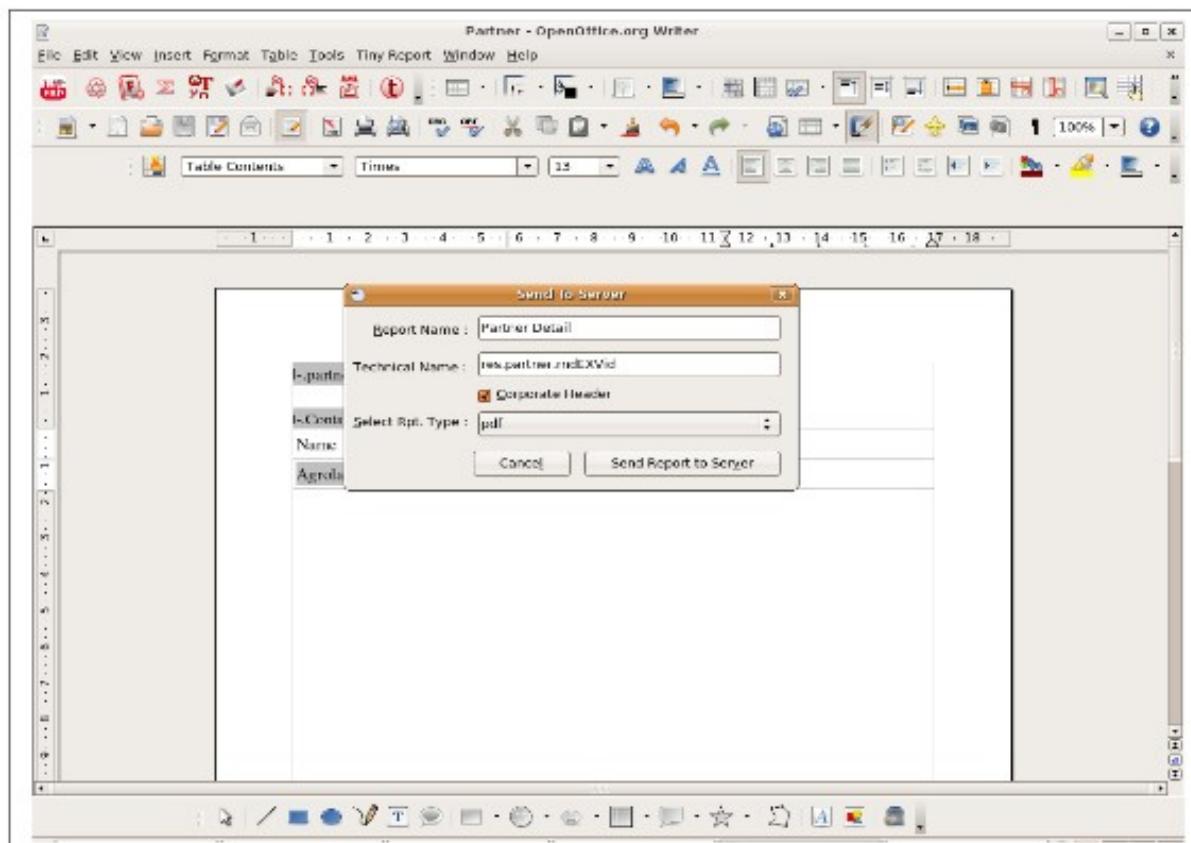


### Export to RML

- This functionality is used to generate rml from sxw.
- You Can access this tool from Open Report -> Export to RML in menubar.
- The basic feature of this functionality is now you can create your rml file in just few mouse clicks. The main requirements of this utility is you must have to save your report in Open Server using Send to the Server functionality.
- When you click on Export to RML It will open save file dialog box in which you can specify file name and destination of rml file to save

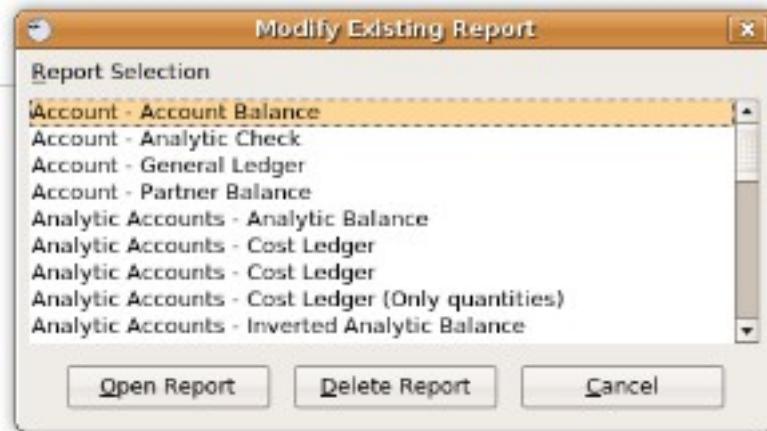
### Send to the Server

This is the most important functionality available in Open Report Design tools \* The basic feature of this functionality is to add new report or update existing report to Open Servein RML as well as SXW format. \* You can access this feature by using clicking on Open Report > Send to the Server in menubar \*

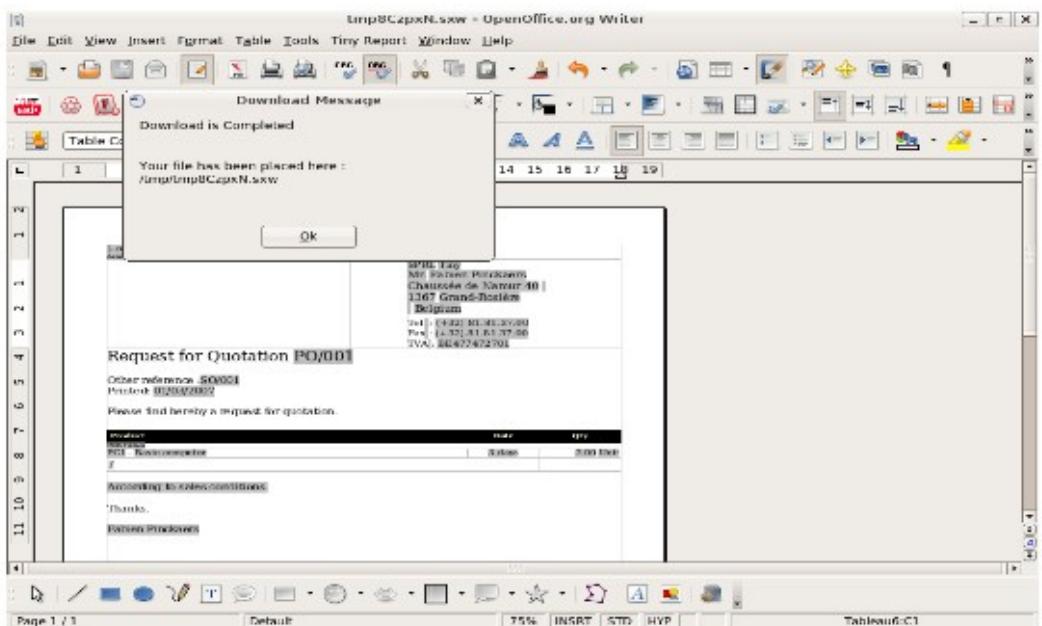


## Modify Existing Report

Openoffice.org Report Design tool provide functionality of modify existing report which is already available in Open server and also delete the report from database. You can access this feature by using clicking on OpenReport > Modify Existing Report in menubar. By clicking on Modify Existing Report I will display dialog box displayed below.



It will open existing report and you can delete the existing report. By clicking on Save to Temp Directory button you will get opened report in new writer window as displayed below.



## About

The about window shows version and copyright information. You can access it from Open Report> About in menu bar.



# INDEX

## B

Bazaar  
    installation, 308

## C

cheat  
    sheet, 297  
cheatsheet, 297

## I

Installation  
    Bazaar, 308

## M

memento, 297

## R

reference, 297

## S

sheet  
    cheat, 297