Open Object Developer Book

Release 1.0

Tiny SPRL

2009-04-10

CONTENTS

Part I

Forewords

CHAPTER

ONE

INTRODUCTION

Open ERP 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 Open ERP.

CHAPTER

TWO

WHO IS THIS BOOK FOR ?

CHAPTER

THREE

CONTENT OF THE BOOK

Book Contents

CHAPTER

FOUR

ABOUT THE AUTHOR(S)

Part II

Part 1 : Getting Started with OpenERP

CHAPTER

FIVE

DEVELOPMENT ENVIRONMENT

5.1 Working with Launchpad

5.1.1 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 trough the process needed to create your login.

This login is only needed if you intend to commit on bazaar on openerp-commiter 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 commiter must show his interest on working for openerp project and his ability to do it in a proper way as the selection for this group is based on meritocracy. It can be by proposing bug fixes, features requested on our *bug tracker* system. You can even suggest additional modules and/or functionalities on our *bug tracker* system.

You contribute or join Open ERP team, : https://help.launchpad.net/Teams/Joining

Contributors are people who wants to help the project getting better, add functionnality and improve stability. Everyone can contribute on the project with his own knowledge by reporting bugs, purposing smart improvment and posting patch.

The community team is available on launchpad: https://launchpad.net/~openerp-community

Member of the quality and commiter team are automatically members of the community.

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:

```
deb http://ppa.launchpad.net/bzr/ubuntu intrepid main
deb-src http://ppa.launchpad.net/bzr/ubuntu intrepid main
```

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 don't have at least 1.3 version, you can 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

If you experience problems with Bazaar, please read the *F.A.Q on Bazaar version control system* before asking any questions.

5.1.2 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.

Quick Summary

To download the latest sources and create your own local branches of OpenERP, do this:

```
bzr branch lp:openerp
cd openerp
./bzr_set.py
```

This will download all the component of openerp (server, client, addons) and create links of modules in addons in your server so that you can use it directly. You can change the bzr_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*, you can push your branch in 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 commiters 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 fixe 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 commiter and then the quality team to be merged in the official release.

[Read more about *Open ERP Team*]

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 interest 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 futur 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 knows what they can use or not.
- Once your code is good enough, you propose your branch for merging
- Your work will be evaluated by one responsible of the commiters 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 commiter team refuses your branch, they will explain why so that you can review your code to better fits guidelines (problem for futur migrations, ...)

The extra-addons branch, that stores all extra modules, is directly accessible to all commiters. If you are a commiter, you can work directly on this branch and commit your own work. This branch do 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 you to create 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 usefull 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 commiters 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 fixe 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 commiter and then the quality team to be merged in the official release.

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 interest 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 futur 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 knows what they can use or not.
- Once your code is good enough, you propose your branch for merging
- Your work will be evaluated by one responsible of the commiters 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 commiter team refuses your branch, they will explain why so that you can review your code to better fits guidelines (problem for futur migrations, ...)

The extra-addons branch, that stores all extra modules, is directly accessible to all commiters. If you are a commiter, you can work directly on this branch and commit your own work. This branch do 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 you to create 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 usefull 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.

5.1.3 Answer Tracker and Bugs 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 commiter (member of the Commiter Team) can set the status to "Fix Commited". Only commiters 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 Commited". 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 commiters can set the bug in the status "Fix Commited". Once quality team finish merging, they change the status to "Fix Released".

5.1.4 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.

5.1.5 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.

The Buleprints for OpenERP are listed here:

- https://blueprints.launchpad.net/openerp
- https://blueprints.launchpad.net/~openerp-commiter

Development Environment

Windows

Linux

Mac-OS

Directory Structure

Server

GTK-Client

Web Client

Configuration

Two configuration files are available:

- one for the client: ~/.terprc
- one for the server: ~/.terp_serverrc

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

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

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

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

tinyerp-server.py -s

Server Configuration File

The server configuration file .terp_serverrc is used to save server startup options. For the version 5.X configuration file is .openerp_serverrc. 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 Open ERP to your language

translate_out File used to export the language Open ERP 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 Will used debugged 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 alternativve 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
```

GTK-Client Configuration

login section

login login name to use to connect to Tiny ERP server

server address used by the server

port port used by the server

path section

share path used to find Tiny ERP shared files

pixmaps path used to find Tiny ERP pixmaps files

tip section

autostart Should the client display tips at startup

position Tip number the client will display

form section

autosave Will the client automatically save the change you made to a record

printer section

preview Preview report before printing

softpath Path to the pdf previewer

softpath_html Path to the html previewer

path Command used to print

logging section

logger log channels to display. List values are: @common@, @common.message@, @view@, @view.form@, @common.options@, @rpc.request@, @rpc.result@, @rpc.exception@

level logging level to show

output file used by the logger

verbose set the log level to INFO

client section

default_path Default path used by the client when saving/loading datas.

Default values:

```
[login]
login = admin
port = 8069
server = 192.168.1.4

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

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

[form]
autosave = False
```

```
[client]
default_path = /home/user
```

Web Client Configuration

Get a clone of each repository:

```
bzr clone lp:~openerp/openobject-server/trunk server
bzr clone lp:~openerp/openobject-client/trunk client
bzr clone lp:~openerp/openobject-client-web/trunk client-web
bzr clone lp:~openerp/openobject-addons/trunk addons
```

If you want to get a clone of the extra-addons repository, you can execute this command:

```
bzr clone lp:~openerp-commiter/openobject-addons/trunk-extra-addons extra-addons
```

run the setup scripts in the respective directories:

```
python2.4 setup.py build
python2.4 setup.py install
```

Currently the initialisation procedure of the server parameter –init=all to populate the database seems to be broken in trunk.

It is recommended to create a new database via the gtk-client. Before that the web-client will not work.

Start OpenERP server like this:

```
./openerp-server.py --addons-path=/path/to/my/addons
```

The `bin/addons` will be considered as default addons directory which can be overriden by the `/path/to/my/addons/`. That is if an addon exists in `bin/addons` as well as `/path/to/my/addons` (custom path) the later one will be given preference over the `bin/addons` (default path).

Command line options

General Options

-{-}version	show program version number and exit
-h, -{-}help	show this help message and exit
-c CONFIG, -{-}config=CONFIG	specify alternate config file
-s, -{-}save	save configuration to ~/.terp_serverrc
-v, -{-}verbose	enable debugging
-{-}pidfile=PIDFILE	file where the server pid will be stored
-{-}logfile=LOGFILE	file where the server log will be stored
-n INTERFACE, -{-}interface=INTERFACE	specify the TCP IP address
-p PORT, -{-}port=PORT	specify the TCP port
-{-}net_interface=NETINTERFACE	specify the TCP IP address for netrpc
-{-}net_port=NETPORT	specify the TCP port for netrpc

-{-}no-netrpc disable netrpc

-{-}no-xmlrpc disable xmlrpc

-i INIT, -{-}init=INIT init a module (use "all" for all modules)

-{-}without-demo=WITHOUT_DEMO load demo data for a module (use "all" for all modules)

-u UPDATE, -{-}update=UPDATE update a module (use "all" for all modules)

-{-}stop-after-init stop the server after it initializes

-{-}debug enable debug mode

-S, -{-}secure launch server over https instead of http

-{-}smtp=SMTP_SERVER specify the SMTP server for sending mail

-{-}price_accuracy=PRICE_ACCURACY specify the price accuracy

Database related options:

-d DB_NAME, -{-}database=DB_NAME specify the database name

-r DB_USER, -{-}db_user=DB_USER specify the database user name

-w DB_PASSWORD, -{-}db_password=DB_PASSWORD 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

Internationalization options:

Use these options to translate Tiny ERP 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

OpenERP Server and Web Client - Start/Stop

OpenERP 4.2

First check that all the required dependencies are installed. Then create the terp database. You have to make sure that your user has the correct credentials to create databases with PostgreSQL. For more information on this subject please refer to the PostgreSQL manual.:

```
$ createdb terp --encoding=unicode
```

Once the database created, you can start OpenERP. The content of the database will automatically be created at the first start.:

```
$ ./tinyerp-server.py
```

OpenERP 5.0 and above

- Check that all the required dependencies are installed.
- Make sure you are logged on as a user that has catalog admin rights in PostgreSQL. Refer to the PostgreSQL manual for more info on this.
- Start the OpenERP Server

```
./openerp-server.py
```

```
* Finally connect to the server with the GTK Client or eTiny and use the Create Database option to c
```

Shutting down the server

The easiest way the shut down the server on a Linux type system is to send the SIG INT signal to the server.

- **Find the Process ID.**
  - # ps ax | grep -i tiny
- **Use the kill comand with the PID.**
  - # kill -2 pid

5.2 Configuration

Two configuration files are available:

- one for the client: ~/.terprc
- one for the server: ~/.terp_serverrc

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

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

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

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

tinyerp-server.py -s

5.2.1 Server Configuration File

The server configuration file .terp_serverrc is used to save server startup options. For the version 5.X configuration file is .openerp_serverrc. 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 Open ERP to your language

translate_out File used to export the language Open ERP 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 Will used debugged 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 alternativve 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
```

5.2.2 GTK-Client Configuration

login section

login login name to use to connect to Tiny ERP server

server address used by the server

port port used by the server

path section

share path used to find Tiny ERP shared files

pixmaps path used to find Tiny ERP pixmaps files

tip section

autostart Should the client display tips at startup

position Tip number the client will display

form section

autosave Will the client automatically save the change you made to a record

printer section

preview Preview report before printing

softpath Path to the pdf previewer

softpath_html Path to the html previewer

path Command used to print

logging section

logger log channels to display. List values are: @common@, @common.message@, @view@, @view.form@, @common.options@, @rpc.request@, @rpc.result@, @rpc.exception@

level logging level to show

output file used by the logger

verbose set the log level to INFO

client section

default_path Default path used by the client when saving/loading datas.

Default values:

```
[login]
login = admin
port = 8069
server = 192.168.1.4

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

```
[logging]
output = stdout
logger =
verbose = True
level = ERROR

[form]
autosave = False

[client]
default_path = /home/user
```

5.2.3 Web Client Configuration

Get a clone of each repository:

```
bzr clone lp:~openerp/openobject-server/trunk server
bzr clone lp:~openerp/openobject-client/trunk client
bzr clone lp:~openerp/openobject-client-web/trunk client-web
bzr clone lp:~openerp/openobject-addons/trunk addons
```

If you want to get a clone of the extra-addons repository, you can execute this command:

```
bzr clone lp:~openerp-commiter/openobject-addons/trunk-extra-addons extra-addons
```

run the setup scripts in the respective directories:

```
python2.4 setup.py build
python2.4 setup.py install
```

Currently the initialisation procedure of the server parameter –init=all to populate the database seems to be broken in trunk.

It is recommended to create a new database via the gtk-client. Before that the web-client will not work.

Start OpenERP server like this:

```
./openerp-server.py --addons-path=/path/to/my/addons
```

The `bin/addons` will be considered as default addons directory which can be overriden by the `/path/to/my/addons/`. That is if an addon exists in `bin/addons` as well as `/path/to/my/addons` (custom path) the later one will be given preference over the `bin/addons` (default path).

5.3 Command line options

5.3.1 General Options

-{-}version	show program version number and exit
-h, -{-}help	show this help message and exit
-c CONFIG, -{-}config=CONFIG	specify alternate config file

-s, -{-}save save configuration to ~/.terp_serverrc

-v, -{-}verbose enable debugging

-{-}pidfile=PIDFILE file where the server pid will be stored

-{-}logfile=LOGFILE file where the server log will be stored

-n INTERFACE, -{-}interface=INTERFACE specify the TCP IP address

-p PORT, -{-}port=PORT specify the TCP port

-{-}net_interface=NETINTERFACE specify the TCP IP address for netrpc

-{-}net_port=NETPORT specify the TCP port for netrpc

-{-}no-netrpc disable netrpc

-{-}no-xmlrpc disable xmlrpc

-i INIT, -{-}init=INIT init a module (use "all" for all modules)

-{-}without-demo=WITHOUT_DEMO load demo data for a module (use "all" for all modules)

-u UPDATE, -{-}update=UPDATE update a module (use "all" for all modules)

-{-}stop-after-init stop the server after it initializes

-{-}debug enable debug mode

-S, -{-}secure launch server over https instead of http

-{-}smtp=SMTP_SERVER specify the SMTP server for sending mail

-{-}price_accuracy=PRICE_ACCURACY specify the price accuracy

5.3.2 Database related options:

-d DB_NAME, -{-}database=DB_NAME specify the database name

-r DB_USER, -{-}db_user=DB_USER specify the database user name

-w DB_PASSWORD, -{-}db_password=DB_PASSWORD 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

5.3.3 Internationalization options:

Use these options to translate Tiny ERP 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

5.4 OpenERP Server and Web Client - Start/Stop

5.4.1 OpenERP 4.2

First check that all the required dependencies are installed. Then create the terp database. You have to make sure that your user has the correct credentials to create databases with PostgreSQL. For more information on this subject please refer to the PostgreSQL manual.:

```
$ createdb terp --encoding=unicode
```

Once the database created, you can start OpenERP. The content of the database will automatically be created at the first start.:

```
$ ./tinyerp-server.py
```

5.4.2 OpenERP 5.0 and above

- Check that all the required dependencies are installed.
- Make sure you are logged on as a user that has catalog admin rights in PostgreSQL. Refer to the PostgreSQL manual for more info on this.
- Start the OpenERP Server

```
./openerp-server.py
```

```
* Finally connect to the server with the GTK Client or eTiny and use the Create Database option to c
```

5.5 Shutting down the server

The easiest way the shut down the server on a Linux type system is to send the SIG INT signal to the server.

- **Find the Process ID.** – # ps ax | grep -i tiny
- **Use the kill comand with the PID.** – # kill -2 pid

CHAPTER

SIX

MOULDER DEVELOPMENT APPROACH

6.1 OpenObject Server and Modules

- **OpenERP** is a Client/Server system that works over a IP Network.
- **OpenERP** programming language is Python.
- **OpenERP** uses Object-Oriented technologies.
- **OpenERP** records its data with a PostgreSQL relational database.
- **OpenERP** business objects are modeled with an Object Relational Mapping (ORM) system.
- **OpenERP** offers three Human Machine Interfaces (HMI) a GTK client, a QT client and a web client (eTiny).
- **OpenERP** uses ReportLab for report generation in (PDF).
- **OpenERP** uses XML for several purpose: describing data, view, reports, data transport (XML-RPC)

6.1.1 Technical Architecture

Server/client, XML-RPC

Open ERP is a based on a client/server architecture. The server and the client communicate using the XML-RPC protocol. XML-RPC is a very simple protocol which allows the client to do remote procedure calls. The function called, its arguments, and the result are sent HTTP and encoded using XML.

For more information on XML-RPC, please see: http://www.xml-rpc.com/

Since version 4.2, there is a new protocol between client/server that have been called net-rpc. It is based on the python cPickle function, it is faster than the xml-rpc.

Client

The logic of Open ERP is entirely on the server side. The client is very simple; his work is to ask data (forms, lists, trees) from the server and to send them back. With this approach, nearly all developments are made on the server side. This makes Open ERP easier to develop and to maintain.

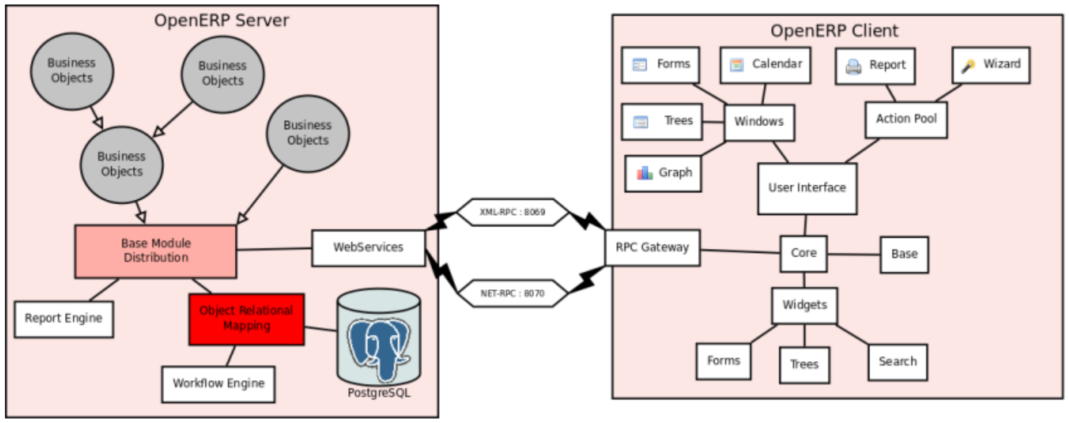
The client doesn't understand what it posts. Even actions like 'Click on the print icon' are sent to the server to ask how to react.

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

Architecture



Explanation of modules Server - Base distribution

We use a distributed communication mechanism inside the Open ERP server. Our engine support 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 on multiple computers. 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 Open ERP:

A sales order generates an invoice and a shipping order Server - Report Engine

Reports in Open ERP 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 Open ERP, they described 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.

6.2 Module Integrations

The are many different modules available for Open ERP and suited for different business models. Nearly all of these are optional (except ModulesAdminBase), making it easy to customize Open ERP 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 Open ERP platform.

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

Here is an example of __terp__.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.

6.3 Inheritance

6.3.1 Traditional Inheritance

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'
```

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 fielda and behaviour, but they differ in heavy programatical 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 inheritancy 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 inheritancy 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.

6.3.2 Inheritance by Delegation

Syntax ::

```
class tiny_object(osv.osv)
    _name = 'tiny.object'
    _table = 'tiny_object'
    _inherits = { 'tiny.object'_1_ : name_col'_1_', 'tiny.object'_2_ : name_col'_2_', ..., 'tin
    (...)
```

The object 'tiny.object' inherits from all the columns and all the methods from the n objects 'tiny.object'_1_, ..., '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'_1_' name_col'_2_'* ... *name_col'_n_'* are of type string and determine the title of the columns in which the foreign keys from 'tiny.object'_1_, ..., '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.

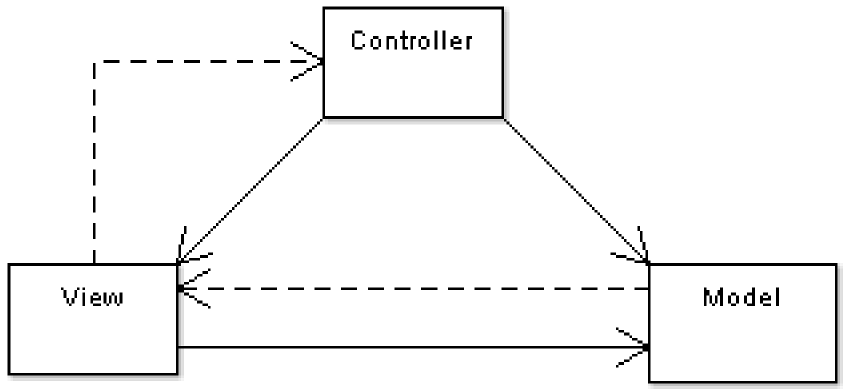
CHAPTER

SEVEN

OPENOBJECT ARCHITECTURE - MVC

7.1 MVC - Model, View, Controller

According to Wikipedia,"a Model-view-controller (MVC) is an architectural pattern used in software engineering. In complex computer applications that present lots of data to the user, one often wishes to separate data (model) and user interface (view) concerns, so that changes to the user interface do not impact the data handling, and that the 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."



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 controller.
- 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.

7.1.1 MVC Model in Tiny ERP

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

- model : The PostgreSQL tables.

- view : views are defined in XML files in Tiny ERP.
- controller : The objects of TinyERP.

7.2 MVCSQL

7.2.1 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 Open ERP:

```
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)

7.2.2 Example 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 Open ERP.

7.2.3 Loops and special fields

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

7.2.4 A complete example

Here is a complete example, from the Open ERP 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
         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 = []
         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]
              else:
                   return [],[]
         else:
              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,
                   })
              if bom.routing_id:
                   for wc_use in bom.routing_id.workcenter_lines:
                        wc = wc_use.workcenter_id
                        cycle = factor * wc_use.cycle_nbr
                        result2.append({
                             'name': bom.routing_id.name,
                             'workcenter_id': wc.id,
                             'sequence': level,
                        'cycle': cycle,
                        'hour': wc_use.hour_nbr + (
                            wc.time_start+wc.time_stop+cycle*wc.time_cycle) *
                            (wc.time_efficiency or 1
                  })
          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
```

Part III

Part 2 : Module Development

CHAPTER

EIGHT

FIRST MODULE TO OPENERP

Open ERP is a Python based client/server program for Enterprise Resource Planning. It consist of a client "tinyerp-client" and a server "tinyerp-server" while the persistence is provided by Postgresql. Open ERP currently uses XML-RPC for communication over a network. Once installed Open ERP has a modular structure that allows modules to be added as needed.

8.1 The Modules - Introduction

The usage of the modules is the way to extend Tiny ERP functionality. The default Tiny ERP installation is organized as a kernel and various modules among which we can distinguish :

- base : The most basic module. Defines ir.property, res.company, res.request, res.currency, res.user, res.partner
- crm : Customer & Supplier Relationship Management.
- sale : Sales Management.
- mrp : Manufacturing Resource Planning.

New modules can be programed easily, and require a little practice of XML and Python.

8.1.1 Module Structure

The Modules

1. Introduction
2. **Files & Directories** (a) __terp__.py
   (b) __init__.py
   (c) **XML Files** i. Actions
      ii. Menu Entries
      iii. Reports
      iv. Wizards
3. Profiles

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: **__terp__.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 Tiny ERP XML files is more detailed in the section Data Loading Files XML. 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 __terp__.py

In the created module directory, you must add a **__terp__.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. Open ERP 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. Open ERP 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 __terp__.py file for the product module:

```
{
    "name" : "Products & Pricelists",
    "version" : "1.0",
    "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.

Objects

All Tiny ERP resources are objects: menus, actions, reports, invoices, partners, ... Tiny ERP 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, ...)

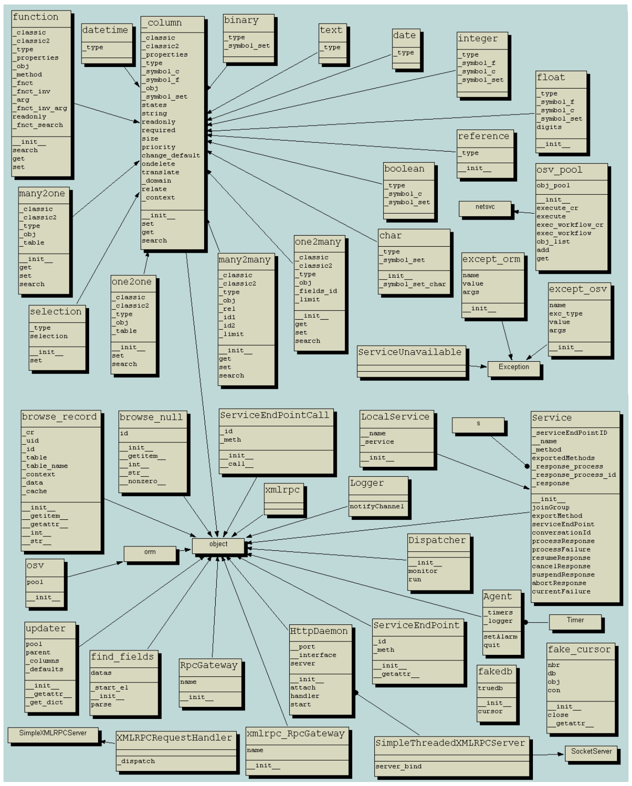


Figure 8.1: *The Physical Objects Model of [OpenERP version 3.0.3]*

PostgreSQL The ORM of Open ERP is constructed over PostgreSQL. It is thus possible to query the object used by Open ERP 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

```
%define=lightblue color=#27adfb%
```

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

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

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

Let's review an example taken from the TinyERP 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="l0">
     <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 attribute indicating the field name
- **eval** – python expression that indicating the value to add
- **ref** – reference to an id defined in this file

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')]]"/>
```

getitem tag

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

- **type** o 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>
```

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` [1] 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 Thourgh the interface and module recorder Then, put the generated XML in your own module

Views

(:title 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

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.

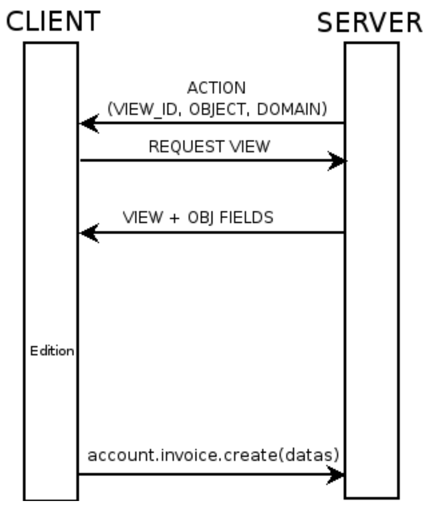
Views are described in XML.

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

[1] http://www.gnu.org/software/autoconf/manual/gettext/PO-Files.html#PO-Files



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

Open ERP 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"?>
<terp>
    <data>
         <wizard string="Employee Info"
                model="hr.employee"
                name="employee.info.wizard"
                id="wizard_employee_info"/>
    </data>
</terp>
```

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"?>
<terp>
     <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>
</terp>
```

Workflow

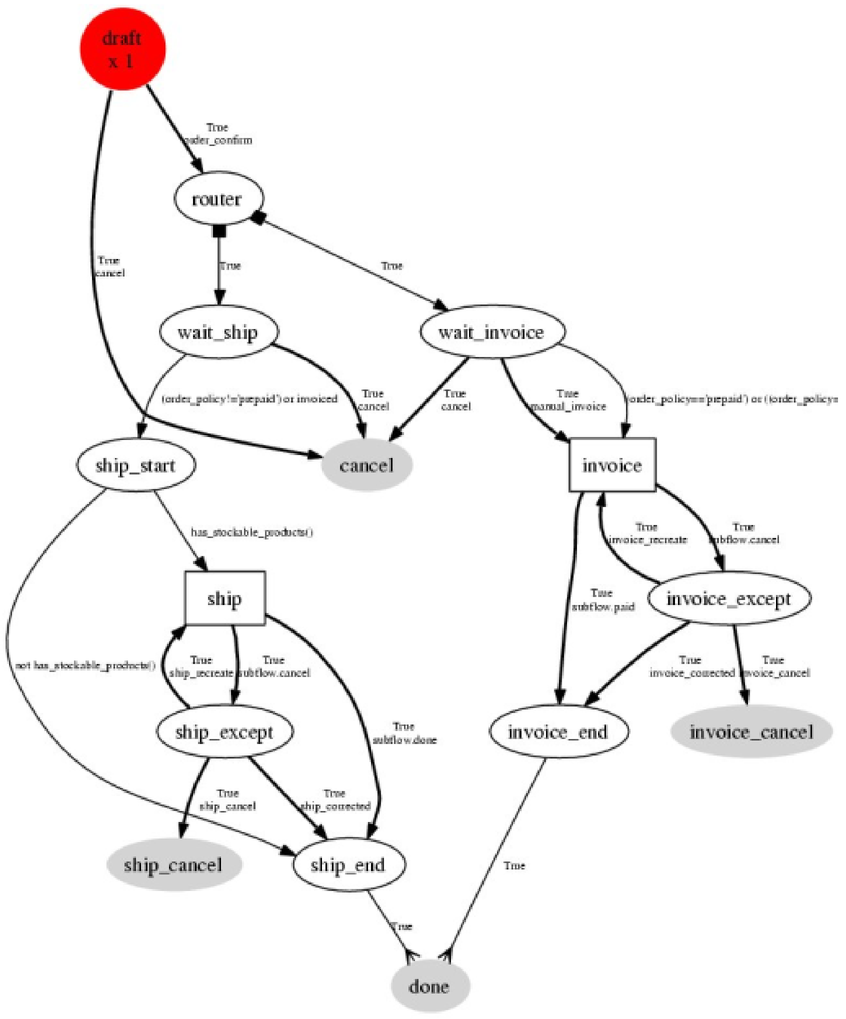
The objects and the views allow you to define new forms very simply, lists/trees and interactions between them. But it is not enough : you have to define the dynamics of these objects.

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

8.1.2 OpenERP Module Descriptor File : __terp__.py

Normal Module

In the created module directory, you must add a **__terp__.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. Open ERP 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. Open ERP 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 __terp__.py file for the *product* module:

```
{
    "name" : "Products & Pricelists",
    "version" : "1.0",
    "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.

Profile Module

The purpose of a profile is to initialize Open ERP 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 __terp__.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/__terp__.py, which corresponds to the manufacturing industry profile in Open ERP.

```
{
     "name":"Manufacturing industry profile",
     "version":"1.0",
     "author":"Open",
     "category":"Profile",
     "depends":["mrp", "crm", "sale", "delivery"],
     "demo_xml":[],
     "update_xml":[],
     "active":False,
     "installable":True,
}
```

8.1.3 Create Module

Getting the skeleton directory

Creating a new module is quickly done by copying the module called "simple" or "custom" (depending on your OpenERP version) into a new directory.

As an example on Ubuntu:

```
$ cd /usr/lib/tinyerp-server/addons/
$ sudo cp -r custom travel
```

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...

Changing the default definition

To change the default settings of the custom module (which is now the "travel" module), get yourself into the "travel" directory and edit *__terp__.py*. *gedit*, in the following example, is just a simple text editor. Feel free to use another one.

```
$ cd travel
$ gedit __terp__.py
```

The file looks like this:

```
#
# Use the custom module to put your specific code in a separate module.
#
{
        "name" : "Module for custom developments",
        "version" : "1.0",
        "author" : "Tiny",
        "category" : "Generic Modules/Others",
        "website": "http://www.tinyerp.com",
        "description": "Sample custom module where you can put your customer specific developments."
        "depends" : ["base"],
        "init_xml" : [],
        "update_xml" : ["custom_view.xml"],
        "active": False,
        "installable": True
}
```

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

```
{
        "name" : "Travel agency module",
        "version" : "1.0",
        "author" : "Tiny",
        "category" : "Generic Modules/Others",
        "website": "http://www.tinyerp.com",
        "description": "A module to manage hotel bookings and a few other useful features.",
        "depends" : ["base"],
        "init_xml" : [],
        "update_xml" : ["custom_view.xml"],
        "active": True,
        "installable": True
}
```

Note the "active" field becomes true.

Changing the main module file

Now you need to update the custom.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 custom.py file should initially look like this (intentionally removing the comments):

> from osv import osv, fields
>
> #class custom_material(osv.osv): # _name = 'network.material' # _inherit = 'network.material' # _columns = { # } # _defaults = { # } #custom_material()

The '#' signs represent comments. You'll have to remove them, rename the class and its attributes to something 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.

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:

```
<terp>
<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_admi
</data>
</terp>
```

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 you take a look at the 20 minutes Flash tutorial or download the travel agency module example.

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

8.1.4 Creating Action

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.

CHAPTER

NINE

OBJECTS, FIELDS AND METHODS

9.1 OpenERP Objects

9.1.1 Introduction

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:

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

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

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

9.2 The ORM - Object Relation Model

9.2.1 The Models

ORM is for Object-Relational Mapping.

OpenERP modeling is based on "objects" but is data is stored in a classical relational database named Postgresql.

ORM job is to fill the gap between Open-objects and sql tables.

Python is the programming langage giving the behavior and data description of Open-objects (This is not stored in the database). "ORM" is the python class ancestor of all Open-objects.

A Open-object is modeling by a static python description for his behavior and data, an a miror sql description for his data storage.

9.3 OpenERP Object Attributes

9.3.1 Objects Introduction

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

9.3.2 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 is optional. The predefined fields are:

9.3.3 Prefined names

_auto

Determines whether a corresponding PostgreSQL table must be generated automatically from the object. Setting _auto to False can be useful in case of Open ERP 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 details.

_constraints

The constraints on the object. See the constraints section for details.

_sql_constraints

The SQL Constraint on the object. See theconstraints SQL section for more 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 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 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)

_table

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

9.4 Object Inheritance - _inherit

9.4.1 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'
```

9.4.2 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 fielda and behaviour, but they differ in heavy programatical 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 inheritancy 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 inheritancy 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.

9.5 Inheritance by Delegation - _inherits

Syntax ::

```
class tiny_object(osv.osv)
    _name = 'tiny.object'
    _table = 'tiny_object'
    _inherits = { 'tiny.object'_1_ : name_col'_1_', 'tiny.object'_2_ : name_col'_2_', ..., 'tiny
    (...)
```

The object 'tiny.object' inherits from all the columns and all the methods from the n objects 'tiny.object'_1_, ..., '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'_1_' name_col'_2_'* ... *name_col'_n_'* are of type string and determine the title of the columns in which the foreign keys from 'tiny.object'_1_, ..., '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.

9.6 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 Open ERP inherits (as you can see in server/bin/osv/fields.py):

```
def __init__(self, string='unknown', required=False, readonly=False,
                domain=[], context="", states={}, priority=0, change_default=False, size=None,
                ondelete="setnull", translate=False, select=False, **args) :
```

9.7 Type of Fields

9.7.1 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'), ... )
```

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={}):
    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
```

9.7.2 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 parameter)
```

- **Optional parameters:**
  - **ondelete: What should happen when the resource this field points to is deleted.** *
    Predefined value: "cascade", "set null"
    * 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',
                 'other.object.id',
                 'actual.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
  - other.object.id and actual.object.id are the fields' names used in the relation table

Example:

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

related

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

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

Functional Field

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="%green%float%black%", fnct_search=None, obj=None, method=False, store=True

where

- **type** is the field type name returned by the function. It can be any field type name except function.
- **store** If you want to store field in database or not. Default is False.
- **method** whether the field is computed by a method (of an object) or a global function
- **fnct** is the function or method that will compute the field value. It must have been declared before declaring the functional field.

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.

- **fnct_inv** is the function or method that will allow writing values in that field.

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** allows you to define the searching behaviour on that field.

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

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

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

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

The return value is a list countaining 3-part tuplets which are used in search funtion:

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

Example Of Functional Field

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.functio
                                      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={...} Enhancement

It will compute the field depends on other objects.

Syntex store={'object_name':(function_name,['field_name1','field_name2'],priority)} It will call function function_name when any changes will be applied on field list ['field1','field2'] on object 'object_name' and output of the function will send as a parameter for main function of the field.

Example In membership module

```
'membership_state': fields.function(_membership_state, method=True, string='Current membership 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)}),
```

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)"/>
</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. It 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.

9.8 ORM methods

`create`

Description

Create a new resource

Signature: def create(cr, uid, vals, context={})

Parameters:

- vals: a dictionary of values for every field. This dictionary must use this form: **{'name_of_the_field': value, ...}**
- context (optional): the actual context dictionary.

Returns: the id of the newly created resource.

Example:

```
id = pooler.get_pool(cr.dbname).get('res.partner.event').create(cr, uid,
       {'name': 'Email sent through mass mailing',
        'partner_id': partner.id,
        'description': 'The Description for Partner Event'})
```

`search`

Description

Search all the resources which satisfy certain criteria

Signature: def search(self, cr, uid, args, offset=0, limit=2000,order=None,context=None, count=False)

Parameters

- **args: a list of tuples containing the search criteria. This list must be of the form: [('name_of_the_field', 'operator', value)**
  =, >, <, <=, >=
  - IN (sql)
  - LIKE, ILIKE (sql)
  - child_of
- offset (optional): do not return the "offset" first results.
- limit (optional): maximum number of results to return.

Returns: the list of ids of matching resources.

Example:

```
ids = pooler.get_pool(cr.dbname).get('res.partner').search(cr, uid, [('category_id', '=', 'Customer'
```

This example will return a list with all the partners that have the category 'Customer'.

`read`

Description

List of fields resources values.

Signature: def read(self, cr, uid, ids, fields=None, context={})

Parameters:

- ids: list of the identifiers of the resources to read (list of integers).
- fields (optional): the list of the interested fields. If a value is not provided for this parameter, the function will check all the fields.
- context (optional): the actual context dictionary.

Returns: A list of dictionaries (a dictionary per resource asked) of the form [{'name_of_the_field': value, ...}, ...]

Example:

```
values = pooler.get_pool(cr.dbname).get('res.partner').
            read(cr, uid, ids, ['name','category_id'], context=context)
```

`browse`

Description

Return one or several resources with the objects form. These object fields can be reached directly with the pointed notation ("object.name_of_the_field"). The "relations" fields are also automatically evaluated to allow you to recover the values in the "neighbors" objects.

Signature: def browse(self, cr, uid, select, offset=0, limit=2000)

Parameters

- **select: this parameter accept data of several types:**
  - an integer : identifier of a resource
  - a list of integers (list of identifiers)
- offset (optional): the number of results to pass.
- limit (optional): the maximum number of results to return.

Returns:

- if an integer (identifier) has been passed as select parameter, return an object having the properties described here above.
- if a list of integer (identifiers) has been passed, return the object list.

Example

Let's consider the case of a partner (object 'res.partner') and of a partner contact (object 'res.partner.address'). Let's suppose that we know the identifier of a partner contact (name contact_id) and we want to recover his name and the account number of the company he works for.

Knowing that the object res.partner contains the field:

```
'bank':fields.char('Bank account',size=64),
```

and the object res.partner.address contains the fields:

```
'partner_id': fields.many2one('res.partner', 'Partner', required=True),
'name': fields.char('Contact Name', size=64),
```

the most simple way to proceed is to use the browse method:

```
addr_obj = self.pool.get('res.partner.address').browse(cr, uid, contact_id)
```

so, to recover the two fields that interest us, you have to write:

```
name = addr_obj.name
account_num = addr_obj.partner_id.bank
```

Note: *This method is only useful locally (on the server itself) and not with the other interfaces !!*

`write`

Description

Writes values in one or several fields of one or several resources

Signature: def write(self, cr, uid, ids, vals, context={})

Parameters:

- ids: the resources identifiers list to modify.
- vals: a dictionary with values to write. This dictionary must be with the form: {'name_of_the_field': value, ...}.
- context (optional): the actual context dictionary.

Returns: True

Example:

```
self.pool.get('sale.order').write(cr, uid, ids, {'state':'cancel'})
```

`unlink`

Description

Delete one or several resources

Signature: def unlink(self, cr, uid, ids)

Parameters:

- ids: the identifiers resources list to delete.

Returns: True

Example:

```
self.pool.get('sale.order').unlink(cr,uid, ids)
```

9.8.1 Methods to manipulate the default values

`default_get`

Description

Get back the value by default for one or several fields.

Signature: def default_get(self, cr, uid, fields, form=None, reference=None)

Parameters:

- fields: the fields list which we want to recover the value by default.
- form (optional): TODO
- reference (optional): TODO

Returns: dictionary of the default values of the form {'field_name': value, ... }

Example:

```
self.pool.get('hr.analytic.timesheet').default_get(cr, uid, ['product_id','product_uom_id'])
```

`default_set`

Description

Change the default value for one or several fields.

Signature: def default_set(self, cr, uid, field, value, for_user=False)

Parameters:

- field: the name of the field that we want to change the value by default.
- value: the value by default.
- for_user (optional): boolean that determines if the new default value must be available only for the current user or for all users.

Returns: True

Example:

```
TODO
```

9.8.2 Methods to manipulate the permissions

`perm_read`

Description **Signature:** def perm_read(self, cr, uid, ids)

Parameters:

- ids: an integer list

Returns: a list of dictionaries with the following keys

- level : access level
- uid : user id
- gid : group id
- create_uid: user who created the resource
- create_date: date when the resource was created
- write_uid: last user who changed the resource
- write_date: date of the last change to the resource

`perm_write`

Description **Signature:** def perm_write(self, cr, uid, ids, fields)

Parameters:

Returns:

Example:

```
self.pool.get('res.partner').perm_read(cr, uid, ids, context)
```

9.8.3 Methods to generate the fields and the views

`fields_get`

Description **Signature:** def fields_get(self, cr, uid, fields = None, context={})

Parameters:

- fields: a list of fields that interest us, if None, all the fields
- context: context['lang']

Result:

Example:

In payment.line in account_payment module

```
def fields_get(self, cr, uid, fields=None, context=None):
    res = super(payment_line, self).fields_get(cr, uid, fields, context)
    if 'communication2' in res:
        res['communication2'].setdefault('states', {})
        res['communication2']['states']['structured'] = [('readonly', True)]
        res['communication2']['states']['normal'] = [('readonly', False)]
    return res
```

`fields_view_get`

Description Signature: def fields_view_get(self, cr, uid, view_id=None, view_type='form', context={}, toolbar=False)

Parameters:

Result:

Example:

In membership module [product.product]:

```
def fields_view_get(self, cr, user, view_id=None, view_type='form', context=None, toolbar=False):
    if ('product' in context) and (context['product']=='membership_product'):
        model_data_ids_form = self.pool.get('ir.model.data').search(cr,user,[('model','=','ir.ui.vie
                                                                   ['membership_products_form','membership_
        resource_id_form = self.pool.get('ir.model.data').
                            read(cr,user,model_data_ids_form,fields=['res_id','name'])
        dict_model={}
        for i in resource_id_form:
            dict_model[i['name']]=i['res_id']
        if view_type=='form':
            view_id = dict_model['membership_products_form']
        else:
            view_id = dict_model['membership_products_tree']
    return super(Product,self).fields_view_get(cr, user, view_id, view_type, context, toolbar)
```

`distinct_field_get`

Description Signature: def distinct_field_get(self, cr, uid, field, value, args=[], offset=0, limit=2000)

Parameters:

Result:

Example:

```
TODO
```

9.8.4 Methods concerning the name of the resources

`name_get`

Description Signature: def name_get(self, cr, uid, ids, context={})

Parameters:

Result: a list of tuples of the form [(id, name), ...]

Example:

In res.partner.address:

```
def name_get(self, cr, user, ids, context={}):
    if not len(ids):
        return []
    res = []
    for r in self.read(cr, user, ids, ['name','zip','city']):
        addr = str(r['name'] or '')
        if r['name'] and (r['zip'] or r['city']):
            addr += ', '
        addr += str(r['zip'] or '') + ' ' + str(r['city'] or '')
        res.append((r['id'], addr))
    return res
```

name_search

Description Signature: def name_search(self, cr, uid, name=, args=[], operator='ilike', context={})

'Parameters:

Result:

Example:

In res.country:

```
def name_search(self, cr, user, name='', args=None, operator='ilike',
        context=None, limit=80):
    if not args:
        args=[]
    if not context:
        context={}
    ids = False
    if len(name) == 2:
        ids = self.search(cr, user, [('code', '=', name)] + args,
                          limit=limit, context=context)
    if not ids:
        ids = self.search(cr, user, [('name', operator, name)] + args,
                          limit=limit, context=context)
    return self.name_get(cr, user, ids, context)
```

CHAPTER

TEN

VIEWS AND EVENTS

10.1 Introduction to Views

As all data of the program 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 have to know that the Open ERP user interface is dynamic, it means that it is not described "statically" by some code, but 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 moment (even during the program execution). After a modification to a displayed view has occurred, you simply need to close the tab corresponding to that 'view' and re-open it for the changes to appear.

10.1.1 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

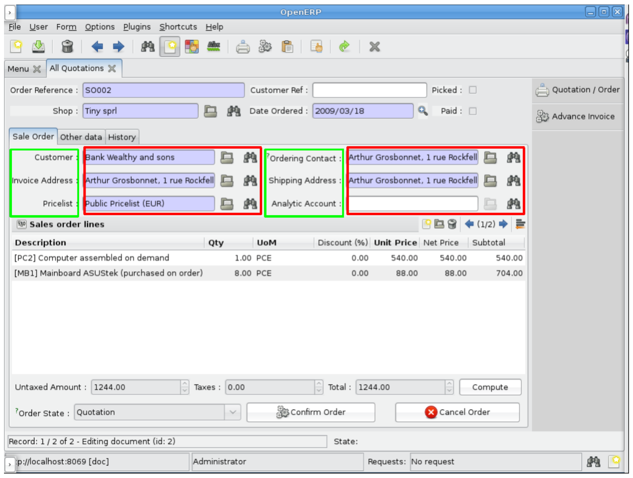
Note: *Since Open ERP 4.1, form views can also contain graphs.*

10.2 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.



Order Reference :	SO002	Customer Ref :		Picked :	☐
Shop :	Tiny sprl	Date Ordered :	2009/03/18	Paid :	☐

Customer :	Bank Wealthy and sons	Ordering Contact :	Arthur Grosbonnet, 1 rue Rockfell
Invoice Address :	Arthur Grosbonnet, 1 rue Rockfell	Shipping Address :	Arthur Grosbonnet, 1 rue Rockfell
Pricelist :	Public Pricelist (EUR)	Analytic Account :	

Description	Qty	UoM	Discount (%)	Unit Price	Net Price	Subtotal
[PC2] Computer assembled on demand	1.00	PCE	0.00	540.00	540.00	540.00
[MB1] Mainboard ASUStek (purchased on order)	8.00	PCE	0.00	88.00	88.00	704.00

Untaxed Amount : 1244.00 Taxes : 0.00 Total : 1244.00

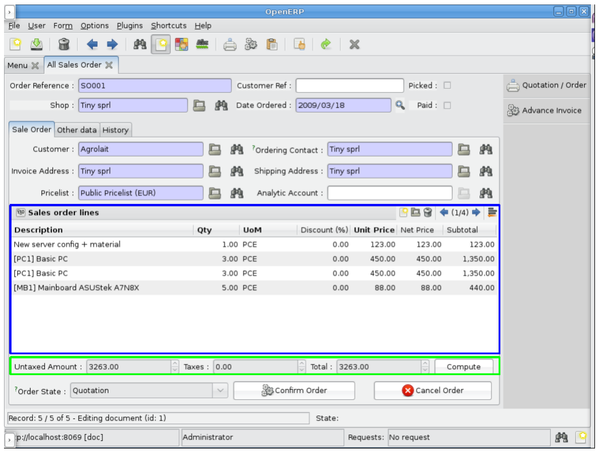
Order State : Quotation

Record: 1 / 2 of 2 - Editing document (id: 2) State:

p://localhost:8069 [doc] Administrator Requests: No request

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.



Order Reference :	SO001	Customer Ref :		Picked :	☐
Shop :	Tiny sprl	Date Ordered :	2009/03/18	Paid :	☐

Customer :	Agrolait	Ordering Contact :	Tiny sprl
Invoice Address :	Tiny sprl	Shipping Address :	Tiny sprl
Pricelist :	Public Pricelist (EUR)	Analytic Account :	

Description	Qty	UoM	Discount (%)	Unit Price	Net Price	Subtotal
New server config + material	1.00	PCE	0.00	123.00	123.00	123.00
[PC1] Basic PC	3.00	PCE	0.00	450.00	450.00	1,350.00
[PC1] Basic PC	3.00	PCE	0.00	450.00	450.00	1,350.00
[MB1] Mainboard ASUStek A7N8X	5.00	PCE	0.00	88.00	88.00	440.00

Untaxed Amount : 3263.00 Taxes : 0.00 Total : 3263.00

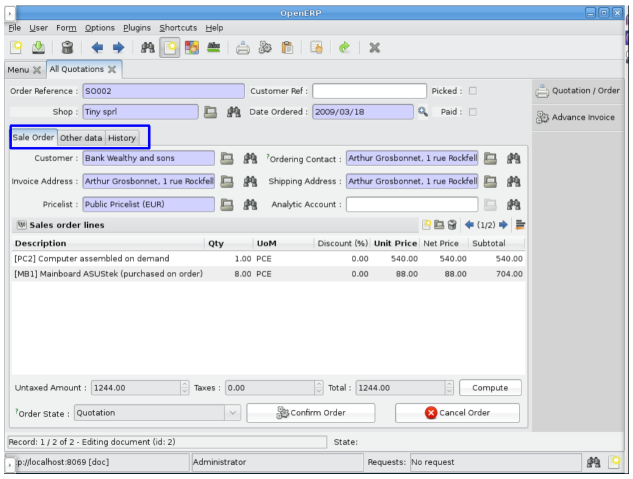
Order State : Quotation

Record: 5 / 5 of 5 - Editing document (id: 1) State:

p://localhost:8069 [doc] Administrator Requests: No request

- 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.



Order Reference : SO002 Customer Ref : Picked :
Shop : Tiny sprl Date Ordered : 2009/03/18 Paid :

Customer : Bank Wealthy and sons ?Ordering Contact : Arthur Grosbonnet, 1 rue Rockfell
Invoice Address : Arthur Grosbonnet, 1 rue Rockfell Shipping Address : Arthur Grosbonnet, 1 rue Rockfell
Pricelist : Public Pricelist (EUR) Analytic Account :

Sales order lines (1/2)

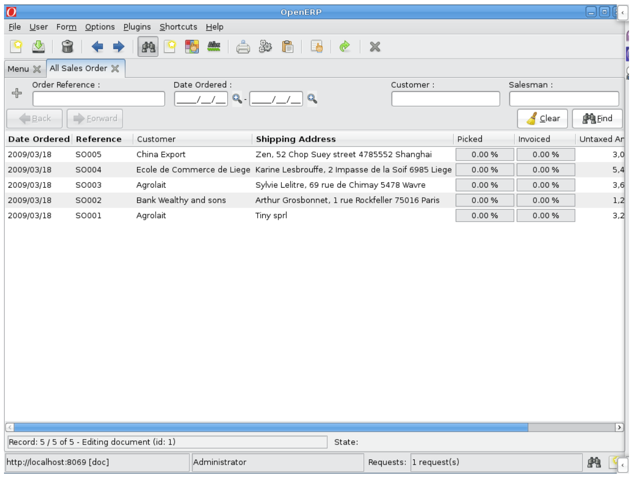
Description	Qty	UoM	Discount (%)	Unit Price	Net Price	Subtotal
[PC2] Computer assembled on demand	1.00	PCE	0.00	540.00	540.00	540.00
[MB1] Mainboard ASUStek (purchased on order)	8.00	PCE	0.00	88.00	88.00	704.00

Untaxed Amount : 1244.00 Taxes : 0.00 Total : 1244.00
?Order State : Quotation

Record: 1 / 2 of 2 - Editing document (id: 2) State:
p://localhost:8069 [doc] Administrator Requests: No request

10.3 Tree views

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



Order Reference : Date Ordered : ____/__/__ - ____/__/__ Customer : Salesman :

Date Ordered	Reference	Customer	Shipping Address	Picked	Invoiced	Untaxed Ar
2009/03/18	SO005	China Export	Zen, 52 Chop Suey street 4785552 Shanghai	0.00 %	0.00 %	3,0
2009/03/18	SO004	Ecole de Commerce de Liege	Karine Lesbrouffe, 2 Impasse de la Soif 6985 Liege	0.00 %	0.00 %	5,4
2009/03/18	SO003	Agrolait	Sylvie Lelitre, 69 rue de Chimay 5478 Wavre	0.00 %	0.00 %	3,6
2009/03/18	SO002	Bank Wealthy and sons	Arthur Grosbonnet, 1 rue Rockfeller 75016 Paris	0.00 %	0.00 %	1,2
2009/03/18	SO001	Agrolait	Tiny sprl	0.00 %	0.00 %	3,2

Record: 5 / 5 of 5 - Editing document (id: 1) State:
http://localhost:8069 [doc] Administrator Requests: 1 request(s)

The different options of those views will be detailed into the next section.

10.4 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 open 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 seperate menu item):

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

Then, the user will be able to switch from one view to the other. Unlike forms and trees, Tiny ERP 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.

10.4.1 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">
```

10.4.2 Field tag

The first field is the X axis. The second one is the Y axis and the optionnal 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 each other field, it will apply an operator
- **operator: the operator to apply is another field is grouped. By default it's '+'. 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 wich is a postgresql view
2. Create a tree view and a graph view on this object

You can get en example in all modules of the form: report_.... Example: report_crm.

10.5 Design Elements

The common structure to all the XML files of Tiny ERP is described in the DataLoadXML "Data Loading Using XML Files" section

The files describing the views are also of the form:

Example

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

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 precise groups for whom the menu is accessible using the groups attribute in menuitem tag.

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

Example

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

Note that you should add an id attribute on the menuitem which is refered 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 view with the lower priority.

10.5.1 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
- **rowspan**: 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.

10.5.2 Data Elements

Field

attributes for the "field" tag

- **select="1"**: mark this field as being one of the research criteria for this resource search view.
- **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 supersede 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.
- **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
- **on_change: define a function that is called when the content of the field changes.** – Example: on_change="onchange_partner(type,partner_id)"
  - See ViewsSpecialProperties 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, butto**
  Format: "{'attribute':[('field_name','operator','value'),('field_name','operator','value')],'attribute2':[('field_name','opera
  - where attribute will be readonly, invisible, required
  - Default value: {}.
  - Example: (in product.product)

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

Example

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

Example

```
<?xml version="1.0"?>
<terp>
    <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 &amp; 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_li
                                </page>
```

Button

<button/>: add a button using the string attribute as label. When clicked, it can trigger methods on the object, workflow transitions or actions (reports, wizards, ...).

- string: define the button's label
- confirm: the message for the confirmation window, if needed. Eg: confirm="Are you sure?"
- **name: the name of the function to call when the button is pressed. In the case it's an object function, it must take 4 argum**
  cr is a database cursor
  - uid is the userID of the user who clicked the button
  - ids is the record ID list
  - **args is a tuple of additional arguments
- states: a comma-separated list of states (from the state field or from the workflow) in which the button must appear. If the states attribute is not given, the button is always visible.
- **type: this attribute can have 3 values** – "workflow" (value by default): the function to call is a function of workflow
  - "object": the function to call is a method of the object
  - "action": call an action instead of a function

Example

```
<button name="order_confirm" states="draft" string="Confirm Order" icon="gtk-execute"/>
```

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/>
```

10.6 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>
```

The inheritance engine will parse the existing view and search for the 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 for the matching node, including the same attributes (unless position).

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

You can use these values in the position attribute:

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

Second 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">
         <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 by one ‘relation_ids’ field.

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

To delete a field from a form, an empty element with position=”replace” atribute is used. Example:

```
<record model="ir.ui.view" id="view_partner_form3">
    <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">
         <field name="lang" position="replace"/>
    </field>
</record>
```

Take into account that only one position="replace" attribute can be used per inherited view so multiple inherited views must be created to make multiple replacements.

10.7 Events

10.7.1 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" select="1" 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}

context in *<record model="ir.actions.act_window" id="a">* you can add a context field, which will be pass to the action.

See the example below:

```
<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>
```

view_type:

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

view_mode:

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

10.7.2 Getting Defaults

Description

Get back the value by default for one or several fields.

Signature: def default_get(self, cr, uid, fields, form=None, reference=None)

Parameters:

- fields: the fields list which we want to recover the value by default.
- form (optional): TODO
- reference (optional): TODO

Returns: dictionary of the default values of the form {'field_name': value, ... }

Example:

```
self.pool.get('hr.analytic.timesheet').default_get(cr, uid, ['product_id','product_uom_id'])
```

`default_set`

Description

Change the default value for one or several fields.

Signature: def default_set(self, cr, uid, field, value, for_user=False)

Parameters:

- field: the name of the field that we want to change the value by default.
- value: the value by default.
- for_user (optional): boolean that determines if the new default value must be available only for the current user or for all users.

Returns: True

Example:

```
TODO
```

CHAPTER

ELEVEN

MENU AND ACTIONS

11.1 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="<integer>"/>
```

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 Open ERP 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_REDO, STOCK_REFRESH, 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*").

11.1.1 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" a
```

11.2 Actions

11.2.1 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'.

11.2.2 Example of events

In Open ERP, 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 filed 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

11.2.3 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').

11.2.4 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.

11.2.5 Window Action

Actions are explained in more detail in section "Administration Modules - Actions". 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.

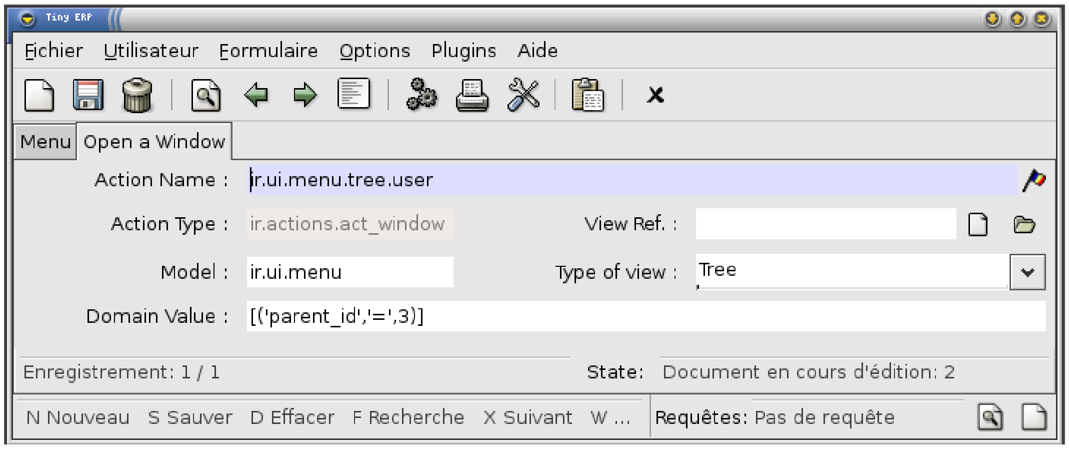
- **view_mode is only considered if view_type is form, and ignored otherwise. The four possibilities are :** – **form,tree** : the view is first displayed as a form, the list view can be displayed by clicking the "alternate view button" ;
  - **tree,form** : the view is first displayed as a list, the 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.

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


Tiny ERP
Fichier Utilisateur Formulaire Options Plugins Aide
Menu
Open a Window
N Nouveau S Sauver D Effacer F Recherche X Suivant W ...

Action Name : ir.ui.menu.tree.user
Action Type : ir.actions.act_window
View Ref. :
Model : ir.ui.menu
Type of view : Tree
Domain Value : [('parent_id','=',3)]
Enregistrement: 1 / 1
State: Document en cours d'édition: 2
Requêtes: Pas de requête

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>
```

11.2.6 Url Action

11.2.7 Wizard Action

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

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

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"?>
<terp>
     <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>
</terp>
```

11.2.8 Report Action

Report declaration

Reports in Open ERP are explained in chapter "Reports Reporting". Here's an example of a XML file that declares a RML report :

```
<?xml version="1.0"?>
<terp>
    <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>
</terp>
```

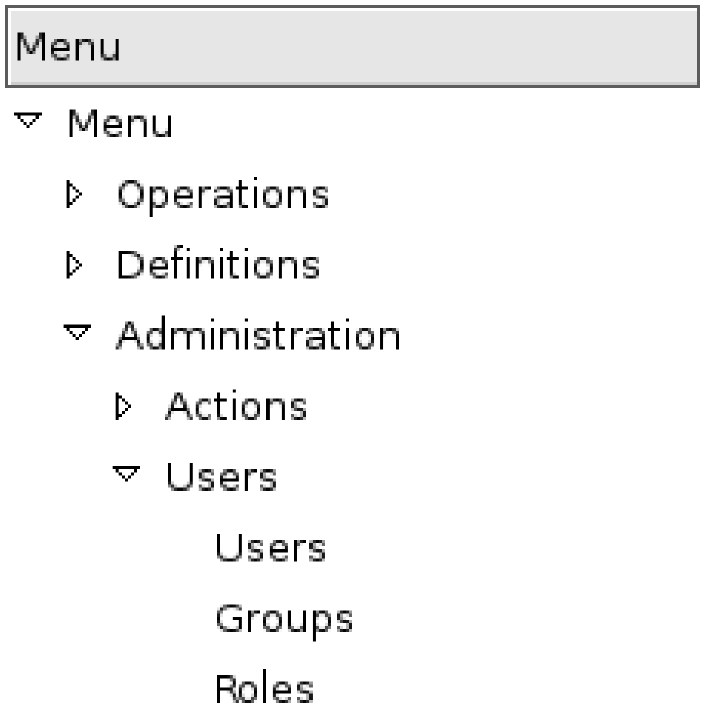
A report is declared using a **report tag** inside a “data” block. The different arguments of a report tag are :

- **id** : an identifier which must be unique.
- **string** : the text of the menu that calls the report (if any, see below).
- **model** : the Open ERP 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.

11.3 Security

Three concepts are differentiated into Tiny ERP;

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 Tiny ERP. To do that, he clicks on this menu: User > Preferences. The Open ERP 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
  - ...

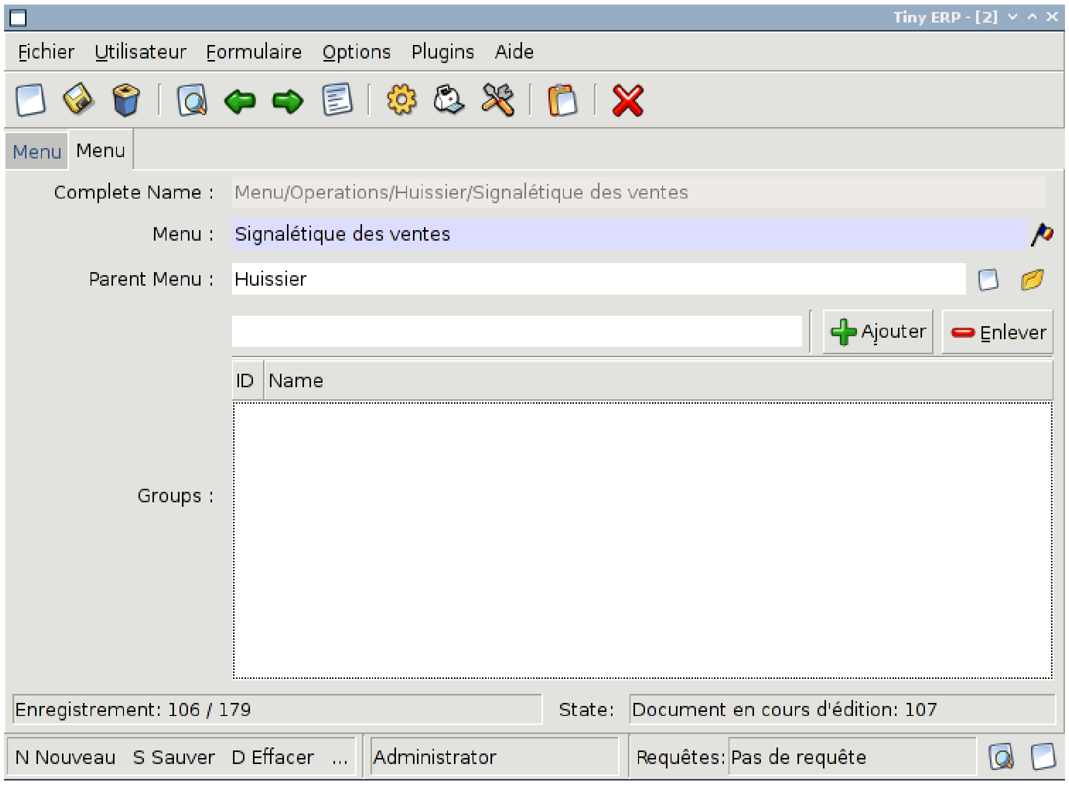
If we want to validate the test of a program (=role Testers), it may be done by a user having one of the following roles: Testers, Chief of the project, Technical manager, CEO.

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.

11.3.1 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.


Tiny ERP - [2]
Fichier Utilisateur Formulaire Options Plugins Aide
Menu Menu
Ajouter
Enlever
ID Name
N Nouveau S Sauver D Effacer ...

Complete Name : Menu/Operations/Huissier/Signalétique des ventes

Menu : Signalétique des ventes

Parent Menu : Huissier

Groups :

Enregistrement: 106 / 179 State: Document en cours d'édition: 107

Administrator Requêtes: Pas de requête

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

Part IV

Part 3 : Business Process Development

CHAPTER

TWELVE

WORKFLOW-BUSINESS PROCESS

12.1 Introduction

The workflow system in Open ERP 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 Tiny ERP very flexible and allows it to easily support changing needs without having to program new functionalities.

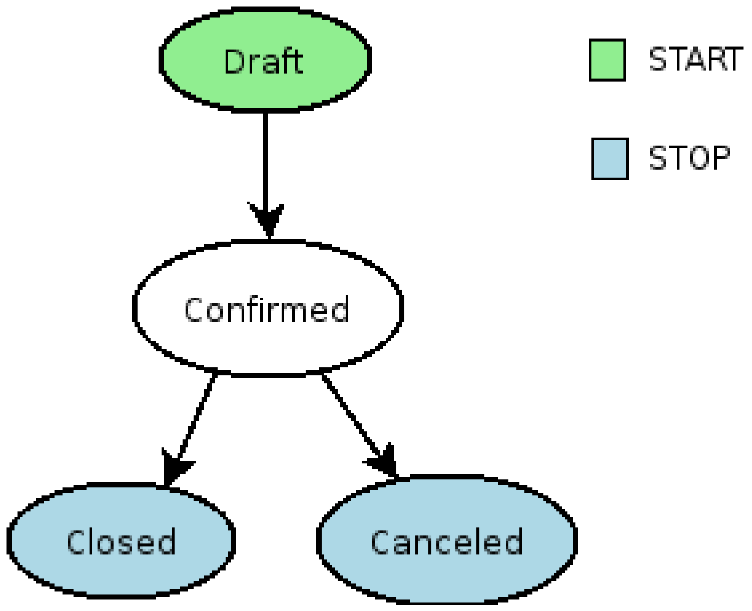
Goals

- 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 its utility, see these three examples:

12.1.1 WkfExample1: Discount On Orders

The first diagram represent a very basic workflow of an order:



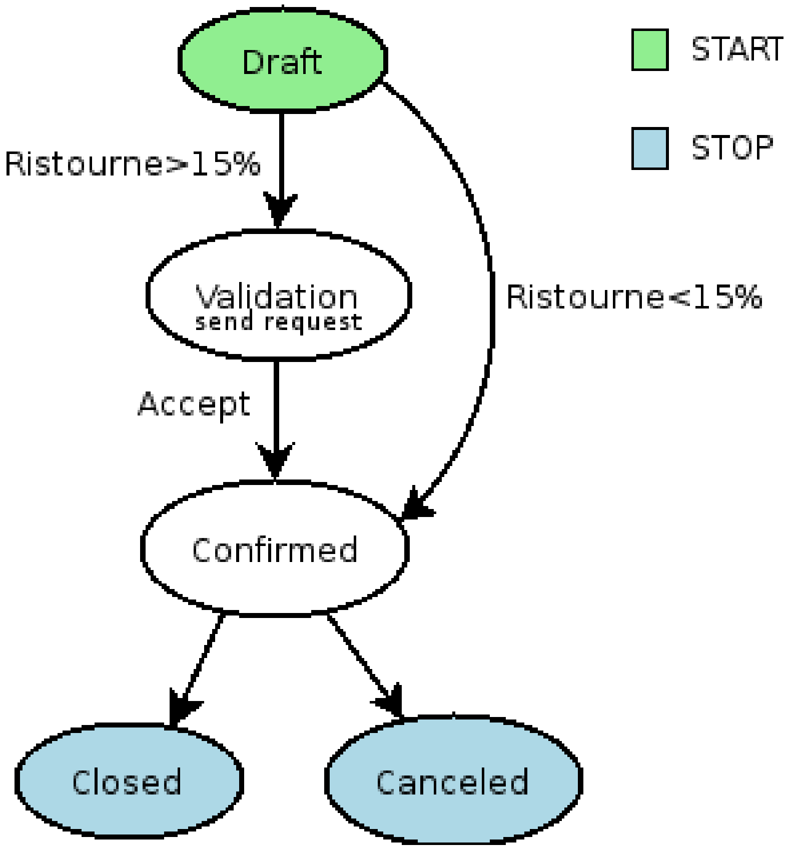
The order starts in the 'draft' state, when it is in redaction and not approved. When the user press on the 'Confirm' button, the invoice is created and the order comes into the 'CONFIRMED' state.

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, suppose 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.

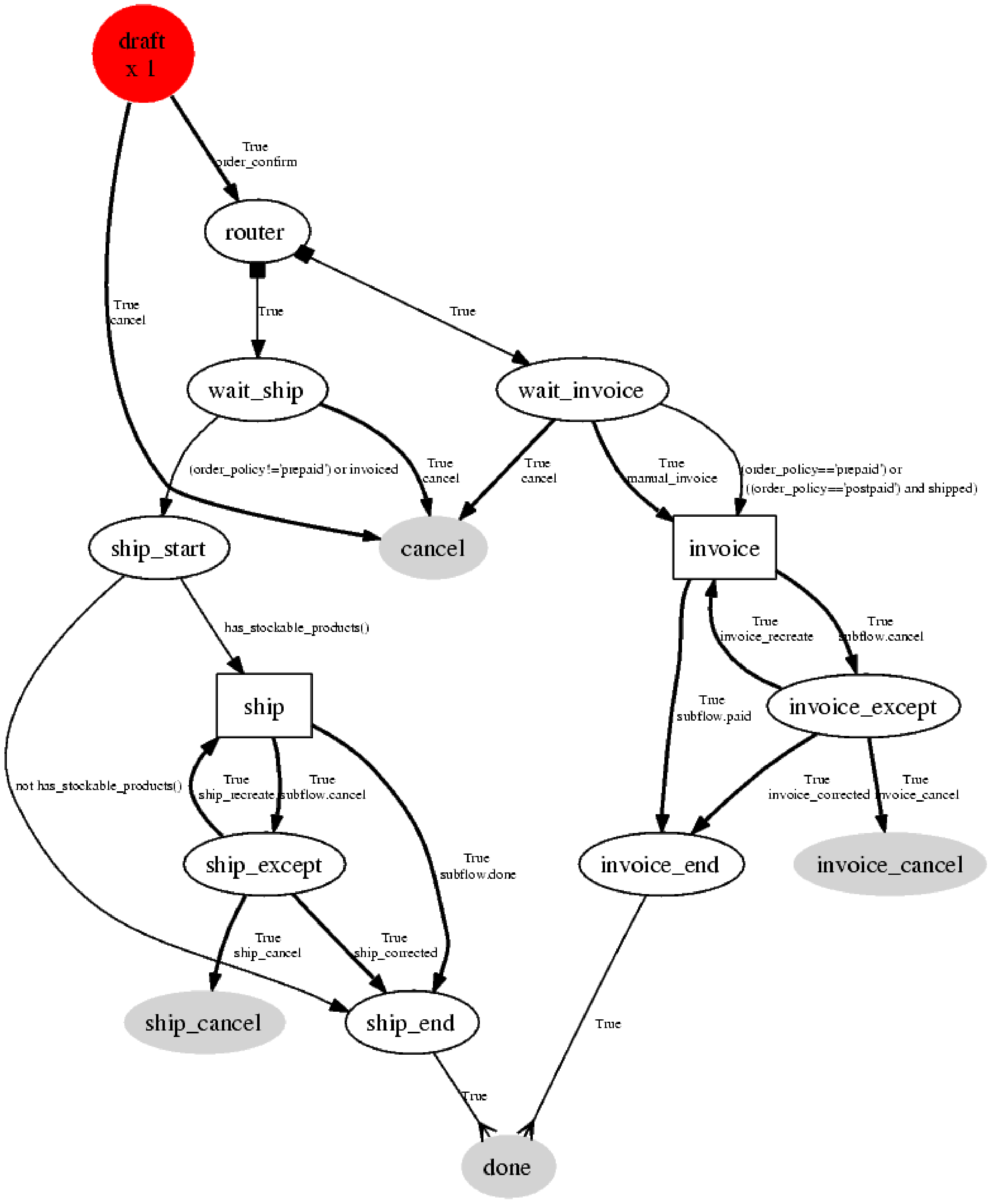
This modification in the sale logic doesn't need any line of python code! A simple modification of the workflow allows us to take this new need into account and add the extra validation step.



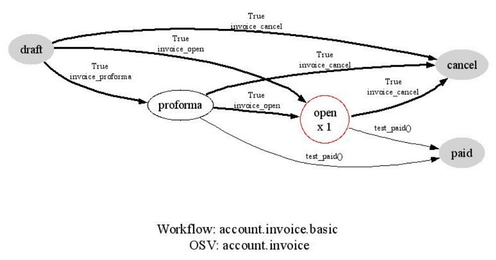
The workflow is thus modified as above and the orders will react as we want to. We then only need to modify the order form view and add a validation button at the desired location.

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 not the order.

12.1.2 WkfExample2: A sale order that generates an invoice and a shipping order.



WkfExample3: Acount invoice basic workflow



12.2 Defining Workflow

Workflows are defined in the file server/bin/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 that are necessary to describe a workflow :

- workflow : the workflow,
- wkf_activity : the activities (nodes),
- wkf_transition : the transitions between the activities.

12.3 General structure of a workflow XML file

The general structure of a workflow XML file is as follows :

```
<?xml version="1.0"?>
<terp>
<data>
<record model="workflow" id=workflow_id>

    <field name="name">workflow.name</field>
    <field name="osv">resource.model</field>
    <field name="on_create">True | False</field>

</record>

</data>
</terp>
```

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 Open ERP syntax of “dotted names”.
- **osv** (here “resource.model”) is the name of the Tiny object we use as a model [-(Remember a Open 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">True</field>

</record>
```

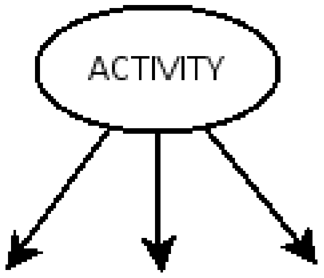
12.4 Activity

12.4.1 Introduction

The wkf_activity class represents the nodes of workflows. These nodes are the actions to be executed.

12.4.2 The fields

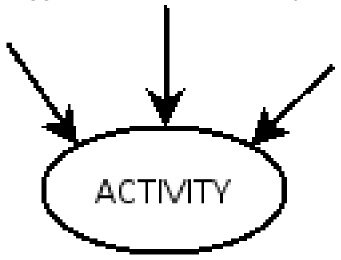
12.4.3 split_mode



- 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 join_mode:



- **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.

12.4.4 kind:

The type of the activity can take several 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.

12.4.5 action:

The action indicates the method to execute when a workitem comes into this activity. The method must be defined in a object which belongs 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`

`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.

Be warned to not use this flag unless your activity really is a "flow start". There are tiny versions that do not care about the tags contents like "true" or "false". Using such tag and tiny version, you will always end up whith an activity which is tagged as "flow start = true", leaving u with a nasty hunt to find out where your workflowdesign could be wrong.

12.4.6 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.

Be warned to not use this flag unless your activity really is a "flow stop". There are tiny versions that do not care about the tags contents like "true" or "false". Using such tag and tiny version, you will always end up whith an activity which is tagged as "flow stop = true", leaving u with a nasty hunt to find out where your workflowdesign could be wrong.

12.4.7 wkf_id

The workflow which this activity belongs to. Defining activities using XML files

12.4.8 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">True | False</field>
      <field name="flow_stop">True | False</field>
  </record>
```

The first two arguments **wkf_id** and name are mandatory. Be warned to not use **flow_start** and **flow_stop** unless your activity really is a **flow start** or **flow_stop**. There are tiny versions that do not care about the tags contents like "True" or "False".

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/bin/addons/sale/sale_workflow.xml** for several examples of activity definitions.

12.5 Transition

12.5.1 Introduction

Workflow transitions are the conditions to be satisfied to go from one activity to the next one. They are represented by one-way arrows joining two activities.

The conditions are of different types:

- role to satisfy by the user
- 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. 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. 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.

12.6 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

12.7 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 possess the necessary 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 developper
    - Developpers
    - 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 developper, Technical manager, CEO.

12.8 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 terminated activities are executed. Other transactions are "rolled back".

12.9 Creating a Workflow

Steps for creating a simple state-changing workflow for a custom module called **mymod**

12.9.1 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),
}
```

12.9.2 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! 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">True</field>
</record>
```

2. Workflow Activity records

   These define the actions that should 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">True</field>
        <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">True</field>
</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">True</field>
</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>
```

Add mymod_workflow.xml to __terp__.py

Edit your module's __terp__.py and add mymod_workflow.xml to the "update_xml" array, so that OpenERP picks it up next time your module is loaded. 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>
```

12.9.3 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 Administration -> Customisation -> Workflow Definitions

When you are testing, remember that the workflow will only apply to NEW records that you create.

12.9.4 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

CHAPTER

THIRTEEN

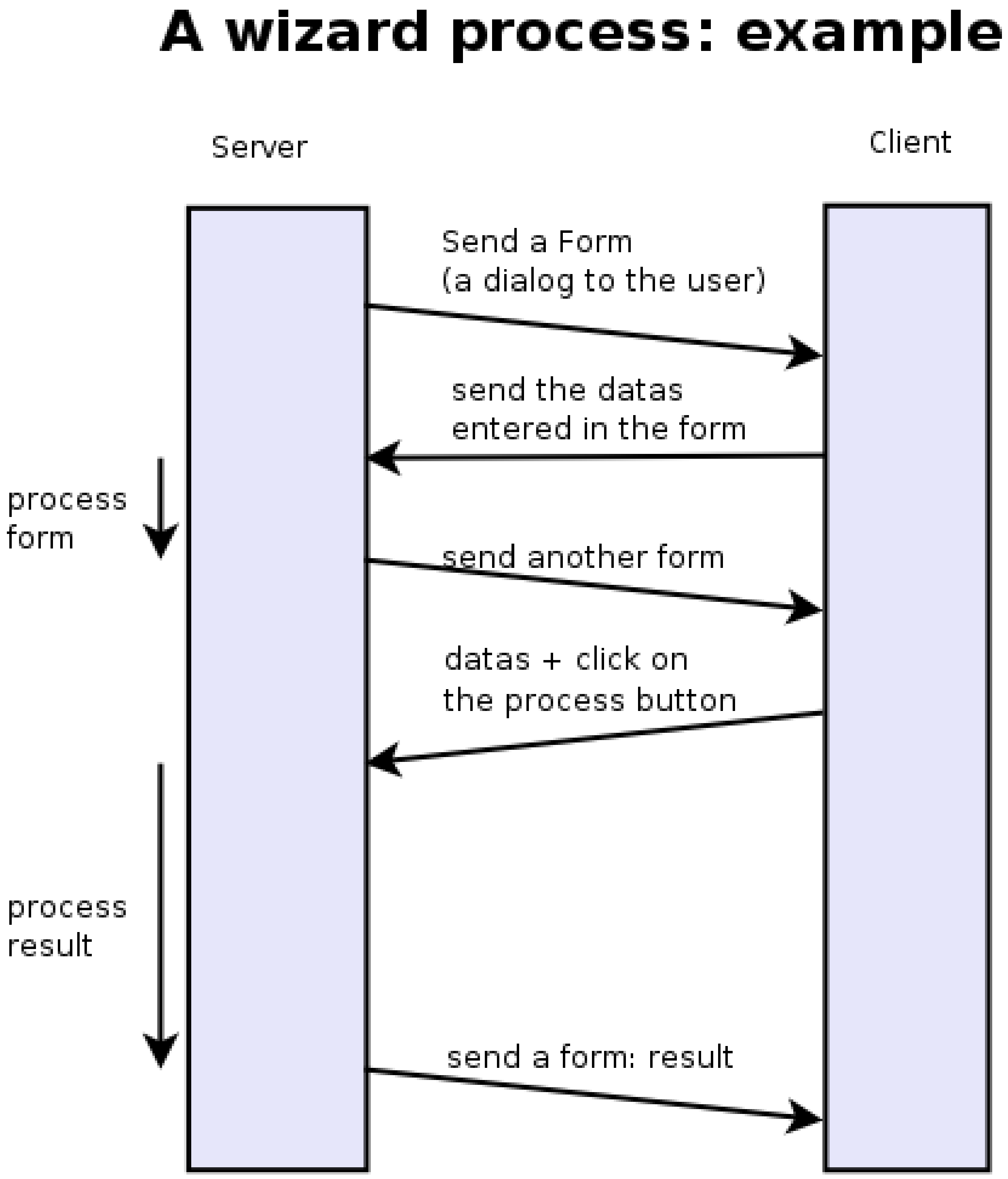
CREATING WIZARD - (THE PROCESS)

13.1 Introduction

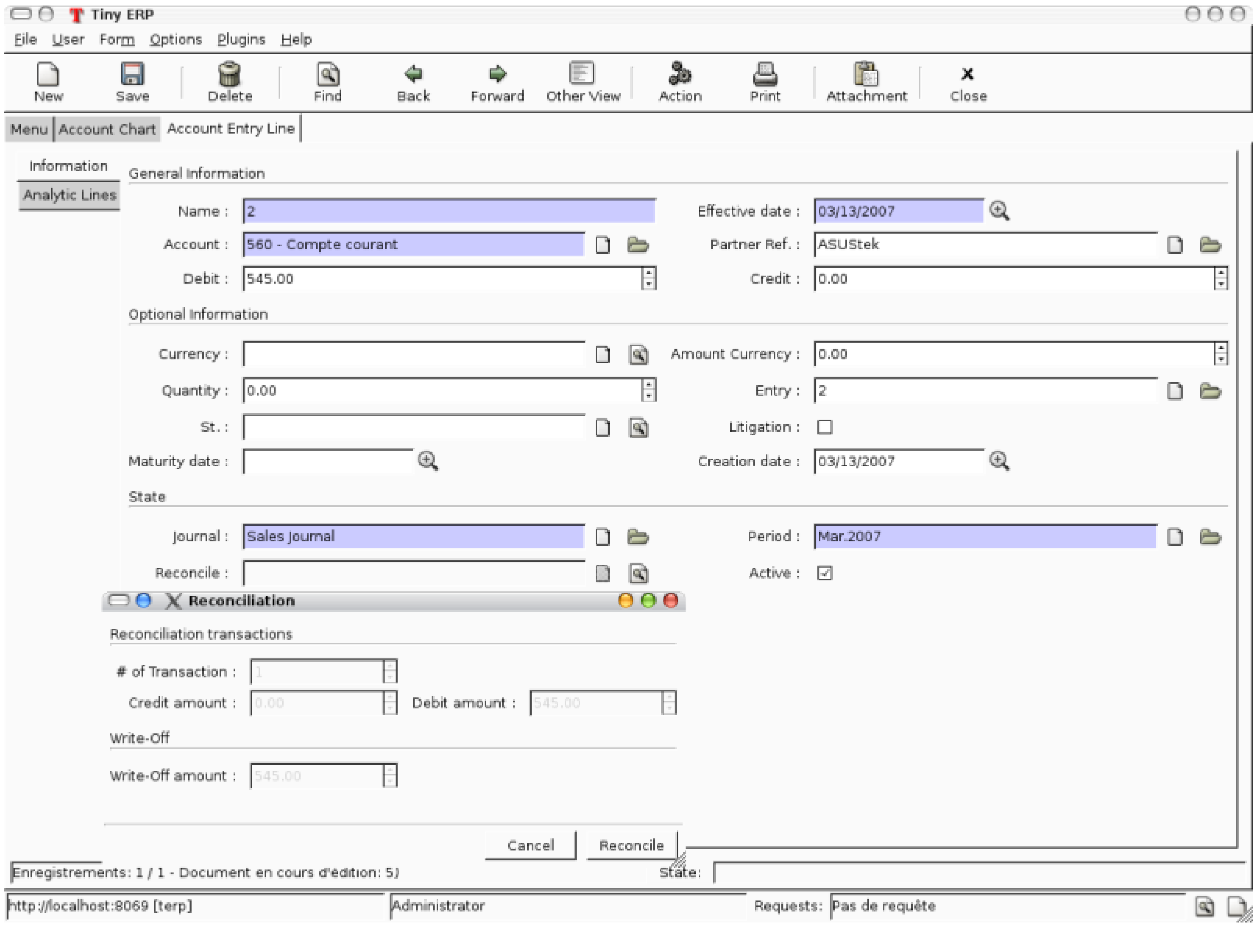
Wizards describe interaction sequences between the client and the server.

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



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
Analytic Lines
Reconciliation
Cancel Reconcile

General Information

Name : 2
Account : 560 - Compte courant
Debit : 545.00
Effective date : 03/13/2007
Partner Ref. : ASUStek
Credit : 0.00

Optional Information

Currency :
Quantity : 0.00
St. :
Maturity date :
Amount Currency : 0.00
Entry : 2
Litigation : ☐
Creation date : 03/13/2007

State

Journal : Sales Journal
Reconcile :
Period : Mar.2007
Active : ☑

Reconciliation transactions

# of Transaction : 1
Credit amount : 0.00
Debit amount : 545.00

Write-Off

Write-Off amount : 545.00

Enregistrements: 1 / 1 - Document en cours d'édition: 5)
State:
http://localhost:8069 [terp]
Administrator
Requests: Pas de requête

13.2 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.

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

13.2.1 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.

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 wich 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"/>
          <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},
      'writeoff': {'string':'Write-Off amount', 'type':'float', 'readonly':True},
      'writeoff_acc_id': {'string':'Write-Off account',
                          'type':'many2one',
                          'relation':'account.account'
                         },
}
```

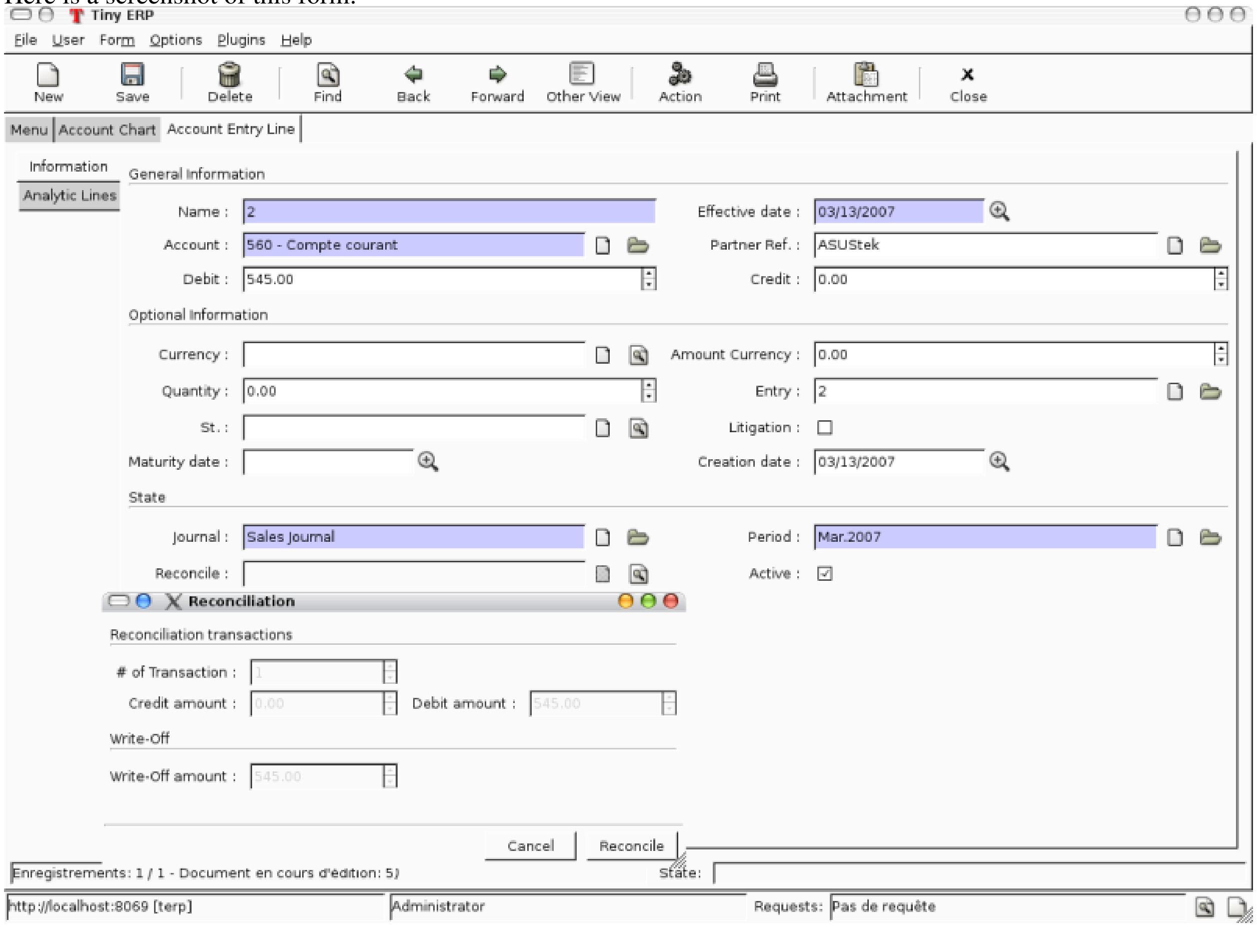
Each step/state of a wizard can have several buttons. Those are located on the bottom right of the dialog box. The list of buttons for each step of the wizard is declared in the state key of its result dictionary.

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:


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
Analytic Lines
Cancel
Reconcile

General Information

Name : 2
Effective date : 03/13/2007
Account : 560 - Compte courant
Partner Ref. : ASUStek
Debit : 545.00
Credit : 0.00

Optional Information

Currency :
Amount Currency : 0.00
Quantity : 0.00
Entry : 2
St. :
Litigation : ☐
Maturity date :
Creation date : 03/13/2007

State

Journal : Sales Journal
Period : Mar.2007
Reconcile :
Active : ☑

Reconciliation

Reconciliation transactions

# of Transaction : 1
Credit amount : 0.00
Debit amount : 545.00

Write-Off

Write-Off amount : 545.00

Enregistrements: 1 / 1 - Document en cours d'édition: 5)
State:
http://localhost:8069 [terp]
Administrator
Requests: Pas de requête

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 an 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'}
```

13.3 Specification

13.3.1 Form

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

13.3.2 Fields

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)

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

13.4 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"?>
<terp>
    <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>
</terp>
```

Attributes for the wizard tag:

- **id** (optional):
- **string**: The string which will be displayed if there are several wizards for one resthe user will be presented a list with 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 attribut**
  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.

__terp__.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 __terp__.py file of the module.

Here is, for example, the **__terp__.py** file for the account module.

```
{
    "name": Open ERP 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"],
}
```

13.5 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.

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>
```

CHAPTER

FOURTEEN

REPORTS

There are mainly three types of reports in Open ERP:

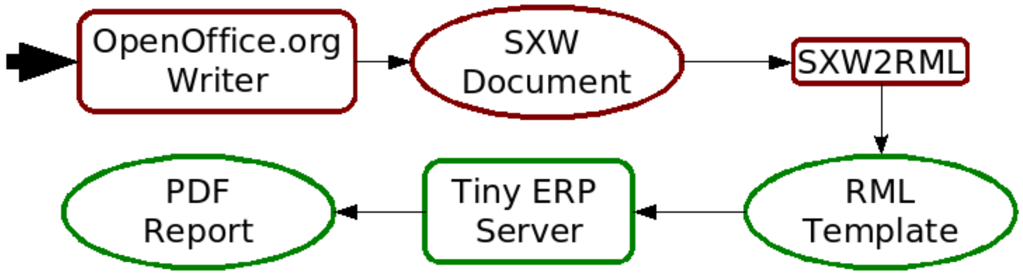
- OpenOffice.org reports
- RML reports
- custom reports (based on PostgreSQL views and displayed within the interface)

This chapter mainly describes OpenOffice.org reports, and then XSL:RML reports. Custom reports are described in section Advanced Modeling - Reporting With PostgreSQL Views.

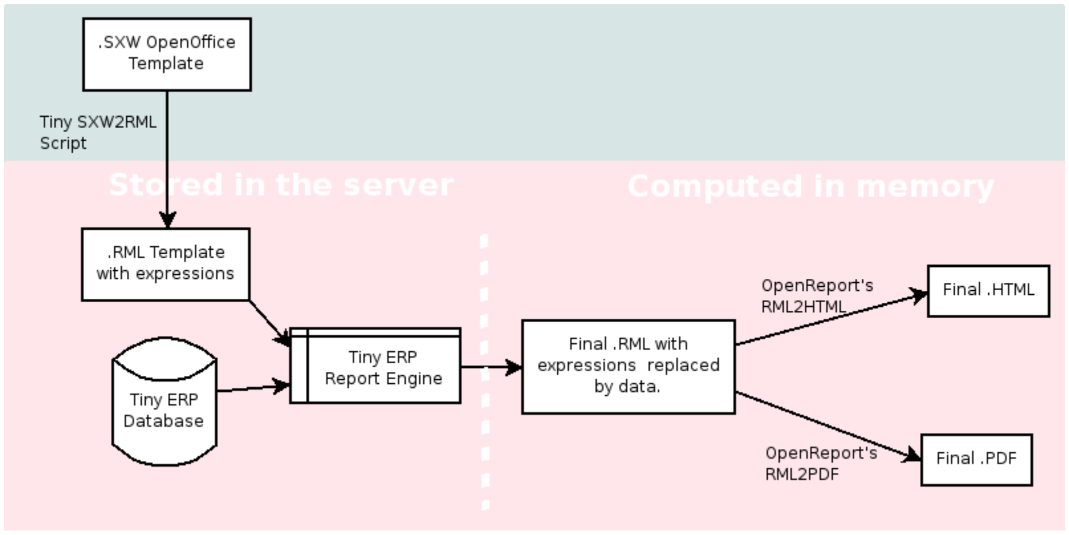
14.1 OpenOffice.org reports

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 Open ERP server will be filled in. This document is only used for developers, as a help-tool to easily generate the .RML file. Open ERP 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.

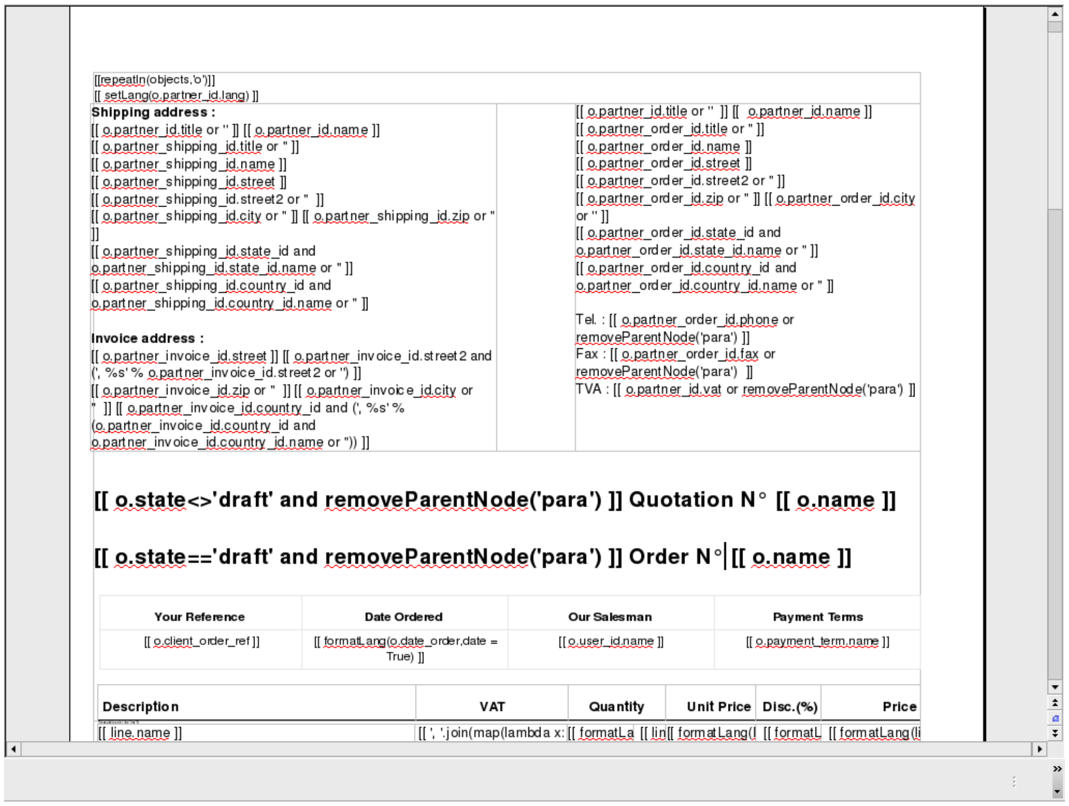
in the .RML file will be replaced by the name of the country of the partner of the printed invoice. This report engine produce the same .RML file where all expressions have been replaced by real data.

The final document

- Finaly the .RML file is converted to PDF or HTML according to the need, using OpenReport's scripts.

14.1.1 Creating a SXW

You can design reports using *OpenOffice*. Here's, as an example, the file @@server/bin/addons/sale/report/order.sxw@@.



[[repeatIn(objects,'o')]]
[[setLang(o.partner_id.lang)]]

Shipping address :
[[o.partner_id.title or '']] [[o.partner_id.name]]
[[o.partner_shipping_id.title or '']]
[[o.partner_shipping_id.name]]
[[o.partner_shipping_id.street]]
[[o.partner_shipping_id.street2 or '']]
[[o.partner_shipping_id.city or '']] [[o.partner_shipping_id.zip or ''
]]
[[o.partner_shipping_id.state_id and
o.partner_shipping_id.state_id.name or '']]
[[o.partner_shipping_id.country_id and
o.partner_shipping_id.country_id.name or '']]

Invoice address :
[[o.partner_invoice_id.street]] [[o.partner_invoice_id.street2 and
(', %s' % o.partner_invoice_id.street2 or '')]]
[[o.partner_invoice_id.zip or '']] [[o.partner_invoice_id.city or
'']] [[o.partner_invoice_id.country_id and (', %s' %
(o.partner_invoice_id.country_id and
o.partner_invoice_id.country_id.name or ''))]]

[[o.partner_id.title or '']] [[o.partner_id.name]]
[[o.partner_order_id.title or '']]
[[o.partner_order_id.name]]
[[o.partner_order_id.street]]
[[o.partner_order_id.street2 or '']]
[[o.partner_order_id.zip or '']] [[o.partner_order_id.city
or '']]
[[o.partner_order_id.state_id and
o.partner_order_id.state_id.name or '']]
[[o.partner_order_id.country_id and
o.partner_order_id.country_id.name or '']]

Tel. : [[o.partner_order_id.phone or
removeParentNode('para')]]
Fax : [[o.partner_order_id.fax or
removeParentNode('para')]]
TVA : [[o.partner_id.vat or removeParentNode('para')]]

[[o.state<>'draft' and removeParentNode('para')]] Quotation N° [[o.name]]

[[o.state=='draft' and removeParentNode('para')]] Order N° [[o.name]]

Your Reference	Date Ordered	Our Salesman	Payment Terms
[[o.client_order_ref]]	[[formatLang(o.date_order,date = True)]]	[[o.user_id.name]]	[[o.payment_term.name]]

Description	VAT	Quantity		Unit Price	Disc.(%)	Price
[[line.name]]	[[', '.join(map(lambda x:	[[formatLa	[[lin	[[formatLang(l	[[formatL	[[formatLang(li

14.1.2 Dynamic content in your report

Dynamic content

In the .SXW/.RML reports, you can put some Python code that accesses the Open ERP 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)** : 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 optionnal third argument that is the name of the .RML tag you want to loop on.
- **setTag('para','xpre')** : change the enclosing RML tag (usually 'para') by 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
- **[[(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 formating
- **[[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.

14.1.3 SXW2RML

Open Report Manual

About

The Open ERP'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 a OpenERP module or as a standalone program.

Open Report has been developed by Fabien Pinckaers.

tiny_sxw2rml can be found at http://www.tinyreport.org/download.html

SXW to RML script setup - Windows users

In order to use the 'tiny_sxw2rml.py' Python script you need the following packages installed:

- Pyhton (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

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

14.1.4 Tiny ERP Server PDF Output

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 folder report 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)
- a boolean to add or not the company header on the report (default True)

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 **__terp__.py** file

Here is an example for the sale order report:

```
<?xml version="1.0"?>
<terp>
        <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>
</terp>
```

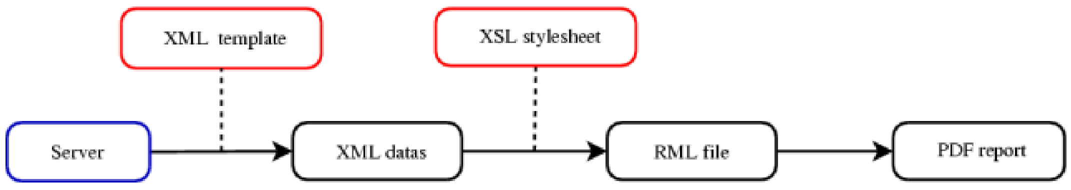
The arguments are:

- **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 located in “[server_dir]/bin/addons/custom”

14.2 XSL:RML reports

RML reports don’t require programming but require two simple XML files to be written:

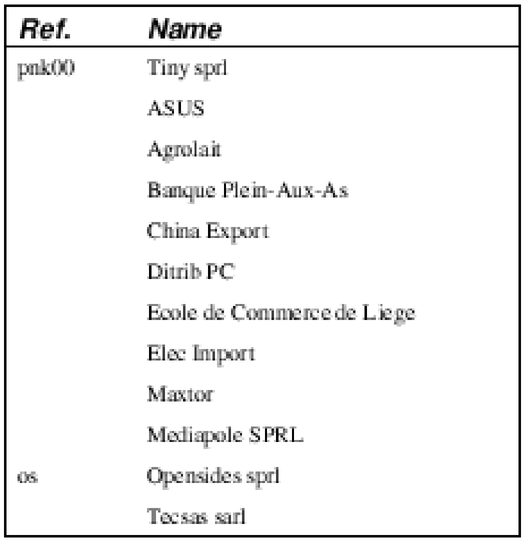
- a file describing the data to export (*.xml)
- a file containing the presentation rules to apply to that data (*.xsl)



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
	Tecsas 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>Ditrib 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 stylesheet

```
<?xml version="1.0" encoding="utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/19

    <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>

        <stylesheet>

            <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>

        </stylesheet>

        <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>

                    <td></td>
                    <td><para>Ditrib 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>
```

Fore more information on the formats used:

- RML : http://reportlab.com/docs/RML_UserGuide_1_0.pdf
- XSL - Specification : http://www.w3.org/TR/xslt
- XSL - Tutorial : http://www.zvon.org/xxl/XSLTutorial/Books/Output/contents.html

All these formats use XML:

- http://www.w3.org/XML/

14.2.1 XML Template

XML templates are simple XML files describing which fields among all available object fields are necessary for the report.

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

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 standar

<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>
```

14.2.2 Introduction to RML

For more information on the RML format, please refer to the official Reportlab documentation.

- http://www.reportlab.com/docs/RML_UserGuide.pdf

14.2.3 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:

Translations

As Open ERP can be used in several langages, 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.

Useful links:

- http://www.reportlab.com/docs/RML_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/XSLTreference/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)

Example (with corporate defaults):

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" :xmlns:fo="http://www

    <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>
```

14.3 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
     <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>
```

14.4 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

    <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>
```

14.5 Bar Codes

14.5.1 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

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>

14.6 How to add a new report

In 4.0.X

> Administration -> Custom -> Low Level -> Base->Actions -> ir.actions.report.xml

14.7 Usual TAGS

14.7.1 Code find in [[ ]] tags is python code.

The context of the code (the variable's values you can use) is the following:

python objects/variables, available when the report start:

"objects" the list of objects to be printed (invoices for example)

"data" comes from the wizard

"time" see python documentation.

"user" the user object launching the report.

python functions you can use:

"setlang('fr')" change the langage used in automated translation (fields...).

"repeatIn(list,varname)" repeat the template (whole doc. or current paragraph?) for each object in the list. Use varname in the template's tags.

"setTag('para','xpre')" change the enclosing RML tag (usually 'para') by an other (xpre is a preformatted paragraph), in the (converted from sxw)rml document (?)

"removeParentNode"

14.7.2 Useful tags:

> [repeatIn(objects,'o')] objects to be printed repeatIn(o.invoice_line,'l') print every line o.quantity * o.price Operations are OK. '07d' int(o.number) number formating 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') format=dd MM YYYY, check python doc for more about "%d", ... [[time.strftime(time.ctime()[0:10])]] [[time.strftime(time.ctime()[-4:])]] prints only date. time.ctime() it prints the actual date & time. [[time.ctime().split()[3]]] prints only time

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 fullfilled. You can construct it with the pyhton 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’ print YES or NO

it works like this:

and: first value is TRUE then print out the second value. First value is FALSE print out first value.

or: first value is TRUE then print out the first value. First value is FALSE print out second value. in this example if o.prop==’draft’ -> TRUE then **(o.prop==’draft’) and ‘YES’** reads **‘Yes’**. Next step is ‘Yes’ or ‘No’ which leads to a printed ‘YES’ (because a string’s logical value is TRUE). If o.prop==’draft’ -> FALSE then it reads FALSE or ‘No’. So ‘No’ is printed. One can use very comlpex structures. To learn more search for some pyhton reference regarding logical opertors.

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’)

14.8 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.

14.8.1 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.

14.8.2 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 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
```

CHAPTER

FIFTEEN

I18N - INTERNATIONALIZATION

Explain about the multiple language application

15.1 Introduction

Part V

Part 4 : Business Process Advance Configuration

CHAPTER

SIXTEEN

SERVER ACTION

16.1 Introduction

Server action is an new feature to the OpenERP available since the version 5.0 beta, This is the interesting features for the customizer, to full fill the customers requirements, This features enables to provides the quick and easy configuration some process which is day to day requirements. Like send email on confirmation of the sale order, or confirmation of the Invoice, log the operation of the invoice (confirm, cancel, etc..). or need to develope some system which runs wizard / report on the confirmation of the sales, purchase, or invoice. So Server action is the only one answer to solve all this kind of problems without doing any development, just a few configuration and the system is ready to answer few of above questions.

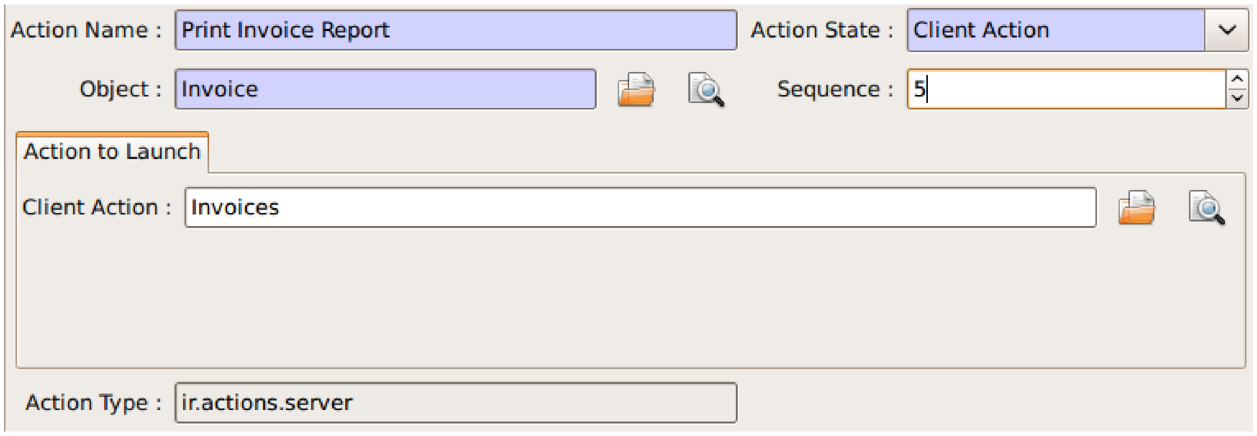
Following are the list of action types which are supplied under the Server Action.

- Client Action
- Trigger
- Email
- SMS
- Create Object
- Write Object
- Multi Action

Each type of action have the special features and different configuration parameters. We will see one by one all type of action how to configure and list of parameters that affect the system

16.2 Client Action

This action executes at client side, this is a good idea to run the wizard or report at client side. Using this type of action we can make the system like ERP will print the invoice after confirmation of the Invoice. Like it will run the payment wizard after confirmation of the invoice. Technically we can run all client action which execute at client side. We can execute ir.actions.report.custom, ir.actions.report.xml, ir.actions.act_window, ir.actions.wizard, or ir.actions.url. Here is an example to show how we can configuration Client action to print the invoice after confirmation of the invoice.



Action Name : Print Invoice Report
Action State : Client Action
Object : Invoice
Sequence : 5
Client Action : Invoices
Action Type : ir.actions.server

This is an good and seems easy to configure the action.

Important fields are

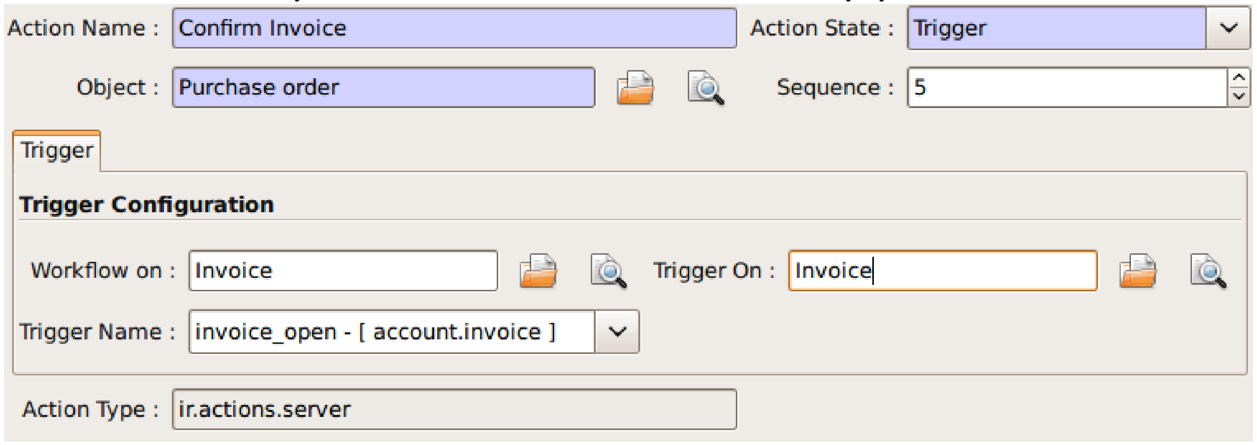
Object Select the object on which we want to implement the Server Action when work flow will execute on this object

Client Action Select the client action that is to execute at client side. Any of the following types.

- ir.actions.report.custom
- ir.actions.report.xml
- ir.actions.act_window
- ir.actions.wizard
- ir.actions.url

16.3 Trigger

Trigger is an really excellent when we want to deal with the work flow of the other object which working the work flow of the first object. For example we want to configure the system like when we confirm the purchase order and create the invoice that newly created invoice should confirm it self automatically by the server action.



Action Name : Confirm Invoice
Action State : Trigger
Object : Purchase order
Sequence : 5
Trigger Configuration
Workflow on : Invoice
Trigger On : Invoice
Trigger Name : invoice_open - [account.invoice]
Action Type : ir.actions.server

This is the easy configuration for the trigger to have the system where the created invoice will confirm it self.

Important fields are

Object Select the object on which we want to implement the Server Action when work flow will execute on this object

Work-flow on Here we select invoice, need to select the model on which the automatic workflow will be called by the action system

Trigger On We need to provide the id of the newly record, here in this case, Purchase order store the id of the Invoice after creating of the invoice in invoice_id field.

Trigger Name This is the signal name which we want to generate on the newly created object.

16.4 Email Action

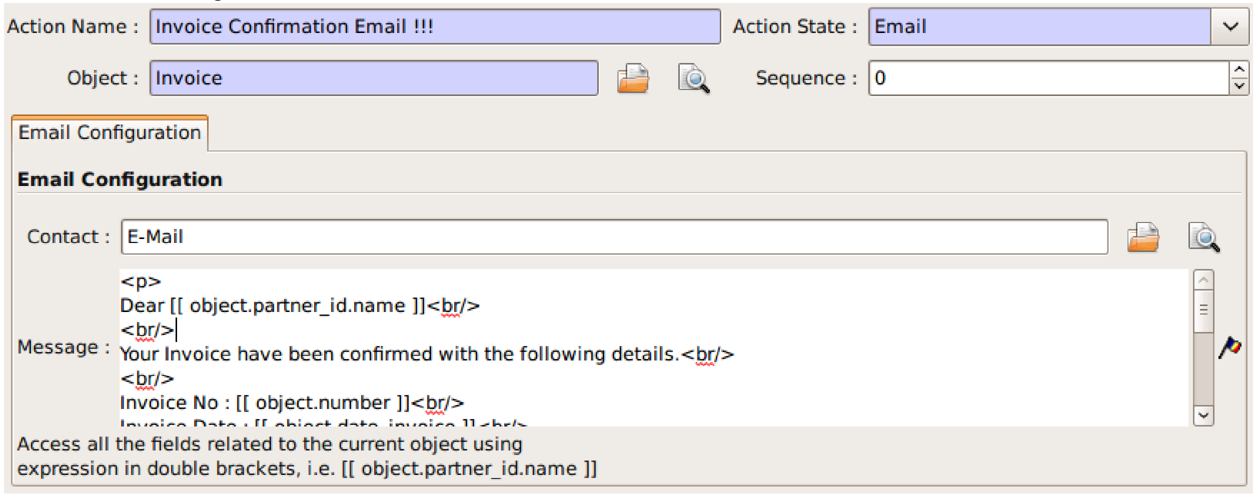
This is the common requirement for all business process, like send the confirmation by the email when sales order, purchase order, invoice, payment, shipping of goods will takes place. For that we need only few things to configure and tiny will send the email very quickly and in easy way. Even not need to setting up the your own email server, you can use your exciting email server and account, of you not have your email server you can use from the free email account by Gmail, Yahoo !, etc..

Server Configuration

supply the following parameters when we run OpenERP Server.

```
--email-from=gajjarmantavya@yahoo.co.in user email address
--smtp=smtp.mail.yahoo.co.in smtp server name or ip
--smtp-port=587 smtp port
--smtp-user=gajjarmantavya user name usually same as the email address name without domain name
--smtp-password=************* password to the user account
--smtp-ssl=False use in case if the server required ssl for sending email
```

Email Action Configuration


Action Name : Invoice Confirmation Email !!!
Action State : Email
Object : Invoice
Sequence : 0

Email Configuration

Contact : E-Mail

Message :
```
<p>
Dear [[ object.partner_id.name ]]<br/>
<br/>
Your Invoice have been confirmed with the following details.<br/>
<br/>
Invoice No : [[ object.number ]]<br/>
```

Access all the fields related to the current object using expression in double brackets, i.e. [[object.partner_id.name]]

Important Fields are

Object Select the object on which we want to implement the Server Action when work flow will execute on this object

Contact We need to select the fields from which action will select the email address to whom we would like to send the email, system will display all the fields related to the current object selected in the Object field

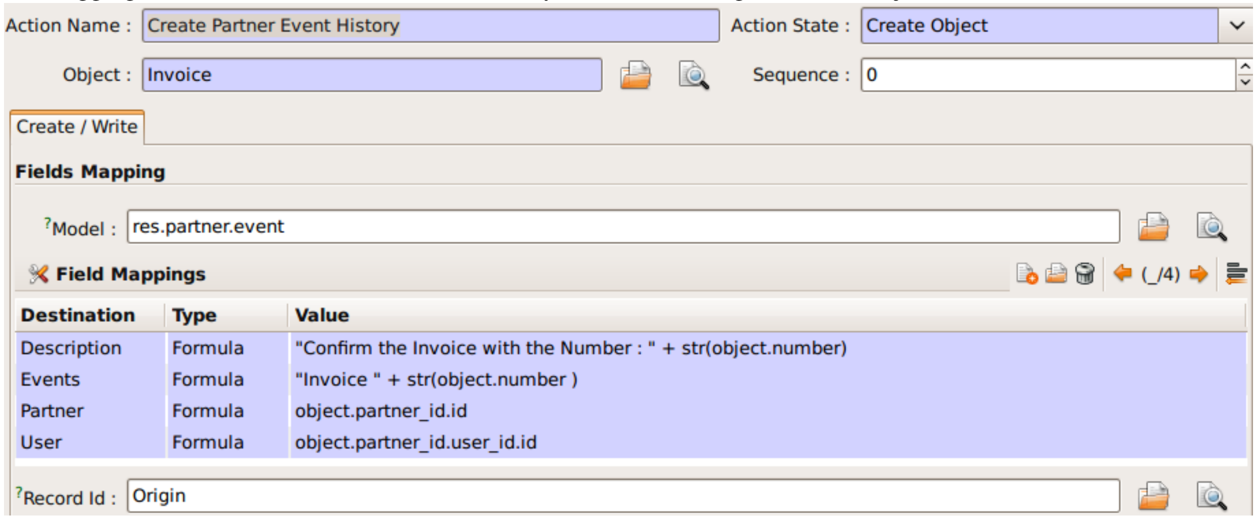
Message You can provide the message template with the fields that related to the current object. And it will be merge when it is going to send the email. This is the same language then the rml which is used to design the report here we can use the [[ ]] + html tage to design in the html format Working with You can select the any fields from the current object, like here we select the [[ ]] invoice in the object.

For example to get the partner name we can use [[object.partner_id.name]]like the same, object refers to the current object and we can access any fields which exist in the model.

After confirmation the invoice we get the confirmation email from the action.

16.5 Create Object

This is an interesting feature for the tiny partners those who want to track the transaction in the OpenERP, like currently in the ERP you can get the Event history on the Partners which logs the only the sales order events. But if we want to start logging the invoice like the same we can easily do like that using the Create object Actions.



Action Name :	Create Partner Event History	Action State :	Create Object
Object :	Invoice	Sequence :	0

Create / Write

Fields Mapping

Model : res.partner.event

Field Mappings (_/4)

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

Record Id : Origin

Create Object action have the easy but tricky configuration, for the movement you have to remember the fields name or check it out from the code it self, in future we will develop the expression builder inside OpenERP so you can build the complex expression.

Important fields are

Object Select the object on which we want to implement the Server Action when work flow will execute on this object

Model This is the target model where the new object is to be created, if its empty it refers to the current object and allow to select the fields from the same, but its advisable to provide the model in all case if different or if the same.

Fields Mapping Need to provide the 3 values

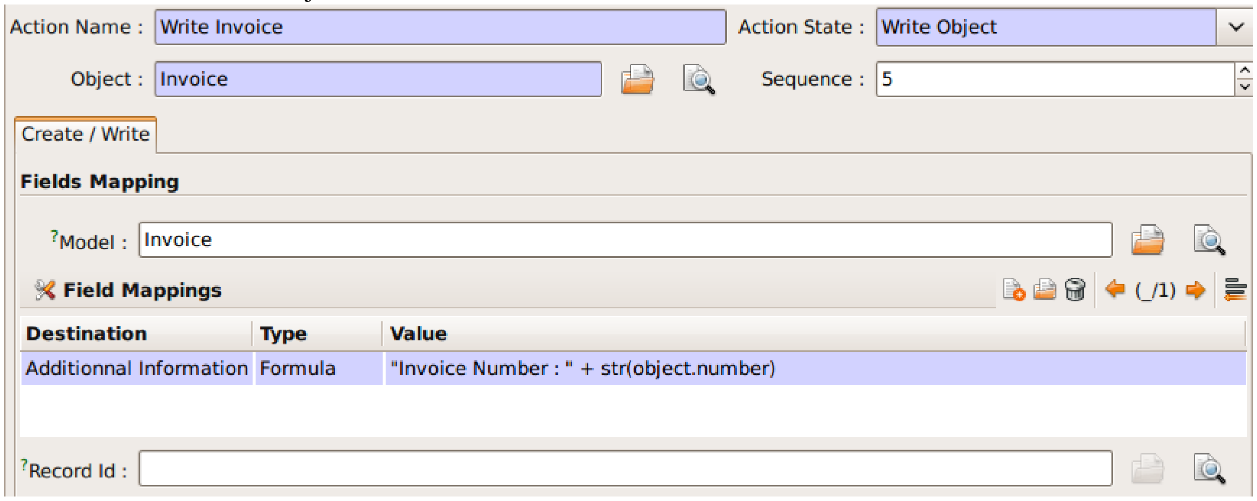
1. Field: any of the fields from the target model
2. type of the value you can give either value or expression
3. provide the value or expression the expression again start with the 'object' keyword and its refers to the current object which selected in to the Object field.

You must select the all required fields from the object

Record Id After creating the new record where the id of the new record if going to store. So in future we can refer the same for the other operations.

16.6 Write Object

The same configuration as defined for the Create Object, here we take an example that it will write the 'Additional Information' on the same object



Action Name :	Write Invoice	Action State :	Write Object
Object :	Invoice	Sequence :	5

Create / Write

Fields Mapping

?Model : Invoice

Field Mappings (_/1)

Destination	Type	Value
Additionnal Information	Formula	"Invoice Number : " + str(object.number)

?Record Id :

Important Fields are

same as the Create Object

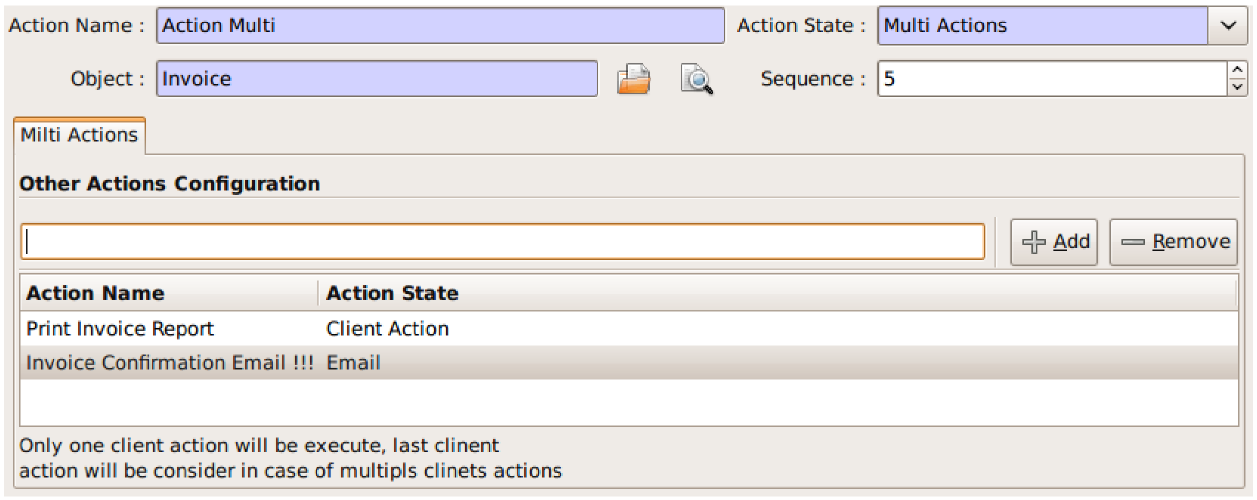
16.7 Multi Action

This is the most interesting action, which allows to execute the multiple server action on the same business operations. Like if you want to print and send the email on confirmation of the invoice. We need to create the 3 Server Actions for that.

- Print Invoice
- Invoice Confirmation Email !!
- Multi Action

The only problem with the Multi Action is that it will execute many actions at the server side, but only one client action will be executed.

For example we would like to print report + execute the wizard this 2 operation is not allowd in the one multi action.



Action Name : Action Multi
Action State : Multi Actions
Object : Invoice
Sequence : 5

Milti Actions

Other Actions Configuration

Action Name	Action State
Print Invoice Report	Client Action
Invoice Confirmation Email !!!	Email

Only one client action will be execute, last clinent
action will be consider in case of multipls clinets actions

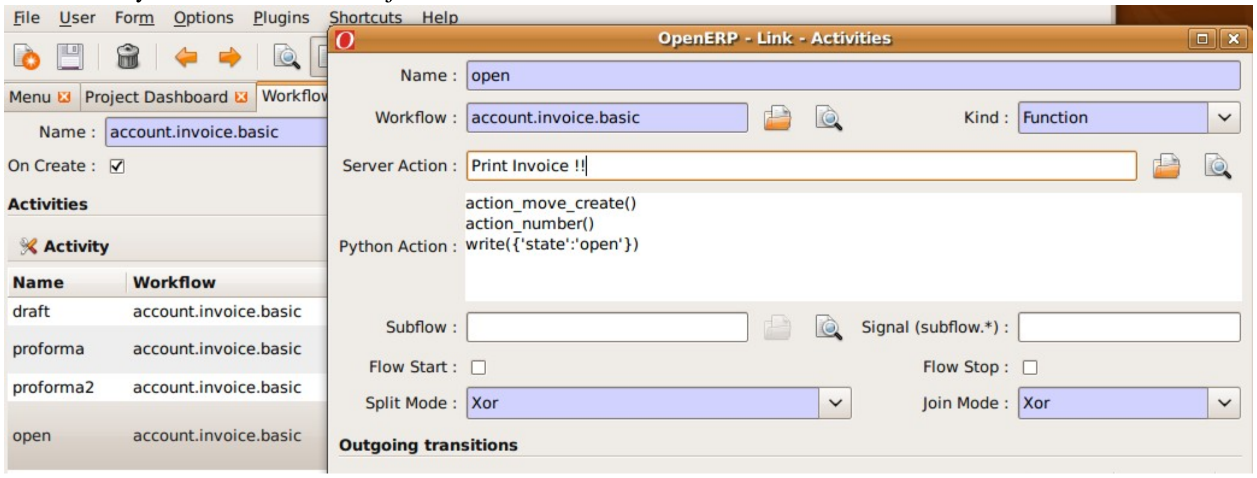
Important Fields are

Object Select the object on which we want to implement the Server Action when work flow will execute on this object

Other Actions We need to select the server action in this fields, we are free to select the as many as actions as we can. Just we need to take care for the problem of the multi action, other things is very easy.

Link it up with the Work flow

The important things is to link the server action with the work flow, its bit easy to link with action with the work flow. Open the work flow editor in GTK, select the work flow and go to the start and select the Sever Action. This will automatically be called when the object comes to that state.



File User Form Options Plugins Shortcuts Help
Menu
Project Dashboard
Workflow
OpenERP - Link - Activities

Name : account.invoice.basic
On Create : ✓

Activities

Activity

Name	Workflow
draft	account.invoice.basic
proforma	account.invoice.basic
proforma2	account.invoice.basic
open	account.invoice.basic

Name : open
Workflow : account.invoice.basic
Kind : Function
Server Action : Print Invoice !!
Python Action :

```
action_move_create()
action_number()
write({'state':'open'})
```

Subflow :
Signal (subflow.*) :
Flow Start : ☐
Flow Stop : ☐
Split Mode : Xor
Join Mode : Xor

Outgoing transitions

Here in this example I added the Action to print the Invoice, when the Invoice will be confirmed.

CHAPTER

SEVENTEEN

DASHBOARD

Open ERP 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 Open ERP object, so no table corresponding to the _columns dictionnary 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 VI

Part 5 : Migration, Upgradation, Testing

CHAPTER

EIGHTEEN

DATA MIGRATION - IMPORT / EXPORT

18.1 Data Importation

18.1.1 Introduction

There are different methods to import your data into Open ERP:

- 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

18.1.2 Importing data through a module

The best way to import data in Open ERP 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 Open ERP 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 datas. 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 __terp__.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 Open ERP. Open ERP has a builtin system to manage identifications columns of the original software.

For this exercice, we will load data from another Open ERP 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 fiel, 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 Open ERP, 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'. Open ERP 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,Camptocamp
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, Open ERP will automatically import the partners and then the address and recreate efficiently the link between the two records. When installing a module, Open ERP 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.

18.1.3 Using Open ERP's ETL

The next version of Open ERP will include an ETL module to allow you to easily manages complex import jobs. If you are interrested in this system, you can check the complete specifications and the available prototype at this location:

bzr branch lp:~openerp-commiter/openobject-addons/trunk-extra-addons/etl

... to be continued ...

CHAPTER

NINETEEN

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 verion, you need to start the server with the **–update=all** argument :

```
tinyerp-server.py --update=all
```

You can also only upgrade specific modules, for example:

```
tinyerp-server.py --update=account,base
```

The database is rebuilt according to information provided in XML files and Python Classes. For more information on these functionalities, go to the section XML files and Defining Objects.

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 Open ERP to its basic configuration.

Part VII

Part 6 : Service base Integration

CHAPTER

TWENTY

WORKING WITH WEB SERVICES

20.1 How to load data ?

1. **Postgresql**
   - Simple, standard
   - Does not respect the WORKFLOW !!!
2. XML files (with –update=)
3. **XML-RPC**
   - Script, same as website interface

How to backup/restore a Postgresql database?

backup

pg_dump terp >terp.sql
restore
createdb terp –encoding=unicode psql terp < terp.sql or psql -d terp -f terp.sql

20.2 The objects methods

1. **create({'field':'value'})**
   - return ID created
2. **search([('arg1','=','value1')...], offset=0, limit=1000)**
   - return [IDS] found
3. **read([IDS], ['field1','field2',...])**
   - return [{'id':1, 'field1':..., 'field2':..., ...}, ...]
4. **write([IDS], {'field1':'value1','field2':3})**
   - return True
5. **unlink([IDS])**
   - return True

CHAPTER

TWENTYONE

XML-RPC WEB SERVICES

Jump to: navigation, search

1. **XML-RPC**
   - standard: http://www.xmlrpc.org
   - RPC Over HTTP
   - Function Parameters & Result encoded in XML
2. **Principle;**
   - **calls to objects methodes;** o read, write o create o unlink (=delete)

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.[1] 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:

XML-RPC Home Page\ XML-RPC for C and C++\ The Apache XML-RPC Project\ Expat: The XML Parser\

21.1 Interfaces

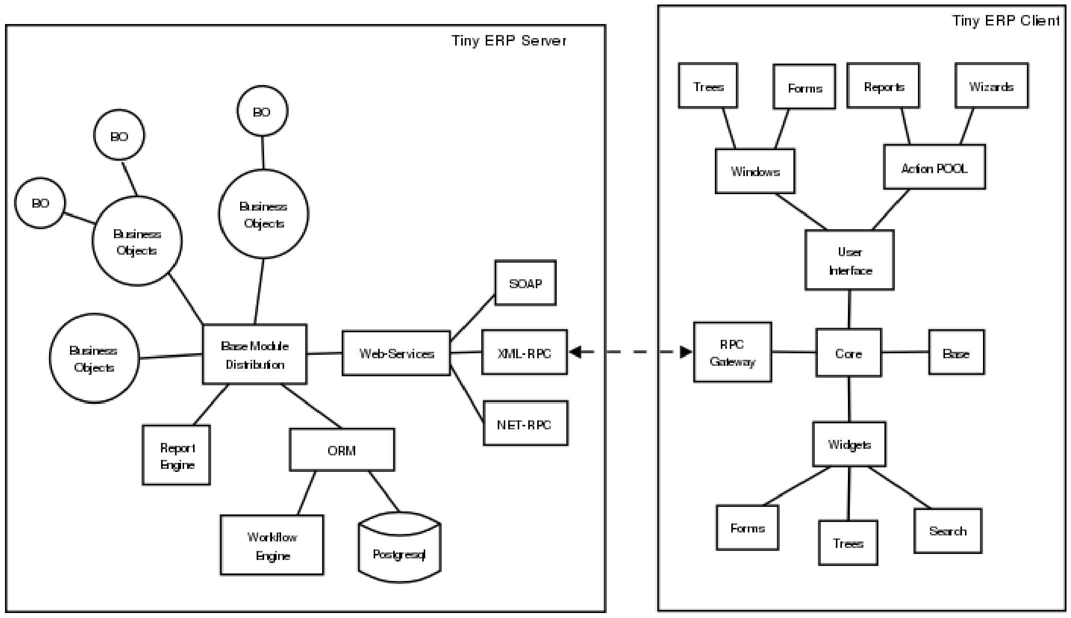
21.1.1 XML-RPC

XML-RPC Architecture

Open ERP is a 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 Open ERP. Open ERP server and Open ERP clients communicate using XML-RPC.



Client

The logic of Open ERP 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 Open ERP 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

21.1.2 Python

Access tiny-server using xml-rpc

Demo script

- **Create a partner and his 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 id
  ```

- **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)
  ```

21.1.3 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 apcahe/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 xmlrpcmsg('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 xmlrpcmsg('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**

  `TODO`

- **Write**

  `TODO`

21.1.4 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 TinyERP errors throw exception because the framework allows only an int as the error code where Tinyerp return a string.

Demo script

- **Find Databases**

```java
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 TinyERP Databases: ",e);
    return -2;
  }
  catch (Exception e)
  {
    logger.warn("Error while retrieving TinyERP Databases: ",e);
    return -3;
  }
}
```

- **Login**

```java
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 TinyERP: ",e);
      return -2;
    }
    catch (Exception e)
    {
      logger.warn("Error while logging to TinyERP: ",e);
      return -3;
    }
  }
```

- **Search**

  ```
  TODO
  ```

- **Create**

  ```
  TODO
  ```

- **Write**

  ```
  TODO
  ```

21.2 Python Example

Example of creation of a partner and his 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_ty
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()
```

21.3 PHP Example

Here is an example on how to insert a new partner using PHP. This example makes use 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 !';

?>
```

Part VIII

Part 7 : Other Topics

CHAPTER

TWENTYTWO

RAD TOOLS

22.1 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** accesible 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-2.2, 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 FilesDia and you will have the export function in Dia.

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.

22.2 Open Office Report Designer

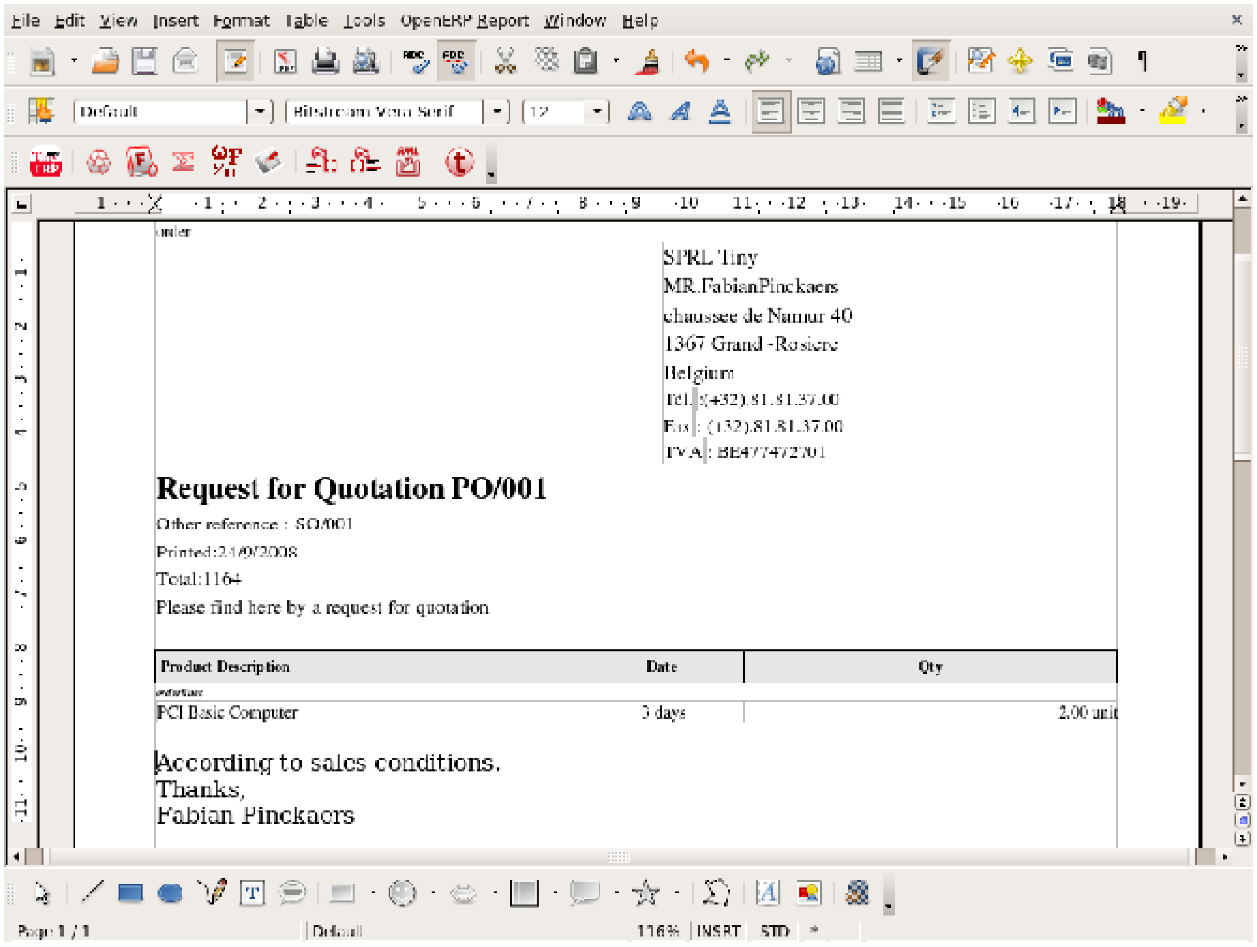
Select Tiny Report > Server parameters or Open ERP Report > Server parameters in the top menu of OpenOffice.org Writer. You can then enter your connection parameters to the Open ERP server. You must select a database `demo_min` in which you've already installed the module `sale` . A message appears if you've made a successful connection.

22.2.1 Modifying a report

The report editor lets you:

- modify existing reports which will then replace the originals in your Open ERP database,
- create new reports for the selected object.

To modify an existing report, select *Tiny Report > Modify Existing Report* . Choose the report `Request for Quotation` in the *Modify Existing Report* dialog box and then click *Save to Temp Directory* .


File Edit View Insert Format Table Tools OpenERP Report Window Help
Default
Bitstream Vera Serif
12
Page 1 / 1
Default
116%
INSRT
STD

order

SPRL Tiny
MR.FabianPinckaers
chaussee de Namur 40
1367 Grand -Rosiere
Belgium
Tel. :(+32).81.81.37.00
Fax : (+32).81.81.37.00
TVA : BE477472701

Request for Quotation PO/001

Other reference : SO/001
Printed:24/09/2008
Total:1164
Please find here by a request for quotation

Product Description	Date	Qty
PC1 Basic Computer	3 days	2.00 unit

According to sales conditions.
Thanks,
Fabian Pinckaers

Modifying a document template

OpenOffice.org then opens the report in edit mode for you. You can modify it using the standard word processing functions of OpenOffice.org Writer.

The document is modified in its English version. It will be translated as usual by Open ERP's translation system when you use it through the client interface, if you've personalized your own setup to translate to another language for you. So you only need to modify the template once, even if your system uses other languages – but you'll need to add translations as described earlier in this chapter if you add fields or change the content of the existing ones.

Tip: ***Attention*** Older reports

The older reports haven't all been converted into the new form supported by Open ERP. The data expressions in the old format are shown within double brackets and not in OpenOffice.org fields.

You can transform an old report format to the new format from the OpenOffice.org menu Tiny Report > Convert Bracket–Fields.

From the Tiny toolbar in OpenOffice.org it's possible to:

- connect to the Open ERP server: by supplying the connection parameters.
- add a loop: select a related field amongst the available fields from the proposed object, for example `Order lines` . When it's printed this loop will be run for each line of the order. The loop can be put into a table (the lines will then be repeated) or into an OpenOffice.org section.
- add a field: you can then go through the whole Open ERP database from the selected object and then a particular

field.

- add an expression: enter an expression in the Python language to calculate values from any fields in the selected object.

Tip: ***Technique*** Python Expressions

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.

> *Tiny Report > Send to server Technical Name Report Name* `Sale Order Mod` *Corporate Header Send Report to Server*

You can check the result in Open ERP using the menu *Sales Management > Sales Orders > All Orders* .

22.2.2 Creating a new report

> *Tiny Report > Open a new report* `Sale Order` *Open New Report Use Model in Report*

The general template is made up of loops (such as the list of selected orders) and fields from the object, which can also be looped. Format them to your requirements then save the template.

The existing report templates make up a rich source of examples. You can start by adding the loops and several fields to create a minimal template.

When the report has been created, send it to the server by clicking *Tiny Report > Send to server* , which brings up the *Send to server* dialog box. Enter the *Technical Name* of `sale.order` , to make it appear beside the other sales order reports. Rename the template as `Sale Order New` in *Report Name* , check the checkbox *Corporate Header* and finally click *Send Report to Server* .

To send it to the server, you can specify if you prefer Open ERP to produce a PDF when the user prints the document, or if Open ERP should open the document for editing in OpenOffice.org Writer before printing. To do that choose `PDF` or `SXW` (a format of OpenOffice.org documents) in the field *Select Report Type*

Open ERP 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 Open ERP 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.

```
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 IX

Part 8 : Appendices

CHAPTER

TWENTYTHREE

APPENDICES INDEX

23.1 Appendices A : Coding Conventions

23.1.1 Python coding

Use tabs: will be replaced by spaces soon...

Take care with default values for arguments: they are only evaluated once when the module is loaded and then used at each call. This means that if you use a mutable object as default value, and then modify that object, at the next call you will receive the modified object as default argument value. This applies to dict and list objects which are very often used as default values. If you want to use such objects as default value, you must either ensure that they won't be modified or use another default value (such as None) and test it. For example:

```
def foo(a=None):

    if a is None:

        a=[]

    # ...
```

This is what is [in the Python documentation]. In addition it is good practice to avoid modifying objects that you receive as arguments if it is not specified. If you want to do so, prefer to copy the object first. A list can easily be copied with the syntax

> copy = original[:]

A lot of other objects, such as dict, define a copy method.

23.1.2 File names

The structure of a module should be like this:

```
/module/

    /__init__.py
    /__terp__.py
    /module.py
    /module_other.py
    /module_view.xml
    /module_wizard.xml
    /module_report.xml
    /module_data.xml
    /module_demo.xml
    /wizard/
    /__init__.py
    /wizard_name.py

/report/

    /__init__.py
    /report_name.sxw
    /report_name.rml
    /report_name.py
```

23.1.3 Naming conventions

- **modules: modules must be written in lower case, with underscores. The name of the module is the name of the directory i**
  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 objec**
  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, acti**
  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

23.2 Releasing a module

23.2.1 Introduction

You can publish your work under our systems to:

- Get help from contributors or interrested partners for the development
- Get feedback from testers and translators
- Get your module in the next distribution/version of Open ERP (if accepted by the editor) so that you do not have to manage migrations, testing per version, ...

23.2.2 Open Forge

Here is the process of publishing a module or patch:

1. Create a project on http://OpenForge.com

2. Upload your work on your Open Forge project
3. Create an entry on the module repository of Open ERP's website

The Open Forge has tools to help your team collaborate, like message forums, tasks tracker and mailing lists; tools to create and control access to Source Code Management repositories. It is the central repository of collaborative developments for Open ERP.

23.3 Translations

Open ERP 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 Open ERP.

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 Open ERP in a language other than English, then you have to 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 Open ERP 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 :

MISSING IMAGE FILE

23.3.1 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.

MISSING IMAGE FILE

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”.

MISSING IMAGE FILE

23.3.2 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!**

23.3.3 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:

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.

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

3. Import this file into Open ERP (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

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 theses 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 th** 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)

23.3.4 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*.

INDEX