OpenERP Server Developers Documentation

Release 7.0b

OpenERP s.a.

November 28, 2015

Contents

CHAPTER 1

OpenERP Server

1.1 Getting started with OpenERP development

1.1.1 Installation from sources

Source code is hosted on Launchpad. In order to get the sources, you will need Bazaar to pull the source from Launchpad. Bazaar is a version control system that helps you track project history over time and collaborate efficiently. You may have to create an account on Launchpad to be able to collaborate on OpenERP development. Please refer to the Launchpad and Bazaar documentation to install and setup your development environment.

The running example of this section is based on an Ubuntu environment. You may have to adapt the steps according to your system. Once your working environment is ready, prepare a working directory that will contain the sources. For a `source` base directory, type:

```
mkdir source;cd source
```

OpenERP provides a setup script that automatizes the tasks of creating a shared repository and getting the source code. Get the setup script of OpenERP by typing:

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

This will create the following two files in your `source` directory:

```
-rw-rw-r--  1 openerp openerp 5465 2012-04-17 11:05 Makefile
-rw-rw-r--  1 openerp openerp 2902 2012-04-17 11:05 Makefile_helper.py
```

If you want some help about the available options, please type:

```
make help
```

Next step is to initialize the shared repository and download the sources. Get the current trunk version of OpenERP by typing:

```
make init-trunk
```

This will create the following structure inside your `source` directory, and fetch the latest source code from `trunk`:

```
drwxrwxr-x  3 openerp openerp 4096 2012-04-17 11:10 addons
drwxrwxr-x  3 openerp openerp 4096 2012-04-17 11:10 misc
drwxrwxr-x  3 openerp openerp 4096 2012-04-17 11:10 server
drwxrwxr-x  3 openerp openerp 4096 2012-04-17 11:10 web
```

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

```
sudo apt-get install graphviz ghostscript postgresql-client \
          python-dateutil python-feedparser python-gdata \
          python-ldap python-libxslt1 python-lxml python-mako \
          python-openid python-psycopg2 python-pybabel python-pychart \
          python-pydot python-pyparsing python-reportlab python-simplejson \
          python-tz python-vatnumber python-vobject python-webdav \
          python-werkzeug python-xlwt python-yaml python-imaging \
          python-matplotlib
```

Next step is to initialize the database. This will create a new openerp role:

```
make db-setup
```

Finally, launch the OpenERP server:

```
make server
```

Testing your installation can be done on http://localhost:8069/. You should see the OpenERP main login page.

1.1.2 Command line options

Using the command

```
./openerp-server --help
```

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
```

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 OpenERP to another language.See i18n section of the user manual. Option '-l' is mandatory.:

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

Options from previous versions

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

1.1.3 Configuration

Two configuration files are available:

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

If they are not found, the server and the client will start with a default configuration. Those files follow the convention used by python's ConfigParser module. Please note that lines beginning with "#" or ";" are comments. The client configuration file is automatically generated upon the first start. The sezrver configuration file can automatically be created using the command

```
./openerp-server -s or ./openerp-server --save
```

You can specify alternate configuration files with

```
-c CONFIG, --config=CONFIG specify alternate config file
```

Configure addons locations

By default, the only directory of addons known by the server is server/bin/addons. It is possible to add new addons by

- copying them in server/bin/addons, or creating a symbolic link to each of them in this directory, or

- specifying another directory containing addons to the server. The later can be accomplished either by running the server with the `--addons-path=` option, or by configuring this option in the openerp_serverrc file, automatically generated under Linux in your home directory by the server when executed with the `--save` option. You can provide several addons to the `addons_path` = option, separating them using commas.

1.1.4 Start-up script

New in version 6.1.

To run the OpenERP server, the conventional approach is to use the *openerp-server* script. It loads the openerp library, sets a few configuration variables corresponding to command-line arguments, and starts to listen to incoming connections from clients.

Depending on your deployment needs, you can write such a start-up script very easily. We also recommend you take a look at an alternative tool called *openerp-command* that can, among other things, launch the server.

Yet another alternative is to use a WSGI-compatible HTTP server and let it call into one of the WSGI entry points of the server.

1.2 Architecture

1.2.1 OpenERP as a multitenant three-tiers architecture

This section presents the OpenERP architecture along with technology details of the application. The tiers composing OpenERP are presented. Communication means and protocols between the application components are also presented. Some details about used development languages and technology stack are then summarized.

OpenERP is a multitenant, three-tiers architecture: database tier for data storage, application tier for processing and functionalities and presentation tier providing user interface. Those are separate layers inside OpenERP. The application tier itself is written as a core; multiple additional modules can be installed in order to create a particular instance of OpenERP adapted to specific needs and requirements. Moreover, OpenERP follows the Model-View-Controller (MVC) architectural pattern.

A typical deployment of OpenERP is shown on *Figure 1*. This deployment is called Web embedded deployment. As shown, an OpenERP system consists of three main components:

- a PostgreSQL database server which contains all OpenERP databases. Databases contain all application data, and also most of the OpenERP system configuration elements. Note that this server can possibly be deployed using clustered databases.
- the OpenERP Server, which contains all the enterprise logic and ensures that OpenERP runs optimally. One layer of the server is dedicated to communicate and interface with the PostgreSQL database, the ORM engine. Another layer allows communications between the server and a web browser, the Web layer. Having more than one server is possible, for example in conjunction with a load balancing mechanism.
- the client running in the a web browser as javascript application.

The database server and the OpenERP server can be installed on the same computer, or distributed onto separate computer servers, for example for performance considerations.

The next subsections give details about the different tiers of the OpenERP architecture.

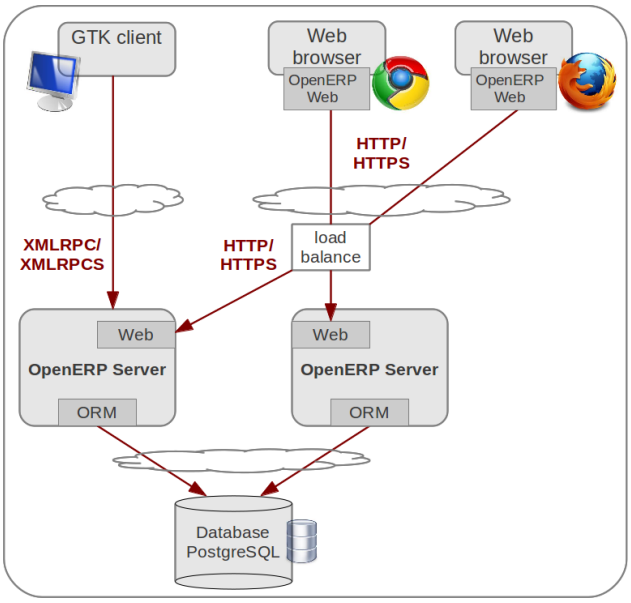

Fig. 1.1: OpenERP 6.1 architecture for embedded web deployment

PostgreSQL database

The data tier of OpenERP is provided by a PostgreSQL relational database. While direct SQL queries can be executed from OpenERP modules, most accesses to the relational database are done through the server Object Relational Mapping layer.

Databases contain all application data, and also most of the OpenERP system configuration elements. Note that this server can possibly be deployed using clustered databases.

OpenERP server

OpenERP provides an application server on which specific business applications can be built. It is also a complete development framework, offering a range of features to write those applications. Among those features, the OpenERP ORM provides functionalities and an interface on top of the PostgreSQL server. The OpenERP server also features a specific layer designed to communicate with the web browser-based client. This layer connects users using standard browsers to the server.

From a developer perspective, the server acts both as a library which brings the above benefits while hiding the low-level details, and as a simple way to install, configure and run the written applications. The server also contains other services, such as extensible data models and view, workflow engine or reports engine. However, those are OpenERP services not specifically related to security, and are therefore not discussed in details in this document.

Server - ORM

The Object Relational Mapping ORM layer is one of the salient features of the OpenERP Server. It provides additional and essential functionalities on top of PostgreSQL server. Data models are described in Python and OpenERP creates the underlying database tables using this ORM. All the benefits of RDBMS such as unique constraints, relational integrity or efficient querying are used and completed by Python flexibility. For instance, arbitrary constraints written in Python can be added to any model. Different modular extensibility mechanisms are also afforded by OpenERP.

It is important to understand the ORM responsibility before attempting to by-pass it and to access directly the underlying database via raw SQL queries. When using the ORM, OpenERP can make sure the data remains free of any

corruption. For instance, a module can react to data creation in a particular table. This behavior can occur only if queries go through the ORM.

The services granted by the ORM are among other :

- consistency validation by powerful validity checks,
- providing an interface on objects (methods, references, ...) allowing to design and implement efficient modules,
- row-level security per user and group; more details about users and user groups are given in the section Users and User Roles,
- complex actions on a group of resources,
- inheritance service allowing fine modeling of new resources

Server - Web

The web layer offers an interface to communicate with standard browsers. In the 6.1 version of OpenERP, the web-client has been rewritten and integrated into the OpenERP server tier. This web layer is a WSGI-compatible application based on werkzeug. It handles regular http queries to server static file or dynamic content and JSON-RPC queries for the RPC made from the browser.

Modules

By itself, the OpenERP server is a core. For any enterprise, the value of OpenERP lies in its different modules. The role of the modules is to implement any business requirement. The server is the only necessary component to add modules. Any official OpenERP release includes a lot of modules, and hundreds of modules are available thanks to the community. Examples of such modules are Account, CRM, HR, Marketing, MRP, Sale, etc.

Clients

As the application logic is mainly contained server-side, the client is conceptually simple. It issues a request to the server, gets data back and display the result (e.g. a list of customers) in different ways (as forms, lists, calendars, ...). Upon user actions, it sends queries to modify data to the server.

The default client of OpenERP is an JavaScript application running in the browser that communicates with the server using JSON-RPC.

1.2.2 MVC architecture in OpenERP

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

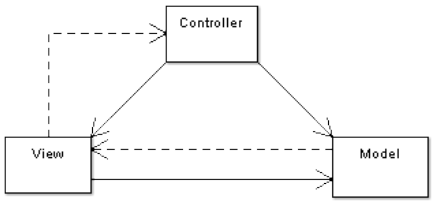

Fig. 1.2: Model-View-Controller diagram

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

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

OpenERP follows the MVC semantic with

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

1.2.3 Network communications and WSGI

OpenERP is an HTTP web server and may also be deployed as an WSGI-compliant application.

Clients may communicate with OpenERP using sessionless XML-RPC, the recommended way to interoperate with OpenERP. Web-based clients communicates using the session aware JSON-RPC.

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

Services can make use of the WSGI stack. WSGI is a standard solution in the Python ecosystem to write HTTP servers, applications, and middleware which can be used in a mix-and-match fashion. By using WSGI, it is possible to run OpenERP in any WSGI compliant server. It is also possible to use OpenERP to host a WSGI application.

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

The HTTP namespaces /openerp/ /object/ /common/ are reserved for the XML-RPC layer, every module restrict it's HTTP namespace to /<name_of_the_module>/

1.3 Modules

1.3.1 Module structure

A module can contain the following elements:

- **Business object** : declared as Python classes extending the class osv.Model, the persistence of these resource is completly managed by OpenERP's ORM.
- **Data** : XML/CSV files with meta-data (views and workflows declaration), configuration data (modules parametrization) and demo data (optional but recommended for testing),
- **Reports** : RML (XML format). HTML/MAKO or OpenOffice report templates, to be merged with any kind of business data, and generate HTML, ODT or PDF reports.

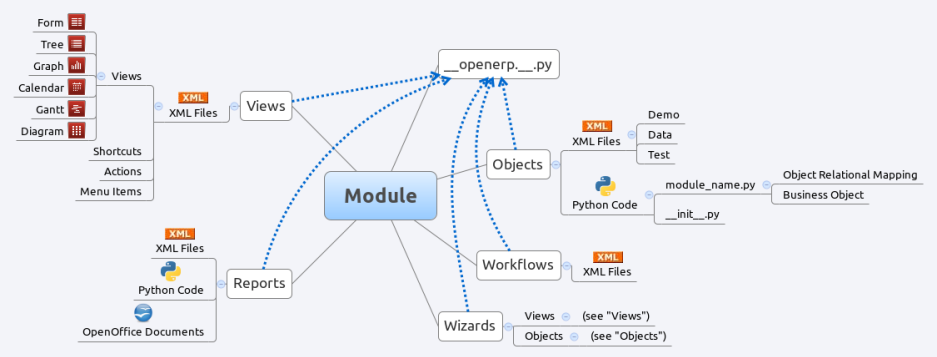

Fig. 1.3: Module composition

Each module is contained in its own directory within either the server/bin/addons directory or another directory of addons, configured in server installation. To create a new module for example the 'OpenAcademy' module, the following steps are required:

- create a `openacademy` subdirectory in the source/addons directory
- create the module import file `__init__.py`
- create the module manifield file `__openerp__.py`
- create **Python** files containing **objects**
- create **.xml files** holding module data such as views, menu entries or demo data
- optionally create **reports** or **workflows**

Python import file __init__.py

The `__init__.py` file is the Python import file, because an OpenERP module is also a regular Python module. The file should import all the other python file or submodules.

For example, if a module contains a single python file named `openacademy.py`, the file should look like:

```
import openacademy
```

Manifest file __openerp__.py

In the created module directory, you must add a **__openerp__.py** file. This file, which must be a Python dict literal, 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.
3. declare additional meta data

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

```
name            The name of the module in English.
version         The version of the module.
summary         Short description or keywords
description     The module description (text).
category        The categrory of the module
author          The author of the module.
```

```
website          URL of the website of the module.
license          The license of the module (default: AGPL-3).
depends          List of modules on which this module depends beside base.
data             List of .xml files to load when the module is installed or updated.
demo             List of additional .xml files to load when the module is
                 installed or updated and demo flag is active.
installable      True or False. Determines whether the module is installable
                 or not.
auto_install     True or False (default: False). If set to ``True``, the
                 module is a link module. It will be installed as soon
                 as all its dependencies are installed.
```

For the `openacademy` module, here is an example of `__openerp__.py` declaration file:

```
{
    'name' : "OpenAcademy",
    'version' : "1.0",
    'author' : "OpenERP SA",
    'category' : "Tools",
    'depends' : ['mail'],
    'data' : [
        'openacademy_view.xml',
        'openacademy_data.xml',
        'report/module_report.xml',
        'wizard/module_wizard.xml',
    ],
    'demo' : [
        'openacademy_demo.xml'
    ],
    'installable': True,
}
```

Objects

All OpenERP resources are objects: invoices, partners. Metadata are also object too: menus, actions, reports... 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.

Those object are declared in python be subclassing osv.Model

The ORM of OpenERP is constructed over PostgreSQL. It is thus possible to query the object used by OpenERP using the object interface (ORM) 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.

XML Files

XML files located in the module directory are used to initialize or update the the database when the module is installed or updated. They are used for many purposes, among which we can cite :

- initialization and demonstration data declaration,

- views declaration,
- reports declaration,
- workflows declaration.

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

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

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

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.

The Record Field tag

The attributes for the field tag are the following:

name [mandatory] the field name

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

ref reference to an id defined in this file

model model to be looked up in the search

search a query

Example

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

```
     <field name="report_xsl">account/report/invoice.xsl</field>
     <field name="report_xml">account/report/invoice.xml</field>
</record>
```

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

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

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

This last record defines the admin user :

- The fields login, password, etc are straightforward.
- The ref attribute allows to fill relations between the records :

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

The field **company_id** is a many-to-one relation from the user object to the company object, and **main_company** is the id of to associate.

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

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

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

Function tag

A function tag can contain other function tags.

model [mandatory] The model to be used

name [mandatory] the function given name

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

Example

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

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

To develop new objects

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

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

Reports are described more in details in the Reporting chapter.

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

A few examples:

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

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

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

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

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

The arrows are the conditions;

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

The squared nodes represent other Workflows;

- the invoice
- the shipping

i18n Changed in version 5.0.

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.

Translations are managed by the Launchpad Web interface. Here, you'll find the list of translatable projects.

Please read the FAQ before asking questions.

1.3.2 Objects, Fields and Methods

OpenERP Objects

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

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

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

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

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

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

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

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

For developers:

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

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

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

The ORM - Object-relational mapping - Models

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

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

OpenERP Object Attributes

Objects Introduction

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

Object definition

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

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

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

Predefined fields

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

_columns (required) The object fields. See the *fields* section for further details.

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

_sql_constraints The SQL Constraint on the object. See the SQL constraints section for further details.

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

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

_inherits The list of osv objects the object inherits from. This list must be given in a python dictionary of the form: {'name_of_the_parent_object': 'name_of_the_field', ...}. See the *object inheritance section* (second form) for further details. Default value: {}.

_log_access Determines whether or not the write access to the resource must be logged. If true, four fields will be created in the SQL table: create_uid, create_date, write_uid, write_date. Those fields represent respectively the id of the user who created the record, the creation date of record, the id of the user who last modified the record, and the date of that last modification. This data may be obtained by using the perm_read method.

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

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

Default value: 'id'.

Examples:

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

_rec_name Name of the field in which the name of every resource is stored. Default value: 'name'. Note: by default, the name_get method simply returns the content of this field.

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

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

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

Object Inheritance - _inherit

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 fields 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.

Inheritance by Delegation - _inherits

Syntax ::

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

The object 'tiny.object' inherits from all the columns and all the methods from the n objects 'tiny.object_a', ..., 'tiny.object_n'.

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

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

Fields Introduction

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

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

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

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

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

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

ondelete How to handle deletions in a related record. Allowable values are: 'restrict', 'no action', 'cascade', 'set null', and 'set default'.

priority Not used?

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

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

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

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

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

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

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

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

domain Domain restriction on a relational field.

Default value: [].

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

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

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

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

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

Type of Fields

Basic Types

boolean A boolean (true, false).

Syntax:

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

integer An integer.

Syntax:

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

float A floating point number.

Syntax:

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

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

Example:

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

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

Syntax:

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

Example:

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

text A text field with no limit in length.

Syntax:

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

date A date.

Syntax:

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

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

Syntax:

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

binary A binary chain

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

Syntax:

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

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

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

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

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

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

Example

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

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

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

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

Relational Types

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

Syntax:

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

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

Syntax:

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

Optional parameters:

- **ondelete: What should happen when the resource this field points to is deleted.**
  - Predefined value: “cascade”, “set null”, “restrict”, “no action”, “set default”
  - Default value: “set null”
- required: True
- readonly: True
- select: True - (creates an index on the Foreign Key field)

Example

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

one2many TODO

Syntax:

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

Optional parameters:

- invisible: True/False
- states: ?

- readonly: True/False

Example

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

many2many TODO

Syntax:

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

Where:

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

Example:

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

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

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

Example:

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

```
        }
res_partner_category2()
```

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

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

Where:

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

Functional Fields

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

Parameters:

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

where

- *fnct* is the function or method that will compute the field value. It must have been declared before declaring the functional field.
- *fnct_inv* is the function or method that will allow writing values in that field.
- *type* is the field type name returned by the function. It can be any field type name except function.
- *fnct_search* allows you to define the searching behaviour on that field.
- *method* whether the field is computed by a method (of an object) or a global function
- *store* If you want to store field in database or not. Default is False.
- *multi* is a group name. All fields with the same *multi* parameter will be calculated in a single function call.

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

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

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

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

Either way, it must return a dictionary of values of the form **{id'_1_': value'_1_', id'_2_': value'_2_',...}.**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example Suppose we create a contract object which is :

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

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

```
class hr_employee(osv.osv):
    _name = "hr.employee"
    _description = "Employee"
    _inherit = "hr.employee"
    _columns = {
```

```
        'contract_ids' : fields.one2many('hr.contract', 'employee_id', 'Contracts'),
        'function' : fields.function(
            _get_cur_function_id,
            type='many2one',
            obj="res.partner.function",
            method=True,
            string='Contract Function'),
    }
hr_employee()
```

Note: three points

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

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

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

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

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

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

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

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

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

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

Here's an example from the membership module:

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

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. If this object has no value, the system will give you the default property.

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

Using properties or normal fields

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

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

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

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

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

Example 1: Account Receivable

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

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

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

Example 2: Product Times

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

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

```
class product_product(osv.osv):

    _inherit = 'product.product'
    _name = 'product.product'
    _columns = {

        'life_time': fields.integer('Product lifetime'),
        'use_time': fields.integer('Product usetime'),
        'removal_time': fields.integer('Product removal time'),
        'alert_time': fields.integer('Product alert time'),
        }

product_product()
```

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

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

ORM methods

Keeping the context in ORM methods

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

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

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

1.3.3 Views and Events

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 OpenERP 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.

Views principles

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

There are two types of views:

1. form views
2. tree views

Note: Since OpenERP 4.1, form views can also contain graphs.

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.

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.
- 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.

On Change

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

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

```
<field name="name_of_field" on_change="name_of_method(other_field'_1_', ..., other_field'_n_')"/>
```

The example below is from the sale order view.

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

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

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

    v={}
    if shop_id:

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

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

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

    return {'value':v}
```

When editing the shop_id form field, the onchange_shop_id method of the sale_order object is called and returns a dictionary where the 'value' key contains a dictionary of the new value to use in the 'project_id', 'pricelist_id' and 'payment_default_id' fields.

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

returns a dictionary with any mix of the following keys:

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

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

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

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

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.

Search views

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

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

Group tag

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

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

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

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

Filter tag

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

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

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

Group By

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

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

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

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

Fields

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

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

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

- a search field really builds a domain composed of `[(field_name, operator, field_value)]`. This domain can be overridden in two ways:
  - `@operator` replaces the default operator for the field (which depends on its type)
  - `@filter_domain` lets you provide a fully custom domain, which will replace the default domain creation

- a search field does not create a context by default, but you can provide an `@context` which will be evaluated and merged into the wider context (as with a `filter` element).

To get the value of the field in your `@context` or `@filter_domain`, you can use the variable `self`:

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

or

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

Range fields (date, datetime, time) The range fields are composed of two input widgets (from and two) instead of just one.

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

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

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

Actions for Search view

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

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

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

Custom Filters

As of v6.0, all search views also features custom search filters, as show below. Users can define their own custom filters using any of the fields available on the current model, combining them with AND/OR operators. It is also possible to save any search context (the combination of all currently applied domain and context values) as a personal

filter, which can be recalled at any time. Filters can also be turned into Shortcuts directly available in the User's homepage.

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

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

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 separate menu item):

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

view_type:

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

view_mode:

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

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

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

The graph view

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

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">
```

Field tag

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

- **group**: if set to true, the client will group all item of the same value for this field. For 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 which 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.

Controlling view actions

When defining a view, the following attributes can be added on the opening element of the view (i.e. `<form>`, `<tree>`...)

`create` set to `false` to hide the link / button which allows to create a new record.

`delete` set to `false` to hide the link / button which allows to remove a record.

`edit` set to `false` to hide the link / button which allows to edit a record.

These attributes are available on form, tree, kanban and gantt views. They are normally automatically set from the access rights of the users, but can be forced globally in the view definition. A possible use case for these attributes is to define an inner tree view for a one2many relation inside a form view, in which the user cannot add or remove related records, but only edit the existing ones (which are presumably created through another way, such as a wizard).

Calendar Views

Calendar view provides timeline/schedule view for the data.

View Specification

Here is an example view:

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

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

string The title string for the view.

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

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

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

day_length An `integer` value to specify working day length. Default is `8` hours.

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

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

- `day`
- `week`
- `month`

Screenshots

Month Calendar:

Week Calendar:

Gantt Views

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

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

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

Color coding of the summary bar is as follows:

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

View Specification

Here is an example view:

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

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

Here is the list of supported attributes for gantt tag:

string The title string for the view.

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

date_stop A datetime field to specify the end date. Ignored if date_delay attribute is specified.

date_delay A numeric field to specify time in hours for a record. This attribute will get preference over date_stop and date_stop will be ignored.

day_length An integer value to specify working day length. Default is 8 hours.

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

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

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

The level tag supports following attributes:

object An openerp object having many2one relationship with view object.

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

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

Drag and Drop

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

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

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

Note: The time is calculated considering day_length so a bar will span more then one day if total time for a task is greater then day_length value.

Screenshots

Design Elements

The files describing the views are of the form:

Example

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

The view definitions contain mainly three types of tags:

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

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

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

Example

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

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

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

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

View Types

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

```
<record id="view_location_tree2" model="ir.ui.view">
    <field name="name">stock.location.tree</field>
    <field name="model">stock.location</field>
    <field name="type">tree</field>
    <field name="priority" eval="2"/>
```

```
    <field name="arch" type="xml">
        <tree
                colors="blue:usage=='view';darkred:usage=='internal'">

            <field name="complete_name"/>
            <field name="usage"/>
            <field
                name="stock_real"
                invisible="'product_id' not in context"/>
            <field
                name="stock_virtual"
                invisible="'product_id' not in context"/>
        </tree>
    </field>
</record>
```

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

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

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

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

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

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.

Data Elements

Field *attributes for the "field" tag*

- **`select="1"`: mark this field as being one of the search criteria for** this resource's search view. A value of 1 means that the field is included in the basic search, and a value of 2 means that it is in the advanced search.
- `colspan="4"`: the number of columns on which a field must extend.
- `readonly="1"`: set the widget as read only
- `required="1"`: the field is marked as required. If a field is marked as required, a user has to fill it the system won't save the resource if the field is not filled. This attribute 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.
- `password="True"`: replace field values by asterisks, "*".
- `string=""`: change the field label. Note that this label is also used in the search view: see select attribute above).
- **`domain`: can restrict the domain.**
  - Example: domain="[('partner_id','=',partner_id)]"
- **`widget`: can change the widget.**
  - **Example: widget="one2many_list"**
    - one2one_list

        - one2many_list
        - many2one_list
        - many2many
        - url
        - email
        - image
        - float_time
        - reference

- **mode: sequences of the views when switching.**
    - Example: mode="tree,graph"
- **on_change: define a function that is called when the content of the field changes.**
    - Example: on_change="onchange_partner(type,partner_id)"
    - See 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, b**
    - Format: "{'attribute':[('field_name','operator','value'),('field_name','operator','value')],'attribute2':[('field_name','op
    - where attribute will be readonly, invisible, required
    - Default value: {}.
    - Example: (in product.product)

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

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

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

Example

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

Example

```
<?xml version="1.0"?>
<openerp>
    <data>
    <record id="view_partner_form" model="ir.ui.view">
            <field name="name">res.partner.form</field>
            <field name="model">res.partner</field>
            <field name="type">form</field>
            <field name="arch" type="xml">
            <form string="Partners">
                <group colspan="4" col="6">
                    <field name="name" select="1"/>
                    <field name="ref" select="1"/>
                    <field name="customer" select="1"/>
```

```
                    <field domain="[('domain', '=', 'partner')]" name="title"/>
                    <field name="lang" select="2"/>
                    <field name="supplier" select="2"/>
                </group>
                <notebook colspan="4">
                    <page string="General">
                        <field colspan="4" mode="form,tree" name="address"
                         nolabel="1" select="1">
                            <form string="Partner Contacts">
                                <field name="name" select="2"/>
                                <field domain="[('domain', '=', 'contact')]" name="title"/>
                                <field name="function"/>
                                <field name="type" select="2"/>
                                <field name="street" select="2"/>
                                <field name="street2"/>
                                <newline/>
                                <field name="zip" select="2"/>
                                <field name="city" select="2"/>
                                <newline/>
                                <field completion="1" name="country_id" select="2"/>
                                <field name="state_id" select="2"/>
                                <newline/>
                                <field name="phone"/>
                                <field name="fax"/>
                                <newline/>
                                <field name="mobile"/>
                                <field name="email" select="2" widget="email"/>
                            </form>
                            <tree string="Partner Contacts">
                                <field name="name"/>
                                <field name="zip"/>
                                <field name="city"/>
                                <field name="country_id"/>
                                <field name="phone"/>
                                <field name="email"/>
                            </tree>
                        </field>
                        <separator colspan="4" string="Categories"/>
                        <field colspan="4" name="category_id" nolabel="1" select="2"/>
                    </page>
                    <page string="Sales &amp;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_list"/>
                    </page>
                    <page string="Notes">
                        <field colspan="4" name="comment" nolabel="1"/>
                    </page>
                </notebook>
            </form>
            </field>
```

```
        </record>
    <menuitem
            action="action_partner_form"
            id="menu_partner_form"
            parent="base.menu_base_partner"
            sequence="2"/>
    </data>
</openerp>
```

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

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

Example 1:

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

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

Example 2:

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

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

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

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

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

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

Buttons have the following attributes:

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

`workflow` (default) The button will send a workflow signal [2] on the current model using the `@name` of the button as workflow signal name and providing the record id as parameter (in a list).

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

`object` The button will execute the method of name `@name` on the current model, providing the record id as parameter (in a list). This call may return an action descriptor to execute.

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

From there, follows the normal action-execution workflow.

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

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

[2] via `exec_workflow` on the `object` rpc endpoint

Warning: `@special` and `@type` are incompatible.

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

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

`@string` The label which should be displayed on the button [3].

`@icon` Display an icon on the button, if absent the button is text-only [4].

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

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

May be ignored by the client.

New in version 6.0.

Example

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

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

Example

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

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

Example

```
<newline/>
```

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>
```

[3] in form view, in list view buttons have no label
[4] behavior in list view is undefined, as list view buttons don't have labels.

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

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

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

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

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

You can use these values in the position attribute:

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

Replacing Content

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

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

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

Deleting Content

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

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

Inserting Content

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

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

Will add `relation_ids` field before the `lang` field.

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

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

Will add `relation_ids` field after the `lang` field.

Multiple Changes

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

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

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

XPath Element

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

```
<record model="ir.ui.view" id="view_partner_form6">
    <field name="name">res.partner.form.inherit6</field>
    <field name="model">res.partner</field>
```

```
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
        <data>
            <xpath
                expr="//field[@name='address']/form/field[@name='email']"
                position="after">
                <field name="age"/>
            </xpath>
            <xpath
                expr="//field[@name='address']/tree/field[@name='email']"
                position="after">
                <field name="age"/>
            </xpath>
        </data>
    </field>
</record>
```

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

Specify the views you want to use

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

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

Using the priority field

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

```
<!--
    Here is the first view for the model 'client'.
    We don't specify a priority field, which means
    by default 16.
-->
<record model="ir.ui.view" id="client_form_view_1">
    <field name="name">client.form.view1</field>
    <field name="model">client</field>
    <field name="type">form</fiel>
    <field name="arch" type="xml">
        <field name="firstname"/>
        <field name="lastname"/>
    </field>
</record>

<!--
    A second view, which show fields in an other order.
    We specify a priority of 15.
-->
<record model="ir.ui.view" id="client_form_view_2">
    <field name="name">client.form.view2</field>
    <field name="model">client</field>
    <field name="priority" eval="15"/>
```

```
24      <field name="type">form</fiel>
25      <field name="arch" type="xml">
26          <field name="lastname"/>
27          <field name="firstname"/>
28      </field>
29  </record>
```

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

Specify per-action view

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

```
1   <!--
2       This action open the default view (in our case,
3       the view with the highest priority, the second one)
4   -->
5   <record
6       model="ir.actions.act_window"
7       id="client_form_action">
8       <field name="name">client.form.action</field>
9       <field name="res_model">client</field>
10      <field name="view_type">form</field>
11      <field name="view_mode">form</field>
12  </record>
13
14  <!--
15      This action open the view we specify.
16  -->
17  <record
18      model="ir.actions.act_window"
19      id="client_form_action1">
20      <field name="name">client.form.action1</field>
21      <field name="res_model">client</field>
22      <field name="view_type">form</field>
23      <field name="view_mode">form</field>
24      <field name="view_id" ref="client_form_view_1"/>
25  </record>
26
27  <menuitem id="menu_id" name="Client main menu"/>
28  <menuitem
29      id="menu_id_1"
30      name="Here we don't specify the view"
31      action="client_form_action" parent="menu_id"/>
32  <menuitem
33      id="menu_id_1"
34      name="Here we specify the view"
35      action="client_form_action1" parent="menu_id"/>
```

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

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

Specify views for related fields

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

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

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

If you want to specify the views to use, you can add a *context* attribute, and specify a view id for each type of view supported, exactly like the action's *view_id* attribute, except that the provided view id must always be fully-qualified with the module name, even if it belongs to the same module:

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

Note: You *have to* put the module name in the view_id, because this is evaluated when the view is displayed, and not when the XML file is parsed, so the module name information is not available. Failing to do so will result in the default view being selected (see below).

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

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

Note: The context keys are named <view_type>_view_ref.

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

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

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

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

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

```
            </tree>
        </field>
</field>
```

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

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

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

1.3.4 Menus and Actions

Menus

Menus are records in the `ir.ui.menu` table. In order to create a new menu entry, you can directly create a record using the `record` tag.

```
<record id="menu_xml_id" model="ir.ui.menu">
  <field name="name">My Menu</field>
  <field name="action" ref="action_xml_id"/>
  <field name="sequence" eval="<integer>"/>
  <field name="parent_id" ref="parent_menu_xml_id"/>
</record>
```

There is a shortcut by using the `menuitem` tag that **you should use preferentially**. It offers a flexible way to easily define the menu entry along with icons and other fields.

```
<menuitem id="menu_xml_id"
    name="My Menu"
    action="action_xml_id"
    icon="NAME_FROM_LIST"
    groups="groupname"
    sequence="<integer>"
    parent="parent_menu_xml_id"
/>
```

Where

- `id` specifies the **xml identifier** of the menu item in the menu items table. This identifier must be unique. Mandatory field.
- `name` defines the menu name that will be displayed in the client. Mandatory field.
- `action` specifies the identifier of the attached action defined in the action table (`ir.actions.act_window`). 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. This is how custom icons for "Projects" or "Human Resources" in OpenERP are defined).
- `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*"`).

The main current limitation of using `menuitem` is that the menu action must be an `act_window` action. This kind of actions is the most used action in OpenERP. However for some menus you will use other actions. For example, the Feeds page that comes with the mail module is a client action. For this kind of menu entry, you can combine both declaration, as defined in the mail module :

```
<!-- toplevel menu -->
<menuitem id="mail_feeds_main" name="Feeds" sequence="0"
    web_icon="static/src/img/feeds.png"
    web_icon_hover="static/src/img/feeds-hover.png" />
<record id="mail_feeds_main" model="ir.ui.menu">
    <field name="action" ref="action_mail_all_feeds"/>
</record>
```

Actions

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
- **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 clicks on a menu,
- The user clicks on the icon 'print' or 'action'.

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

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').

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.

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.
- **context** will be passed to the action itself and added to its global context

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

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

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

Examples of actions

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

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

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

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

Url Action

Report Action

Report declaration

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

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

```
            auto="False"/>
    </data>
</openerp>
```

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

- **id** : an identifier which must be unique.
- **string** : the text of the menu that calls the report (if any, see below).
- **model** : the OpenERP object on which the report will be rendered.
- **rml** : the .RML report model. Important Note : Path is relative to addons/ directory.
- **menu** : whether the report will be able to be called directly via the client or not. Setting menu to False is useful in case of reports called by wizards.
- **auto** : determines if the .RML file must be parsed using the default parser or not. Using a custom parser allows you to define additional functions to your report.

Action creation

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:

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

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

1.3.5 Example of module creation

Getting the skeleton directory

Create a `travel` directory, that will contain our addon. Create **__init__.py** file and **__openerp__.py** files.

Edit the **__openerp__.py** module manifest file:

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

Changing the main module file

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

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

The travel.py file should initially look like this:

```
from osv import osv, fields

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

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

Customizing the view

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

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

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

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

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

1.3.6 Module versioning

OpenERP has been developed with modularity in mind: OpenERP should be flexible enough so it can be adopted by any enterprise, of any size and in any market. By using modules one can adapt OpenERP in many different ways: from completely different business to smaller, company-specific changes.

As modules (and the core framework itself) evolve, it is necessary to identify modules by their version so a sensible set of modules can be chosen for a particular deployment.

There are two trends re-inforcing each others. Firstly OpenERP s.a. will work on a smaller number of official modules and let the community handles more and more development. Secondly all those modules will receive greater exposure on OpenERP Apps where each module will be owned by a single author.

The solution advocated by OpenERP is straightforward and aims to avoid the dependency hell. In particular we don't want to deal with versioned dependencies (i.e. a module depends on a specific version of another module).

For each stable release (e.g. OpenERP 6.1, or OpenERP 7.0) or, said in other words, for each major version, there is only one (major) version of each module. The minor version is bumped for bug fixes but is otherwise not important here.

Making variations on some business needs must be done by creating new modules, possibly depending on previously written modules. If depending on a module proves too difficult, you can write a new module (not a new _version_). But generally Python is flexible enough that depending on the existing module should work.

For the next major version, refactoring modules can be done and similar functionalities can be brought together in a better module.

Example

Whenever a new module is developed or must evolve, the above versioning policy should be respected.

A particular concern one can face when deploying OpenERP to multiple customers is now detailed as an example to provide a few guidelines. The hypotethical situation is as follow. A partner needs to create a new module, called `M`, for

a customer. Shortly after (but still within a typical OpenERP release cycle, so there is no chance to bump the version number except for bug fixes), `M` must be adapted for another customer.

The correct way to handle it is to leave `M` as it is and create a new module, say called `X`, that depends on `M` for the second customer. Both modules have the same version (i.e. 6.1 or 7.0, targeting the corresponding OpenERP version).

If leaving `M` as it is is not possible (which should be very rare as Python is incredibly flexible), the `X` module for the new customer can depend on a new module `N`, derived from `M`. At this point, `N` is a new, differently named module. It is not a `M` module with a increased version number. Your goal should be to make `N` as close as possible to `M`, so that at the next version of OpenERP, the first customer can switch to `N` instead of `M` (or include the changes in a new version of `M`). At that point you are in the ideal situation: you have a module `N` for one customer, and a module `X` depending on N to account for the differences between those two customers.

1.4 Security in OpenERP: users, groups

Users and user roles are critical points concerning internal security in OpenERP. OpenERP provides several security mechanisms concerning user roles, all implemented in the OpenERP Server. They are implemented in the lowest server level, which is the ORM engine. OpenERP distinguishes three different concepts:

- user: a person identified by its login and password. Note that all employees of a company are not necessarily OpenERP users; an user is somebody who accesses the application.
- group: a group of users that has some access rights. A group gives its access rights to the users that belong to the group. Ex: Sales Manager, Accountant, etc.
- security rule: a rule that defines the access rights a given group grants to its users. Security rules are attached to a given resource, for example the Invoice model.

Security rules are attached to groups. Users are assigned to several groups. This gives users the rights that are attached to their groups. Therefore controlling user roles is done by managing user groups and adding or modifying security rules attached to those groups.

1.4.1 Users

Users represent physical persons using OpenERP. They are identified with a login and a password,they use OpenERP, they can edit their own preferences, ... By default, a user has no access right. The more we assign groups to the user, the more he or she gets rights to perform some actions. A user may belong to several groups.

1.4.2 User groups

The groups determine the access rights to the different resources. A user may belong to several groups. If he belongs to several groups, we always use the group with the highest rights for a selected resource. A group can inherit all the rights from another group

Figure 3 shows how group membership is displayed in the web client. The user belongs to Sales / Manager, Accounting / Manager, Administration / Access Rights, Administration / Configuration and Human Resources / Employee groups. Those groups define the user access rights.

Figure 3: Example of group membership for a given user

1.4.3 Rights

Security rules are attached to groups. You can assign several security rules at the group level, each rule being of one of the following types :

- access rights are global rights on an object,
- record rules are records access filters,
- fields access right,
- workflow transition rules are operations rights.

You can also define rules that are global, i.e. they are applied to all users, indiscriminately of the groups they belong to. For example, the multi-company rules are global; a user can only see invoices of the companies he or she belongs to.

Concerning configuration, it is difficult to have default generic configurations that suit all applications. Therefore, like SAP, OpenERP is by default pre-configured with best-practices.

1.4.4 Access rights

Access rights are rules that define the access a user can have on a particular object . Those global rights are defined per document type or model. Rights follow the CRUD model: create, read (search), update (write), delete. For example, you can define rules on invoice creation. By default, adding a right to an object gives the right to all records of that specific object.

Figure 4 shows some of the access rights of the Accounting / Accountant group. The user has some read access rights on some objects.

Figure 4: Access rights for some objects.

Record rules

When accessing an object, records are filtered based on record rules. Record rules or access filters are therefore filters that limits records of an object a group can access. A record rule is a condition that each record must satisfy to be created, read, updated (written) or deleted. Records that do not meet the constraints are filtered.

For example, you can create a rule to limit a group in such a way that users of that group will see business opportunities in which he or she is flagged as the salesman. The rule can be salesman = connected_user. With that rule, only records respecting the rule will be displayed.

Field access rights

New in version 7.0.

OpenERP now supports real access control at the field level, not just on the view side. Previously it was already possible to set a `groups` attribute on a `<field>` element (or in fact most view elements), but with cosmetics effects only: the element was made invisible on the client side, while still perfectly available for read/write access at the RPC level.

As of OpenERP 7.0 the existing behavior is preserved on the view level, but a new `groups` attribute is available on all model fields, introducing a model-level access control on each field. The syntax is the same as for the view-level attribute:

```
_columns = {
    'secret_key': fields.char('Secret Key', groups="base.group_erp_manager,base.group_system")
 }
```

There is a major difference with the view-level `groups` attribute: restricting the access at the model level really means that the field will be completely unavailable for users who do not belong to the authorized groups:

- Restricted fields will be **completely removed** from all related views, not just hidden. This is important to keep in mind because it means the field value will not be available at all on the client side, and thus unavailable e.g. for `on_change` calls.
- Restricted fields will not be returned as part of a call to `fields_get()` or `fields_view_get()` This is in order to avoid them appearing in the list of fields available for advanced search filters, for example. This does not prevent getting the list of a model's fields by querying `ir.model.fields` directly, which is fine.
- Any attempt to read or write directly the value of the restricted fields will result in an `AccessError` exception.
- As a consequence of the previous item, restricted fields will not be available for use within search filters (domains) or anything that would require read or write access.
- It is quite possible to set `groups` attributes for the same field both at the model and view level, even with different values. Both will carry their effect, with the model-level restriction taking precedence and removing the field completely in case of restriction.

Note: The tests related to this feature are in `openerp/tests/test_acl.py`.

Workflow transition rules

Workflow transition rules are rules that restrict some operations to certain groups. Those rules handle rights to go from one step to another one in the workflow. For example, you can limit the right to validate an invoice, i.e. going from a draft action to a validated action.

1.4.5 Menu accesses

In OpenERP, granting access to menus can be done using user groups. A menu that is not granted to any group is accessible to every user. It is possible in the administration panel to define the groups that can access a given menu.

However, one should note that using groups to hide or give access to menus is more within the filed of ergonomics or usability than within the field of security. It is a best practice putting rules on documents instead of putting groups on menu. For example, hiding invoices can be done by modifying the record rule on the invoice object, and it is more efficient and safer than hiding menus related to invoices.

1.4.6 Views customization

Customizing views based on groups is possible in OpenERP. You can put rules to display some fields based on group rules. However, as with menu accesses customization, this option should not be considered for security concerns. This way of customizing views belongs more to usability.

1.4.7 Administration

When installing your particular instance of OpenERP, a specific first user is installed by default. This first user is the Super User or administrator. The administrator is by default added access rights to every existing groups, as well as to every groups created during a new module installation. He also has access to a specific administration interface accessible via the administration menu, allowing the administration of OpenERP.

The administrator has rights to manage groups; he can add, create, modify or remove groups. He may also modify links between users and groups, such as adding or removing users. He also manages access rights. With those privileges, the administrator can therefore precisely define security accesses of every users of OpenERP.

There are user groups that are between normal groups and the super user. Those groups are Administration / Configuration and Administration / Access Rights. It gives to the users of those groups the necessary rights to configure access rights.

1.5 Test framework

In addition to the YAML-based tests, OpenERP uses the unittest2 testing framework to test both the core `openerp` package and its addons. For the core and each addons, tests are divided between three (overlapping) sets:

1. A test suite that comprises all the tests that can be run right after the addons is installed (or, for the core, right after a database is created). That suite is called `fast_suite` and must contain only tests that can be run frequently. Actually most of the tests should be considered fast enough to be included in that `fast_suite` list and only tests that take a long time to run (e.g. more than a minute) should not be listed. Those long tests should come up pretty rarely.
2. A test suite called `checks` provides sanity checks. These tests are invariants that must be full-filled at any time. They are expected to always pass: obviously they must pass right after the module is installed (i.e. just like the `fast_suite` tests), but they must also pass after any other module is installed, after a migration, or even after the database was put in production for a few months.
3. The third suite is made of all the tests: those provided by the two above suites, but also tests that are not explicitely listed in `fast_suite` or `checks`. They are not explicitely listed anywhere and are discovered automatically.

As the sanity checks provide stronger guarantees about the code and database structure, new tests must be added to the `checks` suite whenever it is possible. Said with other words: one should try to avoid writing tests that assume a freshly installed/unaltered module or database.

It is possible to have tests that are not listed in `fast_suite` or `checks`. This is useful if a test takes a lot of time. By default, when using the testing infrastructure, tests should run fast enough so that people can use them frequently. One can also use that possiblity for tests that require some complex setup before they can be successfuly run.

As a rule of thumb when writing a new test, try to add it to the `checks` suite. If it really needs that the module it belongs to is freshly installed, add it to `fast_suite`. Finally, if it can not be run in an acceptable time frame, don't add it to any explicit list.

1.5.1 Writing tests

The tests must be developed under `<addons-name>.tests` (or `openerp.tests` for the core). For instance, with respect to the tests, a module `foo` should be organized as follow:

```
foo/
  __init__.py # does not import .tests
  tests/
    __init__.py # import some of the tests sub-modules, and
                # list them in fast_suite or checks
    test_bar.py # contains unittest2 classes
    test_baz.py # idem
    ... and so on ...
```

The two explicit lists of tests are thus the variables `foo.tests.fast_suite` and `foo.tests.checks`. As an example, you can take a look at the `openerp.tests` module (which follows exactly the same conventions even if it is not an addons).

Note that the `fast_suite` and `checks` variables are really lists of module objects. They could be directly unittest2 suite objects if necessary in the future.

1.5.2 Running the tests

To run the tests (see *above* to learn how tests are organized), the simplest way is to use the `oe` command (provided by the `openerp-command` project).

```
> oe run-tests # will run all the fast_suite tests
> oe run-tests -m openerp # will run all the fast_suite tests defined in `openerp.tests`
> oe run-tests -m sale # will run all the fast_suite tests defined in `openerp.addons.sale.tests`
> oe run-tests -m foo.test_bar # will run the tests defined in `openerp.addons.foo.tests.test_bar`
```

In addition to the above possibilities, when invoked with a non-existing module (or module.sub-module) name, oe will reply with a list of available test sub-modules.

Depending on the unittest2 class that is used to write the tests (see `openerp.tests.common` for some helper classes that you can re-use), a database may be created before the test is run, and the module providing the test will be installed on that database.

Because creating a database, installing modules, and then dropping it is expensive, it is possible to interleave the run of the `fast_suite` tests with the initialization of a new database: the dabase is created, and after each requested module is installed, its fast_suite tests are run. The database is thus created and dropped (and the modules installed) only once.

1.5.3 TestCase subclasses

Note: The *setUp* and *tearDown* methods are not part of the tests. Uncaught exceptions in those methods are errors, not test failures. In particular, a failing *setUp* will not be followed by a *tearDown* causing any allocated resource in the *setUp* to not be released by the *tearDown*.

In the `openerp.tests.common.TransactionCase` and `openerp.tests.common.SingleTransactionCase`, this means the test suite can hang because of unclosed cursors.

1.6 Miscellanous

1.6.1 On Change Methods

Definition of on change methods in a view looks like this:

```
<field name="name" on_change="name_change(name, address, city)"/>
```

And here is the corresponding method in the model:

```
def name_change(self, cr, uid, ids, name, address, city, context=None):
    ...
    return {
        'value': {
            'address': ...
            'city': ...
        }
    }
```

On change methods can be confusing when people use them, here are a list of clarifications to avoid any misconception:

- On change methods can be executed during the creation of a row, long before it is effectively saved into the database.

- Fields are *not* validated before going through a on change methods. As an example, a field marqued as required can be False.
- On change methods can read data in the database but should *never* attempt to write anything, this is always a strong conception problem.
- The format of the values passed to an on change method is exactly the same than the one passed to the write() method. So the on change method must be able to handle any format used for all the fields it process. The following list describe some fields that can have an unusual format.
  - *float*: Due to the way JSON represents numbers and the way the JSON library of Python handles it, a float field will not always be represented as a python float type. When the number can be represented as an integer it will appear as a python integer type. This can be a problem when using some mathematical operations (example: price / 2), so it is a good practice to always cast any number to float when you want to handle floats in on change methods.
  - *one2many and many2many*: There are plenty of misconception about x2many fields in on change methods. The reality is, in fact, quite complex. x2many are defined by a list of operations, each operation was given a number (0 -> create, 1 -> write, ect...) and has its own semantic. To be able to use one2many and many2many in on change methods, you are strongly encourage to use the resolve_2many_commands() method. Here is a sample usage:

```
values = self.resolve_2many_commands(cr, uid, 'my_o2m', my_o2m_values, ['price', 'tax'], cor
```

This code will convert the complex list of operations that makes the o2m value into a simple list of dictionaries containing the fields 'price' and 'tax', which is way simpler to handle in most on change methods. Please note that you can also return a list of dictionaries as the new value of a one2many, it will replace the actual rows contained in that one2many (but it will also remove the previous ones).

1.6.2 Font style in list views

New in version 7.0.

This revision adds font styles in list views. Before this revision it was possible to define some colors in list view. This revision allows to define the a font style, based on an evaluated Python expression. The definition syntax is the same than the colors feature. Supported styles are bold, italic and underline.

Rng modification

This revision adds the `fonts` optional attribute in `view.rng`.

Addon implementation example

In your `foo` module, you want to specify that when any record is in `pending` state then it should be displayed in bold in the list view. Edit your foo_view.xml file that define the views, and add the fonts attribute to the tree tag.

```
<tree string="Foo List View" fonts="bold:state=='pending'">
  [...]
</tree>
```

1.6.3 Need action mechanism

New in version 7.0.

ir.needaction_mixin class

New in version openobject-server.4124.

This revision adds a mixin class for objects using the need action feature.

Need action feature can be used by objects willing to be able to signal that an action is required on a particular record. If in the business logic an action must be performed by somebody, for instance validation by a manager, this mechanism allows to set a list of users asked to perform an action.

This class wraps a class (ir.ir_needaction_users_rel) that behaves like a many2many field. However, no field is added to the model inheriting from ir.needaction_mixin. The mixin class manages the low-level considerations of updating relationships. Every change made on the record calls a method that updates the relationships.

Objects using the need_action feature should override the `get_needaction_user_ids` method. This methods returns a dictionary whose keys are record ids, and values a list of user ids, like in a many2many relationship. Therefore by defining only one method, you can specify if an action is required by defining the users that have to do it, in every possible situation.

This class also offers several global services,:

- `needaction_get_record_ids`: for the current model and uid, get all record ids that ask this user to perform an action. This mechanism is used for instance to display the number of pending actions in menus, such as Leads (12)
- `needaction_get_action_count`: as `needaction_get_record_ids` but returns only the number of action, not the ids (performs a search with count=True)
- `needaction_get_user_record_references`: for a given uid, get all the records that ask this user to perform an action. Records are given as references, a list of tuples (model_name, record_id).

New in version openobject-server.4137.

This revision of the needaction_mixin mechanism slighty modifies the class behavior. The `ir_needaction_mixin` class now adds a function field on models inheriting from the class. This field allows to state whether a given record has a needaction for the current user. This is usefull if you want to customize views according to the needaction feature. For example, you may want to set records in bold in a list view if the current user has an action to perform on the record. This makes the class not a pure abstract class, but allows to easily use the action information. The field definition is:

```
def get_needaction_pending(self, cr, uid, ids, name, arg, context=None):
    res = {}
    needaction_user_ids = self.get_needaction_user_ids(cr, uid, ids, context=context)
    for id in ids:
        res[id] = uid in needaction_user_ids[id]
    return res

_columns = {
    'needaction_pending': fields.function(get_needaction_pending, type='boolean',
                    string='Need action pending',
                    help='If True, this field states that users have to perform an action. \
                            This field comes from the needaction mechanism. Please refer \
                            to the ir.needaction_mixin class.'),
}
```

ir.needaction_users_rel class

New in version openobject-server.4124.

This class essentially wraps a database table that behaves like a many2many. It holds data related to the needaction mechanism inside OpenERP. A row in this model is characterized by:

- `res_model`: model of the record requiring an action
- `res_id`: ID of the record requiring an action
- `user_id`: foreign key to the res.users table, to the user that has to perform the action

This model can be seen as a many2many, linking (res_model, res_id) to users (those whose attention is required on the record)

Menu modification

Changed in version openobject-server.4137.

This revision adds three functional fields to `ir.ui.menu` model :

- `uses_needaction`: boolean field. If the menu entry action is an act_window action, and if this action is related to a model that uses the need_action mechanism, this field is set to true. Otherwise, it is false.
- `needaction_uid_ctr`: integer field. If the target model uses the need action mechanism, this field gives the number of actions the current user has to perform.
- **REMOVED** `needaction_record_ids`: many2many field. If the target model uses the need action mechanism, this field holds the ids of the record requesting the user to perform an action. **This field has been removed on version XXXX**.

Those fields are functional, because they depend on the user and must therefore be computed at every refresh, each time menus are displayed. The use of the need action mechanism is done by taking into account the action domain in order to display accurate results. When computing the value of the functional fields, the ids of records asking the user to perform an action is concatenated to the action domain. A counting search is then performed on the model, giving back the number of action the users has to perform, limited to the domain of the action.

Addon implementation example

In your `foo` module, you want to specify that when it is in state `confirmed`, it has to be validated by a manager, given by the field `manager_id`. After making `foo` inheriting from `ir.needaction_mixin`, you override the `get_needaction_user_ids` method:

```
[...]
_inherit = [`ir.needaction_mixin]
[...]
def get_needaction_user_ids(self, cr, uid, ids, context=None):
  result = dict.fromkeys(ids)
  for foo_obj in self.browse(cr, uid, ids, context=context):
    # set the list void by default
    result[foo_obj.id] = []
    # if foo_obj is confirmed: manager is required to perform an action
    if foo_obj.state == 'confirmed':
      result[foo_obj.id] = [foo_obj.manager_id]
  return result
```

1.6.4 User avatar

New in version 7.0.

This revision adds an avatar for users. This replaces the use of gravatar to emulate avatars, used in views like the tasks kanban view. Two fields have been added to the res.users model:

- `avatar_big`, a binary field holding the image. It is base-64 encoded, and PIL-supported. Images stored are resized to 540x450 px, to limitate the binary field size.
- `avatar`, a function binary field holding an automatically resized version of the avatar_big field. It is also base-64 encoded, and PIL-supported. Dimensions of the resized avatar are 180x150. This field is used as an inteface to get and set the user avatar.

When changing the avatar through the avatar function field, the new image is automatically resized to 540x450, and stored in the avatar_big field. This triggers the function field, that will compute a 180x150 resized version of the image.

An avatar field has been added to the users form view, as well as in Preferences. When creating a new user, a default avatar is chosen among 6 possible default images.

1.6.5 Bulk Import

OpenERP has included a bulk import facility for CSV-ish files for a long time. With 7.0, both the interface and internal implementation have been redone, resulting in `load()`.

Note: the previous bulk-loading method, `import_data()`, remains for backwards compatibility but was re-implemented on top of `load()`, while its interface is unchanged its precise behavior has likely been altered for some cases (it shouldn't throw exceptions anymore in many cases where it previously did)

This document attempts to explain the behavior and limitations of `load()`.

Data

The input `data` is a regular row-major matrix of strings (in Python datatype terms, a `list` of rows, each row being a `list` of `str`, all rows must be of equal length). Each row must be the same length as the `fields` list preceding it in the argslist.

Each field of `fields` maps to a (potentially relational and nested) field of the model under import, and the corresponding column of the `data` matrix provides a value for the field for each record.

Generally speaking each row of the input yields a record of output, and each cell of a row yields a value for the corresponding field of the row's record. There is currently one exception for this rule:

One to Many fields

Because O2M fields contain multiple records "embedded" in the main one, and these sub-records are fully dependent on the main record (are no other references to the sub-records in the system), they have to be spliced into the matrix somehow. This is done by adding lines composed *only* of o2m record fields below the main record:

```
+-------+-------+===========+===========+-------+-------+
|value01|value02||o2m/value01|o2m/value02||value03|value04|
+-------+-------+-----------+-----------+-------+-------+
|       |       ||o2m/value11|o2m/value12||       |       |
+-------+-------+-----------+-----------+-------+-------+
|       |       ||o2m/value21|o2m/value22||       |       |
+-------+-------+===========+===========+-------+-------+
|value11|value12||o2m/value01|o2m/value02||value13|value14|
+-------+-------+-----------+-----------+-------+-------+
```

```
|       |       ||o2m/value11|o2m/value12||       |       |
+-------+-------+===========+===========+-------+-------+
|value21|value22|           |           |value23|value24|
+-------+-------+-----------+-----------+-------+-------+
```

the sections in double-lines represent the span of two o2m fields. During parsing, they are extracted into their own `data` matrix for the o2m field they correspond to.

Import process

Here are the phases of import. Note that the concept of "phases" is fuzzy as it's currently more of a pipeline, each record moves through the entire pipeline before the next one is processed.

Extraction

The first phase of the import is the extraction of the current row (and potentially a section of rows following it if it has One to Many fields) into a record dictionary. The keys are the `fields` originally passed to `load()`, and the values are either the string value at the corresponding cell (for non-relational fields) or a list of sub-records (for all relational fields).

This phase also generates the `rows` indexes for any *Messages* produced thereafter.

Conversion

This second phase takes the record dicts, extracts the *database ID* and *external ID* if present and attempts to convert each field to a type matching what OpenERP expects to write.

- Empty fields (empty strings) are replaced with the `False` value
- Non-empty fields are converted through `ir_fields_converter`

Note: if a field is specified in the import, its default will *never* be used. If some records need to have a value and others need to use the model's default, either specify that default explicitly or do the import in two phases.

Char, text and binary fields Are returned as-is, without any alteration.

Boolean fields The string value is compared (in a case-insensitive manner) to `0`, `false` and `no` as well of any translation thereof loaded in the database. If the value matches one of these, the field is set to `False`.

Otherwise the field is compared to `1`, `true` and `yes` (and any translation of these in the database). The field is always set to `True`, but if the value does not match one of these a warning will also be output.

Integers and float fields The field is parsed with Python's built-in conversion routines (`int` and `float` respectively), if the conversion fails an error is generated.

Selection fields The field is compared to 1. the values of the selection (first part of each selection tuple) and 2. all translations of the selection label found in the database.

If one of these is matched, the corresponding value is set on the field.

Otherwise an error is generated.

The same process applies to both list-type and function-type selection fields.

Many to One field If the specified field is the relational field itself (`m2o`), the value is used in a `name_search`. The first record returned by `name_search` is used as the field's value.

If `name_search` finds no value, an error is generated. If `name_search` finds multiple value, a warning is generated to warn the user of `name_search` collisions.

If the specified field is a *external ID* (`m2o/id`), the corresponding record it looked up in the database and used as the field's value. If no record is found matching the provided external ID, an error is generated.

If the specified field is a *database ID* (`m2o/.id`), the process is the same as for external ids (on database identifiers instead of external ones).

Many to Many field The field's value is interpreted as a comma-separated list of names, external ids or database ids. For each one, the process previously used for the many to one field is applied.

One to Many field For each o2m record extracted, if the record has a `name`, *external ID* or *database ID* the *database ID* is looked up and checked through the same process as for m2o fields.

If a *database ID* was found, a LINK_TO command is emmitted, followed by an UPDATE with the non-db values for the relational field.

Otherwise a CREATE command is emmitted.

Date fields The value's format is checked against `DEFAULT_SERVER_DATE_FORMAT`, an error is generated if it does not match the specified format.

Datetime fields The value's format is checked against `DEFAULT_SERVER_DATETIME_FORMAT`, an error is generated if it does not match.

The value is then interpreted as a datetime in the user's timezone. The timezone is specified thus:

- If the import `context` contains a `tz` key with a valid timezone name, this is the timezone of the datetime.
- Otherwise if the user performing the import has a `tz` attribute set to a valid timezone name, this is the timezone of the datetime.
- Otherwise interpret the datetime as being in the `UTC` timezone.

Create/Write

If the conversion was successful, the converted record is then saved to the database via `(ir.model.data)._update`.

Error handling

The import process will only catch 2 types of exceptions to convert them to error messages: `ValueError` during the conversion process, and sub-exceptions of `psycopg2.Error` during the create/write process.

The import process uses savepoint to:

- protect the overall transaction from the failure of each `_update` call, if an `_update` call fails the savepoint is rolled back and the import process keeps going in order to obtain as many error messages as possible during each run.
- protect the import as a whole, a savepoint is created before starting and if any error is generated that savepoint is rolled back. The rest of the transaction (anything not within the import process) will be left untouched.

Messages

A message is a dictionary with 5 mandatory keys and one optional key:

`type` the type of message, either `warning` or `error`. Any `error` message indicates the import failed and was rolled back.

`message` the message's actual text, which should be translated and can be shown to the user directly

`rows` a dict with 2 keys `from` and `to`, indicates the range of rows in `data` which generated the message

`record` a single integer, for warnings the index of the record which generated the message (can be obtained from a non-false `ids` result)

`field` the name of the (logical) OpenERP field for which the error or warning was generated

`moreinfo` (optional) A string, a list or a dict, leading to more information about the warning.

- If `moreinfo` is a string, it is a supplementary warnings message which should be hidden by default
- If `moreinfo` is a list, it provides a number of possible or alternative values for the string
- If `moreinfo` is a dict, it is an OpenERP action descriptor which can be executed to get more information about the issues with the field. If present, the `help` key serves as a label for the action (e.g. the text of the link).

1.6.6 Perfoming joins in select

New in version 7.0.

Starting with OpenERP 7.0, an `auto_join` attribute is added on *many2one* and *one2many* fields. The purpose is to allow the automatic generation of joins in select queries. This attribute is set to False by default, therefore not changing the default behavior. Please note that we consider this feature as still experimental and should be used only if you understand its limitations and targets.

Without `_auto_join`, the behavior of expression.parse() is the same as before. Leafs holding a path beginning with many2one or one2many fields perform a search on the relational table. The result is then used to replace the leaf content. For example, if you have on res.partner a domain like `[('bank_ids.name', 'like', 'foo')]` with bank_ids linking to res.partner.bank, 3 queries will be performed :

- 1 on res_partner_bank, with domain `[('name', '=', 'foo')]`, that returns a list of res.partner.bank ids (bids)
- 1 on res_partner, with a domain `['bank_ids', 'in', bids)]`, that returns a list of res.partner ids (pids)
- 1 on res_partner, with a domain `[('id', 'in', pids)]`

When the `auto_join` attribute is True on a relational field, the destination table will be joined to produce only one query.

- the relational table is accessed using an alias: `'"res_partner_bank" as res_partner__bank_ids`. The alias is generated using the relational field name. This allows to have multiple joins with different join conditions on the same table, depending on the domain.

- there is a join condition between the destination table and the main table: `res_partner__bank_ids."partner_id"=res_partner."id"`
- the condition is then written on the relational table: `res_partner__bank_ids."name" = 'foo'`

This manipulation is performed in expression.parse(). It checks leafs that contain a path, i.e. any domain containing a '.'. It then checks whether the first item of the path is a *many2one* or *one2many* field with the `auto_join` attribute set. If set, it adds a join query and recursively analyzes the remaining of the leaf, using the same behavior. If the remaining path also holds a path with auto_join fields, it will add all tables and add every necessary join conditions.

Chaining joins allows to reduce the number of queries performed, and to avoid having too long equivalent leaf replacement in domains. Indeed, the internal queries produced by this behavior can be very costly, because they were generally select queries without limit that could lead to huge ('id', 'in', [...]) leafs to analyze and execute.

Some limitations exist on this feature that limits its current use as of version 7.0. **This feature is therefore considered as experimental, and used to speedup some precise bottlenecks in OpenERP**.

List of known issues and limitations:

- using `auto_join` bypasses the business logic; no name search is performed, only direct matches between ids using join conditions
- ir.rules are not taken into account when analyzing and adding the join conditions

List of already-supported corner cases :

- one2many fields having a domain attribute. Static domains as well as dynamic domain are supported
- auto_join leading to functional searchable fields

Typical use in OpenERP 7.0:

- in mail module: notification_ids field on mail_message, allowing to speedup the display of the various mailboxes
- in mail module: message_ids field on mail_thread, allowing to speedup the display of needaction counters and documents having unread messages

1.7 Deploying with Gunicorn

Starting with OpenERP 6.1, the server and web addons are WSGI compliant. In particular, support for the Gunicorn HTTP server is available. For some background information and motivation, please read http://www.openerp.com/node/1106. To install Gunicorn, please refer to Gunicorn's website.

1.7.1 Summary

Configuring and starting an OpenERP server with Gunicorn is straightfoward. The different sections below give more details but the following steps are all it takes:

```
1. Use a configuration file, passing it to ``gunicorn`` using the ``-c``
   option.
2. Within the same configuration file, also configure OpenERP.
3. Run ``gunicorn openerp:wsgi.core.application -c gunicorn.conf.py``.
```

1.7.2 Sample configuration file

A sample `gunicorn.conf.py` configuration file for Gunicorn can be found in the OpenERP server source tree. It is fairly well commented and easily customizable for your own usage. While reading the remaining of this page, it is advised you take a look at the sample `gunicorn.conf.py` file as it makes things easier to follow.

1.7.3 Configuration

Gunicorn can be configured by a configuration file and/or command-line arguments. For a list of available options, you can refer to the official Gunicorn documentation http://gunicorn.org/configure.html.

When the OpenERP server is started on its own, by using the `openerp-server` script, it can also be configured by a configuration file or its command-line arguments. But when it is run via Gunicorn, it is no longer the case. Instead, as the Gunicorn configuration file is a full-fledged Python file, we can `import openerp` in it and configure directly the server.

The principle can be summarized with this three lines (although they are spread across the whole sample `gunicorn.conf.py` file):

```
import openerp
conf = openerp.tools.config
conf['addons_path'] = '/home/openerp/addons/trunk,/home/openerp/web/trunk/addons'
```

The above three lines first import the `openerp` library (i.e. the one containing the OpenERP server implementation). The second one is really to shorten repeated usage of the same variable. The third one sets a parameter, in this case the equivalent of the `--addons-path` command-line option.

Finally, Gunicorn offers a few hooks so we can call our own code at some points in its execution. The most important one is the `on_starting` hook. It lets us properly initialize the `openerp` library before Gunicorn starts handling requests. `pre_request` and `post_request` are called before and after requests are handled. We provide functions in `openerp.wsgi.core` that can be used to define those hooks: a typical Gunicorn configuration for OpenERP will thus contains:

```
on_starting = openerp.wsgi.core.on_starting
pre_request = openerp.wsgi.core.pre_request
post_request = openerp.wsgi.core.post_request
```

1.7.4 Running

Once a proper configuration file is available, running the OpenERP server with Gunicorn can be done with the following command:

```
> gunicorn openerp:wsgi.core.application -c gunicorn.conf.py
```

`openerp` must be importable by Python. The simplest way is to run the above command from the server source directory (i.e. the directory containing the `openerp` module). Alternatively, the module can be installed on your machine as a regular Python library or added to your `PYTHONPATH`.

1.7.5 Running behind a reverse proxy

If you intend to run Gunicorn behind a reverse proxy (nginx is recommended), an alternative entry point is available in `openerp.wsgi.proxied`. That entry point uses werkzeug's ProxyFix class to set a few headers. You first have to explicitely import that sub-module if you want to use it. So add this line in the configuration file:

```
import openerp.wsgi.proxied
```

and then adapt the command-line:

```
> gunicorn openerp:wsgi.proxied.application -c gunicorn.conf.py
```

CHAPTER 2

OpenERP Server API

2.1 ORM and models

CHAPTER 3

Concepts

Database ID The primary key of a record in a PostgreSQL table (or a virtual version thereof), usually varies from one database to the next.

External ID

D

E