Odoo Module Guideline

1. Introduction

This page introduces the coding guidelines for modules which are built by OCE Vietnam.

1.1. Modules

Use of the singular form in module name (or use "multi"), except when compound of module name or object Odoo that is already in the plural. If your module's purpose is to serve as a base for other modules, prefix its name with base_.

When extending an Odoo module, prefix yours with that module's name

In **__manifest__.py**:

- Avoid empty keys
- Make sure it has license and images keys
- Make sure the text, Porcities, is appended to the author text.

1.2. Version number

The version number in the module manifest should follow the semantic versioning rule

- **Major**: Odoo version
- **Minor**: increments when non-breaking new features are added. A module upgrade will probably be needed.
- **Patch**: increments when bug fixes were made. Usually a server restart is needed for the fixes to be made available.

1.3. Directories

A module is organized in a few directories:

controllers/ : contains controllers (http routes)
data/ : data xml
models/ : model definitions
report/ : reporting models (BI/analysis), Webkit/RML print report templates
static/ : contains the web assets, separated into css/, js/, img/, etc.
templates/: if you have several web templates and several backend views you can slip them here
views/ : contains the views and templates, and QWeb report print template
wizards/ : wizard model and views

1.4. File naming

For models, views and data declarations, split files by the model involved, either created or inherited. When they are XML files, a suffix should be included with its category.

For example: a view for res_partner model should go in a filename ***views/res_partner_views.xml***.

An exception can be made when the model is a model intended to be used only as a ***one2many*** model nested on the main model. In this case, you can include the model definition inside it.

Example: ***sale.order.line*** model can be together with sale.order in the file ***models/sale_order.py***.

For model named <main_model> the following files may be created:

- ***models/<main_model>***.py
- ***data/<main_model>_data***.xml
- ***templates/<main_model>_template***.xml
- ***views/<main_model>_views***.xml

For controllers, if there is **only one file** it should be named ***main.py***. If there are **several controller classes or functions** you can split them into **several files**.

For static files, the name pattern is **<model_name>**.ext (i.e. static/js/im_chat.js, static/css/im_chat.css, etc.). Do not link data (image, libraries) outside Odoo: do not use an url to an image but copy it in our codebase instead.

1.5. Installation hooks

When "***pre_init_hook***", "***post_init_hook***", "***uninstall_hook***" and "***post_load***" are used, they should be placed in "***hooks.py***" located at the root of module directory structure and keys in the manifest file keeps the same as the following

```
#__manifest__.py
{
 .....
 'pre_init_hook': 'pre_init_hook',
 'post_init_hook': 'post_init_hook',
 'uninstall_hook': 'uninstall_hook',
 'post_load': 'post_load',
 ....
}
```

Remember to add into the "__init__.py" the following imports as needed

```
#__init__.py
from .hooks import pre_init_hook, post_init_hook, uninstall_hook,
post_load
```

For applying monkey patches use post_load hook. In order to apply them just if the module is installed.

1.6. Complete structure

The complete tree should look like this

```
addons/<my_module_name>/
 |-- __init__.py
 |-- __manifest__.py
 |-- hooks.py
 |-- controllers/
 | |-- __init__.py
 | `-- main.py
 |-- data/
 | `-- <main_model>.xml
 |-- models/
 | |-- __init__.py
 | |-- <main_model>.py
 | `-- <inherited_model>.py
 |-- report/
 | |-- __init__.py
 | |-- report.xml
 | |-- <bi_reporting_model>.py
 | |-- report_<rml_report_name>.rml
 | |-- report_<rml_report_name>.py
 | `-- <webkit_report_name>.mako
 |-- security/
 | |-- ir.model.access.csv
 | `-- <main_model>_security.xml
 |-- static/
 | |-- img/
 | | |-- my_little_kitten.png
 | | `-- troll.jpg
 | |-- lib/
 | | `-- external_lib/
 | `-- src/
 | |-- js/
 | | `-- <my_module_name>.js
 | |-- css/
```

```
| | `-- <my_module_name>.css
| |-- less/
| | `-- <my_module_name>.less
| `-- xml/
| `-- <my_module_name>.xml
|-- tests/
| |-- __init__.py
| |-- <test_file>.py
| `-- <test_file>.yml
|-- views/
| |-- <main_model>_views.xml
| |-- <inherited_main_model>_views.xml
| `-- report_<qweb_report>.xml
|-- templates/
| |-- <main_model>.xml
| `-- <inherited_main_model>.xml
|-- wizards/
| |-- __init__.py
| |-- <wizard_model>.py
| `-- <wizard_model>.xml
```

1.7. External dependencies

In python files where you use external dependencies, you will need to add try-except with a debug log.

```
try:
 import external_dependency
except (ImportError, IOError) as err:
 _logger.debug(err)
```

2. XML files

2.1. Format

When declaring a record in XML:

- Indent using four spaces
- Place id attribute before model
- For field declarations, the name attribute is first. Then place the value either in the field tag, either in the eval attribute, and finally other attributes (widget, options, etc.) ordered by importance
- Try to group the records by model. In case of dependencies between action/menu/views, the convention may not be applicable. Use the naming convention defined at the next point.
- The tag <data> is only used to set not-updated data with noupdate=1 when your data file contains a mix of "noupdate" data files containing a mix of "noupdate" data. Otherwise, you should use on of these:
    - <odoo>: for noupdate=0 or demo data (demo data is non-updatable by default)
    - <odoo noupdate='1'>
- Do not prefix the xmlid by the current module's (<record id="view_id"..., not <record id="current_module.view_id")

```
<record id="view_id" model="ir.ui.view">
 <field name="name">view.name</field>
 <field name="model">object_name</field>
 <field name="priority" eval="16"/>
 <field name="arch" type="xml">
 <tree>
 <field name="my_field_1"/>
  <field name="my_field_2" string="My Label" widget="statusbar"
statusbar_visible="draft,sent,                    progress,done"
statusbar_colors='{"invoice_except":"red","waiting_date":"blue"}
' />  </tree>
 </field>
</record>
```

2.2. Naming xml_id

2.2.1. Data Records

Use the following pattern, where <model_name> is the name of the model that the record is an instance of: <model_name>_<record_name>

```
<record id="res_users_important_person" model="res.users">
 ...
</record>
```

2.2.2. Security, View and Action

Use the following patterns, where <model_name> is the name of the model that the menu, view, etc. belongs to (e.g. for a res.users form view, the name would be res_users_view_form):

- For a **menu**: ***<model_name>_menu***
- For a **view**: ***<model_name>_view_<view_type>***, where ***view_type*** is kanban, form, tree, search, etc.
- For an **action**: the main action respects ***<model_name>_action***. Others are suffixed with***_<detail>***, where detail is an underscore lowercase string explaining the action (should not be long). This is used only if multiple actions are declared for the model.
- For a **group**: ***<model_name>_group_<group_name>*** where ***group_name*** is the name of the group, generally "user", "manager", etc.
- For a **rule**: ***<model_name>_rule_<concerned_group>*** where concerned_group is the short name of the concerned group ("user" for the "model_name_group_user", "public" for public user, "company" for multi-company rules, etc.).

```
<!-- views and menus -->
<record id="model_name_menu" model="ir.ui.menu">
 ...
</record>

<record id="model_name_view_form" model="ir.ui.view">
 ...
</record>

<record id="model_name_view_kanban" model="ir.ui.view">
 ...
</record>

<!-- actions -->
<record id="model_name_action" model="ir.actions.act_window">
 ...
</record>

<record                                  id="model_name_action_child_list"
model="ir.actions.act_window">
 ...
</record>

<!-- security -->
<record id="model_name_group_user" model="res.groups">
 ...
</record>

<record id="model_name_rule_public" model="ir.rule">
 ...
</record>

<record id="model_name_rule_company" model="ir.rule">
 ...
</record>
```

2.2.3. Inherited XML

A module can extend a view only one time.

The naming rules should be followed even when a view is inherited, the module name prevents xid conflicts. In the case where an inherited view has a name which does not follow the guidelines set above, prefer naming the inherited view after the original over using a name which follows the guidelines. This eases looking up the original view and other inheritance if they all have the same name.

```
<record id="original_id" model="ir.ui.view">
 <field name="inherit_id" ref="original_module.original_id"/>
 ...
</record>
```

Use of ***<...position="replace">*** **is not recommended** because could show the error Element... cannot be located in the parent view from other inherited views with this field. If you need to use this option, **it must have an explicit comment explaining** why it is absolutely necessary and also use a high value in its priority (**greater than 100 is recommended**) to avoid the error.

```
<record id="view_id" model="ir.ui.view">
 <field name="name">view.name</field>
 <field name="model">object_name</field>
 <field name="priority">110</field> <!--Priority greater than 100-->
 <field name="arch" type="xml">
 <!-- It is necessary because...-->
 <xpath expr="//field[@name='my_field_1']" position="replace"/>
 </field>
</record>
```

Also, we can hide an element from the view using **invisible="1".**

3. Python

3.1. PEP8 options

Using the linter flake8 can help to see syntax and semantic warnings or errors. Project Source Code should adhere to PEP8 and PyFlakes standards with a few exceptions.

3.2. Imports

The imports are ordered as

1. Standard library imports
2. Known third party imports (One per line sorted and split in python stdlib)
3. Odoo imports (odoo)
4. Imports from Odoo modules (rarely, and only if necessary)
5. Local imports in the relative form
6. Unknown third party imports (One per line sorted and split in python stdlib)

Inside these 6 groups, the imported lines are alphabetically sorted.

```
# 1: imports of python lib
import base64
import logging
import re
import time

# 2: import of known third party lib
import lxml

# 3: imports of odoo
import odoo
from odoo import api, fields, models # alphabetically ordered
from odoo.tools.safe_eval import safe_eval
from odoo.tools.translate import _

# 4: imports from odoo modules
from odoo.addons.website.models.website import slug
from odoo.addons.web.controllers.main import login_redirect

# 5: local imports
from . import utils

# 6: Import of unknown third party lib
```

```
_logger = logging.getLogger(__name__)
try:
 import external_dependency_python_N
except ImportError:
 _logger.debug('Cannot `import external_dependency_python_N`.')
```

****Note:** You can use isort to automatically sort imports. Install with pip install isort and use with isort myfile.py.

3.3. Idioms

For Python 3 (Odoo >= 11.0), **no need** for utf-8 coding line as this is implicit.

Prefer **% over.format(), prefer %(varname) instead of positional**. This is better for translation and security.

Always favor Readability over conciseness or using the language features or idioms.

Use list comprehension, dict comprehension, and basic manipulation using map, filter, sum, etc.. They make the code more pythonic, easier to read and are generally more efficient. The same applies for recordset methods: use filtered, mapped, sorted, etc.

Exceptions: Use from **odoo.exceptions import UserError** or find more appropriate exception in odoo.exceptions.py

Document your code: Docstring on methods should explain the purpose of a function, not a summary of the code. Simple comments for parts of code which do things are not immediately obvious. **Too many comments** are usually a sign that the code is unreadable and **needs to be refactored**.

Use **meaningful** variable/class/method names

If **a function is too long or too indented** due to loops, this is a sign that it **needs to be refactored into smaller functions**.

If a function call, dictionary, or list is **broken in two lines**, break it at the opening symbol. This adds a four space indent to the next line instead of starting the next line at the opening symbol. Example:

```
partner_id = fields.Many2one(
 "res.partner",
 "Partner",
 "Required",)
```

When making a comma separated list, dict, tuple, etc.. with one element per line, append a comma to the last element. This makes it so the next element added only changes one line in the changeset instead of changing the last element to simply add a comma.

If an argument to a function call is not immediately obvious, prefer using named parameters.

Use English variable names and write comments in English. String which needs to be displayed in other languages should be translated using the translation system.

3.4. Symbols

3.4.1. Odoo Python Classes

Use UpperCamelCase for code

```
class AccountInvoice(models.Model):
 ...
```

3.4.2. Variable names

Always give your variables **a meaningful name**. You may know what it's referring to now, but you won't in 2 months, and others don't either. One letter variables are acceptable only in lambda expressions and loop indices, or perhaps in pure math expressions (and even there it doesn't hurt to use a real name).

```
# unclear and misleading
a = {}
sfields = {}

# better
results = {}
selected_fields = {}
```

Use **underscore lowercase notation** for common variables (snake_case)

Do not suffix variable names with **_id or _ids** if they **do not contain an ids or lists of ids**.

```
res_partner = self.env['res.partner']
partners = res_partner.browse(ids)
partner_id = partners[0].id
```

Use underscore uppercase notation for global variables or constants

```
CONSTANT_VAR1 = 'Value'
...
class ...
...
```

3.5. Models

Model names

- Use dot lowercase name for models. Example: sale.order
- Use names in a singular form. sale.order instead of sale.orders

Method conventions

- **Compute field**: the compute method pattern is **_compute_<field_name>**
- **Inverse method**: the inverse method pattern is **_inverse_<field_name>**
- **Search method**: the search method pattern is **_search_<field_name>**
- **Default method:** the default method pattern is **_default_<field_name>**
- **Onchange method**: the onchange method pattern is **_onchange_<field_name>**
- **Constraint method**: the constraint method pattern is **_check_<field_name>**
- **Action method**: an object action method is prefixed with **action_** . If the method uses only **one record**, add **self.ensure_one()** at the beginning of the method.
- **In a model attribute order** should be
    - Private attribute (_name, _description, _inherit, etc.)
    - Fields declarations
    - SQL constraints
    - Default method and _default_get
    - Compute and search methods in the same order than field declaration
    - Constraints methods (@api.constrains) and onchange methods (@api.onchange)
    - CRUD methods (ORM overrides)
    - Action methods
    - Other business methods

```
class Event(models.Model):
 # Private attributes
 _name = 'event.event'
 _description = 'Event'

 # Fields declaration
 name = fields.Char(default=lambda self: self._default_name())
 seats_reserved = fields.Integer(
 oldname='register_current',
 string='Reserved Seats',
 store=True,
 readonly=True,
 compute='_compute_seats',
 )
 seats_available = fields.Integer(
 oldname='register_avail',
 string='Available Seats',
 store=True,
 readonly=True,
 compute='_compute_seats',
 )
 price = fields.Integer(string='Price')

 # SQL constraints
 _sql_constraints = [
 ('name_uniq', 'unique(name)', 'Name must be unique'),
 ]

 # Default methods
 def _default_name(self):
 ...

  # compute and search fields, in the same order that fields
declaration
 @api.multi
 @api.depends('seats_max', 'registration_ids.state')
 def _compute_seats(self):
 ...
```

```
# Constraints and onchanges
@api.constrains('seats_max', 'seats_available')
def _check_seats_limit(self):
...

@api.onchange('date_begin')
def _onchange_date_begin(self):
...

# CRUD methods
def create(self):
...

# Action methods
@api.multi
def action_validate(self):
self.ensure_one()
...

# Business methods
def mail_user_confirm(self):
...
```

3.6. Fields

One2many and **Many2many** fields should always have ids as suffix (example: sale_order_line_ids)

Many2one fields should have id as suffix (example: partner_id, user_id, etc.)

Default functions should be declared with a lambda call on self. The reason for this is so a default function **can be inherited**. Assigning a function pointer directly to the default parameter does not allow for inheritance.

```
a_field(..., default=lambda self: self._default_get())
```

4. Tests

As a general rule, a bug fix should come with a unit test which would fail without fixing itself. This is to assure that regression will not happen in the future. It also is a good way to show that the fix works in all cases.

New modules or additions should ideally test all the functions defined. The coveralls utility will comment on pull requests indicating if coverage increased or decreased. If it has decreased, this is usually a sign that a test should be added. The overall web interface can also show which lines need test cases.

5. README.md

A README file is a guide that gives users a detailed description of a project that you have pushed to your repository.
A good README file should include:

- Project's title
  This is the name of the project. It describes the whole project in one sentence, and helps people understand what the main goal and aim of the project is.
- Description
  Your description is an extremely important aspect of your project. This is an important component of your project that many new developers often overlook.
  A good one takes advantage of the opportunity to explain and showcase:
  - What your application does
  - Why you used the technologies you used
  - Some of the challenges you faced and features you hope to implement in the future.

  A good README helps you stand out among the large crowd of developers who put their work on Github.
- Table of Contents (Optional)
  If your README is very long, you might want to add a table of contents to make it easy for users to find what they need. It helps them navigate to different parts of the file.
- Installation Instruction
  You should include the steps required to install your project such as external libraries used, etc.
- Usage (Optional)
  Provide instructions and examples so users/contributors can use the project. This will make it easy for them in case they encounter a problem - they will always have a place of reference.
  You can also include screenshots to show examples of the running project, especially new settings, new workflows, etc.

- Credits
  You need to list your collaborators/team members. Also, if you followed tutorials to build that particular project, include links to those here as well. This is just a way to show your appreciation and also to help others get a first hand copy of the project
  There are 3 things we need to mention for this:
    - Authors
    - Contributors
    - Maintainers
- Badges
  For our modules, we need to include these badges:

Badge	Content	
License OPL-1	Depend on your Odoo version: https://www.odoo.com/documentation/12.0/legal/licenses.html https://www.odoo.com/documentation/13.0/legal/licenses.html https://www.odoo.com/documentation/14.0/legal/licenses.html	
Github URL	Github URL	