OCA Guidelines (2018/08/24)

Table of contents

Introduction

This page introduces the coding guidelines for projects hosted under OCA. These guidelines aim to improve the quality of the code: better readability of source, better maintainability, better stability and fewer regressions.

These are loosely based on the Odoo Guidelines and Old Odoo Guidelines with adaptations to improve their guidelines and make them more suitable for this project's own needs. Readers used to the Odoo Guidelines can skip to the Differences With Odoo Guidelines section.

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 (i.e. mrp_operations_...).
- If your module's purpose is to serve as a base for other modules, prefix its name with base_. I.e. base_location_nuts.
- When creating a localization module, prefix its name with l10n_CC_, where CC is its country code. I.e. l10n_es_pos.
- When extending an Odoo module, prefix yours with that module's name. I.e. mail_forward.
- When combining an Odoo module with another from OCA, Odoo's name goes before. I.e., if you want to combine Odoo's crm with OCA's partner_firstname, the name should be crm_partner_firstname.
- Use the description template but remove sections with no meaningful content.
- In '___manifest___.py':
  - Avoid empty keys
  - Make sure it has the license and images keys.
  - Make sure the text ,Odoo Community Association (OCA) is appended to the author text.
  - The website key must be https://github.com/OCA/<repo>, so as to provide the most relevant link to discover more information about

the addon. That link shows the repository README. Alternatively https://github.com/OCA/<repo>/tree/<branch>/<addon> may be used, to provide a direct link to the addon README, which includes proper credits (authors, contributors and theirs companies), and links to the relevant information on the OCA website.

- Don't use your company logo or your corporate branding. Using the author and the list of contributors is enough for people to know about your employer/company and contact you.

Version numbers

The version number in the module manifest should be the Odoo major version (e.g. 12.0) followed by the module x.y.z version numbers. For example: 12.0.1.0.0 is expected for the first release of an 12.0 module.

The x.y.z versioning scheme follows the major.minor.patch convention:

- x (Major): Increments when there are significant changes to the data model or views. These changes may require data migration and could impact dependent modules.
- y (Minor): Increments when new features are added that do not break backward compatibility. A module upgrade will likely be necessary.
- z (Patch): Increments when bug fixes are made. Typically, a server restart is needed to apply these fixes.

When introducing breaking changes, it is expected that you include instructions or scripts to facilitate migration on current installations.

For more details on semantic versioning, please consult the SemVer documentation.

Migrations

When you introduce a breaking change, you *must* provide a migration script to make it possible to upgrade from lower versions. For a migration to another major version of Odoo, it's quite probable you'll need a migration script too. In such cases, migration scripts are highly appreciated, but a note in the README about relevant changes needing migration is sufficient too so that later contributors can add migration scripts without having to analyze all changes again.

For forward porting a module, consult: https://github.com/OCA/maintainer-tools/wiki#migration

Directories and files

A module is organized in a few directories and files:

- 'controllers/': contains controllers (http routes)
- 'data/': data xml

- ‘demo/‘: demo xml
- ‘examples/: external files ‘lib/, ...
- ‘i18n/: ‘translations <#translations>_
- ‘migrations/‘: directory (e.g. 15.0.1.0.0) with scripts to run before or after updating the module
- ‘models/‘: model definitions
- ‘readme/: files (e.g. DESCRIPTION.rst, USAGE.rst, CONTRIBUTORS.rst) used to build the module’s ‘README.rst <https://github.com/OCA/maintainer-tools/tree/master/template/module/README.rst>_ file.
- ‘reports/‘: reporting models (BI/analysis), Webkit/RML print report templates
- ‘security/‘: security groups, giving them access to models, with rules to limit the access to some records
- ‘static/: contains the web assets, separated into ‘css/, js/, img/,
- ‘templates/‘: if you have several web templates and several backend views you can split them here
- ‘tests/: ‘unit tests <#tests>_ to assure that the module works as expected
- ‘views/‘: contains the views and templates, and QWeb report print templates
- ‘wizards/‘: wizard model and views
- ‘README.rst‘: is built from the files in the readme directory. You should not create or modify it
- ‘___init___.py‘: python file specifying which python files/directories to load
- ‘___manifest___.py‘: information about the module (in JSON format)
- ‘exceptions.py‘: for validation errors
- ‘hooks.py‘: hooks to load before/after installing, before uninstalling, after loading

See complete structure below.

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, demo data for res.partner should go in a file named demo/res_partner_demo.xml and a view for partner should go in a file named 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
- demo/<main_model>_demo.xml
- templates/<main_model>_template.xml
- views/<main_model>_views.xml

For controller, 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 <module_name>.ext (i.e. static/js/im_chat.js, static/css/im_chat.css, static/xml/im_chat.xml, ...). Don't link data (image, libraries) outside Odoo: don't use an url to an image but copy it in our codebase instead.

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

```
{
    '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. For example:

```
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.

Complete structure

The complete tree should look like this:

```
addons/<my_module_name>/
|-- controllers/
|   |-- __init__.py
|   `-- main.py
|-- data/
|   `-- <main_model>.xml
|-- demo/
|   `-- <inherited_model>.xml
|-- examples/
|   `-- my_example.csv
```

```
|-- i18n/
|   |-- en_GB.po
|   |-- es.po
|   `-- module_name.pot
|-- migrations/
|   `-- 16.0.x.y.z/
|       |-- pre-migration.py
|       `-- post-migration.py
|-- models/
|   |-- __init__.py
|   |-- <main_model>.py
|   `-- <inherited_model>.py
|-- readme/
|   |-- CONTRIBUTORS.rst
|   |-- DESCRIPTION.rst
|   `-- USAGE.rst
|-- reports/
|   |-- __init__.py
|   |-- reports.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
|-- templates/
|   |-- <main_model>.xml
|   `-- <inherited_main_model>.xml
|-- tests/
|   |-- __init__.py
```

```
|   |-- <test_file>.py
|   `-- <test_file>.yml
|-- views/
|   |-- <main_model>_views.xml
|   |-- <inherited_main_model>_views.xml
|   `-- report_<qweb_report>.xml
|-- wizards/
|   |-- __init__.py
|   |-- <wizard_model>.py
|   `-- <wizard_model>.xml
|-- README.rst
|-- __init__.py
|-- __manifest__.py
|-- exceptions.py
`-- hooks.py
```

Filenames should use only [a-z0-9_]

Use correct file permissions: folders 755 and files 644.

External dependencies

Manifest

___manifest___.py

If your module uses extra dependencies of python or binaries you should add the external_dependencies section to ___manifest___.py.

```
{
    'name': 'Example Module',
    'external_dependencies': {
        'bin': [
            'external_dependency_binary_1',
            'external_dependency_binary_2',
        ],
        'python': [
            'external_dependency_python_1',
            'external_dependency_python_2',
        ],
    },
    'installable': True,
}
```

An entry in bin needs to be in PATH, check by running which external_dependency_binary_N.

An entry in python needs to be in PYTHONPATH, check by running python -c "import external_dependency_python_N".

ImportError

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

```
try:
    import external_dependency_python_N
    import external_dependency_python_M
    EXTERNAL_DEPENDENCY_BINARY_N_PATH = tools.find_in_path('external_dependency_binary_N')
    EXTERNAL_DEPENDENCY_BINARY_M_PATH = tools.find_in_path('external_dependency_binary_M')
except (ImportError, IOError) as err:
    _logger.debug(err)
```

This rule doesn't apply to the test files since these files are loaded only when running tests and in such a case your module and their external dependencies are installed.

This rule doesn't apply neither to Odoo >= v12, as an unmet dependency in an uninstalled module doesn't block the service thanks to this commit:

https://github.com/odoo/odoo/commit/8226aa1db828d2a559c7ffaa31a27ef3e5ba4d0b

README

If your module uses extra dependencies of python or binaries, please explain how to install them in the README.rst file in the section Installation.

requirements.txt

As specified in the Repositories Section, you should also define the python packages to install in a file requirements.txt in the root folder of the repository. This will be used for travis.

oca_dependencies.txt

List the OCA project dependencies, one per line Add a repository url and branch if you need a forked version

Examples:

To depend on the standard version of sale-workflow, use:

```
sale-workflow
```

To explicitely give the URL of a fork, and still use the version specified in `.travis.yml`, use:

```
sale-workflow https://github.com/OCA/sale-workflow
```

To provide both the URL and a branch, use:

```
sale-workflow https://github.com/OCA/sale-workflow branchname
```

To use a specific commit version, set the branch (required) and the commit SHA to select:

```
sale-workflow https://github.com/OCA/sale-workflow branchname f848e37
```

XML files

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, ...) 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 naming convention defined at the next point
- The tag <data> is only used to set not-updatable data with noupdate=1 when your data file contains a mix of "noupdate" data. Otherwise, you should use one 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 name (<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=
        </tree>
    </field>
</record>
```

Records

- For records of model ir.filters use explicit user_id field.

```
<record id="filter_id" model="ir.filters">
    <field name="name">Filter name</field>
```

```
    <field name="model_id">filter.model</field>
    <field name="user_id" eval="False"/>
</record>
```

More info here.

Views

- For v8 and above it is recommended to avoid using the string attribute on list views (<tree>) which has been deprecated and is no longer displayed.
- For v9 and above it is recommended to avoid using the colors and fonts attributes on list views (<tree>) which have been deprecated in favor of decoration-{$name}.

QWeb

- t-*-options QWeb directives (t-field-options, t-esc-options and t-raw-options) should not be used in v10 and above, as they are to be removed after version 10.

Naming xml_id

Data Records

Use the followng 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>
```

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, ...
- 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', ...

- 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, ...).

```
<!-- 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>
```

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 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".

Demo Records

Suffix all demo record XML IDs with demo. This allows them to be easily distinguished from regular records, which otherwise requires examining the source or reinstalling the module with demo data disabled.

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

Python

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:

- In ___init___.py only
  - F401: module imported but unused

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.

Idioms

- 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, ... They make the code more pythonic, easier to read and are generally more efficient
- The same applies for recordset methods: use filtered, mapped, sorted, ...
- Exceptions: Use from odoo.exceptions import Warning as UserError (v8) or from odoo.exceptions import UserError (as of v9) or find a 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 which 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, list or tuple is broken into 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, ... 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 parameter.

- Use English variable names and write comments in English. Strings which need to be displayed in other languages should be translated using the translation system
- Avoid use of `api.v7` decorator in new code, unless there is already an API fragmentation in parent methods.

Symbols

Odoo Python Classes

Use UpperCamelCase for code in api v8, underscore lowercase notation for old api.

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

class account_invoice(orm.Model):
    ...
```

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 maths 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)
- Since new API works with records or recordsets instead of id lists, don't 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 ...
...
```

SQL

No SQL Injection

Care must be taken not to introduce SQL injections vulnerabilities when using manual SQL queries. The vulnerability is present when user input is either incorrectly filtered or badly quoted, allowing an attacker to introduce undesirable clauses to a SQL query (such as circumventing filters or executing **UPDATE** or **DELETE** commands).

The best way to be safe is to never, NEVER use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string.

The second reason, which is almost as important, is that it is the job of the database abstraction layer (psycopg2) to decide how to format query parameters, not your job! For example psycopg2 knows that when you pass a list of values it needs to format them as a comma-separated list, enclosed in parentheses!

```
# the following is very bad:
#   - it's a SQL injection vulnerability
#   - it's unreadable
#   - it's not your job to format the list of ids
cr.execute('select distinct child_id from account_account_consol_rel ' +
           'where parent_id in ('+','.join(map(str, ids))+')')

# better
cr.execute('SELECT DISTINCT child_id '\
           'FROM account_account_consol_rel '\
           'WHERE parent_id IN %s',
           (tuple(ids),))
```

This is very important, so please be careful also when refactoring, and most importantly do not copy these patterns!

Here is a memorable example to help you remember what the issue is about (but do not copy the code there).

Before continuing, please be sure to read the online documentation of pyscopg2 to learn of to use it properly:

- The problem with query parameters
- How to pass parameters with psycopg2
- Advanced parameter types

Never commit the transaction

The Odoo framework is in charge of providing the transactional context for all RPC calls. The principle is that a new database cursor is opened at the beginning of each RPC call, and committed when the call has returned, just before transmitting the answer to the RPC client, approximately like this:

```
def execute(self, db_name, uid, obj, method, *args, **kw):
    db, pool = pooler.get_db_and_pool(db_name)
    # create transaction cursor
    cr = db.cursor()
    try:
        res = pool.execute_cr(cr, uid, obj, method, *args, **kw)
        cr.commit() # all good, we commit
    except Exception:
        cr.rollback() # error, rollback everything atomically
        raise
    finally:
        cr.close() # always close cursor opened manually
    return res
```

If any error occurs during the execution of the RPC call, the transaction is rolled back atomically, preserving the state of the system.

Similarly, the system also provides a dedicated transaction during the execution of tests suites, so it can be rolled back or not depending on the server startup options.

The consequence is that if you manually call cr.commit() anywhere there is a very high chance that you will break the system in various ways, because you will cause partial commits, and thus partial and unclean rollbacks, causing among others:

- inconsistent business data, usually data loss ;
- workflow desynchronization, documents stuck permanently ;
- tests that can't be rolled back cleanly, and will start polluting the database, and triggering error (this is true even if no error occurs during the transaction);

Unless:

- You have created your own database cursor explicitly! And the situations where you need to do that are exceptional! And by the way if you did create your own cursor, then you need to handle error cases and proper rollback, as well as properly close the cursor when you're done with it.

  And contrary to popular belief, you do not even need to call cr.commit() in the following situations:

  - in the _auto_init() method of an models.Model object: this is taken

care of by the addons initialization method, or by the ORM transaction when creating custom models
  - in reports: the commit() is handled by the framework too, so you can update the database even from within a report
  - within models.TransientModel methods: these methods are called exactly like regular models.Model ones, within a transaction and with the corresponding cr.commit()/rollback() at the end ;
  - etc. (see general rule above if you have in doubt!)

- All cr.commit() calls outside of the server framework from now on must have an explicit comment explaining why they are absolutely necessary, why they are indeed correct, and why they do not break the transactions. Otherwise they can and will be removed!

- You can avoid the cr.commit using cr.savepoint method.

```
try:
    with cr.savepoint():
        # Create a savepoint and rollback this section if any exception is raised.
        method1()
        method2()
# Catch here any exceptions if you need to.
except (except_class1, except_class2):
    # Add here the logic if anything fails. NOTE: Don't need rollback sentence.
    pass
```

- You can isolate a transaction for a valid cr.commit using 'Environment':

```
with odoo.api.Environment.manage():
    with odoo.registry(self.env.cr.dbname).cursor() as new_cr:
        # Create a new environment with new cursor database
        new_env = api.Environment(new_cr, self.env.uid, self.env.context)
        # with_env replace original env for this method
        # A good comment here of why this isolated transaction is required.
        self.with_env(new_env).write({'name': 'hello'})  # isolated transaction to comm
    # You don't need to close nor to commit your cursor as they are done when exiting '
# You don't need clear caches because is cleared when finish "with"
```

Do not bypass the ORM

You should never use the database cursor directly when the ORM can do the same thing! By doing so you are bypassing all the ORM features, possibly the transactions, access rights and so on.

And chances are that you are also making the code harder to read and probably less secure (see also previous guideline: No SQL Injection):

```
# very very wrong
cr.execute('select id from auction_lots where auction_id in (' +
```

```
            ','.join(map(str, ids)) + ') and state=%s and obj_price>0',
            ('draft',))
auction_lots_ids = [x[0] for x in cr.fetchall()]

# no injection, but still wrong
cr.execute('select id from auction_lots where auction_id in %s '
            'and state=%s and obj_price>0',
            (tuple(ids), 'draft',))
auction_lots_ids = [x[0] for x in cr.fetchall()]

# better
auction_lots_ids = self.search(cr, uid, [
    ('auction_id', 'in', ids),
    ('state', '=', 'draft'),
    ('obj_price', '>', 0),
])
```

Models

- Model names
  - Use dot lowercase name for models. Example: sale.order
  - Use name 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_<constraint_name>
  - Action method: an object action method is prefix with action_. Its decorator is @api.multi, but since it use only one record, add self.ensure_one() at the beginning of the method.
  - @api.one method: For v8 is recommended use @api.multi and avoid use @api.one, for compatibility with v9 where is deprecated @api.one.
- In a Model attribute order should be
  1. Private attributes (_name, _inherit, _description, ...)
  2. Fields declarations
  3. SQL constraints
  4. Default method and _default_get
  5. Compute and search methods in the same order than field declaration
  6. Constrains methods (@api.constrains) and onchange methods (@api.onchange)
  7. CRUD methods (ORM overrides)
  8. Action methods
  9. And finally, other business methods.

```python
class Event(models.Model):
    # Private attributes: model declaration (_name and inherits), _description, ...
    _name = 'event.event'
    _inherit = ['event.event', 'mail.thread']
    _description = 'Event'
    _order = 'name'

    # 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):
    ...
```

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, ...)
- If the technical name of the field (the variable name) is the same to the string of the label, don't put string parameter for new API fields, because it's automatically taken. If your variable name contains "_" in the name, they are converted to spaces when creating the automatic string and each word is capitalized. (example:

  old api 'name': fields.char('Name', ...) new api 'name': fields.Char(...))
- 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())
  ```

Exceptions

The pass into block except is not a good practice!

By including the pass we assume that our algorithm can continue to function after the exception occurred

If you really need to use the pass consider logging that exception

```
try:
    sentences
except Exception:
    _logger.debug('Why the exception is safe....', exc_info=1))
```

Javascript

- use strict; is recommended for all javascript files
- Use ESLint with this configuration
- Never add minified Javascript libraries
- Use UpperCamelCase for class declarations

CSS

- Prefix all your classes with o_<module_name> where module_name is the technical name of the module (sale, im_chat, ...) or the main route reserved by the module (for website module mainly, i.e. o_forum for website_forum module). The only exception for this rule is the webclient: it simply use o_ prefix.
- Avoid using ids
- Use bootstrap native classes
- Use underscore lowercase notation to name classes

Tests

As a general rule, a bug fix should come with a unittest which would fail without the fix 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 coveralls web interface can also show which lines need test cases.

NOTE: if you add an example module to showcase modules' features you should name it `module_name_example` (ie: cms_form and cms_form_example). In this way coverage analysis will ignore this extra module by default.

Avoid Flaky Tests

Flaky tests are unit tests that produce inconsistent or unreliable results. These tests may pass or fail intermittently, even when the code being tested hasn't changed. Flakiness can be caused by various factors, such as race conditions, external dependencies, or non-deterministic behavior in the code. How to avoid them?

You can find online some general studies and guidelines about the subject:

- https://docs.gitlab.com/ee/development/testing_guide/flaky_tests.html
- https://semaphoreci.com/community/tutorials/how-to-deal-with-and-eliminate-flaky-tests
- https://martinfowler.com/articles/nonDeterminism.html

In the context of Odoo, the following are some tips to avoid flaky tests:

Beware with `subTest` pollution

If you test uses `subTest`, keep in mind that the DB transaction and the environmental cache aren't reset between subtests.

Remember that some decorators are internally implemented using subtests.

A bad example:

```
from odoo.tests import TransactionCase, users

class TestSomething(TransactionCase):
  @users('admin', 'demo')
  def test_something(self):
    """2nd run will fail with duplicate key on login field."""
    self.env["res.users"].create({"name": "Foo", "login": "foo"})
```

Test with the lowest permissions possible

By default, tests are executed without permission restrictions. This can produce false positives because later, real users that will do the same flows might encounter access errors.

Try to test with the lowest permissions possible. Odoo test tools include `new_test_user` and `users` helpers to help you with that.

You can also set the `uid` or `env` attributes of the test case in its `setUpClass` to ensure the rest of the test is ran under a constrained environment.

Beware with unexpected metadata carried by a record's env

When you store a record in a class or instance variable, the record has a `.env` attribute that carries things such as the DB cursor, the user ID, the sudo status and the context. Even if you change the test user, those things will probably remain the same.

A bad example:

```
from odoo.exceptions import AccessError
from odoo.tests import TransactionCase, users, new_test_user

class TestSomething(TransactionCase):
```

```
  @classmethod
  def setUpClass(cls):
    super().setUpClass()
    cls.partner = cls.env["res.partner"].create({"name": "Foo"})
    new_test_user(cls.env, "test_portal", groups="base.group_portal")

  @users("test_portal")
  def test_cannot_read(self):
    # OK
    self.assertEqual(self.uid, self.portal.id)
    # KO; self.partner.env.uid is not self.uid, but you didn't notice
    self.assertRaises(AccessError, self.partner.read)
    # This would be OK, though
    partner = self.partner.with_env(self.env)
    self.assertRaises(AccessError, partner.read)
```

Avoid dynamic dates

If you are testing a feature that depends on a date, you should avoid using `datetime.now()` or `datetime.today()`. Instead, use fixed dates. A way to do this is to use the freezegun library <https://pypi.org/project/freezegun/>___ (which is a standard Odoo test dependency since Odoo 14.0):

```
from datetime import datetime
from freezegun import freeze_time
from odoo.tests import TransactionCase

class TestSomething(TransactionCase):
  @freeze_time("2024-01-01 10:10:10")
  def test_creation_time(self):
    partner = self.env["res.partner"].create({"name": "Foo"})
    self.assertEqual(partner.create_date, datetime(2024, 1, 1, 10, 10, 10))
```

Avoid contacting external services

If your module connects with some kind of external service, you should mock its tests to avoid false negatives when the service is down or slow. You can help yourself with the unittest.mock library. Since Odoo 16.0, it is used behind the scenes when you call `cls.classPatch()`, `cls.startClassPatcher()`, `self.patch()` (also available in older versions) or `self.startPatcher()` in a test case:

```
from odoo.tests import TransactionCase

class TestSomething(TransactionCase):
  @classmethod
  def setUpClass(cls):
```

```
    super().setUpClass()
    cls.classPatch(
      self.env["res.partner"], "patched_method", lambda self: "Some response"
    )

  def test_something(self):
    ...  # Your test here
```

Example with `unittest.mock` directly:

```
from unittest.mock import patch
from odoo.tests import TransactionCase

class TestSomething(TransactionCase):
  @patch("my_module.external_service")
  def test_something(self, mock_external_service):
    mock_external_service.return_value = "Some response"
    # Your test here
```

The vcrpy library is also helpful for recording and replaying HTTP interactions.

Remember that what you want to test is your code, not the external service. There are monitoring tools for that. Also remember that, if there is a bug in the external service, it is not your responsibility to fix it. And if you want to contemplate that possibility, you just have to test the buggy behavior in a deterministic manner, also using mocks.

Still, sometimes you may still want to test the real service, just to make sure they didn't change their API. In this case, do just like upstream Odoo does and mark your test with these standard test tags:

- `-standard` to skip these tests by default.
- `external` to mark them as depending on connection with the outside world, whose state is not under your control.
- `external_l10n`, just like `external` but for localization tests.

You will be able then to isolate those test runs separately:

```
odoo -i my_module -d my_database -u my_module --test-tags=external,external_l10n
```

Avoid relying on demo data

Demo data can be altered manually or by other third party module (through its fully XML-ID identifier). Uncontrolled inputs can produce false positives or negatives in your tests.

If you create the test data within your test suite, you will have more consistent and resilient results.

Investigating Travis Test Failures

It can sometimes be difficult to reproduce a Travis test failure locally due to subtle environment differences. In these scenarios it can be helpful to connect to the Runbot container generated for that branch/PR via SSH, where the environment will be very similar to Travis. You can do this by running:

```
` ssh -p [port] -L 18080:localhost:18069 odoo@runbot[1 or 2].odoo-community.org‘
```

The correct Runbot subdomain can be found by checking the info on https://runbot.odoo-community.org/runbot for your particular repo and branch. The port can also be found there by clicking on the gear icon next to the relevant Runbot instance and adding 1 to the port number in the dropdown.

In order to be authenticated, your public SSH key will need to be associated with your GitHub account **before** the Runbot instance is generated. You must also be the author of the commit that triggered the Runbot build.

Once you've connected to the container, you can run tests as follows:

```
cp -r ~/data_dir/filestore/odoo_template ~/data_dir/filestore/[github_username]
createdb -T odoo_template [github_username]
[~/odoo-9.0/odoo.py or ~/odoo-10.0/odoo-bin] -d [github_username] --db-filter=[github_usern
```

The test instance can be accessed through your browser at http://localhost:18080/ thanks to SSH port forwarding. To rebuild the DB as needed, run:

```
dropdb [github_username]
createdb -T odoo_template [github_username]
```

WARNING: Do not stop the default Odoo service running in the container as this will bring down the entire Runbot instance.

Git

Commit message

Write a **short** commit summary without prefixing it. It should not be longer than 50 characters: This is a commit message

Then, in the message itself, specify the part of the code impacted by your changes (module name, lib, transversal object, ...) and a description of the changes. This part should be multiple lines no longer than 80 characters.

- Commit messages are in English
- Merge proposals should follow the same rules as the title of the propsal is the first line of the merge commit and the description corresponds to commit description.
- Always put meaningful commit messages: commit messages should be self explanatory (long enough) including the name of the module that has been

changed and the reason behind that change. Do not use single words like "bugfix" or "improvements".
- Avoid commits which simultaneously impact lots of modules. Try to split into different commits where impacted modules are different. This is helpful if we need to revert changes on a module separately.
- Only make a single commit per logical change set. Do not add commits such as "Fix pep8", "Code review" or "Add unittest" if they fix commits which are being proposed
- Use present imperative (Fix formatting, Remove unused field) avoid appending 's' to verbs: Fixes, Removes
- Use tags as listed in the Odoo Guidelines with the following extensions:
  - [**MIG**] for migrating a module

```
[FIX] website: remove unused alert div

Fix look of input-group-btn
Bootstrap's CSS depends on the input-group-btn element being the first/last
child of its parent.
This was not the case because of the invisible and useless alert.
```

```
[IMP] web: add module system to the web client

This commit introduces a new module system for the javascript code.
Instead of using global ...
```

When you open a PR, please check if the commit message is cut with ellipsis. For example:

```
[FIX] module_foo: and this is my very long m[...]
```

When this happens, it means your message is too long. You should shorten it. Start by rephrasing and keeping the summary very synthetic. The explanation or motivation should be kept in the description of the commit.

Review

Peer review is the only way to ensure good quality of the code and to be able to rely on the other developers. The peer review in this project will be managed through Pull Requests. It will serve the following main purposes:

- Having a second look on a code snippet to avoid unintended problems / bugs
- Avoid technical or business design flaws
- Allow the coordination and convergence of the developers by informing the community of what has been done
- Allow the responsibles to look at every devs and keep the interested people informed of what has been done
- Prevent addon incompatibilities when/if possible

- The rationale for peer review has its equivalent in Linus's law, often phrased: "Given enough eyeballs, all bugs are shallow"

Meaning "If there are enough reviewers, all problems are easy to solve". Eric S. Raymond has written influentially about peer review in software development: http://en.wikipedia.org/wiki/Software_peer_review.

Please respect a few basic rules:

- Read and follow the rules stated for the module maturity levels.
- At least one of the review above must be from a member of the PSC or having write access on the repository (here one of the OCA Core Maintainers. can do the job. You can notify them on Github using `@OCA/core-maintainers`)
- If you are in a hurry just send a mail at contributors@odoo-community.org or ask by chat in either of:
  - OCA discord server
  - Matrix space #oca:matrix.org (bridged to discord).
- Is the module generic enough to be part of community addons?
- Is the module duplicating features with other community addons?
- Does the documentation allow to understand what it does and how to use it?
- Is the problem it tries to resolve adressed the good way, using good concepts?
- Are there some use cases?
- Is there any setup in code? Should not!
- Are there demo data?
- Are the commit messages short, clear and clean?

Further reading:

- https://insidecoding.wordpress.com/2013/01/07/code-review-guidelines/

There are the following important parts in a review:

- Start by thanking the contributor / developer for their work. No matter the issue of the PR, someone has done work for you, so be thankful for that.
- Be cordial and polite. Nothing is obvious in a PR.
- The description of the changes should be clear enough for you to understand their purpose and, if applicable, contain a demo in order to allow people to run and test the code
- Choose the review tag (comment, approve, rejected, needs information,...) and don't forget to add a type of review to let people know:
  - Code review: means you look at the code
  - Test: means you tested it functionally speaking

While making the merge, please respect the author using the --author option

when committing. The author is found using the git log command. Use the commit message provided by the contributor if any.

It makes sense to be picky in the following cases:

- The origin/reason for the patch/dev is not documented very well
- No adapted / convenient description written in the ___manifest___.py file for the module
- Tests or scenario are not all green and/or not adapted
- Having tests is very much encouraged
- Issues with license, copyright, authorship
- Respect of Odoo/community conventions
- Code design and best practices

The long description try to explain the **why** not the **what**, the **what** can be seen in the diff.

Pull requests can be closed if:

- there is no activity for 6 months

Github

Teams

- Team name must not contain odoo or openerp.
- Team name for localization is "Belgium Maintainers" for Belgium.

Repositories

Naming

- Project name must not contain odoo or openerp.
- Project name for localization is l10n-belgium for Belgium.
- Project name for connectors is connector-magento for Magento connector.

Branch configuration

Python packages to install must be preferably defined in requirements.txt than travis.yml file.

Requirements.txt avoid to repeat packages in all travis.yml files of repositories in case of using with oca_dependencies.txt file.

Issues

- Issues are used for blueprints and bugs.

Differences With Odoo Guidelines

Not the entire Odoo guidelines fit OCA modules needs. In many cases rules need to be more stringent. In other cases, conventions are improved for better maintainability in an ecosystem of many smaller modules.

The differences include:

- Module Structure
  - Using one file per model
  - Separating data and demo data xml folders
  - Not changing xml_ids while inheriting
  - Add guideline to use external dependencies
  - Define a separated file for installation hooks
- XML
  - Avoid use current module in xml_id
  - Use explicit user_id field for records of model ir.filters
- Python
  - Use Python standards
  - Fuller PEP8 compliance
  - Using relative import for local files
  - More python idioms
  - A way to deal with long comma-separated lines
  - Hints on documentation
  - Don't use CamelCase for model variables
  - Use underscore uppercase notation for global variables or constants
- SQL
  - Add section for No SQL Injection
  - Add section for don't bypass the ORM
  - Add section for never commit the transaction
- Field
  - A hint for function defaults
  - Use default label string if is possible
  - Add the inverse method pattern
- Tests Section Added
- Git
  - No prefixing of commits
  - Default git commit message standards
  - Squashing changes in pull requests when necessary
  - Use of present imperative
- Github Section
- Review Section

Backporting Odoo Modules

Suggesting a backport of a module among an OCA repository is possible, but you must respect a few rules:

- You need to keep the license of the module coded by Odoo SA
- You need to add the OCA as author (and Odoo SA of course)
- You need to make the module "OCA compatible": PEP8, OCA convention and so on so it won't break our CI like runbot, Travis and so.
- You need to add a disclaimer in the README.rst file with the following text:

  > This module is a backport from Odoo SA and as such, it is not included in the OCA CLA. That means we do not have a copy of the copyright on it like all other OCA modules.

Translations

OCA uses Weblate for translation, at https://translation.odoo-community.org/

Pull requests should never directly modify `.po` files, because this could produce merge conflicts with pushes coming from weblate. Modifying `.po` files is the responsibility of weblate only.

Pull requests adding new modules can add their own `.po` files.