Odoo Guidelines

This page introduces the new Odoo Coding Guidelines. Those aim to improve the quality of the code (e.g. better readability of source) and Odoo Apps. Indeed, proper code eases maintenance, aids debugging, lowers complexity and promotes reliability.

These guidelines should be applied to every new module, and new developpment. These guidelines will be applied to old module **only** in case of code refactoring (migration to new API, big refactoring, ...).

Warning

These guidelines are written with new modules and new files in mind. When modifying existing files, the original style of the file strictly supersedes any other style guidelines. In other words, never modify existing files in order to apply these guidelines, to avoid disrupting the revision history of each line. For more details, see our [pull request guide](https://odoo.com/submit-pr).

**Module structure**

**Directories**

A module is organised in important directories. Those contain the business logic; having a look at them should make understand the purpose of the module.

* *data/* : demo and data xml
* *models/* : models definition
* *controllers/* : contains controllers (HTTP routes).
* *views/* : contains the views and templates
* *static/* : contains the web assets, separated into *css/, js/, img/, lib/, ...*

Other optional directories compose the module.

* *wizard/* : regroups the transient models (formerly *osv\_memory*) and their views.
* *report/* : contains the reports (RML report **[deprecated]**, models based on SQL views (for reporting) and other complex reports). Python objects and XML views are included in this directory.
* *tests/* : contains the Python/YML tests

**File naming**

For *views* declarations, split backend views from (frontend) templates in 2 differents files.

For *models*, split the business logic by sets of models, in each set select a main model, this model gives its name to the set. If there is only one model, its name is the same as the module name. For each set named <main\_model> the following files may be created:

* models/*<main\_model>*.py
* models/*<inherited\_main\_model>*.py
* views/*<main\_model>*\_templates.xml
* views/*<main\_model>*\_views.xml

For instance, *sale* module introduces sale\_order and sale\_order\_line where sale\_order is dominant. So the <main\_model> files will be named models/sale\_order.py and views/sale\_order\_views.py.

For *data*, split them by purpose : demo or data. The filename will be the main\_model name, suffixed by *\_demo.xml* or *\_data.xml*.

For *controllers*, the only file should be named *main.py*. Otherwise, if you need to inherit an existing controller from another module, its name will be *<module\_name>.py*. Unlike *models*, each controller class should be contained in a separated file.

For *static files*, since the resources can be used in different contexts (frontend, backend, both), they will be included in only one bundle. So, CSS/Less, JavaScript and XML files should be suffixed with the name of the bundle type. i.e.: *im\_chat\_common.css*, *im\_chat\_common.js* for 'assets\_common' bundle, and *im\_chat\_backend.css*, *im\_chat\_backend.js* for 'assets\_backend' bundle. If the module owns only one file, the convention will be *<module\_name>.ext* (i.e.: *project.js*). Don't link data (image, libraries) outside Odoo: do not use an URL to an image but copy it in our codebase instead.

Regarding *data*, split them by purpose: data or demo. The filename will be the *main\_model* name, suffixed by *\_data.xml* or *\_demo.xml*.

Regarding *wizards*, naming convention is :

* *<main\_transient>*.py
* *<main\_transient>*\_views.xml

Where *<main\_transient>* is the name of the dominant transient model, just like for *models*. <main\_transient>.py can contains the models 'model.action' and 'model.action.line'.

For *statistics reports*, their names should look like :

* *<report\_name\_A>*\_report.py
* *<report\_name\_A>*\_report\_views.py (often pivot and graph views)

For *printable reports*, you should have :

* *<print\_report\_name>*\_reports.py (report actions, paperformat definition, ...)
* *<print\_report\_name>*\_templates.xml (xml report templates)

The complete tree should look like

addons/<my\_module\_name>/

|-- \_\_init\_\_.py

|-- \_\_manifest\_\_.py

|-- controllers/

| |-- \_\_init\_\_.py

| |-- <inherited\_module\_name>.py

| `-- main.py

|-- data/

| |-- <main\_model>\_data.xml

| `-- <inherited\_main\_model>\_demo.xml

|-- models/

| |-- \_\_init\_\_.py

| |-- <main\_model>.py

| `-- <inherited\_main\_model>.py

|-- report/

| |-- \_\_init\_\_.py

| |-- <main\_stat\_report\_model>.py

| |-- <main\_stat\_report\_model>\_views.xml

| |-- <main\_print\_report>\_reports.xml

| `-- <main\_print\_report>\_templates.xml

|-- 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

|-- views/

| |-- <main\_model>\_templates.xml

| |-- <main\_model>\_views.xml

| |-- <inherited\_main\_model>\_templates.xml

| `-- <inherited\_main\_model>\_views.xml

`-- wizard/

|-- <main\_transient\_A>.py

|-- <main\_transient\_A>\_views.xml

|-- <main\_transient\_B>.py

`-- <main\_transient\_B>\_views.xml

Note

File names should only contain [a-z0-9\_] (lowercase alphanumerics and \_)

Warning

Use correct file permissions : folder 755 and file 644.

**XML files**

**Format**

To declare a record in XML, the **record** notation (using *<record>*) is recommended:

* Place id attribute before model
* For field declaration, 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 record by model. In case of dependencies between action/menu/views, this 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

<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" />

</tree>

</field>

</record>

Odoo supports custom tags acting as syntactic sugar:

* menuitem: use it as a shortcut to declare a ir.ui.menu
* workflow: the <workflow> tag sends a signal to an existing workflow.
* template: use it to declare a QWeb View requiring only the arch section of the view.
* report: use to declare a [report action](https://www.odoo.com/documentation/10.0/reference/actions.html#reference-actions-report)
* act\_window: use it if the record notation can't do what you want

The 4 first tags are prefered over the *record* notation.

**Naming xml\_id**

**Security, View and Action**

Use the following pattern :

* 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 a lowercase string briefly explaining the action. 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, ...).
* For a group : *<model\_name>*\_group\_*<group\_name>* where *group\_name* is the name of the group, generally 'user', 'manager', ...

<!-- views and menus -->

<record id="model\_name\_view\_form" model="ir.ui.view">

...

</record>

<record id="model\_name\_view\_kanban" model="ir.ui.view">

...

</record>

<menuitem

id="model\_name\_menu\_root"

name="Main Menu"

sequence="5"

/>

<menuitem

id="model\_name\_menu\_action"

name="Sub Menu 1"

parent="module\_name.module\_name\_menu\_root"

action="model\_name\_action"

sequence="10"

/>

<!-- 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="module\_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>

Note

View names use dot notation my.model.view\_type or my.model.view\_type.inherit instead of *"This is the form view of My Model"*.

**Inherited XML**

The naming pattern of inherited view is *<base\_view>*\_inherit\_*<current\_module\_name>*. A module may only extend a view once. Suffix the orginal name with \_inherit\_*<current\_module\_name>* where *current\_module\_name* is the technical name of the module extending the view.

<record id="inherited\_model\_view\_form\_inherit\_my\_module" model="ir.ui.view">

...

</record>

**Python**

**PEP8 options**

Using a linter can help show syntax and semantic warnings or errors. Odoo source code tries to respect Python standard, but some of them can be ignored.

* E501: line too long
* E301: expected 1 blank line, found 0
* E302: expected 2 blank lines, found 1
* E126: continuation line over-indented for hanging indent
* E123: closing bracket does not match indentation of opening bracket's line
* E127: continuation line over-indented for visual indent
* E128: continuation line under-indented for visual indent
* E265: block comment should start with '# '

**Imports**

The imports are ordered as

1. External libraries (one per line sorted and split in python stdlib)
2. Imports of odoo
3. Imports from Odoo modules (rarely, and only if necessary)

Inside these 3 groups, the imported lines are alphabetically sorted.

# 1 : imports of python lib

import base64

import re

import time

from datetime import datetime

# 2 : imports of odoo

import odoo

from odoo import api, fields, models # alphabetically ordered

from odoo.tools.safe\_eval import safe\_eval as eval

from odoo.tools.translate import \_

# 3 : imports from odoo modules

from odoo.addons.website.models.website import slug

from odoo.addons.web.controllers.main import login\_redirect

**Idiomatics Python Programming**

* Each python file should have # -\*- coding: utf-8 -\*- as first line.
* Always favor *readability* over *conciseness* or using the language features or idioms.
* Don't use .clone()

# bad

new\_dict = my\_dict.clone()

new\_list = old\_list.clone()

# good

new\_dict = dict(my\_dict)

new\_list = list(old\_list)

* Python dictionary : creation and update

# -- creation empty dict

my\_dict = {}

my\_dict2 = dict()

# -- creation with values

# bad

my\_dict = {}

my\_dict['foo'] = 3

my\_dict['bar'] = 4

# good

my\_dict = {'foo': 3, 'bar': 4}

# -- update dict

# bad

my\_dict['foo'] = 3

my\_dict['bar'] = 4

my\_dict['baz'] = 5

# good

my\_dict.update(foo=3, bar=4, baz=5)

my\_dict = dict(my\_dict, \*\*my\_dict2)

* Use meaningful variable/class/method names
* Useless variable : Temporary variables can make the code clearer by giving names to objects, but that doesn't mean you should create temporary variables all the time:

# pointless

schema = kw['schema']

params = {'schema': schema}

# simpler

params = {'schema': kw['schema']}

* Multiple return points are OK, when they're simpler

# a bit complex and with a redundant temp variable

def axes(self, axis):

axes = []

if type(axis) == type([]):

axes.extend(axis)

else:

axes.append(axis)

return axes

# clearer

def axes(self, axis):

if type(axis) == type([]):

return list(axis) # clone the axis

else:

return [axis] # single-element list

* Know your builtins : You should at least have a basic understanding of all the Python builtins (<http://docs.python.org/library/functions.html>)

value = my\_dict.get('key', None) # very very redundant

value= my\_dict.get('key') # good

Also, if 'key' in my\_dict and if my\_dict.get('key') have very different meaning, be sure that you're using the right one.

* Learn list comprehensions : Use list comprehension, dict comprehension, and basic manipulation using map, filter, sum, ... They make the code easier to read.

# not very good

cube = []

for i in res:

cube.append((i['id'],i['name']))

# better

cube = [(i['id'], i['name']) for i in res]

* Collections are booleans too : In python, many objects have "boolean-ish" value when evaluated in a boolean context (such as an if). Among these are collections (lists, dicts, sets, ...) which are "falsy" when empty and "truthy" when containing items:

bool([]) is False

bool([1]) is True

bool([False]) is True

So, you can write if some\_collection: instead of if len(some\_collection):.

* Iterate on iterables

# creates a temporary list and looks bar

for key in my\_dict.keys():

"do something..."

# better

for key in my\_dict:

"do something..."

# creates a temporary list

for key, value in my\_dict.items():

"do something..."

# only iterates

for key, value in my\_dict.iteritems():

"do something..."

* Use dict.setdefault

# longer.. harder to read

values = {}

for element in iterable:

if element not in values:

values[element] = []

values[element].append(other\_value)

# better.. use dict.setdefault method

values = {}

for element in iterable:

values.setdefault(element, []).append(other\_value)

* As a good developper, document your code (docstring on methods, simple comments for tricky part of code)
* In additions to these guidelines, you may also find the following link interesting: <http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html> (a little bit outdated, but quite relevant)

**Programming in Odoo**

* Avoid to create generators and decorators: only use the ones provided by the Odoo API.
* As in python, use filtered, mapped, sorted, ... methods to ease code reading and performance.

**Make your method works in batch**

When adding a function, make sure it can process multiple records. Typically, such method is decorated with api.multi decorator (or takes a list of *id*, if written in old api). Then you will have to iterate on self to treat each record.

@api.multi

def my\_method(self)

for record in self:

record.do\_cool\_stuff()

Avoid to use api.one decorator : this will probably not do what you expected, and extending a such method is not as easy than a *api.multi* method, since it returns a list of result (ordered by recordset ids).

For performance issue, when developping a 'stat button' (for instance), do not perform a search or a search\_count in a loop in a api.multi method. It is recommended to use read\_group method, to compute all value in only one request.

@api.multi

def \_compute\_equipment\_count(self):

""" Count the number of equipement per category """

equipment\_data = self.env['hr.equipment'].read\_group([('category\_id', 'in', self.ids)], ['category\_id'], ['category\_id'])

mapped\_data = dict([(m['category\_id'][0], m['category\_id\_count']) for m in equipment\_data])

for category in self:

category.equipment\_count = mapped\_data.get(category.id, 0)

**Propagate the context**

In new API, the context is a frozendict that cannot be modified. To call a method with a different context, the with\_context method should be used :

records.with\_context(new\_context).do\_stuff() # all the context is replaced

records.with\_context(\*\*additionnal\_context).do\_other\_stuff() # additionnal\_context values override native context ones

Passing parameter in context can have dangerous side-effects. Since the values are propagated automatically, some behavior can appears. Calling create() method of a model with *default\_my\_field* key in context will set the default value of *my\_field* for the concerned model. But if curing this creation, other object (such as sale.order.line, on sale.order creation) having a field name *my\_field*, their default value will be set too.

If you need to create a key context influencing the behavior of some object, choice a good name, and eventually prefix it by the name of the module to isolate its impact. A good example are the keys of mail module : *mail\_create\_nosubscribe*, *mail\_notrack*, *mail\_notify\_user\_signature*, ...

**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.

# very very wrong

self.env.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 self.env.cr.fetchall()]

# no injection, but still wrong

self.env.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 self.env.cr.fetchall()]

# better

auction\_lots\_ids = self.search([('auction\_id','in',ids), ('state','=','draft'), ('obj\_price','>',0)])

**No SQL injections, please !**

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

self.env.cr.execute('SELECT distinct child\_id FROM account\_account\_consol\_rel ' +

'WHERE parent\_id IN ('+','.join(map(str, ids))+')')

# better

self.env.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 (<http://initd.org/psycopg/docs/usage.html#the-problem-with-the-query-parameters>)
* How to pass parameters with psycopg2 (<http://initd.org/psycopg/docs/usage.html#passing-parameters-to-sql-queries>)
* Advanced parameter types (<http://initd.org/psycopg/docs/usage.html#adaptation-of-python-values-to-sql-types>)

**Keep your methods short/simple when possible**

Functions and methods should not contain too much logic: having a lot of small and simple methods is more advisable than having few large and complex methods. A good rule of thumb is to split a method as soon as: - it has more than one responsibility (see <http://en.wikipedia.org/wiki/Single_responsibility_principle>) - it is too big to fit on one screen.

Also, name your functions accordingly: small and properly named functions are the starting point of readable/maintainable code and tighter documentation.

This recommendation is also relevant for classes, files, modules and packages. (See also <http://en.wikipedia.org/wiki/Cyclomatic_complexity>)

**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:

1. inconsistent business data, usually data loss
2. workflow desynchronization, documents stuck permanently
3. 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)

Here is the very simple rule:

You should **NEVER** call cr.commit() yourself, **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.Transient* 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 !

**Use translation method correctly**

Odoo uses a GetText-like method named "underscore" \_( ) to indicate that a static string used in the code needs to be translated at runtime using the language of the context. This pseudo-method is accessed within your code by importing as follows:

from odoo.tools.translate import \_

A few very important rules must be followed when using it, in order for it to work and to avoid filling the translations with useless junk.

Basically, this method should only be used for static strings written manually in the code, it will not work to translate field values, such as Product names, etc. This must be done instead using the translate flag on the corresponding field.

The rule is very simple: calls to the underscore method should always be in the form \_('literal string') and nothing else:

# good: plain strings

error = \_('This record is locked!')

# good: strings with formatting patterns included

error = \_('Record %s cannot be modified!') % record

# ok too: multi-line literal strings

error = \_("""This is a bad multiline example

about record %s!""") % record

error = \_('Record %s cannot be modified' \

'after being validated!') % record

# bad: tries to translate after string formatting

# (pay attention to brackets!)

# This does NOT work and messes up the translations!

error = \_('Record %s cannot be modified!' % record)

# bad: dynamic string, string concatenation, etc are forbidden!

# This does NOT work and messes up the translations!

error = \_("'" + que\_rec['question'] + "' \n")

# bad: field values are automatically translated by the framework

# This is useless and will not work the way you think:

error = \_("Product %s is out of stock!") % \_(product.name)

# and the following will of course not work as already explained:

error = \_("Product %s is out of stock!" % product.name)

# bad: field values are automatically translated by the framework

# This is useless and will not work the way you think:

error = \_("Product %s is not available!") % \_(product.name)

# and the following will of course not work as already explained:

error = \_("Product %s is not available!" % product.name)

# Instead you can do the following and everything will be translated,

# including the product name if its field definition has the

# translate flag properly set:

error = \_("Product %s is not available!") % product.name

Also, keep in mind that translators will have to work with the literal values that are passed to the underscore function, so please try to make them easy to understand and keep spurious characters and formatting to a minimum. Translators must be aware that formatting patterns such as %s or %d, newlines, etc. need to be preserved, but it's important to use these in a sensible and obvious manner:

# Bad: makes the translations hard to work with

error = "'" + question + \_("' \nPlease enter an integer value ")

# Better (pay attention to position of the brackets too!)

error = \_("Answer to question %s is not valid.\n" \

"Please enter an integer value.") % question

In general in Odoo, when manipulating strings, prefer % over .format() (when only one variable to replace in a string), and prefer %(varname) instead of position (when multiple variables have to be replaced). This makes the translation easier for the community translators.

**Symbols and Conventions**

* Model name (using the dot notation, prefix by the module name) :
  + When defining an Odoo Model : use singular form of the name (*res.partner* and *sale.order* instead of *res.partnerS* and *saleS.orderS*)
  + When defining an Odoo Transient (wizard) : use <related\_base\_model>.<action> where *related\_base\_model* is the base model (defined in *models/*) related to the transient, and *action* is the short name of what the transient do. For instance : account.invoice.make, project.task.delegate.batch, ...
  + When defining *report* model (SQL views e.i.) : use <related\_base\_model>.report.<action>, based on the Transient convention.
* Odoo Python Class : use camelcase for code in api v8 (Object-oriented style), underscore lowercase notation for old api (SQL style).

class AccountInvoice(models.Model):

...

class account\_invoice(osv.osv):

...

* Variable name :
  + use camelcase for model variable
  + use underscore lowercase notation for common variable.
  + since new API works with record or recordset instead of id list, don't suffix variable name with *\_id* or *\_ids* if they not contain id or list of id.

ResPartner = self.env['res.partner']

partners = ResPartner.browse(ids)

partner\_id = partners[0].id

* 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, ...)
* Method conventions
  + Compute Field : the compute method pattern is *\_compute\_<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.
* In a Model attribute order should be
  + Private attributes (\_name, \_description, \_inherit, ...)
  + Default method and \_default\_get
  + Field declarations
  + Compute and search methods in the same order as field declaration
  + Constrains methods (@api.constrains) and onchange methods (@api.onchange)
  + CRUD methods (ORM overrides)
  + Action methods
  + And finally, other business methods.

class Event(models.Model):

# Private attributes

\_name = 'event.event'

\_description = 'Event'

# Default methods

def \_default\_name(self):

...

# Fields declaration

name = fields.Char(string='Name', default=\_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')

# compute and search fields, in the same order of fields declaration

@api.multi

@api.depends('seats\_max', 'registration\_ids.state', 'registration\_ids.nb\_register')

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 (and name\_get, name\_search, ...) overrides

def create(self, values):

...

# Action methods

@api.multi

def action\_validate(self):

self.ensure\_one()

...

# Business methods

def mail\_user\_confirm(self):

...

**Javascript and CSS**

**For javascript :**

* use strict; is recommended for all javascript files
* Use a linter (jshint, ...)
* Never add minified Javascript Libraries
* Use camelcase for class declaration
* Unless your code is supposed to run on every page, target specific pages using the if\_dom\_contains function of website module. Target an element which is specific to the pages your code needs to run on using JQuery.

odoo.website.if\_dom\_contains('.jquery\_class\_selector', function () {

/\*your code here\*/

});

**For 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 uses *o\_* prefix.
* Avoid using id
* Use Bootstrap native classes
* Use underscore lowercase notation to name class

**Git**

**Commit message**

Prefix your commit with

* **[IMP]** for improvements
* **[FIX]** for bug fixes
* **[REF]** for refactoring
* **[ADD]** for adding new resources
* **[REM]** for removing of resources
* **[MOV]** for moving files (Do not change content of moved file, otherwise Git will loose track, and the history will be lost !), or simply moving code from a file to another one.
* **[MERGE]** for merge commits (only for forward/back-port)
* **[CLA]** for signing the Odoo Individual Contributor License

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.

* Always include a meaningful commit message: it should be self explanatory (long enough) including the name of the module that has been changed and the reason behind the change. Do not use single words like "bugfix" or "improvements".
* Avoid commits which simultaneously impact multiple modules. Try to split into different commits where impacted modules are different (It will be helpful if we need to revert a module separately).

[FIX] website, website\_mail: remove unused alert div, fixes 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] fields: reduce memory footprint of list/set field attributes

[REF] web: add module system to the web client

This commit introduces a new module system for the javascript code.

Instead of using global ...

Note

Use the long description to explain the *why* not the *what*, the *what* can be seen in the diff