Mixins and Useful Classes

Odoo implements some useful classes and mixins that make it easy for you to add often-used behaviours on your objects. This guide will details most of them, with examples and use cases.

Messaging features

Messaging integration

Basic messaging system

Integrating messaging features to your model is extremely easy. Simply inheriting the `mail.thread` model and adding the messaging fields (and their appropriate widgets) to your form view will get you up and running in no time.

> **ⓘ Example**
> Let's create a simplistic model representing a business trip. Since organizing this kind of trip usually involves a lot of people and a lot of discussion, let's add support for message exchange on the model.
>
> ```
> class BusinessTrip(models.Model):
>     _name = 'business.trip'
>     _inherit = ['mail.thread']
>     _description = 'Business Trip'
>
>     name = fields.Char()
>     partner_id = fields.Many2one('res.partner', 'Responsible')
>     guest_ids = fields.Many2many('res.partner', 'Participants')
> ```
>
> In the form view:
>
> ```
> <record id="businness_trip_form" model="ir.ui.view">
>     <field name="name">business.trip.form</field>
>     <field name="model">business.trip</field>
>     <field name="arch" type="xml">
>         <form string="Business Trip">
>             <!-- Your usual form view goes here
>             ...
>             Then comes chatter integration -->
>             
>                 <field name="message_follower_ids" widget="mail_followers"/>
>                 <field name="message_ids" widget="mail_thread"/>
>             
>         </form>
>     </field>
> </record>
> ```

Once you've added chatter support on your model, users can easily add messages or internal notes on any record of your model; every one of those will send a notification (to all followers for messages, to employee (*base.group_user*) users for internal notes). If your mail gateway and catchall address are correctly configured, these notifications will be sent by e-mail and can be replied-to directly from your mail client; the automatic routing system will route the answer to the correct thread.

Server-side, some helper functions are there to help you easily send messages and to manage followers on your record:

Posting messages

message_post(*self*, *body=''*, *subject=None*, *message_type='notification'*, *subtype=None*, *parent_id=False*, *attachments=None*, ***kwargs*)

Post a new message in an existing thread, returning the new mail.message ID.

Parameters:

body (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – body of the message, usually raw HTML that will be sanitized

message_type (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – see mail_message.message_type field

parent_id (**int** (https://docs.python.org/3/library/functions.html#int)) – handle reply to a previous message by adding the parent partners to the message in case of private discussion

attachments (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **tuple** (https://docs.python.org/3/library/stdtypes.html#tuple) **(** **str** (https://docs.python.org/3/library/stdtypes.html#str) **,** **str** (https://docs.python.org/3/library/stdtypes.html#str) **)** **)**) – list of attachment tuples in the form **(name,content)** , where content is NOT base64 encoded

****kwargs** – extra keyword arguments will be used as default column values for the new mail.message record

Returns:

ID of newly created mail.message

Return type:

int (https://docs.python.org/3/library/functions.html#int)

message_post_with_view(views_or_xmlid, **kwargs):

Helper method to send a mail / post a message using a view_id to render using the ir.qweb engine. This method is stand alone, because there is nothing in template and composer that allows to handle views in batch. This method will probably disappear when templates handle ir ui views.

Parameters:

or ir.ui.view record (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – external id or record of the view that should be sent

message_post_with_template(*template_id*, ***kwargs*)

Helper method to send a mail with a template

Parameters:

template_id – the id of the template to render to create the body of the message

***kwargs** – parameter to create a mail.compose.message wizzard (which inherit from mail.message)

Receiving messages

These methods are called when a new e-mail is processed by the mail gateway. These e-mails can either be new thread (if they arrive via an alias) or simply replies from an existing thread. Overriding them allows you to set values on the thread's record depending on some values from the email itself (i.e. update a date or an e-mail address, add CC's addresses as followers, etc.).

message_new(*msg_dict*, *custom_values=None*)

Called by **message_process** when a new message is received for a given thread model, if the message did not belong to an existing thread.

The default behavior is to create a new record of the corresponding model (based on some very basic info extracted from the message). Additional behavior may be implemented by overriding this method.

Parameters:

msg_dict (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – a map containing the email details and attachments. See **message_process** and **mail.message.parse** for details

custom_values (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – optional dictionary of additional field values to pass to create() when creating the new thread record; be careful, these values may override any other values coming from the message

Return type:

int (https://docs.python.org/3/library/functions.html#int)

Returns:

the id of the newly created thread object

message_update(*msg_dict*, *update_vals=None*)

Called by **message_process** when a new message is received for an existing thread. The default behavior is to update the record with **update_vals** taken from the incoming email.

Additional behavior may be implemented by overriding this method.

Parameters:

msg_dict (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – a map containing the email details and attachments; see **message_process** and **mail.message.parse()** for details.

update_vals (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – a dict containing values to update records given their ids; if the dict is None or is void, no write operation is performed.

Returns:

True

Followers management

message_subscribe(*partner_ids=None, channel_ids=None, subtype_ids=None, force=True*)

Add partners to the records followers.

Parameters:

partner_ids (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)**) – IDs of the partners that will be subscribed to the record

channel_ids (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)**) – IDs of the channels that will be subscribed to the record

subtype_ids (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)**) – IDs of the subtypes that the channels/partners will be subscribed to (defaults to the default subtypes if **None**)

force – if True, delete existing followers before creating new one using the subtypes given in the parameters

Returns:

Success/Failure

Return type:

bool (https://docs.python.org/3/library/functions.html#bool)

message_unsubscribe(*partner_ids=None, channel_ids=None*)

Remove partners from the record's followers.

Parameters:

partner_ids (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)**) – IDs of the partners that will be subscribed to the record

channel_ids (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)**) – IDs of the channels that will be subscribed to the record

Returns:

True

Return type:

bool (https://docs.python.org/3/library/functions.html#bool)

message_unsubscribe_users(*user_ids=None*)

Wrapper on message_subscribe, using users.

Parameters:

user_ids (**list** (https://docs.python.org/3/library/stdtypes.html#list) **(** **int** (https://docs.python.org/3/library/functions.html#int) **)**) – IDs of the users that will be unsubscribed to the record; if None, unsubscribe the current user instead.

Returns:

True

Return type:

bool (https://docs.python.org/3/library/functions.html#bool)

Logging changes

The **mail** module adds a powerful tracking system on fields, allowing you to log changes to specific fields in the record's chatter. To add tracking to a field, simple set the tracking attribute to True.

> **ⓘ Example**
> Let's track changes on the name and responsible of our business trips:
>
> ```
> class BusinessTrip(models.Model):
>     _name = 'business.trip'
>     _inherit = ['mail.thread']
>     _description = 'Business Trip'
>
>     name = fields.Char(tracking=True)
>     partner_id = fields.Many2one('res.partner', 'Responsible',
>                                  tracking=True)
>     guest_ids = fields.Many2many('res.partner', 'Participants')
> ```
>
> From now on, every change to a trip's name or responsible will log a note on the record. The **name** field will be displayed in the notification as well to give more context about the notification (even if the name did not change).

Subtypes

Subtypes give you more granular control over messages. Subtypes act as a classification system for notifications, allowing subscribers to a document to customize the subtype of notifications they wish to receive.

Subtypes are created as data in your module; the model has the following fields:

name (mandatory) - Char (orm.html#odoo.fields.Char)
name of the subtype, will be displayed in the notification customization popup

description - Char (orm.html#odoo.fields.Char)
description that will be added in the message posted for this subtype. If void, the name will be added instead

internal - Boolean (orm.html#odoo.fields.Boolean)
messages with internal subtypes will be visible only by employees, aka members of the `base.group_user` group

parent_id - Many2one (orm.html#odoo.fields.Many2one)
link subtypes for automatic subscription; for example project subtypes are linked to task subtypes through this link. When someone is subscribed to a project, he will be subscribed to all tasks of this project with subtypes found using the parent subtype

relation_field - Char (orm.html#odoo.fields.Char)
as an example, when linking project and tasks subtypes, the relation field is the project_id field of tasks

res_model - Char (orm.html#odoo.fields.Char)
model the subtype applies to; if False, this subtype applies to all models

default - Boolean (orm.html#odoo.fields.Boolean)
wether the subtype is activated by default when subscribing

sequence - Integer (orm.html#odoo.fields.Integer)
used to order subtypes in the notification customization popup

hidden - Boolean (orm.html#odoo.fields.Boolean)
wether the subtype is hidden in the notification customization popup

Interfacing subtypes with field tracking allows to subscribe to different kind of notifications depending on what might interest users. To do this, you can override the `_track_subtype()` function:

_track_subtype(*init_values*)
Give the subtype triggered by the changes on the record according to values that have been updated.

Parameters:
init_values (**dict** (https://docs.python.org/3/library/stdtypes.html#dict)) – the original values of the record; only modified fields are present in the dict

Returns:
a subtype's full external id or False if no subtype is triggered

> **ⓘ Example**
> Let's add a `state` field on our example class and trigger a notification with a specific subtype when this field change values.
> First, let's define our subtype:

```
<record id="mt_state_change" model="mail.message.subtype">
    <field name="name">Trip confirmed</field>
    <field name="res_model">business.trip</field>
    <field name="default" eval="True"/>
    <field name="description">Business Trip confirmed!</field>
</record>
```

Then, we need to override the `track_subtype()` function. This function is called by the tracking system to know which subtype should be used depending on the change currently being applied. In our case, we want to use our shiny new subtype when the `state` field changes from *draft* to *confirmed*:

```
class BusinessTrip(models.Model):
    _name = 'business.trip'
    _inherit = ['mail.thread']
    _description = 'Business Trip'

    name = fields.Char(tracking=True)
    partner_id = fields.Many2one('res.partner', 'Responsible',
                                 tracking=True)
    guest_ids = fields.Many2many('res.partner', 'Participants')
    state = fields.Selection([('draft', 'New'), ('confirmed', 'Confirmed')],
                             tracking=True)

    def _track_subtype(self, init_values):
        # init_values contains the modified fields' values before the changes
        #
        # the applied values can be accessed on the record as they are already
        # in cache
        self.ensure_one()
        if 'state' in init_values and self.state == 'confirmed':
            return self.env.ref('my_module.mt_state_change')
        return super(BusinessTrip, self)._track_subtype(init_values)
```

Customizing notifications

When sending notifications to followers, it can be quite useful to add buttons in the template to allow quick actions directly from the e-mail. Even a simple button to link directly to the record's form view can be useful; however in most cases you don't want to display these buttons to portal users.

The notification system allows customizing notification templates in the following ways:

- Display *Access Buttons*: these buttons are visible at the top of the notification e-mail and allow the recipient to directly access the form view of the record
- Display *Follow Buttons*: these buttons allow the recipient to directly quickly subscribe from the record
- Display *Unfollow Buttons*: these buttons allow the recipient to directly quickly unsubscribe from the record

Display *Custom Action Buttons*: these buttons are calls to specific routes and allow you to make some useful actions directly available from the e-mail (i.e. converting a lead to an opportunity, validating an expense sheet for an Expense Manager, etc.)

These buttons settings can be applied to different groups that you can define yourself by overriding the function **`_notification_recipients`**.

`_notification_recipients(`*`message`*`, `*`groups`*`)`

Give the subtype triggered by the changes on the record according to values that have been updated.

Parameters:

`message` (**`record`**) – **`mail.message`** record currently being sent

`groups` (**`list`** (https://docs.python.org/3/library/stdtypes.html#list) **`(`** **`tuple`** (https://docs.python.org/3/library/stdtypes.html#tuple) **`)`**) –
list of tuple of the form (group_name, group_func,group_data) where:

group_name
is an identifier used only to be able to override and manipulate groups. Default groups are **`user`** (recipients linked to an employee user), **`portal`** (recipients linked to a portal user) and **`customer`** (recipients not linked to any user). An example of override use would be to add a group linked to a res.groups like Hr Officers to set specific action buttons to them.

group_func
is a function pointer taking a partner record as parameter. This method will be applied on recipients to know whether they belong to a given group or not. Only first matching group is kept. Evaluation order is the list order.

group_data
is a dict containing parameters for the notification email with the following possible keys - values:

has_button_access
whether to display Access <Document> in email. True by default for new groups, False for portal / customer.

button_access
dict with url and title of the button

has_button_follow
whether to display Follow in email (if recipient is not currently following the thread). True by default for new groups, False for portal / customer.

button_follow
dict with url adn title of the button

has_button_unfollow
whether to display Unfollow in email (if recipient is currently following the thread). True by default for new groups, False for portal / customer.

button_unfollow
dict with url and title of the button

actions

list of action buttons to display in the notification email. Each action is a dict containing url and title of the button.

Returns:

a subtype's full external id or False if no subtype is triggered

The urls in the actions list can be generated automatically by calling the `_notification_link_helper()` function:

`_notification_link_helper(self, link_type, **kwargs)`

Generate a link for the given type on the current record (or on a specific record if the kwargs `model` and `res_id` are set).

Parameters:

link_type (`str` (https://docs.python.org/3/library/stdtypes.html#str)) –

link type to be generated; can be any of these values:

`view`
link to form view of the record

`assign`
assign the logged user to the `user_id` field of the record (if it exists)

`follow`
self-explanatory

`unfollow`
self-explanatory

`method`
call a method on the record; the method's name should be provided as the kwarg `method`

`new`
open an empty form view for a new record; you can specify a specific action by providing its id (database id or fully resolved external id) in the kwarg `action_id`

Returns:

link of the type selected for the record

Return type:

str (https://docs.python.org/3/library/stdtypes.html#str)

ⓘ Example

Let's add a custom button to the Business Trip state change notification; this button will reset the state to Draft and will be only visible to a member of the (imaginary) group Travel Manager (`business.group_trip_manager`)

```
class BusinessTrip(models.Model):
    _name = 'business.trip'
    _inherit = ['mail.thread', 'mail.alias.mixin']
    _description = 'Business Trip'

    # Pevious code goes here

    def action_cancel(self):
        self.write({'state': 'draft'})

    def _notification_recipients(self, message, groups):
        """ Handle Trip Manager recipients that can cancel the trip at the last
        minute and kill all the fun. """
        groups = super(BusinessTrip, self)._notification_recipients(message, groups)

        self.ensure_one()
        if self.state == 'confirmed':
            app_action = self._notification_link_helper('method',
                                method='action_cancel')
            trip_actions = [{'url': app_action, 'title': _('Cancel')}]

        new_group = (
            'group_trip_manager',
            lambda partner: bool(partner.user_ids) and
            any(user.has_group('business.group_trip_manager')
            for user in partner.user_ids),
            {
                'actions': trip_actions,
            })

        return [new_group] + groups
```

Note that that I could have defined my evaluation function outside of this method and define a global function to do it instead of a lambda, but for the sake of being more brief and less verbose in these documentation files that can sometimes be boring, I choose the former instead of the latter.

Overriding defaults

There are several ways you can customize the behaviour of `mail.thread` models, including (but not limited to):

`_mail_post_access` - `Model` (orm.html#odoo.models.Model) attribute

the required access rights to be able to post a message on the model; by default a `write` access is needed, can be set to `read` as well

Context keys:

These context keys can be used to somewhat control `mail.thread` features like auto-subscription or field tracking during calls to `create()` or `write()` (or any other method where it may be useful).

`mail_create_nosubscribe` : at create or message_post, do not subscribe the current user to the record thread

`mail_create_nolog` : at create, do not log the automatic '<Document> created' message

`mail_notrack` : at create and write, do not perform the value tracking creating messages

`tracking_disable` : at create and write, perform no MailThread features (auto subscription, tracking, post, ...)

`mail_auto_delete` : auto delete mail notifications; True by default

`mail_notify_force_send` : if less than 50 email notifications to send, send them directly instead of using the queue; True by default

`mail_notify_user_signature` : add the current user signature in email notifications; True by default

Mail alias

Aliases are configurable email addresses that are linked to a specific record (which usually inherits the `mail.alias.mixin` model) that will create new records when contacted via e-mail. They are an easy way to make your system accessible from the outside, allowing users or customers to quickly create records in your database without needing to connect to Odoo directly.

Aliases vs. Incoming Mail Gateway

Some people use the Incoming Mail Gateway for this same purpose. You still need a correctly configured mail gateway to use aliases, however a single catchall domain will be sufficient since all routing will be done inside Odoo. Aliases have several advantages over Mail Gateways:

Easier to configure

A single incoming gateway can be used by many aliases; this avoids having to configure multiple emails on your domain name (all configuration is done inside Odoo)

No need for System access rights to configure aliases

More coherent

Configurable on the related record, not in a Settings submenu

Easier to override server-side

Mixin model is built to be extended from the start, allowing you to extract useful data from incoming e-mails more easily than with a mail gateway.

Alias support integration

Aliases are usually configured on a parent model which will then create specific record when contacted by e-mail. For example, Project have aliases to create tasks or issues, Sales Team have aliases to generate Leads.

> The model that will be created by the alias **must** inherit the `mail_thread` model.

Alias support is added by inheriting `mail.alias.mixin` ; this mixin will generate a new `mail.alias` record for each record of the parent class that gets created (for example, every `project.project` record having its `mail.alias` record initialized on creation).

> Aliases can also be created manually and supported by a simple **Many2one (orm.html#odoo.fields.Many2one)** field. This guide assumes you wish a more complete integration with automatic creation of the alias, record-specific default values, etc.

Unlike `mail.thread` inheritance, the `mail.alias.mixin` **requires** some specific overrides to work correctly. These overrides will specify the values of the created alias, like the kind of record it must create and possibly some default values these records may have depending on the parent object:

`_get_alias_model_name(vals)`

Return the model name for the alias. Incoming emails that are not replies to existing records will cause the creation of a new record of this alias model. The value may depend on `vals` , the dict of values passed to `create` when a record of this model is created.

Parameters:

dict (`vals`) – values of the newly created record that will holding the alias

Returns:

model name

Return type:

str (https://docs.python.org/3/library/stdtypes.html#str)

`_get_alias_values()`

Return values to create an alias, or to write on the alias after its creation. While not completely mandatory, it is usually required to make sure that newly created records will be linked to the alias' parent (i.e. tasks getting created in the right project) by setting a dictionary of default values in the alias' `alias_defaults` field.

Returns:

dictionnary of values that will be written to the new alias

Return type:

dict (https://docs.python.org/3/library/stdtypes.html#dict)

The `_get_alias_values()` override is particularly interesting as it allows you to modify the behaviour of your aliases easily. Among the fields that can be set on the alias, the following are of particular interest:

`alias_name` - `Char` (orm.html#odoo.fields.Char)
name of the email alias, e.g. 'jobs' if you want to catch emails for <jobs@example.odoo.com (mailto:jobs@example.odoo.com)>

`alias_user_id` - `Many2one` (orm.html#odoo.fields.Many2one) (`res.users`)
owner of records created upon receiving emails on this alias; if this field is not set the system will attempt to find the right owner based on the sender (From) address, or will use the Administrator account if no system user is found for that address

`alias_defaults` - `Text` (orm.html#odoo.fields.Text)
Python dictionary that will be evaluated to provide default values when creating new records for this alias

`alias_force_thread_id` - `Integer` (orm.html#odoo.fields.Integer)
optional ID of a thread (record) to which all incoming messages will be attached, even if they did not reply to it; if set, this will disable the creation of new records completely

`alias_contact` - `Selection` (orm.html#odoo.fields.Selection)
Policy to post a message on the document using the mailgateway

everyone: everyone can post

partners: only authenticated partners

followers: only followers of the related document or members of following channels

Note that aliases make use of delegation inheritance (orm.html#reference-orm-inheritance), which means that while the alias is stored in another table, you have access to all these fields directly from your parent object. This allows you to make your alias easily configurable from the record's form view.

> **ⓘ Example**
> Let's add aliases on our business trip class to create expenses on the fly via e-mail.

```
class BusinessTrip(models.Model):
    _name = 'business.trip'
    _inherit = ['mail.thread', 'mail.alias.mixin']
    _description = 'Business Trip'

    name = fields.Char(tracking=True)
    partner_id = fields.Many2one('res.partner', 'Responsible',
                                 tracking=True)
    guest_ids = fields.Many2many('res.partner', 'Participants')
    state = fields.Selection([('draft', 'New'), ('confirmed', 'Confirmed')],
                             tracking=True)
    expense_ids = fields.One2many('business.expense', 'trip_id', 'Expenses')
    alias_id = fields.Many2one('mail.alias', string='Alias', ondelete="restrict",
                               required=True)

    def _get_alias_model_name(self, vals):
    """ Specify the model that will get created when the alias receives a message ""
        return 'business.expense'

    def _get_alias_values(self):
    """ Specify some default values that will be set in the alias at its creation ""
        values = super(BusinessTrip, self)._get_alias_values()
        # alias_defaults holds a dictionnary that will be written
        # to all records created by this alias
        #
        # in this case, we want all expense records sent to a trip alias
        # to be linked to the corresponding business trip
        values['alias_defaults'] = {'trip_id': self.id}
        # we only want followers of the trip to be able to post expenses
        # by default
        values['alias_contact'] = 'followers'
        return values

class BusinessExpense(models.Model):
    _name = 'business.expense'
    _inherit = ['mail.thread']
    _description = 'Business Expense'

    name = fields.Char()
    amount = fields.Float('Amount')
    trip_id = fields.Many2one('business.trip', 'Business Trip')
    partner_id = fields.Many2one('res.partner', 'Created by')
```

We would like our alias to be easily configurable from the form view of our business trips, so let's add the following to our form view:

```
<page string="Emails">
    <group name="group_alias">
        <label for="alias_name" string="Email Alias"/>
        <div name="alias_def">
            <!-- display a link while in view mode and a configurable field
            while in edit mode -->
            <field name="alias_id" class="oe_read_only oe_inline"
                    string="Email Alias" required="0"/>
            <div class="oe_edit_only oe_inline" name="edit_alias"
                 style="display: inline;" >
                <field name="alias_name" class="oe_inline"/>
                @
                <field name="alias_domain" class="oe_inline" readonly="1"/>
            </div>
        </div>
        <field name="alias_contact" class="oe_inline"
                string="Accept Emails From"/>
    </group>
</page>
```

Now we can change the alias address directly from the form view and change who can send e-mails to the alias.

We can then override `message_new()` on our expense model to fetch the values from our email when the expense will be created:

```
class BusinessExpense(models.Model):
    # Previous code goes here
    # ...

    def message_new(self, msg, custom_values=None):
        """ Override to set values according to the email.

        In this simple example, we simply use the email title as the name
        of the expense, try to find a partner with this email address and
        do a regex match to find the amount of the expense."""
        name = msg_dict.get('subject', 'New Expense')
        # Match the last occurence of a float in the string
        # Example: '50.3 bar 34.5' becomes '34.5'. This is potentially the price
        # to encode on the expense. If not, take 1.0 instead
        amount_pattern = '(\d+(\.\d*)?|\.\d+)'
        expense_price = re.findall(amount_pattern, name)
        price = expense_price and float(expense_price[-1][0]) or 1.0
        # find the partner by looking for it's email
        partner = self.env['res.partner'].search([('email', 'ilike', email_address)]
                                                 limit=1)
        defaults = {
            'name': name,
            'amount': price,
            'partner_id': partner.id
        }
        defaults.update(custom_values or {})
        res = super(BusinessExpense, self).message_new(msg, custom_values=defaults)
        return res
```

Activities tracking

Activities are actions users have to take on a document like making a phone call or organizing a meeting. Activities come with the mail module as they are integrated in the Chatter but are *not bundled with mail.thread*. Activities are records of the `mail.activity` class, which have a type (`mail.activity.type`), name, description, scheduled time (among others). Pending activities are visible above the message history in the chatter widget.

You can integrate activities using the `mail.activity.mixin` class on your object and the specific widgets to display them (via the field `activity_ids`) in the form view and kanban view of your records (`mail_activity` and `kanban_activity` widgets, respectively).

> **ⓘ Example**
>
> Organizing a business trip is a tedious process and tracking needed activities like ordering plane tickets or a cab for the airport could be useful. To do so, we will add the activities mixin on our model and display the next planned activities in the message history of our trip.
>
> ```
> class BusinessTrip(models.Model):
>     _name = 'business.trip'
>     _inherit = ['mail.thread', 'mail.activity.mixin']
>     _description = 'Business Trip'
>
>     name = fields.Char()
>     # [...]
> ```
>
> We modify the form view of our trips to display their next activites:
>
> ```
> <record id="businness_trip_form" model="ir.ui.view">
>     <field name="name">business.trip.form</field>
>     <field name="model">business.trip</field>
>     <field name="arch" type="xml">
>         <form string="Business Trip">
>             <!-- Your usual form view goes here -->
>             
>                 <field name="message_follower_ids" widget="mail_followers"/>
>                 <field name="activity_ids" widget="mail_activity"/>
>                 <field name="message_ids" widget="mail_thread"/>
>             
>         </form>
>     </field>
> </record>
> ```

You can find concrete examples of integration in the following models:

- `crm.lead` in the CRM (*crm*) Application
- `sale.order` in the Sales (*sale*) Application
- `project.task` in the Project (*poject*) Application

Website features

Visitor tracking

The **utm.mixin** class can be used to track online marketing/communication campaigns through arguments in links to specified resources. The mixin adds 3 fields to your model:

campaign_id : **Many2one** (orm.html#odoo.fields.Many2one) field to a **utm.campaign** object (i.e. Christmas_Special, Fall_Collection, etc.)

source_id : **Many2one** (orm.html#odoo.fields.Many2one) field to a **utm.source** object (i.e. Search Engine, mailing list, etc.)

medium_id : **Many2one** (orm.html#odoo.fields.Many2one) field to a **utm.medium** object (i.e. Snail Mail, e-Mail, social network update, etc.)

These models have a single field **name** (i.e. they are simply there to distinguish campaigns but don't have any specific behaviour).
Once a customer visits your website with these parameters set in the url (i.e. https://www.odoo.com/?campaign_id=mixin_talk&source_id=www.odoo.com&medium_id=website (https://www.odoo.com/?campaign_id=mixin_talk&source_id=www.odoo.com&medium_id=website)), three cookies are set in the visitor's website for these parameters. Once a object that inherits the utm.mixin is created from the website (i.e. lead form, job application, etc.), the utm.mixin code kicks in and fetches the values from the cookies to set them in the new record. Once this is done, you can then use the campaign/source/medium fields as any other field when defining reports and views (group by, etc.).
To extend this behaviour, simply add a relational field to a simple model (the model should support the *quick create* (i.e. call to **create()** with a single **name** value) and extend the function **tracking_fields()** :

```
class UtmMyTrack(models.Model):
    _name = 'my_module.my_track'
    _description = 'My Tracking Object'

    name = fields.Char(string='Name', required=True)


class MyModel(models.Models):
    _name = 'my_module.my_model'
    _inherit = ['utm.mixin']
    _description = 'My Tracked Object'

    my_field = fields.Many2one('my_module.my_track', 'My Field')

    @api.model
    def tracking_fields(self):
        result = super(MyModel, self).tracking_fields()
        result.append([
        # ("URL_PARAMETER", "FIELD_NAME_MIXIN", "NAME_IN_COOKIES")
            ('my_field', 'my_field', 'odoo_utm_my_field')
        ])
        return result
```

This will tell the system to create a cookie named *odoo_utm_my_field* with the value found in the url parameter **`my_field`** ; once a new record of this model is created by a call from a website form, the generic override of the **`create()`** method of **`utm.mixin`** will fetch the default values for this field from the cookie (and the **`my_module.my_track`** record will be creatwed on the fly if it does not exist yet).
You can find concrete examples of integration in the following models:

- **`crm.lead`** in the CRM (*crm*) Application
- **`hr.applicant`** in the Recruitment Process (*hr_recruitment*) Application
- **`helpdesk.ticket`** in the Helpdesk (*helpdesk* - Odoo Enterprise only) Application

Website visibility

You can quite easily add a website visibility toggle on any of your record. While this mixin is quite easy to implement manually, it is the most often-used after the **`mail.thread`** inheritance; a testament to its usefulness. The typical use case for this mixin is any object that has a frontend-page; being able to control the visibility of the page allows you to take your time while editing the page and only publish it when you're satisfied.
To include the functionnality, you only need to inherit **`website.published.mixin`** :

```
class BlogPost(models.Model):
    _name = "blog.post"
    _description = "Blog Post"
    _inherit = ['website.published.mixin']
```

This mixin adds 2 fields on your model:

website_published : **Boolean** (orm.html#odoo.fields.Boolean) field which represents the status of the publication

website_url : **Char** (orm.html#odoo.fields.Char) field which represents the URL through which the object is accessed

Note that this last field is a computed field and must be implemented for your class:

```
def _compute_website_url(self):
    for blog_post in self:
        blog_post.website_url = "/blog/%s" % (log_post.blog_id)
```

Once the mechanism is in place, you just have to adapt your frontend and backend views to make it accessible. In the backend, adding a button in the button box is usually the way to go:

```
<button class="oe_stat_button" name="website_publish_button"
    type="object" icon="fa-globe">
    <field name="website_published" widget="website_button"/>
</button>
```

In the frontend, some security checks are needed to avoid showing 'Editing' buttons to website visitors:

```
<div id="website_published_button" class="float-right"
     groups="base.group_website_publisher"> <!-- or any other meaningful group -->
    <t t-call="website.publish_management">
      <t t-set="object" t-value="blog_post"/>
      <t t-set="publish_edit" t-value="True"/>
      <t t-set="action" t-value="'blog.blog_post_action'"/>
    </t>
</div>
```

Note that you must pass your object as the variable **object** to the template; in this example, the **blog.post** record was passed as the **blog_post** variable to the **qweb** rendering engine, it is necessary to specify this to the publish management template. The **publish_edit** variable allow the frontend button to link to the backend (allowing you to switch from frontend to backend and vice-versa easily); if set, you must specify the full external id of the action you want to call in the backend in the **action** variable (note that a Form View must exist for the model).
The action **website_publish_button** is defined in the mixin and adapts its behaviour to your object: if the class has a valid **website_url** compute function, the user is redirected to the frontend when he clicks on the button; the user can then publish the page directly from the frontend. This ensures that no online publication can happen by accident. If there is not compute function, the boolean **website_published** is simply triggered.

Website metadata

This simple mixin simply allows you to easily inject metadata in your frontend pages.

```
class BlogPost(models.Model):
    _name = "blog.post"
    _description = "Blog Post"
    _inherit = ['website.seo.metadata', 'website.published.mixin']
```

This mixin adds 3 fields on your model:

- **website_meta_title** : **Char** (orm.html#odoo.fields.Char) field that allow you to set an additional title to your page
- **website_meta_description** : **Char** (orm.html#odoo.fields.Char) field that contains a short description of the page (sometimes used in search engines results)
- **website_meta_keywords** : **Char** (orm.html#odoo.fields.Char) field that contains some keywords to help your page to be classified more precisely by search engines; the "Promote" tool will help you select lexically-related keywords easily

These fields are editable in the frontend using the "Promote" tool from the Editor toolbar. Setting these fields can help search engines to better index your pages. Note that search engines do not base their results only on these metadata; the best SEO practice should still be to get referenced by reliable sources.

Others

Customer Rating

The rating mixin allows sending email to ask for customer rating, automatic transitioning in a kanban processes and aggregating statistics on your ratings.

Adding rating on your model

To add rating support, simply inherit the **rating.mixin** model:

```
class MyModel(models.Models):
    _name = 'my_module.my_model'
    _inherit = ['rating.mixin', 'mail.thread']

    user_id = fields.Many2one('res.users', 'Responsible')
    partner_id = fields.Many2one('res.partner', 'Customer')
```

The behaviour of the mixin adapts to your model:

- The **rating.rating** record will be linked to the **partner_id** field of your model (if the field is present).
  - this behaviour can be overridden with the function **rating_get_partner_id()** if you use another field than **partner_id**

The `rating.rating` record will be linked to the partner of the `user_id` field of your model (if the field is present) (i.e. the partner who is rated)

this behaviour can be overridden with the function `rating_get_rated_partner_id()` if you use another field than `user_id` (note that the function must return a `res.partner`, for `user_id` the system automatically fetches the partner of the user)

The chatter history will display the rating event (if your model inherits from `mail.thread`)

Send rating requests by e-mail

If you wish to send emails to request a rating, simply generate an e-mail with links to the rating object. A very basic email template could look like this:

```
<record id="rating_my_model_email_template" model="mail.template">
            <field name="name">My Model: Rating Request</field>
            <field name="email_from">${object.rating_get_rated_partner_id().email or ''
            <field name="subject">Service Rating Request</field>
            <field name="model_id" ref="my_module.model_my_model"/>
            <field name="partner_to" >${object.rating_get_partner_id().id}</field>
            <field name="auto_delete" eval="True"/>
            <field name="body_html"><![CDATA[
% set access_token = object.rating_get_access_token()
<p>Hi,</p>
<p>How satsified are you?</p>
<ul>
    <li><a href="/rate/${access_token}/5">Satisfied</a></li>
    <li><a href="/rate/${access_token}/3">Not satisfied</a></li>
    <li><a href="/rate/${access_token}/1">Very unsatisfied</a></li>
</ul>
]]></field>
</record>
```

Your customer will then receive an e-mail with links to a simple webpage allowing them to provide a feedback on their interaction with your users (including a free-text feedback message).
You can then quite easily integrate your ratings with your form view by defining an action for the ratings:

```
<record id="rating_rating_action_my_model" model="ir.actions.act_window">
    <field name="name">Customer Ratings</field>
    <field name="res_model">rating.rating</field>
    <field name="view_mode">kanban,pivot,graph</field>
    <field name="domain">[('res_model', '=', 'my_module.my_model'), ('res_id', '=', acti
</record>

<record id="my_module_my_model_view_form_inherit_rating" model="ir.ui.view">
    <field name="name">my_module.my_model.view.form.inherit.rating</field>
    <field name="model">my_module.my_model</field>
    <field name="inherit_id" ref="my_module.my_model_view_form"/>
    <field name="arch" type="xml">
        <xpath expr="//div[@name='button_box']" position="inside">
            <button name="%(rating_rating_action_my_model)d" type="action"
                    class="oe_stat_button" icon="fa-smile-o">
                <field name="rating_count" string="Rating" widget="statinfo"/>
            </button>
        </xpath>
    </field>
</record>
```

Note that there are default views (kanban,pivot,graph) for ratings which allow you a quick bird's eye view of your customer ratings.

You can find concrete examples of integration in the following models:

- **`project.task`** in the Project (*rating_project*) Application
- **`helpdesk.ticket`** in the Helpdesk (*helpdesk* - Odoo Enterprise only) Application