Understanding the ORM decorators

ORM decorators are important for the ORM, and allow it to give those methods specific uses.

Let’s see the ORM decorators we have available, and when each should be used.

Record handling decorators

Most of the time, we want a custom method to perform some actions on a recordset. For this, we should use @api.multi, and in that case, the self argument will be the recordset to work with. The method’s logic will usually include a for loop iterating on it. This is surely the most frequently used decorator.

If no decorator is used on a model method, it will default to  @api.multi behavior.

In some cases, the method is prepared to work with a single record (a singleton). Here we could use the  @api.one decorator, but this is not advised because for Version 9.0 it was announced it would be deprecated and may be removed in the future.

Instead, we should use @api.multi and add to the method code a line with self.ensure\_one(), to ensure it is a singleton as expected. Despite being deprecated, the @api.one decorator is still supported. So it’s worth knowing that it wraps the decorated method, doing the for-loop iteration to feed it one record at a time. So, in an @api.onedecorated method, self is guaranteed to be a singleton. The return values of each individual method call are aggregated as a list and then returned.

The return value of @api.one can be tricky: it returns a list, not the data structure returned by the actual method. For example, if the method code returns a dict, the actual return value is a list of dict values. This misleading behavior was the main reason the method was deprecated.

In some cases, the method is expected to work at the class level, and not on particular records. In some object-oriented languages this would be called a static method. These class-level static methods should be decorated with @api.model. In these cases, self should be used as a reference for the model, without expecting it to contain actual records.

Methods decorated with @api.model cannot be used with user interface buttons. In those cases, @api.multi should be used instead.

Specific purpose decorators

A few other decorators have more specific purposes and are to be used together with the decorators described earlier:

* @api.depends(fld1,...) is used for computed field functions, to identify on what changes the (re)calculation should be triggered. It must set values on the computed fields, otherwise it will error.
* @api.constrains(fld1,...) is used for validation functions, and performs checks for when any of the mentioned fields are changed. It should not write changes in the data. If the checks fail, an exception should be raised.
* @api.onchange(fld1,...) is used in the user interface, to automatically change some field values when other fields are changed. The self argument is a singleton with the current form data, and the method should set values on it for the changes that should happen in the form. It doesn’t actually write to database records, instead it provides information to change the data in the UI form.

When using the preceding decorators, no return value is needed. Except for onchangemethods that can optionally return a dict with a warning message to display in the user interface.

As an example, we can use this to perform some automation in the To-Do form: when Responsible is set to an empty value, we will also empty the team list. For this, edit the todo\_stage/models/todo\_task\_model.py file to add the following method:

@api.onchange('user\_id')

def onchange\_user\_id(self):

if not user\_id:

self.team\_ids = None

return {

'warning': {

'title': 'No Responsible',

'message': 'Team was also reset.'

}

}

Here, we are using the @api.onchange decorator to attach some logic to any changes in the user\_id field, when done through the user interface. Note that the actual method name is not relevant, but the convention is for its name to begin with onchange\_.

Inside an onchange method, self represents a single virtual record containing all the fields currently set in the record being edited, and we can interact with them. Most of the time, this is what we want to do: to automatically fill values in other fields, depending on the value set to the changed field. In this case, we are setting the team\_ids field to an empty value.

The onchange methods don’t need to return anything, but they can return a dictionary containing a warning or a domain key:

* The warning key should describe a message to show in a dialogue window, such as: {'title': 'Message Title', 'message': 'Message Body'}.
* The domain key can set or change the domain attribute of other fields. This allows you to build more user-friendly interfaces, by having to-many fields list only the selection option that make sense for this case. The value for the domain key looks like this: {'team\_ids': [('is\_author', '=', True)]}

Using the ORM built-in methods

The decorators discussed in the previous section allow us to add certain features to our models, such as implementing validations and automatic computations.

We also have the basic methods provided by the ORM, used mainly to perform CRUD (create, read, update and delete) operations on our model data. To read data, the main methods provided are search() and browse().

Now we will explore the write operations provided by the ORM, and how they can be extended to support custom logic.

Methods for writing model data

The ORM provides three methods for the three basic write operations:

* .**create(values)** creates a new record on the model. Returns the created record.
* .**write(values)** updates field values on the recordset. Returns nothing.
* .**unlink()** deletes the records from the database. Returns nothing.

The values argument is a dictionary, mapping field names to values to write.

In some cases, we need to extend these methods to add some business logic to be triggered whenever these actions are executed. By placing our logic in the appropriate section of the custom method, we can have the code run before or after the main operations are executed.

Using the TodoTask model as an example, we can make a custom create(), which would look like this:

@api.model

def create(self, vals):

*# Code before create: should use the `vals` dict*

new\_record = super(TodoTask, self).create(vals)

*# Code after create: can use the `new\_record` created*

return new\_record

[Python](https://subscribe.packtpub.com/learn-python/) 3 introduced a simplified way to use super() that could have been used in the preceding code samples. We chose to use the [Python](https://subscribe.packtpub.com/learn-python/) 2 compatible form. If we don’t mind breaking Python 2 support for our code, we can use the simplified form, without the arguments referencing the class name and self. For example: super().create(vals)

A custom write() would follow this structure:

@api.multi

def write(self, vals):

*# Code before write: can use `self`, with the old values*

super(TodoTask, self).write(vals)

*# Code after write: can use `self`, with the updated values*

return True

While extending create() and write()  opens up a lot of possibilities, remember in many cases we don’t need to do that, since there are tools also available that may be better suited:

* For field values that are automatically calculated based on other fields, we should use computed fields. An example of this is to calculate a header total when the values of the lines are changed.
* To have field default values calculated dynamically, we can use a field default bound to a function instead of a fixed value.
* To have values set on other fields when a field is changed, we can use onchangefunctions. An example of this is when picking a customer, setting their currency as the document’s currency that can later be manually changed by the user. Keep in mind that on change only works on form view interaction and not on direct write calls.
* For validations, we should use constraint functions decorated with @api.constraints(fld1,fld2,...). These are like computed fields but, instead of computing values, they are expected to raise errors.

Consider carefully if you really need to use extensions to the create or write methods. In most cases, we just need to perform some validation or automatically compute some value, when the record is saved. But we have better tools for this: validations are best implemented with @api.constrains methods, and automatic calculations are better implemented as computed fields. In this case, we need to compute field values when saving. If, for some reason, computed fields are not a valid solution, the best approach is to have our logic at the top of the method, accumulating the changes needed into the valsdictionary that will be passed to the final super() call.

For the write() method, having further write operations on the same model will lead to a recursion loop and end with an error when the worker process resources are exhausted. Please consider if this is really needed. If it is, a technique to avoid the recursion loop is to set a flag in the context. For example, we could add code such as the following:

if not self.env.context.get('todo\_task\_writing'):

self.with\_context(todo\_task\_writing=True).write(

some\_values)

With this technique, our specific logic is guarded by an if statement, and runs only if a specific marker is not found in the context. Furthermore, our self.write() operations should use with\_context to set that marker. This combination ensures that the custom login inside the if statement runs only once, and is not triggered on further write() calls, avoiding the infinite loop.

These are common extension examples, but of course any standard method available for a model can be inherited in a similar way to add our custom logic to it.

Methods for web client use over RPC

We have seen the most important model methods used to generate recordsets and how to write to them, but there are a few more model methods available for more specific actions, as shown here:

* read([fields]) is similar to the browse method, but, instead of a recordset, it returns a list of rows of data with the fields given as its argument. Each row is a dictionary. It provides a serialized representation of the data that can be sent through RPC protocols and is intended to be used by client programs and not in server logic.
* search\_read([domain], [fields], offset=0, limit=None, order=None) performs a search operation followed by a read on the resulting record list. It is intended to be used by RPC clients and saves them the extra round trip needed when doing a search followed by a read on the results.

Methods for data import and export

The import and export operations, are also available from the ORM API, through the following methods:

* load([fields], [data]) is used to import data acquired from a CSV file. The first argument is the list of fields to import, and it maps directly to a CSV top row. The second argument is a list of records, where each record is a list of string values to parse and import, and it maps directly to the CSV data rows and columns. It implements the features of CSV data import, such as the external identifiers support. It is used by the web client Import feature.
* export\_data([fields], raw\_data=False) is used by the web client Export function. It returns a dictionary with a data key containing the data: a list of rows. The field names can use the .id and /id suffixes used in CSV files, and the data is in a format compatible with an importable CSV file. The optional raw\_data argument allows for data values to be exported with their Python types, instead of the string representation used in CSV.

Methods for the user interface

The following methods are mostly used by the web client to render the user interface and perform basic interaction:

* name\_get()  returns a list of (ID, name) tuples with the text representing each record. It is used by default for computing the display\_name value, providing the text representation of relation fields. It can be extended to implement custom display representations, such as displaying the record code and name instead of only the name.
* name\_search(name='', args=None, operator='ilike', limit=100) returns a list of (ID, name) tuples, where the display name matches the text in the name argument. It is used in the UI while typing in a relation field to produce the list with the suggested records matching the typed text. For example, it is used to implement product lookup both by name and by reference, while typing in a field to pick a product.
* name\_create(name) creates a new record with only the title name to use for it. It is used in the UI for the “quick-create” feature, where you can quickly create a related record by just providing its name. It can be extended to provide specific defaults for the new records created through this feature.
* default\_get([fields]) returns a dictionary with the default values for a new record to be created. The default values may depend on variables such as the current user or the session context.
* fields\_get() is used to describe the model’s field definitions, as seen in the View Fieldsoption of the developer menu.
* fields\_view\_get() is used by the web client to retrieve the structure of the UI view to render. It can be given the ID of the view as an argument or the type of view we want using view\_type='form'. For example, you may try this: self.fields\_view\_get(view\_type='tree').

The Mail and Social features API

Odoo has available global messaging and activity planning features, provided by the Discuss app, with the technical name mail.

The mail module provides the mail.thread abstract class that makes it simple to add the messaging features to any model. To add the mail.thread features to the To-Do tasks, we just need to inherit from it:

class TodoTask(models.Model):

**\_name = 'todo.task'**

**\_inherit = ['todo.task', 'mail.thread']**

After this, among other things, our model will have two new fields available. For each record (sometimes also called a document) we have:

* mail\_follower\_ids stores the followers, and corresponding notification preferences
* mail\_message\_ids lists all the related messages

The followers can be either partners or channels. A partner represents a specific person or organization. A channel is not a particular person, and instead represents a subscription list.

Each follower also has a list of message types that they are subscribed to. Only the selected message types will generate notifications for them.

Message subtypes

Some types of messages are called subtypes. They are stored in the mail.message.subtypemodel and accessible in the Technical | Email | Subtypes menu.

By default, we have three message subtypes available:

* **Discussions**, with mail.mt\_comment XMLID, used for the messages created with the Send message link. It is intended to send a notification.
* **Activities,** with mail.mt\_activities XMLID, used for the messages created with the Schedule activity link. It is intended to send a notification.
* **Note**, with mail.mt\_note XMLID, used for the messages created with the Log note link. It is not intended to send a notification.

Subtypes have the default notification settings described previously, but users are able to change them for specific documents, for example, to mute a discussion they are not interested in.

Other than the built-in subtypes, we can also add our own subtypes to customize the notifications for our apps. Subtypes can be generic or intended for a particular model. For the latter case, we should fill in the subtype’s res\_model field with the name of the model it should apply to.

Posting messages

Our business logic can make use of this messaging system to send notifications to users. To post a message we use the message\_post() method. For example:

self.message\_post('Hello!')

This adds a simple text message, but sends no notification to the followers. That is because by default the mail.mt\_note subtype is used for the posted messages. But we can have the message posted with the particular subtype we want. To add a message and have it send notifications to the followers, we should use the following:

self.message\_post('Hello again!', subtype='mail.mt\_comment')

We can also add a subject line to the message by adding the subject parameter. The message body is HTML, so we can include markup for text effects, such as  for bold text or for italics.

The message body will be sanitized for security reasons, so some particular HTML elements may not make it to the final message.

Adding followers

Also interesting from a business logic viewpoint is the ability to automatically add followers to a document, so that they can then get the corresponding notifications. For this we have several methods available to add followers:

* message\_subscribe(partner\_ids=) adds Partners
* message\_subscribe(channel\_ids=) adds Channels
* message\_subscribe\_users(user\_ids=) adds Users

The default subtypes will be used. To force subscribing a specific list of subtypes, just add the subtype\_ids= with the specific subtypes you want to be subscribed.

In this article, we went through an explanation of the features the ORM API proposes, and how they can be used when creating our models. We also learned about the mail module and the global messaging features it provides.