GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

UD06. DESARROLLO DE MÓDULOS. CONTROLADOR

Sistemas de Gestión Empresarial
2 Curso // CFGS DAM // Informática y Comunicaciones
Alfredo Oltra

Cicles Formatius

ÍNDEX

Versión: 240202.0127

Licencia

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

☑ **Atención**. Importante prestar atención a esta información.

Interesante. Ofrece información sobre algun detalle a tener en cuenta.

1 INTRODUCCIÓN

El controlador define la capa de negocio, es decir, lo que el módulo tienen que hacer a partir de los datos definidos en los modelos. En ocasiones, para clarificar, se suele decir que el controlador son los métodos que hay en los modelos.

☑ La arquitectura MVC (Modelo-Vista-Controlador) nos permite desarrollar cada una de estas partes por separado. No obstante, es habitual que el controlador se desarrolle en los mismos ficheros Python en los que se hace el modelo.

En la unidad anterior hemos visto los *fields* computados y cómo funcionan las funciones en python y *Odoo*. En este apartado vamos a ver las facilidades que proporciona el framework de *Odoo* para manipular el *ORM* (Object Relational Mapping).

☑ Llegados a este punto, se supone que hay un nivel mínimo de conocimientos de programación y del lenguaje de programación python.

La capa ORM tiene unos métodos para manipular los datos sin necesidad de hacer sentencias SQL contra la base de datos.

☑ *Odoo* tiene una herramienta para acceder por terminal en vez de por la web. Para ello debemos escribir en la terminal cuando reiniciamos el servidor: `odoo shell`

El acceso a la terminal nos permite probar las instrucciones del *ORM* sin tener que editar archivos y reiniciar el servidor. Cuando en este capítulo veamos ejemplos de código del estilo:

```
>>> print(self)
```

Indica que se han hecho en la terminal de *Odoo* y que sería interesante hacerlo para probar su funcionamiento.

Si estamos utilizando *Odoodock*, el acceso se realiza ejecutando desde la carpeta odoodock el comando:

```
$ docker exec -it odoodock-web-1 bash -c "odoo shell -d NOMBREBD"
```

donde *NOMBREBD* es el nombre de la base de datos.

2 ENVIRONMENT

El llamado *environment* o *env* guarda algunos datos contextuales interesantes para trabajar con el ORM, como el cursor en la base de datos, el usuario actual o el contexto (que guarda algunos metadatos).

Todos los *recordset* (y *singleton*) tienen un *environment* accesible mediante el atributo env. En cuanto queremos acceder a un *recordset* dentro de otro, podemos usar *env*:

```
>>> self.env['res.partner']
res.partner
>>> self.env['res.partner'].search([['is_company', '=', True], ['customer', '=',
True]])
res.partner(7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23, 28, 74)
```

Dentro del *environment* encontramos el identificador del usuario actual (*uid*) o el cursor.

```
>>> env.uid
1
```

Además tambień nos encotramos al *context*. El atributo *context* es un diccionario de *python* que contiene datos útiles para las vistas y los métodos. Las funciones en *Odoo* reciben el *context* y lo consultan o actualizan si lo necesitan. Puede tener casi de todo, pero al menos siempre el idioma y la zona horaria.

```
>>> env.context
{'lang': 'es_ES', 'tz': 'Europe/Brussels'}
```

Ya hemos usado anteriormente *context* y lo usaremos en esta unidad didáctica. Para ver su importancia es conveniente repasar los usos comunes dentro de *Odoo:*

`active_id`

Indica el identificador del registro activo en ese momento. Es muy útil cuando en una vista queremos que los *fields One2many* abran un formulario con el *field Many2one* por defecto. Para ello se le pasa el *active_id* de esta manera: `context="{'default_<field many2one>':active_id}"`.

Como se puede ver, lo que hace este atributo es ampliar el *context* y añadir una clave con un valor. Los formularios en *Odoo* recogen este *context* y buscan las claves que sean *default_<field>.* Si las encuentran, ponen un valor por defecto.

Esto también funciona en los action que veremos en las vistas en caso de que se desee un *field* por defecto. El *active_id* también está en el *context* y se puede acceder desde una función con la instrucción `self.env.context.get('active_id').`

`group_by`

En la vista *search* se almacena el criterio de agrupación añadiendo la clave `group_by` en el atributo *context*.

3 RECORDSET

Para *Odoo*, un conjunto de registros de un modelo se llama *Recordset* y un conjunto con un solo registro de un modelo es un *Singleton*. La interacción con el ORM se basa en manipular *recordsets* o recorrerlos para ir manipulando los *singletons*.

```
def do_operation(self):
    print self # => a.model(1, 2, 3, 4, 5) (Recordset)
    for record in self:
        print record # => a.model(1), a.model(2), a.model(3), ... (Singletons)
```

Recordamos que para acceder a los datos, hay que ir a los *singletons* y no a los *recordsets.* En el ejemplo siguiente se supone que *a* es un recordset de tipo *school_course* con dos registros. La forma de obtener esos *id* es:

```
>>> a
school.course(1, 2)
>>> for i in a:
...  i.name
...
'2DAM'
'1DAM'
>>> a.name
ValueError: too many values to unpack
```

Si se intenta acceder a los datos en un *recordset* da error. Los datos de los *fields* relacionales dan un *recordset*, incluso aunque solo tengan un elemento:

```
>>> a[0].students
res.partner(14, 26, 33, 27, 10)
```

Los *recordsets* son iterables como una lista/array y tienen, además, operaciones de conjuntos que permiten facilitar el trabajo con ellos:

- `record in set`**:** retorna *True* si *record* está en el conjunto *set*.
- `set1 | set2`**:** unión de conjuntos. También funciona el signo +.
- `set1 & set2`**:** intersección de conjuntos (sets).
- `set1 - set2`**:** diferencia de conjuntos (sets).

Además, cuenta con funciones propias de la programación funcional:

`filtered()`

Retorna un *recordset* con los elementos del *recordset* que pasen el filtro. Para pasar el filtro, se necesita que retorne un *True*, ya sea una función *lambda* o un *field* booleano. Similar a `filter`.

```
records.filtered(lambda r: r.company_id == user.company_id)
records.filtered("partner_id.is_company")
```

`sorted()`

Retorna un *recordset* ordenado según el resultado de una función *lambda* aplicado a su *key function.* Similar a `sort`.

Una *key function* es una fucnión que se utiliza para extraer un valor clave de cada elemento de un iterable. La función se aplica a cada elemento antes de realizar la comparación y los elementos se ordenan según los resultados.

```
# sort records by name
records.sorted(key=lambda r: r.name)
records.sorted(key=lambda r: r.name, reverse=True)
```

`mapped()`

Le aplica una función a cada elemento del *recordset* y retorna un *recordset* con los cambios pedidos. La función también puede ser una cadena con el nombre de un campo. Similar a `map`.

```
records.mapped(lambda r: r.field1 + r.field2)
# devuelve una lista con los nombres de los registros
records.mapped('name')
# devuelve un recordset de los estudiantes asociados
record.mapped('students_ids')

record.mapped('partner_id.bank_ids')
```

4 MÉTODOS DEL *ORM*

Gran parte de la potencia del ORM es debida a los métodos que incluye para realizar de manera sencilla las operaciones más habituales.

`search()`

A partir de la definición de un dominio extrae un *recordset* con los registros que coinciden

```
>>> # searches the current model
>>> self.search([('is_company', '=', True), ('customer', '=', True)])
res.partner(7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23,
28, 74)
>>> self.search([('is_company', '=', True)], limit=1).name
'Agrolait'
```

Una función asociada es `search_count` que funciona de forma similar, pero retornando únicamente la cantidad de registros encontrados.

Los parámetros que acepta `search` son:

- `args`: un dominio de búsqueda. Si se deja como `[]` nos mostrará todos los registros.
- `offset (int)`: número de resultados a ignorar. Se puede combinar con *limit* si queremos paginar nuestra llamada a *search*.
- `limit (int)`: número máximo de resultados a extraer.
- `order (str)`: string de ordenación con el mismo formato que en SQL (por ejemplo: `order='date DESC'`).
- `count (bool)`: para que se comporte como `search_count()`.

`create()`

Crea y retorna un nuevo *singleton* a partir de la definición de varios de sus *fields*.

```
>>> self.create({'name': "Nombre"})
res.partner(78)
```

`write()`

Escribe información en el *recordset* desde el cual se invoca:

```
self.write({'name': "Nuevo nombre"})
```

Hay un caso especial de `write` cuando se intenta escribir en un *Many2many*. En esa situación es posible necesitar realizar operaciones un poco más complejas como, por ejemplo, la vinculación con registros existentes o la creación de los registros relacionados. En esos casos existen dos formas de actuar en función de la versión de *Odoo* con la que estemos trabajando.

Versión 14 y anteriores.

```
self.write({'sessions':[(4,s.id)]})
self.write({'sessions':[(0,0, { date: '2023-22-12', hash: 'adehf6734ee4845'})]})
self.write({'sessions':[(6,0,
                        [ref('vehicle_tag_leasing'),
                         ref('fleet.vehicle_tag_compact'),
                         ref('fleet.vehicle_tag_senior')] )]})
```

Como se observa, se le pasa una tupla de 2 o 3 elementos. El primero es un código numérico indicando qué se quiere hacer. El segundo y el tercer código dependen del primero.

Estos son los significados de los números:

- *(0,_ ,{'field': value})***:** crea un nuevo registro y lo vincula.
- *(1,id,{'field': value}):* actualiza los valores de un registro ya vinculado.
- *(2,id,_):* desvincula y elimina el registro.
- *(3,id,_)*: desvincula, pero no elimina el registro de la relación.
- *(4,id,_):* vincula un registro que ya existe.
- *(5,_,_):* desvincula, pero no elimina todos los registros.
- *(6,_,[ids]):* reemplaza la lista de registros.

Versión 15 y superiores

El proceso es similar al anterior, pero en este caso se utilizan funciones internas de la clase *Command*.

```
self.write({'sessions':[(fields.Command.link(s.id))]})
self.write({'sessions':[(fields.Command.create({ date: '2023-22-12', hash:
'adehf6734ee4845'}))]
self.write({'sessions':[(fields.Command.set(
                         [ref('vehicle_tag_leasing'),
                          ref('fleet.vehicle_tag_compact'),
                          ref('fleet.vehicle_tag_senior')])}]
```

`browse()`

A partir de una lista de *ids*, extrae un *recordset*. No se usa mucho actualmente, aunque en ocasiones es más fácil trabajar sólo con las “ids” y luego volver a buscar los “recordsets”.

```
>>> self.browse([7, 18, 12])
res.partner(7, 18, 12)
```

`exists()`

Retorna si un registro existe todavía en la base de datos.

`ref()`

A partir de un *External ID*, retorna el *recordset* correspondiente.

```
>>> env.ref('base.group_public')
res.groups(2)
```

Todos los registros de todos los modelos que tiene *Odoo* pueden tener una *External ID*. Esta es una cadena de caracteres que lo identifica independientemente del modelo al que pertenezca. *Odoo* tiene una tabla en la base de datos que relaciona los *External ID* con los *ids* reales de cada registro. De esta manera, podemos llamar a un registro con un nombre fácil de recordar y no preocuparnos de que cambie el *id* auto-numérico. Veremos más de *External ID* en el documento de ficheros de datos.

`ensure_one()`

Se asegura que un *recordset* es en realidad un singleton. Por ejemplo:

```
    @api.multi
     def get_classroom_by_course_id(self, course):
         self.ensure_one()
         return self.classrooms_ids.filtered(lambda t: t.course_id.id ==
course.id)['classroom_id']
```

`unlink()`

Borra de la base de datos un registro.

```
    @api.multi
     def unlink(self):
         for x in self:
             x.catid.unlink()
         return super(product_uom_class, self).unlink()
```

En el ejemplo anterior, sobreescribimos el método `unlink` para borrar en cascada.

`ids`

Se trata de un atributo de los *recordsets* que tiene una lista de las *ids* de los registros del *recordset*.

`copy()`

Retorna una copia del *recordset* actual.

5 DOMINIO DE BÚSQUEDA

Un dominio de búsqueda es una expresión que se utiliza para restringir los resultados de una búsqueda. Puede ser utilizado por cualquier vista, incluyendo las vistas de árbol, de formulario o los informes de los que hablaremos más adelante.

Los dominios se definen mediante tuplas de tres elementos, en el formato `(nombre de campo, operador, valor)`, donde:

- *nombre_del_campo*: es una cadena con valor el nombre de uno de los campos del objeto modelo, aunque puede haber referencias a campos relacionados, por ejemplo: *modulo_id.nombre*
- *operador*: es una cadena con un operador de comparación válido: *=, !=, >, >=, <, <=, like, ilike, in, not in*.
- *valor*: un valor válido para comparar con el campo *nombre_del_campo*.

Estos dominio pueden combinarse para crear dominios más complejos, mediante el uso de 3 operadores lógicos:

- &, and, es un operador binario y, en el caso de no estar indicado ningún operador es el operador por defecto
- |, or, es un operador binario
- !, not, es un operador unario

La combinación de los elementos se realiza insertando las tuplas de los dominios junto con los operadores en una lista utilizando notación polaca.

Por ejemplo, para una representación SQL de:

```
name = `SGE` AND ((adult = True AND course = `2`) OR (adult = False AND course = `1`))
```

quedaría:

```
[`&`,(`name`,`=`,`SGE`),`|`,`&`,(`adult`,`=`,`True`),(`course`=`2`)),`&`,(`adult`,`=`,False),(course`,`=`,`1`)]
```

6 ONCHANGE

En los formularios existe la posibilidad de que se ejecute un método cuando se cambia el valor de un *field*. Su uso habitual suele ser para cambiar el valor de otros *fields* o avisar al usuario de que se ha equivocado en algo. Para utilizarlo usaremos el decorador `@api.onchange`.

☑ *onchange* tiene implicaciones en la vista y el controlador. Todo el código se escribe en *python* cuando se define el modelo, pero *Odoo* hace que el framework de Javascript asocie un *action* (que pide al servidor ejecutar el *onchange)* al hecho de modificar un *field* y se espera al resultado de la función para modificar otros *fields* o avisar al usuario.

```
@api.onchange('amount', 'unit_price')
def _onchange_price(self):
    # set auto-changing field
    self.price = self.amount * self.unit_price
    # Can optionally return a warning and domains
    return {
        'warning': {
            'title': "Something bad happened",
            'message': "It was very bad indeed",
            'type': 'notification',
        }
    }
```

En el ejemplo se cambia el valor del *field* precio y se retorna un *warning*. Es solo un ejemplo, sin embargo observa como tiene **'***type':'notification***'** para que se muestre en ese tipo de notificación. Si no se indica esto, se mostraría como un diálogo.

```
@api.onchange('seats', 'attendee_ids')
def _verify_valid_seats(self):
     if self.seats < 0:
         return {
             'warning': {
                 'title': "Incorrect 'seats' value",
                 'message': "The number of available seats may not be negative",
             },          }
     if self.seats < len(self.attendee_ids):
          return {
             'warning': {
                 'title': "Too many attendees",
                 'message': "Increase seats or remove excess attendees",
             },
         }
```

En este segundo ejemplo no se cambia nada sino que se comprueba la cantidad de asientos y retorna un error si no hay suficientes o si el usuario se ha equivocado con el número.

```
@api.onchange('pais')
def _filter_empleado(self):
      return { 'domain': {'empleado': [('country','=', self.pais.id)]} }
```

Por último, en este tercer ejemplo lo que retorna es un *domain*. Esto provoca que el *field* empleado tenga un filtro definido en tiempo de edición del formulario.

Llegados a este punto es habitual que se genere confusión entre tres funcionalidades comentadas: los *constraints* (en sus diferentes formas), el *onchange* y el *depends.*

1. *Constraints* evita que la base de datos sea modificada, pero no puede modificar ningún tipo de dato. Actúan sobre un *singleton.*
2. *onchange* permite el cambio de un campo y la notificación de un error, pero siempre de un campo de la misma vista (o modelo). El mecanismo se acciona cuando cambiamos otro (u otros) valores del modelo. Actúan sobre un *singleton*.
3. *depends* únicamente cambia el valor de un campo cuando se modifique el valor (o valores) de otros campos del sistema (no sólo del mismo modelo). El mecanismo se activa cuando intentamos acceder al campo (por ejemplo para renderizarlo). Por esa razón tiene que incluir lógica para cualquier situación que se pueda dar, por ejemplo para cuando el registro se muestra por primera vez. Actúan sobre un *recordset*, por lo que es necesario iterar a través de él para aplicar las modificaciones.

7 BIBLIOGRAFIA

1. Documentación de Odoo
2. odoo-master

8 AUTORES

A continuación ofrecemos en orden alfabético (por apellido) el listado de autores que han hecho aportaciones a este documento.

- Jose Castillo Aliaga
- Sergi García Barea
- Alfredo Oltra