

Sistemas de Gestión Empresarial

UD 07. Desarrollo de módulos de Odoo: Controlador, Herencia, Informes, Wizards.

Actualizado Agosto 2021


Licencia





Reconocimiento – NoComercial – CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

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

 Importante

 Atención

 Interesante

ÍNDICE DE CONTENIDO


Controlador	3
Enviroment	5
Métodos del ORM	5
Onchange	8
Ficheros de datos	9
Reports	11
Herencia	12
Wizards	14
Bibliografía	20
Autores (en orden alfabético)	21

UD07. DESARROLLO DE MÓDULOS DE ODOO


1. CONTROLADOR

Odoo tiene una arquitectura MVC y permite desarrollar cada parte por separado. No obstante, el controlador se suele desarrollar en los mismos ficheros Python en los que se hace el modelo. El controlador son los métodos que hay en los modelos.

Ya 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.

 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.

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)
```

Para acceder a los datos, hay que ir a los singletons, no a los recordsets.

```
>>> a
```

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

Intentar acceder a los datos en un recordset da error. Los datos de los fields relacionales dan un recordset:

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

Los recordsets son iterables como un array y tienen, además, unas operaciones de conjuntos para combinarlos:

- **record in set** retorna si el record está en el set
- **set1 | set2** Unión de sets. También funciona el signo +
- **set1 & set2** Intersección de sets
- **set1 - set2** Diferencia de 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.

```
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.

```
# 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.

```
# returns a list of summing two fields for each record in the set
records.mapped(lambda r: r.field1 + r.field2)
# returns a list of names
records.mapped('name')
# returns a recordset of partners
record.mapped('partner_id')
# returns the union of all partner banks, with duplicates removed
```

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

1.1 Enviroment

El llamado enviroment 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 algunas metadatos).

Todos los Recordset tienen un enviroment accesible con 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 enviroment encontramos al Context. Se trata de 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 por lo menos contiene el user ID, el idioma y la zona horaria.

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

El Context ya lo hemos usado o lo usaremos en esta unidad didáctica. Es conveniente repasar los usos comunes que Odoo tiene por defecto para ver su importancia:

- **active_id** : Cuando en una vista queremos que los fields One2many abran un formulario con el field Many2one por defecto le pasamos 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 si queremos 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')**
- En la vista search también guardamos el criterio de agrupación con el **group_by** en el **context**.

1.2 Métodos del ORM

Veamos ahora uno por uno los métodos que proporciona el ORM de Odoo para facilitar la gestión de los recordset:

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 igual, pero sólo retorna la cantidad de registros encontrados.

Estos son los parámetros que acepta search:

- args -- Un dominio de búsqueda. Se puede dejar como [] para encontrarlos a todos.
- offset (int) -- Número de resultados a ignorar. Se puede combinar con limit si queremos 'paginar' el search
- limit (int) -- Número máximo de resultados a extraer.
- order (str) -- String de ordenación como en SQL (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': "New Name"})
res.partner(78)
```

write():

Escribe sobre el recordset desde el cual se invoca

```
self.write({'name': "Newer Name"})
```

Hay un caso especial en el **write()** cuando se intenta escribir en un **Many2many**. Lo normal es pasar una lista de ids. Pero si se va a escribir en un Many2many que ya tiene elementos se deben usar unos códigos especiales:

```
self.write({'sessions': [(4,s.id)]})
self.write({'sessions': [(6,0,
                        [ref('vehicle_tag_leasing'),
                          ref('fleet.vehicle_tag_compact'),
                          ref('fleet.vehicle_tag_senior')] )]]})
```

Como se ve, 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 tercero 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.

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. Esto 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 id 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 autonumérico. Veremos más de External ID en el apartado de ficheros de datos.

ensure_one():

Se asegura que un recordset es en realidad un singleton.

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.

1.3 Onchange

En los formularios existe la posibilidad de que se ejecute un método cuando se cambia el valor de un field. Suele ser para cambiar el valor de otros fields o avisar al usuario de que se ha equivocado en algo. Para ello está 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' al hecho de modificar un field que pide al servidor ejecutar el 'onchange' y se espera al resultado de la función para modificar fields o avisar al usuario. Por tanto, tampoco sería muy raro que se explicara como mejora de los formularios en el apartado de la vista.

Por otro lado, los fields computed que tienen el '@api.depends' tienen un comportamiento similar al onchange cuando cambia el field del cual depende.

Veamos primero algunos ejemplos:

```
@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',
        }
    }

@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",
            },
        }

@api.onchange('pais')
```



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

En el primer ejemplo cambia el valor del field precio y retorna un warning. Es sólo un ejemplo, pero observa como tiene 'type': 'notification' para que se muestre en ese tipo de notificación de no indicarlo se muestra como un diálogo.

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

En el tercer ejemplo lo que retorna es un **domain**. Esto provoca que el field Many2one al que afecta tenga un filtro definido en tiempo de edición del formulario.

Si el usuario se equivoca hay tres maneras de tratar con ese error: Constraints, onchange y sobrescribir el método write y create para comprobar que no hay errores. En último caso no debería usarse si no se puede hacer con una constraint. Las constraints y onchange se complementan bien. Por un lado previenes el error del usuario y por el otro lo previenes realmente antes de guardarlo en la base de datos.

2. FICHEROS DE DATOS

Ya hemos usado ficheros de datos creando las vistas. Si nos fijamos, veremos que es un XML con una etiqueta <odoo>, otra <data> y dentro los <record> de cada vista. Así es como le decimos a Odoo lo que se debe guardar en la base de datos.

```
<odoo>
  <data>
    <record model="{model name}" id="{record identifier}">
      <field name="{a field name}">{a value}</field>
    </record>
  </data>
</odoo>
```

En el modelo ponemos el nombre del modelo en el que guardará. En id el External ID y luego cada uno de los fields a los que queremos darle valor.

En este punto, es necesario detenerse para investigar los External ID en Odoo. La primera consideración a tener en cuenta es que estamos utilizando un ORM que transforma las declaraciones de clases que heredan de models.Model en tablas de PostgreSQL y los Records declarados en XML en registros de esas tablas. Todos los registros del ORM tienen una columna id que los identifica de forma unívoca en su tabla. Esto permite que, durante la ejecución del programa, funcionen las claves ajenas entre modelos. Esto no tiene ninguna diferencia respecto al modelo tradicional sin ORM. El problema al que se enfrentan los programadores de Odoo es que hay que crear ficheros de datos XML en los que se definen relaciones entre modelos antes de instalar el módulo. Estas relaciones no se pueden referir al id porque es un código autonumérico que no es predecible en el momento de programar. Para solucionarlo se inventa el ID externo o External ID.

Este identificador está escrito en lenguaje humano y ha de ser distinto a cualquier identificador del programa. Para garantizar eso se recomienda poner el nombre del módulo, un punto y un nombre que identifique la utilidad y significado del registro. Hay que tener en cuenta que todos los elementos de Odoo pueden tener un

identificador externo: módulos, modelos, vistas, acciones, menús, registros, fields... Por eso hay que establecer unas reglas. Por ejemplo: `school.teacher_view_form` serviría para el formulario que muestra a los profesores del módulo `school`.

Podemos buscar los identificadores externos directamente en el modo desarrollador de Odoo en el apartado de ajustes > técnico > identificadores externos.

Cuando se hacen los ficheros de datos, los fields simples son muy sencillos de rellenar. Los Binary y Image tienen que estar en **Base64**, pero eso se puede conseguir fácilmente con el comando `base64` de GNU/Linux.

Lo más complicado son los field relacionales. Para conseguirlo hay que utilizar los identificadores externos, ya que no es recomendado en ningún caso usar el campo `id`. En realidad se guardará el `id` en la base de datos, pero después de evaluar el External ID.

Para rellenar un Many2one hay que usar `ref()`:

```
<field name="product_id" ref="product.product1"/>
```

En ocasiones queremos que el valor sea calculado con Python durante el momento de la instalación del módulo. Para ello usamos `eval()`:

```
<field name="date" eval="(datetime.now()+timedelta(-1)).strftime('%Y-%m-%d')"/>
<field name="product_id" eval="ref('product.product1')"/> # Equivalente al ejemplo anterior
<field name="price" eval="ref('product.product1').price"/>
```

Para los x2many se deben usar `eval()` con `ref()` y una tripleta que indica lo que hay que hacer:

```
<field name="tag_ids"
eval="[(6,0,[ref('fleet.vehicle_tag_leasing'),ref('fleet.vehicle_tag_compact'),
ref('fleet.vehicle_tag_senior')])]"/>
```

Esas tripletas tienen los siguientes significados:

- `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.

También se puede usar para borrar registros:

```
<delete model="cine.session" id="session_cine1_1"></delete>
```

3. REPORTS

Es muy probable necesitar imprimir algunos documentos a partir de Odoo o simplemente enviarlos en PDF por correo. Para ello están los reports. Todos los ERP tienen un sistema de extracción de

documentos y muchos lo hacen al menos en PDF, como es el caso de Odoo.

El formato PDF tiene sus peculiaridades y es complicado manejarlo directamente como se hace con HTML. Por eso Odoo confía en un renderizador de HTML a PDF que utiliza el motor de WebKit (que es uno de los motores de renderizado libres más populares). Para ello hace una llamada al sistema para que ejecute wkhtmltopdf que es un programa que transforma por terminal un HTML en PDF. Es necesario, por tanto, instalarlo en el sistema.

Generalmente un report es llamado con una acción desde el cliente web. Esta acción es de tipo **ir.actions.report**.

Esta acción de tipo report necesita una plantilla hecha con QWeb para interpretarla, transformarla en HTML y luego invocar a wkhtmltopdf para que lo transforme en PDF.

... Tanto el action como la plantilla se guardan en la base de datos en records, pero los dos tienen atajos para no escribir la etiqueta <record>, así que los usaremos, estos son <report> para el action y <template> para la plantilla.

Veamos un ejemplo sencillo:

```
<report
  id="report_session"
  model="openacademy.session"
  string="Session Report"
  name="openacademy.report_session_view"
  file="openacademy.report_session"
  report_type="qweb-pdf" />

<template id="report_session_view">
  <t t-call="report.html_container">
    <t t-foreach="docs" t-as="doc">
      <t t-call="report.external_layout">
        <div class="page">
          <h2 t-field="doc.name"/>
          <p>From <span t-field="doc.start_date"/> to <span
t-field="doc.end_date"/></p>
          <h3>Attendees:</h3>
          <ul>
            <t t-foreach="doc.attendee_ids" t-as="attendee">
              <li><span t-field="attendee.name"/></li>
            </t>
          </ul>
        </div>
      </t>
    </t>
  </t>
</template>
```

Como se puede ver, QWeb tiene una variable llamada docs, que es la lista de registros a mostrar en el informe.

4. HERENCIA

Odoo proporciona mecanismos de herencia para las tres partes del MVC.

En el caso de la herencia en el modelo, el ORM permite 3 tipos: De clase, por prototipo y por delegación:

De Clase	<p>Herencia simple.</p> <p>La clase original queda ampliada por la nueva clase.</p> <p>Añade nuevas funcionalidades (atributos y / o métodos) en la clase original.</p> <p>Las vistas definidas sobre la clase original continúan funcionando.</p> <p>Permite sobrescribir métodos de la clase original.</p> <p>En PostgreSQL, continúa mapeada en la misma tabla que la clase original, ampliada con los nuevos atributos que pueda incorporar.</p>	<p>Se utiliza el atributo <code>_inherit</code> en la definición de la nueva clase Python:</p> <pre><code>_inherit = obj</code></pre> <p>El nombre de la nueva clase debe seguir siendo el mismo que el de la clase original: <code>_name = obj</code></p>
Por Prototipo	<p>Herencia simple.</p> <p>Aprovecha la definición de la clase original (como si fuera un «prototipo»).</p> <p>La clase original continúa existiendo.</p> <p>Añade nuevas funcionalidades (atributos y / o métodos) a las aportadas por la clase original.</p> <p>Las vistas definidas sobre la clase original no existen (hay que diseñar de nuevo).</p> <p>Permite sobrescribir métodos de la clase original.</p> <p>En PostgreSQL, queda mapeada en una nueva tabla.</p>	<p>Se utiliza el atributo <code>_inherit</code> en la definición de la nueva clase Python:</p> <pre><code>_inherit = obj</code></pre> <p>Hay que indicar el nombre de la nueva clase: <code>_name = nuevo_nombre</code></p>
Por Delegación	<p>Herencia simple o múltiple.</p> <p>La nueva clase «delega» ciertos funcionamientos a otras clases que incorpora en su interior.</p> <p>Los recursos de la nueva clase contienen un recurso de cada clase de la que derivan.</p> <p>Las clases base continúan existiendo.</p> <p>Añadir las funcionalidades propias (atributos y / o métodos) que corresponda.</p> <p>Las vistas definidas sobre las clases bases no existen en la nueva clase.</p> <p>En PostgreSQL, queda mapeada en diferentes tablas: una tabla para los atributos propios, mientras que los recursos de las clases derivadas residen en las tablas correspondientes a dichas clases.</p>	<p>Se utiliza el atributo <code>_inherits</code> en la definición de la nueva clase Python:</p> <pre><code>_inherits = ...</code></pre> <p>Hay que indicar el nombre de la nueva clase: <code>_name = nuevo_nombre</code></p>

💬 Veamos cuándo hay que usar cada tipo de herencia:

Odoo es un programa que ya existe, por tanto, a la hora de programar es diferente a cuando lo hacemos desde 0 aunque usemos un framework. Por lo general no necesitamos crear cosas totalmente nuevas, tan solo ampliar algunas funcionalidades de Odoo. Por tanto, la herencia más utilizada es la herencia de clase. Esta amplía una clase existente, pero esta clase sigue funcionando como antes y todas las vistas y relaciones permanecen.

Usaremos la herencia por prototipo cuando queremos hacer algo más parecido a la herencia de los lenguajes de programación. Esta no modifica el original, pero obliga a crear las vistas y las relaciones desde cero.

La herencia por delegación sirve para aprovechar los fields y funciones de otros modelos en los nuestros. Cuando creamos un registro de un modelo heredado de esta manera, se crea también en el modelo padre un registro al que está relacionado con un Many2one. El funcionamiento es parecido a poner manualmente ese Many2one y todos los fields related. Un ejemplo fácil de entender es el caso entre `product.template` y `product.product` que hereda de este. Con esta estructura, se puede hacer un producto base y luego con `product.product` se puede hacer un producto para cada talla y color, por ejemplo.

Veamos un ejemplo de cada tipo de herencia:

```
class res_partner(models.Model): # De clase
    _name = 'res.partner'
    _inherit = 'res.partner'
    debit_limit = fields.Float('Payable limit')
...

class res_alarm(model.Model): # Clase padre
    _name = 'res.alarm'
...

class calendar_alarm(model.Model): # Por prototipo
    _name = 'calendar.alarm'
    _inherit = 'res.alarm'
...

class calendar_alarm(model.Model): # Por delegación
    _name = 'calendar.alarm'
    _inherits = {'res.alarm': 'alarm_id'}
...
```

Si se hace herencia de clase y se añaden fields que queremos ver, hay que ampliar la vista existente. Para ello usaremos un record en XML con una sintaxis especial. Lo primero es añadir esta etiqueta:

```
<field name="inherit_id" ref="modulo.id_xml_vista_padre"/>
```

Después, en el `<arch>` no hay que declarar una vista completa, sino una etiqueta que ya exista en la vista padre y qué hacer con esa etiqueta. Lo que se puede hacer es:

- `inside` (por defecto): los valores se añaden "dentro" de la etiqueta.
- `after`: añade el contenido después de la etiqueta.
- `before`: añade el contenido antes de la etiqueta.

- replace: reemplaza el contenido de la etiqueta.
- attributes: Modifica los atributos.

Veamos algunos ejemplos:

```
<field name="arch" type="xml">
  <data>
    <field name="campo1" position="after">
      <field name="nuevo_campo1"/>
    </field>
    <field name="campo2" position="replace"/>
    <field name="campo3" position="before">
      <field name="nuevo_campo3"/>
    </field>
  </data>
</field>

<xpath expr="//field[@name='order_line']/tree/field[@name='price_unit'" position="before">
  <xpath expr="//form/*" position="before">
    <header>
      <field name="status" widget="statusbar"/>
    </header>
  </xpath>
```

Como se puede ver en el ejemplo, se puede usar la etiqueta <xpath> para encontrar etiquetas más difíciles o que estén repetidas.

Es posible que tengamos una herencia de clase en el modelo pero no queramos usar nada de la vista original en otro menú. Para ello podemos especificar para cada action las vistas a las que está asociado. Odoo cuando va a mostrar algo que dice un action busca la vista que le corresponde. En caso de no encontrarla busca la vista de ese modelo con más prioridad. Por tanto, dentro del action:

```
<field name="view_ids" eval="[(5, 0, 0),(0, 0, {'view_mode': 'tree', 'view_id':
ref('tree_external_id')}),(0, 0, {'view_mode': 'form', 'view_id': ref('form_external_id')}),]" />
```

Esto generará en la tabla intermedia del Many2many con las vistas las relaciones que consultará Odoo antes de elegir una vista por prioridad.

5. WIZARDS

Un wizard es un asistente que nos ayuda paso a paso a realizar alguna gestión en Odoo. Los formularios son suficientes para introducir datos, pero en ocasiones pueden ser poco intuitivos o farragosos.

En realidad, un wizard no utiliza ninguna tecnología específica que no utilicen otras partes de Odoo. Se trata de un formulario mostrado generalmente en una ventana modal por encima de la ventana principal. Los datos de ese formulario no son permanentes en la base de datos, ya que sólo es una

ayuda para, finalmente, modificarla cuando acabemos el asistente.

Para hacer esos datos no persistentes se usa un tipo de modelo llamado **Transientmodel**. Este se guarda temporalmente en la base de datos y es accesible sólo durante la ejecución del wizard.

De los Wizard, lo más nuevo son los TransientModel. Pero este apartado nos servirá para hacer un recopilatorio de las técnicas vistas en estas unidades como son los actions, onchange, buttons, context... Además, los estudiaremos desde otro punto de vista y de formas más avanzadas en algunos casos.

TransientModel tiene las siguientes características y limitaciones:

- No es permanente en la base de datos y no tenemos que preocuparnos de borrar los registros temporales.
- No necesitamos permisos explícitos para poder acceder a estos modelos.
- Pueden tener Many2one con modelos permanentes pero no al contrario.

Veamos un ejemplo completo:

```
class course_wizard(models.TransientModel):
    _name = 'school.course_wizard'
    state = fields.Selection([('1', 'Course'),
                              ('2', 'Classrooms'), ('3', 'Students'), ('4', 'Enrollment')], default='1')
    name = fields.Char()
    c_name = fields.Char(string='Classroom Name')
    c_level = fields.Selection([('1', '1'), ('2', '2')], string='Classroom Level')
    classrooms = fields.Many2many('school.classroom_aux')
    s_name = fields.Char(string='Student Name')
    s_birth_year = fields.Integer(string='Student Birth Year')
    s_dni = fields.Char(string='DNI')
    students = fields.Many2many('school.student_aux')

    @api.model
    def action_course_wizard(self):
        action = self.env.ref('school.action_course_wizard').read()[0]
        return action

    def next(self):
        if self.state == '1':
            self.state = '2'
        elif self.state == '2':
            self.state = '3'
        elif self.state == '3':
            self.state = '4'
        return {
            'type': 'ir.actions.act_window',
            'res_model': self._name,
            'res_id': self.id,
            'view_mode': 'form',
            'target': 'new',
        }
```

```
def previous(self):
    if self.state == '2':
        self.state = '1'
    elif self.state == '3':
        self.state = '2'
    elif self.state == '4':
        self.state = '3'
    return {
        'type': 'ir.actions.act_window',
        'res_model': self._name,
        'res_id': self.id,
        'view_mode': 'form',
        'target': 'new',
    }

def add_classroom(self):
    for c in self:
        c.write({'classrooms':[(0,0,{'name':c.c_name,'level':c.c_level})]})
    return {
        'type': 'ir.actions.act_window',
        'res_model': self._name,
        'res_id': self.id,
        'view_mode': 'form',
        'target': 'new',
    }

def add_student(self):
    for c in self:
        c.write({'students':[(0,0,{'name':c.s_name,'dni':c.s_dni,
                                   'birth_year':c.s_birth_year})]})
    return {
        'type': 'ir.actions.act_window',
        'res_model': self._name,
        'res_id': self.id,
        'view_mode': 'form',
        'target': 'new',
    }

def commit(self):
    return {
        'type': 'ir.actions.act_window',
        'res_model': self._name,
        'res_id': self.id,
        'view_mode': 'form',
        'target': 'new',
    }
```



```

def create_course(self):
    for c in self:
        curs = c.env['school.course'].create({'name': c.name})
        students = []
        for cl in c.classrooms:
            classroom = c.env['school.classroom'].create({'name': cl.name, 'course': curs.id,
                                                            'level': cl.level})
            for st in cl.students:
                student = c.env['res.partner'].create({'name': st.name,
                                                         'dni': st.dni,
                                                         'birth_year': st.birth_year,
                                                         'is_student': True,
                                                         'classroom': classroom.id
                                                         })

                students.append(student.id)
            curs.write({'students': [(6, 0, students)]})

    return {
        'type': 'ir.actions.act_window',
        'res_model': 'school.course',
        'res_id': curs.id,
        'view_mode': 'form',
        'target': 'current',
    }

class classroom_aux(models.TransientModel):
    _name = 'school.classroom_aux'
    name = fields.Char()
    level = fields.Selection([('1', '1'), ('2', '2')])
    students = fields.One2many('school.student_aux', 'classroom')

class student_aux(models.TransientModel):
    _name = 'school.student_aux'
    name = fields.Char()
    birth_year = fields.Integer()
    dni = fields.Char(string='DNI')
    classroom = fields.Many2one('school.classroom_aux')

```

En este código tan largo hay que observar algunas cosas:

Hemos declarado 3 transientmodels que se parezcan a sus equivalentes en modelos normales. Esto nos permite hacer relaciones como Many2many dentro del wizard.

Hay un field state que servirá para ver en un widget tipo statusbar el progreso del wizard. Hay

botones para ir adelante y atrás en el mismo. Estos botones retornan actions que refrescan el mismo modelo e id para que no se cierre la ventana.

El asistente muestra un formulario genérico para crear alumnos y cursos e ir agregandolos a una lista.

Finalmente, pasa todos los datos de los modelos transitorios a los modelos permanentes cuando pulsamos el botón de finalizar.

```
<record model="ir.ui.view" id="school.course_wizard_form">
  <field name="name">course wizard form</field>
  <field name="model">school.course_wizard</field>
  <field name="arch" type="xml">
    <form>
      <header>
        <button name="previous" type="object"
          string="Previous" class="btn btn-secondary" states="2,3,4"/>
        <button name="next" type="object"
          string="Next" class="btn oe_highlight" states="1,2,3"/>
        <field name="state" widget="statusbar"/>
      </header>
    </sheet>

    <group states="1,2,3,4">
      <field name="name" attrs="{ 'readonly': [ ('state', '!=', '1') ] }"/>
    </group>
    <group col="5" string="Classrooms" states="2">
      <field name="c_name"/>
      <field name="c_level"/>
      <button name="add_classroom" type="object"
        string="Add Classroom" class="oe_highlight"></button>
    </group>
    <group states="2">
      <field name="classrooms">
        <tree editable="bottom">
          <field name="name"/>
          <field name="level"/>
        </tree>
      </field>
    </group>
    <group col="7" string="Students" states="3">
      <field name="s_name"/>
      <field name="s_birth_year"/>
      <field name="s_dni"/>
      <button name="add_student" type="object"
        string="Add Student" class="oe_highlight"></button>
```

```

        </group>
        <group states="3">
        <field name="students" />
        </group>
        <group states="4" string="All Students">
        <field name="students" >
            <tree editable="bottom">
            <field name="name"/>
            <field name="birth_year"/>
            <field name="dni"/>
            <field name="classroom"/>
            </tree>
        </field>
        </group>

        <group states="4" string="Classrooms">
        <field name="classrooms" >
            <tree editable="bottom">
            <field name="name"/>
            <field name="students" widget="many2many_tags"/>
            </tree>
        </field>
        </group>
        <button name="commit" type="object"
            string="Commit" class="oe_highlight" states="4"/>
        <footer>
        <button name="create_course" type="object"
            string="Create" class="oe_highlight" states="4"/>
        <button special="cancel" string="Cancel"/>
        </footer>
    </sheet>
</form>
</field>
</record>

<record id="school.action_course_wizard" model="ir.actions.act_window">
<field name="name">Launch course wizard</field>
<field name="type">ir.actions.act_window</field>
<field name="res_model">school.course_wizard</field>
<field name="view_mode">form</field>
<field name="view_id" ref="school.course_wizard_form" />
<field name="target">new</field>
</record>

```

En esta vista es interesante observar cómo los distintos apartados son escondidos o mostrados en función del field state.

Este wizard se podría llamar con un menú asociado con ese action, con un botón que activará ese action o con un enlace en un elemento de la web que lo llame manualmente.

Desde un menú:

```
<menuitem
name="Course Wizard"
id="school.menu_course_wizard"
parent="school.menu_courses"
action="school.action_course_wizard"/>
```

Desde un botón:

```
<button
type="action"
name="%(school.action_course_wizard)d"
string="Course Wizard"></button>
```

Desde un elemento HTML generado por un web controller (no explicados en este documento):

```
class MyController(http.Controller):
    @http.route('/school/course/', auth='user', type='json')
    def course(self):
        return {
            'html': """
                <div id="school_banner">
                    <link href="/school/static/src/css/banner.css"
                        rel="stylesheet">
                    <h1>Curs</h1>
                    <p>Creación de cursos:</p>
                    <a class="course_button" type="action" data-reload-on-close="true"
role="button" data-method="action_course_wizard" data-model="school.course_wizard">
                        Crear Curs
                    </a>
                </div> """
        }
```

6. BIBLIOGRAFÍA

<https://www.odoo.com/documentation/master/>

https://ioc.xtec.cat/materials/FP/Materials/2252_DAM/DAM_2252_M10/web/html/index.html

<https://castilloinformatica.es/wiki/index.php?title=Odoo>

7. AUTORES (EN ORDEN ALFABÉTICO)

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

- Jose Castillo Aliaga
- Sergi García Barea