

Sistemas de Gestión Empresarial

UD 07. Desarrollo de módulos de Odoo: Controlador, Herencia y Web Controllers.

Actualizado Diciembre 2024

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

1. Controlador	2
1.1 Enviroment	4
1.2 Métodos del ORM	5
1.3 Onchange	7
2. Ficheros de datos	9
3. Reports (Informes)	10
4. Herencia	11
5. Wizards (Asistentes)	14
6. Web controllers	21
6.1 ¿Qué son los Web Controllers?	22
6.2 Pasar parámetros al Web Controller	24
6.2.1 Parámetros por el POST o GET:	24
6.2.2 Parámetros por REST:	25
6.2.3 Parámetros por JSON:	25
6.2.4 Cors con Odoo	26
6.2.5 Autenticación	26
6.2.6 Hacer una API REST	26
6.2.7 Comunicar una SPA Vue/React/Angular con Odoo	28
7. Dependencias externas	29
8. Módulos de ejemplo con comentarios	29
9. Bibliografía	29
10. Autores (en orden alfabético)	29

UD07. DESARROLLO DE MÓDULOS DE ODOO: CONTROLADOR, HERENCIA, WEB CONTROLLERS.

1. CONTROLADOR

Odoo tiene una arquitectura MVC (Modelo-Vista-Controlador) que 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.

Para clarificar, podríamos decir que el controlador son los métodos que hay en los modelos.

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).

! Atención: 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.

Interesante: 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)
```

Recordamos que para acceder a los datos, hay que ir a los “singletons” y 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
```

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 (sets). 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” <https://docs.python.org/3/howto/sorting.html#key-functions>. Similar a “sort”.

```
# 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. Similar a “map”.

```
# 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 algunos metadatos).

Todos los “recordset” tienen un “enviroment” 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 “**enviroment**” encontramos a “**context**”. El atributo “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 al menos siempre contiene el “user ID”, el idioma y la zona horaria.

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

El “**context**” ya lo hemos usado anteriormente y 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 atributo “**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 de forma similar, pero retornando únicamente la cantidad de registros encontrados.

Estos son los parámetros que acepta “search”:

- **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': "New Name"})
res.partner(78)
```

write():

Escribe información en el “recordset” desde el cual se invoca:

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

Hay un caso especial de “**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 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.

browse():

A partir de una lista de “ids”, extrae un “recordset”. No se usa mucho actualmente, aunque en ocasiones es más fácil trabajar solo 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)
```

 **Interesante:** 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 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, sobrescribimos 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.

read():

Retorna el recordset transformado en una lista de diccionarios de Python tradicionales. Se usa a menudo para “exportar” los datos a aplicaciones que no son Odoo como cuando hacemos una REST API o, simplemente, si es más cómodo trabajar con los datos en crudo en el algoritmo correspondiente.

```
>>> a = self.env['res.partner'].search([('id','=',1)])
>>> a
res.partner(1,)
>>> a.read()
[{'id': 1, 'name': 'My Company (San Francisco)', 'display_name': 'My Company (San Francisco)',
'date': False, 'title': False, 'parent_id': False, 'parent_name': False, 'child_ids': [39,
40],....
```

1.3 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”.

🗨 **Interesante:** “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 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',
        }
    }
```

En el primer 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 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)]} }
```

En este 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.

... **Interesante:** 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 (solo debería usarse si no se puede hacer con una “constraint”). Las “constraints” y “onchange” se complementan bien: con la “constraint”: previenes el error del usuario y con el “onchange” previenes realmente antes de guardarlo en la base de datos.

Los **Onchange** pueden cambiar los valores de otro field en el formulario. En realidad no se guardan estos cambios hasta que el formulario no es guardado completamente (lo cual tiene sentido). Para conseguirlo, Onchange trabaja con un **registro virtual** que es una copia de los datos originales. Es por eso que, en un Onchange, no es fácil modificar definitivamente el registro en el que se está trabajando actualmente. De hecho, no es recomendable en absoluto hacerlo.

En caso de necesitar acceder a los datos originales (en ocasiones necesitamos el id), se puede con un atributo especial llamado **self._origin**, que es una referencia al registro original.

2. FICHEROS DE DATOS

Ya hemos utilizado ficheros de datos creando vistas. Si nos fijamos, observaremos 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 este ejemplo, en el modelo ponemos el nombre del modelo en el que guardará y en “**id**” el “**External ID**”. Tras ello, indicamos 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 auto-numérico que no es predecible en el momento de programar.

Para solucionarlo se inventó el “**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, etc. 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**”.

 **Interesante:** 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” e “Image” tienen que estar en formato **Base64**, pero eso se puede conseguir fácilmente con el comando “base64” de GNU/Linux o sitios web como <https://www.base64decode.org/>.

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 se hace 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 (INFORMES)

Es muy probable necesitar imprimir algunos documentos a partir de Odoo o simplemente enviarlos en PDF por correo. Para ello están los “reports” (informes). Todos los ERP tienen un sistema de extracción de documentos y generalmente todos lo hacen mínimo 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, haberlo instalado 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 transformarlo en PDF.

💬 **Interesante:** 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>
  </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: <code>_inherit = obj</code></p> <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: <code>_inherit = obj</code></p> <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: <code>_inherits = ...</code></p> <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” como “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(models.Model): # Clase padre
    _name = 'res.alarm'
    ...

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

class calendar_alarm(models.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="camp03" 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 de referenciar 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 le indica 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 (ASISTENTES)

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.


Un wizard en Odoo es una ventana emergente que permite al usuario completar varios pasos o procesos en una sola sesión, mientras que un formulario normal en Odoo es una vista de un solo paso que permite al usuario ingresar o editar información en un solo registro. Los wizards suelen ser utilizados para tareas complejas o procesos que involucran varios pasos, mientras que los formularios normales suelen ser utilizados para tareas más simples o para ingresar o editar información en un solo registro.

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 este solo es una ayuda para, finalmente, modificar la base de datos cuando acabemos con 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 solo durante la ejecución del “wizard”.

En Odoo, un modelo transitorio (o transient model) es un tipo especial de modelo que se utiliza para almacenar temporalmente datos que son necesarios para un proceso específico, pero que no tienen que ser guardados permanentemente en la base de datos. Los wizards a menudo se hacen con TransientModel, ya que suelen requerir almacenar temporalmente datos ingresados por el usuario mientras se completa el proceso en varios pasos. Al finalizar el proceso, los datos guardados en el modelo transitorio son eliminados, ya que ya no son necesarios.

Además, al usar un modelo transitorio en lugar de un modelo regular, se puede evitar crear registros no deseados en la base de datos y se puede mantener la base de datos limpia.

 **Interesante:** 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”, etc. 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.
- Pueden tener “Many2one” con modelos permanentes, pero no al contrario (“One2Many”).

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 largo ejemplo de código hay que observar algunas cosas:

Hemos declarado tres “TransientModels” que se parecerán 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 agregándolos 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>
    </form>
  </field>
  <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>
  </sheet>
</record>
```

```

        </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.

Por ejemplo, se podría llamar desde un menú:

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

O se podría llamar desde un botón:

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

O se podría llamar desde un elemento HTML generado por un “web controller”:

```
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> """
        }
```

Otra forma de llamarlo es desde el código Python de alguna función que atienda a una acción de tipo **object**. Para ello, ha de retornar un diccionario con los datos de la acción que se va a enviar al cliente para que la ejecute:

```
return {
    "type": "ir.actions.act_window",
    "name": "Course Wizard",
    "res_model": "school.couse_wizard",
    'view_mode': 'form',
    "target": "new"
}
```

O, si ya está la acción guardada en la base de datos, transformándola en un diccionario con **read()**:

```
return self.env.ref('school.action_course_wizard').read()[0]
```

Los wizards, como hemos explicado anteriormente, son buenos para repasar conceptos sobre los modelos y las vistas e ir un paso más allá. En concreto, plantean algunos problemas recurrentes que

tienen soluciones interesantes y no muy documentadas. Es interesante estudiarlas porque nos da una visión más profunda del funcionamiento de Odoo:

Obtener datos por contexto en el Wizard:

En ocasiones, el Wizard es invocado desde un formulario o una lista en la que ya hay algunos datos que pueden ser necesarios para el mismo. Para ello, se pueden usar las funciones del contexto en fields con default:

```
def _default_classroom(self):
    return self.env['school.classroom'].browse(self._context.get('active_id'))
classroom = fields.Many2one('school.classroom', default=_default_classroom)
```

En este caso, obtenemos el **active_id**, que siempre está cuando invocamos un **action** desde una vista. Pero si queremos otros datos, hay que enviarlos:

```
<button name="%(school.action_course_wizard)d"
        type="action" string="Create Course"
        context="{ 'teacher_context': parent.id, 'classroom_context': active_id}" />
```

De paso, vemos cómo enviar el **active_id** del formulario (**parent.id**) que contiene un tree, donde pondremos este botón.

OnChange en wizards:

En principio, los Onchange funcionan igual. Pueden asignar valores a otros fields, filtrar otros fields o enviar mensajes de error. Pero hay un problema y es que, si pulsamos el típico botón de siguiente, esos datos del registro virtual se refrescan con los del original y se pierden. Por tanto, debemos hacer uso del atributo **self._origin** si queremos que esos cambios pasen al siguiente estado del wizard.

```
@api.onchange('destiny')
def _onchange_destiny(self):
    if len(self.destiny)>0:
        road_available = self.origin.roads & self.destiny.roads
        self._origin.write({'road': road_available.id})
        self.road = road_available.id
        return {}
```

El problema de los domains es más complejo, ya que hay que enviarlos por contexto al siguiente paso. Para ello, hay que manipular el context original (que es un diccionario inmutable) añadiendo otros atributos que puedan ser aprovechados en un field del siguiente paso:

```
# En el action del return de next:
'context': dict(self._context, cities_available_context= (self.cities_available.city).ids,
origin_context = self.origin.id),
<!-- En la vista: -->
<field name="destiny" domain = "[('id','in',context.get('cities_available_context',[]))]" />
```

La función **context.get** es una herramienta que proporcionan las plantillas QWeb para consultar el contexto.

6. WEB CONTROLLERS

Odoo tiene, de forma oficial, 3 clientes web distintos: El “backend”, el TPV (Terminal Punto de Venta) y el “frontend”. Los tres funcionan de forma independiente y con algunas diferencias. Para la gestión de la empresa normalmente es suficiente con el “backend”, pero para el TPV o la página web no siempre es fácil o conveniente usar la solución de Odoo.

Odoo proporciona una manera de conectarse a su servidor, mediante el uso de XML-RPC, la cual es bastante simple y funciona bien cuando se está conectando alguna aplicación hecha con PHP, Python o Java, por ejemplo. De hecho, en la documentación oficial hay ejemplos para estos 3 lenguajes. https://www.odoo.com/documentation/17.0/developer/reference/external_api.html

No obstante, lenguajes como Javascript tienen problemas con este tipo de conexión. No es imposible conectar Javascript mediante XML-RPC y existen bibliotecas para hacerlo, sin embargo, Javascript y los frameworks del mismo están mucho más preparados para consultar vía API REST, por ejemplo, que para utilizar XML-RPC.

Por otro lado, los formularios, listas o Kanban de Odoo se pueden quedar limitados si queremos mostrar los datos de alguna manera determinada.

Para ello existen varias soluciones, pero normalmente optaremos por hacer un nuevo Componente (o “widget”, en versiones anteriores a la 14) o por insertar un fragmento de HTML obtenido por un Web Controller (caso tratado en este apartado).

Como vemos, hay muchas opciones y casi siempre se pueden hacer las cosas de muchas formas. Además, la documentación oficial de Odoo a este respecto es muy escueta y en ocasiones obsoleta.

Es por eso que elegir no siempre es fácil. Reflexionemos sobre qué podemos hacer y cómo:

- **Queremos un cambio menor en la apariencia:** añadir reglas CSS nuevas en el directorio static.
- **Una nueva forma de visualizar o editar los datos:** crear un Componente o Widget.
- **Una nueva forma de visualizar o editar un registro entero:** crear una vista.
- **Comunicar una página web hecha con PHP con Odoo:** XML-RPC.
- **Hacer una web:** en general se recomienda usar el módulo de web de Odoo y modificarla.
- **Hacer una app con Java que se conecta al servidor Odoo:** usar XML-RPC o hacer una API REST con los Web Controllers.
- **Hacer una web estática desde cero:** usar los Web Controller para generar HTML a partir de plantillas QWeb.
- **Hacer una SPA desde cero con Vue, React o Angular:** utilizar los Web Controller para generar JSON a partir de los datos. Hacer una API REST con Web Controllers.

Las dos últimas opciones son las que vamos a explorar en este apartado.

6.1 ¿QUÉ SON LOS WEB CONTROLLERS?

Se trata de funciones que responden a “URIs” (Identificador de Recursos Uniformes) concretas.

El servidor Odoo tiene un sistema de rutas para atender las peticiones. Por ejemplo, cuando accedemos a “/web” lo que proporciona es el “backend”. Si accedemos a “/pos/web” lo que nos proporciona es el “Point of Sale”. Nosotros podemos crear nuestras propias rutas para obtener páginas web personalizadas u otros datos como XML o JSON.

Para hacer estas rutas usamos los Web Controllers. De hecho, cuando creamos un nuevo módulo con “scaffold”, se crea un fichero “controllers/controllers.py” que, por defecto, está comentado.

Veamos un ejemplo mínimo de controlador para analizarlo:

```
class MyController(http.Controller):
    @http.route('/school/course/', auth='user', type='json')
    def course(self):
    ...
```

Lo primero que podemos ver es que la clase hereda de “**http.controller**”. Recordemos, por ejemplo, que los modelos heredan de “**models.Model**”.

Esta clase le da las propiedades necesarias para atender peticiones HTTP y queda registrada como un controlador web. Esta clase tendrá como atributos unas funciones con un decorador específico. Estas funciones se ejecutarán cada vez que el usuario acceda a la ruta determinada en el decorador.

El decorador “**@http.route**” permite indicar:

- La ruta que atenderá.
- El tipo de autenticación con **auth=**, que puede ser **users** si requiere que esté autenticado, **public** si puede estar autenticado (y si no lo está, lo trata como un usuario público) o **none** para no tener en cuenta para nada la autenticación o el usuario actual.
- El tipo de datos que espera **recibir y enviar** con **type=“http”** o **type=“json”**. Esto quiere decir que puede aceptar datos con la sintaxis HTTP de GET o POST o que espera a interpretar un body con información en formato JSON.
- Los métodos HTTP que acepta con **methods=**.
- La manera en la que trata las peticiones Cross-origin con **cors=**.
- Si necesita una autenticación con **csrf=**.

Las dos últimas opciones se deben poner en caso de hacer peticiones por una aplicación externa.

En ese caso se pondrá: **cors=‘*’, csrf=False**, ya que queremos que acepte cualquier petición externa y que no autentique con csrf, ya que tendremos que implementar nuestra propia autenticación.

Esta función recibirá parámetros, como veremos en el siguiente apartado, y retornará una respuesta. En caso de ser **type=‘json’** retornará un JSON y en caso de ser **type=‘http’**, retornará preferiblemente un HTML. Dejemos el JSON para más adelante y centrémonos en crear HTML.

Odoo tiene su motor de plantillas HTML que es QWeb, a partir de este se puede construir el HTML con datos que queramos. También podemos **no usar el motor de plantillas y generar algorítmicamente el HTML desde Python**. Esta segunda opción solo es recomendada si va a ser muy simple o estático o si queremos tener un gran control sobre el HTML generado.

Veamos el ejemplo que proporciona Odoo cuando hacemos “scaffold” para crear un módulo:

```
@http.route('/school/school/objects/', auth='public')
def list(self, **kw):
    return http.request.render('school.listing', {
        'root': '/school/school',
        'objects': http.request.env['school.school'].search([]),
    })
```

Aquí se utiliza una nueva herramienta que es “**http.request.render()**”. Esta función utiliza una plantilla para crear un HTML a partir de unos datos.

Veamos ahora la plantilla “school.listing”:

```
<template id="school.listing">
    <ul>
        <li t-foreach="objects" t-as="object">
            <a t-attf-href="#{ root }/objects/#{ object.id }">
                <t t-esc="object.display_name"/>
            </a>
        </li>
    </ul>
</template>
```

En esta plantilla, “objects” es el “recordset” que recorrerá y mostrará una lista de los registros enviados por el “request.render”.

! **Atención:** como hemos comentado anteriormente, esta forma de generar HTML es opcional y es posible generar el HTML desde Python.

Todo lo anterior funciona perfectamente si accedemos a esta ruta desde el mismo navegador en el que hemos hecho “login” anteriormente. De esta manera se cumplirá que no estamos haciendo una petición Cross-origin (desde otro dominio) y que estamos autenticados.

El caso de hacer una petición desde otro lugar (otro dominio) lo exploraremos en el apartado del **API REST**. Pero es posible que queramos mostrar cosas a usuarios que no tenemos autenticados, como por ejemplo un catálogo en una web.

Ya tenemos **auth='public'**, no obstante, si no está autenticado, la función “search” va a fallar. Para que no falle se puede poner “sudo()” antes de la función. De esta manera ignora si el usuario tiene permisos o si está autenticado.

```
'objects': http.request.env['school.school'].sudo().search([]),
```

! **Atención:** mucho cuidado con el uso de “sudo()”. En general, es mejor confiar en la autenticación de Odoo para todo. Si no es posible, hay que establecer un sistema de autenticación adecuado. En caso de ser información totalmente pública, hay que limitar el acceso de los usuarios a lo mínimo imprescindible.

6.2 PASAR PARÁMETROS AL WEB CONTROLLER

Los métodos decorados con “**@http.route**” aceptan parámetros enviados por el “body” mediante POST, parámetros enviados mediante “?” de GET o en la misma URL como en el caso de un servicio REST. Veamos las tres opciones:

6.2.1 Parámetros por el POST o GET:

En caso de ser `type='http'`, espera un body de POST tradicional o un GET con “?” y “&”, similar a:

```
parametro=valor&otroparametro=otrovalor
```

Si sabemos el nombre de los parámetros, los podemos poner en la función decorada. En caso de desconocerlos, podemos poner “****args**” y de esa manera “**args**” será una lista de parámetros. No es preciso que sea “args”, de hecho, muchas veces en nomenclado como “****kw**”.

Veamos un ejemplo:

```
@http.route('/school', auth='public', cors='*', type='http')
def get_course(self, model, obj, **kw):
    model = http.request.env[model].sudo().search([('id', '=', obj)])
    .mapped(lambda p: p.read()[0])
    return model
```

Este ejemplo se puede llamar con un POST en el que enviemos los parámetros “`model=course&obj=23`” o con un GET en el que la URI, suponiendo estamos haciendo pruebas en un servidor en “localhost” sería: <http://localhost:8069/school?model=course&obj=23>

6.2.2 Parámetros por REST:

Modifiquemos este ejemplo para aceptar peticiones al modo de los servicios REST:

```
@http.route('/school/<model>/<obj>', auth='public', cors='*', type='http')
def get_course(self, model, obj, **kw):
    model = http.request.env[model].sudo().search([('id', '=', obj)])
    .mapped(lambda p: p.read()[0])
    return model
```

No hay más que hacer una petición POST o GET sin enviar nada pero con la URL (suponiendo “localhost”): <http://localhost:8069/school/course/23>

6.2.3 Parámetros por JSON:

Tan solo tenemos que cambiar `type='http'` por `type='json'` y enviar un POST. Odoo necesita que el POST tenga como cabecera “**Content-Type: application/json**” y el “body” tendrá esta sintaxis en el caso del ejemplo anterior:

```
'{"jsonrpc": "2.0", "method": "call", "params": {"model": "course", "obj": "23"}}'
```

Para probar todas estas peticiones se recomienda usar programas como “PostMan” o la extensión de Visual Studio Code “Thunder Client”, que permiten hacer muchas pruebas rápidamente. En caso de hacer alguna prueba puntual, se puede usar curl con comandos como este:

```
curl -i -X POST -H "Content-Type: application/json" \
-d '{"jsonrpc": "2.0", "method": "call", "params": {"model": "course", "obj": "23"}}' \
localhost:8069/school
```

El hecho de poner `type='json'` o `type='http'` cambia el comportamiento de la función decorada. En cuanto a la forma de aceptar los datos ha quedado claro, pero la forma de retornarlos también

cambia. Cuando es de tipo **“http”**, le decimos que retorne un HTML, no obstante, no solo retorna un HTML, sino que modifica las cabeceras para indicar que es de tipo **“http”**. Si queremos aceptar datos por el método GET será más fácil decir que es de **type='http'**, pero si queremos que retorne un JSON, hay que indicarlo manualmente manipulando el **“http.response”**:

```
return http.Response(  
    json.dumps(resp),  
    status=200,  
    mimetype='application/json')
```

El uso de **“http.response”** no es necesario en caso de retornar un JSON por un decorador de tipo **“json”** o un HTML por un decorador **“http”**, ya que el propio decorador ya añade las cabeceras necesarias y serializa los datos.

En ocasiones puede haber un problema con el objeto que retornamos, ya que la función **“json.dumps”** no siempre podrá serializar todo tipo de datos. Es posible indicarle que función aplicar para que intente serializar datos **“no serializables”** con el parámetro **“default”**. Por ejemplo:

```
return http.Response(  
    json.dumps(resp), default=str  
    status=200,  
    mimetype='application/json')
```

Con este ejemplo, a los datos no serializables, les intenta aplicar la función **“str”**. Esto es útil, por ejemplo, con los datos en formato **“datetime”**. Sin embargo, habrá casos en los que no será suficiente y deberemos realizarlos manualmente.

6.2.4 CORS CON ODOO

En caso de hacer peticiones desde otra URL, como en el caso de las API, hay que configurar el **cors=** con los dominios de los clientes que acepta. Normalmente, pondremos **cors=“*”** como en algunos ejemplos anteriores, con el fin de aceptar conexiones desde cualquier dominio.

Al hacer peticiones Cross-Origin, las cookies que envía el servidor al autenticar no se registran en el navegador, por lo que es **necesario implementar un protocolo de sesión por “Token”**.

Si queremos que todo Odoo acepte CORS, lo mejor es configurar un Nginx como proxy y ya de paso que establecemos esta configuración, configurar el HTTPS.

6.2.5 AUTENTICACIÓN

Este es el código del controlador de autenticación en Odoo:

```
@http.route('/web/session/authenticate', type='json', auth="none")  
def authenticate(self, db, login, password, base_location=None):  
    request.session.authenticate(db, login, password)  
    return request.env['ir.http'].session_info()
```

En caso de hacer una aplicación web en la misma URL y necesitar autenticación con un usuario de Odoo, tendremos que hacer una petición con JSON a esta ruta.

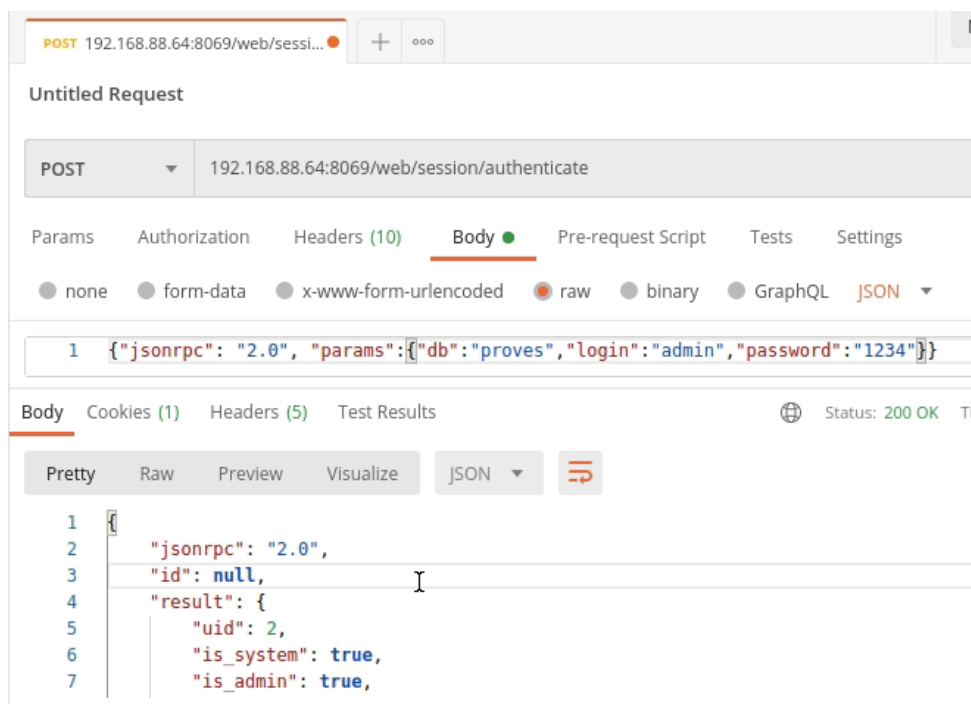
Si estéis utilizando Visual Studio Code como entorno de desarrollo, existe un plugin llamado “Thunder Client” que puede ayudaros a testear desde el propio entorno este tipo de llamadas.

Este plugin puede obtenerse en:

<https://marketplace.visualstudio.com/items?itemName=rangav.vscode-thunder-client>

Otro sistema popular para realizar este tipo de pruebas es “Postman”, disponible en <https://www.postman.com/>:

Aquí vemos un ejemplo con el programa PostMan



6.2.6 HACER UNA API REST

Odoo está más orientado a crear webs con su framework o en su URL que para hacer de API. Pero si queremos hacer una aplicación móvil o una aplicación web externa que consulte sus datos, podemos optar por crear una API REST. Debemos tener en cuenta los siguientes factores:

- Hay que poner **`cors=*`** para poder acceder.
- Debemos desactivar **`csrf`**, ya que no lo podemos usar.
- No podemos utilizar directamente la autenticación con Odoo. Necesitamos implementar algún protocolo para mantener la sesión, algo como “Tokens JWT”.
- En las API REST, el método de la petición es el verbo, así que hay que obtener el método para hacer cosas distintas.
- Si en el decorador ponemos **`type='json'`**, no puede aceptar peticiones GET, ya que no tienen un body. No obstante, si ponemos **`type='http'`** hay que retornar un JSON igualmente. Depende de lo que pidamos por GET y cómo lo implementemos, podemos tener un problema al convertir de recordset a JSON con **`json.dumps()`**, pero lo podemos solventar con **`default=str`** (como hemos comentado antes) o con una herramienta interna de Odoo sacada de la biblioteca **`odoo.tools.date_utils`**.

Veamos un ejemplo:

```
@http.route('/school/api/<model>', auth="none", cors='*', csrf=False,
methods=["POST", "PUT", "PATCH" ], type='json')
def apiPost(self, **args):
    print('***** API POST PUT *****')
    print(args, http.request.httprequest.method)
    model = args['model']
    if (http.request.httprequest.method == 'POST'):
        record = http.request.env['school.' + model].sudo().create(args['data'])
        return record.read()
    if (http.request.httprequest.method == 'PUT' or http.request.httprequest.method == 'PATCH'):
        record = http.request.env['school.' + model].sudo().search([('id', '=', args['id'])])[0]
        record.write(args['data'])
        return record.read()
    return http.request.env['ir.http'].session_info()

@http.route('/school/api/<model>', auth="none", cors='*', csrf=False, methods=["GET", "DELETE"],
type='http')
def apiGet(self, **args):
    print('***** API GET DELETE *****')
    print(args, http.request.httprequest.method)
    model = args['model']
    search = []
    if 'id' in args:
        search = [('id', '=', args['id'])]
    if (http.request.httprequest.method == 'GET'):
        record = http.request.env['school.' + model].sudo().search(search)
        return http.Response( # Retornará un array sin el formato '{"jsonrpc":"2.0"...
            json.dumps(record.read(), default=date_utils.json_default),
            status=200,
            mimetype='application/json'
        )
    if (http.request.httprequest.method == 'DELETE'):
        record = http.request.env['school.' + model].sudo().search(search)[0]
        record.unlink()
        return http.Response(
            json.dumps(record.read(), default=date_utils.json_default),
            status=200,
            mimetype='application/json'
        )
    return http.request.env['ir.http'].session_info()
```

En este ejemplo falta todo lo relativo a la autenticación y algunas comprobaciones para evitar errores, pero se puede ver cómo hacemos una cosa distinta en función del método HTTP. Resulta más fácil de gestionar el GET y el POST por separado por el type.

6.2.7 COMUNICAR UNA SPA VUE/REACT/ANGULAR CON ODOO

Este apartado no tiene mucho que ver con el módulo, sin embargo, es interesante como enlace con el módulo “Desarrollo Web en Entorno Cliente” del CFGS DAW o como introducción a un proyecto final de ciclo. Este sería el servicio de Angular que hace peticiones a la API REST del apartado anterior:

```

@Injectable({
  providedIn: 'root'
})
export class CourseService {
  courseURL = environment.url+'course'; // La URL está en enviroment
  postOptions = { headers: new HttpHeaders({ "Content-type": "application/json; charset=UTF-8"
  });
  constructor( private http: HttpClient) { }
  createCourse(course:ICourse): Observable<ICourse[]> {
    let postBody =
    `{"jsonrpc":"2.0","method":"call","params":{"data":"${JSON.parse(course)}}`;
    let obs: Observable<ICourse[]> =
    this.http.post<{result: ICourse[]}> (this.courseURL,this.postBody,this.postOptions)
    .pipe(map(response => response.result))
    return obs;
  }
}

```

De forma similar podría realizarse con otras bibliotecas de programación reactiva como Vue o React. En <https://www.odooinvue.org/> hay un ejemplo de como conectarse a Odoo con Vue, utilizando el framework para desarrollar código multiplataforma <https://quasar.dev>.

7. DEPENDENCIAS EXTERNAS

Es posible que al desarrollar módulos, se usen dependencias externas, ya sea en forma de bibliotecas de Python y/o ejecutables del sistema. Estas dependencias deben indicarse en el fichero “__manifest__.py” y solventarse a mano en el sistema donde se pondrá en marcha el servicio.

Por ejemplo, para indicar las dependencias de una biblioteca para generar códigos de barras llamada <https://pypi.org/project/python-barcode/>.

Añadiremos una linea similar a:

```
'external_dependencies': {"python": ['python-barcode', "python-barcode[images]"], "bin": []},
```

Si estáis trabajando con un sistema “Docker Compose” como el propuesto en unidades anteriores, para solventar las dependencias deberéis acceder al contenedor e instalarlas ahí. Veamos un ejemplo de como acceder a una shell dentro del contenedor “web” creado con “Docker Compose”:

```
docker-compose exec web bash
```

Y una vez dentro del contenedor, instalamos las dependencias

```

pip3 instal python-barcode
pip3 install python-barcode[images]

```

8. MÓDULOS DE EJEMPLO CON COMENTARIOS

Se pueden encontrar ejemplos de módulos de Odoo comentados con los conceptos tratados durante la unidad en <https://github.com/sergarb1/OdooModulosEjemplos>

9. 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>

<https://konodoo.com/blog/konodoo-blog-de-tecnologia-1>

10. 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