

Sistemas de Gestión Empresarial

UD 06. Desarrollo de módulos de Odoo: Modelo y Vista

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

Introducción	4
La base de datos de Odoo	6
Composición de un módulo	7
Composición de un módulo	7
Modelo	8
Vista	18
Vista Tree	21
Colores en las líneas:	21
Líneas editables:	22
Campos invisibles:	22
Cálculos de totales:	22
Vista Form	22
Definir una vista tree específica en los X2many:	23
Widgets:	23
Valores por defecto en los One2many:	24
Domains en los Many2one:	25
Formularios dinámicos:	25

Asistentes	26
Vista Kanban	26
Vista Calendar	27
Vista Graph	28
Vista Search	28
Bibliografía	29
Autores (en orden alfabético)	29

UD06. DESARROLLO DE MÓDULOS DE ODOO

1. INTRODUCCIÓN

Odoo es un ERP-CRM de código abierto que se distribuye bajo dos tipos de despliegue:

- On-premise en GNU/Linux o Windows con dos versiones (Community, gratuita y Enterprise, de pago).
- SaaS (Software As A Service): La empresa que desarrolla Odoo también proporciona su servicio en la nube.

De esta manera, una empresa puede tener su propio Odoo en un servidor local, en una nube propia o en una nube de terceros. También puede tener acceso a Odoo por el SaaS de Odoo o de otras terceras empresas que proporcionen Odoo como SaaS.

La licencia de Odoo ha ido cambiando a lo largo del tiempo. La actual, de la versión 14 'community' es LGPLv3.

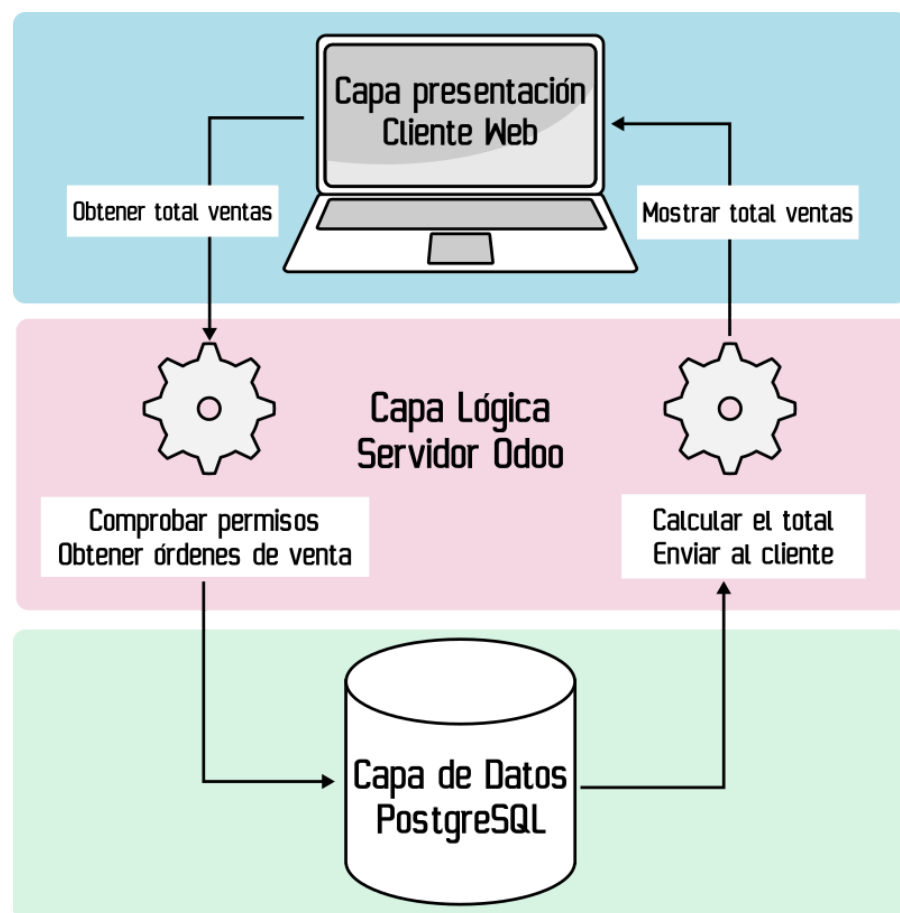
Odoo tiene una arquitectura cliente-servidor de 3 capas:

- La base de datos en un servidor PostgreSQL.
- El servidor Odoo, que engloba la lógica de negocio y el servidor web.
- La capa de cliente que es una SPA (Single Page Application). La capa cliente está subdividida en al menos 3 interfaces muy diferenciadas:
 - El Backend donde se administra la base de datos por parte de los empleados de la empresa.
 - El Frontend o página web, donde pueden acceder los clientes y empleados. Puede incluir una tienda y otras aplicaciones.
 - El TPV para los terminales punto de venta que pueden ser táctiles.

Además de usar su propio cliente, Odoo admite que otras aplicaciones interactúen con su servidor usando XML-RPC. También se pueden desarrollar 'web controllers' para crear un API para aplicaciones web o móviles, por ejemplo.

A parte de la arquitectura de 3 capas, Odoo es un sistema modular. Esto quiere decir que se puede ampliar con módulos oficiales o propios. De hecho, hay un módulo base que contiene el funcionamiento básico del servidor y a partir de ahí se cargan todos los demás módulos.

Cuando instalamos Odoo, antes de instalar ningún módulo, tenemos acceso al backend donde gestionar poco más que las opciones y usuarios. Es necesario instalar los módulos necesarios para un funcionamiento mínimo. Por ejemplo, lo más típico es instalar al menos los módulos de ventas, compras, CRM y contabilidad.



Para ampliar las funcionalidades o adaptar Odoo a las necesidades de una empresa, no hay que modificar el código fuente de Odoo. Tan sólo necesitamos crear un módulo. Los **módulos de Odoo** pueden modificar el comportamiento del programa, la estructura de la base de datos o la interfaz de usuario. En principio, un módulo se puede instalar y desinstalar y los cambios que implicaba el módulo se revierten completamente.


Odoo facilita el desarrollo de módulos porque, además de un ERP, es un framework de programación. Odoo tiene su propio framework tipo **RAD** (Rapid Application Development). Esto significa que con poco esfuerzo se pueden conseguir aplicaciones con altas prestaciones y seguras.

! El poco esfuerzo es relativo. Para desarrollar correctamente en Odoo son necesarios amplios conocimientos de Python, XML, HTML, Javascript y otras tecnologías asociadas como QWeb, JQuery, XML-RPC... La curva de aprendizaje es alta y la documentación es escasa. Además, los errores son más difíciles de interpretar al no saber todo lo que está pasando por debajo. La frustración inicial se verá compensada con una mayor agilidad y menos errores.

Este framework se basa en algunos de los principios generales de los RAD modernos:

- La capa **ORM** (Object Relational Mapping) entre los objetos y la base de datos.

- La combinación Clase Python ↔ ORM ↔ Tabla PostgreSQL se conoce como **Modelo**
- El programador no efectúa el diseño de la base de datos, sólo de las clases y sus relaciones.
- Tampoco es necesario hacer consultas SQL, casi todo se puede hacer con los métodos de ORM de Odoo.
- La arquitectura **MVC** (Modelo-Vista-Controlador).
 - El modelo se programa declarando clases de Python que heredan de `models.Model`. Esta herencia provoca que actúe el ORM y se mapean en la base de datos.
 - La vista se define normalmente en archivos XML y son listas, formularios, calendarios, gráficos, menús... Este XML será enviado al cliente web donde el framework Javascript de Odoo lo transforma en HTML.
 - El controlador también se define en ficheros Python, normalmente junto al modelo. El controlador son los métodos que proporcionan la lógica de negocio.
- Odoo tiene una arquitectura de **Tenencia Múltiple**. De manera que un único servidor puede proporcionar servicio a muchos clientes de bases de datos diferentes.
- Odoo proporciona un diseñador de informes.
- El framework facilita la traducción de la aplicación a muchos idiomas.


 Puesto que Odoo facilita la traducción, es una buena práctica programar todo en inglés, tanto el nombre de las variables como los textos que mostramos a los usuarios. Posteriormente podemos añadir las traducciones necesarias.

2. LA BASE DE DATOS DE ODOO

Gracias al ORM, no hay un diseño definido de la base de datos. La base de datos de una empresa puede tener algunas tablas muy diferentes a otras en función del mapeado que el ORM haya hecho con las clases activas en esa empresa.

Por tanto, es difícil encontrar un diseño entidad-relación o algo similar en la documentación de Odoo. Hay algunos modelos bien conocidos como `res.partner` (clientes, proveedores...) o `sale.order` (Orden de venta) que están en casi todas las empresas y versiones de Odoo. Pero ni siquiera estos tienen en la base de datos las mismas columnas o relaciones que en otras empresas.

Pero muchas veces necesitamos saber el nombre del modelo, del campo o de la tabla en la base de datos. Para ello, Odoo proporciona en su backend el modo desarrollador para saber el modelo y campo poniendo el ratón encima de un campo de los formularios.

 El nombre de las clases de Python siempre ha de ser en minúscula y con el punto para separar por jerarquía. El nombre de un modelo, por tanto, será siempre: `modulo.modelo`. Si el modelo tiene un nombre compuesto, se separa por `_`. En la base de datos, el punto se sustituye por una barra baja.

💬 Aconsejamos dedicar unos minutos a conocer la base de datos usando el modo desarrollador y el cliente de terminal de PostgreSQL. Para ello, podemos repasar las consultas SQL sacando, por ejemplo, el nombre de los clientes que no han hecho ningún pedido. Conseguir hacer esta consulta en el cliente de PostgreSQL demuestra que se ha podido analizar los modelos y campos en el modo desarrollador y que se recuerdan las consultas SQL básicas del curso pasado.

3. COMPOSICIÓN DE UN MÓDULO

Odoo es un programa modular. Tanto el servidor como el cliente se componen de módulos que extienden al módulo 'base'. Cualquier cosa que se quiera modificar en Odoo se ha de hacer creando un módulo.

📖 Puesto que Odoo es de código abierto y todo el código está en Python, que no es un lenguaje compilado, podemos alterar los ficheros Python o XML de los módulos oficiales, cambiando lo que nos interese. Esto puede funcionar, pero es una mala práctica, ya que:

- Cualquier actualización de los módulos oficiales borraría nuestros cambios.
- Si no actualizamos, perderemos acceso a nueva funcionalidades y estaremos expuestos a problemas de seguridad
- Revertir cambios es más difícil y la solución suele pasar por volver a la versión oficial.

Podemos crear módulos para modificar, eliminar o ampliar partes de otros módulos. También podemos crear módulos para añadir funcionalidades completamente nuevas a Odoo sin interferir con el resto del programa. En cualquier caso, el sistema modular está diseñado para que se puedan instalar y desinstalar módulos sin afectar al resto del programa.

Por ejemplo, puede que una empresa no necesite todos los datos que pide Odoo al registrar un producto. Lo que pueden hacer es un módulo que elimine de la vista los campos innecesarios. Si luego se ve que esos campos eran necesarios, sólo hay que desinstalar el módulo y vuelven a aparecer.

Este sistema modular funciona porque, cada vez que se reinicia el servidor o se actualiza un módulo, se interpretan los ficheros Python que definen los modelos y el ORM mapea las novedades en la base de datos. Además se cargan los datos de los ficheros XML en la base de datos y se actualizan los datos que han cambiado.

3.1 Composición de un módulo

Los módulos modifican partes de Modelo-Vista-Controlador. De esta manera, un módulo se compone de ficheros Python, XML, CSS o Javascript entre otros. Todos estos archivos deben estar en una carpeta con el nombre del módulo. Hay una estructura de subcarpetas y de nombres de archivos que casi todos los módulos respetan. Pero todo depende de lo que ponga en el fichero `__manifest__.py`. Este fichero contiene un diccionario de Python con información del módulo y la ubicación de los demás ficheros. Además, el archivo `__init__.py` indica qué ficheros Python se han de importar.

Dentro de un módulo podemos encontrar:

- Ficheros Python que definen los modelos y los controladores.
- Ficheros XML que definen datos que deben ir a la base de datos. Dentro de estos datos, podemos encontrar:
 - Definición de las vistas y las acciones.
 - Datos de demo.
 - Datos estáticos que necesita el módulo.
- Ficheros estáticos como imágenes, CSS, Javascript... que debe ser cargado por la interfaz web.
- Controladores web para gestionar las peticiones web.

Los módulos se guardan en un directorio indicado en la opción `--addons-path` del servidor o en el fichero de configuración. Pueden estar en más de un directorio y dependen del tipo de instalación o la distribución o versión que se instale.

Para crear un módulo se puede hacer manualmente creando la carpeta, el manifest, los directorios y ficheros o utilizar una herramienta llamada `scaffold` proporcionada por Odoo.

```
$ odoo scaffold <module name> <where to put it>
```

Una vez ejecutado este comando, tenemos la estructura básica de directorios y ficheros con un poco de código de ejemplo.

4. MODELO

Los modelos son una abstracción propia de muchos frameworks y relacionada con el ORM. Un modelo se define como una clase Python que hereda de la clase `"models.Model"`. Al heredar de esta clase, adquiere unas propiedades de forma transparente para el programador. A partir de este momento, las clases del lenguaje de programación quedan por debajo de un nivel más de abstracción. Una clase heredada de `"models.Model"` se comporta de la siguiente manera:

- Puede ser accedida como modelo, como recordset o como singleton. Si es accedida como modelo, tiene métodos de modelo para crear recordsets, por ejemplo. Si es accedida como recordset, se puede acceder a los datos que guarda.
- Puede tener atributos internos de la clase, ya que sigue siendo Python. Pero los atributos que se guardan en la base de datos se han de definir como `fields`. Un `field` es una instancia de la clase `"fields.Field"`, y tiene sus propios atributos y funciones. Odoo analizará el modelo, buscará los atributos tipo `field` y sus propiedades y mapeará todo esto en el ORM.
- Los métodos definidos para los recordset reciben un argumento llamado `self` que puede ser un recordset con una colección de registros. Por tanto, deben iterar en el `self` para hacer su función en cada uno de los registros.
- El modelo representa a la tabla entera en la base de datos. Un recordset representa a una

colección de registros de esa tabla y también al modelo. Un singleton es un recordset de un solo elemento.

- Los modelos tienen sus propias funciones para no tener que acceder a la base de datos para modificar o crear registros. Además, incorporan restricciones de integridad.

Este es el ejemplo de un modelo con un solo field:

```
class AModel(models.Model):
    _name = 'a.model'
    _description = 'descripción opcional'

    name = fields.Char(
        string="Name",           # El nombre en el label (Opcional)
        compute="_compute_name_custom", # En caso de ser computado, el nombre de la función
        store=True,              # En caso de ser computado, si se guarda o no
        select=True,             # Forzar que esté indexado
        default='Nombre',        # Valor por defecto, puede ser una función
        readonly=True,           # El usuario no puede escribir directamente
        inverse="_write_name"    # En caso de ser computada y se modifique
        required=True,           # Field obligatorio
        translate=True,          # Si se puede traducir
        help='blabla',           # Ayuda al usuario
        company_dependent=True,  # Transforma columna a ir.property
        search='_search_function', # En caso de ser computado, cómo buscar en él.
        copy = True              # Si se puede copiar con copy()
    )
```

Sobre el código anterior, veamos en detalle todo lo que pasa:

- Se define una clase de Python que hereda de “models.Model”
- Se definen dos atributos `_name` y `_description`. El `_name` es obligatorio en los modelos y es el nombre del modelo. Aquí se ve la abstracción, ya no se accederá a la clase `AModel`, sino al modelo `a.model`.
- Luego está la definición de otro atributo tipo field que será mapeado por el ORM en la base de datos. Como se puede ver, llama al constructor de la clase “fields.Char” con unos argumentos. Todos los argumentos son opcionales en el caso de Char. Hay constructores para todos los tipos de datos.

Es muy probable que a estas alturas no entiendas el porqué de la mayoría del código anterior. Los frameworks requieren entender muchas cosas antes de poder empezar. No obstante, con ese pequeño fragmento de código ya tenemos solucionado el almacenamiento en la base de datos, la integridad de los datos y parte de la interacción con el usuario.

OdoO está pensado para que sea fácilmente modificable por la web. Sin necesidad de entrar al código. Esto es muy útil para prototipar las vistas, por ejemplo. Una de las funcionalidades es el modo desarrollador, que permite, entre otras muchas cosas,

explorar los modelos que tiene en este momento el servidor.

Los modelos tienen algunos atributos del modelo, como `_name` o `_description`. Otros atributos tipo `field`, que se mapean en la base de datos y a los que el usuario tiene acceso y métodos que conforman el controlador. A continuación vamos a detallar todos los tipos de `field` que hay y sus posibilidades:

En primer lugar están los `fields` de datos normales:

- Integer
- Char
- Text
- Date
- Datetime
- Float
- Boolean
- Html
- Binary: Archivos binarios que guarda en base64. Antes de Odoo 13 en este tipo de `fields` se guardaban las imágenes.

Dentro de los `fields` de datos, hay algunos un poco más complejos:

- Image: A partir de la versión 13 de Odoo se pueden guardar imágenes en este `field`. Hay que definir el `max_width` o `max_height` y se redimensionará al guardar.
- Selection: Guarda un dato, pero hay que decirle con una lista de tuplas las opciones que tiene:

```
type = fields.Selection([('1','Basic'),('2','Intermediate'),('3','Completed')])
aselection = fields.Selection(selection='a_function_name') # se puede definir en una función.
```

Todos los ‘`fields`’ mencionados tienen un constructor que funciona de la misma manera que en el ejemplo anterior. Pueden tener un nombre, un valor por defecto, pueden estar definidos por una función...

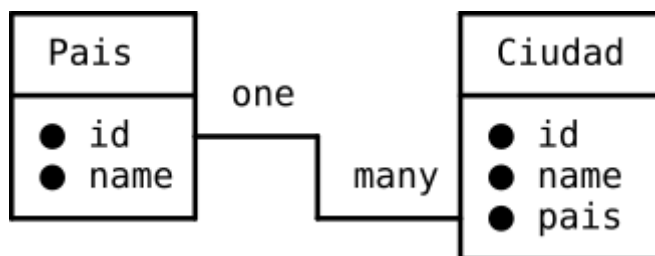
A lo largo de este texto se verán muchos ejemplos de cómo se han definido `fields` según las necesidades.

A continuación, vamos a ver los `fields` relacionales. Dado que el ORM evita que tengamos que crear las tablas y sus relaciones en la base de datos, se necesitan unos campos que definan esas relaciones. Por ejemplo, un pedido de venta tiene un cliente y un cliente puede hacer muchos pedidos de venta. A su vez, ese pedido tiene muchas líneas de pedido, que son sólo de ese pedido y tienen un producto, que puede estar en muchas líneas de venta. Todas estas relaciones acaban estando en la base de datos con claves ajenas. Pero con los frameworks que implementan ORM,

todo esto es mucho más sencillo.

Estos son los fields relacionales de Odoo:

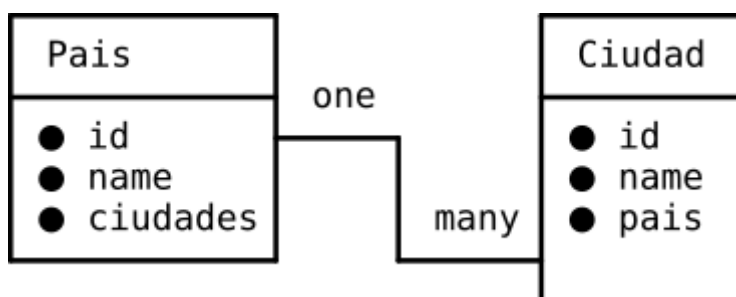
- **Many2one:** Es el más simple, indica que el modelo en el que está tiene una relación muchos a uno con otro modelo. Esto significa que un registro tiene relación con un solo registro del otro modelo, mientras que el otro registro puede tener relación con muchos registros del modelo que tiene el Many2one. En la tabla de la base de datos, esto se traducirá en una clave ajena a la otra tabla.



```
pais_id = fields.Many2one('modulo.pais') # La forma más común
pais_id = fields.Many2one(comodel_name='modulo.pais') # Con el nombre del argumento
pais_id = fields.Many2one(comodel_name='modulo.pais', delegate=True)
```

! Al principio puede parecer contraintuitivo el nombre de Many2one con el tipo de relación. Reflexiona sobre este diagrama y haz otras pruebas para acostumbrarte a este tipo de relación.

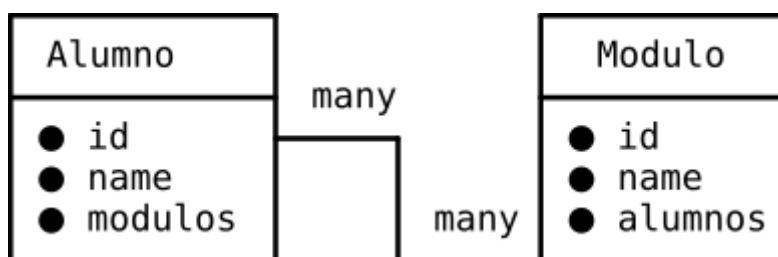
- **One2many:** La inversa del Many2one, de hecho, necesita que exista un Many2one en el otro modelo relacionado. Este field no supone ningún cambio en la base de datos, ya que es el equivalente a hacer un 'SELECT' sobre las claves ajenas de la otra tabla. El One2many se comporta como un campo calculado cada vez que se va a ver.



```
pais_id = fields.Many2one('modulo.pais') # Esto en el modelo Ciudad
...
ciudades_ids = field.One2many('modulo.ciudad', 'pais_id') # El nombre del modelo y el field que
tiene el Many2one necesario para que funcione.
```

- **Many2many:** Se trata de una relación muchos a muchos. Esto se acaba mapeando como una

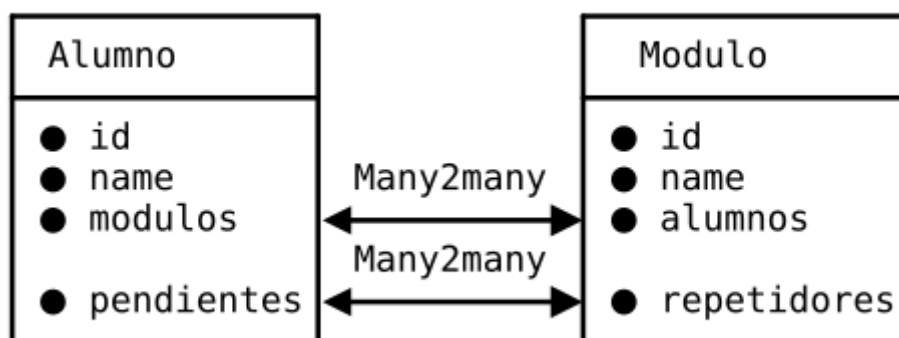
tabla intermedia con claves ajenas a las dos tablas. Hacer los Many2many simplifica mucho la gestión de estas tablas intermedias y evita redundancias o errores. La mayoría de los Many2many son muy fáciles de gestionar, pero algunos necesitan conocer realmente qué ha pasado en el ORM.



```

modulos_ids = fields.Many2many('modulo.modulo') # Esto en el modelo Alumno
...
alumnos_ids = field.Many2many('modulo.alumno') # Esto en el modelo Modulo.
  
```

En el ejemplo anterior, Odoo interpretará que estos dos Many2many corresponden a la misma relación y creará una tabla intermedia con un nombre generado a partir del nombre de los dos modelos. No obstante, no tenemos control sobre la tabla intermedia. Puede que necesitemos tener dos relaciones Many2many independientes sobre los mismos dos modelos. Observemos este diagrama:



Hay dos relaciones, las de alumnos con módulos y las de alumnos repetidores con módulos pendientes. No deben coincidir, pero si no se especifica una tabla intermedia diferente, Odoo considerará que es la misma relación. Por eso hay que especificar la tabla intermedia con la sintaxis completa para evitar errores:

```

alumnos_ids = fields.Many2many(comodel_name='modulo.alumno',
    relation='modulos_alumnos', # El nombre de la tabla intermedia
    column1='modulo_id', # El nombre en la tabla intermedia de la clave a este modelo
  
```

```

column2='alumno_id') # El nombre de la clave al otro modelo.

repetidores_ids = fields.Many2many(comodel_name='modulo.alumno',
    relation='modulos_alumnos_repetidores', # El nombre de la tabla intermedia
    column1='modulo_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='alumno_id') # El nombre de la clave al otro modelo.
...
modulos_ids = field.Many2many(comodel_name='modulo.modulo',
    relation='modulos_alumnos', # El nombre de la tabla intermedia
    column1='alumno_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='modulo_id') # El nombre de la clave al otro modelo.

pendientes_ids = field.Many2many(comodel_name='modulo.modulo',
    relation='modulos_alumnos_repetidores', # El nombre de la tabla intermedia
    column1='alumno_id', # El nombre en la tabla intermedia de la clave a este modelo
    column2='modulo_id') # El nombre de la clave al otro modelo.

```

Las relaciones Many2one, One2many y Many2many suponen la mayoría de las relaciones necesarias en cualquier programa. Hay otro tipo de fields relacionales especiales que facilitan la programación:

- **related:** En realidad no es un tipo de field, sino una posible propiedad de cualquiera de los tipos. Lo que hace un field related es mostrar un dato que está en un registro de otro modelo con el cual se tiene una relación Many2one.

Si tomamos como ejemplo el anterior de las ciudades y países, imaginemos que queremos mostrar la bandera del país en el que está la ciudad. La bandera será un campo 'Image' que estará en el modelo país, pero lo queremos mostrar también en el modelo ciudad. La solución mala sería guardar la bandera en cada ciudad. La buena solución es usar un related:

```

pais_id = fields.Many2one('modulo.pais') # Esto en el modelo Ciudad
bandera = fields.Image(related='pais_id.bandera') # Suponiendo que existe el field bandera y es de
tipo Image.

```

📖 El related puede tener 'store=True' si queremos que lo guarde en la base de datos. En la mayoría de casos es redundante y no sirve. Pero puede que por razones de rendimiento, o para poder buscar, se deba guardar. Esto no respeta la tercera forma normal. En ese caso, Odoo se encarga de mantener la coherencia de los datos.

📖 Otro uso posible de los field related puede ser hacer referencia a fields del propio modelo para tener los datos repetidos. Esto es muy útil en las imágenes, por ejemplo, para almacenar versiones con distintas resoluciones. También puede ser útil para mostrar los mismos fields con varios widgets.

- **Reference:** Una referencia a un campo arbitrario de un modelo. En realidad no provoca una relación en la base de datos. Lo que guarda es el nombre del modelo y del campo en un field char.
- **Many2oneReference:** Un Many2one pero en el que también hay que indicar el modelo al que hace referencia. No son muy utilizados.

... En algunas ocasiones, influidos por el pensamiento de las bases de datos relacionales, podemos decidir que necesitamos una relación One2one. Odoo dejó de usarlas hace tiempo y recomienda en su lugar unir los dos modelos en uno. No obstante, se puede imitar con dos Many2many computados o un One2many limitado a un solo registro. En los dos casos, será tarea del programador garantizar el buen funcionamiento de esa relación.

Una vez estudiado el concepto de modelo y de los 'fields', detengámonos un momento a analizar este código que define 2 modelos:

```
# -*- coding: utf-8 -*-

from odoo import models, fields, api
from openerp.exceptions import ValidationError

####

class net(models.Model):
    _name = 'networks.net'
    _description = 'Networks Model'
    name = fields.Char()
    net_ip = fields.Char()
    mask = fields.Integer()
    net_map = fields.image()
    net_class = fields.Selection([('a', 'A'), ('b', 'B'), ('c', 'C')])
    pcs = fields.One2many('networks.pc', 'net')
    servers = fields.Many2many('networks.pc', relation='net_servers')

class pc(models.Model):
    _name = 'networks.pc'
    _description = 'PCs Model'
    name = fields.Char(default="PC")
    number = fields.Integer()
    ip = fields.Char()
    ping = fields.Float()
    registered = fields.Date()
    uptime = fields.Datetime()
    net = fields.Many2one('networks.net')
    user = fields.Many2one('res.partner')
    servers = fields.Many2many('networks.net', relation='net_servers')
```

Como se puede ver, están casi todos los tipos básicos de field. También podemos ver fields relacionales. Prestemos atención al Many2one 'net' de los PC que permite que funcione el On2many 'pcs' del modelo 'networks.net'. También son interesantes los Many2many en los que declaramos el nombre de la relación para controlar el nombre de la tabla intermedia.

Una vez repasados los tipos de fields y visto un ejemplo, ya podríamos hacer un módulo con datos estáticos y relaciones entre los modelos. Nos faltaría la vista para poder ver estos modelos en el cliente web. Puedes pasar directamente al apartado de la vista si quieres tener un módulo funcional

mínimo. Pero en el modelo quedan algunas cosas que explicar.

Fields computed

Los fields que hemos visto guardan algo en la base de datos. No obstante, puede que no queramos que estén guardados en la base de datos, sino que se recalculen cada vez que vamos a verlos. En ese caso, hay que utilizar campos computados.

Un field computed se define igual que uno normal, pero entre sus argumentos hay que indicar el nombre de la función que lo computa:

```
taken_seats = fields.Float(string="Taken seats", compute='_taken_seats')

@api.depends('seats', 'attendee_ids') # El decorador @api.depends() indica que se llama a
                                     # la función cada vez que cambie el valor de los fields
                                     # seats i attendee_ids.

def _taken_seats(self):
    for r in self: # El for recorre self, que es un recordset con los registros activos      if
not r.seats: # r es un singleton y se puede acceder a los atributos como un objeto
        r.taken_seats = 0.0 # esta asignación ya hace que se vea el resultado.
    else:
        r.taken_seats = 100.0 * len(r.attendee_ids) / r.seats
```

Los field computed no se guardan en la base de datos. Pero puede que necesitemos que se guarde (por ejemplo, para buscar sobre ellos) En ese caso podemos usar store=True. Esto es peligroso, ya que puede que no recalculen más ese campo. El decorador @api.depends() soluciona ese problema si el computed depende de otro field.

En caso de no querer guardar en la base de datos pero querer buscar en el campo, Odoo proporciona la función search. Esta función se ejecuta cuando se intenta buscar por ese campo. Esta función retornará un dominio de búsqueda (esto se explicará más adelante). El problema es que tampoco puede ser un dominio muy complejo y limita la búsqueda.

💬 Hay un truco para poder tener fields computed con store=False y a la vez poder buscar o ordenar. Lo que se puede hacer es otro field del mismo tipo que no sea computed, pero que se sobrescriba cuando se ejecuta el método del que sí es computed. De esta manera, se guarda en la base de datos, pero se recalcula cada vez. El problema es que deben estar los dos fields en la vista. Pero eso se soluciona poniendo invisible="1" en el field computed. El usuario no lo ve, pero Odoo lo recalcula.

En pocas ocasiones necesitamos escribir directamente en un field computed. Si es computed será porque su valor depende de otros factores. Si se permitiera escribir en un field computed, no sería coherente con los fields de los que depende. Pero sí que podemos permitir que se escriba directamente si hacemos la función inverse, la cual ha de sobrescribir los fields de los que depende el computed para que el cálculo sea el que introduce el usuario.

Valores por defecto

En muchas ocasiones, para facilitar el trabajo a los usuarios, algunos fields pueden tener un valor por defecto. Este valor por defecto puede ser siempre el mismo o ser computado en el momento en


que se inicia el formulario. A diferencia de los fields computed, se permite por defecto que el usuario lo modifique y no depende de lo que el usuario va introduciendo en otros fields.

Para que un field tenga el valor por defecto, hay que poner en su constructor el argumento `default=`. En caso de ser un valor estático, sólo hay que poner el valor. En caso de ser un valor por defecto calculado, se puede poner la función que lo calcula.

```
name = fields.Char(default="Unknown")
user_id = fields.Many2one('res.users', default=lambda self: self.env.user)
start_date = fields.Date(default=lambda self: fields.Date.today())
active = fields.Boolean(default=True)
def compute_default_value(self):
    return self.get_value()
a_field = fields.Char(default=compute_default_value)
```


En el ejemplo anterior hay mucho que explicar. En la primera línea se ve la manera más fácil de asignar un valor por defecto, una simple cadena de caracteres para el field Char. En la segunda línea se usa una función lambda que obtiene el usuario. En la tercera una que obtiene la fecha. Después va un default simple que asigna un valor booleano. Por último, tenemos una función que luego es referenciada en el default del último field. De todo esto, hay dos cosas que probablemente no te suenen aún.

La primera cosa son las funciones lambda. Estas son funciones anónimas definidas en el lugar donde se van a invocar. No pueden tener más de una línea.

 No es función de estos apuntes explicar cómo se programa en Python. Si no las conocías, es muy recomendable que busques por internet y amplíes información sobre las funciones Lambda. A lo largo de estos apuntes las volveremos a usar y podrás ver más ejemplos.

La segunda cosa que hay es la definición de la función antes de ser invocada. Python es un lenguaje interpretado y no puede invocar funciones que no se han definido antes de ser invocadas. En el caso de los fields computed, esto no tiene importancia porque se invocan cuando el usuario necesita ver el field. Pero los defaults se ejecutan cuando se reinicia el servidor Odoo e interpreta todos los ficheros Python.

Este efecto se puede ver de forma clara en la función lambda que calcula la hora. Si en vez de lo que hay en el ejemplo, pusiéramos directamente la función `fields.Date.today()`, pondría la fecha de reinicio del servidor y no la fecha de creación del registro. En cambio, al referenciar a una función lambda, esta se ejecuta cada vez que el programa entra en esa referencia. Al igual que al referenciar a una función normal con nombre.

 No hay mejor manera de aprender que probar las cosas. Recomendamos crear un modelo nuevo con valores por defecto de todas las formas y probar a poner la fecha sólo con la función de fecha y luego dentro de la función lambda y comprobar lo que pasa.

Restricciones (constraints)

No en todos los fields los usuarios pueden poner de todo. Por ejemplo, podemos necesitar limitar el

precio de un producto en función de unos límites preestablecidos. Si el usuario crea un nuevo producto y se pasa al poner el precio, no debe dejarle guardar. Las restricciones se consiguen con un decorador llamado `@api.constraint()` el cual ejecutará una función que comprobará si el usuario ha introducido correctamente los datos. Veamos un ejemplo:

```
from odoo.exceptions import ValidationError

...

@api.constrains('age')
def _check_age(self):
    for record in self:
        if record.age > 20:
            raise ValidationError("Your record is too old: %s" % record.age)
```

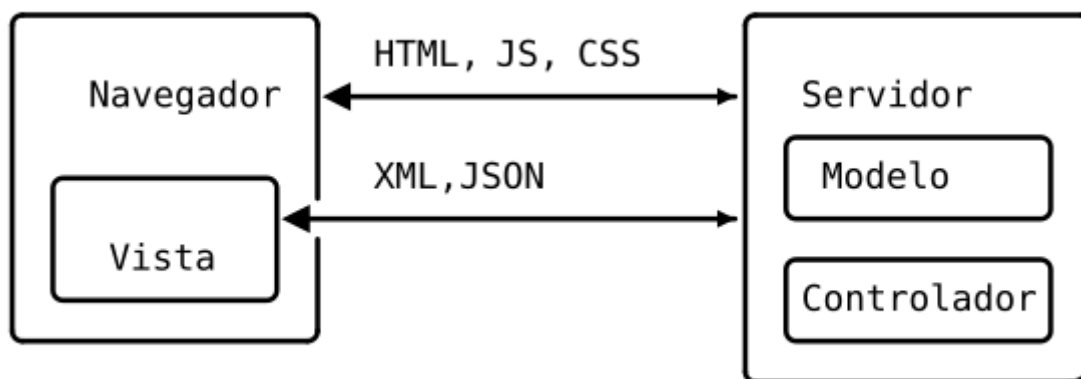
Se supone que hay un field llamado 'age'. Cuando es modificado y se intenta guardar se llama a la función que tiene el decorador. Esta recorre el recordset (recordemos que es importante poner siempre el `for record in self`). Para cada singleton compara la edad y si alguna es mayor de 20, lanza un error de validación. Este lanzamiento impide que se guarde en la base de datos y avisa al usuario de su error.

En ocasiones es más cómodo poner una restricción SQL que hacer el algoritmo que lo comprueba. Además de más eficiente en términos de computación. Para ello podemos usar una **`_sql_constraints`**:

```
_sql_constraints = [
    ('name_uniq', 'unique(name)', 'Custom Warning Message'),
    ('contact_uniq', 'unique(contact)', 'Custom Warning Message')
]
```

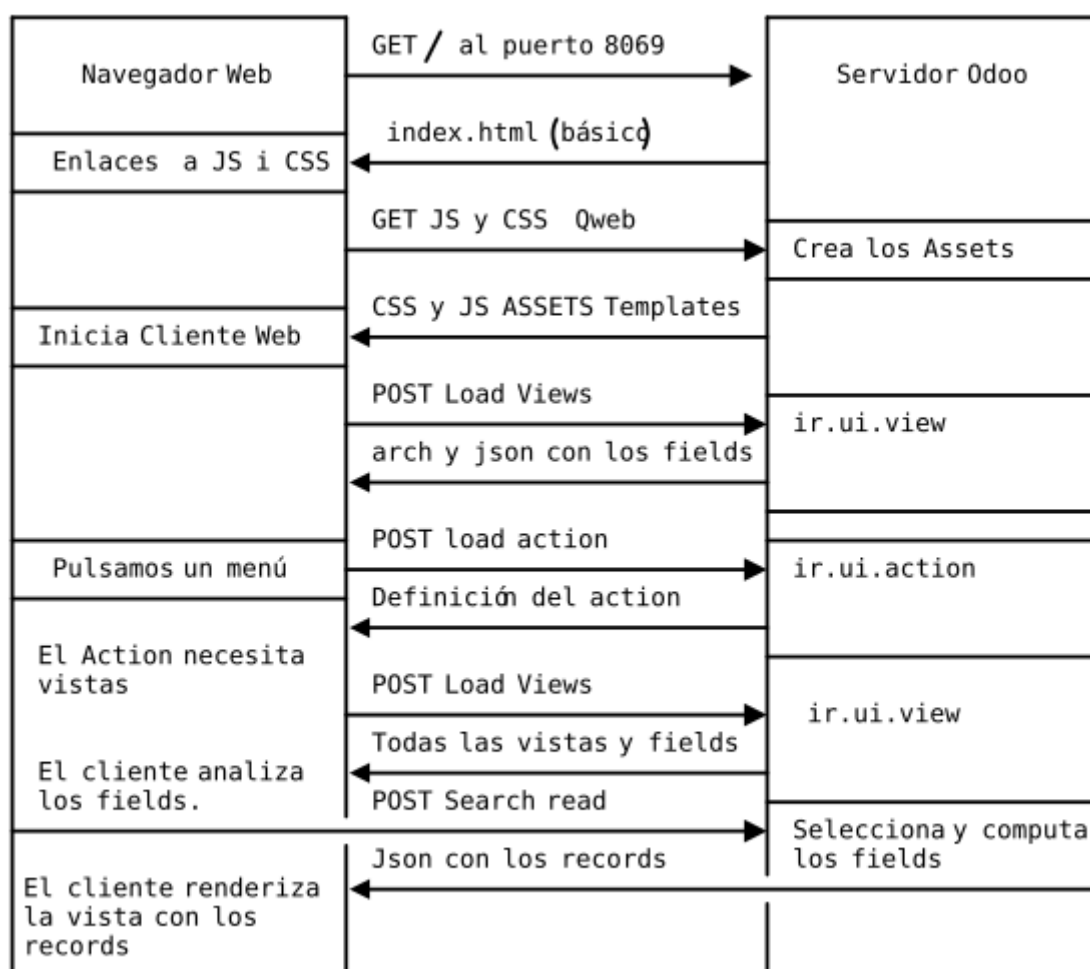
Las SQL constraints son una lista de tuplas en las que está el nombre de la restricción, la restricción SQL y el mensaje en caso de fallo.

5. VISTA



El esquema Modelo-Vista-Controlador que sigue Odoo, la vista se encarga de todo lo que tiene que ver con la interacción con el usuario. En Odoo, la vista es un programa completo de cliente en Javascript que se comunica con el servidor con mensajes breves. La vista tiene tres partes muy diferentes: El backend, la web y el TPV. Nosotros vamos a centrarnos en la vista del backend.

En la primera conexión con el navegador web, el servidor Odoo le proporciona un HTML mínimo, una SPA en Javascript y un CSS. Esto es un cliente web del servidor, es lo que se considera la vista. Pero tampoco carga la vista completa, ya que podría ser inmensa. Cada vez que los menús o botones de la vista requieren cargar la visualización de unos datos, piden al servidor un XML que defina cómo se van a ver esos datos y un JSON con los datos. Entonces la vista renderiza los datos según el esquema del XML y los estilos definidos en el cliente. Esta visualización se hace con unos elementos llamados Widgets, que son combinaciones de CSS, HTML y Javascript que definen el aspecto y comportamiento de un tipo de datos en una vista en concreto. Todo esto es lo que vamos a ver con detalle en este apartado.



Los XML que definen los elementos de la vista se guardan en la base de datos y son consultados como cualquier otro modelo. De esta manera se simplifica la comunicación entre el cliente web y el servidor y el trabajo del controlador. Puesto que cuando creamos un módulo, queremos definir sus vistas, debemos crear archivos XML para guardar cosas en la base de datos. Estos serán referenciados en el `__manifest__.py` en el apartado de data.

🗨 Observad lo que ha pasado cuando se crea un módulo con scaffold. En el `__manifest__.py` hay una referencia a un XML en el directorio views. Este XML tiene un ejemplo comentado de los principales elementos de las vistas, que veremos a continuación.

La vista tiene varios elementos necesarios para funcionar. Los más evidentes son las propias definiciones de las vistas, guardadas en el modelo **ir.ui.view**. Estos elementos tienen al menos los fields que se van a mostrar y pueden tener información sobre la disposición, el comportamiento o el aspecto de los fields. Otros elementos fundamentales son los menús, que están distribuidos de forma jerárquica y se guardan en el modelo **ir.ui.menu**. Los otros elementos fundamentales son las

acciones o **actions**, que enlazan una acción del usuario (como pulsar en un menú) con una llamada al servidor desde el cliente para pedir algo (como cargar una nueva vista). Las 'actions' están guardadas en varios modelos dependiendo del tipo. A continuación vamos a ver con más detalle todos estos elementos. Pero antes un ejemplo completo:

```
<odoo>
  <data>
    <!-- explicit list view definition -->
    <record model="ir.ui.view" id="prueba.student_list">
      <field name="name">Student list</field>
      <field name="model">prueba.student</field>
      <field name="arch" type="xml">
        <tree>
          <field name="name"/>
          <field name="topics"/>
        </tree>
      </field>
    </record>
    <!-- actions opening views on models -->
    <record model="ir.actions.act_window" id="prueba.student_action_window">
      <field name="name">student window</field>
      <field name="res_model">prueba.student</field>
      <field name="view_mode">tree,form</field>
    </record>
    <!-- Top menu item -->
    <menuitem name="prueba" id="prueba.menu_root"/>
    <!-- menu categories -->
    <menuitem name="Administration" id="prueba.menu_1" parent="prueba.menu_root"/>
    <!-- actions -->
    <menuitem name="Students" id="prueba.menu_1_student_list" parent="prueba.menu_1"
      action="prueba.student_action_window"/>
  </data>
</odoo>
```

En este ejemplo se ven "records" en XML que indican que se va a guardar en la base de datos. Estos "records" dicen el modelo donde se guardará y la lista de "fields" que queremos guardar.

El primer "record" define una vista tipo tree que es una lista de estudiantes donde se verán los campos "name y topics". El segundo "record" es la definición de un "action" tipo "window", es decir, que abre una ventana para mostrar unas vistas de tipo "tree y form" (formulario). Los otros definen tres niveles de menú: el superior, el intermedio y el menú desplegable que contiene el "action". Cuando el usuario navegue por los dos menús superiores y presione el tercer elemento de menú se ejecutará ese action que cargará la vista "tree" definida y una vista "form" inventada por Odoo.

Para poder ver un modelo en el cliente web de Odoo no necesitamos más que un menú que accione un action tipo window sobre ese modelo. Odoo es capaz de inventar las vistas básicas para poder verlo. No obstante suelen ser menos atractivas y útiles que las que definimos nosotros.

Quedémonos por el momento con la definición básica de un "action window" y de los menús. Vamos a centrarnos antes en las vistas.

5.1 Vista Tree

La vista tree muestra una lista de “records” sobre un modelo. Veamos el ejemplo básico:

```
<record model="ir.ui.view" id="prueba.student_list">
  <field name="name">Student list</field>
  <field name="model">prueba.student</field>
  <field name="arch" type="xml">
    <tree>
      <field name="name"/>
      <field name="topics"/>
    </tree>
  </field>
</record>
```

Como se ve, esta vista se guardará en el modelo **ir.ui.view** con un **external ID** llamado ‘prueba.student_list’. Tiene más posibles fields, pero los mínimos necesarios son **name**, **model** y **arch**. El field arch guarda el XML que será enviado al cliente para que renderice la vista.

Dentro del field arch está la etiqueta **<tree>** que indica que es una lista y dentro de esta etiqueta tenemos más fields que son los fields del modelo prueba.student que queremos que se vean.

Esta vista tree se puede mejorar de muchas formas. Veamos algunas de ellas:

Colores en las líneas:

Odoo no da libertad absoluta al desarrollador en este aspecto y permite un número limitado de estilos para dar a las líneas en función de alguna condición. Estos estilos son como los de Bootstrap:

- decoration-bf - Líneas en BOLD
- decoration-it - Líneas en ITALICS
- decoration-danger - Color LIGHT RED
- decoration-info - Color LIGHT BLUE
- decoration-muted - Color LIGHT GRAY
- decoration-primary - Color LIGHT PURPLE
- decoration-success - Color LIGHT GREEN
- decoration-warning - Color LIGHT BROWN

```
<tree decoration-info="state=='draft'" decoration-danger="state=='trashed'">
```

En este ejemplo se ve cómo se asigna un estilo en función del valor del field state.

Se puede comparar un field date con una variable de QWeb llamada **current_date**:

```
<tree decoration-info="start_date==current_date">
```

Líneas editables:

Si no necesitamos un formulario para modificar algunos fields, podemos hacer el tree editable.

! Si lo hacemos editable no se abrirá un formulario cuando el usuario haga click en un elemento de la lista.

Para hacerlo editable hay que poner el atributo **editable="[top | bottom]"**. Además pueden tener un atributo **on_write** que indica qué hacer cuando se edita.

Campos invisibles:

Algunos campos sólo han de estar para definir el color de la línea, servir como lanzador de un field computed o ser buscados, pero el usuario no necesita verlos. Para eso se puede poner el atributo **invisible="1"** en el field que necesitemos.

Cálculos de totales:

En los fields numéricos, si queremos que odoo muestre la suma total, podemos usar el atributo **sum**.

Así quedaría un tree con todo lo que hemos explicado:

```
<record model="ir.ui.view" id="prova.student_list">
  <field name="name">Student list</field>
  <field name="model">prova.student</field>
  <field name="arch" type="xml">
    <tree decoration-info="qualification<5" editable="top">
      <field name="name"/>
      <field name="topics" invisible="true"/>
      <field name="qualification" sum="Total Qualifications"/>
    </tree>
  </field>
</record>
```

5.2 Vista Form

Esta vista permite editar o crear un nuevo registro en el modelo que represente. Muestra un formulario que tiene versión editable y versión sólo vista. Al tener dos versiones y necesitar más complejidad, la vista form tiene muchas más opciones.

! En esta vista hay que tener en cuenta que al final se traducirá en elementos HTML y CSS y que los selectores CSS son estrictos con el orden y jerarquía de las etiquetas. Por tanto no todas las combinaciones funcionan siempre.

El formulario deja cierta libertad al desarrollador para controlar la disposición de los fields y la estética. No obstante, hay un esquema que conviene seguir. Un formulario puede ser la etiqueta

<form> con etiquetas de fields dentro, igual que el tree. Pero conseguir un buen resultado será más complicado y hay que introducir elementos HTML. Odoo propone unos contenedores con unos estilos predefinidos que funcionan bien y estandarizan los formularios de toda la aplicación.

Para que un formulario quede bien y no ocupe toda la pantalla se puede usar la etiqueta <sheet> que englobe al resto de etiquetas. Si la usamos, los fields perderán el label, por lo que debemos usar la etiqueta <group string="Nombre del grupo"> antes de las de los fields. También se puede poner en cada field <label for="nombre del field">.

Si hacemos varios groups o groups dentro de groups, el CSS de Odoo ya alinea los fields en columnas o los separa correctamente. Pero si queremos separar manualmente algunos fields, podemos usar la etiqueta <separator string="Nombre del separador"/>.

Otro elemento de separación y organización es el <notebook> <page string="título">. Que crea unas pestañas que esconden partes del formulario y permiten que quepa en la pantalla.

! Las combinaciones de group, label, separator, notebook y page son muchas. Se recomienda ver cómo han hecho los formularios en algunas partes de Odoo. Los formularios oficiales tienen muchas cosas más complejas. Algunas de ellas las veremos a continuación.

Una vez mencionados los elementos de estructura del formulario, vamos a ver cómo modificar la apariencia de los fields:

Definir una vista tree específica en los X2many:

Los One2many y Many2many se muestran, por defecto, dentro de un formulario como una subvista tree. Odoo coge la vista tree con más prioridad del modelo al que hace referencia el X2many y la incrusta dentro del formulario. Esto provoca dos problemas: Que si cambias esa vista también cambian los formularios y que las vistas tree cuando son independientes suelen mostrar más fields de los que necesitas dentro de un formulario que las referencia. Por eso se puede definir un tree dentro del field:

```
<field name="subscriptions" colspan="4">
  <tree>...</tree>
</field>
```

Widgets:

Un widget es un componente del cliente web que sirve para representar un dato de una forma determinada. Un widget tiene una plantilla HTML, un estilo con CSS y un comportamiento definido con Javascript. Si queremos, por ejemplo, mostrar y editar fechas, Odoo tiene un widget para los Datetime que muestra la fecha con formato de fecha y muestra un calendario cuando estamos en modo edición.

Algunos fields pueden mostrarse con distintos widgets en función de lo que queramos. Por ejemplo, las imágenes por defecto están en un widget que permite descargarlas, pero no verlas en la web. Si le ponemos widget="image" las mostrará.

Es posible hacer nuestros propios widgets, pero requiere saber modificar el cliente web, lo cual no está contemplado en esta unidad didáctica.

Aquí tenemos algunos widgets disponibles para fields numéricos:

- integer: El número sin comas. Si está vacío, muestra un 0.
- char: El número aunque muestra el campo más ancho. Si está vacío muestra un hueco.
- id: No se puede editar.
- float: El número con decimales.
- percentpie: Un gráfico circular con el porcentaje.
- progressbar: Una barra de progreso:
- monetary: Con dos decimales.
- field_float_rating: Estrellas en función de un float.

Para los fields de texto tenemos algunos que con su nombre se explican solos: char, text, email,url,date,html.

Para los booleanos, a partir de Odoo 13 se puede mostrar una cinta al lado del formulario con el widget web_ribbon.

Los fields relacionales se muestran por defecto como un selection o un tree, pero pueden ser:

- many2onebutton que indica sólo si está seleccionado.
- many2many_tags que muestra como etiquetas:

! La lista de widgets es muy larga y van entrando y saliendo en las distintas versiones de Odoo. Recomendamos explorar los módulos oficiales y ver cómo se utilizan para copiar el código en nuestro módulo. Hay muchos más, algunos sólo están si has instalado un determinado módulo, porque se hicieron a propósito para ese módulo, aunque los puedes aprovechar si lo pones como dependencia.

Valores por defecto en los One2many:

Cuando tenemos un One2many en una vista form, nos da la opción de crear nuevos registros. Recordemos que un One2many no es más que una representación del Many2one que hay en el otro modelo. Por tanto, si creamos nuevos registros necesitamos que el Many2one del modelo a crear referencie al registro que estamos modificando del modelo que tiene el formulario.

Para conseguir que el formulario que sale en la ventana emergente tenga ese valor por defecto, lo que haremos será pasar por contexto un dato con un nombre especial que será interpretado: **context="{ 'default_<field many2one>': active_id }"**.

Esto se puede hacer también en un action. De hecho, al pulsar un elemento del tree se ejecuta un action también:

```
<field name="context">{"default_doctor": True}</field>
```


! El concepto de contexto en Odoo no está explicado todavía. De momento pensemos que es un cajón de sastre donde poner las variables que queremos pasar de la vista al controlador y viceversa.

La palabra reservada **active_id** es una variable que apunta al id del registro que está activo en ese formulario.

Domains en los Many2one:

Aunque se pueden definir en el modelo, puede que en una vista determinada necesitemos un domain más específico. Así se definen:

```
<field name="hotel" domain="[('ishotel', '=', True)]"/>
```

Formularios dinámicos:

El cliente web de Odoo es una web reactiva. Esto quiere decir que reacciona a las acciones del usuario o a eventos de forma automática. Parte de esta reactividad se puede definir en la vista formulario haciendo que se modifique en función de varios factores. Esto se consigue con el atributo **attrs** entre otros de los fields.

Se puede **ocultar condicionalmente** un field:

```
<field name="boyfriend_name" attrs="{ 'invisible': [('married', '!=', False)]}" />  
<field name="boyfriend_name" attrs="{ 'invisible': [('married', '!=', 'selection_key')]}" />
```

Se puede **mostrar o ocultar en modo edición o lectura**:

```
<field name="partido" class="oe_edit_only"/>  
<field name="equipo" class="oe_read_only"/>
```

Muchos formularios tienen estados y se comportan como un asistente. En función de cada estado se pueden mostrar o ocultar fields. Hay un atajo al ejemplo anterior si tenemos un field que específicamente se llama **state**. Con el atributo **states** se puede mostrar o ocultar elementos de la venta:

```
<group states="i,c,d">  
  <field name="name"/>  
</group>
```

Dentro del temas de ocultar fields condicionalmente, está la opción de **ocultar una columna de un tree** de un X2many:

```
<field name="lot_id" attrs="{ 'column_invisible': [('parent.state', 'not in', ['sale', 'done'])]" />
```

```
}"/>
```

💬 Fíjate en el ejemplo anterior que dice `parent.state`, esto hace referencia al field state del modelo padre de ese tree. Hay que tener en cuenta que ese tree se muestra con el modelo al que hace referencia el X2many, pero está dentro de un formulario de otro modelo.

Dentro de los formularios dinámicos, se puede **editar condicionalmente un field**. Esto quiere decir que permitirá al usuario modificar un field en función de una condición:

```
<field name="name2" attrs="{ 'readonly': [('condition', '=', False)] }"/>
```

Veamos ahora un ejemplo con todos los attrs:

```
<field name="name" attrs="{ 'invisible': [('condition1', '=', False)],  
                           'required': [('condition2', '=', True)],  
                           'readonly': [('condition3', '=', True)] }" />
```

Asistentes

Los formularios de Odoo pueden ser asistentes con las técnicas que acabamos de estudiar. A partir de Odoo 11 se usa el field status, el atributo states y con un widget statusbar que muestra ese field en la parte superior del formulario como unas flechas.

```
<field name="state" widget="statusbar"  
statusbar_visible="draft,sent,progress,invoiced,done" />
```

5.3 Vista Kanban

La vista tree y la vista form son las que funcionan por defecto en cualquier action. El resto de vistas, como la Kanban, necesitan una definición en XML para funcionar. Además, dada la gran cantidad de opciones que tenemos al hacer un Kanban, no disponemos de etiquetas como en el form que después se traduzcan en HTML o CSS y den un formato estándar y confortable. Cuando estamos definiendo una vista Kanban entramos en el terreno del lenguaje QWeb y del HTML o CSS explícito.

💬 Aprender los detalles de QWeb, HTML y CSS necesarios para dominar el diseño de los Kanban supone mucho espacio que no podemos dedicar en este capítulo. De momento, la mejor manera de hacerlos es entender cómo funcionan los ejemplos y mirar, copiar y pegar el código de los Kanban que ya están hechos.

Veamos un ejemplo mínimo de Kanban y describamos después para qué sirven las etiquetas y atributos.

```
<record model="ir.ui.view" id="terraform.planet_kanban_view">
  <field name="name">Student kanban</field>
  <field name="model">school.student</field>
  <field name="arch" type="xml">
    <kanban>
      <!-- Estos fields se cargan inicialmente y pueden ser utilizados
      por la lógica del Kanban -->
      <field name="name" />
      <field name="id" /> <!-- Es importante añadir el id para el
      record.id.value posterior -->
      <field name="image" />
      <templates>
        <t t-name="kanban-box">
          <div class="oe_product_vignette">
            <!-- Aprovechando un CSS de products -->
            <a type="open">
              
              </a>
            <!-- Para obtener la imagen necesitamos una función javascript
            que proporciona Odoo Llamada kanban-image y esta necesita
            el nombre del modelo, el field y el id para encontrarla -->
            <!-- record es una variable que tiene QWeb para acceder a las
            propiedades del registro que estamos mostrando. Las propiedades
            accesibles son las que hemos puesto en los fields de arriba. -->
            <div class="oe_product_desc">
              <h4>
                <a type="edit"> <!-- Abre un formulario de edición -->
                  <field name="id"></field>
                  <field name="name"></field>
                </a>
              </h4>
            </div>
          </div>
        </t>
      </templates>
    </kanban>
  </field>
</record>
```

5.4 Vista Calendar

Esta vista muestra un calendario si los datos de los registros del modelo tienen al menos un field que indica una fecha y otro que indique una fecha final o una duración. La sintaxis es muy simple, veamos un ejemplo:

```
<record model="ir.ui.view" id="school.travel_calendar">
  <field name="name">travel calendar</field>
```

```

<field name="model">school.travel</field>
<field name="arch" type="xml">
<calendar string="Travel Calendar" date_start="launch_time"
date_delay="distance" <!-- Puede ser delay (en horas) o date_stop -->
color="origin_school"> <!-- El color indica el field que lo modifica
                        No un color literalmente -->
    <field name="name"/>
</calendar>
</field>
</record>

```

Por defecto el delay lo divide en días según la duración de la jornada laboral. Esta se puede modificar con el atributo **day_lenght**

5.5 Vista Graph

La vista graph permite mostrar una gráfica a partir de algunos fields numéricos que tenga el modelo. Esta vista puede ser de tipo pie, bar o line y se comporta agregando los valores que ha de mostrar. En este ejemplo se ve un gráfico en el que se mostrará la evolución de las notas en el tiempo de cada estudiante. Por eso el tiempo se pone como **row**, el estudiante como **col** y los datos a mostrar que son las calificaciones como **measure**. En caso de olvidar poner la al estudiante como 'col' nos mostraría la evolución en el tiempo de la suma de las notas de todos los estudiantes.

```

<record model="ir.ui.view" id="school.qualifications_graph">
  <field name="name">Qualifications graph</field>
  <field name="model">school.qualifications</field>
  <field name="arch" type="xml">
    <graph string="Qualifications History" type="line">
      <field name="time" type="row"/>
      <field name="student" type="col"/>
      <field name="qualification" type="measure"/>
    </graph>
  </field>
</record>

```

5.6 Vista Search

Esta no es una vista como las que hemos visto hasta ahora. Entra dentro de la misma categoría y se guarda en el mismo modelo, pero no se muestra ocupando la ventana, sino la parte superior donde se sitúa el formulario de búsqueda. Lo que permite es definir los criterios de búsqueda, filtrado o agrupamiento de los registros que se muestran en el cliente web. Veamos un ejemplo completo de vista search:

```

<search>
  <field name="name"/>
  <field name="inventor_id"/>
  <field name="description" string="Name and description" filter_domain="[('name', 'ilike', self), ('description', 'ilike', self)]"/>
  <field name="boxes" string="Boxes or @" filter_domain="[('name', 'ilike', self), ('description', 'ilike', self)]"/>
  <filter name="my_ideas" string="My Ideas" domain="[('inventor_id', '=', uid)]"/>
  <filter name="more_100" string="More than 100 boxes" domain="[('boxes', '>', 100)]"/>
  <filter name="Today" string="Today" domain="[('date', '>=', datetime.datetime.now().strftime('%Y-%m-%d 00:00:00')), ('date', '<=', datetime.datetime.now().strftime('%Y-%m-%d 23:59:59'))]"/>
</search>

```

```
datetime.datetime.now().strftime('%Y-%m-%d 23:23:59')))]"/>
<filter name="group_by_inventor" string="Inventor" context="{ 'group_by': 'inventor_id' }"/>
<filter name="group_by_exit_day" string="Exit" context="{ 'group_by': 'exit_day:day' }"/>
</search>
```

La etiqueta **field** dentro de un search permite indicar por qué fields buscará el input de búsqueda de la web. Se puede poner el atributo **filter_domain** si queremos incorporar una búsqueda más avanzada incluso con varios fields. Se usará la sintaxis de los dominios que ya hemos usado en Odoo en otras ocasiones.

La etiqueta **filter** establece un filtro predefinido que se aplicará pulsando en el menú. Necesita el atributo **domain** para que haga la búsqueda.

La etiqueta **filter** también puede servir para agrupar en función de un criterio. Para ello, hay que poner en el **context** la clave **group_by**, de forma que hará una búsqueda y agrupará por el criterio que le digamos. Hay un tipo especial de agrupación por fecha (último ejemplo) en la que podemos especificar si queremos agrupar por día, mes u otros.

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