UD5: Creación de módulos. Estructura de Odoo.

1. El lenguaje python en Odoo

La versión que nos interesa manejar es la que utiliza el Odoo en el que estamos trabajando, para Odoo 15 necesitamos la versión 3.7 o superior para que funcione correctamente.

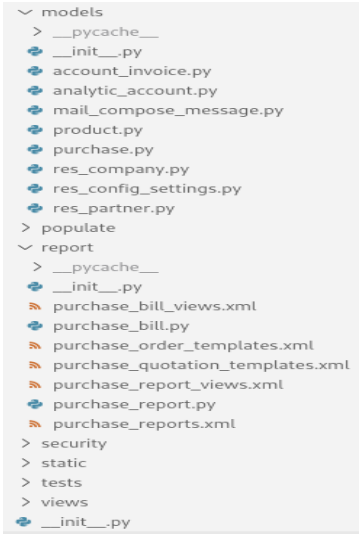
En este apartado veremos algunas características de Python que nos puedan ayudar en la creación de nuevos módulos para Odoo.

● Los módulos y los paquetes.

Un módulo en Python es un fichero que contiene elementos de Python y cuyo nombre es el del módulo seguido de .py. Los módulos se pueden importar en otros archivos de Python para ser utilizados y permiten importar otros módulos. Normalmente ponemos todas las declaraciones import al principio del módulo.

```
from odoo import models,fields
```

Los Paquetes son una forma de estructurar el espacio de nombres de módulos de Python usando la separación por un punto. Por ejemplo, el nombre del módulo purchases.product designa un submódulo product en un paquete llamado purchases. Los archivos __init__.py son obligatorios para que Python trate los directorios que contienen los archivos como paquetes. Por ejemplo tenemos el módulo purchases con los siguientes elementos.



Los podriamos utilizar en otros elementos con:

```
from purchases.report  import purchases_bill
from purchases import *
```

(con esto lo importamos todo lo que esta contenido en purchases)

B) Las clases en Python.

Uno de los elementos principales con los que trabajaremos en Odoo seran las classes. Vamos a ver como se define una clase en Python.

Las clases nos permiten empaquetar datos y funcionalidades. Al crear una nueva clase, creamos un objeto nuevo y lo podemos instanciar. Las instancias de clase también pueden tener funciones o métodos (definidos dentro de la clase).

Para crear una clase:

class NombreDeLaClase

<statement-1>

....

<statement-N>

Los modelos en Odoo extienden la clase models.Models (que es el modelo básico, también pueden ser models.TransientModel o models.AbstractModel)

Ejemplo de una clase Python en Odoo:

```
from odoo import  fields, models
class Course(models.Model):
   _name = 'openacademy.course'
  _description="Descripcion de la clase"
   name = fields.Char(string="Title", required=True)
   description = fields.Text()
   sessions = fields.One2many('openacademy.session',"course_id", string="Sessions")
```

Las clases se pueden instanciar o hacer referencia a atributos o métodos, para esto se utiliza la sintaxis objeto.nombre, donde nombre es el uno de los atributos o de los métodos de la clase y el objeto seria una instancia de la clase:

Ejemplo:

La clase

```
class Course(models.Model):
   _name = 'openacademy.course'
   name = fields.Char(string="Title", required=True)
   description = fields.Text()


    def fun(self): #funcion de la clase Course
      return 'hello world'
```

Instanciamos la clase y accedemos al método:

```
x = Course()
xf = x.fun
  while True:
    print(xf())
```

C) Manejo de errores

Una de las cosas mas importantes en programación es el control de errores de ejecución o errores en los datos.

Aunque una expresión sea correcta, puede generar un error al ejecutarla. Este tipo de errores se llaman excepciones. Pero, la mayoría de las excepciones no se gestionan por código, y muestran mensajes de error. Por ejemplo: ZeroDivisionError, NameError o TypeError.

Podemos utilizar código para gestionar algunas excepciones. Por ejemplo, pedimos que se ponga como dato un entero:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

La declaración try … except tiene una cláusula else opcional, que, cuando está presente, debe seguir todas las cláusulas except. Es útil para el código que debe ejecutarse si la cláusula try no lanza una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

➢ LAS EXCEPCIONES EN ODOO:

El módulo de excepciones de Odoo define algunos tipos de excepción básicos. Esos tipos son entendidos por la capa RPC y será tratado como un "error del servidor".

Algunas de la excepciones del propio Odoo:

- Error de acceso/contraseña:

exception odoo.exceptions.AccessDenied(message='Acceso denegado')[source]

- Error de derechos de acceso:

exception odoo.exceptions.AccessError(msg)[source]

- Registro(s) perdido(s):

exception odoo.exceptions.MissingError(msg)[source]

- Error genérico gestionado por el cliente:

exception odoo.exceptions.UserError(msg)[source]

- Violación de las restricciones de Python:

exception odoo.exceptions.ValidationError(msg)[source]

B) LANZAR EXCEPCIONES EN ODOO:

La declaración raise permite forzar a que ocurra una excepción específica. Raise tiene un único argumento que indica la excepción que se va a generar. Este argumento debe ser o una instancia de excepción, o una clase de excepción (una clase que hereda de Exception).

Ejemplo en Odoo:

```
from odoo.exceptions import ValidationError #Hacemos el import de la excepcion
@api.constrains('campo') #función que lanza una excepción cuando el valor del campo sea mayor
def _validar_campo(self):  # de 20
    for record in self:
        if record.campo > 20:
            raise ValidationError("El valor del campo es muy grande: %s" % record.campo)
```

➢ CONTROL DE ERRORES CON FUNCIONES:

En Odoo podemos controlar errores con funciones de Python:

```
@api.onchange('valor1', 'valor2') #Se produce la comprobación cuando cambian alguno de los valores
                                  #pasados en el @api de Odoo
def _control_valores(self):
    self.valor2 = self.valor1 * self.valor2
        return {
        'warning': {
            'title': "Algo no ha ido bien",
            'message': "Hay un problema con los valores",
        }
    }
```

2.Estructura de un modulo en odoo.

Todo los que se pueda hacer para modificar Odoo se hace con módulos, los módulos son estructuras de carpetas y ficheros que aportan funcionalidad a Odoo. Veremos esta estructura con detalle y así nos resultará mas fácil crear y modificar elementos de Odoo.

A) Composición de un módulo:

Los módulos de Odoo amplían o modifican partes de Modelo-Vista-Controlador. De este modo, un módulo puede tener:

- Objetos de negocio: Son lo que llamamos modelo, son clases en python utilizando las características y sintaxis propias del ORM de Odoo.
- Ficheros de datos: Son archivos hechos en XML que definen vistas, carga de datos, acciones, menús,...
- Controladores web: Es la parte que se encarga de gestionar las peticiones del navegadores web que actua como cliente.
- Datos estáticos: Iconos del módulo, elementos CSS, o javascript que utiliza la interficie web para mostrar la información.

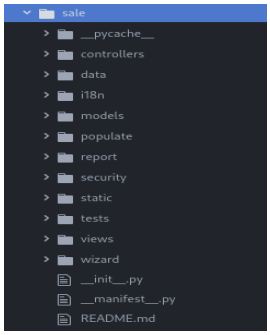
La estructura de ficheros de un módulo:

Un módulo de python se declara en un fichero de manifiesto que da información sobre el módulo, que hace el módulo, de que otros módulos depende y cómo se tiene que instalar o actualizar.

Un módulo es un paquete de Python que necesita un __init__.py para instanciar todos los ficheros python. El resto de ficheros se cargan en __manifest__.py

- **Ejemplo de la estructura de un módulo de Odoo.**

Vamos a ver la estructura con un módulo de los que lleva Odoo, por ejemplo el módulo sale de Odoo, podemos ver que lleva todo tipo de elementos organizados por carpetas, CSS, iconos, XML,...:

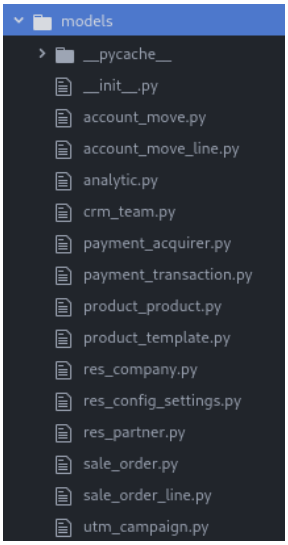


Podemos ver la estructura de carpetas que presenta, vamos a ver el fichero __init__.py

```
# -*- coding: utf-8 -*-
# Part of Odoo. See LICENSE file for full copyright and licensing details.

from . import models
from . import controllers
from . import report
from . import wizard
from . import populate
```

Podemos ver que se importan todas las carpetas donde pueda haver contenido python.

Accedemos a la carpeta models:



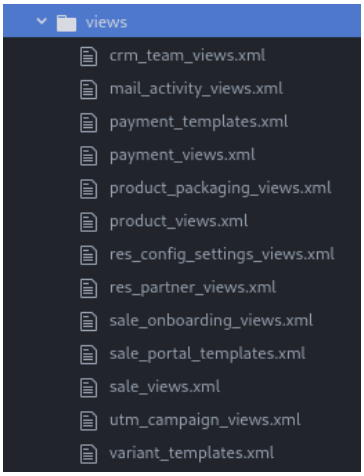
Vemos que también contiene un fichero __init__.py donde se cargan todos los .py de la carpeta.

```python
# -*- coding: utf-8 -*-
# Part of Odoo. See LICENSE file for full copyright and licensing details.

from . import analytic
from . import account_move
from . import account_move_line
from . import crm_team
from . import payment_acquirer
from . import payment_transaction
from . import product_product
from . import product_template
from . import res_company
from . import res_config_settings
from . import res_partner
from . import sale_order
from . import sale_order_line
from . import utm_campaign
```

Tenemos los ficheros .py con las clases y funciones necesarias. Vamos a ver uno de estos ficheros.

```python
# -*- coding: utf-8 -*-
# Part of Odoo. See LICENSE file for full copyright and licensing details.
import base64

from odoo import api, fields, models, _
from odoo.modules.module import get_module_resource


class ResCompany(models.Model):
    _inherit = "res.company"

    portal_confirmation_sign = fields.Boolean(string='Online Signature', default=True)
    portal_confirmation_pay = fields.Boolean(string='Online Payment')
    quotation_validity_days = fields.Integer(default=30, string="Default Quotation Validity (Days)")

    # sale quotation onboarding
    sale_quotation_onboarding_state = fields.Selection([('not_done', "Not done"), ('just_done', "Just done"), ('done', "Done"), ('closed', "Closed")], string="S
    sale_onboarding_order_confirmation_state = fields.Selection([('not_done', "Not done"), ('just_done', "Just done"), ('done', "Done")], string="State of the
    sale_onboarding_sample_quotation_state = fields.Selection([('not_done', "Not done"), ('just_done', "Just done"), ('done', "Done")], string="State of the onb

    sale_onboarding_payment_method = fields.Selection([
        ('digital_signature', 'Sign online'),
        ('paypal', 'PayPal'),
        ('stripe', 'Stripe'),
        ('other', 'Pay with another payment acquirer'),
        ('manual', 'Manual Payment'),
    ], string="Sale onboarding selected payment method")

    @api.model
    def action_close_sale_quotation_onboarding(self):
        """ Mark the onboarding panel as closed. """
        self.env.company.sale_quotation_onboarding_state = 'closed'
```

Accedemos a la carpeta de vistas, donde estan definidas las diferentes vistas asociadas a los modelos anteriores, en ellas podemos tener los menus y submenus asociados, las acciones de ventana y las vistas.



Veamos una de ellas:

- Menus

```xml
<?xml version="1.0" encoding="utf-8"?>
<odoo>

        <!-- Top menu item -->
        <menuitem id="sale_menu_root"
            name="Sales"
            web_icon="sale_management,static/description/icon.png"
            active="False"
            sequence="30"/>

        <menuitem id="sale_order_menu"
            name="Orders"
            parent="sale_menu_root"
            sequence="2"/>
```

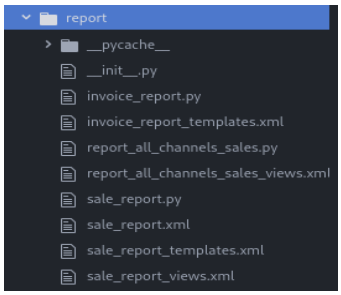
- Acciones:

```xml
<record id="product_template_action" model="ir.actions.act_window">
    <field name="name">Products</field>
    <field name="type">ir.actions.act_window</field>
    <field name="res_model">product.template</field>
    <field name="view_mode">kanban,tree,form,activity</field>
    <field name="view_id" ref="product.product_template_kanban_view"/>
    <field name="search_view_id" ref="product.product_template_search_view"/>
    <field name="context">{"search_default_filter_to_sell":1, "sale_multi_pricelist_product_template": 1}</field>
    <field name="help" type="html">
        <p class="o_view_nocontent_smiling_face">
            Create a new product
        </p><p>
            You must define a product for everything you sell or purchase,
            whether it's a storable product, a consumable or a service.
        </p>
    </field>
</record>
```

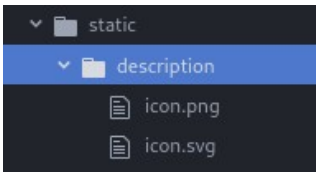
- Vistas:

```xml
<record id="product_template_form_view" model="ir.ui.view">
    <field name="name">product.template.product.website.form</field>
    <field name="model">product.template</field>
    <field name="inherit_id" ref="product.product_template_form_view"/>
    <field name="arch" type="xml">
        <xpath expr="//page[@name='sales']" position="attributes">
            <attribute name="invisible">0</attribute>
        </xpath>
    </field>
</record>
```

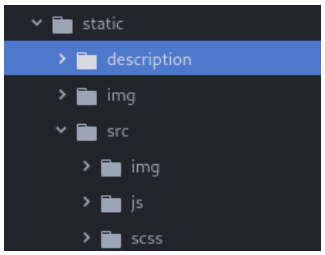
En la carpeta Report tendremos definidos los informes que se pueden generar por defecto en este módulo:



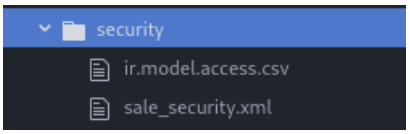
Los iconos los tenemos en la carpeta static/description:



Los ficheros CSS y javascript los encontraremos en la carpeta static/src



Los ficheros para establecer permisos y grupos para la seguridad del módulo los tenemos en la carpeta security:



Hemos visto los principales archivos y carpetas de un módulos, cuando empecemos a desarrollar nuevos elementos veremos con detalle todo aquello que necesitemos.

3.Creación de un módulo en Odoo.

Para crear un módulo en Odoo se puede hacer creando la estructura completa o utilizando una orden específica del propio Odoo que nos crea la estructura necesaria de un nuevo módulo, pero antes de crear un nuevo módulo en odoo debemos preparar el sistema para que lo podamos cargar e instalar en Odoo.

A) Preparación del servidor.

Prepararemos el sistema para la creación de nuevos módulos y detectar posible errores.

Al servidor donde tenemos el odoo crearemos una carpeta para gestionar los módulos nuevos. Por ejemplo la crearemos en /opt/odoo/modulos.

Añadir esta ruta al addons_path:

- Editar el fichero de configuración de odoo, que lo encontraremos a /etc/odoo.conf .
- A la línea addons_path, añadimos la ruta separado por una coma.

Reiniciamos el servicio de odoo:

sudo systemctl restart odoo.

Cada vez que hacemos cambios en un módulo y queremos que se vean tendremos que reiniciar el servicio de odoo.

Ver los errores que han surgido:

Si durante el proceso de creación de los módulos deja de funcionar el odoo, podemos ver lo que pasa en Odoo desde el fichero log. Si no recordáis la ruta de los logs, vais al fichero donde está creado el servicio, lo encontraréis en /etc/systemd/system y editáis el servicio de odoo y tenéis la ruta de los logs.

/var/log/odoo/odoo-server.log

En este fichero tenemos información del que pasa al odoo, errores incluidos.

B) Creación de un módulo de forma manual:

Por cada módulo hay una carpeta con el nombre del módulo. Para crear un módulo tendremos que crear una carpeta dentro de la carpeta (addons) o crear una carpeta nueva para nuestros módulos y añadirla a la ruta addons_path, y seguir los siguientes pasos:

- Crear el archivo de inicio de módulo: __init__.py (fijaos que esta raya __, son dos rayas bajas). Este archivo le dice a python que otros ficheros de python tiene que abrir.
- Crear el archivo con la descripción del módulo: __manifest__.py (para las nuevas versiones de odoo, para versiones anteriores a la 9 este ficheros se denomina __openerp__.py). El archivo tiene que contener los siguientes valores dentro de un diccionario. Este fichero le da al odoo la información del módulo.
  - Python (estructura de datos cerrada con claves {}):
  - name: nombre del módulo.
  - version: versión del módulo.
  - description: una descripción del módulo.
  - author: persona o entidad que ha desarrollado el módulo.
  - website: sitio web del módulo.
  - license: tipo de licencia del módulo (por defecto GNU / GPL). La Licencia pública general de GNU o más conocida por su nombre en inglés GPL (o simplemente las siglas del inglés GNU / GPL) es la licencia más ampliamente usada en el mundo del software y garantiza a los usuarios finales (personas, organizaciones, compañías) la libertad de usar, estudiar, compartir (copiar) y

modificar el software.

- depends: lista de módulos de los cuales depende el módulo, el módulo base es lo más utilizado puesto que en él se definen algunos datos que son necesarias las vistas, informes, etc.
- data: lista de los ficheros XML que se cargarán con la instalación del módulo.
- Installable: determina si el módulo es instalable o no.

- Crear los archivos Phyton con la definición de objetos: nombre_modulo.py, por medio de clases del lenguaje Phyton definimos el modelo y el controlador.
- Vista o vistas del objeto: número_modulo_número_objeto.xml.
- Subcarpetes informes, wizard, test, ...: Contienen archivos del tipo de objeto utilizado, en este caso, informes, asistentes y accesos directos, datos de prueba, etc.

C) Creación del módulo con scaffold:

Para crear el módulo se puede utilizar un comando propio de Odoo que crea toda la estructura del módulo de forma automática .

Desde la ubicación de la instalación de Odoo, vemos que esta la orden que arranca Odoo: odoo-bin

```
odoo@ubuntu:~/odoo$ ls
addons           debian   MANIFEST.in  README.md        setup
CONTRIBUTING.md  doc      odoo         requirements.txt  setup.cfg
COPYRIGHT        LICENSE  odoo-bin     SECURITY.md       setup.py
```

Vemos que es un fichero ejecutable:

```
odoo@ubuntu:~/odoo$ ls -l odoo-bin
-rwxrwxr-x 1 odoo odoo 180 dic  2 10:38 odoo-bin
```

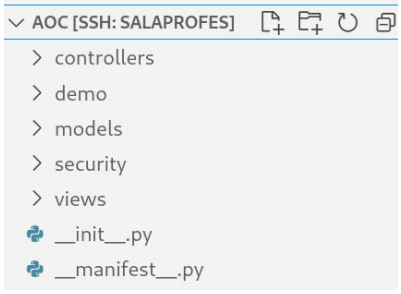
Con el usuario que tenemos para Odoo, podemos ejecutar odoo-bin con el parámetro scaffold para generar un nuevo módulo

./odoo-bin scaffold NombreModulo RutaCarpetaModulos.

Por ejemplo si queremos crear un módulo nuevo que se llame AOC y que este en la carpeta /opt/odoo/modulos ejecutariamos.

```
odoo@ubuntu:~/odoo$ ./odoo-bin scaffold AOC /opt/odoo/modulos
```

Accedemos a la ruta indicada y comprobamos que nos ha creado la estructura, podemos acceder al servidor por ssh o por la conexión creada en el IDE.



Desde VisualCode por ejemplo vemos la estructura de ficheros y directorios que nos ha creado.

Si accedemos al cliente Web de Odoo y buscaremos el módulo que acabamos de crear, este ya es instalable aunque no guarda ninguna información en el sistema ya que no hemos definido los datos.

4. La lógica de la aplicación ORM.

En este apartado veremos la lógica de los módulos de Odoo con detalle, ya sabemos que Odoo mapea sus objetos en una base de datos con ORM, para ello necesitamos los modelos y los campos. También tenemos las vistas que junto con los menús y las acciones permiten la interacción con los modelos.

A) Los modelos de Odoo:

Los modelos se crean como clases de python que extienden la clase models.TipoModelo de Odoo y que contienen los campos y los métodos para hacer funcionar el ORM. Los tipos de modelos pueden ser:

- Model: para los modelos usuales y que pasaran a ser tablas en la base de datos
- TransientModel: Sirven para guardar los datos de forma temporal, pueden almacenarse en la base de datos pero que no se guardan de forma permanente.
- AbstractModel: Para crear superclases abstractas que podrán ser utilizadas por diferentes modelos mediante la herencia.

Los modelos, al heredar de models.Model, necesitan asignar valores a algunas de sus variables, como por ejemplo _name.

Ya hemos visto que al crear un nuevo módulo con la orden scaffold nos crea toda una estructura de carpetas y archivos, una de estas carpetas llamada models es donde vamos a crear todos los archivos .py con las clases que nos crearan los diferentes modelos (es decir, tablas de la base de datos), que necesitamos para el nuevo módulo.

➢ ATRIBUTOS DEL MODELO

Las clases del modelo pueden tener atributos, veamos algunos:

- _name: Este es el identificador interno para el modelo que estamos creando. Normalmente utilizamos la nomenclatura NombreModulo.tabla. Esta variable tiene que tener un valor ya que es nombre con el que posteriormente se hará referencia al modelo cuando se definan los elementos como los menús, las acciones,…
- _description: permite poner un texto que describe el modelo
- _order: Permite ordenar los registros del modelo.

- _table: Este es el nombre de la tabla de la base de datos relacionada con el modelo. Normalmente este paso es automático y el nombre que se la da a la tabla es el nombre del modelo cambiando el punto por un guion bajo (_). Pero con este atributo podemos indicar el nombre que queramos.
- _inherit e _inherits: Este atributo se utiliza para aplicar la herencia en los modelos (lo veremos mas adelante).

Al crear un módulo nuevo crearemos tantas clases como tablas necesitemos en la base de datos.

Ejemplo de un modelo que extiende models.Model, por tanto se creará una tabla en la base de datos con nombre aoc_tabla1, sabemos que la tabla1 pertenece al módulo aoc.

```
-*- coding: utf-8 -*-
from odoo import models, fields, api
class aoc(models.Model):
    _name = 'aoc.tabla1'
    _description = 'aoc.tabla1'
```

B) Los campos de Odoo (fields).

Las elementos que se definen en los modelos son los fields o también serán las columnas de las tablas de la base de datos. Estos elementos como cualquier dato han de ser de un tipo determinado: Integer, Float, Boolean, char... o especiales como many2one, one2many, related...

➢ CAMPOS ESPECIALES.

Odoo incorpora una serie de campos reservados y necesarios para su utilización dentro del mismo. Estos campos no los tenemos que crear nosotros, se crean automáticamente:

- id: Identificador único del modelo.
- create_date: fecha de creación del módulo
- create_uid : usuario que ha creado el modelo.
- write_date : última actualización del modelo
- write_uid : usuario que realizó la última actualización.
- __last_update: Última modificación.

También existen otro tipo de campo que presentan unas propiedades especiales, veamos algunos de ellos:

- name: Este campo es el que se utilizará en Odoo para crear el Identificador Externo del modelo o este sera el campo referencia en las relaciones entre modelos.
- active: Nos dice si el elemento esta activo o no, esto nos permite ocultar elementos de la base de datos que no necesitemos.

- state: Es un campo de tipo selección y permite crear un ciclo de vida de un modelo.

➢ CAMPOS NORMALES:

Vamos a ver los fields para los datos:

- Integer: para números enteros.
- Char: Para datos tipo carácter.
- Text: Pata texto
- Date o Datetime : Campo para fechas y/o horas.
- Float: campos de valores numéricos.
- Boolean: True o False
- Html : Guarda un texto, pero se representa de manera especial en el cliente.
- Binary : Permite guardar imágenes por ejemplo. Utiliza codificación base64
- Selection: Muestra un select con las opciones que indiquemos.

Todos estos Campos tiene en común una serie de atributos:

- String: muestra el nombre del field, es el nombre que se verá en la vista.
- Required: es de tipo booleano y por defecto esta a False, si lo ponemos a True, el campo no se puede quedar vacío al poner datos en el modelo.
- Help: proporciona ayuda al usuario por rellenar el campo.
- Index: es de tipo booleano y por defecto es False, indica a Odoo que este campo será el índice de la base de datos. Si no ponemos este atributo el ORM creará un campo id.
- Invisible: indica si el campo será visible en la vista, es de tipo booleano y por defecto es Falso.
- Readonly: si el campo es readonly no se podrá modificar, es un booleano y por defecto es False.
- Default: Permite establecer un valor por defecto en el campo; es un valor estático, o una función que toma un conjunto de registros y devuelve un valor.

Hay determinados campos que presentan características propias, como los campos Selection donde definiremos los elementos para seleccionar, otros campos vienen del resultado de una función que realiza algún cálculo,…

Ejemplo:

```
class aoc(models.Model):

    _name = 'aoc.tabla1'
```

```
name = fields.Char(
    string="Name",                      # Etiqueta del campo
    compute="_funcio_calculada",        # campo calculado, necesita la función que realiza el cálculo
    store=True,                         # Si el campo se guardará
    readonly=True,                      # campo de solo lectura
    inverse="_write_name"               # Función inversa
    required=True,                      # Si el campo es requerido
    help='ayuda',                       # Ayuda para el campo
    search='_search_function',          # funcion de busqueda
 )

estado = fields.Selection([ ('0','Bueno'), ( 1','Regular'),  ('2','Malo')], string="Estado", default="0")
# Por defecto estará seleccionada la primera opción
```

- **CAMPOS RELACIONALES:**

Como sabemos las tablas en una base de datos están relacionadas unas con otras, en Odoo la relación se establece en los modelos. En Odoo existen un tipo de campo que permite definir relaciones entre los modelos, ya sean uno a muchos, muchos a uno o muchos a muchos. En el caso de las relaciones muchos a muchos, en una base de datos relacional, implica que se debe crear una tabla intermedia, pero en Odoo no hace falta hacerla ya que el propio ORM se encarga de hacer las tablas, claves y restricciones de integridad necesarias en estas relaciones. Este tipo de campo tienen un parámetro que permite definir el borrado de los datos: Cascade: Permite eliminar los registros relacionados, Restrict: Impide la eliminación de la fila referenciada en la relación y Set null: Modifica el campo a Nulo o False, cuando se elimine el elemento relacionado.

- Many2one: campo de tipo relación muchos a uno.
- One2many: Inversa del Many2one. Necesita un Many2one en la tabla relacionada. No implica datos adicionales en la base de datos y se calcula como un select en la base de datos donde el id del modelo actual coincida con el Many2one del otro modelo.
- Many2many: relación muchos a muchos. Si queremos tener más de una relación Many2many entre los dos mismos modelos, hay que utilizar la sintaxis completa donde especificamos el nombre de la relación y el nombre de las columnas de la relación de los dos modelos.

Ejemplos:

1.Tenemos un modelo coche y queremos añadirle un campo para el propietario del coche, el propietario vendrá del modelo res.partner de Odoo, de manera que un coche pertenece a un propietario pero un propietario puede tener mas de un coche. Al definir la relación en coches seria:

```
class coche(models.Model):
    _name = 'aoc.coche'
    propietario = fields.Many2one('res.partner',string=’Propietario del coche')  #Nos permitirá elegir de la tabla como un select
```

2.Tenemos dos tablas una guarda los países y otra tabla ciudades, de manera que una ciudad pertenece a un país y un país tiene muchas ciudades.

```
class pais(models.Model):
   _name = 'mod.pais'
   ciudades = fields.One2many('mod.ciudad','pais',string='Ciudades',ondelete='restrict')


class ciutat(models.Model):
   _name = 'mod.ciudad'
   pais = fields.Many2one("mod.pais", string='Pais', ondelete='restrict')
```

- **CAMPOS CALCULADOS:**

Existe un tipo de campo que se puede calcular y no necesita extraerse de la base de datos, se utiliza el parámetro compute en los atributos del campo. Si para calcular dicho campo necesitamos los valores de otros campos del modelo, debemos utilizar una api de Odoo que nos permita pasarlos a la función.

Los Decoradores:

Los decoradores modifican la forma en la que se llama una función. Entre otras cosas, modifican el contenido de self.

@api.depends() Este decorador llama a la función siempre que el campo del que depende se modifique. Por defecto, self es un recordset, por lo tanto, hay que hacer un for.

@api.model Se utiliza para funciones que afectan al modelo y no a los recordsets.

@api.constrains() Se utiliza para comprobar las constrains. Self es un recordset. Normalmente se utiliza en un form, funciona como si utilizaramos self directamente.

@api.onchange() Se ejecuta cada vez que modificamos el field indicado en la vista. En este caso self será un singleton.

Ejemplo de campo calculado (utilizamos el decorador @api.depends):

```
precio=fields.Float(string="Precio")
ejemplares=fields.Integer(string="Ejemplares")
importetotal=fields.Float(string="ImporteTotal",compute="_importetotal")
# Este campo no sera guardado en la base de datos, y siempre se re calculara al querer mostrarlo
  @api.depends('ejemplares','precio')
# El decorador @api.depends() indica que se llamará a la función cada vez que se modifiquen los campos ejemplares y precio
#Si no lo pusiéramos, solo se actualizaría al recargar la acción.
 def _importetotal(self):
     for r in self
        r.importetotal = r.ejemplares*r.precio
# El for recorre el self, que contiene  un recordset con todos los elementos del modelo  en una vista tree o si es un form, será un singleton.
#Esta función recibe un parámetro implícito self, que es como el this de java, realiza una iteración sobre el parámetro (es obligatorio el for ya que este método podría ser llamado para una lista de registros y no para un único registro). Dentro del for ya hacemos el cálculo que necesitamos.
```

Los campos calculados son de solo lectura por defecto. Para poder poner valores en un campo computado, podemos utilizar la función inversa. Esta función invierte el cálculo y establece el valor del campo.

Ejemplo:

```
documento = fields.Char(compute='_leer_documento', inverse='_escrivir_documento')
def _leer_documento(self):
    for record in self:
        with open(record.get_document_path) as f:
            record.documento = f.read()
def _escrivir_documento(self):
    for record in self:
        if not record.documento: continue
        with open(record.get_document_path()) as f:
            f.write(record.documento)
```

C) Las acciones y los menús

El cliente web de Odoo contiene unos menús y estos al ser accionados muestran otros menús y/o las pantallas del programa, esto ocurre porque hemos hecho una acción, las acciones básicamente tienen:

- type: El tipo de acción que es. Cuando la definimos en el XML, el type no hay que especificarlo, puesto que lo indica el modelo en que se guarda.
- name: El nombre, que puede ser mostrado en la pantalla o no.

Las acciones y los menús se declaran en ficheros de datos en XML, las acciones pueden ser llamadas de tres formas:

- Haciendo clic en un menú.
- Haciendo clic en botones de las vistas.
- Como acciones contextuales en los objetos.

1. Acciones tipo window: son ir.actions.act_window, se definen en un record de xml y pueden ser llamadas desde el elemento menuitem.

Ejemplo:

```
<record model="ir.actions.act_window" id="Id_Accion">
        <field name="name">Nombre de la acción</field>
        <field name="res_model">Modelo que abrirá</field>
        <field name="view_mode">Vistas relacionadas con el modelo</field>
        <field name="help" type="html">
          <p class="oe_view_nocontent_create">Ayuda sobre la ación </p>
        </field>
  </record>
```

Aquí llamamos a la acción:

<menuitem id="Id_del_menú" name="Nombre del menú" action="Id_Accion" />

Odoo permite declarar acciones también como respuesta de funciones. Un ejemplo de esto son los acciones para los wizards. De hecho, podemos hacer que un botón devuelva una acción y así poder abrir una vista nueva (ya veremos esta parte en los wizards).

2. Acciones tipo Server: Las acciones tipo server funcionan en un modelo base y pueden ser ejecutadas automáticamente o con el menú contextual de acción que se podemos ver en la parte de arriba de la vista.

Las acciones que puede hacer un server action son:

- Ejecutar un código python en el servidor.
- Crear un nuevo record.
- Escribir en un record existente.

Los menús en Odoo se crean en un fichero xml, con la estructura siguiente:

<menuitem id="Id_del_menú" name="Nombre del menú" action="Id_Accion" parent="menú superior" groups="grupos de seguridad />

D) Las vistas de Odoo

Las vistas son la manera en la que se representan los modelos. En caso de que no declaramos las vistas, se pueden referenciar por su tipo y Odoo generará una vista de lista o formulario estándar para poder ver los registros de cada modelo. Las vistas se guardan en el modelo ir.ui.view.

```
<record id="MODEL_view_TYPE" model="ir.ui.view">
  <field name="name">NAME</field>
  <field name="model">MODEL</field>
  <field name="arch" type="xml">
    <VIEW_TYPE>
      <VIEW_SPECIFICATIONS/>
    </VIEW_TYPE>
  </field>
</record>
```

En Odoo existen diferentes tipos de vistas, veremos algunas de ellas: tree, form, kanban, search, calendar, graph,...

La vista Tree, lista o árbol es una de las mas comunes Junto con la de formulario, estos tipos de vista suelen ser la que tienen los modelos por defecto, pero existen algunos modelos que por defecto tiene una vista kanban o Calendario.

5. Informes.

Odoo nos permite crear ficheros que mostraran información de los datos guardados, de una manera clara y precisa. Estos ficheros de datos se llaman informes. Odoo lleva incorporados informes propios (ya sean de módulos o en general), pero también nos permite crear otros nuevos que nos permitan ver la información de manera personalizada. Un ejemplo de informe serian las facturas del módulo de facturación.

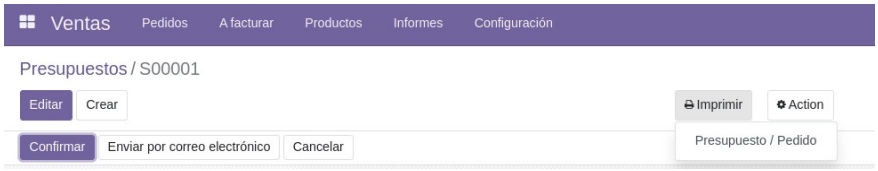
A) Informes.

Los informes también llamados reports en Odoo. El nuevo motor de reports utiliza una combinación de QWeb, BootStrap y Wkhtmltopdf.

Un informe consta de dos elementos:

- Un registro en la base de datos en el modelo: ir.actions.report.xml con los parámetros básicos del informe
- Una vista Qweb para el contenido.

Los informes de Odoo por defecto, consisten en generar una web en lenguaje HTML/QWeb, con formato similar a una vista. Una vez creado, y con la librería wkhtmltopdf (que debemos tener instalada y configurada en nuestro servidor de Odoo), permite la conversión a pdf. Los informes creados de este modo pueden imprimirse. Siempre que haya un informe disponible se puede ver un botón para generarlo.



Este seria un informe predeterminado del módulos Ventas de Odoo, que al hacer clic en el mismo nos generará un documento en pdf (siempre que tengamos la librería wkhtmltopdf).

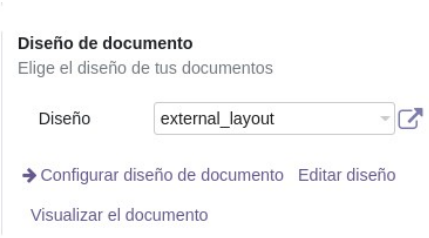
Al darle a imprimir nos descarga un documento con la estructura del informe:

B) Plantillas:

Todos los informes generados por Odoo contienen un encabezamiento y un pie de página genéricos y comunes que incorporan información sobre la empresa y la fecha de generación del informe, external_layout añadirá la cabecera y el pie de página por

defecto.

Podemos modificar las plantillas desde el cliente Web de Odoo, en Ajustes>Opciones generales . En el apartado Empresas:



C) Creación de Informes.

Para generar un nuevo informe, este estará definido en un fichero xml (La ruta del report deberemos añadirla al fichero __manifest__.py, en el apartado data).

Contenido del archivo xml, primero se crean los datos del informe: Los reports simplifican con la etiqueta report la creación de una acción de tipo report. Automáticamente sitúan un botón arriba del tree o form para imprimir.

```
<report
    /* id único del informe */
    id="nombre_informe"
  /* Título del informe se utiliza como nombre de archivo si no se especifica   print_report_name. */
    string="Titulo del informe"
    /* Modelo de datos q utiliza para el informe*/
    model="modelo sobre el que se generará el informe"
    /* Tipo de renderizado,qweb-pdf per a PDF, qweb-html per a HTML*/
    report_type="qweb-pdf"
    /* Para identificarlo en el menú de Odoo */
    name="nombre_modulo.nombre_informe"
  />
```

A continuación se generarán los datos que formaran el informe:

```
/*id: Este identificador de la plantilla debe coincidir con el name del report.*/
<template id="nombre_plantilla">
/* En este hacemos una llamada al contenedor */
    <t t-call="web.html_container">
/*docs el la lista de objetos a imprimir como el self*/
     <t t-foreach="docs" t-as="o">
/* Llamada a la plantilla que les da forma*/
      <t t-call="web.external_layout">
 /* Estas secciones se repiten en todas las páginas para poner contenido
        <div class="header"></div>
        <div class="page"></div>
```

```
    <div class="footer"></div>
  </t>
 </t>
</template>
```

El cuerpo del PDF será el contenido dentro del . El id de la plantilla debe ser el nombre especificado en la declaración del informe. Al tratarse de una plantilla QWeb, se puede acceder a todos los campos de los objetos docs recibidos por la plantilla.

1. EL LENGUAJE QWEB:

QWeb es el motor de plantillas de Odoo. Los elementos son etiquetas XML que empiezan por t-

- t-field: Para mostrar el contenido de un field
- t-if: Para hacer condicionales.

```
<t t-if = "condicion">
    <p> Test </ p>
</ t>
```

- t-foreach: Para hacer bucles.

```
<t t-foreach="[1, 2, 3]" t-as="i">
    <p><t t-esc="i"/></p>
</t>
```

Podréis encontrar mas información sobre esto en la documentación oficial de Odoo.

6.Herencia

El framework OpenObject facilita el mecanismo de la herencia, esta técnica sirve para que se puedan adaptar módulos ya existentes y así se puede garantizar que las actualizaciones de los módulos no puedan deshacer los cambios realizados.

La herencia se puede aplicar en los tres componentes del patrón MVC:

- En el modelo: permite la ampliación de los modelos existentes o diseñar nuevas clases a partir de las existentes.
- En la vista: permite modificar el comportamiento de vistas existentes o diseñar nuevas vistas.
- En el controlador: permite sobrescribir los métodos existentes o diseñar otros nuevos.

1. Herencia en el modelo:

La herencia en el modelo existen de tres tipos diferentes, existe la herencia tradicional y esta se pueden realizar de dos formas diferente y la herencia por prototipo.

1. Herencia de clase: Es el tipo de herencia mas simple que hay, en este caso se sustituye la clase original por la nueva clase y en esta se añaden aquellos elementos necesarios, ya sean atributos o funciones. También se pueden sobrescribir los métodos de la clase original. En postgres permanece la misma tabla, pero ampliada con los nuevos atributos.
2. Herencia por prototipo: Herencia simple, en este tipo de herencia se aprovecha la original, a la cual se le añade funcionalidad (atributos i/o métodos), también se pueden sobrescribir los métodos de la clase original. Pero a diferencia de la anterior en este caso se crea una nueva tabla en postgres y las vistas del original ya no nos sirven, tendríamos que crearlas de nuevo.
3. Herencia por delegación: Permite herencia simple o múltiple. Se crea una nueva clase que hereda de una o mas clases y puede contener elementos de todas las clases de las cuales hereda. Se deben crear las vistas correspondientes y en postgres se mapea mas de una tabla (una para los nuevos elementos y el resto de elementos los encontraremos en las tablas originales)

Dominios:

Si queremos que el campo heredado muestre solo algunos elementos podemos poner un dominio que nos filtre la información, se puede poner tanto en las vistas como en los modelos:

En la vista:

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

En el modelo:

```
campo = fields.Many2one('modulo.modelo', string="Infor", domain=[('campo2', '=', True)])
```

- **COMO SE DEFINE LA HERENCIA:**

- Herencia de clase: Se utiliza el atributo _inherit en la definición de la nueva clase Python:_inherit = obj. El nombre de la nueva clase tiene que continuar siendo el mismo que el de la clase original:_name = obj

```
class Original(models.Model):
    _name = 'original.zero'
    _description = 'Clase original para la herencia'
    name = fields.Char(default="A")

class herencia(models.Model):
    _name = 'original.zero' // este atributo no hace falta ponerlo
    _inherit = 'original.zero'
    _description = 'Clase que hereda de la original'
    description = fields.Char(default="Nuevo atributo")
```

- Herencia por prototipo: Se utiliza el atributo _inherit en la definición de la nueva clase Python:_inherit = obj. Debemos darle nombre a la nueva clase.

```
class Original(models.Model):
    _name = 'original.zero'
    _description = 'Clase original para la herencia'
    name = fields.Char()

    def call(self):
        return self.check("modelo Original")

    def check(self, s):
        return "This is {} record {}".format(s, self.name)

class herencia(models.Model):
    _name = 'heredada.uno' // este atributo no hace falta ponerlo
    _inherit = 'original.zero'
    _description = 'Clase que hereda de la original'

   def call(self):
        return self.check("modelo Heredado")
```

- Herencia por delegación: Se utiliza el atributo _inherits en la definición de la nueva clase Python: _inherits = ... Hay que indicar el nombre de la nueva clase.

```
class clase1(models.Model):
    _name = 'herencia.uno'
    _description = 'primera clase'
    campo1 = fields.Float(string='campo uno')

class clase2(models.Model):
    _name = 'herencia.dos'
    _description = 'segunda clase'
    campo2 = fields.Char(string='campo dos')

class clase3(models.Model):
    _name = 'herencia .tres'
    _description = 'Tercera clase que hereda'

    _inherits = {
        'herencia.uno': 'campo1_id',
        'herencia.dos': 'campo2_id',
    }

    name = fields.Char(string='Nambre')
    campo1_id = fields.Many2one('herencia.uno', required=True, ondelete="cascade")
    campo2_id = fields.Many2one('herencia.dos', required=True, ondelete="cascade")
```

2. Herencia en la vista:

La herencia de clase posibilita continuar utilizando las vistas definidas sobre el objeto padre, pero muchas veces nos interesa modificar dichas vistas, como por ejemplo para poder añadir los atributos que hemos añadido con la herencia o reemplazar

otros.

Para esto existe la herencia en la vista y se define:

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

En el caso que el el objeto del cual se hereda no pertenezca al mismo módulo, en la referencia se añade el nombre del módulo, es una buena práctica ponerlo por defecto:

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

En las vistas heredadas podemos añadir, borrar o reemplazar elementos.

Existe una elemento xpath que permite localizar los otros elementos de las vistas para modificarlas, borrarlas o añadir otros.

```
<xpath expr="//Elemento[@name='nombreDelElementobuscado']" position="Posicion">
```

Donde Elemento: field, page,target,... y Posicion: after, before,move, replace...

Ejemplo1: añadir una pagina nueva con un nuevo campo

```
<xpath expr="//page[@name='internal_notes']" position="after">
<page name="Nombre nueva pagina" string="Nombre">
 <group>
  <field name="nuevo campo" />
 </group>
</page>
</xpath>
```

Ejemplo2: reemplazar un campo por otro

```
<field name="arch" type="xml">
  <field name="campo" position="replace">
    <field name="nevo_campo" ... />
  </field>
 </field>
```

Ejemplo3:

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

3. Herencia en el controlador:

Este tipo de herencia lo aplicamos de forma inconsciente, en el momento en que

queremos sobrescribir los métodos de la capa ORM de OpenObject en el diseño de muchos módulos.

En Python utilizamos la función super() para invocar el método de la clase superior cuando vamos a sobrescribir en una clase heredada, en lugar de utilizar la sintaxis nombreClasseBase.metodo(self…).

La herencia en el controlador se manifiesta únicamente cuando hay que sobrescribir alguno de los métodos del objeto del cual se hereda y para hacerlo adecuadamente hay que tener en cuenta que el método sobrescrito en el objeto heredado:

- Si queremos sustituir el método del objeto base sin aprovechar ninguna funcionalidad: el método del objeto heredado no invoca el método sobrescrito.
- Si queremos aprovechar la funcionalidad del método del objeto base: el método del objeto heredado invoca el método sobrescrito.

Ejemplo:

```
class ejemplo(models.Model):
   _inherit = 'res.partner'
  metodo = fields.Boolean(string='Ha pasado el metodo super')

   @api.model
   def create(self, values):
      # Anulamos la función original para el modelo res.partner
      record = super(res_partner, self).create(values)

     # Cambiar los valores de una variable en la función super
      record['metodo'] = True
      print 'Paso la funcion. metodo valor: ' + str(record['metodo'])

   # Devolver el registro para que se apliquen los cambios y se almacene todo.
     return record
```

7. Wizards

Muchas veces necesitamos realizar tareas repetitivas o simplemente queremos automatizar algunas de estas tareas, el ERP Odoo permite automatizar o lanzar tareas mediante una herramienta llamada wizard.

A) Definición de Wizard:

Un Wizard es un asistente que permite interactuar con los elementos de Odoo, un asistente no es mas que un modelo en Odoo que extiende la clase TransientModel en lugar de Model. Esta clase TransientModel cumple las siguientes características:

- Los registros del wizard no están destinados a ser persistentes; se eliminan automáticamente de la base de datos después de un cierto tiempo. Por eso se llaman transitorios.
- Los modelos de wizrd a partir de Odoo 14 necesitan permisos de acceso explícitos.

- Los registros de los wizards pueden tener referencias Many2one o Many2many con el registros de los modelos normales, pero no al contrario.
- Los registros de los modelos normales pueden tener One2many a Wizards, pero cada cierto tiempo se eliminan.

B) Creación de Wizards en Odoo:

Hay diferentes formas de lanzar un Wizard:

- Se puede crear un botón en una vista o desde el menú de la parte de arriba de la vista que lanza al Wizard, mediante una acción. El propio wizard puede ser llamado de formas diversas:
  - Por una acción ya preexistente en la base de datos con %()d y un botón de tipo acción.
  - Por un acción ya preexistente que tenga binding_model y por tanto aparezca en el menú de arriba de una vista en ese mismo modelo.
  - Por una acción generada en Python y devuelto por una función.

Ciclo de un wizard:

- La acción, en los wizards, suele abrir una ventana modal (target="new") donde se muestran algunos campos del TransientModel.
- La ventana contiene un formulario que suele tener un botón para enviar, crear o cualquier otro tipo de acción y también pueden contener el botón Cancelar que tiene su sintaxis específica.
- En caso de tener un wizard mas complejo, en el cual se pueden llenar campos tipo Many2many u One2many, podríamos necesitar más transientModels para hacer relaciones. Debemos tener en cuenta que no se pueden hacer relaciones con modelos normales.
- Finalmente, el wizard acabará creando o modificando algunos modelos permanentes de la base de datos, para ello se crea una función que puede devolver una acción para mostrar las instancias creadas o para refrescar la vista que lo ha llamado.
- En ocasiones, necesitamos que el wizard obtenga información de quien lo ha llamado. Obtendremos el active_id (campo que tienen todos los modelos en Odoo) del modelo que lo ha llamado con self._context.get('active_id') dentro del modelo del wizard.

C) EStructura de un wizard:

Para empezar se debe crear el modelo de wizard, que no es mas que una clase en python que hereda de TransientModel:

```
from odoo import models, fields, api

class Wizard(models.TransientModel):

   _name = 'modulo.wizard'

# Retorna el modelo que esta abierto

    def _perdefecte(self):

      return self.env['modulo.modelo'].browse(self._context.get('active_id'))

#campos que relacionan con el modelo original

   campo1 = fields.Many2one('modulo.modelo',string="Modelo", required=True)

  campo2 = fields.Many2many('modelo', string="Datos")

#Función que será llamada desde la vista del wizard ( su nombre coincide con el nombre del #botón de la vista del wizard)

   def launch(self):

     funcion_wizard()

     return {}
```

Debemos crear una vista que se corresponda con el modelo de wizard:

```
<record model="ir.ui.view" id="wizard_view">
        <field name="name">wizard.ejemplo</field>
        <field name="model">modulo.wizard</field>
        <field name="arch" type="xml">
           <form string="Ejemplo Wizard">
              <group>
                 <field name="campo1"/>
                 <field name="campo2"/>
               </group>
             <footer>

#Aqui creamos los botones que lanzan la función del wizard y el de cancelar
                 <button name="launch" type="object"   string="lanzar" class="oe_highlight"/>
                 <button special="cancel" string="Cancelar"/>
              </footer>
           </form>
        </field>
     </record>
#Esta es la acción que lanza el wizard
        <record id="lanzar_wizard" model="ir.actions.act_window">
        <field name="name">Lanzar wizard</field>
        <field name="res_model">modulo.wizard</field>
        <field name="view_mode">form</field>

#En este caso se lanza el wizard en una ventana emergente
          <field name="target">new</field>

#binding_model es el modelo donde se puede lanzar el wizard. Con esto solo ya  aparece en el menú superior de acciones.
          <field name="binding_model_id" ref="model_wizard_ejemplo"/>
```

Si quisiéramos lanzar el wizard desde un botón y no desde el menú superior, debemos crear un botón, Observamos la sintaxis del name, que coincide con el nombre de la acción siempre que el botón sea de tipo action. Este botón lo implementaremos en la vista desde la que se llamará al wizard.

<button name="%(lanzar_wizard)d" type="action" string="Lanzar wizard" class="oe_highlight" />

D) Algunos métodos del ORM que permiten modificar, añadir o eliminar datos desde los wizards.

- Search(): A partir de un domain de Odoo, proporciona un recordset con todos los elementos que coinciden con el criterio.

  self.search([('campo', '=', True)

- create(): Muestra un recordset a partir de una definición de varios fields:

  self.create({'name': "New Name"})

- write(): Escribe en los campos fields dentro de todos los elementos del recordset, no devuelve nada.

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

  Escribir en un many2many: La manera más sencilla es pasar una lista de ids. Pero si ya existen elementos antes, necesitamos unos códigos especiales:

Por ejemplo:

self.sesiones = [(4,s.id)]

self.write({'sesiones':[(4,s.id)]})

- browse(): A partir de una lista de ids, devuelve un recordset.

  self.browse([1, 3, 5])

- exists(): Devuelve si un recordset en concreto todavía está en la base de datos.

  if not record.exists():

  raise Exception("The record has been deleted")

- ensure_one(): Se asegura de que el record en concreto sea un singleton.
- Read(): Se trata de un método de bajo nivel para leer un field en concreto de los records. Es mejor utilizar browse()
- sorted(key=None, reverso=False) Devuelve el recordset ordenado por un criterio.
- name_get() Devuelve el nombre que tendrá el record cuando sea referenciado externamente. Es el valor por defecto del field display_name. Este método, por defecto, muestra el field name si está. Se puede sobrescribir para mostrar otro campo o mezcla de ellos.

- copy() Crea una copia del singleton y permite aportar nuevos valores para los fields de la copia.

Expresiones: (para ref y eval)

(2, ID) Corte la relación de enlace entre los datos y elimine este registro (llama al método de desvinculación).

(3, ID) Corta la relación de enlace entre los datos del maestro y el esclavo, pero no elimine este registro

(4, ID) añade un registro existente de id al conjunto. No se puede utilizar en One2many.

(5,_,_) Eliminar todos los registros del conjunto, es decir, lo que equivale a utilizar el comando 3 en cada registro explícitamente.

(0, 0, (valores)) Crea un nuevo registro basado en el valor de los valores

(1, ID, {valores}) Registro de actualización (escriba el valor de los valores)

(6, 0, [IDs]}) Reemplaza el registro original con el registro en IDs (equivalente a ejecutar (5) primero y ejecutar (4, ID))

8.Carga de datos:

Es importante poder recuperar datos o cargar datos, Odoo permite la exportación y la carga de datos a o desde archivos.

A) Exportar e importar datos en Odoo:

Muchas veces necesitamos sacar o poner datos en nuestro ERP, vamos a ver como podemos exportar e importar datos en Odoo

- **EXPORTAR DATOS**

Con Odoo, puede exportar los valores de cualquier campo en cualquier registro. Para ello, active la vista de lista en los elementos que necesitan ser exportados, haga clic en Acción, y, a continuación, en Exportar.

Al exportar nos aparece una nueva ventana, para seleccionar los campos que queremos exportar y en que tipo de archivo los vamos a exportar.

A la parte izquierda de la imagen estan todos los campos posibles y a la parte derecha tenemos el formato de fichero y los campos que se exportaran por defecto, podemos añadir nuevos campos desde la parte izquierda y eliminar los que aparecen por defecto, haciendo click en la papelera correspondiente.

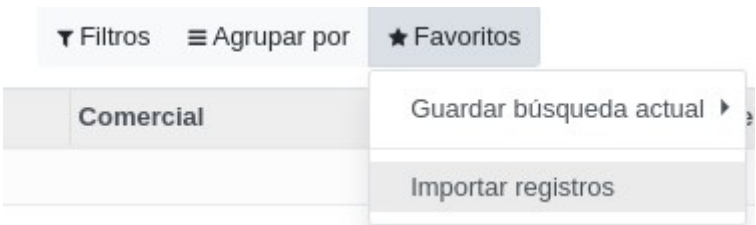
Al exportar, puede elegir entre dos formatos: .csv y .xls. Con .csv, los elementos se separan con una coma, mientras que .xls contiene información sobre todas las hojas de trabajo de un archivo, incluyendo tanto el contenido como el formato.

Para los informes recurrentes, puede ser interesante guardar los preajustes de exportación. Seleccione todas las que necesite y haga clic en la barra de plantillas. Allí, haz clic en Nueva plantilla y dale un nombre. La próxima vez que necesite exportar la misma lista, simplemente seleccione la plantilla relacionada.

- **IMPORTAR DATOS:**

Para importar datos en Odoo debemos tener unos ficheros bastante específicos, ya que deben seguir la estructura pedida desde Odoo. Los ficheros deben ser .csv o .xls (.xlsx)

Para importar, debemos seleccionar la vista del elemento al cual queremos poner datos y desde favoritos >Importar registros :



Nos abre una nueva ventana y cargaremos el fichero desde

Debemos generar una plantilla con la estructura de datos a gestionar, para que al importar estos lo hagan correctamente. Desde Importar registros, en la nueva ventana que aparece, nos permite descargar una plantilla que nos servirá para poner los datos que después importaremos.

B) Ficheros de datos:

Cuando hacemos un módulo de Odoo, se pueden definir datos que se guardarán en la base de datos. Estos datos pueden ser necesarias para el funcionamiento del módulo, de demostración o incluso parte de la vista, dichos datos se cargan desde un fichero xml con una estructura determinada.
Si queremos que el contenido del archivo de datos se aplique solo una vez, podemos especificarlo con una etiqueta en Odoo "noupdate" establecida en 1.

Si se espera que parte de los datos del archivo se aplique una vez, puede colocar esta parte del archivo en un dominio <data noupdate=1>, si ponemos el valor 0 siempre sobrescribirá los datos existentes al actualizar el módulo.

```
<odoo>
   <data noupdate="1">
      <!-- Solo aquello que se aplicará una vez -->
    </data>
    <data>
   <!-- Aquello que se aplicará cada vez que se actualice el módulo-->
   </data>
</odoo>
```

La estructura del fichero xml seria la siguiente:

```
<? Xml version = "1.0" encoding = "UTF-8"?>
<odoo>
  <data noupdate = "1">
     <record id = "idRecurso" model = "modelo1">
       <field name = "campo1"> valor </ field>
       <field name = "campo2"> valor </ field>
            ...
     </ record>
     <record id = "idRecurso" model = "modelo2">
       <field name = "campo1"> valor </ field>
```

```
        <field name = "campo2"> valor </ field>
        ...
      </ record>
     ...
   </ data>
</ odoo>
```

La etiqueta function llama a un método en un modelo, con los parámetros proporcionados. Tiene dos parámetros obligatorios model y name que especifican respectivamente el modelo y el nombre del método a llamar.
Los parámetros pueden proporcionarse mediante eval (debe evaluarse una secuencia de parámetros para llamar al método).

```
<? Xml version = "1.0" encoding = "UTF-8"?>
<odoo>
  <data noupdate = "1">
     <record id = "idRecurso" model = "modelo1">
       <field name = "campo1"> valor </ field>
       <field name = "campo2"> valor </ field>
            ...
     </ Record>
    <function model="modelo" name="funcionModelo"  eval="[[('parametro1', '=' ,'condicion' )]]"/>

    <record id = "idRecurso" model = "modelo2">
       <Field name = "campo1"> valor </ field>
       <Field name = "campo2"> valor </ field>
        ...
     </ record>
    ...
  </ data>
</ odoo>
```

El nombre del fichero debe cargarse dentro de __manifest__.py en el apartado data, pero si son datos de demostración se cargan en el apartado demo.

9.Usuarios y permisos.

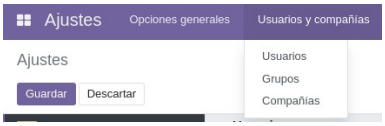
Odoo proporciona un mecanismo para gestionar o restringir el acceso a los datos.

Dicho mecanismo está vinculado a usuarios específicos a través de grupos: un usuario pertenece a cualquier número de grupos, y los mecanismos de seguridad están asociados a los grupos.

A) Grupos en Odoo

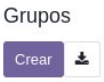
Los grupos se crean como registros normales en el modelo res.groups, y se les concede acceso al menú a través de las definiciones del mismo. Sin embargo, incluso sin un menú, los objetos pueden ser accesibles indirectamente, por lo que los permisos reales a nivel de objeto (leer, escribir, crear, borrar) deben ser definidos para los grupos, normalmente se introducen a través de archivos CSV dentro de los módulos. También es posible restringir el acceso a campos específicos en una vista u objeto utilizando el atributo de grupos en el campo correspondiente.

Desde el cliente Web podemos acceder a los grupos:



Si accedemos a grupos, observamos los que hay ya creados en el sistema:

Desde aquí podemos crear nuevos grupos



Podemos crear nuevos grupos en los propios módulos, mediante ficheros xml que se llaman security.xml. Este fichero debe cargarse dentro de __manifest__.py en el apartado data.

El fichero security.xml debe seguir una plantilla, dicha plantilla incluye un elemento “record” por cada una de las definiciones que puede contener el fichero y que será diferente según el tipo de definición (grupo o usuario, menú, regla asignada a un grupo).

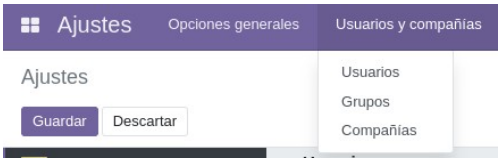
```
<?xml version="1.0" encoding="utf_8"?>
 <odoo>
   <data noupdate="1">
     <record id="idGrup" model="res.groups">
        <field name="name">NombreGrupo</field>
      </record>
   </data>
</odoo>
```

El atributo noupdate del elemento data con el valor “1” para indicar que cuando se actualice el módulo no se instale el esquema de seguridad que incorpora el módulo, puesto que podría sobrescribir el esquema configurado por el administrador de la empresa, y el valor “0” para que se instale siempre, sobrescribiendo el esquema existente. En caso de que haya partes del esquema de seguridad que no se tengan que sobrescribir y otras que sí, se separan en dos elementos data diferentes, uno con noupdate=“1” y el otro con noupdate=“0”

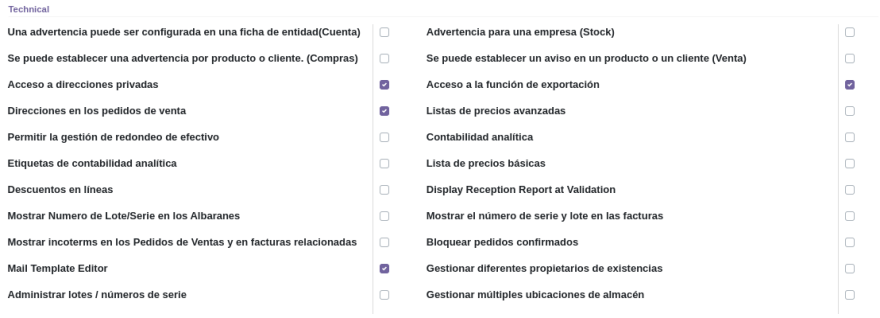
B) Incluir los usuarios en los grupos

Para asociar usuarios a los grupos podemos hacerlo por el cliente Web o por el fichero xml correspondiente a los grupos.

Desde el cliente Web:Accedemos a los usuarios



Accedemos a usuarios y seleccionamos el usuario que queremos añadir a un grupo. Editamos el usuario y seleccionamos el grupo o grupos a los cuales queremos que pertenezca el usuario.



Desde el propio módulo, accedemos al fichero security.xml y añadimos un nuevo elemento para incluir el usuario al grupo, podemos hacerlo de dos maneras:

- Añadiendo un “record” del modelo “res.users”, indicando el grupo al que queramos añadir al usuario.
- Añadiendo al grupo un campo con el nombre “users”.
- El atributo name=“category_id” es para ubicar el grupo por debajo de una categoría, que tiene que estar definida en el mismo módulo o en otro y que hay que indicar al atributo ref.
- El atributo name=“implied_ids” es para indicar que la pertenencia al grupo supone la pertenencia automática a los grupos indicados en el atributo eval.

```
<?xml version="1.0" encoding="utf_8"?>
 <odoo>
 <data noupdate="1">
    <record id="idGrup" model="res.groups">
       <field name="name">NombreGrupo</field>
        <field name="category_id" ref="..."/>
        <field name="implied_ids" eval="..."/>
      <field name="users" eval="..."/>
     </record>
   <record id="idUser" model="res.users">
   </record>
  </data>
</odoo>
```

inclusión de un usuario en un grupo con eval=“[(num,ref(‘…’))]”), función que es propia de Odoo, y sirve para añadir registros en un campo One2many o Many2many, se pueden utilizar también en funciones del ORM.

Ejemplo para ocultar un menú:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>
   <record id="grupo_informe_menu" model="res.groups">
      <field name="name">Ver Menu Informes </field>
```

```
    <field name="users" eval="[(4, ref('base.user_admin'))]"/>
  </record>
<record model="ir.ui.menu" id="account.menu_finance_reports">
<field name="groups_id" eval="[(6,0,[ref('account_security_reports.grupo_informe_menu')])]"/>
  </record>
</odoo>
```

C) Establecer permisos a los grupos y reglas de registro:

Se pueden establecer y ver los permisos y las reglas de registro sobre los elementos de Odoo desde el cliente Web, accedemos a los grupos, desde allí seleccionamos un grupo y en las pestañas permisos de acceso y reglas de registro podremos visualizarlo todo y añadir nuevos elementos.

También podemos ver y crear nuevos registros y permisos desde Ajustes>Técnico

- **PERMISOS DESDE FICHEROS**

Para establecer los permisos sobre los diferentes elementos de Odoo, podemos hacerlo desde el propio cliente Web o desde un archivo xml que se llamará ir.model.access.csv y que incluiremos en el archivo __manifest__.py, quedando este de la forma siguiente (el orden importa ya que el segundo fichero necesita los grupos creados en el primer fichero)

```
'data': [
    'security/security.xml',
    'security/ir.model.access.csv',
```

ir.model.access.csv: Empezamos a escribir por la línea 2, la primera la dejamos tal cual y estableceremos diferentes reglas empezando por la id de la regla. En la línea 1 se indica que significa cada campo:

id: La identificación del nuevo registro. Odoo por cada línea de este archivo genera un nuevo registro para el modelo ir.modelo.access.

name: El nombre que le asignamos a la regla, tendría que ser descriptivo para que sea mas fácil identificarlo.

modelo_id: Este es el nombre del modelo, el que colocaron en el parámetro _name. Antes poner como prefijo modelo_ después el nombre del modelo sustituyendo los puntos por guiones bajos.

group_id: id: Aquí va el grupo a que le darán los permisos, como prefijo va el nombre del módulo, que es el nombre de la carpeta que contiene los archivos del módulo.

Los permisos: para tener el permiso correspondiente este debe estar a 1, si esta a 0 no tiene el permiso.

perm_read: permisos de lectura.

perm_write: permisos de escritura.

perm_create: permisos de creación.

perm_unlink: Permisos de borrado.

Ejemplo:

id,name,modelo_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_modulo_modelo,lmodulo.modelo,modelo_modulo_modelo,modulo.grupo,1,1,1,1

- **REGLAS DE REGISTRO:**

Una regla de registro restringe los derechos de acceso a un subconjunto de registros de un modelo dado. Una regla es un registro del modelo ir.rule, y está asociada a un modelo, un número de grupos (campo many2many), los permisos a los que se aplica la restricción y un dominio. El dominio especifica a qué registros se limitan los derechos de acceso.

Una regla de registro tiene

- Un modelo sobre el que se aplica un conjunto de permisos (por ejemplo, si se establece perm_read, la regla sólo se comprobará al leer un registro).
- Un conjunto de grupos de usuarios a los que se aplica la regla, si no se especifica ningún grupo la regla es global.
- Un dominio utilizado para comprobar si un registro dado coincide con la regla y si es accesible o no. El dominio se evalúa con dos variables en contexto:
  - usuario es el registro del usuario actual
  - hora es el módulo de tiempo

Las reglas globales y las reglas de grupo (reglas restringidas a grupos específicos

frente a grupos que se aplican a todos los usuarios) se utilizan de forma muy diferente:

A) Las reglas globales son sustractivas, deben coincidir todas para que un registro sea accesible.

B) Las reglas de grupo son aditivas, si alguna de ellas coincide (y todas las reglas globales coinciden), el registro es accesible.

Esto significa que la primera regla de grupo restringe el acceso, pero cualquier otra regla de grupo lo amplía, mientras que las reglas globales sólo pueden restringir el acceso (o no tienen efecto).

Ejemplo:

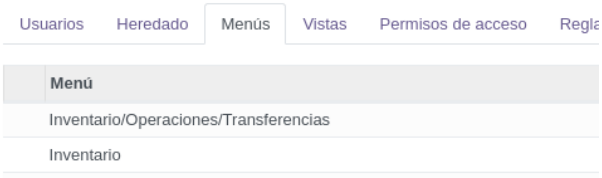
```
<record id="IdRegla" model="ir.rule">
   <field name="name">Nombre</field>
   <field name="model_id" ref="modulo.modelo"/>
   <field name="groups" eval="[(4, ref('modulo.grupo'))]"/>
   <field name="perm_read" eval="0"/>
   <field name="perm_write" eval="0"/>
   <field name="perm_create" eval="0"/>
   <field name="perm_unlink" eval="1" />
   <field name="domain_force">[('state','=','cancel')]</field>
</record>
```

D) Aplicar la seguridad sobre los menús de Odoo y los wizards:

Los permisos que se han establecido sobre los modelos, los podemos aplicar a los menus y los wizards para que estos sean visibles y accesibles desde el cliente Web, para ello deberemos añadir en la etiqueta de creación del elemento de menú los diferentes grupos para que los permisos se apliquen.

```
<menuitem name="NombreMenu" id="Id_menu" parent="Id_parent" groups="grupo1,grupo2,..." action="id_accion"/>
```

También se pueden establecer desde el cliente Web, accedemos a los grupos y desde allí seleccionamos un grupo y en la pestaña Menús, añadimos el menú al cual queremos aplicar los permisos.



En el caso de los wizards, se establecen los permisos tal y como lo hemos hecho sobre un modelo cualquiera, en el fichero ir.model.access.csv:

id_regla,nombre regla,model_modulo_modelo,grupo,1,1,1,1 (en este caso tiene todos los permisos y tendremos en cuenta que el modelo es el del wizard)