MINI APUNTES ODOO 1

Sistemas de Gestión Empresarial

19 DE NOVIEMBRE DE 2024

ERIK AMO TOQUERO

Contenido

Introducción a Odoo

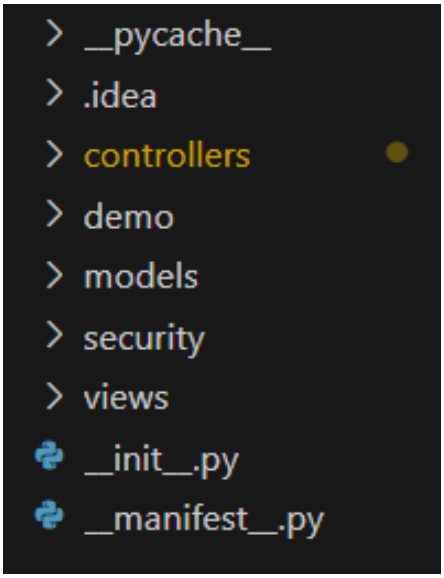
Odoo es un sistema gestor de empresas de distintos tamaños muy popular debido a su fácil personalización y su variedad de aplicaciones para gestionar diversas áreas de negocios.

A nosotros nos interesa Odoo por su **modularidad**: Se divide en distintos módulos independientes entre ellos y encargados de gestionar distinta información.

Contenido de un Módulo

Dentro de un módulo de Odoo ya creado, hay estas carpetas:

- __pycache__ -> Cachés del módulo
- controllers -> Controladores de requests de http
- demo -> Plantillas
- models -> Contiene los modelos de la aplicación, dígase las tablas
- security -> Contiene un archivo que modifica el acceso de los distintos usuarios al módulo
- views -> Contiene las vistas de la aplicación, dígase la interfaz
- __init__.py -> Contiene lo que tiene que iniciarse al iniciar el módulo
- __manifest__.py -> Contiene datos sobre el módulo: Su nombre, los datos que carga...



Edición de un Módulo

En esta parte, voy a centrarme en lo necesario para modificar el módulo hasta conseguir el producto que buscamos:

Modelo

El modelo es el contenido que va a aparecer en una tabla, un objeto que guarda las distintas propiedades que van a aparecer plasmadas. En un modelo pueden aparecer:

- **_name** y **_description**: Guardan el identificador del modelo en Odoo.
- **Atributos (fields):** Los atributos son las columnas de las tablas, hay distintos tipos de fields
- **Métodos (acciones):** Se activan desde botones y realizan acciones desde las tablas

_name y _description son únicos y obligatorios en cada modelo, y el fichero deberá tener una apariencia tal que así

```
from odoo import models, fields, api
class pelicula(models.Model):
    _name = 'modulo.modelo '
    _description = 'modulo.modelo '

    atributo1 = fields...
    atributo2 = fields...
    atributo3 = fields...
    atributo4 = fields...

    def _get_code(self):
        [...]
```

Fields

Los fields son los atributos que tiene un modelo. Tienen unos atributos en común:

- **required (boolean):** Dice si es o no necesario introducir un valor
- **string (String):** Define un nombre para la columna en las vistas
- **compute (String):** Define el nombre de una función para completarlo
- **help (String):** Define un mensaje de error
- **readonly (boolean):** Indica si se puede escribir o no

Hay distintos tipos de field dependiendo de cuál sea el valor que buscamos:

Char

Es una cadena de caracteres de una única línea. Se utiliza para nombres o enlaces normalmente.

```
codigo_peli = fields.Char(string="Código", compute = "_get_code")
```

Text

Es una cadena de caracteres multilínea. Se usa para descripciones o párrafos detallados

```
description = fields.Text(string="Descripción")
```

Integer

Marca un valor entero en la clase.

```
id = fields.Integer()
```

Float

Marca un valor de coma flotante. Se puede utilizar para precios o porcentajes

```
precio = fields.Double()
```

Date

Marca una fecha

```
film_date = fields.Date(string="Fecha de grabación")
```

Datetime

Marca una fecha y una hora

```
start_date = fields.Datetime(string="Fecha de inicio")
```

Selection

Es un combobox: Da distintas opciones en un campo y el usuario debe elegir una. Hay dos nuevas propiedades:

- selection: Guarda una lista de pares clave-valor (ambos en String, independientemente de su tipo), de las cuales en el combobox aparecen los valores y la aplicación guarda de forma interna la clave
- default: La clave que aparece por defecto marcada

```
language = fields.Selection(selection=[('español', 'Español'), ('ingles',
 'Inglés'), ('aleman', 'Alemán'), ('frances', 'Francés')],
default='español', required = True, string="Idioma")
```

(En este, por defecto es el Español)

Binary

Guarda valores binarios. Se utiliza para subir archivos o imágenes (Para este último caso, funciona mejor Image)

```
image = fields.Image(string="Imagen")
fichero = fields.Binary(string="Archivo")
```

Vista

Las vistas es la parte gráfica de los módulos, es la forma en la que los van a ver y manipular los usuarios.

Los archivos de las vistas se tienen que encontrar en el archivo Manifest.

```
# always loaded
'data': [
    'security/ir.model.access.csv',
    'views/views.xml',
    'views/templates.xml',
    'views/pelicula.xml',
    'views/genero.xml',
    'views/tecnica.xml',
],
```

Hemos dado distintos de modelos:

- **Form:** Es el formulario que se ve cuando entra a editar el modelo

```
<record model="ir.ui.view" id="vista_modulo_modelo_form">
  <field name="name">vista_modulo_modelo_form</field>
  <field name="model">modulo.modelo</field>
  <field name="arch" type="xml">
    <form string="formulario_modulo " >
      <sheet>
        <group name="nombre_grupo">
          <field name="atributo1"/>
          <field name="atributo2"/>
          <field name="atributo3"/>
        </group>
      </sheet>
    </form>
  </field>
</record>
```

- **Tree**: Es la vista principal en forma de tabla

```
<record model="ir.ui.view" id="vista_modulo_modelo_tree">
  <field name="name">vista_modulo_modelo_tree</field>
  <field name="model">modulo.modelo</field>
  <field name="arch" type="xml">
    <tree>
      <field name="atributo1"/>
      <field name="atributo2"/>
      <field name="atributo3"/>
    </tree>
  </field>
</record>
```

- **Kanban**: Es otro modelo de vista principal, donde se ve el contenido en forma de tarjetas. Funciona con marcas de HTML con un par de pasos más, usando clases o_kanban.

```
<record id="vista_modulo_modelo_kanban" model="ir.ui.view">
  <field name="name">Modelo</field>
  <field name="model">filmotecaerik.pelicula</field>
  <field name="arch" type="xml">
    <kanban>
      <field name="atributo1"/>
      <field name="atributo2 "/>
      <field name="atributo3 "/>
      <templates>
        <t t-name="kanban-box">
          <div t-attf-class="oe_kanban_global_click">
            <div class="o_kanban_image">
              <img alt="Imagen"
              t-att-src="kanban_image(modulo.modelo, 'atributo_img', record.id.raw_value)"/>
            </div>
            <div class="o_kanban_details">
              <strong class="o_kanban_record_title">
                <field name="name"/>
              </strong>
              <div t-if="record.atriuto1.value">
                <t t-esc="record.atributo1.value"/>
              </div>
            </div>
          </div>
        </t>
      </templates>
    </kanban>
  </field>
</record>
```

- Action: Acciones de las vistas: Todas las vistas a las que el usuario puede acceder

```
<record model="ir.actions.act_window" id="accion_modulo_modelo_form">
      <field name="name">Listado</field>
      <field name="type">ir.actions.act_window</field>
      <field name="res_model">modulo.modelo</field>
      <field name="view_mode">tree,form,kanban</field>
      <field name="help" type="html">
        <p class="oe_view_nocontent_create">
          Modelo
        </p>
        <p> Click <strong> 'Crear' </strong> para añadir nuevos elementos
         </p>
      </field>
    </record>
```

Menús

Dentro de las vistas, se introduce un valor llamado **menuitem**. Este valor sirve para establecer una jerarquía de menús para acceder a Actions. La rama principal es lo que se ve al dar a "Aplicaciones", y de ahí van pasando a ser submenús y submenús de submenús.

```
<menuitem name="Módulo" id="menu_modulo_raiz"/>
<menuitem name="Apartado" id="menu_modulo_modelo" parent="
menu_modulo_raiz "/>
<menuitem name="Subapartado" id="menu_modulo_modelo_submenu" parent="
menu_modulo_modelo" action="accion_modulo_modelo_form"/>
```

El action lo tiene que tener el eslabón más pequeño de todos.

Botones

Los botones son elementos de las vistas que permiten clicarlos para ejecutar una función interna de su modelo (La función tiene que estar creada en el py del modelo):

```
<button name="funcion " string="Nombre" class="oe_highlight" type="object"/>
```

(Class es el tipo de botón, string es el texto y name es el nombre de la función)

Seguridad

Dentro del archivo de seguridad ir.model.access.csv, van las distintas directivas para que los grupos de usuarios puedan acceder a distintos modelos:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_modulo_modelo,modulo.modelo,model_modulo_modelo,base.group_user,1,1,1,1
```

Los campos son:

- Id: El nombre de la directiva
- Nombre: el nombre del modelo al que afecta
- ID de Modelo:
- Id de Grupo
- Permisos: Lectura, escritura, creación y borrado (0 desactivado, 1 activado)

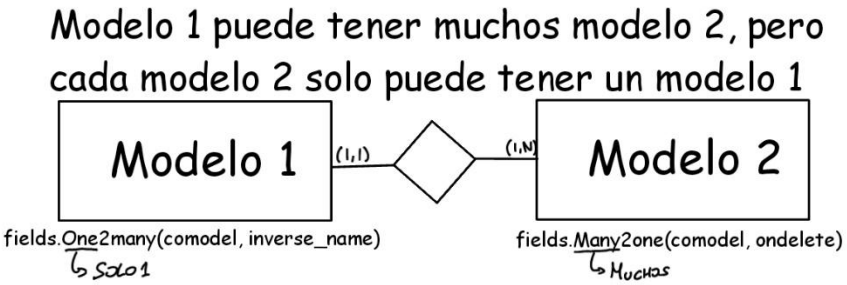
El fichero ir.model.access.csv tiene que aparecer activado en el fichero Manifest

Relaciones

Los distintos modelos se pueden relacionar entre ellos (como en una base de datos normal y corriente). Hay 2 tipos de relaciones: 1:N y N:N

Relaciones 1:N

Es una relación que se puede describir como “En la tabla 1 hay muchas ocurrencias de la tabla 2, pero en la tabla 2 solo puede haber una ocurrencia de la tabla 1”.



Dependiendo de qué modelo estemos editando, el field será de un tipo u otro:

- One2many(comodel, inverse) -> Va en la tabla de la relación (1,1)

```
tabla2_id = fields.One2many(comodel_name="modulo.modelo2 ", inverse_name="tabla1_id")
```

- Many2one(comodel, ondelete) -> Va en la tabla de la relación (1,N)

```
tabla1_id = fields.Many2one("modulo.modelo1", ondelete = "cascade")
```

Relaciones N:N

Es una relación que se puede describir como “En la tabla 1 hay muchas ocurrencias de la tabla 2 y en la tabla 2 hay muchas ocurrencias de la tabla 1”.

Ambas tablas utilizan el atributo fields.Many2many(comodel, relation, column1, column2), donde:

- comodel es el _name del modelo con el que le vas a relacionar
- relation es el nombre de la relación, **igual en ambos modelos**
- column1 es el nombre del atributo Many2many del modelo actual
- column2 es el nombre del atributo Many2many otro modelo

```
#En la tabla modelo1
modelo2_ids = fields.Many2many("modulo.modelo2", relation = "mod1_mod2",
column1 = "modelo2_ids", column2 = "modelo1_ids")
#En la tabla modelo2
modelo1_ids = fields.Many2many("modulo.modelo1", relation = "mod1_mod2",
column1 = "modelo1_ids", column2 = "modelo2_ids")
```

API

Todo el asunto de los modelos más adelante sirve también para poder crear una API con los datos que se consiga de las tablas. Utiliza la librería de Python de Odoo http y http.response. Utiliza de forma interna las peticiones de búsqueda de PostgreSQL

```
class pelicula_controller(http.Controller):

    @http.route('/api/modelo, auth='public', method=['GET'], csrf=False)
    def get_peliculas(self, **kw):
        try:
            peliculas = http.request.env['modulo.modelo'].sudo().search_read([],
['codigo_peli', 'name', 'description'])
            res = json.dumps(peliculas, ensure_ascii=False).encode('utf-8')
            return Response(res, content_type='application/json;charset=utf-8',
status=200)
        except Exception as e:
            return Response(json.dumps({'error': str(e)}),
content_type='application/json;charset=utf-8', status=505)
```

Vista Search

Las vistas search permiten filtrar y agrupar conjuntos de modelos según ciertos campos. Estas vistas se caracterizan por tener de campo principal (donde las tree tienen tree y demás), el campo “search”. Tienen la etiqueta filter con las propiedades name (nombre del grupo), string (Lo que pone), domain (condición), [context (su agrupación) (solo para grupos)]

```
<record model="ir.ui.view" id="vista_filmotecaerik_pelicula_search">
  <field name="name">vista_modulo_modelo_search</field>
  <field name="model">modulo.modelo</field>
  <field name="arch" type="xml">
    <search string="Filtrar">
      <field name="atributo1"/>
      <field name="fecha1"/>
      <group expand="0" string="Group By"><!--Agrupar grupos de filtros-->
        <filter name="groupby_atributo1" string="Atributo1" domain="[(
'atributo1, '=', 'Hola')]" context="{'group_by':'atributo1'}" help="Agrupar
por si es español"/>
      </group>
      <filter name="filter_by_fecha" string="Fecha" domain="[( 'fecha1', '=',
'2024')]"/>
    </search>
  </field>
</record>
```

CRUD en Módulos – ORM de Odoo

El ORM (Object-Relational Mapping) permite interactuar con la base de datos de forma fácil y eficiente por medio de clases y objetos de Python, facilitando así las consultas SQL.

Características:

- **Mapeo de objetos:** Las tablas de PosgreSQL se corresponden con cada modelo en Python. Se puede trabajar con ello así: result=self.env[‘modulo.modelo’].search([(‘clave’ = ‘valor’)])
    - **self.env[modelo]:** Indica el modelo en el que va a buscar
    - **.search():** Método de búsqueda de un objeto específico
    - **result:** Resultado
    - **result.clave:** Retorna el valor de la columna de nombre clave;
- **Manipulación de datos:** Permite hacer las operaciones CRUD desde Python:
    - **C – Create =** self.env[modelo].create(objeto)

```
modelo1:Modelo1 = Modelo1(atributo1 = "Hola")
self.env['modulo.modelo'].create(modelo1)
```

    - **R – Lectura =** self.env[modelo].search([condición])

```
modelo2:Modelo1 = self.env['modulo.modelo'].search([('atributo1', '=', 'Hola')])
```

    - **U – Actualización =** self.env[modelo].search([condición]).write({json})

```
modelo2.write({"atributo1":"Adios"})
```

    - **D – Borrado =** self.env[modelo].search([condición]).unlink()

```
modelo2.unlink()
```

- **Control de Acceso:** El ORM facilita el acceso a los datos, gestionando los permisos en la carpeta security
- **Optimización de consultas:** El ORM optimiza el rendimiento de la Base de datos por medio del almacenamiento en Caché
- **Restricciones:** Permite validar y restringir acciones que se apliquen al guardar datos.