

UD 8. Conceptos avanzados de Odoo ERP

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.

ÍNDICE

1.	INTRODUCCIÓN	3
2.	AGREGAR BOTONES EN LAS VISTAS	3
3.	VISTAS AVANZADAS: SEARCH	5
4.	ORM EN ODOO	7
5.	API: INTEGRACIÓN CON OTROS SISTEMAS	10

1. INTRODUCCIÓN

En esta unidad vamos a ver algunos conceptos avanzados relacionados con Odoo:

2. AGREGAR BOTONES EN LAS VISTAS

Vamos a hacerlo con el módulo filmoteca → copiar el módulo filmoteca en vuestra carpeta odoo_dev (lo tenéis en el Teams)

Estos pasos son opcionales; los vamos a hacer para diferenciar este módulo del que ya teníamos:

En manifest.py modificar el nombre del módulo:

```
'name': "filmoteca pruebas",
```

Y en la vista filmoteca, en el menú raíz también:

```
<menuitem name="Filmoteca Pruebas" id="menu_filmoteca_raiz"/>
```

Vamos a añadir un botón en el formulario película, para que cambie el valor del campo color (tipo Boolean). Este sería el código necesario para añadir un botón en una vista:

```
<form string="formulario_pelicula" >
  <sheet>
    <div class="oe_button_box" name="button_box">
      <button name="" type="object" class="oe_stat_button"
        string="" icon=""
      />
    </div>
```

Es necesario indicar el name; type y class se dejan con los valores que aparecen; string hay que poner un nombre y asociar un icono (página fontawesome.com): <https://fontawesome.com/icons>

Los iconos son una definición de una clase css. Dentro de icon hay que especificar el nombre que aparece debajo del icono. Así quedaría el código final del botón:

```
<form string="formulario_pelicula" >
  <sheet>
    <div class="oe_button_box" name="button_box">
      <button name="toggle_color" type="object" class="oe_stat_button"
        string="Película color" icon="fa-solid fa-brush"
      />
    </div>
```

Ahora hay que copiar el name del boton y en el modelo pelicula hacer lo siguiente: crear una función con el name del botón; lo que hace es cambiar el estado del campo color (True <-> False). Este código va debajo de los campos relaciones:

```
def toggle_color(self):
    self.color = not self.color
```

Probarlo:

	Película Col...
Is Spanish ?	<input checked="" type="checkbox"/>
Image ?	UPLOAD YOUR FILE
Language ?	Español
Opinion ?	Regular
Color ?	<input checked="" type="checkbox"/>
Género ?	Suspense

Si en un formulario no se ve una imagen en un campo de tipo imagen; añadir lo siguiente:

```
<group name="group_right">  
  <field name="is_spanish"/>  
  <field name="image" widget="image"/>  
</group>
```

Is Spanish ? ☒

Image ?



3. VISTAS AVANZADAS: SEARCH

Vista Kanban: se usa para incluir imágenes y datos. En nuestro caso, ya lo tenemos implementado en el módulo filmoteca (vista Kanban de película)

Vamos a hacer una vista search dentro de filmoteca.xml → debajo del código de la vista kanban, copiamos el esqueleto que tenemos en el fichero ejemplo_search.xml y rellenamos los datos correctos.

Para definir el diseño de la vista: podemos indicar los campos por los que se puede filtrar, se pueden generar filtros predefinidos y agrupaciones predefinidas.

En primer lugar hay que indicar los campos por los que filtrar: name, is_spanish, film_date...

Luego está la parte de las agrupaciones predefinidas: group by. A domain se le pasa un array de tuplas y se puede filtrar determinados registros (ponemos los que is_spanish = True). En context se pasa un diccionario

Ahora hay que definir los filtrados predefinidos: filter by →

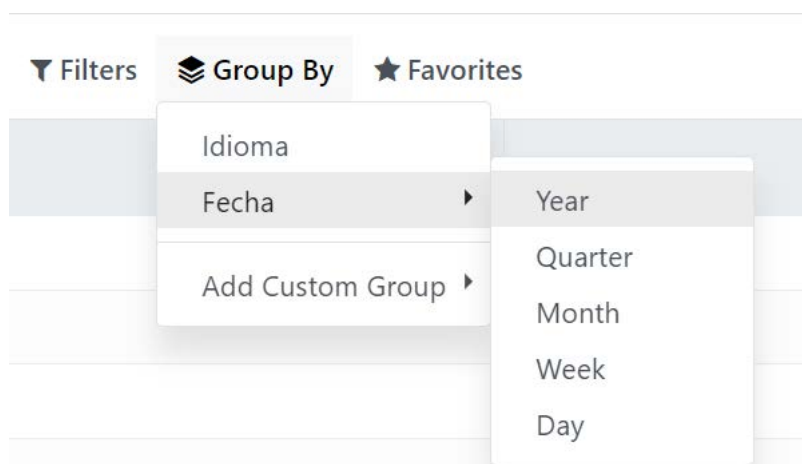
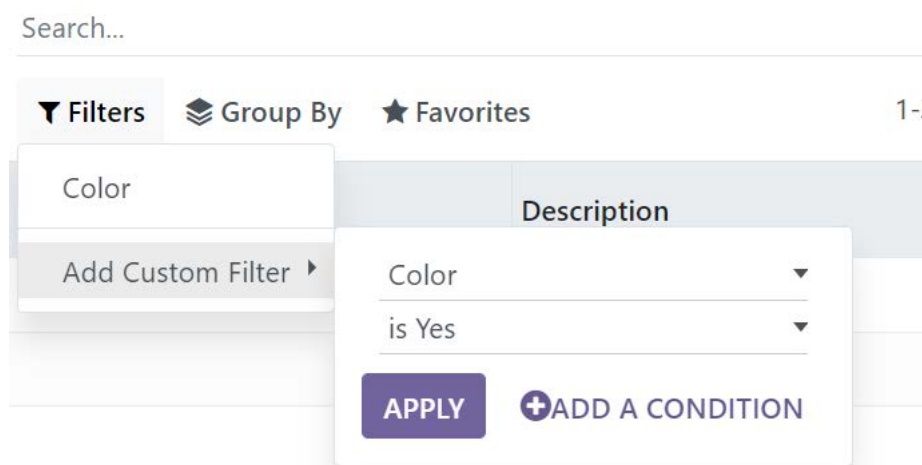
La vista search quedaría de esta forma:

```

<record model="ir.ui.view" id="vista_filmoteca_pelicula_search">
  <field name="name">vista_filmoteca_pelicula_search</field>
  <field name="model">filmoteca.pelicula</field>
  <field name="arch" type="xml">
    <search string="Filtrar películas">
      <field name="name"/>
      <field name="is_spanish"/>
      <field name="film_date"/>
      <field name="color"/>
      <group expand="0" string="Group By">
        <filter name="groupby_is_spanish" string="Idioma" domain="(['is_spanish','=', 'True'])"
          context="{ 'group_by': 'is_spanish' }">
          help="Agrupar por idioma"/>
        <filter name="groupby_film_date" string="Fecha" context="{ 'group_by': 'film_date' }">
          help="Agrupar por fecha"/>
      </group>
      <filter name="filter_by_color" string="Color" domain="(['color','=', 'True'])"
        help="Películas en color"/>
    </search>
  </field>
</record>

```

Y se vería de esta forma:



4. ORM EN ODOO

ORM: paradigma de POO; permite tratar las consultas a la BBDD como si fueran objetos → no hay que escribir las sentencias SQL, sino que se pueden utilizar funciones del ORM para tratar con las entidades de BBDD.

Para probarlo, vamos a agregar botones para operar con las entidades de BBDD:

Seguimos con el módulo filmoteca; vamos a añadir los botones en la vista tree de películas.

Primero hay que escribir el código necesario para crear un botón en la vista tree de películas:

```
<record model="ir.ui.view" id="vista_filmoteca_pelicula_tree">
  <field name="name">vista_filmoteca_pelicula_tree</field>
  <field name="model">filmoteca.pelicula</field>
  <field name="arch" type="xml">
    <tree>
      <field name="name"/>
      <field name="film_date"/>
      <field name="description"/>
      <button name="f_create" string="Crear" class="oe_highlight" type="object"/>
    </tree>
  </field>
</record>
```

Y luego definir la función del botón en el modelo película:

```
def toggle_color(self):
    self.color = not self.color

def f_create(self):
    pelicula = {
        "name": "Prueba ORM",
        "color": True,
        "genero_id": 1,
        "start_date": str(datetime.date(2022,8,8))
    }
    print(pelicula)
    self.env['filmoteca.pelicula'].create(pelicula)
```

Como estamos añadiendo una fecha: start_date, usamos datetime → arriba hay que importarlo:

import datetime


Para agregar una nueva entidad, se hace a través de un JSON (en python se definen como diccionarios); para almacenarla en BBDD: usamos el método create del ORM: self.env

Probarlo → actualizar todo → Ahora aparece el botón Crear → hacer clic sobre él. Se ha creado una entidad directamente a través del ORM

<input type="checkbox"/> Lo que el viento se llevo	Crear
<input type="checkbox"/> Prueba ORM	Crear

Code ? SUSPENSE_4
Nombre ? Prueba ORM
Description ?
Film Date ?
Duration ? 0
Start Date ? 08/08/2022 02:00:00
End Date ? 08/08/2022 02:00:00
Técnicas ?

Name	Description	Photo
Add a line		

Is Spanish ? ☐
Image ? 
Language ? Español
Opinion ? Regular
Color ? ☒
Género ? Suspense

Vamos a crear otro botón (Buscar/Editar) en la vista tree de películas:

```
<button name="f_create" string="Crear" class="oe_highlight" type="object"/>
<button name="f_search_update" string="Buscar/Editar" class="oe_highlight" type="object"/>
```

Vamos al modelo y hacemos la implementación de la función f_search_update → para hacer las búsquedas se emplean 2 métodos: search al que se le pasa un array de tuplas donde se especifica lo que queremos buscar y otro método es: browse al que se le pasa un array con una serie de identificadores que queremos recuperar:

```
def f_search_update(self):
    pelicula = self.env['filmoteca.pelicula'].search([('name', '=', 'Prueba ORM')])
    print('search()', pelicula, pelicula.name)
```

Probarlo:

<input type="checkbox"/> Memento Mori	12/04/2023	Crear	Buscar/Editar
<input type="checkbox"/> Buscando a Nemo		Crear	Buscar/Editar
<input type="checkbox"/> Lo que el viento se llevo		Crear	Buscar/Editar
<input type="checkbox"/> Prueba ORM		Crear	Buscar/Editar

Aparece el botón Buscar/Editar y al ejecutarlo, el resultado se ve en el log de Docker.

Vamos a ver el método browse (para probarlo, introducir un id que exista en vuestra tabla; podéis verlo desde pgadmin):

```
def f_search_update(self):
    pelicula = self.env['filmoteca.pelicula'].search([('name', '=', 'Memento Mori')])
    print('search()', pelicula, pelicula.name)

    pelicula_b = self.env['filmoteca.pelicula'].browse([3])
    print('browse()', pelicula_b, pelicula_b.name)
```

El resultado se ve en el log de Docker

Vamos a ver cómo se actualiza un campo: con el método write, al que se le pasa un JSON (diccionario de python):

```
def f_search_update(self):
    pelicula = self.env['filmoteca.pelicula'].search([('name', '=', 'Memento Mori')])
    print('search()', pelicula, pelicula.name)

    # pelicula_b = self.env['filmoteca.pelicula'].browse([3])
    # print('browse()', pelicula_b, pelicula_b.name)

    pelicula.write({
        "name": "Memento Mori EDITADO"
    })
```

Si lo probamos, vemos que funciona, al hacer clic sobre Buscar/Editar, se modifica en campo name de Memento Mori:

<input type="checkbox"/> Nombre	Film Date	Description	
<input type="checkbox"/> Memento Mori EDITADO	12/04/2023		Crear Buscar/Editar
<input type="checkbox"/> Buscando a Nemo			Crear Buscar/Editar
<input type="checkbox"/> Lo que el viento se llevo			Crear Buscar/Editar
<input type="checkbox"/> Prueba ORM			Crear Buscar/Editar
<input type="checkbox"/> Prueba ORM			Crear Buscar/Editar

Ahora vamos a ver cómo borrar un registro de la base de datos; primero creamos la siguiente línea en el fichero pelicula.xml:

```
</field name="description"/>
<button name="f_create" string="Crear" class="oe_hightlight" type="object"/>
<button name="f_search_update" string="Buscar/Editar" class="oe_hightlight" type="object"/>
<button name="f_delete" string="Eliminar" class="oe_hightlight" type="object"/>
```

Ahora vamos al modelo pelicula.py a implementar la función f_delete:

```
def f_delete(self):
    pelicula = self.env['filmoteca.pelicula'].browse([1])
    pelicula.unlink()
```

Y al probarlo, vemos que se ha borrado el primer registro de la BBDD de películas: Memento Mori EDITADO:

<input type="checkbox"/> Buscando a Nemo	Crear	Buscar/Editar	Eliminar
<input type="checkbox"/> Lo que el viento se llevo	Crear	Buscar/Editar	Eliminar
<input type="checkbox"/> Prueba ORM	Crear	Buscar/Editar	Eliminar
<input type="checkbox"/> Prueba ORM	Crear	Buscar/Editar	Eliminar

Para comprobar las modificaciones, se puede hacer desde pgadmin y vemos que se ha creado el registro y borrado lo que hemos indicado.

5. API: INTEGRACIÓN CON OTROS SISTEMAS

Vamos a ver cómo sincronizar Odoo con otros sistemas; para ello, vamos a publicar los datos en un sistema y que otros los consuman. Vamos a hacerlo a través de una API. Usaremos JSON; porque con Python se puede operar con ello de forma muy simple.

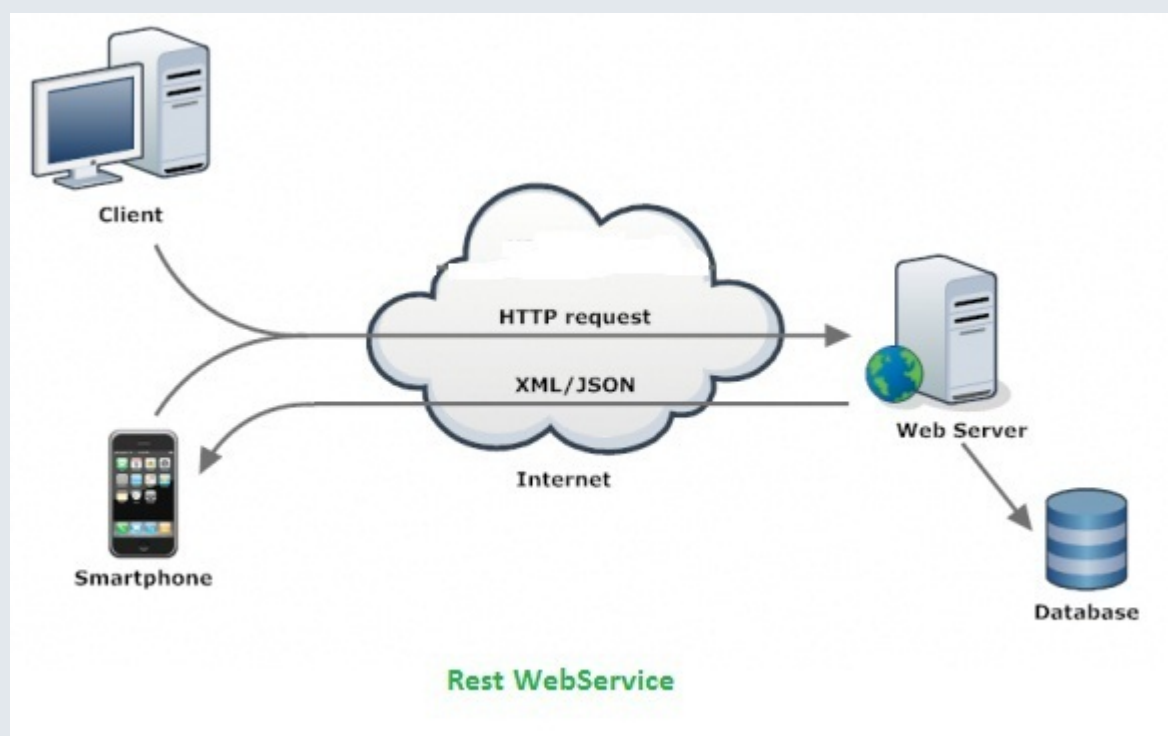
Un servicio web es una aplicación que se encuentra en el lado servidor y permite que otra aplicación cliente conecte con ella a través de Internet para el intercambio de información utilizando el protocolo HTTP.

Una de las principales características de los servicios web es que no es necesario que ambas aplicaciones (servidor y cliente) estén escritas en el mismo lenguaje, lo que hace que la interoperabilidad sea máxima. Por ejemplo, podríamos crear un servicio web en Python y utilizarlo conectándonos desde una aplicación móvil con Android, desde otra aplicación programada en Java o incluso desde otro servicio web escrito con .NET.

Además, utilizan el protocolo HTTP para el intercambio de información, lo que significa que la

conexión se establece por el puerto 80 que es el mismo que utilizan los navegadores y que es prácticamente seguro que se encuentre abierto en cualquier organización protegida por firewall. Esto hace que no sea necesario tener especial cuidado abriendo puertos innecesarios para poder conectarnos a ellos. Antes de la llegada de los servicios web como los conocemos ahora existían otros protocolos más complicados que requerían de servicios y puertos adicionales, incrementando el riesgo de ataques en las organizaciones que los ponían en marcha.

Esquema de funcionamiento de un servicio web:



Los Servicios Web REST son Servicios Web que cumplen una serie de requisitos según un patrón de arquitectura definida hacia el año 2000 y que se ha extendido siendo el patrón predominante a la hora de implementar este tipo de aplicaciones.

Básicamente consiste en seguir una serie de reglas que definen dicha arquitectura. Entre ellas están el uso del protocolo HTTP por ser el más extendido a lo largo de Internet en la actualidad. Además, cada recurso del servicio web tiene que ser identificado por una dirección web (una URL) siguiendo una estructura determinada. Además, la respuesta tendrá que tener una estructura determinada en forma de texto que normalmente vendrá en alguno de los

formatos abiertos más conocidos como XML o JSON.

Ejemplos de URLs que definen el acceso a las operaciones de una servicio web REST:

Task	Method	Path
Create a new task	POST	/tasks
Delete an existing task	DELETE	/tasks/{id}
Get a specific task	GET	/tasks/{id}
Search for tasks	GET	/tasks
Update an existing task	PUT	/tasks/{id}

Esas URLs y su estructura son lo que definen lo que se conoce como la API del Servicio Web, que son las diferentes operaciones a las que los clientes tienen acceso para comunicarse con el mismo. En este caso se trata de una API Web.

Una Web API es una API (Application Programming Interface) implementada para un Servicio Web de forma que éste puede ser accesible mediante el protocolo HTTP, en principio por cualquier cliente web (navegador) aunque existen librerías que permiten que cualquier tipo de aplicación (escritorio, web, móvil, otros servicios web, . . .) accedan a la misma para comunicarse con dicho servicio web.

La Web API es una de las partes de los Servicios Web que, tal y como comentábamos anteriormente, mejoran sustancialmente la interoperabilidad de éstos con los potenciales clientes ya que permiten que sólo haya que implementar un único punto de entrada para comunicarse con el servicio web independientemente del tipo de aplicación que lo haga. De esa manera el desarrollador del Servicio Web define la lógica de negocio en el lado servidor y los diferentes clientes que quieran comunicarse con el mismo lo hacen a través de la Web API realizando solicitudes a las diferentes URLs que definen las operaciones disponibles.

Vamos al módulo filmoteca → carpeta controllers → archivo controllers.py

Dejar todo el contenido comentado; excepto la línea de from odoo... y añadir otro import:

```
controllers > controllers.py
1  # -*- coding: utf-8 -*-
2  from odoo import http
3  import json
```

Lo que hay que hacer es crear una clase nueva que hereda de http.Controller.

A continuación, aparece esta línea: @http.route(se le pasa la URL en la que va a responder el método, auth tiene varios parámetros posibles, la vamos a poner public y con ello se accede directamente a los datos; aunque esto no es seguro; method http que vamos a usar; csrf a False.

Para definir la función: get_peliculas(self, **kw indica que podemos pasar el número de atributos que queramos)

Se usa un bloque try...except.

Primero hay que buscar las películas: http.request.env[nombre modelo; sudo() y search_read(array de tuplas con criterios de filtrado y los campos a recuperar)

Luego hay que generar la respuesta: res=json.dumps (transforma ese objeto en una estructura JSON; se le pasa el diccionario de películas. Encode

Ahora hay que hacer un return Response(respuesta, content_type y charset, status=200)

Hay que añadir el import de Response

Ahora hay que controlar el error (status=505)

El contenido del archivo controllers.py sería:

```
# -*- coding: utf-8 -*-
from odoo import http
from odoo.http import Response
import json

class PeliculaController(http.Controller):

    @http.route('/api/peliculas', auth='public', method=['GET'], csrf=False)
    def get_peliculas(self, **kw):
        try:
            peliculas = http.request.env['filmoteca.pelicula'].sudo().search_read([], ['id', 'name', 'color'])
            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)
```

Para comprobar que todo va bien se puede hacer de dos formas: desde el navegador o desde postman:

The screenshot shows a web browser at the top displaying the JSON response from the API. Below it, the Postman interface is shown with a GET request to `http://localhost:8069/api/peliculas` sent successfully. The response status is 200 OK, with a response time of 20 ms and a size of 327 B. The response body is displayed in JSON format, showing an array of three movie objects.

```
[[{"id": 1, "name": "Memento Mori", "color": true}, {"id": 2, "name": "Buscando a Nemo", "color": true}, {"id": 3, "name": "Lo que el viento se llevo", "color": false}]]
```

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies (2) Headers (4) Test Results

200 OK 20 ms 327 B Save Response

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 1,
4     "name": "Memento Mori",
5     "color": true
6   },
7   {
8     "id": 2,
9     "name": "Buscando a Nemo",
10    "color": true
11  },
12 ]
```

JSON completo:

```
[
{
"id": 1,
"name": "Memento Mori",
"color": true
},
{
"id": 2,
"name": "Buscando a Nemo",
"color": true
},

```

```
{  
  "id": 3,  
  "name": "Lo que el viento se llevo",  
  "color": false  
}  
]
```

Ejercicio: Copiar el fichero controllers en el módulo “manage”, hacer las modificaciones oportunas y comprobar que funciona