

A dark blue vertical bar runs along the left edge of the page. A blue arrow points to the right from this bar, containing the text 'SGE'.

SGE

TEMA 5

CREACIÓN DE UN MÓDULO EN
ODOO

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

DAM

IES COMERCIO

Contenido

1. INTRODUCCIÓN.....	2
2. DESARROLLANDO UN MÓDULO EN ODOO.....	5
2.1. ESTRUCTURA DEL MÓDULO.....	7
2.2. ARCHIVO MANIFEST.PY	7
3. CREAR MODELOS.....	9
4. CREACIÓN DE LAS VISTAS.....	13
5. PERMISOS.....	25
6. CREACIÓN DE RELACIONES.....	27
7. VISTAS DE BÚSQUEDA.....	33
8. CAMPOS CALCULADOS.....	35
9. CREACIÓN DE INFORMES.....	37
10. DATOS PRECARGADOS.....	38
11. EJERCICIOS NO GUIADOS	39
12. BIBLIOGRAFÍA.....	39

1. INTRODUCCIÓN

En esta unidad de trabajo, en contraposición con la unidad de trabajo anterior, veremos cómo realizar modificaciones en el ERP utilizando el lenguaje de programación Python. De esta forma, desarrollaremos módulos que nos permitirán modificar el ERP de una forma automática y sencilla, simplemente instalando el módulo.

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


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

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

La licencia de Odoo ha ido cambiando a lo largo del tiempo. La actual, de la versión 15 'Community' es LGPLv3.

Odoo tiene una arquitectura cliente-servidor de 3 capas:

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

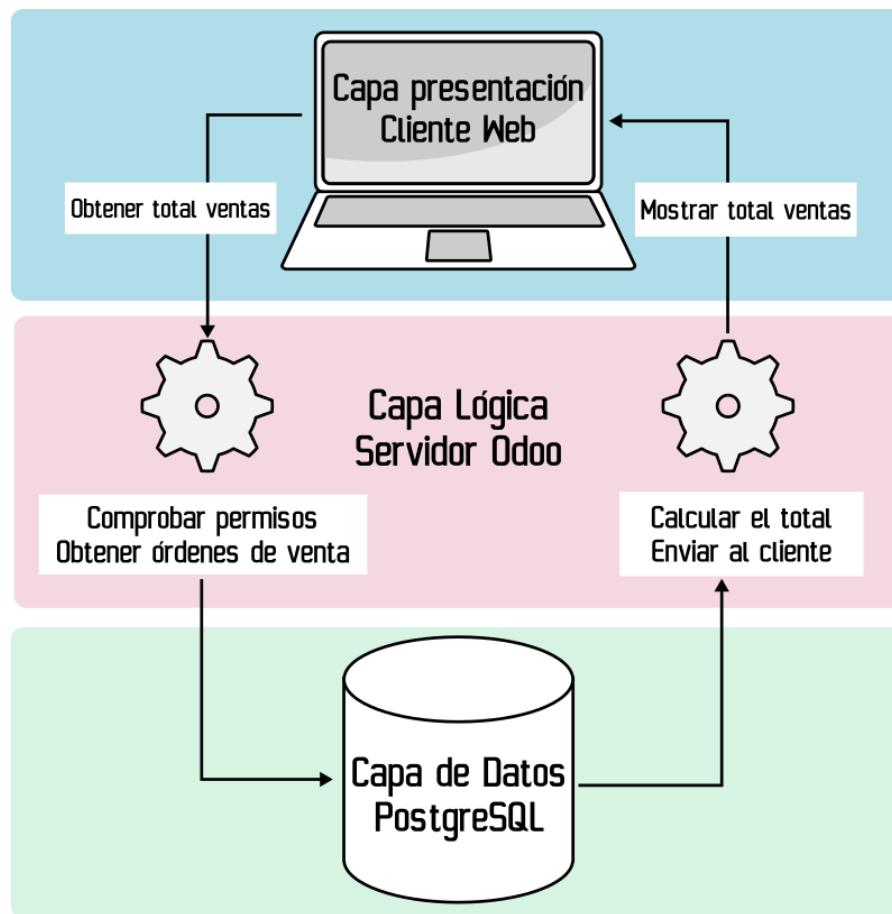
 **Atención:** además de usar su propio cliente, Odoo admite que otras aplicaciones interactúen con su servidor usando XML-RPC. También se pueden desarrollar “web controllers” para crear una API para aplicaciones web o móviles, por ejemplo.

Aparte de la arquitectura de 3 capas, Odoo es un **sistema modular**. Esto quiere decir que se puede ampliar con módulos de terceros (oficiales o no) y módulos desarrollados por nosotros mismos.

De hecho, en Odoo hay un módulo “base” que contiene el funcionamiento básico del servidor y a partir de ahí se cargan todos los demás módulos.

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

Para ampliar las funcionalidades o adaptar Odoo a las necesidades de una empresa, no hay que modificar el código fuente de Odoo. Tan solo necesitamos crear un módulo.



Los **módulos de Odoo** pueden modificar el comportamiento del programa, la estructura de la base de datos y/o la interfaz de usuario. En principio, un módulo se puede instalar y desinstalar y los cambios que implicaba el módulo se revierten completamente.

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

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

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

- La capa **ORM (Object Relational Mapping)** entre los objetos y la base de datos.
 - La combinación **Clase Python ↔ ORM ↔ Tabla PostgreSQL** se conoce como **Modelo**.
 - El programador no efectúa el diseño de la base de datos, solo de las clases y sus relaciones.
 - Tampoco es necesario hacer consultas SQL, casi todo se puede hacer con los métodos de ORM de Odoo.
- La arquitectura **MVC (Modelo-Vista-Controlador)**.
 - El modelo se programa declarando clases de Python que heredan de “**models.Model**”. Esta herencia provoca que actúe el ORM y se mapean en la base de datos.
 - **Las vistas** se definen normalmente en archivos XML y son listas, formularios, calendarios, gráficos, menús, etc. Este XML será enviado al cliente web donde el framework Javascript de Odoo lo transforma en HTML.
 - **El controlador** también se define en ficheros Python, normalmente junto al modelo. El controlador son los métodos que proporcionan la lógica de negocio.
- Odoo proporciona un diseñador de informes.
- El framework facilita la traducción de la aplicación a muchos idiomas.

1.1. La base de datos de Odoo

Gracias al ORM, no hay un diseño definido de la base de datos. La base de datos de una empresa puede tener algunas tablas muy diferentes a otras en función del mapeado que el ORM haya hecho con las clases activas en esa empresa. Por tanto, es difícil encontrar un diseño entidad-relación o algo similar en la documentación de Odoo.

Odoo tiene algunos modelos ya creados y bien documentados como:


- “**res.partner**” (clientes, proveedores, etc.).
- “**sale.order**” (Orden de venta).

Estos modelos existen de base debido a que están en casi todas las empresas y versiones de Odoo.

Pero ni siquiera estos tienen en la base de datos las mismas columnas o relaciones que en otras empresas. Muchas veces necesitamos saber el nombre del modelo, del campo o de la tabla en la base de datos. Para ello, Odoo proporciona en su “backend” el “modo desarrollador” para saber el modelo y campo poniendo el ratón encima de un campo de los formularios.

2. DESARROLLANDO UN MÓDULO EN ODOO.

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

 **Importante:** puesto que Odoo es de código abierto y todo el código está en Python, que no es un lenguaje compilado, podemos alterar los ficheros Python o XML de los módulos oficiales, cambiando lo que nos interese.

Esto puede funcionar, pero es una mala práctica, ya que:

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

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

Por ejemplo: puede que una empresa no necesite todos los datos que pide Odoo al registrar un producto. Como solución, podemos hacer un módulo que elimine de la vista los campos innecesarios. Si luego se comprueba que esos campos eran necesarios, solo hay que desinstalar el módulo y vuelven a aparecer.

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

Los módulos modifican partes de **Modelo-Vista-Controlador**. De esta manera, un módulo se compone de ficheros **Python, XML, CSS o Javascript** entre otros. Todos estos archivos deben estar en una **carpeta** con el **nombre del módulo**.

Hay una estructura de subcarpetas y de nombres de archivos que casi todos los módulos respetan. Pero todo depende de lo que ponga en el fichero “**__manifest__.py**”. Este fichero contiene un **diccionario de Python** con **información del módulo** y la **ubicación** de los demás **ficheros**. Además, el archivo “**__init__.py**” indica qué ficheros Python se han de **importar**.

El primer concepto que tenemos que tener claro es que un **módulo** es un **directorio** dentro de la carpeta **addons** de nuestro servidor. Este directorio tendrá una serie de archivos que **definirán el módulo**.

Tal y como lo dice la [documentación al respecto](#), un módulo de Odoo se compone de los siguientes elementos:

- **Objetos de negocio** -> Modelos: .Py
- **Archivos de datos** (XML, csv): sirven para declarar **metainformación**, valores de **configuración** y datos de **demostración**.
- **Controladores web**: manejadores de **solicitudes web**.
- **Archivos estáticos** para la web (css, js).

Odoo sugiere un conjunto de buenas prácticas y una [serie de convenciones](#) (vean ese link) a la hora de su desarrollo, que nos dicen, entre otras cosas, cómo debemos llamar los archivos. Voy a dar un ejemplo de la estructura de archivos que nosotros utilizamos para armar un módulo Odoo, es la misma estructura que Odoo utiliza para desarrollar sus propios módulos.

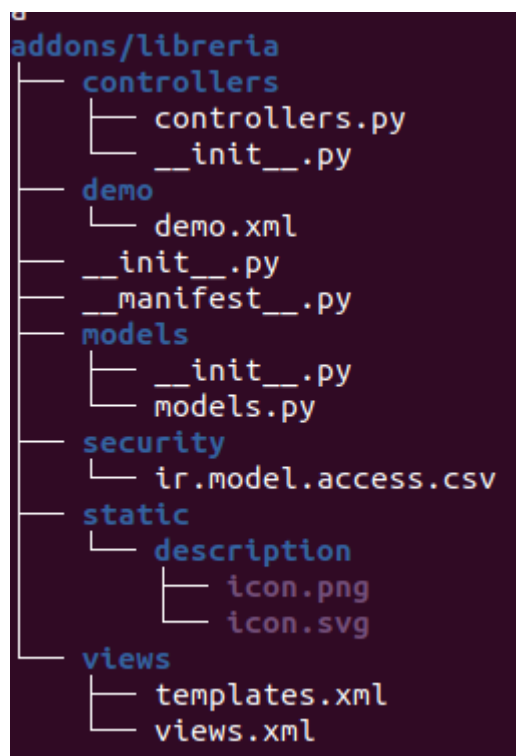
Dentro de un módulo podemos encontrar:

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

Los módulos se guardan en un directorio indicado en la opción “**--addons-path**” al **lanzar el servidor** o en el **fichero** de configuración “**odoo.conf**”. Los módulos pueden estar en **más de un directorio** y **dependen** del tipo de **instalación** o la **distribución** o **versión** que se instale.

Vamos a explicar para qué sirve cada carpeta y archivo de esta simple estructura inicial (aunque sus nombres son muy sugestivos).

- Carpeta **data**: generalmente contiene archivos **css** y **XML** que nos permitirán **precargar información** en nuestro sistema. Un típico ejemplo podría ser cuando queremos que nuestro sistema provea al usuario una lista de colores para que estas selecciones uno.
- Carpeta **models**: contiene los archivos **.Py** donde se declaran los **modelos de negocio**.
- Carpeta **security**: Contiene archivos **XML** donde declaramos los **grupos de usuarios**, **permisos** y **reglas de seguridad** de nuestro modulo.



- Carpeta **views**: Contiene archivos **XML** donde declaramos las **vistas** para nuestros modelos de negocio.
- Archivo **__init__.py**: Este archivo es característico del desarrollo en Python y nos sirve para indicar desde qué otros directorios deben **importarse** paquetes de Python, por ejemplo, del directorio **models** (carpeta models). Es el **primero** que se lee e indica a Python que módulos tiene importar (controllers y models).
- Archivo **__manifest__.py**: Este archivo es el alma del módulo Odoo y nos permite describir por completo **como se ensambla** el módulo, su **nombre**, su **versión**, su **descripción**, sus **dependencias**, los **archivos de datos** que debe levantar, etc.

2.1. ESTRUCTURA DEL MÓDULO.

En Odoo la estructura básica del módulo puede ser creada automáticamente. Odoo ofrece un mecanismo para ayudar a comenzar un nuevo módulo, el comando **odoo-bin** tiene un subcomando **scaffold** para crear un **módulo vacío**:

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

El comando crea un **subdirectorio** para un **módulo** y automáticamente crea un montón de **archivos estándar** para dicho módulo. La mayoría de ellos contienen sólo **código comentado** o **XML**. A lo largo de este tutorial se explicará el uso de la mayoría de estos archivos.

Ejercicio: Ejecuta el siguiente comando situado en la raíz del proyecto Odoo 15.

```
./odoo-bin scaffold libreria addons
```

2.2. ARCHIVO MANIFEST.PY

Manifest.py: Diccionario **clave valor** que sirve a Odoo para rellenar la página del **módulo de aplicaciones**.

El archivo de manifiesto, en otras versiones de Odoo se llamó **__openerp__.py**, pero Odoo a partir de su versión 9 comenzó sucesivamente a eliminar la palabra openerp de todas partes. Creo que ahora tiene un nombre realmente adecuado, porque este archivo es como el director de la orquesta, él posee la batuta y decide quien se levanta y quien no.

A continuación, vamos a ver los **parámetros** del archivo y luego vamos a ver un ejemplo de su declaración.

- **name** Nombre del módulo.
- **version** Versión del módulo.
- **summary** Descripción corta del módulo.
- **description** Descripción más **extensa** del módulo.
- **category** Categoría del módulo. Están declaradas en el módulo base de Odoo y son:
 - ✓ Customer Relationship
 - ✓ Management

- ✓ Sales Management
- ✓ Project Management
- ✓ Knowledge Management
- ✓ Warehouse Management
- ✓ Manufacturing
- ✓ Invoicing & Payments
- ✓ Accounting & Finance
- ✓ Purchase Management
- ✓ Human Resources
- ✓ Extra Tools
- ✓ Marketing
- ✓ Point of Sale
- ✓ Advanced Reporting

- **author** Aquí va vuestro nombre.
- **website** Su web.
- **license** La licencia que elijan, por defecto es AGPL-3
- **depends** Lista de módulos de los cuales depende este módulo.
- **data** Lista de archivos XML y csv que se deben levantar (los archivos donde declaran sus vistas, etc.).
- **demo** Archivos XML adicionales que se usan para levantar información de demo para probar su módulo, si es que así lo desean.
- **installable** True o False. Define si este módulo es instalable, podría ser un módulo que solo sirve para inyectarlo como dependencia.
- **auto_install** True o False (por defecto: False). Si es True, entonces el módulo se instala automáticamente cuando todos sus módulos dependientes están instalados. (Si no tiene módulos de dependencia, se instala solo cuando instalamos Odoo)

El **orden** en que se declaran los archivos **XML ES IMPORTANTE**. Si por ejemplo en la parte de **data** tienen dos archivos XML, uno en donde declaran las vistas y otro donde declaran menús y actions, entonces **primero** deben indicar que se levante el archivo de **vistas** y luego el archivo donde estén las **actions**, dado que en las actions haremos referencia al ID de las vistas.

Deben usar las **categorías declaradas** por ODOO, esto es importante para el día en que quieran subir su modulo a la app **store** Odoo.

Ejercicios

1. Rellena los datos del archivo manifest.py para el módulo librería.
2. Añade un logo al módulo librería, para ello crea una carpeta static dentro del módulo librería y otra dentro llamada description y ahí poner la imagen icon.png

3. CREAR MODELOS.

Los **modelos** van a corresponder con **tablas** de **bases de datos** y son el fundamento de nuestro módulo. En este ejemplo, vamos a realizar un módulo para una librería. Nuestros objetos modelo serán libro y categoría. Los archivos de modelo se escriben en Python y siguen la arquitectura ORM que nos proporciona Odoo.

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

Una clase heredada de **“models.Model”** se comporta de la siguiente manera:

- Puede ser accedida como modelo, como **“recordset”** (conjunto de registros) o como **“singleton”** (un único registro). Si es accedida como modelo, tiene métodos de modelo para crear “recordsets”, por ejemplo. Si es accedida como “recordset”, se puede **acceder** a los **datos que guarda**.
- Puede tener atributos internos de la clase, ya que sigue siendo Python. Pero los **atributos** que se guardan en la **base de datos** se han de definir como **“fields”**. Un “field” es una instancia de la clase **“fields.Field”**, y tiene sus propios atributos y funciones.
 - **Odoo analizará el modelo, buscará los atributos tipo “field” y sus propiedades y mapeará automáticamente todo esto en el ORM.**
- Los métodos definidos para los **“recordset”** reciben un argumento llamado **“self”** que puede ser un **“recordset”** con una colección de **registros**. Por tanto, deben **iterar** en el self para hacer su función en cada uno de los registros.
- Un modelo representa a la tabla entera en la base de datos. Un **“recordset”** representa a una **colección de registros** de esa **tabla** y también al **modelo**. Un **“singleton”** es un **“recordset”** de **un solo elemento**.
- Los modelos tienen sus propias **funciones** para no tener que acceder a la base de datos para modificar o crear registros. Además, incorporan **restricciones de integridad**.

Este es el ejemplo de un modelo con solamente un “field”:

```
class AModel(models.Model):
    _name = 'a.model'
    _description = 'descripción opcional'
    name = fields.Char(
        string="Name",           # El nombre en el label (Opcional)
        compute="_compute_name_custom", # En caso de ser computado, el nombre de la función
        store=True,              # En caso de ser computado, si se guarda o no
        select=True,             # Forzar que esté indexado
```


```

default='Nombre',          # Valor por defecto, puede ser una función
readonly=True,            # El usuario no puede escribir directamente
inverse='_write_name'     # En caso de ser computada y se modifique
required=True,            # Field obligatorio
translate=True,          # Si se puede traducir
help='blabla',            # Ayuda al usuario
company_dependent=True,   # Transforma columna a ir.property
search='_search_function', # En caso de ser computado, cómo buscar en él.
copy = True               # Si se puede copiar con copy()
)

```

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

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

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

Los modelos tienen algunos atributos del modelo, como “**_name**” o “**_description**”. Otro atributo de modelo importante es “**_rec_name**” que indica de que **atributo toma nombre el registro** y que por defecto apunta al atributo “**name**” (no confundir con “**name**”).

- En las vistas (que veremos más adelante), en algunos campos se basa en el atributo marcado por “**_rec_name**”, que por defecto es “**name**”. Si no tenemos un atributo “**name**” o queremos que sea otro el que, de nombre, podemos modificarlo con “**_rec_name=nombreatributo**”.

3.1. Atributos tipo “field” simples

Los modelos tienen otros atributos tipo “field”, que se mapean en la base de datos y a los que el usuario tiene acceso y métodos que conforman el controlador. A continuación vamos a detallar todos los tipos de “field” que hay y sus posibilidades:

- **Char**, acepta un segundo argumento opcional, “**size**”, que corresponde al tamaño máximo del texto. Es recomendable usarlo solo si se tiene una buena razón.
- **Text**, se diferencia de Char en que puede albergar texto de varias líneas, pero espera los mismos argumentos.

- **Selection**, es una lista de selección desplegable. El primer argumento es la lista de opciones seleccionables y el segundo es la cadena de título. La lista de selección es una tupla ('value', 'Title') para el valor almacenado en la base de datos y la cadena de descripción correspondiente. Cuando se amplía a través de la herencia, el argumento selection_add puede ser usado para agregar opciones a la lista de selección existente.
- **Html**, es almacenado como un campo de texto, pero tiene un manejo específico para presentar el contenido HTML en la interfaz.
- **Integer**, solo espera un argumento de cadena de texto para el campo de título.
- **Float**, tiene un argumento opcional, una tupla (x,y) con los campos de precisión: 'x' como el número total de dígitos; 'y' representa los dígitos decimales.
- **Date y Datetime**, estos datos son almacenados en formato UTC. Se realizan conversiones automáticas, basadas en las preferencias del usuario o la usuaria, disponibles a través del contexto de la sesión de usuario. Esto es discutido con mayor detalle en el Capítulo 6.
- **Boolean**, solo espera sea fijado el campo de título, incluso si es opcional.
- **Binary** también espera este único argumento.

Además de estos, también existen los campos relacionales, los cuales serán introducidos en este mismo capítulo. Pero por ahora, hay mucho que aprender sobre los tipos de campos y sus atributos.

Atributos de campo comunes (<https://odoo.rgbconsulting.com/es/tipos-campos-odoo/>)

Los campos también tienen un conjunto de atributos:

- **string**, es el título del campo, usado como su etiqueta en la UI. La mayoría de las veces no es usada como palabra clave, ya que puede ser fijado como un argumento de posición.
- **default**, fija un valor predefinido para el campo. Puede ser un valor estático o uno fijado anticipadamente, pudiendo ser una referencia a una función o una expresión lambda.
- **size**, aplica solo para los campos Char, y pueden fijar el tamaño máximo permitido.
- **translate**, aplicable para los campos de texto, Char, Text y Html, y hacen que los campos puedan ser traducidos: puede tener varios valores para diferentes idiomas.
- **help**, proporciona el texto de ayuda desplegable mostrado a los usuarios y usuarias.
- **readonly** = True, hace que el campo no pueda ser editado en la interfaz.
- **required** = True, hace que el campo sea obligatorio.
- **index** = True, creará un índice en la base de datos para el campo.
- **copy** = False, hace que el campo sea ignorado cuando se usa la función Copiar. Los campos no relacionados de forma predeterminada pueden ser copiados.
- **groups**, permite limitar la visibilidad y el acceso a los campos solo a determinados grupos. Es una lista de cadenas de texto separadas por comas, que contiene los ID XML del grupo de seguridad.

Algunos aspectos que debes tener en cuenta a la hora de crear los modelos son:

- Todas las clases de odoo extienden de `Models.model`
- Es importante que recuerdes añadir siempre un campo `_name` a cada uno de los modelos que crees. Gracias a este campo podrás definir relaciones sin problemas y la búsqueda en las vistas funcionará directamente. Es un campo `privado` y se suele nombrar con `NombreMódulo.nombreObjeto`
- El `nombre` de la `clase` suele nombrarse de la siguiente forma `NombreMódulo_NombreObjeto`

3.2. Restricciones (`constraints`)

No en todos los “fields” los usuarios pueden poner de todo. Por ejemplo, podemos necesitar limitar el precio de un producto en función de unos límites preestablecidos. Si el usuario crea un nuevo producto y se pasa al poner el precio, no debe dejarle guardar. Las restricciones se consiguen con un decorador llamado “`@api.constraint()`” el cual ejecutará una función que revisará si el usuario ha introducido correctamente los datos. Veamos un ejemplo:

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

Ejercicio. Para el módulo librería crea los siguientes objetos en el archivo `models.py` de la carpeta `models`:

```
class libreria_categoria(models.Model):
    _name = "libreria.categoria"
    _description = 'libreria.categoria'
    _sql_constraints = [('name', 'unique(name)', 'Ya existe una categoria con ese nombre . El nombre debe ser unico'), ]
    name = fields.Char(string="Nombre", required=True, help="Introduce el nombre de la categoria")
    descripcion = fields.Text(string="Descripción")

class libreria_libro(models.Model):
    _name = "libreria.libro"
    name = fields.Char(string="Titulo", required=True)
    precio = fields.Float(string="Precio")
    ejemplares = fields.Integer(string="Ejemplares")
    fecha = fields.Date(string="Fecha de compra")
    segmano = fields.Boolean(string="Segunda mano")
    estado = fields.Selection([(0, 'Bueno'), (1, 'Regular'), (2, 'Malo')], string="Estado", default="0")
```

Ejercicio. Instala el módulo librería y comprueba que se han creado los Modelos.

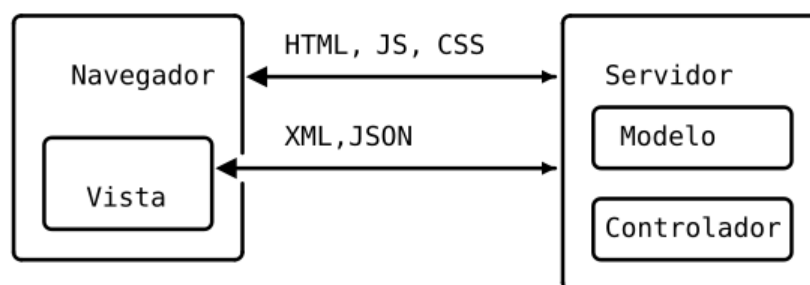
En el siguiente video puedes ver cómo crear los modelos y añadirles campos básicos.

<https://www.youtube.com/watch?v=DurV9im-PdM>

4. CREACIÓN DE LAS VISTAS.

Las vistas se definen de manera muy similar a como habíamos visto en la unidad anterior. Tendremos vistas de tipo **formulario**, **árbol (tree)**, **kamban**, **búsqueda**, etc. En nuestro módulo las meteremos en un **directorio** concreto (**views**), de cara a tenerlas identificadas. Podremos meter **varias vistas** dentro de un mismo fichero **XML** o bien **separarlas** en **varios ficheros XML**.

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

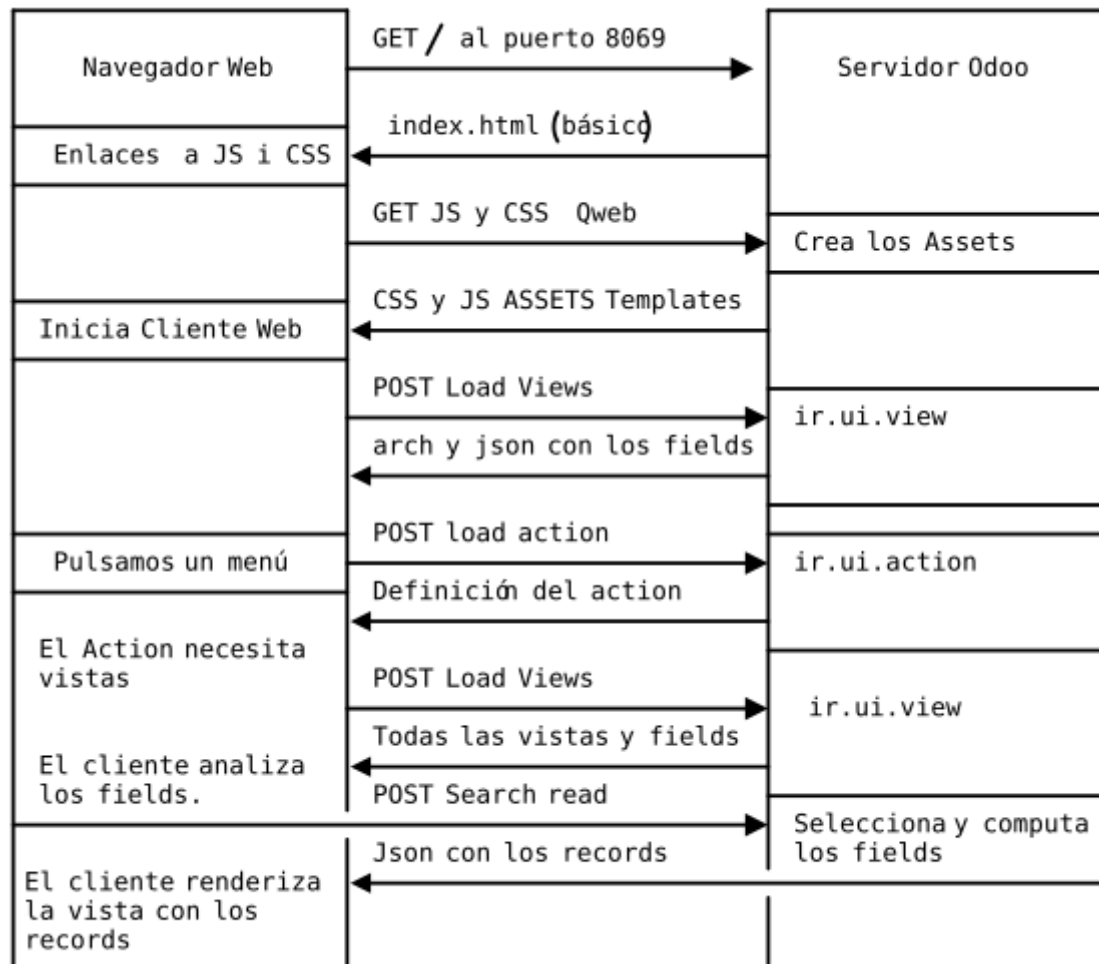


En la **primera conexión** con el navegador web, el servidor Odoo le proporciona un **HTML** mínimo, una **SPA** (Single Page Application) en **Javascript** y un **CSS**. Esto es un cliente web para el servidor y es lo que se considera la vista.

Pero Odoo tampoco carga la vista completa, ya que podría ser inmensa. Cada vez que los menús o botones de la vista **requieren cargar la visualización** de unos datos, piden al servidor un **XML** que defina cómo se van a ver esos **datos** y un **JSON** con los **datos**.

Entonces la **vista renderiza** los **datos** según el esquema del **XML** y los **estilos definidos** en el cliente.

Esta visualización se hace con unos elementos llamados **Widgets**, que son combinaciones de **CSS, HTML y Javascript** que definen el **aspecto y comportamiento** de un tipo de datos en una vista en concreto. Todo esto es lo que vamos a ver con detalle en este apartado.



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

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

La vista tiene varios **elementos necesarios** para funcionar:

- **Definiciones de vistas:** son los más evidentes. Son las propias definiciones de las vistas, guardadas en el modelo “**ir.ui.view**”. Estos elementos tienen al menos los “**fields**” que se van a mostrar y pueden tener información sobre la **disposición**, el **comportamiento** o el **aspecto** de los “fields”.
- **Menús:** son otros elementos fundamentales. Están distribuidos de forma jerárquica y se guardan en el modelo “**ir.ui.menu**”.
- **Actions:** las acciones o “**actions**” enlazan una **acción del usuario** (como pulsar en un menú) con una **llamada al servidor** desde el cliente para pedir algo (como cargar una nueva vista). Las ‘actions’ están guardadas en **varios modelos** dependiendo del **tipo**.

Aquí un ejemplo completo:

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

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

El primer “record” define una vista tipo “**tree**” que es una lista de estudiantes donde se verán los campos “name” y “topics”. El segundo “record” es la definición de un “**action**” tipo “window”, es decir, que abre una ventana para mostrar unas vistas de tipo “**tree**” y

form” (formulario). Los otros definen tres niveles de **menú**: el superior, el intermedio y el menú desplegable que contiene el **“action”**. Cuando el usuario navegue por los dos menús superiores y presione el tercer elemento de menú se ejecutará ese **“action”** que cargará la vista **“tree”** definida y una vista **“form”** inventada por Odoo.

4.1. Vista Tree

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

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

Como se puede observar, esta vista se guardará en el modelo **“ir.ui.view”** con un **external ID** llamado **“prueba.student_list”**. Tiene más posibles **“fields”**, pero los mínimos necesarios son **“name”**, **“model”** y **“arch”**. El **“field arch”** guarda el XML que será enviado al cliente para que renderice la vista. Dentro del **“field arch”** está la etiqueta **“<tree>”** que indica que es una lista y dentro de esta etiqueta tenemos más **“fields”** (los **“fields”** del modelo **“prueba.student”**) que queremos mostrar.

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

Colores en las líneas

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

- **decoration-bf**: líneas en BOLD
- **decoration-it**: líneas en ITALICS
- **decoration-danger**: color LIGHT RED
- **decoration-info**: color LIGHT BLUE
- **decoration-muted**: color LIGHT GRAY
- **decoration-primary**: color LIGHT PURPLE
- **decoration-success**: color LIGHT GREEN
- **decoration-warning**: color LIGHT BROWN

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

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

Líneas editables

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

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

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

Campos invisibles

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

Cálculos de totales

En los “fields” **numéricos**, si queremos mostrar la **suma total**, podemos usar el atributo “**sum**”.

Ejemplo de cómo quedaría una vista “tree” con todo lo que hemos explicado:

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

4.2. Vista Form

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

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

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

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

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

Si hacemos varios “**groups**” o “groups **dentro de groups**”, el CSS de Odoo ya alinea los “fields” en **columnas** o los separa correctamente. Sin embargo, si queremos **separar** manualmente algunos “fields”, podemos utilizar la etiqueta “<separator string=’Nombre del separador’>”.

Otro elemento de separación y organización es “<notebook> <page string=’título’>”, que crea unas **pestañas** que esconden partes del formulario y permiten que quepa en la pantalla.

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

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

4.3. **Widgets**

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

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

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

Aquí tenemos algunos “widgets” disponibles para “fields” numéricos:

- **integer**: el número sin comas. Si está vacío, muestra un 0.
- **char**: el carácter, aunque muestra el campo más ancho. Si está vacío muestra un hueco.
- **id**: no se puede editar.
- **float**: el número con decimales.
- **percentpie**: un gráfico circular con el porcentaje.
- **progressbar**: una barra de progreso:
- **monetary**: con dos decimales.
- **field_float_rating**: estrellas en función de un float.



Points

45.00%

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

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

4.4. Formularios dinámicos

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

Se puede ocultar condicionalmente un “field”:

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

Se puede mostrar u ocultar en modo edición o lectura:

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

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

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

También existe la opción de ocultar una columna de un “tree” de un “X2many”:

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

💬 **Interesante:** fijate en el ejemplo anterior que dice “parent.state”. Esto hace referencia al “field state” del modelo padre de ese “tree”. Hay que tener en cuenta que ese “tree” se muestra con el modelo al que hace referencia el “X2many”, pero está dentro de un formulario de otro modelo.

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

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

Veamos ahora un ejemplo con todos los “attrs”:

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

Asistentes

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

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

4.5. Vista **Kanban**

La vista “tree” y la vista “form” son las que funcionan por defecto en cualquier “action”. El resto de vistas, como la “Kanban”, necesitan una definición en **XML** para funcionar.

Además, dada la gran cantidad de opciones que tenemos al hacer un “Kanban”, no disponemos de etiquetas como en el “form” que después se traduzcan en HTML o CSS y den un formato estándar y confortable. Cuando estamos definiendo una vista “Kanban” entramos en el terreno del lenguaje “**QWeb**” y del **HTML o CSS explícito**.

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

Veamos un ejemplo mínimo de “Kanban” donde describiremos posteriormente para qué sirven las etiquetas y atributos.

```
<record model="ir.ui.view" id="terraform.planet_kanban_view">  
    <field name="name">Student kanban</field>  
    <field name="model">school.student</field>  
    <field name="arch" type="xml">
```

```

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

```

4.6. Vista Calendar

Esta vista muestra un calendario si los datos de los registros del modelo tienen al menos un “**field**” que indica una **fecha** y **otro** que indique una **fecha final o una duración**.

La sintaxis es muy simple, veamos un ejemplo:

```

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

```

```

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

```

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

4.7. Vista **Graph**

La vista “graph” permite mostrar una gráfica a partir de algunos “**fields**” **numéricos** que tenga el modelo. Esta vista puede ser de tipo “**pie**” (tarta en inglés), “**bar**” o “**line**” y se comporta agregando los valores que ha de mostrar.

En este ejemplo se ve un gráfico en el que se mostrará la evolución de las notas en el tiempo de cada estudiante. Por eso el tiempo se pone como “**row**”, el estudiante como “**col**” y los datos a mostrar que son las calificaciones como “**measure**”. En caso de olvidar poner al estudiante como “**col**” nos mostraría la evolución en el tiempo de la suma de las notas de todos los estudiantes.

```

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

```

Ejercicio. Vamos a modificar el archivo views.xml:

```
<odoo>
<data>
<!-- VISTAS MODELO CATEGORIA -->

    <record model="ir.ui.view" id="libreria.categoria_tree">
        <field name="name">libreria.categoria.tree</field>
        <field name="model">libreria.categoria</field> <!-- Este campo hace referencia a
la variable '_name' del modulo respectivo -->
        <field name="arch" type="xml">
            <tree>
                <field name="name"/>
                <field name="descripcion"/>
            </tree>
        </field>
    </record>

    <record model="ir.ui.view" id="libreria.categoria_form">
        <field name="name">libreria.categoria.form</field>
        <field name="model">libreria.categoria</field>
        <field name="arch" type="xml">
            <form>
                <group colspan="2" col="2">
                    <field name="name"/>
                    <field name="descripcion"/>
                </group>
            </form>
        </field>
    </record>
<!-- VISTAS MODELO LIBRO -->

    <record model="ir.ui.view" id="libreria.libro_tree">
        <field name="name">libreria.libro.tree</field>
        <field name="model">libreria.libro</field>
        <field name="arch" type="xml">
            <tree>
                <field name="name"/>
                <field name="precio"/>
                <field name="ejemplares"/>
            </tree>
        </field>
    </record>

    <record model="ir.ui.view" id="libreria.libro_form">
        <field name="name">libreria.libro.form</field>
        <field name="model">libreria.libro</field>
        <field name="arch" type="xml">
            <form>
                <group colspan="2" col="2">
```



```

        <field name="name"/>
    <field name="precio"/>
        <field name="ejemplares"/>
            <field name="fecha"/>
        <field name="segmano"/>
            <field name="estado"/>
        </group>
    </form>
</field>
</record>

<!--ACCIONES DE VENTANA -->
    <!-- MODELO CATEGORIA -->

<record model="ir.actions.act_window"
id="libreria.categoria_action_window">
    <field name="name"> libreria.categoria.action_window </field>
    <field name="res_model">libreria.categoria</field>
    <field name="view_mode">tree,form</field>
</record>
    <!-- MODELO LIBRO -->
    <record model="ir.actions.act_window" id="libreria.libro_action_window">
    <field name="name">libreria.libro.action_window</field>
    <field name="res_model">libreria.libro</field> <!--Campo _name-->

    <field name="view_mode">tree,form</field>
    </record>
<!--MENÚS -->
    <menuitem name="Libreria" id="libreria.menu_root"
action="libreria.libro_action_window"/>
    <menuitem name="Categorías" id="libreria.categoria_menu"
parent="libreria.menu_root" action="libreria.categoria_action_window"/>
<!--Referencia al ID-->
    <menuitem name="Libros" id="libreria.libro_menu" parent="libreria.menu_root"
action="libreria.libro_action_window"/> <!--Referencia al ID-->
</data>
</odoo>

```

Ejercicio. Comprueba que la vista está declarada en el manifest y actualiza el módulo para comprobar que funciona.

En el siguiente video puedes ver cómo definir las vistas y los menús para el módulo librería que estamos creando:

<https://www.youtube.com/watch?v=Q7ey7eeE494>

5. PERMISOS.

Odoo necesita conocer que permisos tienen los usuarios/roles del sistema para cada modelo particular de nuestro módulo. En el fichero “**__manifest__.py**” se indica en la **ruta** a un fichero donde se detallan estos permisos, de una forma similar a:

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

El contenido del fichero es una **cabecera**, indicando que es cada campo (de una manera muy descriptiva), seguido de un conjunto de líneas, cada una definiendo una **ACL** (Access Control List).

Veamos un ejemplo:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
acl_lista_tareas,lista_tareas.lista_default,model_lista_tareas_lista,base.group_user,1,1,1,1
```

En ese ejemplo se define:

- Una ACL con id “acl_lista_tareas”.
- Un nombre que indique que afecta al modelo “lista_tareas.lista” (y se indica con “lista_tareas.lista_default_model”).
- El modelo “lista_tareas.lista”, indicado por su **External ID** como “model_lista_tareas_lista”.
- El grupo al que se aplica esta ACL. Indicando “base.group_user” se aplica a todos los usuarios.
- Una lista de permisos (lectura, escritura, creación y borrado) donde “1” indica “permiso concedido” y “0” indica “permiso denegado”.

Si queremos **definir grupos propios** para la **ACL**, aparte de los que pueda poseer Odoo, podemos hacerlo indicando en “__manifest__.py” un fichero de **definición de grupos** de forma similar a esta:

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

Veamos un ejemplo de definición de grupo:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>
  <record id="grupo_bibliotecario" model="res.groups">
    <field name="name">Bibliotecario</field>
    <field name="users" eval="[(4, ref('base.user_admin'))]" />
  </record>
</odoo>
```

Con este ejemplo, hemos creado el “grupo_bibliotecario” y lo hemos poblado añadiendo los usuarios que pertenezcan al grupo de administradores (“base.user_admin”). Para usarlo en el fichero “csv” con las ACL, simplemente

deberemos indicar en el campo grupo “grupo_bibliotecario”. Más información en www.odoo.yenthevg.com/creating-security-groups-odoo/ y en https://www.odoo.com/documentation/15.0/es/developer/howtos/rdtraining/05_securityintro.html

Para poder tener **acceso al módulo**, es necesario dejar creados **grupos** que gestionen los **accesos** a las **partes de nuestro módulo**. En el caso de la librería crearemos un único **grupo LibreríaResponsable** que tendrá **todos los privilegios** sobre el módulo librería. De manera análoga podríamos crear otros grupos que diesen permisos más restrictivos sobre el módulo.

Para ello debemos editar dos archivos, un **security.xml** y un **archivo .csv** donde indicaremos que permisos tiene cada grupo sobre cada objeto de Odoo individual.

Ejercicio. Modificar los permisos del módulo

En la carpeta security, vamos a crear el archivo security.xml vamos a indicar que grupos vamos a tener

```
<odoo>
<data>
  <record id="libreria_manager" model="res.groups">
    <field name="name">LibreriaResponsable </field>
  </record>
</data>
</odoo>
```

En el archivo csv indicamos los permisos, la primera línea no se modifica y significa Nombre regla, campo regla, model_objeto, **grupo**, lectura, escritura, creación, borrado
En la segunda línea añadimos:

```
access_libreria_categoria2,libreria.categoria2,model_libreria_categoria,libreria_manager,1,1,1,1
```

```
access_libreria_libro2,libreria.libro2,model_libreria_libro,libreria_manager,1,1,1,1
```

Indicamos en el manifest en el apartado de data:

```
‘security/security.xml’
```

```
‘security/ir.model.access.csv’
```

Y en los menús podemos modificar el grupo (groups=“librería_manager”)

Para poder ver si funciona debemos añadir al usuario administrador al grupo de librería responsable.

En el siguiente video puedes revisar como realizar esta tarea siguiendo el ejemplo que estamos desarrollando:

6. CREACIÓN DE RELACIONES.

Generalmente los modelos que creamos tendrán relaciones entre sí o con otros objetos del sistema. Las relaciones que podemos crear en Odoo son:

- **one2many**
- **many2one**
- **any2many**

Las tres son análogas a las relaciones **SQL**. Es importante destacar que para crear una relación one2many en un modelo debemos crear la relación many2one en el modelo con el que establecemos la relación. Por otra parte, es vital que los modelos tengan un campo name para que las relaciones funcionen.

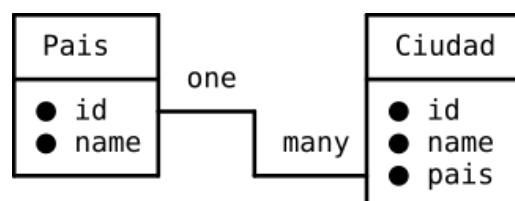
A continuación, vamos a observar los “fields” relacionales. Dado que el ORM evita que tengamos que crear las tablas y sus relaciones en la base de datos, cuando existen relaciones entre modelos se necesitan unos campos que definan esas relaciones.

Ejemplo: un pedido de venta tiene un cliente y un cliente puede hacer muchos pedidos de venta. A su vez, ese pedido tiene muchas líneas de pedido, que son solo de ese pedido y tienen un producto, que puede estar en muchas líneas de venta.

En situaciones como la del ejemplo, estas relaciones acaban estando en la base de datos con claves ajenas. Pero con los frameworks que implementan ORM, todo esto es mucho más sencillo.

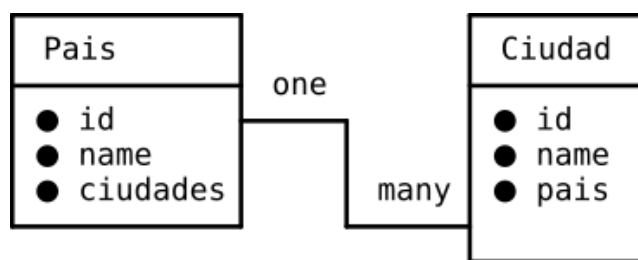
Para ello utilizaremos los “fields” relacionales de Odoo:

- **Many2one:** es el más simple. Indica que el modelo en el que está tiene una relación **muchos a uno** con otro modelo. Esto significa que un registro tiene relación con un único registro del otro modelo, mientras que el otro registro puede tener relación con muchos registros del modelo que tiene el “**Many2one**”. En la tabla de la base de datos, esto se traducirá en una clave ajena a la otra tabla.
 - Ejemplo donde se pretende que cada ciudad almacene su país.



```
pais_id = fields.Many2one('modulo.pais') # La forma más común, en el Modelo Ciudad
pais_id = fields.Many2one(comodel_name='modulo.pais') # Otra forma, con argumento
```

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

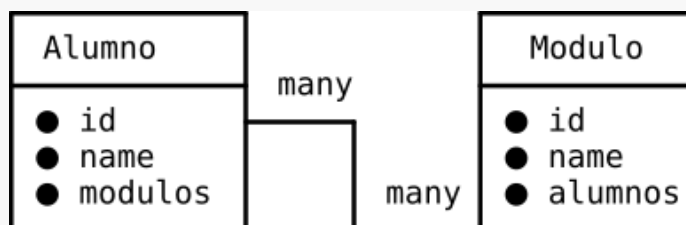


```

pais_id = fields.Many2one('modulo.pais') # Esto en el modelo Ciudad, indica un many2one
ciudades_ids = field.One2many('modulo.ciudad', 'pais_id') # En el modelo Pais.
  
```

El nombre del modelo y el field que tiene el Many2one necesario para que funcione.

- **Many2many:** Se trata de una relación muchos a muchos. Esto se acaba mapeando como una tabla intermedia con claves ajenas a las dos tablas. Hacer los “Many2many” simplifica mucho la gestión de estas tablas intermedias y evita redundancias o errores. La mayoría de los “Many2many” son muy fáciles de gestionar, pero algunos necesitan conocer realmente qué ha pasado en el ORM.
 - Ejemplo que indica que un alumno puede tener muchos módulos y un módulo puede tener muchos alumnos.

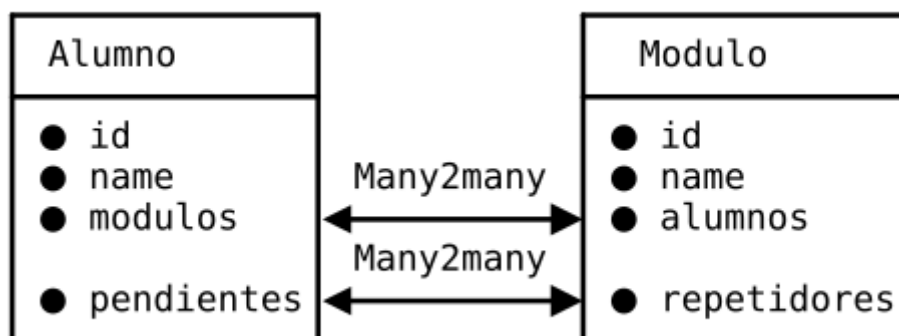


```

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

En el ejemplo anterior, Odoo interpretará que estos dos “Many2many” corresponden a la misma relación y creará una tabla intermedia con un nombre generado a partir del nombre de los dos modelos. No obstante, no tenemos control sobre la tabla intermedia.

Puede que, en otros contextos, necesitemos tener dos relaciones “Many2many” independientes sobre dos mismos modelos. Observemos este diagrama:



Existen dos relaciones “Many2many”:

- Las de “alumnos con módulos”, descrita en el ejemplo anterior.
- Una nueva relación, donde se relacionan alumnos repetidores con módulos pendientes.

No deben coincidir, pero si no se especifica una tabla intermedia diferente, **Odoo considerará que es la misma relación**. En estos casos hay que especificar la tabla intermedia con la sintaxis completa para evitar errores:

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


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


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

Si tomamos como ejemplo el anterior de las ciudades y países, imaginemos que queremos mostrar la bandera del país en el que está la ciudad. La **bandera** será un campo “Image” que estará en el modelo **país**, pero lo queremos mostrar también en el modelo **ciudad**. Para ello tenemos dos posibles soluciones:


- La solución mala sería guardar la bandera en cada ciudad.
- La buena solución es usar un “field related” para acceder a la bandera.

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

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

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

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

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

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

```
# -*- coding: utf-8 -*-
from odoo import models, fields, api
from openerp.exceptions import ValidationError
####
class net(models.Model):
    _name = 'networks.net'
    _description = 'Networks Model'
    name = fields.Char()
    net_ip = fields.Char()
    mask = fields.Integer()
    net_map = fields.image()
    net_class = fields.Selection([('a','A'),('b','B'),('c','C')])
    pcs = fields.One2many('networks.pc','net')
    servers = fields.Many2many('networks.pc',relation='net_servers')
class pc(models.Model):
    _name = 'networks.pc'
    _description = 'PCs Model'
    name = fields.Char(default="PC")
    number = fields.Integer()
    ip = fields.Char()
    ping = fields.Float()
    registered = fields.Date()
    uptime = fields.Datetime()
    net = fields.Many2one('networks.net')
    user = fields.Many2one('res.partner')
    servers = fields.Many2many('networks.net',relation='net_servers')
```

Como se puede ver, están casi todos los tipos básicos de field. También podemos ver “fields relacionales”. Prestemos atención al “Many2one” llamado “net” de los “PC” que permite que funcione el “One2many” llamado “pcs” del modelo “networks.net”. También son interesantes los “Many2many” en los que declaramos el nombre de la relación para controlar el nombre de la tabla intermedia.

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

6.1. Definir una vista “tree” específica en los “X2many”

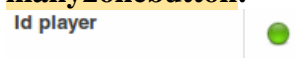
Los “One2many” y “Many2many” se muestran, por defecto, dentro de un formulario como una subvista “tree”. Odoo coge la vista “tree” con más prioridad del modelo al que hace referencia el “X2many” y la incrusta dentro del formulario. Esto provoca dos problemas:

- Si cambias esa vista también cambian los formularios.
- Las vistas “tree” cuando son independientes suelen mostrar más “fields” de los que necesitas dentro de un formulario que las referencia.

Por eso se puede definir un “tree” dentro del “field”:

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

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

- **many2onebutton:** indica solamente si está seleccionado.

- **many2many_tags:** que muestra los datos como etiquetas.



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

Ejercicio. Vamos a definir una relación entre categoría y libros de forma que un libro pertenece a una categoría y una categoría contiene muchos libros.

En la clase libros añade el siguiente campo:

```
categoria = fields.Many2one("libreria.categoria", string="Categoría", required=True, ondelete="cascade")
```

En la clase categoría añade el siguiente campo:

```
libros = fields.One2many("libreria.libro", "categoria", string="Libros")
```

Modifica la vista de libros (form) y añade el campo categoría:

```
<field name="categoria"/>
```

Modifica la vista de categoría (form) para que se visualicen los libros de esa categoría:

```

<field name="libros">
  <tree>
    <field name="name"/>
    <field name="precio"/>
    <field name="ejemplares"/>
  </tree>
</field>

```

En el siguiente video puedes ver cómo crear relaciones en Odoo entre los objetos libro y categoría:

<https://www.youtube.com/watch?v=HNsTUG-MWlc>

Para más información sobre las relaciones puedes visitar estas webs:

https://wiki.nuxpy.com/index.php/Campos_relacionales

Ejercicio. Importa los datos proporcionados en los archivos libros.xml y categorías.xls

7. VISTAS DE BÚSQUEDA.

Las vistas de búsqueda permiten redefinir la forma en la que buscamos en Odoo. Tenemos dos opciones a la hora de realizar una vista de búsqueda:

- **campos de búsqueda:** Simplemente son campos del modelo por los que es posible buscar. Por ejemplo, **descripción** dentro del objeto libro.
- **filtros.** Son búsquedas ya definidas con un **criterio definido por nosotros**. Digamos que el cliente está interesado en buscar siempre los libros con un precio menor o igual que cinco. Podríamos escribir entonces el siguiente filtro:

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

Veamos un ejemplo completo de vista “search”:

```

<search>
  <field name="name"/>
  <field name="inventor_id"/>
  <field name="description" string="Name and description" filter_domain="[('name', 'ilike', self), ('description', 'ilike', self)]"/>
  <field name="boxes" string="Boxes or @" filter_domain="[('boxes', '=', self), ('kg', '=', self)]"/>

```

```

<filter name="my_ideas" string="My Ideas" domain="[( 'inventor_id', '=', uid)]"/>
<filter name="more_100" string="More than 100 boxes" domain="[( 'boxes', '>', 100)]"/>
<filter name="Today" string="Today" domain="[( 'date', '>=',
datetime.datetime.now().strftime('%Y-%m-%d 00:00:00')), ('date', '<=',
datetime.datetime.now().strftime('%Y-%m-%d 23:23:59'))]"/>
<filter name="group_by_inventor" string="Inventor" context="{ 'group_by': 'inventor_id' }"/>
<filter name="group_by_exit_day" string="Exit" context="{ 'group_by': 'exit_day:day' }"/>
</search>

```

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

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

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

Ejercicio. Añade en el archivo views.xml la siguiente vista de búsqueda

```

<record model="ir.ui.view" id="libreria.libro_search_view">
  <field name="name">libreria.libro.search</field>
  <field name="model">libreria.libro</field>
  <field name="arch" type="xml">
    <search>
      <field name="name" string="Titulo"/>
      <field name="categoria" string="Categoria"/>
      <filter name="baratos" domain="[( 'precio', '<=', 5)]"/>
    </search>
  </field>
</record>

```

Utilizando dominios es posible escribir **expresiones lógicas complejas** que se adapten a cualquier necesidad.

Dominio → [(campo, operador, valor)]

En el dominio del filtro podemos utilizar todos los campos definidos en el modelo que queremos filtrar. Los operadores de evaluación disponibles en Odoo.

	Descripción	Operador	Descripción
>	Más grande que	like	Comparador de cadenas que distingue entre mayúsculas y minúsculas
<	Más pequeño que	ilike	Comparador de cadenas que no distingue entre mayúsculas y minúsculas
>=	Más grande o igual que	in	Contenido en (comparar un valor con una lista de valores)
<=	Más pequeño o igual que	not in	No contenido en (comparar un valor con una lista de valores)
=	Igual que	not	Negación (operador booleano que invierte el valor)
!=	Diferente de	child_of	Comparador utilizado en estructuras de tipo árbol

Algunos ejemplos de dominios son:

- [('qty_available', '<', 10)] – utilizado en la lista de productos para mostrar solo los productos con cantidad en stock menor que 10
- [('state', 'ilike', 'confirmed')] – utilizado en la lista de ventas para mostrar solo las ventas confirmadas
- [('categ_id', 'child_of', 5)] – utilizado en la lista de categorías de producto para mostrar solo las categorías que tienen como padre la categoría 5

En el siguiente video podéis ver cómo crear una vista de búsqueda para el módulo librería:

<https://www.youtube.com/watch?v=PbIyhtM0k8Q>

8. CAMPOS CALCULADOS.

Hasta ahora, los “fields” que hemos visto almacenaban algo en la base de datos. No obstante, puede que **no** queramos que algunos datos estén **guardados** en la **base de datos**, sino que se **recalculen cada vez** que vamos a verlos. En ese caso, hay que utilizar campos computados.

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

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

Para crear un campo calculado en Python debemos escribir una función dentro de la clase con el modelo, con una estructura determinada.

Ejercicio. Crea un capo calculado para el importe total de un libro

Para definirlo hay que crear un método que haga ese cálculo

```
importeTotal=fields.Float(string="Importe Total",compute="_importetotal",store=True)
@api.depends('precio','ejemplares')
def _importetotal(self):
    for r in self: #iteramos sobre el registro que nos llega
        r.importeTotal=r.ejemplares*r.precio
```

Debes añadirlo a la vista formulario de libro.

En el siguiente video puedes ver los detalles de cómo crear un campo calculado en Python:

<https://www.youtube.com/watch?v=fqHLAG9K4RA>

9. CREACIÓN DE INFORMES.

Al igual que en la UT4, podemos crear informes para nuestros objetos en Odoo. Un **informe** se crea a través del motor de informes **Qweb** y el resultado suele ser un **PDF** (aunque podría tener otro formato), que se le permite descargar al usuario.

Ejercicio. Crea un informe para el módulo librería

```
<odoo>
<data>
  <report
    id="report_libro"
    model="libreria.libro"
    string="informe libro"
    name="libreria.report_libro_view"
    file="libreria.report_libro"
    report_type="qweb-pdf"/>
  <template id="report_libro_view">
    <t t-call="web.html_container">
      <t t-foreach="docs" t-as="libro">
        <t t-call="web.external_layout">
          <div class="page">
            <h2 t-field="libro.name"/>
            <div>
              <strong>Precio:</strong>
              <span t-field="libro.precio"/>
            </div>
            <div>
              <strong>Ejemplares:</strong>
              <span t-field="libro.ejemplares"/>
            </div>
            <div>
              <strong>Categoría:</strong>
              <span t-field="libro.categoria"/>
            </div>
          </div>
        </t>
      </t>
    </t>
  </template>
</data>
</odoo>
```

En el manifest.py hay que añadir en data: 'reports/report_libro.xml'

En el siguiente video puedes ver todos los detalles sobre cómo crear un informe, en este caso para imprimir los detalles de un libro:

<https://www.youtube.com/watch?v=JR6OVqt4oko>

10. DATOS PRECARGADOS.

Este último apartado hace referencia a como establecer un **icono** para nuestro módulo y como **precargar datos** de cara a que el módulo no aparezca totalmente vacío.

Ejercicio. Añadir datos precargados.

Dentro de la carpeta data modificamos el archivo data.xml

```
<odoo>
<data>
  <record model="libreria.categoria" id="categoria_Novela">
    <field name="name">Novela</field>
  </record>
  <record model="libreria.categoria" id="categoria_ensayo">
    <field name="name">Ensayo</field>
  </record>
  <record model="libreria.libro">
    <field name="name">Alegría</field>
    <field name="categoria" ref="categoria_ensayo"/>
    <field name="ejemplares">10</field>
    <field name="precio">7</field>
  </record>
</data>
```

</odoo> Debemos añadir en el manifest en el apartado de data:

Data/data.xml

En el siguiente video puedes ver cómo realizar estos últimos pasos de manera sencilla y de esta manera retocar tu módulo y dejarlo preparado para su distribución:

https://www.youtube.com/watch?v=Qcf_5OhxnvU

11. EJERCICIOS NO GUIADOS

1. Modifica la relación entre categoría y libro a many2many y comprueba su funcionamiento.
2. Añade una constraint de Python para el campo ejemplares de forma que no se pueda introducir números negativos:

AYUDA

```
@api.constrains('ejemplares')
def _comprobar_ejemplares(self):
    for record in self:
        if record.ejemplares < 0:
            raise ValidationError("Mensaje")
```

3. Añade otras constraints a tu elección. Puedes hacerlo mediante sql_constraint o mediante Python.
4. Añade un campo calculado para Categoría de forma que sume el importe total de libros una categoría.
5. Modifica la vista de formulario de libros para que sólo se muestre el campo de estado en el caso de que el libro sea de segunda mano y que sea requerido.
6. Añade un campo editorial para los libros, debes relacionarlo con res_partner.
7. Añade un campo autores para libros, debes relacionarlo con res_partner.
8. Añade un campo tipo imagen para libros que se llame portada. Añade el campo a la vista formulario para poder añadirla.
9. Crea una vista Kanban para los libros y la categoría. En la vista de libros debe visualizarse la portada.
10. Haz que todos los usuarios tengan permisos de lectura sobre el modelo y sólo el grupo de librería responsable tenga de escritura y modificación.

12. BIBLIOGRAFÍA

<https://www.odoo.com/documentation/15.0/howtos/backend.html>

<https://re-odoo-10.readthedocs.io/capitulos/modelos-estructura-datos-aplicacion/>

<https://re-odoo-10.readthedocs.io/capitulos/vistas-disenar-la-interfaz/>

<https://castilloinformatica.es/wiki/index.php?title=Odoo>

https://www.odoo.com/documentation/15.0/es/developer/howtos/rdtraining/11_constraints.html

<https://www.odoo.com/documentation/15.0/es/developer/reference/backend/orm.html#odoo.fields.Field>