

Sistemas de Gestión Empresarial

UD 06. Desarrollo de vistas de Odoo.




Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 **Importante**

 **Atención**

 **Interesante**

ÍNDICE DE CONTENIDO

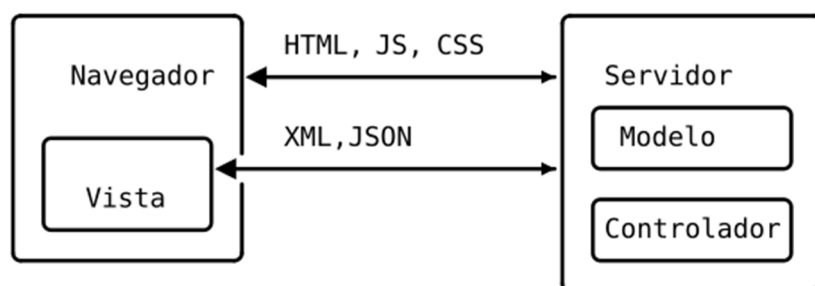
1.	Introducción	3
2.	Vista.	3
2.1	Vista list.	6
2.1.1	Colores en las líneas	6
2.1.2	Líneas editables	6
2.1.3	Campos invisibles	7
2.1.4	Cálculos de totales	7
2.2	Vista Form	7
2.2.1	Definir una vista “list” específica en los “X2many”.	8
2.2.2	Widgets	8
2.2.3	Valores por defecto en los One2many.	9
2.2.4	Domains en los Many2one.	10
2.2.5	Formularios dinámicos.	10
2.2.6	Botones con llamada a funciones del modelo.	10
2.3	Vista Kanban.	10
2.4	Vista Calendar	11
2.5	Vista Graph.	12
2.6	Vista Search.	12
3.	Seguridad en modelos Odoo	13
4.	Anexo QWeb	14
5.	Bibliografía	17

UD06. DESARROLLO DE VISTAS EN ODOO.

1. INTRODUCCIÓN

Las vistas son la interfaz que permite mostrar los datos contenidos en un modelo. Un modelo puede tener varias vistas, que simplemente son diferentes formas de mostrar los mismos datos.

2. VISTA.

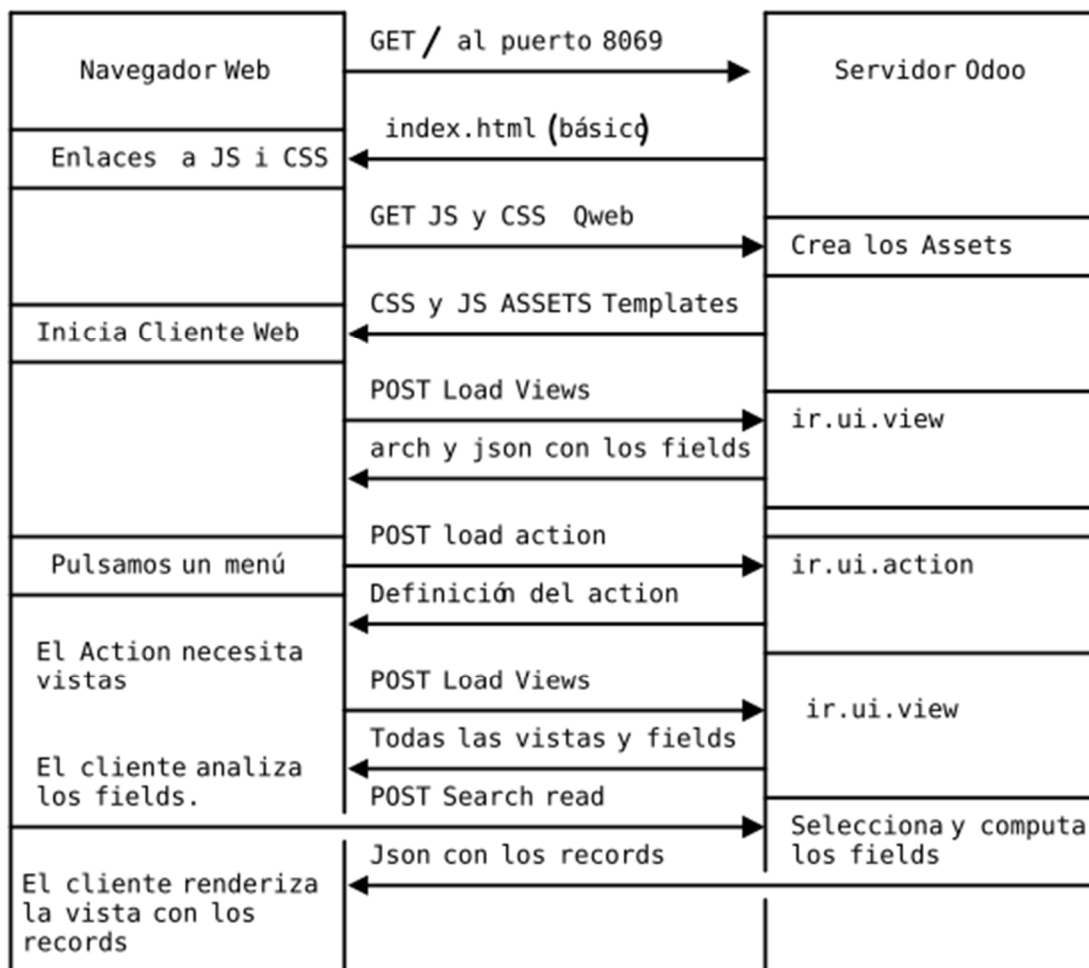


El esquema Modelo-Vista-Controlador que sigue Odoo, la vista se encarga de todo lo que tiene que ver con la interacción con el usuario. En Odoo, la vista es un programa completo de cliente en Javascript que se comunica con el servidor con mensajes breves. La vista tiene tres partes muy diferentes: El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

En la primera conexión con el navegador web, el servidor Odoo le proporciona un HTML mínimo, una SPA (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">
      <list>
        <field name="name"/>
        <field name="topics"/>
      </list>
    </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">list,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 “ñist” 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 “list 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 “list” definida y una vista “form” inventada por Odoo.

 **Interesante:** para poder observar un modelo en el cliente web de Odoo no necesitamos más que un menú que accione un “action” tipo “window” sobre ese modelo. Odoo es capaz de inventar las vistas básicas que nos permiten observarlo. No obstante, suelen ser menos atractivas y útiles que las que definimos nosotros.

Quedémonos por el momento con la definición básica de un “action window” y de los menús.

Vamos a centrarnos antes en las vistas.

2.1 Vista list.

La vista “list” 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">
    <list>
      <field name="name"/>
      <field name="topics"/>
    </list>
  </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 “**<list>**” 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 “list” se puede mejorar de muchas formas. Veamos algunas de ellas:

2.1.1 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

```
<list 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”.

💬 **Interesante:** se puede comparar un “field date” con una variable de “QWeb” llamada “**current_date**”: `<list decoration-info="start_date==current_date">`

2.1.2 Líneas editables

Si no necesitamos un formulario para modificar algunos “fields”, podemos hacer el “list”

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.

2.1.3 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="true"** en el “field” que necesitemos.

2.1.4 Cálculos de totales

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

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

```
<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">
    <list decoration-info="qualification<5" editable="top">
      <field name="name"/>
      <field name="topics" invisible="true"/>
      <field name="qualification" sum="Total Qualifications"/>
    </list>
  </field>
</record>
```

2.2 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 “list”. 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”.

2.2.1 Definir una vista “list” específica en los “X2many”.

Los “One2many” y “Many2many” se muestran, por defecto, dentro de un formulario como una subvista “list”. Odoo coge la vista “list” 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 “list” cuando son independientes suelen mostrar más “fields” de los que necesitas dentro de un formulario que las referencia.

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

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

2.2.2 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.

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”**.

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

- **many2onebutton**: indica solamente si está seleccionado.
- **many2many_tags**: que muestra los datos como etiquetas.

Reforc

grup1 x grup3 x

Id player



⚠ **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.

2.2.3 Valores por defecto en los One2many.

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

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

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

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

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

La palabra reservada **“active_id”** es una variable que apunta al “id” del registro que está activo en ese formulario. Para saber más de cómo usar “context” podéis consultar <https://www.cybrosys.com/blog/how-to-use-context-and-domain-in-odoo-13> y <https://odootricks.tips/about/building-blocks/context/>.

2.2.4 Domains en los Many2one.

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

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

2.2.5 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.

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”. Desde la versión 17 no admite ni el campo states, ni la etiqueta attr.

Ahora hay que colocar el atributo invisible, asociado al valor de un campo. Se puede poner aun campo o una agrupación.

```
<group invisible="state!='2'">
  <field name="name"/>
</group>
```

2.2.6 Botones con llamada a funciones del modelo.

Desde una vista se puede llamar a funciones del modelo que usa la vista. Para ello en la vista podemos incluir:


```
<div class="oe_button_box">
  <button name="regenerar" class="oe_stat_button" type="object" icon="fa-address-book"
string="Regenerar password">
  </button>
</div>
```

Sería necesario añadir una función en el modelo, en este caso llamada regenerar

2.3 Vista Kanban.

La vista “list” 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>
```

2.4 Vista Calendar

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

La sintaxis es muy simple, veamos un ejemplo:

```
<record model="ir.ui.view" id="school.travel_calendar">
```

```

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

```

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

2.5 Vista Graph.

La vista “graph” permite mostrar una gráfica a partir de algunos “fields” numéricos que tenga el modelo. Esta vista puede ser de tipo “pie” (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>

```

2.6 Vista Search.

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

Veamos un ejemplo completo de vista “search”:

```

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

```

```

"['|', ('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.

3. SEGURIDAD EN MODELOS ODOO

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 una 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/rdrtraining/05_securityintro.html

https://www.odoo.com/documentation/16.0/es/developer/tutorials/getting_started/05_securityintro.html

4. ANEXO QWEB

En Odoo, un template es un archivo que define la estructura y presentación de la interfaz de usuario, ya sea para el frontend (como vistas web, formularios, o páginas de un sitio web) o para el backend (informes, vistas de registros, etc.). Los templates en Odoo son fundamentales para la generación dinámica de contenido y se utilizan tanto para la creación de vistas como para la personalización de los informes en formato HTML y PDF.

Odoo utiliza su propio motor de plantillas, llamado QWeb, para renderizar estas plantillas. QWeb es un motor basado en XML que permite separar la lógica de presentación del código de la aplicación, lo que facilita la reutilización de componentes y la personalización de las vistas.

Las etiquetas QWeb son un conjunto de directivas que se usan para generar y manipular contenido HTML en Odoo mediante su motor de plantillas, QWeb. QWeb es el motor de plantillas de Odoo utilizado tanto en el frontend (para la interfaz de usuario) como en el backend (para la generación de informes, vistas, etc.).

QWeb tiene una sintaxis propia basada en XML que permite renderizar dinámicamente contenido y manejar lógica en el lado del servidor. A continuación, te explicaré las etiquetas más comunes de QWeb que puedes utilizar para estructurar tus plantillas en Odoo.

Principales Etiquetas QWeb

1. **<t>**: La etiqueta principal de QWeb

La etiqueta **<t>** es la etiqueta principal para usar QWeb en Odoo. Se utiliza para definir bloques de código y lógica dentro de las plantillas.

2. **<t-name>**: Define el nombre del template.

```
<t t-name="kanban-box">
```

3. **<t-esc>**: Escapar y mostrar contenido

La etiqueta `<t t-esc="expression"/>` es utilizada para mostrar el valor de una expresión. Escapa el contenido automáticamente, es decir, convierte los caracteres especiales en entidades HTML para evitar inyecciones de código.

```
<t t-esc="product.name" />
```

Esto muestra el nombre de un producto de manera segura.

4. **<t-set>**: Establecer una variable

La etiqueta `<t t-set="var_name" t-value="value"/>` permite definir variables dentro de un template. Estas variables son útiles cuando necesitas almacenar valores intermedios.

```
<t t-set="greeting" t-value="Hello, ' + user.name"/>
<p><t t-esc="greeting"/></p>
```

5. **<t-foreach>**: Iterar sobre una lista

Usada para iterar sobre listas u objetos iterables. Similar a un bucle for en Python.

```
<t t-foreach="products" t-as="product">
  <div>
    <t t-esc="product.name"/>
  </div>
</t>
```

En este ejemplo, products es una lista de objetos, y el bucle itera sobre ella, asignando a cada elemento el alias product.

6. **<t-if>** y **<t-else>**: Condicionales

Se utilizan para implementar lógica condicional dentro de las plantillas.

```
<t t-if="user.logged_in">
  <p>Welcome, <t t-esc="user.name" />!</p>
</t>
<t t-else="">
  <p>Please log in to continue.</p>
</t>
```

En este caso, si user.logged_in es True, se muestra el mensaje de bienvenida, de lo contrario, se muestra el mensaje para que el usuario inicie sesión.

7. **<t-raw>**: No escapar contenido

La etiqueta `<t t-raw="expression"/>` permite mostrar contenido sin escapar (usado para insertar HTML o texto con etiquetas sin procesar).

```
<t t-raw="product.description"/>
```

8. **<t-call>**: Llamar a otro template

Permite invocar otros templates dentro de un template actual. Es útil para crear componentes reutilizables.

```
<t t-call="other_template"/>
```

Esto inserta el contenido de other_template dentro del template donde se llama.

9. **<t-include>**: Incluir otro archivo de plantilla

Permite incluir otros archivos de plantilla, de forma similar a t-call, pero con la diferencia de que t-include carga el template de manera independiente.

```
<t t-include="module.template_name"/>
```

10. **<t-debug>**: Depuración

Es útil para mostrar información de depuración durante el desarrollo de templates.

```
<t t-debug="True">This is a debug message</t>
```

11. **<t-att-*>**: Atributos dinámicos

Se usan para establecer atributos dinámicos en los elementos HTML. Son muy útiles cuando necesitas configurar dinámicamente atributos como class, src, href, etc.

```

```

En este caso, el atributo src del elemento img se configurará con el valor de la variable image_url.

Sintaxis Básica

Ejemplo Completo:

```
<t t-name="product_template">
  <div class="product">
    <h1><t t-esc="product.name"/></h1>
    <p><t t-if="product.price > 100">Price: <t t-esc="product.price"/></t></p>
    <t t-foreach="product.features" t-as="feature">
      <li><t t-esc="feature"/></li>
    </t>
  </div>
</t>
```

Explicación:

<t t-name="product_template">: Define el nombre del template.

<t t-esc="product.name"/>: Muestra el nombre del producto.

<t t-if="product.price > 100">: Muestra el precio si el precio del producto es mayor que 100.

<t t-foreach="product.features" t-as="feature">: Itera sobre la lista product.features y muestra cada característica del producto.

Consideraciones Adicionales

Inyecciones de datos: Al usar t-esc, asegúrate de que los datos se escapen correctamente para

evitar inyecciones de código malicioso (XSS). Si quieres mostrar HTML sin escapar, puedes usar t-raw.

Composición de templates: Es recomendable estructurar tu aplicación utilizando la composición de plantillas. Usa t-call y t-include para organizar tus componentes y reutilizarlos fácilmente.

Desempeño: QWeb está diseñado para ser eficiente, pero el uso excesivo de operaciones como t-foreach o t-if puede afectar el rendimiento, especialmente si las listas son grandes. Trata de minimizar los bucles y condicionales dentro de las plantillas y realiza la mayor parte de la lógica en el backend si es posible.

5. BIBLIOGRAFÍA

<https://www.odoo.com/documentation/master/>

https://ioc.xtec.cat/materials/FP/Materials/2252_DAM/DAM_2252_M10/web/html/index.html

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

<https://konodoo.com/blog/konodoo-blog-de-tecnologia-1>