

# Proyecto Final TFG

## Módulo en Odoo sobre las ideas



**Pablo Marín Aleixandre**

**Desarrollo de Aplicaciones Multiplataforma (DAM)**

**CIP FORMACIÓ PROFESSIONAL BATOI**

**Antonio Vicente Santos Silvestre**

**Curso 2023 - 2024**

# Índice

<b>Introducción.....</b>	<b>3</b>
<b>¿Qué es Odoo?.....</b>	<b>4</b>
<b>Demostración gráfica del módulo.....</b>	<b>5</b>
<b>Explicación de mi módulo.....</b>	<b>9</b>
Estructura básica.....	9
Modelo - idea.management.....	10
Campos.....	10
Funciones.....	16
Modelo - idea.management.vote.....	20
Campos.....	20
Funciones.....	22
Vistas.....	24
Modelo - idea.management.....	24
Tree.....	24
Form.....	25
Search.....	29
Calendar.....	30
Pivot.....	31
Graph.....	32
Kanban.....	34
idea_management_vote_view.xml.....	37
Form.....	37
Report y vista - Ideas.....	39
Security.....	42
<b>Recursos.....</b>	<b>45</b>
<b>Necesidades y sugerimientos de la formación.....</b>	<b>46</b>
<b>Opinión personal.....</b>	<b>47</b>
<b>Conclusión.....</b>	<b>48</b>
<b>Bibliografía.....</b>	<b>49</b>

# Introducción

Me llamó la atención la idea de imaginar una situación en la que los usuarios de una empresa no estuvieran constantemente enviando correos electrónicos internos para compartir ideas y sugerencias. Me surgió entonces la idea de desarrollar un módulo dentro de Odoo que pudiera funcionar como una plataforma de "lluvia de ideas". Este módulo estaría diseñado para permitir a los usuarios de Odoo, así como a los empleados de la empresa en general, expresar sus ideas de manera centralizada y estructurada.

En este escenario, los administradores de Odoo serían responsables de evaluar las ideas presentadas dando su apto o no para dicha idea. Aquellas ideas que fueran consideradas apropiadas pasarían a una fase de votación, en la que los demás usuarios tendrían la oportunidad de expresar su opinión y votar por las propuestas que consideraran más prometedoras.

Además de la votación, se proporcionaría un espacio en la votación para que los usuarios ofrecieran feedback sobre cada idea, lo que permitiría mejoras antes de su implementación. De esta manera entre todos los usuarios, se podrían pulir las ideas propuestas, y además también fomentaría e incentivaría a un ambiente de colaboración y creatividad dentro de la empresa.

La implementación de este módulo de "lluvia de ideas" en Odoo no solo simplificaría el proceso de generación y evaluación de ideas, sino que también ayudaría a que los miembros de la empresa sean más participativos, en la que todos se sientan valorados y motivados a contribuir con sus ideas para el crecimiento y la mejora continua de la empresa.

## ¿Qué es Odoo?

Bien, para quien no conozca Odoo, es un sistema de gestión empresarial (ERP) de código abierto que proporciona una lista de aplicaciones muy amplia diseñadas para gestionar diversas áreas de una empresa, como ventas, compras, inventario, contabilidad, etc.

El propósito principal de Odoo es centralizar y automatizar los procesos de la empresa, lo que ayuda a mejorar la eficiencia, reducir los costos y facilitar la toma de decisiones. Gracias a Odoo, las empresas pueden gestionar fácilmente sus operaciones diarias, como por ejemplo la gestión de clientes, la facturación o incluso el seguimiento de inventario.

Odoo es muy potente, personalizable y escalable, lo que significa que puede adaptarse a las necesidades específicas y concretas de cada empresa, ya sea una pequeña empresa, una empresa mediana o una gran empresa multinacional. Además, otro gran punto a favor, es que es de código abierto, por ello, ofrece flexibilidad para que los desarrolladores puedan personalizar y ampliar sus funcionalidades según los requisitos del negocio.

## Demostración gráfica del módulo

En primer lugar me gustaría decir que para instalar mi módulo deberemos de seguir las instrucciones que se encuentra en el Readme.md que se encuentra en el repositorio de Github: [https://github.com/Pablofasl7/ideas\\_module.git](https://github.com/Pablofasl7/ideas_module.git)

Ahora una vez lo tengamos instalado, veremos que no nos aparece en el menú de aplicaciones, no hay que preocuparse, pero bien, ¿esto a que se debe?

Lo que debemos de hacer es irnos a los ajustes >> usuarios y a cada usuario asignarle los permisos que nosotros creamos correspondientes, el menú siguiente se nos creará en el menú de permisos del usuario:

IDEA	
Acceso para votar de los managers ?	<input type="checkbox"/>
Manager ?	<input type="checkbox"/>
Acceso para votar de los usuarios ?	<input type="checkbox"/>
User ?	<input type="checkbox"/>

### - Empezando por el permiso de **User**:

Los miembros de este grupo tienen el permiso para votar una vez que las ideas han sido aprobadas. Esto permite que los usuarios participen en el proceso de votación de ideas ya validadas por los administradores.

### - Seguimos con el grupo de **Manager**:

El grupo 'Manager' está diseñado para los administradores del sistema. Este grupo incluye todos los permisos del grupo 'User', permitiendo a los managers no solo aprobar ideas, sino también votar en ellas. Además, se asigna el usuario root (base.user\_root) como miembro del grupo "Manager", lo que implica que este usuario tiene todos los derechos administrativos sobre la gestión de ideas.

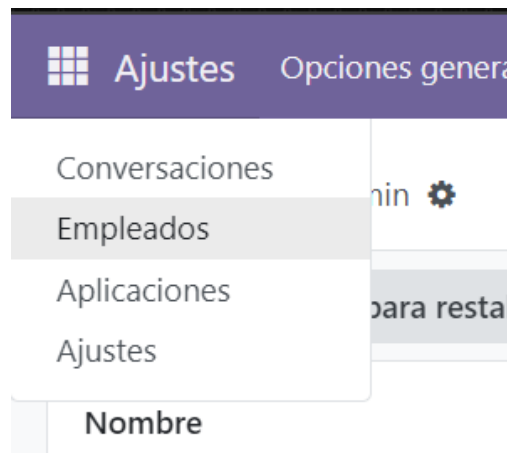
### - Ahora pasamos con el permisos de **votar** para los **usuarios**:

Este grupo define un permiso específico para que los usuarios puedan votar en las ideas. Los miembros de este grupo pueden participar en la votación de ideas que están disponibles para recibir votos.

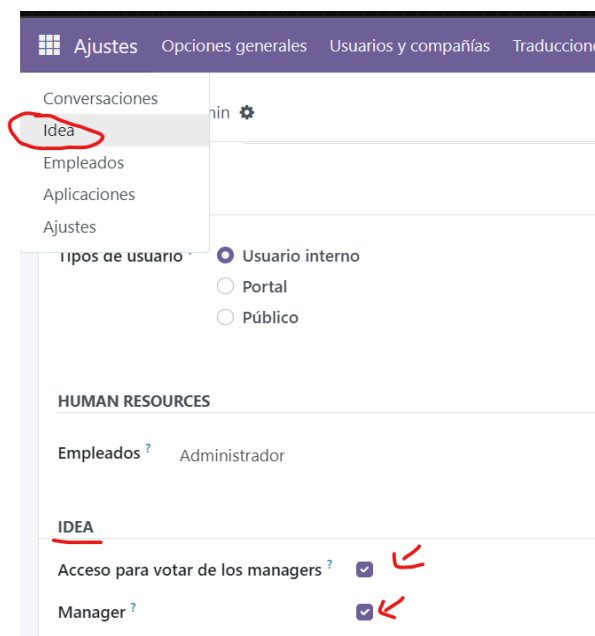
Y ahora por último vamos con los permisos para **votar** de los **managers**:

Este grupo está destinado a los administradores con permisos avanzados. El grupo incluye todos los permisos del grupo 'Acceso para votar de los usuarios', permitiendo a los managers votar y gestionar los votos en las ideas. También se asigna el usuario root (base.user\_root) como miembro de este grupo, asegurando que los administradores principales tengan la capacidad de supervisar y gestionar el proceso de votación.

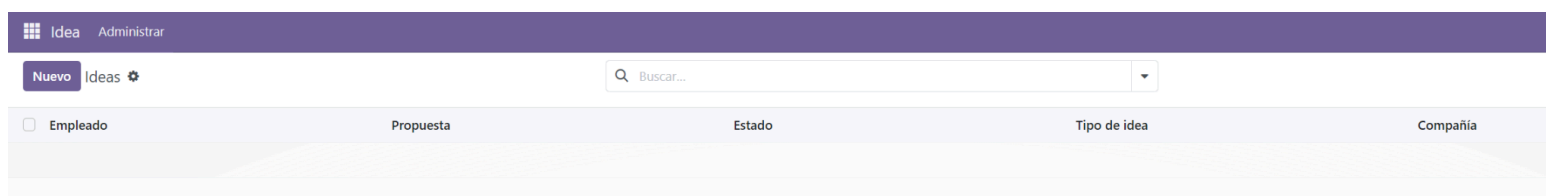
Como podemos apreciar si no selecciono ningún permiso para mi usuario, el módulo no me aparecerá:



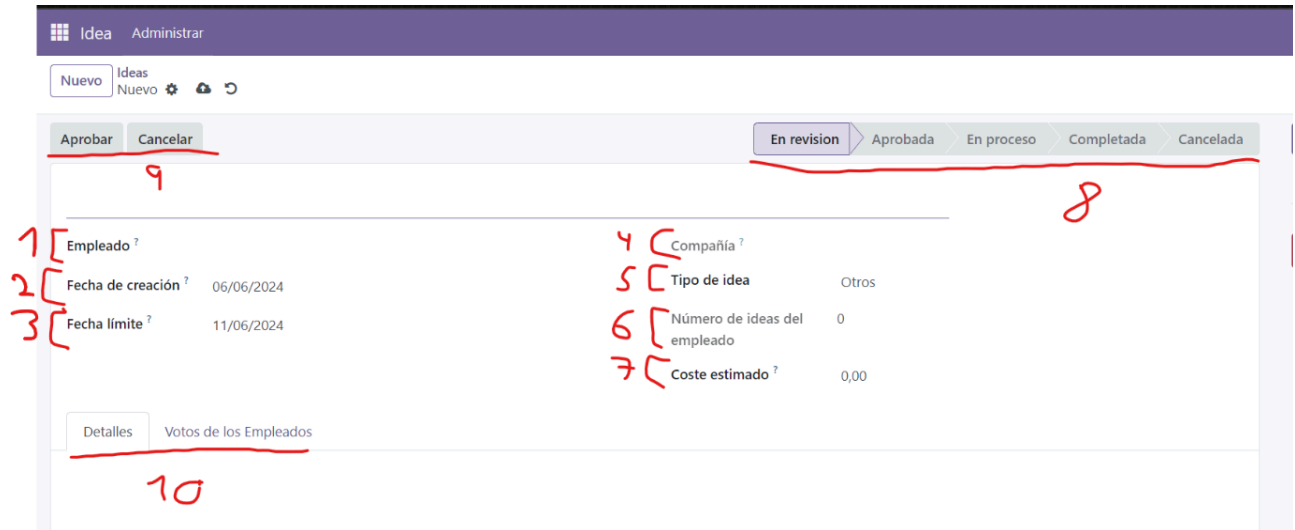
Pero ahora sin embargo, si aplicamos algunos permisos, veremos como ya nos aparece en el menú de aplicaciones:



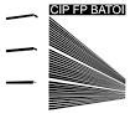
Bien, una vez entremos en el módulo, nos va a aparecer la vista **tree**, que explicaremos más adelante:



Y si vamos a crear una idea se nos abrirá la vista **form** que desde ahí podremos rellenar los campos de la idea y la crearemos, ahora vamos a pasar a explicar brevemente los campos que tenemos en la vista form:



1. **Empleado:** este campo es un 'Many2one' que se relaciona con el modelo 'hr.employee'. Representa al empleado que ha propuesto la idea.
2. **Fecha de creación:** Este campo almacena la fecha en la que la idea fue creada. Su valor por defecto es la fecha actual.
3. **Fecha límite:** Este campo almacena la fecha límite para la evaluación o implementación de la idea
4. **Compañía:** Aquí tenemos otro campo 'Many2one' que se relaciona con el modelo 'res.partner'. Representa la compañía asociada con la idea.
5. **Tipo de idea:** Ahora tenemos un campo que es una selección que permite categorizar la idea en tres tipos: mejoras, plantear proyecto, y otros.
6. **Número de ideas del empleado:** Ahora pasamos con un campo calculado, que muestra la cantidad de ideas propuestas por el mismo empleado.
7. **Price:** Aquí almacenamos el coste estimado de implementar la idea.



8. **Estado:** Este campo es una selección que indica el estado actual de la idea. Los valores son: En revisión, Aprobada, En proceso, Completada, y Cancelada.
9. **Botones:** Aquí tenemos los botones que nos van a permitir cambiar el estado de la idea.
10. **Pages:** Aquí tenemos distintas páginas, una llamada “Detalles” por si el empleado que ha dado la idea o cualquier administrador quiere dar explicaciones extras sobre la idea, o la otra page llamada “Votos de los empleados”, que ahí es donde se registraran los votos de todos los empleados que quieran votar en el plazo de votación de la idea, aquí un ejemplo de cómo se vería un voto:

<div>Detalles</div> <div>Votos de los Empleados</div>		
Empleado	Valoraciones	Comentarios
Mitchell Admin	Excelente	Espectacular.



## Explicación de mi módulo

### Estructura básica

Empezamos con la estructura básica como son los ficheros ‘\_\_init\_\_.py’ y el ‘\_\_manifest\_\_.py’ vamos a empezar con el primer fichero init que está en la carpeta base del módulo:

```
from . import models
```

Vamos a tener que importar los modelos (en mi caso el ‘models’ es el nombre de la carpeta donde almaceno los modelos), por otro lado, necesitamos otro init en la carpeta models, para así poder cargar nuestros modelos:

```
from . import idea_management
from . import idea_management_vote
```

Y ahora por último tenemos el archivo manifest:

```
# -*- coding: utf-8 -*-
# Copyright <2024> <Pablo Marín>
# License AGPL-3.0 or later (https://www.gnu.org/licenses/agpl).
{
    "name": "Ideas module",
    "version": "1.0",
    "author": "Pablo Marín",
    "license": "AGPL-3",
    "depends": [
        "base",
        "mail",
        "hr",
    ],
    "data": [
        "security/idea_security.xml",
        "security/ir.model.access.csv",
        "reports/idea_management_report_templates.xml",
        "views/idea_menu.xml",
        "views/idea_management_view.xml",
        "views/idea_management_vote_view.xml",
        "views/idea_management_report_view.xml",
    ],
}
```

Este fichero contiene la configuración y los metadatos necesarios para que Odoo instale el módulo correctamente, en mi caso tenemos el nombre del módulo, la versión, al autor, y la licencia utilizada. Seguimos con el “depends” (las dependencias), que son una lista de otros módulos de Odoo que deben estar instalados para que este módulo funcione de manera correcta. En este caso, depende de los módulos base, mail y hr.

Y por último tenemos el “data” (los datos), son una lista de archivos xml y csv que deben cargarse al instalar el módulo. Y estos archivos sirven para configurar varias partes del módulo, como las vistas, menús...

## Modelo - idea.management

### Campos

```
_name = 'idea.management'
_description = 'Idea'
_inherit = ['mail.thread.cc',
            'mail.thread.blacklist',
            'mail.activity.mixin']

_primary_email = 'email_from'
vote_ids = fields.One2many('idea.management.vote',
'idea_id', string='Votos', help="Votos de los empleados.")

name = fields.Char(string = 'Propuesta', required=True, help = 'Esto
es el nombre de la propuesta / idea')

create_date = fields.Date(string = 'Fecha de creación',
default=_date_default_today, help = 'Fecha de creación')

deadline = fields.Date(string = 'Fecha límite', help = 'Fecha de
finalización')
```

Empezamos creando la Clase y le asignamos un “\_name” que así es como se guardará el nombre del modelo en la base de datos, seguido, también tenemos la descripción del modelo.

Por otro lado tenemos varios campos que los vamos a explicar a continuación:

**\_inherit:** Esta línea quiere decir que el modelo hereda funcionalidades de otros modelos ya creados en el sistema, en este caso para manejar actividades relacionadas con correos electrónicos.

**\_primary\_email:** Esta línea especifica cuál es el campo principal que se utilizará para identificar los correos electrónicos relacionados con este modelo.

**vote\_ids:** En este campo, se establece una relación uno a muchos con otro modelo llamado 'idea.management.vote' (que lo explicaré más adelante). Este campo permite registrar los votos de los empleados en relación con las ideas.

**name:** Ahora, definimos un campo de tipo carácter (texto), que almacena el nombre de la idea/propuesta. Este campo es obligatorio completar y desde odoo tendremos visible la información de “help” si pasamos el cursor por encima del campo.

**create\_date:** En esta línea definimos un campo de tipo fecha, que registra la fecha en que se creó la idea/propuesta. Por defecto, tomamos la fecha actual como valor (gracias a la función que voy a declarar a continuación):

```
def _date_default_today(self):
    return fields.Date.today()
```

**deadline:** Establecemos otro campo de tipo fecha, que indica la fecha límite para la idea/propuesta.

Bien, una vez explicados estos campos, vamos a seguir con más campos que tengo declarados:

```
days_duration = fields.Integer(compute='_compute_days_duration',
string='Duración (días): ', store=True)

idea_type = fields.Selection(
    [('mejoras', 'Mejoras'),
    ('proyecto', 'Plantear proyecto'),
    ('otros', 'Otros'),],
    string="Tipo de idea",
    default="otros")

details = fields.Text(help = 'Descripción de la idea')
price = fields.Float(string="Coste estimado", help = 'Coste estimado
de la idea')

state = fields.Selection(
    [('revision', 'En revision'),
    ('aprobada', 'Aprobada'),
    ('proceso', 'En proceso'),
    ('completada', 'Completada'),
    ('cancelada', 'Cancelada')],
    string = 'Estado',
    default = 'revision')
```

**days\_duration:** En esta línea creamos un campo llamado 'days\_duration' que almacenamos la duración en días del plazo de validez que tiene la idea. Este campo es de tipo entero, y además, se define que este campo será calculado por una función llamada '\_compute\_days\_duration' (la mostraré a continuación).

También se indica que el campo se almacena en la base de datos mediante el store=True.

Ahora vamos a explicar la función computada de “\_compute\_days\_duration”:

```
@api.depends('create_date', 'deadline')
def _compute_days_duration(self):
    for idea in self:
        if idea.create_date and idea.deadline:
            duration = (idea.deadline - idea.create_date).days
            idea.days_duration = duration if duration >= 0 else 0
        else:
            idea.days_duration = 0
```

En la primera línea, mediante el api.depends, estamos diciéndole a la función que tiene que depender de los campos 'create\_date' y 'deadline', esto se verá reflejado en que cuando el valor de los campos cambie, la función se ejecutará automáticamente para mostrar el valor real de 'days\_duration'.

Por otro lado hacemos un bucle para iterar sobre cada instancia de la clase idea dentro del conjunto de instancias de self. Después nos tenemos que encargar de comprobar si tanto la fecha de creación como la fecha límite de la idea existen, si ambas fechas están presentes, se realiza el cálculo de la duración.

La diferencia se calcula entre la fecha límite y la fecha de creación (haciendo una resta), y el “.days” convierte el resultado en días enteros.

Y por último, el “duration if duration >= 0 else 0”: asigna el resultado del cálculo al campo days\_duration de la idea. Si la duración es mayor o igual a cero, se asigna ese valor. En caso de que no, se asigna cero, lo que quiere decir que la duración se establecerá como cero si la fecha límite es anterior a la fecha de creación.

**idea\_type:** Aquí definimos un campo de selección, que nos permite elegir entre diferentes tipos de ideas (que son los valores que hemos establecido). Este campo proporciona una lista de opciones que podemos elegir (en este caso: “mejoras”, “proyecto” u “otros”).

**details:** Esta línea crea un campo de texto, que permite añadir una descripción detallada de la idea. Por otro lado, el parámetro 'help' proporciona una breve descripción o consejo sobre la función de este campo.

**price:** Aquí definimos un campo de tipo float (número decimal), que almacena el coste estimado asociado a la idea. El parámetro 'string' define la etiqueta que se mostrará en la interfaz de usuario para identificar este campo, mientras que el parámetro 'help' proporciona más información sobre su uso (que la podemos ver poniendo el cursor encima del campo).

**state:** Aquí tenemos otro campo selection, que representa el estado actual de la idea. Puede estar en varios estados (En revisión, Aprobada, En proceso, Completada o Cancelada). El estado por defecto es 'En revisión'.

Vamos a seguir con más campos que tengo declarados en mi modelo:

```
active = fields.Boolean(string='Activo', default=True, help = 'Para
archivar la idea.')

archive_cancelled = fields.Boolean(compute='_compute_archive_cancelled',
store=True)

assigned = fields.Boolean(string = 'Assigned',
compute='_compute_assigned')

employee_id = fields.Many2one(comodel_name='hr.employee',
string='Empleado', help="Empleado que ha tenido la idea.")

partner_id = fields.Many2one(comodel_name='res.partner',
string='Compañía', compute='_compute_compañía', store=True, help =
'Compañía')

email_from = fields.Char(string='Email from')
voter_id = fields.Many2one('hr.employee', string='Empleado que vota',
help="Empleado que ha votado la idea.")
```

**active:** Este campo es booleano y se utiliza para indicar si una idea está activa o no. Por defecto, se establece en True, lo que significa que la idea está activa. La etiqueta string define el nombre visible del campo en la interfaz de usuario. Y como ya hemos comentado antes, el parámetro “help” proporciona una breve descripción del campo.

**archive\_cancelled:** Este campo también es booleano y además, se calcula automáticamente con una función computada (\_compute\_archive\_cancelled) y se almacena en la base de datos mediante el store=True.

Esta es la función computada:

```
@api.depends('state', 'active')
def _compute_archive_cancelled(self):
    for idea in self:
        if idea.state == 'cancelada' and idea.active:
            idea._archive_idea()

def _archive_idea(self):
    for idea in self:
        idea.active = False
```

- **\_compute\_archive\_cancelled:** Se activa cuando cambian los estados de la idea o su estado de activación (state y active). Si el estado de la idea es 'cancelada' y está activa, entonces llama a la segunda función \_archive\_idea para archivar la idea.
- **\_archive\_idea:** Y esta otra función, se encarga de desactivar la idea (mediante el active = False). Es decir, esta función archiva la idea.

**assigned:** Este campo es similar al anterior, también es booleano, pero este campo es calculado automáticamente a través de la función “\_compute\_assigned”. Se utiliza para indicar si una idea ha sido asignada a alguien.

Aquí tenemos la función computada:

```
@api.depends('employee_id')
def _compute_assigned(self):
    for record in self:
        record.assigned = self.employee_id and True or False
```

- **\_compute\_assigned:** Se activa cuando cambia el campo employee\_id. Utiliza el @api.depends para indicar que depende del campo employee\_id. Para cada registro en el conjunto de registros representado por self, comprobamos si employee\_id tiene valor. Si tiene un valor, asigna True al campo ‘assigned’ del registro actual, pero de lo contrario, le asignamos False.

**employee\_id:** Este campo Many2one se utiliza para relacionar la idea con empleado del modelo hr.employee (un registro), se utiliza para indicar qué empleado ha tenido la idea.

**partner\_id:** Este campo es Many2one, y se utiliza para relacionar la idea con un registro en el modelo res.partner, que representa una compañía.

También hacemos uso de una función compute “\_compute\_compañía” para que el valor de este campo se calcule automáticamente basándose en el employee\_id. Además, se almacena en la base de datos (store=True).

Función computada:

```
@api.depends('employee_id')
def _compute_compañía(self):
    for record in self:
        if record.employee_id:
            record.partner_id =
record.employee_id.company_id.partner_id
        else:
            record.partner_id = False
```

- **\_compute\_compañía:** Esta función se ejecuta cada vez que el campo ‘employee\_id’ de un registro cambia. Como ya hemos comentado anteriormente, lo que se encarga de hacer esta función es establecer el valor del campo partner\_id (que es la compañía del employee\_id).

Es decir, employee\_id tiene valor, el partner\_id se establece como la compañía asociada al empleado. Y en el caso de que el employee\_id esté vacío, el partner\_id se establece como Falso.

**email\_from:** Este campo de tipo Char (Texto) almacena la dirección de correo electrónico desde la que se recibió la idea.

**voter\_id:** Este campo es bastante similar al campo employee\_id, lo utilizamos también para relacionar la idea con un empleado en el modelo hr.employee, pero en este caso, representa al empleado que ha votado por la idea.

## Funciones

Ahora vamos a pasar con las **funciones** que tengo declaradas dentro de mi modelo:

```
def aprobar(self):
    self.ensure_one()
    self.write({
        'state': 'aprobada'})
```

**aprobar:** Esta función se utiliza para cambiar el estado de una idea de 'En revisión' a 'Aprobada'. Primero, vemos el 'self.ensure\_one()', que asegura que estamos tratando con un solo registro a la vez.

Después, con el self.write() actualizamos el campo 'state' del registro actual a 'aprobada'.

```
def proceso(self):
    self.ensure_one()
    self.state = 'proceso'
```

**proceso:** Esta función se utiliza para cambiar el estado de una idea a 'En proceso'. Igual que la función 'aprobar', el self.ensure\_one() se asegura que solo estamos tratando con un registro. Y después, modificamos el valor del 'state' directamente, mediante el self.state = 'proceso'.

```
def completada(self):
    self.ensure_one()
    self.state = 'completada'
```

**completada:** Esta función cambia el estado de una idea a 'Completada'. También nos aseguramos de que solo estamos trabajando con un registro. Y después, modificamos el valor del 'state' directamente: self.state = 'completada'.

```
def cancelada(self):
    self.ensure_one()
    self.state = 'cancelada'
```

**cancelada:** Esta función cambia el estado de una idea a 'Cancelada'. Y como en todas las demás funciones, primero nos aseguramos que solo estamos tratando con un único registro. Y después, modificamos el valor del 'state' directamente como en las demás funciones: self.state = 'cancelada'.



```
@api.constrains('price')
def _price_positive(self):
    if self.price < 0:
        raise ValidationError(_("El coste no puede ser negativo."))
```

**\_price\_positive:** Esta función se ejecuta cada vez que el campo 'price' de un registro cambia. De lo que se encarga es garantizar que el valor del campo price no sea negativo, y en el caso de que 'price' sea negativo, crearemos y lanzaremos una excepción de validación que muestra un mensaje indicando que el coste no puede ser negativo.

```
@api.constrains('deadline')
def _check_deadline(self):
    for record in self:
        if record.deadline and record.deadline < fields.Date.today():
            raise ValidationError("La fecha límite no puede ser anterior a la fecha actual.")
```

**\_check\_deadline:** Esta función se ejecuta cada vez que el campo 'deadline' de un registro cambia. Se encarga de asegurar que la fecha límite ('deadline') no sea anterior a la fecha actual. En el caso de que la fecha límite sea anterior a la fecha actual, como en el caso anterior, se crea y lanzamos una excepción de validación que indica que la fecha límite no puede ser anterior a la fecha actual.

```
@api.onchange('create_date')
def _update_deadline(self):
    if self.create_date:
        self.deadline = self.create_date + timedelta(days=5)
```

**\_update\_deadline:** Esta función se ejecuta cuando cambia el campo 'create\_date'. Y lo que realiza esta función es automatizar la acción de añadir una fecha límite, en este caso suma cinco días a la fecha actual y establece este valor como la fecha límite (deadline). De esta manera, cuando se cree una nueva idea, y se establece la fecha de creación, automáticamente se añadirá una fecha límite (5 días mas que la fecha de creación)

```
@api.depends('deadline')
def _check_deadline_expired(self):
    for record in self:
        if record.deadline and record.deadline < fields.Date.today():
            record.state = 'completada'
```

**\_check\_deadline\_expired:** Esta función se ejecuta cuando cambia el campo 'deadline'. Se encarga de comprobar si la fecha límite es anterior a la fecha actual. En caso que sea verdad, cambia el estado de la idea (state) a 'completada', en otras palabras, nos referimos a que la idea ha superado su fecha límite y se marca como completada automáticamente.

```
def open_vote_form(self):
    view_id = self.env.ref('ideas_module.view_idea_vote_form').id

    return {
        'name': 'Voto del Empleado',
        'view_type': 'form',
        'view_mode': 'form',
        'res_model': 'idea.management.vote',
        'views': [(view_id, 'form')],
        'type': 'ir.actions.act_window',
        'target': 'new',
        'context': {
            'default_idea_id': self.id,
            'default_idea_name': self.name,
            'default_employee_id': self.env.user.employee_id.id,
        },
    }
```

**open\_vote\_form:** Esta función se utiliza para abrir un formulario (de votación) relacionado con la idea específica. Lo que tenemos que tener en cuenta sobre esta función son:

- **view\_id:** Se obtiene el ID de la vista que utilizaremos para mostrar el formulario de votación.
- El **diccionario** que devolvemos, contiene información sobre cómo vamos a mostrar el formulario de votación, incluyendo el nombre de la ventana, el tipo y modo de vista, el modelo de datos que vamos a utilizar, las vistas a mostrar, el tipo de acción, el destino de la acción y el contexto.
- El **contexto** lo utilizamos para pasar información adicional al formulario de votación, en este caso el ID, el nombre de la idea y también el ID del empleado que está votando. Esto permite rellenar automáticamente ciertos campos del formulario con esa información que es bastante útil.

```
ideas_empleado = fields.Integer(string='Número de ideas del empleado',
compute='_compute_ideas_employee')

@api.depends('employee_id')
def _compute_ideas_employee(self):
    for record in self:
        other_tickets= self.env['idea.management'].search([('employee_id',
            '=', record.employee_id.id)])
        record.ideas_empleado = len(other_tickets)
```

Antes de explicar la función, declaramos un campo computado llamado “ideas\_empleado”, que lo que hace es contar el número de ideas que tiene asociadas un empleado.

Ahora, pasando con la función computada, se ejecuta cada vez que cambia el campo ‘employee\_id’. Dentro de esta función, se busca en el modelo **idea.management** todas las ideas que tienen el mismo employee\_id que el registro actual y se cuenta cuántas hay. Y el número que nos da, es el que se asigna al campo ideas\_empleado.

## Modelo - idea.management.vote

### Campos

```
_name = 'idea.management.vote'
_description = 'Idea Management Vote'

rating = fields.Selection(
    [('0', 'Very Low'), ('1', 'Baja'), ('2', 'Normal'), ('3', 'Alto'),
    ('4', 'Muy alto'), ('5', 'Excelente')],
    string="Valoraciones")

comments = fields.Char(string="Comentarios", help="Aquí puedes dejar
comentarios que leerán tanto los administradores como los demás
empleados (por ejemplo para mejorar la idea).")

employee_id=fields.Many2one(comodel_name='hr.employee',string='Empleado',
help="El empleado que ha votado la idea.")

idea_id=fields.Many2one('idea.management', string='Nombre de la idea',
readonly=True, help="La idea que ha votado.")

registered_votes = []
```

**\_name:** Así es como se guardará el nombre del modelo en la base de datos, es decir, es el nombre técnico, que en este caso es 'idea.management.vote'.

**\_description:** Proporciona una breve descripción de la función del modelo (en este caso es gestionar los votos).

**rating:** Ahora definimos un campo de selección llamado 'rating', que permite a los usuarios asignar una nota a una idea. El rango que tengo establecido de notas en mi caso, va desde 'Very Low' hasta 'Excellent'. Este campo registra la valoración que los empleados dan a una idea.

**comments:** Este trozo de código define un campo de tipo “texto corto” (Char) llamado 'comments', que permite a los empleados dejar comentarios sobre la idea que están votando. Principalmente este campo se ha hecho porque estos comentarios pueden ser útiles para proporcionar sugerencias sobre cómo mejorar la idea.

**employee\_id:** Posteriormente, ahora definimos un campo de relación Many2One llamado 'employee\_id', que conseguimos establecer una relación con el modelo 'hr.employee'. Este campo se encarga de registrar qué empleado ha votado en la idea concreta.

**idea\_id:** Y ahora, en este trozo de código se define otro campo de relación Many2One llamado 'idea\_id', que se encarga de establecer una relación con el modelo 'idea.management' (el modelo que hemos explicado anteriormente).

La función de este campo es indicar a qué idea se refiere el voto registrado.

Y le añadimos el parámetro de 'readonly=True', que significa que este campo será de solo lectura, es decir, una vez que se ha registrado un voto para una idea, no se puede cambiar la idea asociada a ese voto.

**registered\_votes:** Y ahora por último, lo que hacemos en el código es declarar e inicializar un array vacío que en el futuro va a contener los votos de los empleados asociados a esa idea en concreto.

## Funciones

A continuación vamos a pasar con las **funciones** que tengo dentro de mi modelo:

```
def save_vote(self):
    for vote in self:
        existing_vote = next((v for v in self.registered_votes if v['idea_id'] ==
vote.idea_id.id and v['employee_id'] == vote.employee_id.id), None)
        if not existing_vote:
            self.env['idea.management.vote'].write({
                'employee_id': vote.employee_id.id,
                'rating': vote.rating,
                'comments': vote.comments,
                'idea_id': vote.idea_id.id,
            })
            self.registered_votes.append({
                'employee_id': vote.employee_id.id,
                'idea_id': vote.idea_id.id,
            })
        else:
            raise ValidationError("No puedes votar dos veces la misma idea.")
```

**save\_vote:** Esta función se encarga de guardar los votos de los empleados en la idea en concreto. voy a pasar a explicar un poco más en concreto lo que hace el código:

- Iteramos sobre cada voto recibido (for vote in self).
- Para cada voto, tenemos que verificar si ya existe un voto registrado en la idea y que tenga adjudicado el mismo empleado. Esto se hace buscando en la lista que hemos declarado anteriormente (registered\_votes), y comprobamos si hay un voto con la misma combinación de idea\_id y employee\_id.
- Si no existe ningún voto registrado para la idea y el mismo empleado, crea un nuevo registro en la base de datos utilizando el modelo idea.management.vote y los datos del voto actual (el empleado que vota, la valoración que le ha dado a la idea, los comentarios y la idea).
- Aparte de guardar el voto en la base de datos, debemos de agregar la información de este voto a la lista “registered\_votes”. Esto permite realizar un seguimiento y comprobaciones de los votos que han sido registrados para así poder evitar duplicados.
- Y por último, en el caso de que ya exista un voto registrado para la idea en cuestión, y sea del mismo empleado, se genera una excepción del tipo

"ValidationError" indicando que un empleado no puede votar dos veces en la misma idea.

```
def cancel_vote(self):
    id_vote = self.env.context.get('active_id')
    if id_vote:
        idea_record = self.env['idea.management'].browse(id_vote)
        if idea_record:
            last_vote = idea_record.vote_ids[-1]
            last_vote.unlink()
```

**cancel\_vote:** Esta función, se encarga de cancelar el voto que estamos rellenando actualmente, que está asociado al empleado actual, voy a explicar un poco más en concreto lo que hace este método:

- Obtenemos el ID del **voto** activo actualmente en el contexto
- Si encontramos un ID de voto válido, buscamos en el registro de la idea correspondiente utilizando ese ID.
- Después, cuando encontremos el registro de la idea, accedemos al último voto registrado para esa idea.
- Y ahora, por último, eliminamos ese voto más reciente (que es el voto actual que estamos completando en el formulario), y entonces cancela el voto del empleado no dejando que se guarde.

## Vistas

### Modelo - idea.management

#### Tree

En primer lugar vamos a pasar a explicar la vista **tree**:

```
<record id="view_idea_management_tree" model="ir.ui.view">
  <field name="name">view.idea_management.tree</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <tree string="Idea">
      <field name="employee_id"/>
      <field name="name"/>
      <field name="state"/>
      <field name="idea_type"/>
      <field name="partner_id"/>
    </tree>
  </field>
</record>
```

Lo que hacemos primero es, definir una vista llamada "view\_idea\_management\_tree", y la utilizamos para mostrar una lista de ideas en forma de árbol.

Por otro lado, también cabe explicar, que en la línea 3, le decimos a qué modelo está asociada esta vista (idea.management) lo que significa que mostraremos los datos de este modelo específico.

Después, dentro de la etiqueta <tree>, especificamos la estructura de la vista del árbol. Cada <field> representará un campo del modelo idea.management que se mostrará en la lista, que en este caso los **campos** que tengo incluidos en la vista son:

- employee\_id.
- name
- state
- idea\_type
- partner\_id

Y de esta manera conseguimos que en nuestra vista tree nos aparezcan los campos que nosotros hemos seleccionado del modelo idea.management.



## Form

En segundo lugar vamos a explicar la vista **form**:

Que para esta vista vamos a ir por partes, en primer lugar vamos a explicar la estructura básica junto al `<header>`:

```
<record id="view_idea_management_form" model="ir.ui.view">
  <field name="name">view.idea_management.form</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <form string="Idea">
      <header>
        <button string="Aprobar" name='aprobar' type="object" invisible="state!='revision'" /> <!-- Si ponemos invisible=
        <button string="En proceso" name="proceso" type="object" invisible="state!='aprobada'" />
        <button string="Completar" name="completada" type="object" invisible="state not in 'proceso,aprobada'"/>
        <button string="Votar" name="open_vote_form" type="object" invisible="state not in 'proceso'"/>
        <button string="Cancelar" name="cancelada" type="object" invisible="state not in 'revision,aprobada,proceso'"/>
        <field name="state" widget="statusbar"/>
      </header>
    </form>
  </field>
</record>
```

En este formulario hemos incluido un “header”, donde se tenemos varios botones que cada uno tiene una función, aparte de esto, tenemos como en la vista “tree”, el id de la vista, el name que le vamos a declarar y además el modelo al que hace referencia esta vista y al que vamos a poder referenciar los campos y las funciones.

Los botones del header representan acciones que podemos realizar en una idea, como si está en revisión, podemos aprobarla, después, marcarla como en proceso, y por último completarla, votar por ella o incluso cancelarla.

Cada botón está vinculado a una función declarada en Python, como ya hemos visto antes, el nombre de la función corresponde al “name” de cada botón. Por ejemplo, al hacer clic en el botón "Aprobar", se llamará al método aprobar en el modelo.

Y también, utilizamos el atributo invisible en los botones para controlar cuándo va a estar visible. Por ejemplo, el botón "Aprobar" solo será visible si el estado de la idea es "revision". Esto significa que solo se podrá aprobar una idea si está en estado de revisión.

Y ahora sí, para finalizar con esta parte, el campo ‘state’ se representa utilizando el widget de barra de estado (widget="statusbar"), esto va a indicar visualmente el estado actual de la idea en una forma de barra con sus diferentes estados.

```
<sheet>
  <div class="oe_title">
    <h1>
      <field name="name"/>
    </h1>
  </div>
</group>
<group>
  <group>
    <field name="employee_id"/>
    <field name="create_date"/>
    <field name="deadline"/>
  </group>
</group>
```

Y ahora en este otro fragmento de código, definimos una hoja en la interfaz del usuario (mediante el sheet). Y esto lo hacemos para así poder presentar los datos de una manera más forma organizada y estructurada.

Y ahora ya dentro de la hoja, añadimos un contenedor (el div) con la etiqueta “oe\_title” para decirle que va a ser el título principal de la hoja (va a mostrar el nombre de la idea). También le añadimos la etiqueta <h1> para así definir que el título va a estar en negrita y tamaño grande.

Continuamos hablando de los “group”, que varios grupos que se utilizan para organizar visualmente los campos, como podemos ver primero me declaro un grupo grande y después dentro de ese grupo, declaro otro para mostrar los campos:

- employee\_id
- create\_date
- deadline

Que se van a mostrar en una única columna.

```
<group>
  <field name="partner_id"/>
  <field name="idea_type"/>
  <field name="ideas_empleado"/>
  <field name="price"/>
  <field name="active" invisible="True"/>
</group>
</group>
```

Y por aquí seguido al código anterior, tenemos aquí otro grupo (que en la vista final se mostrará como otra columna), que añadimos los campos:

- partner\_id
- idea\_type
- ideas\_empleado
- price
- active

Y después también cerramos el otro grupo para indicar que ya no vamos a querer añadir más campos en estas columnas.

```
<notebook>
  <page string="Detalles">
    <field name="details"/>
  </page>

  <page string="Votos de los Empleados">
    <field name="vote_ids" readonly="1">
      <tree string="Votes" readonly="1">
        <field name="employee_id"/>
        <field name="rating"/>
        <field name="comments"/>
      </tree>
    </field>
  </page>
</notebook>
</sheet>
```

Pasamos con el siguiente trozo de código dentro del form, y vemos que añadimos la etiqueta “notebook” que esa etiqueta se encarga de ser como un contenedor de pestañas donde se pueden organizar varias páginas.

En mi caso tengo la primera página del notebook, llamada "Detalles". En esta página, se muestra el campo llamado "details", que su función va a ser que el empleado pueda añadir la descripción de la idea.

Y seguido a esto, tenemos ya la segunda página, que muestra una lista de los votos de los empleados asociados a la idea. Mostramos los votos relacionados a la idea pero con solo permisos de lectura “readonly=“1”, para que los usuarios no puedan editar su voto después de ya realizar el voto o simplemente porque queremos que lo hagan desde el formulario correspondiente siguiendo una jerarquía.

Y los campos que vamos a mostrar en el voto del empleado van a ser:

- **employee\_id** (empleado que vota).
- **rating** (valoración que recibe la idea).
- **comments** (comentarios asociados a la idea donde expresas tu opinión o proporcionas posibles mejoras o distintas formas de hacerlo).

Y ya por último cerramos las etiquetas del “notebook” y de la hoja “sheet”.

```
<div class="oe_chatter">
  <field name="message_follower_ids"/>
  <field name="activity_ids"/>
  <field name="message_ids" options="{ 'post_refresh':
'recipients' }"/>
</div>
</form>
</field>
</record>
```

Y por último, una de las cosas más interesantes de esta vista form, ya que definimos una sección en la interfaz del usuario llamada "oe\_chatter", que se utiliza para mostrar elementos relacionados con la comunicación y la actividad. refiriéndose a un registro (en este caso tendremos uno por cada idea).

Pero ahora vamos a explicarlo un poco más en detalle:

- En el campo de **message\_follower\_ids**: Muestra los seguidores del registro, las personas que quieren recibir notificaciones sobre los cambios relacionados con el registro (con la idea).
- Ahora pasamos con el campo **activity\_ids**: Aquí mostramos las actividades relacionadas con el registro. Las actividades pueden ser tareas, reuniones... Pero que deben estar relacionadas con el registro (idea).
- Y ahora por último, antes de cerrar el contenedor, el formulario... Pasamos con el campo **message\_ids**: Este campo muestra los mensajes relacionados con el registro. Puede incluir comentarios, correos electrónicos... El parámetro “post\_refresh” indica que los mensajes se actualizarán automáticamente después de que se publique uno nuevo. Además, gracias también al parámetro “recipients” especificamos que se muestren los destinatarios de los mensajes.

## Search

Ahora vamos a continuar hablando sobre la vista **search**:

```
<record id="view_idea_management_search" model="ir.ui.view">
  <field name="name">view.idea_management.search</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <search string="Idea">
      <!-- Filtrar por nombre y por empleado asignado -->
      <field name="name"/>
      <field name="employee_id"/>
      <!-- Filtrar por tipos de idea -->
      <filter string="Mejoras" name="idea_type" domain="[('idea_type', '=', 'mejoras')]" />
      <filter string="Plantear proyecto" name="idea_type" domain="[('idea_type', '=', 'proyecto')]" />
      <filter string="Otros" name="idea_type" domain="[('idea_type', '=', 'otros')]" />
      <separator />
      <filter string="Archivado" name="inactive" domain="[('active', '=', False)]" />
      <!-- Agrupar -->
      <filter string="Empleado" name="assigned_to" context="{ 'group_by': 'employee_id' }" />
      <filter string="Estado" name="state" context="{ 'group_by': 'state' }" />
      <separator string="Separador" />
      <filter string="Tipo de idea" name="idea_type" context="{ 'group_by': 'idea_type' }" />
      <filter string="Compañía" name="partner_id" context="{ 'group_by': 'partner_id' }" />
    </search>
  </field>
</record>
```

Dentro de esta vista de búsqueda, especificamos diferentes campos y filtros que los usuarios van a poder utilizar para buscar ideas en odoo, en mi vista tengo definido lo siguiente:

- **Búsqueda directa:** Los campos 'name' y 'employee\_id' los mostramos como campos de búsqueda directa, es decir, los usuarios pueden ingresar valores en el campo de búsqueda que aparece una vez dentro del módulo, y de esa manera puedes filtrar directamente por el nombre de la idea o por el empleado que ha tenido la idea.
- **Filtros:** Se definen varios filtros, los primeros están basados en el campo "idea\_type". Estos filtros nos permiten filtrar las ideas según si son "Mejoras", "Plantear proyecto" u "Otros". Y también, se agrega un filtro para mostrar ideas archivadas, utilizando el campo 'active'. Este filtro (Archivado), permite a los usuarios ver las ideas que están marcadas como (active = False) en odoo.
- **Agrupaciones:** También definimos filtros de agrupación para los campos 'employee\_id', 'state', 'idea\_type' y 'partner\_id'. Gracias a estos filtros permitimos agrupar las ideas por diferentes criterios, como el empleado asignado, el estado de la idea, el tipo de idea y la compañía que está asociada.

## Calendar

Seguimos ahora con la vista **calendar**:

```
<record id="view_idea_management_calendar" model="ir.ui.view">
  <field name="name">view.idea_management.calendar</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <calendar string="Ideas" date_start="create_date"
color="employee_id" hide_time="true">

      <field name="employee_id" avatar_field="image_128"/>
      <field name="name"/>
      <field name="create_date"/>
      <field name="deadline"/>
      <field name="days_duration"/>
    </calendar>
  </field>
</record>
```

Como ya bien dice el nombre, este código sirve para definir una vista de calendario en la interfaz del usuario. Como en las vistas anteriores tenemos que asignarle un identificador que debe ser único y el model, después asignar el nombre que queremos que tenga la vista junto al modelo al que va a hacer referencia (en este caso 'idea.management').

Ahora pasamos con los atributos del <calendar>:

- **string:** Define el título de la vista de calendario (que en este caso es "Ideas").
- **date\_start:** Especifica que la fecha de inicio de cada evento en el calendario será la fecha de creación de la idea (el campo 'create\_date').
- **color:** Este atributo indica que vamos a utilizar el campo 'employee\_id' para asignar diferentes colores a los eventos del calendario. Esto es bastante útil para distinguir las ideas de cada empleado.
- **hide\_time:** Indica que se ocultarán las horas en los eventos del calendario, mostrando solo las fechas.

Y ahora también tenemos los campos que vamos a mostrar en cada evento calendario:

- **employee\_id:** Mostramos el campo 'employee\_id' en cada evento del calendario. Pero no solo eso, además, se utiliza el campo 'image\_128' del modelo asociado para mostrar un avatar del empleado junto a su nombre.
- **name:** En este campo vamos a mostrar el nombre de la idea.
- **create\_date:** Aquí tenemos el campo que mostrará la fecha en la que fue creada la idea.
- **deadline:** En este campo lo que vamos a mostrar va a ser la fecha límite de la idea.
- **days\_duration:** Y por último mostramos este campo computado que mostrará los días que durará la idea.

## Pivot

Ahora vamos a pasar con la siguiente vista, la **pivot**:

```
<record id="view_idea_management_pivot" model="ir.ui.view">
  <field name="name">view.idea_management.pivot</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <pivot string="Idea">
      <field name="employee_id" type="row"/>
      <field name="create_date" interval="month" type="col"/>
      <field name="price" widget="monetary"/>
    </pivot>
  </field>
</record>
```

Lo mismo que las demás vistas (me refiero a que les asignamos un id, model, name y modelo al que hace referencia). Y ahora sí, pasamos a definir el arch que contiene el <pivot>, que tenemos 3 campos que son los que se mostrarán en la vista pivot:

- **name:** Este campo se utiliza para organizar los datos por el id del empleado (employee\_id). Cada fila de la vista pivot se representa por un empleado y sus datos de idea.
- **create\_date:** Este campo se utiliza para organizar los datos por fecha de creación (create\_date). Se agruparán por meses (interval="month") y se mostrará en las columnas de la vista pivot.

- **price:** Y ahora por último, este campo representa el precio estimado de idea. El atributo `widget="monetary"` indica que se mostrará como un valor monetario.

## Graph

Continuamos con las vistas y ahora es el turno de la vista **graph**:

```
<record id="view_idea_management_graph" model="ir.ui.view">
  <field name="name">view.idea_management.graph</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <graph string="Idea">
      <field name="employee_id"/>
      <field name="state"/>
      <field name="price"/>
      <field name="revision_count" string="En Revisión"/>
      <field name="aprobada_count" string="Aprobadas"/>
      <field name="proceso_count" string="En Proceso"/>
      <field name="completada_count" string="Completadas"/>
      <field name="cancelada_count" string="Canceladas"/>
    </graph>
  </field>
</record>
```

Esta vista está diseñada para mostrar un gráfico relacionado con las ideas, en el cual los campos que vamos a mostrar por pantalla en la interfaz del usuario, los vamos a extraer del modelo 'idea.management'.

El gráfico se define utilizando el elemento `<graph>`, y se asigna el título "Idea" mediante el atributo `string`.

Ahora vamos a pasar a contar un poco más en detalle los campos que mostramos en el gráfico:

- **employee\_id:** Este campo representa al empleado asociado con la idea.
- **state:** Este campo representa el estado actual de la idea.
- **price:** Este campo representa el coste aproximado de la idea.

Y ahora tenemos unos campos que he creado para llevar la cuenta de todas las ideas que pertenecen a un 'state' (a un estado) de cada empleado y se calculan mediante una función `compute`, primero vamos a explicar los campos y después la función:



```
revision_count = fields.Integer(string="En Revisión", compute='_compute_state_counts',
aprobada_count = fields.Integer(string="Aprobadas", compute='_compute_state_counts', store=True)
proceso_count = fields.Integer(string="En Proceso", compute='_compute_state_counts', store=True)
completada_count = fields.Integer(string="Completadas", compute='_compute_state_counts', store=True)
cancelada_count = fields.Integer(string="Canceladas", compute='_compute_state_counts', store=True)
```

- **revision\_count**: Este campo representa la cantidad de ideas en estado de revisión.
- **aprobada\_count**: Representa la cantidad de ideas que han sido aprobadas.
- **proceso\_count**: Indica la cantidad de ideas que están en proceso.
- **completada\_count**: Muestra cuántas ideas han sido completadas.
- **cancelada\_count**: Indica la cantidad de ideas que han sido canceladas.

Y esta es la función compute que se encarga de calcular cuántas ideas hay en cada estado (state) asociado a cada empleado:

```
@api.depends('state')
def _compute_state_counts(self):
    self.revision_count = len(self.filtered(lambda r: r.state == 'revision'))
    self.aprobada_count = len(self.filtered(lambda r: r.state == 'aprobada'))
    self.proceso_count = len(self.filtered(lambda r: r.state == 'proceso'))
    self.completada_count = len(self.filtered(lambda r: r.state == 'completada'))
    self.cancelada_count = len(self.filtered(lambda r: r.state == 'cancelada'))
```

Esta función se ejecutará cada vez que cambie el valor del campo 'state'. Es decir, lo que hace esta función es contar el número de registros en nuestro modelo, y los va sumando por el estado en el que se encuentren y una vez se sumen, se almacenan en su correspondiente variable (por ejemplo, 'self.revision\_count' solo va a almacenar el total de las ideas que tengan de estado = 'revision', el 'self.aprobada\_count' solo almacenará las ideas que tengan de estado 'aprobada' y así con todos los estados por los que puede pasar una idea, y se asociará a los empleados).

## Kanban

Y ahora la última vista respecto al modelo 'idea.management' es la **kanban**:

Esta vista, igual que la "form" la vamos a explicar por partes porque es bastante extensa:

```
<record id="view_idea_management_kanban" model="ir.ui.view">
  <field name="name">view.idea_management.kanban</field>
  <field name="model">idea.management</field>
  <field name="arch" type="xml">
    <kanban default_group_by="employee_id">
      <progressbar field="state" colors="{&quot;revision&quot;;: &quot;secondary&quot;;, &quot;aprobada&quot;;: &quot;warning&quot;;,
        &quot;proceso&quot;;: &quot;info&quot;;, &quot;completada&quot;;: &quot;success&quot;;,
        &quot;cancelada&quot;;: &quot;danger&quot;}" />
      <field name="state" invisible="1" />
      <field name="idea_type" invisible="1" />
    </kanban>
  </field>
</record>
```

Como en las demás vistas, tenemos que asignar un id único para poder hacer referencia a esta vista, después ponerle un name e indicarle el modelo al que hace referencia, a continuación dentro del <field> "arch" vamos a empezar a definir la vista kanban:

- **default\_group\_by="employee\_id"**: Esto quiere decir que la agrupación se realizará por defecto por el campo 'employee\_id', en otras palabras, quiere decir que las ideas se agruparán y se mostrarán según el empleado asociado.
- **progressbar**: Agregamos este elemento para agregar unas barras de progreso al Kanban. Las barras de progreso están basadas en el 'state' (estado) de las ideas y van a coger un color en relación con el estado actual de cada idea. Por ejemplo, las ideas en estado "revision" se mostrarán con un color gris clarito, y las que están en estado "aprobada" se verán con un amarillo.
- **Campos**: Los campos 'state' e 'idea\_type' están invisibles, lo que significa que no se muestran en el Kanban, pero aún así, estarán disponibles en odoo para su uso (por ejemplo, que otros campos se pueden basar en esos campos pese a estar invisibles).

Y ahora vamos a explicar el código que viene a continuación de la etiqueta "templates".

```
<templates>
  <t t-name="kanban-menu">
    <t t-if="widget.editable"><a role="menuitem" class="dropdown-item" data-type="edit">Edit</a></t>
    <t t-if="widget.deletable"><a role="menuitem" class="dropdown-item" data-type="delete">Delete</a></t>
  </t>
  <t t-name="kanban-box">
    <div class="oe_kanban_content flex-grow-1">
      <div class="oe_kanban_details">
        <strong class="o_kanban_record_title"><field name="name"/></strong>
      </div>
    </div>
  </t>
</templates>
```

La primera plantilla que utilizamos dentro del `<templates>` se llama 'kanban-menu' y contiene elementos `<a>` que representan opciones de menú para editar y eliminar elementos en la vista kanban.

- Estos elementos tienen asociados 2 widgets (`widget.editable` y `widget.deletable`). Si ambos widgets son `True`, se muestra un 'enlace' (que en realidad son 2 opciones dentro de un menú) que son, una para editar y otra para eliminar y si solamente es `true` el '`widget.editable`' se podrá editar pero no eliminar, y si es `true` el '`widget.deletable`', se podrá eliminar, pero no editar.

La segunda plantilla se llama 'kanban-box' y la utilizo porque define la apariencia de cada tarjeta en la vista kanban. Esta plantilla contiene una estructura HTML que define cómo se muestra la información de cada tarjeta. Tenemos la etiqueta del `name`, que se utiliza para mostrar el nombre del elemento dentro de un campo de texto.

```
<div class="o_kanban_record_subtitle">
  <field name="create_date"/>
</div>
<div class="oe_kanban_bottom_left">
  <field name="state"/>
</div>
<div class="o_kanban_quick_actions">
  <button name="change_state" class="btn btn-sm btn-primary" type="object" title="Cambiar estado">Cambiar Estado</button>
</div>
```

Empezamos con las 3 primeras líneas, que sirven para que en la parte superior de la tarjeta gracias al `o_kanban_record_subtitle`, se muestra la fecha de creación de la idea. También es imprescindible utilizar el campo '`create_date`' del modelo que tenemos vinculado a la vista (`idea.management`).

Seguimos con las 3 líneas siguientes, el segundo 'div' (contenedor), este div hará que el campo '`state`' de la idea aparezca en la esquina inferior izquierda de la tarjeta, gracias también a la clase `oe_kanban_bottom_left`.

El tercer div, son las "acciones rápidas" en este caso, solo tengo una acción que un botón llamado "Cambiar Estado". La acción que está vinculada a este botón es el método '`change_state`' definido en el modelo que tenemos asociado (`idea.management`), cuando se hace clic en este botón, se ejecutará la función

(cambiará el 'state' de la idea). Este botón tiene la clase 'btn btn-sm btn-primary', esto significa que será un botón pequeño y con un estilo de fondo.

```
<div class="oe_kanban_footer">
  <div class="o_kanban_record_bottom">
    <div class="oe_kanban_bottom_left">
      <field name="state"/>
      <field name="activity_ids" widget="kanban_activity"/>
    </div>
    <div class="oe_kanban_bottom_right">
      <field name="employee_id" widget="many2one_avatar_user"/>
    </div>
  </div>
</div>
</div>
</t>
</templates>
</kanban>
```

Ahora pasamos con el final de la vista **kanban**, gracias a la clase 'oe\_kanban\_footer' podemos definir el pie de página del panel Kanban. Con el footer, nos referimos al área que se encuentra debajo de cada tarjeta del panel Kanban.

Y ahora tenemos que definir dentro del segundo div, esta clase: "o\_kanban\_record\_bottom" que sirve para definir la parte inferior de la tarjeta en el panel Kanban. Después tenemos que la parte inferior de la tarjeta se divide en 2 secciones, left and right, primero vamos a hablar un poco sobre la sección right:

- **oe\_kanban\_bottom\_left:** Aquí tenemos el campo 'price', este campo representa el coste estimado de la idea, y además de ello, se utiliza el widget "monetary" para formatear el valor del campo como una cantidad monetaria. También tenemos el campo 'activity\_ids', que este campo se utiliza para representar las actividades relacionadas con la idea. Y utilizamos el widget 'kanban\_activity' para mostrar estas actividades dentro del panel Kanban.
- **oe\_kanban\_bottom\_right:** Y en la derecha, se muestra el campo 'employee\_id', que representa al empleado asociado con la idea. Y por otro lado, se utiliza el widget "many2one\_avatar\_user", para que se muestre la imagen del empleado junto con su nombre.

Y por último cerramos todas las etiquetas correspondientes.

# Vistas

## idea\_management\_vote\_view.xml

### Form

Ahora voy a explicar la única vista del modelo “idea\_management\_vote” que se utiliza cuando estamos en el menú de la idea y clicamos en el botón “votar”, se abrirá el formulario (**form**) que vamos a definir:

```
<record id="view_idea_vote_form" model="ir.ui.view">
  <field name="name">view.idea_vote_form</field>
  <field name="model">idea.management.vote</field>
  <field name="arch" type="xml">
    <form string="Voto">
      <header>
        <field name="idea_id" readonly="1"/>
      </header>
      <h1>Rating</h1>
      <field name="rating" widget="priority"/>
      <sheet>
        <group>
          <field name="comments"/>
        </group>
        <group>
          <field name="employee_id" readonly="True"/>
        </group>
      </sheet>
      <footer>
        <button string="Guardar" name="save_vote" type="object" class="btn-primary"/>
        <button string="Cancelar" name="cancel_vote" type="object" class="btn-secondary"/>
      </footer>
    </form>
  </field>
</record>
```

Como acabo de comentar, esta vista es de tipo “form”, como ya veníamos haciendo en las demás vistas, debemos de asignarle a esta vista un identificador único, después un nombre y por último el modelo al que hace referencia.

Después de esto, lo que debemos hacer es declarar dentro del “arch” la estructura del <form>, iniciamos el formulario con el título “Voto” y a continuación, declaramos el <header> y dentro vamos a mostrar el campo ‘idea\_id’ en modo de solo lectura. Solamente los usuarios podemos ver a qué idea nos referimos al votar, pero no podremos modificar este campo.

Ya fuera del header, definimos un <h1> (para agregar el título de “Rating” antes de agregar el campo correspondiente para poder votar), y ahora sí, la siguiente línea es el campo para poder votar (campo “rating”) y le añadimos el widget de “**priority**”

para que así las posibles notas aparezcan en forma de estrellas y quede de manera visualmente.

Ahora ya vamos a abrir el cuerpo del formulario con la etiqueta `<sheet>` y agregamos grupos para organizar mejor nuestros campos en la interfaz del usuario.

Dentro de un grupo vamos a añadir el campo “comments” que simplemente hace la función de añadir un campo de texto donde los usuarios van a poder añadir sus comentarios (por ejemplo para mejorar la idea del empleado o cosas que no se haya podido tener en cuenta), después dentro del otro grupo añadimos el campo “employee\_id”, pero solo en modo lectura, y así indicamos qué empleado ha hecho el voto.

Y por último, en el footer (pie de página del formulario), declaramos 2 botones:

- **Guardar:** Agregamos un botón con el string "Guardar" que, cuando hacemos click, invocamos el método ‘save\_vote’ del modelo, para guardar los datos del voto.
- **Cancelar:** Agregamos un botón con el string "Cancelar", que llama al método ‘cancel\_vote’ para cancelar la acción.

Y ahora cerramos todas las etiquetas correspondientes para tener la vista funcional.

## Report y vista - Ideas

Lo que vamos a realizar en este report y vista, es crear un informe para la idea que nosotros queramos, vamos a poner el código y después explicamos lo que hace:

```
<odoo>
<template id="report_idea_management_document">
  <t t-call="web.external_layout">
    <div class="page">
      <div class="oe_structure"/>
      <div class="mt-4 mb-4 text-center">
        <h2 class="mt16">
          <span t-field="doc.name"/>
        </h2>
      </div>
      <div class="row mt32 mb32" id="informations">
        <div class="col-auto col-3 mw-100 mb-2">
          <strong>Fecha:</strong>
          <p class="m-0" t-field="doc.create_date"/>
        </div>
        <div t-if="doc.partner_id" class="col-auto col-3 mw-100 mb-2">
          <strong>Compañía:</strong>
          <p class="m-0" t-field="doc.partner_id"/>
        </div>
      </div>
      <table class="table table-sm o_main_table">
        <thead style="display: table-row-group">
          <tr>
            <th name="th_employee_id" class="text-left bg-primary text-light">Usuario</th>
            <th name="th_rating" class="text-right bg-primary text-light">Valoración</th>
            <th name="th_comments" class="text-right bg-primary text-light">Comentario</th>
          </tr>
        </thead>
        <tbody class="sale_tbody">
          <t t-foreach="doc.vote_ids" t-as="vote">
            <tr class="bg-200 font-weight-bold">
              <td name="td_employee_id">
                <span t-field="vote.employee_id"/>
              </td>
              <td name="td_rating" class="text-right">
                <span t-field="vote.rating"/>
              </td>
              <td name="td_comments" class="text-right">
                <span t-field="vote.comments"/>
              </td>
            </tr>
          </t>
        </tbody>
      </table>
    </div>
  </t>
</template>
<template id="report_idea_management">
  <t t-call="web.html_container">
    <t t-foreach="docs" t-as="doc">
      <t t-call="ideas_module.report_idea_management_document"/>
    </t>
  </t>
</template>
</odoo>
```

En primer lugar, utilizamos la plantilla base 'web.external\_layout' para mantener coherencia con los informes de Odoo. Por otro lado, incluimos la estructura principal y se define un bloque de contenido dentro de una página (<div class="page">).

Ahora mostramos el nombre de la idea, pero lo hacemos mediante un encabezado <h2>, después mostramos la fecha de creación de la idea, y si está disponible (si el empleado pertenece a una compañía), también mostramos la compañía,

Para continuar, definimos una tabla con columnas para mostrar los votos de los empleados, y hablando de las filas, cada fila de la tabla muestra el usuario (empleado que ha votado), la valoración dada y el comentario asociado al voto.

- **report\_idea\_management\_document:** Definimos la estructura del informe para que solo haya un único documento de idea y tenga la estructura que le hemos definido anteriormente.
- **report\_idea\_management:** Es el id del template, que va a tratar de recorrer una lista de documentos y aplicar la plantilla.

Entonces, sabiendo como funciona el código, el funcionamiento general sería el siguiente:

En primer lugar, generamos el informe para las ideas, y se toma cada documento de idea (doc) y se presenta utilizando la estructura definida en 'report\_idea\_management\_document'.

En segundo lugar, la información de la idea (el nombre, la fecha de creación y la compañía asociada) se presenta de la manera definida.

Y por último, se incluye la tabla que lista los votos asociados a la idea, con el empleado que votó, la valoración y los comentarios.


También cabe recalcar que tengo una vista que se llama 'idea\_management\_report\_view.xml' que tengo declarado un action, para que en la interfaz de usuario aparezca el botón de poder imprimir/descargar el informe de la idea, se generará un informe PDF para una o más ideas en el modelo idea.management utilizando la plantilla 'ideas\_module.report\_idea\_management'.

```
<record id="action_report_idea_management" model="ir.actions.report">
  <field name="name">Idea Report</field>
  <field name="model">idea.management</field>
  <field name="report_type">qweb-pdf</field>
  <field name="report_name">ideas_module.report_idea_management</field>
```



```
<field name="report_file">ideas_module.report_idea_management</field>
<field name="print_report_name">'Idea - %s' % (object.name)</field>
<field name="binding_model_id" ref="model_idea_management"/>
<field name="binding_type">report</field>
</record>
```

Y ahora voy a dejar un pequeño ejemplo de cómo sería un report:



**YourCompany**  
 250 Executive Park Blvd, Suite 3400  
 94134 San Francisco  
 California  
 España

## Votación para implementar nuevo servidor

**Fecha:**  
 20/05/2024

**Compañía:**  
 YourCompany

Usuario	Valoración	Comentario
Angel	Muy alto	Me parece una muy buena idea debido al gran crecimiento de la empresa y lo veo necesario.
Pablo	Excelente	Estupenda idea!! Y además los costes no son demasiado excesivos.
Alex	Normal	No me parece mala la idea, de hecho lo necesitamos, pero no creo que este sea el momento adecuado, tenemos muchas tareas pendientes y creo que habría que darles prioridad.

Como podemos ver en el ejemplo, también le hemos aplicado a la tabla unos estilos para que se vea más agradable visualmente.

## Security

Con seguridad, me quiero referir a las líneas de código que definen las reglas de acceso para nuestro módulo en Odoo.

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_idea_user,access_idea_user,model_idea_management,idea_management_user,1,0,1,0
access_idea_manager,access_idea_manager,model_idea_management,idea_management_manager,1,1,1,1
access_idea_vote_user,access_idea_user,model_idea_management_vote,idea_management_user,1,1,1,1
access_idea_vote_manager,access_idea_manager,model_idea_management_vote,idea_management_manager,1,1,1,1
```

Cada línea en este archivo CSV define una regla de acceso para su respectivo modelo y un grupo de usuarios específico. La estructura es la siguiente:

**id:** Es el Identificador único de la regla de acceso.

**name:** Nombre de la regla de acceso.

**model\_id:id:** Identificador del modelo al que se aplica la regla.

**group\_id:id:** Identificador del grupo de usuarios al que se aplica la regla.

**perm\_read:** Permiso de lectura (el 1 es para permitir y el 0 para denegar).

**perm\_write:** Permiso de escritura (el 1 es para permitir y el 0 para denegar).

**perm\_create:** Permiso de creación (el 1 es para permitir y el 0 para denegar).

**perm\_unlink:** Permiso de eliminación (el 1 es para permitir y el 0 para denegar).

- **access\_idea\_user:** Nombre del acceso: 'access\_idea\_user', modelo: 'model\_idea\_management', grupo: 'idea\_management\_user' y permisos: Lectura (1), Escritura (0), Creación (1), Eliminación (0).
- **access\_idea\_manager:** Nombre del acceso: 'access\_idea\_manager', modelo: 'model\_idea\_management', grupo: 'idea\_management\_manager' y permisos: Lectura (1), Escritura (1), Creación (1), Eliminación (1).
- **access\_idea\_vote\_user:** Nombre del acceso: 'access\_idea\_user', modelo: 'model\_idea\_management\_vote', grupo: 'idea\_management\_user' y permisos: Lectura (1), Escritura (1), Creación (1), Eliminación (1)
- **access\_idea\_vote\_manager:** Nombre del acceso: 'access\_idea\_manager', modelo: 'model\_idea\_management\_vote', grupo: 'idea\_management\_manager' y permisos: Lectura (1), Escritura (1), Creación (1), Eliminación (1)

Y ahora vamos a explicar el fichero xml de seguridad, que sirve para definir categorías de módulos, los grupos de usuarios y los permisos asociados a estos grupos.

```
<odoo>

<record id="ir_module_category_idea_management" model="ir.module.category">
  <field name="name">Idea</field>
</record>

<record id="idea_management_user" model="res.groups">
  <field name="name">User</field>
  <field name="category_id" ref="ir module category idea management"/>
  <field name="comment">
    Usuarios que podrán votar una vez la idea haya sido aprobada.
  </field>
</record>

<record id="idea_management_manager" model="res.groups">
  <field name="name">Manager</field>
  <field name="category_id" ref="ir module category idea management"/>
  <field name="implied_ids" eval="[(4, ref('idea_management_user'))]" />
  <field name="users" eval="[(4, ref('base.user_root'))]" />
  <field name="comment">
    Administradores que podrán aprobar ideas de todos y además también pueden votar.
  </field>
</record>

<record id="idea_management_vote_user" model="res.groups">
  <field name="name">Acceso para votar de los usuarios</field>
  <field name="category_id" ref="ir module category idea management"/>
  <field name="comment">Usuarios que podrán votar en las ideas.</field>
</record>

<record id="idea_management_vote_manager" model="res.groups">
  <field name="name">Acceso para votar de los managers</field>
  <field name="category_id" ref="ir module category idea management"/>
  <field name="implied_ids" eval="[(4, ref('idea_management_vote_user'))]" />
  <field name="users" eval="[(4, ref('base.user_root'))]" />
  <field name="comment">Administradores que pueden votar y gestionar votos en las ideas.</field>
</record>
```

**ir\_module\_category\_idea\_management:** Definimos una nueva categoría de módulos llamada "Idea".

Vamos a pasar ahora con los grupos de los usuarios:

- **idea\_management\_user:** nombre del grupo: 'User', categoría: 'ir\_module\_category\_idea\_management', comentario: Explica que los usuarios en este grupo pueden votar una vez que la idea ha sido aprobada.

- **idea\_management\_manager:** nombre del grupo: 'Manager', categoría: 'ir\_module\_category\_idea\_management', implicamos y utilizamos al grupo 'idea\_management\_user', lo que significa que los managers también tienen los permisos de los usuarios, también se incluye el usuario administrador (base.user\_root), y el comentario: Indica que los administradores pueden aprobar ideas y votar.
- **idea\_management\_vote\_user:** nombre del grupo: 'Acceso para votar de los usuarios', categoría: 'ir\_module\_category\_idea\_management' y el comentario: indica que los usuarios en este grupo pueden votar en las ideas.
- **idea\_management\_vote\_manager:** nombre del grupo: 'Acceso para votar de los managers', categoría: 'ir\_module\_category\_idea\_management'. También implicamos al grupo 'idea\_management\_vote\_user', lo que significa que los managers también tienen los permisos de votación de los usuarios. Incluimos el usuario administrador (base.user\_root) y el comentario: Indica que los administradores pueden votar y gestionar votos en las ideas.

Para resumir, de forma rápida y breve:

El archivo **CSV**: Define los permisos de acceso para los modelos idea.management y idea.management.vote para diferentes grupos de usuarios.

Y el archivo **xml**: Define la estructura de los grupos de usuarios y sus relaciones. También especifica qué grupos pueden realizar algunas acciones, como por ejemplo votar, aprobar ideas...

En conclusión, estos archivos configuran la seguridad que tenemos en nuestro módulo, asegurando que solo los usuarios con los permisos adecuados puedan realizar acciones específicas.

## Recursos

- **Visual Studio Code:** Un entorno de desarrollo integrado (IDE) potente y versátil que ofrece soporte para una amplia gama de lenguajes de programación, extensiones y herramientas de desarrollo.
- **Docker:** Una plataforma que permite empaquetar aplicaciones y sus dependencias en contenedores aislados, facilitando la distribución y ejecución de la aplicación en diferentes entornos. En mi caso lo utilizo para lanzar el postgres y el Odoo.
- **Python:** Es un lenguaje de programación de alto nivel, interpretado y de uso general, conocido por su sintaxis clara y legible, que facilita el desarrollo rápido y eficiente de aplicaciones. Python lo he utilizado para el desarrollo del módulo en Odoo.
- **XML:** Es un lenguaje de marcado utilizado para definir la estructura y el contenido de los datos en una forma legible. En el contexto de Odoo, XML se utiliza para definir las vistas de usuario.
- **GitHub:** Una plataforma de desarrollo colaborativo, que utiliza Git para gestionar los repositorios de código, facilitar la colaboración y la integración de los miembros del equipo, también para alojar el código fuente del proyecto de manera pública o privada.

## Necesidades y sugerimientos de la formación

En este curso que he completado, he encontrado que las necesidades educativas están bien alineadas con lo que se pide en el mercado laboral, proporcionando una base sólida para que posteriormente nos sea útil para el desempeño profesional. Sin embargo, yo creo que hay áreas específicas que podrían haberse aprovechado más, ya que yo considero que son muy útiles y no le hemos dado casi atención.

Para mi, en concreto, la formación en el uso y desarrollo de módulos en Odoo, es especialmente relevante y útil en el mundo laboral, ya que odoo es ampliamente utilizado. El módulo en este caso de 'Sistemas de Gestión Empresarial' ofrece una introducción a Odoo, permitiéndonos comprender las funcionalidades básicas y su aplicación práctica en entornos empresariales.

Pero en mi opinión, para lo útil que es y su importancia que le he visto personalmente en el mundo laboral, yo le hubiese dedicado más tiempo a aprender sobre Odoo y también a familiarizarnos y desarrollar un poco más módulos en Odoo, para así poder acostumbrarnos y saber mas o menos como funciona, ya que así después para el proyecto final del curso donde se reúnen todos los módulos también hubiese sido útil para poder tener más idea a la hora de trabajar con Odoo.

Para resumir este punto diría que la conclusión es que hay una muy buena y sólida base de los conceptos aprendidos, pero la única 'queja' que le pondría (y no es ninguna queja porque sí que se llega a ver aunque no de manera muy extensa) es el tema de crear módulos en Odoo y de entender de una forma más clara, cómo funciona este ERP (aunque también es entendible que no se pueda ver mucho más de este tema debido a la gran escasez de tiempo que hay en el segundo curso de este ciclo formativo).

## Opinión personal

Este proyecto me ha despertado mucha curiosidad y ganas de investigar más sobre Odoo, que cuando conocí este sistema de gestión empresarial no las tenía.

Desarrollar este módulo que se podría denominar como "Lluvia de ideas", ha sido una experiencia enriquecedora a nivel personal y también profesional, ya que me he formado con muchos conceptos de Odoo y de Python.

Cuando se me ocurrió esta idea, mi objetivo principal era facilitar el intercambio de ideas dentro de una empresa (lo vi un buen objetivo y bastante práctico).

Desde el punto de vista técnico, como acabo de comentar, la implementación de este módulo me permitió mejorar mis habilidades en el desarrollo de módulos para Odoo. A lo largo del proyecto, tuve que enfrentarme a varias complicaciones que me aparecían por el camino, como podrían ser:

- La configuración de permisos.
- La gestión de los votos de los empleados
- Generar un reporte de la idea

Cada uno de estos retos me llevó a investigar, aprender y aplicar nuevas técnicas de desarrollo.

Gracias a este proyecto, me ha llevado a pensar en las necesidades de los usuarios y cómo podría mejorar su experiencia, haciendo lo más útil, sencillo y eficaz para ellos. Este enfoque centrado en el usuario me ha ayudado a desarrollar una perspectiva más empática a la hora de desarrollar mi código.

Sinceramente, la posibilidad de dar voz a todos los miembros de una empresa y permitirles contribuir con sus ideas es una muy buena forma de fomentar la participación, el compromiso y la relación entre los trabajadores.

## Conclusión

En conclusión, desarrollar el módulo de "lluvia de ideas" para Odoo ha sido una experiencia muy bonita y enriquecedora.

He podido cumplir con todas mis expectativas que tenía inicialmente con este módulo e incluso después de poder investigar sobre lo que se podía hacer en Odoo, he añadido funcionalidades que no sabía que existían (por ejemplo que generará informes pdf personalizados para mis ideas, por otro lado también crear varios tipos de permisos para restringir el acceso para los usuarios que no puedan crear ideas nuevas ni cambiar el estado de las mismas...) En definitiva he cumplido con las ideas que tenía inicialmente e incluso como acabo de describir, incluso he añadido funcionalidades nuevas que no tenía previstas.

Me ha permitido mejorar mis habilidades técnicas, entender mejor la gestión de proyectos, y aprender de una forma más cercana cómo es el funcionamiento de los trabajadores dentro de una empresa. Este proyecto me ha enseñado que, más allá del código, lo más importante es cómo las soluciones que desarrollamos pueden impactar positivamente a las personas y a las organizaciones.

La verdad que estoy bastante orgulloso de haber trabajado en este módulo y ahora que ya tengo la base que he formado con este módulo, tengo ganas de seguir desarrollando nuevos módulos para odoo.



# Bibliografía

1. <https://github.com/odoo/odoo>
2. [https://www.odoo.com/documentation/17.0/developer/tutorials/master\\_odoo\\_web\\_framework/03\\_customize\\_kanban\\_view.html](https://www.odoo.com/documentation/17.0/developer/tutorials/master_odoo_web_framework/03_customize_kanban_view.html)
3. [https://www.odoo.com/documentation/17.0/developer/tutorials/pdf\\_reports.html](https://www.odoo.com/documentation/17.0/developer/tutorials/pdf_reports.html)
4. [https://www.odoo.com/es\\_ES/forum/ayuda-1/archive-unarchive-option-missing-201560](https://www.odoo.com/es_ES/forum/ayuda-1/archive-unarchive-option-missing-201560)
5. [https://www.odoo.com/es\\_ES/forum/ayuda-1/what-is-self-env-92304](https://www.odoo.com/es_ES/forum/ayuda-1/what-is-self-env-92304)