Capítulo 6. Modelos: estructuración de los datos de la aplicación

En En el Capítulo 3 (/book/business/9781789532470/3) , **Su primera aplicación Odoo** , vimos una descripción general de todos los componentes principales involucrados en la creación de una aplicación para Odoo. En este y en los próximos capítulos, entraremos en detalles sobre cada una de las capas que componen una aplicación: modelos, vistas y lógica empresarial.

En este capítulo, aprenderemos más sobre la capa de modelo y cómo usar modelos para diseñar las estructuras de datos que soportan aplicaciones. Exploraremos las capacidades de los modelos y campos, incluida la definición de relaciones de modelos, la adición de campos calculados y la creación de restricciones de datos.

Los siguientes temas se tratarán en este capítulo:

- Proyecto de aprendizaje: mejorar la aplicación Biblioteca
- Creando modelos
- Creando campos
- Relaciones entre modelos.
- Campos calculados
- Restricciones del modelo
- Sobre los modelos base de Odoo

Sección anterior Sección (/book/business/9781789532470/5/ch05lvl1sec60/further-reading)

siguiente (/book/business/9781789532470/6/ch06lvl1sec61/technical-requirements)

Requerimientos técnicos

El código de este capítulo se basa en el código creado en el Capítulo 3 (/book/business/9781789532470/3) , **Su primera aplicación Odoo** . El código necesario se puede encontrar en el `ch03/` directorio del repositorio de Git en https://github.com/PacktPublishing/Odoo-12-Development-Essentials-Fourth-Edition (https://github.com/PacktPublishing/Odoo-12-Development-Essentials-Fourth-Edition) . El código de este capítulo se puede encontrar en https://github.com/PacktPublishing/Odoo-12-Development-Essentials-Fourth-Edition/tree/master/ch05/library_app (https://github.com/PacktPublishing/Odoo-12-Development-Essentials-Fourth-Edition/tree/master/ch05/library_app) .

Debe tenerlo en su ruta de complementos e instalar el `library_app` módulo. Los ejemplos en este capítulo modificarán y agregarán código en ese módulo adicional.

Sección anterior Sección (/book/business/9781789532470/6)

siguiente (/book/business/9781789532470/6/ch06lvl1sec62/learning-project-improving-the-library-app)

Proyecto de aprendizaje: mejora de la aplicación Biblioteca

En el Capítulo 3 (/book/business/9781789532470/3) , **Su primera aplicación Odoo** , creamos `library_app` el módulo adicional, implementando un `library.book` modelo simple para representar el catálogo de libros. En este capítulo, volveremos a visitar ese módulo. para enriquecer los datos que podemos almacenar con respecto a cada libro.

Agregaremos una jerarquía de categorías, para usar en la categorización de libros, con lo siguiente:

- **Nombre** : el título de la categoría
- **Principal** : la categoría principal a la que pertenece
- **Subcategorías** : las categorías que tienen este como padre
- **Libro o autor destacado** : un libro o autor seleccionado que representa esta categoría

El modelo de libro ya tiene campos para información esencial, y agregaremos algunos más para mostrar los diversos tipos de datos disponibles en Odoo.

También agregaremos algunas restricciones al modelo de libros:

- El título y la fecha de publicación deben ser únicos.
- Los ISBN ingresados deben ser válidos

Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec61/technical-requirements)

siguiente Sección (/book/business/9781789532470/6/ch06lvl1sec63/creating-models)

Creando modelos

Los modelos están en el corazón del marco de trabajo de Odoo. Ellos describenestructura de datos de la aplicación, y son el puente entre el servidor de aplicaciones y el almacenamiento de la base de datos. La lógica de negocios se puede implementar alrededor de los modelos para proporcionar las características de la aplicación, y las interfaces de usuario se crean sobre ellas.

A continuación, aprenderemos sobre los atributos genéricos de un modelo, utilizados para influir en su comportamiento, y los diversos tipos de modelos que tenemos disponibles para usar: regular, transitorio y abstracto.

Atributos del modelo

Las clases de modelo pueden usar atributos adicionales que controlan algunos de sus comportamientos. Estos son los atributos más utilizados:

- `_name` es el identificador interno del modelo de Odoo que estamos creando. Esto es obligatorio al crear un nuevo modelo.
- `_description` es un título fácil de usar para los registros del modelo, que se muestra cuando el modelo se ve en la interfaz de usuario. Esto es opcional pero recomendado.
- `_order` establece el orden predeterminado para usar cuando se examinan los registros del modelo o se muestran en una vista de lista. Es una cadena de texto que se usará como `order by` cláusula SQL , por lo que puede ser cualquier cosa que pueda usar allí, aunque tiene un comportamiento inteligente y admite nombres de campo traducibles y muchos a uno.

Nuestro modelo de libro ya está utilizando los atributos `_name` y `_description` . Podemos agregarlo `_order` para que se ordene de manera predeterminada por el título del libro y luego por orden inverso de la fecha de publicación (del más nuevo al más antiguo):

Dupdo

```
class Book(models.Model):
    _name = 'library.book'
    _description = 'Book'
    _order = 'name, date_published desc'
```

Hay algunos atributos más que se puede usar en casos avanzados:

- `_rec_name` indica el campo a utilizar como descripción del registro cuando se hace referencia desde campos relacionados, como una relación de muchos a uno. Por defecto, usa el `name` campo, que es un campo común en los modelos, pero este atributo nos permite usar cualquier otro campo para ese propósito.

- `_table` es el nombre de la tabla de la base de datos que admite el modelo. Por lo general, el ORM lo define automáticamente, utilizando el nombre del modelo con los puntos reemplazados por guiones bajos. Pero, es posible configurarlo para indicar un nombre de tabla específico.
- `_log_access=False` se puede especificar para evitar la creación automática de los campos de seguimiento de auditoría: `create_uid` , `create_date` , `write_uid` ,y `write_date` .
- `_auto=False` can be specified to prevent the automatic creation of the underlying database table. If needed, the `init()` class method should be overridden to create the supporting database object, a table or a view.

We can also have the `_inherit` and `_inherits` attributes, used to extend modules. We will look closer at them later in this chapter.

Models and Python classes

Odoo models are represented by Python classes. In the preceding code, we have a Python class, `Book` , based on the `models.Model` class, which defines a new Odoo model called `library.book` .

Odoo models are kept in a central registry, available in the `env` environment object (referred to as a `pool` in the old API). It is a dictionary that keeps references to all the model classes available in the database, and its entries can be referenced by a model name. Specifically, the code in a model method can use `self.env['library.book']` to retrieve the model class representing the `library.book` model.

You can see that model names are important since they are the keys used to access the registry. The convention for model names is to use a list of lowercase words joined with dots, such as `library.book` or `library.book.category` . Other examples from the core modules are `project.project` , `project.task` , and `project.task.type` .

We should use the singular form for model names: `library.book` rather than `library.books` .

> **Note**
>
> For historical reasons, it is possible to find some core models that don't follow this, such as `res.users` , but this is not the rule.

Model names must be globally unique. Because of this, the first word should correspond to the main application the module relates to. For our Library app features, we will prefix all models with `library` . Other examples from the core modules are `project` , `crm` , and `sale` .

Python classes, on the other hand, are local to the Python file where they are declared. The identifier used for them is only significant for the code in that file, and is rarely relevant. Since there is no risk of collision with classes in other modules, there is no need for class identifiers to be prefixed by the main application they relate to.

The convention for class identifiers is to use `CamelCase` . This follows the Python standard defined by the PEP8 coding conventions.

Transient and abstract models

In the preceding code, and in most Odoo models, classes are based on the `models.Model` class. These types of model have permanent database persistence: database tables are created for them and their records are stored until explicitly deleted.

However, Odoo also provides two other model types to be used: transient and abstract models.

Transient models are based on the `models.TransientModel` class and are used for wizard-style user interaction. Their data is still stored in the database, but it is expected to be temporary. A vacuum job periodically clears old data from these tables. For example, the **`Load a Language`** dialog window, found in the **`Settings`** | **`Translations`** menu, uses a Transient

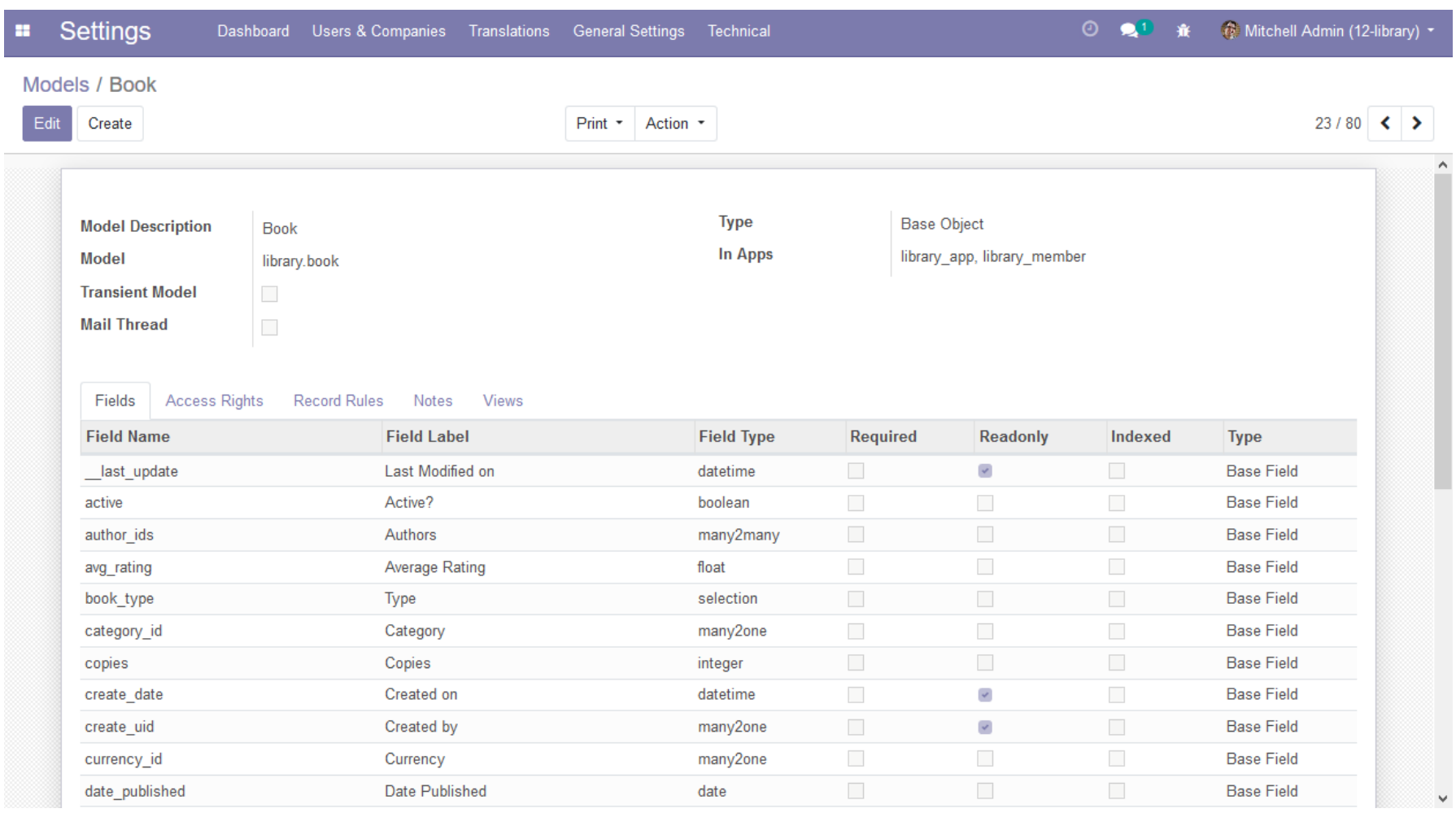
model to store user selections and implement wizard logic. An example of using a Transient model is discussed in Chapter 8 (/book/business/9781789532470/8), **Business Logic – Supporting Business Processes**.

Abstract models are based on the `models.AbstractModel` class and have no data storage attached to them. They can be used as reusable feature sets, to be mixed in with other models using Odoo's inheritance capabilities. For example, `mail.thread` is an abstract model provided by the Discuss app, used to add message and follower features to other models.

Inspecting existing models

The models and fields created through the Python classes have their metadata available through the user interface. With the **`Developer Mode`** enabled, in the **`Settings`** top menu, navigate to the **`Technical`** | **`Database Structure`** | **`Models`** menu item. Here, you will find the list of all the models available in the database.

Clicking on a model in the list will open a form with its details:



This is a good tool for inspecting the structure of a model, since, in one place, you can see the results of all the customization of different modules. In this case, as you can see near the top right of the form, in the **`In Apps`** field, the `library.book` definitions for this model come from both the `library_app` and `library_member` modules.

In the lower area, we have some tabs available with additional information:

- **`Fields`** with a quick reference for the model's fields
- **`Access Rights`** with the access control rules granted to security groups
- **`Views`** with the list of views available for this model

We can find the model's **`External Identifier`** here using the **`Developer menu`**, with the **`View Metadata`** option. The model's external identifiers, or XML IDs, are automatically generated by the ORM but are fairly predictable—for the `library.book` model, the external identifier is `model_library_book`. These XML IDs are frequently used in the CSV files defining the security ACLs.

Note

As we have seen in Chapter 1 (/book/business/9781789532470/1), **Quick Start Using the Developer Mode**, the `Models` form is editable! It's possible to create and modify models, fields, and views from here. You can use this to build prototypes that would later be implemented as add-on modules.

Previous Section (/book/business/9781789532470/6/ch06lvl1sec62/learning-project-improving-the-library-app)

Next Section (/book/business/9781789532470/6/ch06lvl1sec64/creating-fields)

Creando campos

Después de crear un nuevo modelo, el siguienteEl paso es agregarle campos. Odoo admite todos los tipos de datos básicos esperados, como cadenas de texto, enteros, números de coma flotante, booleanos, fechas, fecha-hora y datos de imágenes o binarios.

Exploremos los diversos tipos de campo disponibles en Odoo.

Tipos de campo básicos

Volveremos a visitar el modelo de libro para presentar los diversos tipos de campo disponibles. nos debeedite el `library_app/models/library_book.py` archivo y edite la `Book` clase para que se vea así:

Dupdo

```
# class Book(models.Model):
    # ...

    # String fields: name = fields.Char('Title')
    isbn = fields.Char('ISBN')
 book_type = fields.Selection(
        [('paper','Paperback'),
         ('hard','Hardcover'),
         ('electronic','Electronic'),
         ('other', 'Other')],
        'Type')
    notes = fields.Text('Internal Notes')
    descr = fields.Html('Description')

    # Numeric fields:
 copies = fields.Integer(default=1)
    avg_rating = fields.Float('Average Rating', (3, 2))
    price = fields.Monetary('Price', 'currency_id')
    currency_id = fields.Many2one('res.currency')  # price helper
```

Aquí, tenemos una muestra de los tipos de campo no relacionales disponibles en Odoo con los argumentos posicionales esperados por cada uno.

Nota

Python permite dos tipos de argumentos: posicional y palabra clave. Se espera que los argumentos posicionales se usen en un orden específico. Por ejemplo, `f(x, y)` debería llamarse con algo como `f(1, 2)` . Los argumentos de palabras clave se pasan con el nombre del argumento. Para el mismo ejemplo, podríamos usar `f(x=1, y=2)` , o incluso mezclar ambos estilos con `f(1, y=2)` . Puede encontrar más información sobre argumentos de palabras clave en la documentación oficial de Python en https://docs.python.org/3/tutorial/controlflow.html#keyword-arguments (https://docs.python.org/3/tutorial/controlflow.html#keyword-arguments) .

Para la mayoría de los campos no relacionales, el primer argumento es el título del campo, correspondiente al `string` argumento del campo. Se utiliza como texto predeterminado para las etiquetas de la interfaz de usuario. Es opcional y, si no se proporciona, se generará automáticamente una cadena de título a partir del nombre del campo, reemplazando los guiones bajos con espacios y escribiendo en mayúscula la primera letra de cada palabra.

Estos son los tipos de campo no relacionales disponibles, junto con los argumentos posicionales esperados por cada:

- `Char(string)` es una sola línea de texto El único argumento posicional esperado es la etiqueta del campo de cadena.
- `Text(string)` Es un texto multilínea. El único argumento posicional es la cadena de la etiqueta del campo.

- `Selection(selection, string)` es una lista desplegable de selección. Los argumentos posicionales de selección son una `[('value', 'Title'),]` lista de tuplas. El primer elemento de tupla es el valor almacenado en la base de datos. El segundo elemento de tupla es la descripción presentada en la interfaz de usuario. Esta lista puede ser extendida por otros módulos usando el `selection_add` argumento de la palabra clave.
- `Html(string)` se almacena como un campo de texto, pero tiene un manejo específico de la interfaz de usuario para la presentación de contenido HTML. Por razones de seguridad, se desinfecta de manera predeterminada, pero este comportamiento se puede anular.
- `Integer(string)` solo espera un argumento de cadena para el título del campo.
- `Float(string, digits)` tiene un segundo dígito de argumento opcional, una `(x,y)` tupla con la precisión del campo. `x` es el número total de dígitos; de esos, `y` son dígitos decimales.
- `Monetary(string, currency_field)` es similar a un campo flotante, pero tiene un manejo específico para la moneda. El `currency_field` segundo argumento se utiliza para establecer el campo que almacena la moneda que se utiliza. Por defecto, se espera que sea el `currency_id` campo.
- `Date(string)` y los `Datetime(string)` campos esperan solo el texto de cadena como argumento posicional.
- `Boolean(string)` contiene valores verdaderos o falsos, como es de esperar, y solo tiene un argumento posicional para el texto de la cadena.
- `Binary(string)` almacena datos binarios de tipo archivo y también espera solo el argumento de cadena. Puede ser manejado por código Python usando `base64` cadenas codificadas.

Nota

`Changed in Odoo 12` Los campos `Date` y `Datetime` ahora se manejan en el ORM como objetos de fecha. En versiones anteriores, se manejaban como representaciones de cadenas de texto y, para ser manipulados, necesitaban una conversión explícita hacia y desde los objetos de fecha de Python.

`Text` campos, `Char` , `Text` ,y `Html` , tienen unos atributos específicos:

- `size` (**Char** únicos campos) establece un tamaño máximo permitido. Se recomienda no usarlo a menos que haya una buena razón para ello, por ejemplo, un número de seguro social con una longitud máxima permitida.
- `translate` hace que los contenidos del campo sean traducibles, manteniendo diferentes valores para diferentes idiomas.
- `trim` , establecido en True de forma predeterminada, activa el recorte automático del espacio en blanco circundante, realizado por el cliente web. Esto se puede deshabilitar explícitamente mediante la configuración `trim=false` .

Nota

`Changed in Odoo 12` El atributo de campo de recorte se introdujo en Odoo 12. En versiones anteriores, los campos de texto guardaban el espacio en blanco circundante.

Aparte de estos, también tenemos los campos relacionales, que se presentarán más adelante en este capítulo. Sin embargo, antes de eso, todavía hay más por saber sobre los atributos de los tipos de campo.

Atributos de campo comunes

Los campos tienen atributos adicionales disponibles para nosotros definir su comportamiento.

Estos son los atributos generalmente disponibles, generalmente utilizados como argumentos de palabras clave:

- `string` es la etiqueta predeterminada del campo, que se utilizará en la interfaz de usuario. Excepto para `Selection` y los `Relational` campos, es el primer argumento posicional, por lo que la mayoría de las veces no se usa como argumento de palabra clave. Si no se proporciona, se genera automáticamente a partir del nombre del campo.
- `default` establece un valor predeterminado para el campo. Puede ser un valor específico (como `default=True` en el `active` campo) o una referencia invocable, ya sea una función con nombre o una función anónima.
- `help` proporciona el texto para la información sobre herramientas que se muestra a los usuarios cuando pasan el mouse sobre el campo en la interfaz de usuario.
- `readonly=True` hace que el campo no sea editable en la interfaz de usuario de forma predeterminada. Esto no se aplica a nivel API; el código en los métodos modelo todavía será capaz de escribir en él. Es solo una configuración de interfaz de usuario.
- `required=True` hace que el campo sea obligatorio en la interfaz de usuario de forma predeterminada. Esto se aplica a nivel de la base de datos al agregar una `NOT NULL` restricción en la columna.
- `index=True` agrega un índice de base de datos en el campo, para operaciones de búsqueda más rápidas a expensas de operaciones de escritura ligeramente más lentas.
- `copy=False` tiene el campo ignorado cuando se utiliza la función de registro duplicado, el `copy()` método ORM. Los valores de campo se copian de manera predeterminada, excepto para muchos campos relacionales, que no se copian de manera predeterminada.
- `groups` permite limitar el acceso y la visibilidad del campo a solo algunos grupos. Espera una lista separada por comas de ID XML para grupos de seguridad, como `groups='base.group_user,base.group_system'` .
- `states` espera valores de mapeo de diccionario para atributos de IU dependiendo de los valores del `state` campo. Los atributos que se pueden utilizar son `readonly` , `required` y `invisible` , por ejemplo, `states={'done': [('readonly',True)]}` .

Nota

Tenga en cuenta que el `states` atributo de campo es equivalente al `attrs` atributo en las vistas. Además, tenga en cuenta que las vistas admiten un `states` atributo, pero tiene un uso diferente: acepta una lista de estados separados por comas para controlar cuándo el elemento debe ser visible.

Aquí hay un ejemplo del uso de argumentos de palabras clave para atributos de campo:

Dupdo

```
    name = fields.Char(
'Title',
        default=None,
        index=True,
        help='Book cover title.',
        readonly=False,
        required=True,
        translate=False,
    )
```

Como se mencionó anteriormente, el atributo predeterminado puede tener un valor fijo o una referencia a una función para calcular dinámicamente el valor predeterminado. Para cálculos triviales, podemos usar una `lambda` función para evitar la sobrecarga de crear una función o método con nombre. Aquí hay un ejemplo común, calcular un valor predeterminado con la fecha y hora actuales:

Dupdo

```
    last_borrow_date = fields.Datetime(
        'Last Borrowed On',
        default=lambda self: fields.Datetime.now(),
    )
```

El `default` valor también puede ser una referencia de función o una cadena con el nombre de una función aún por declarar:

Dupdo

```
    last_borrow_date = fields.Datetime(
        'Last Borrowed On',
default='_default_last_borrow_date',
    )

    def _default_last_borrow_date(self):
        return fields.Datetime.now()
```

Estos dos atributos pueden ser útiles cuando la estructura de datos del módulo cambia entre versiones:

- `deprecated=True` registra una advertencia cada vez que se utiliza el campo.
- `oldname='field'` se utiliza cuando se cambia el nombre de un campo en una versión más nueva, lo que permite que los datos del campo antiguo se copien automáticamente en el nuevo campo, en la próxima actualización del módulo.

Nombres de campo especiales

Algunos nombres de campo son especiales, ya sea porque están reservados por el ORM para fines especiales, o porque algunas características incorporadas utilizan algunos nombres de campo predeterminados.

El `id` campo está reservado para usarse como un número automático que identifica de forma única cada registro y la clave primaria de la base de datos. Se agrega automáticamente a cada modelo.

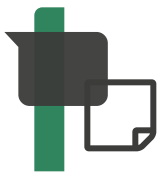
Los siguientes campos se crean automáticamente en los nuevos modelos, a menos que se establezca el `_log_access=False` atributo del modelo:

- `create_uid` es para el usuario que creó el registro.
- `create_date` es para la fecha y hora en que se crea el registro.
- `write_uid` es para que el último usuario modifique el registro.
- `write_date` es para la última fecha y hora en que se modificó el registro.

La información en estos campos está disponible en el cliente web, para cualquier registro, en el **`Developer Mode`** menú, con la **`View Metadata`** opción.

Algunas características de API incorporadas esperan nombres de campo específicos de forma predeterminada. Nuestras vidas serían más fáciles si pudiéramos evitar el uso de estos nombres de campo para fines distintos a los previstos. Algunos de ellos están realmente reservados y no se pueden usar para otros fines:

- `name` (generalmente a `Char`) se usa de forma predeterminada como el nombre para mostrar del registro. Por lo general, se trata de una `Char` , pero también puede ser una `Text` o un `Many2one` tipo de campo. El campo utilizado para el nombre para mostrar se puede cambiar con el `_rec_name` atributo Modelo.
- `active` (tipo `Boolean`) nos permite desactivar registros. Los registros con `active=False` se excluirán automáticamente de las consultas. Este filtro automático se puede deshabilitar agregando `{'active_test': False}` al contexto actual. Se puede usar como un **archivo de** registro o **una** función de **borrado suave** .
- `state` (tipo `Selection`) representa los estados básicos del ciclo de vida del registro. Permite usar el `states` atributo de campo para tener un comportamiento de UI diferente según el estado del registro. Dinámicamente modificar la vista: los campos se pueden hacer `readonly` , `required` o `invisible` en estados registro específico.
- `parent_id` y `parent_path` (escriba `Integer` y `Char`) tienen un significado especial para padreo relaciones jerárquicas infantiles. Los discutiremos en detalle más adelante en este capítulo.



Nota

Changed in Odoo 12 Las relaciones jerárquicas ahora usan el `parent_path` campo, que reemplaza los campos `parent_left` y `parent_right` (de tipo Integer) usados en versiones anteriores, ahora en desuso.

Hasta ahora, hemos discutido los campos no relacionales. Pero una buena parte de la estructura de datos de una aplicación consiste en describir las relaciones entre entidades. Miremos eso ahora.

‹ Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec63/creating-models)

siguiente › (/book/business/9781789532470/6/ch06lvl1sec65/relationships-between-models)

Relaciones entre modelos.

Las aplicaciones comerciales no triviales tienen un modelo de datos estructurado y necesitan relacionar los datos de las diferentes entidades involucradas. Para hacer esto, necesitamos usar campos relacionales.

Mirando de nuevo a nuestra aplicación Biblioteca, en el modelo de libro podemos ver las siguientes relaciones:

- Cada libro puede tener un editor. Esta es una relación de **muchos a uno** , implementada en la base de datosmotor como clave foránea. Lo inverso es una relación de **uno a muchos** , lo que significa que cada editor puede tener muchos libros.
- Cada libro puede tener muchos autores. Esa es una relación de **muchos a muchos** . La relación inversa también es de muchos a muchos, ya que cada autor puede tener muchos libros.

Exploraremos cada una de estas relaciones en las siguientes secciones.

Un caso particular son las relaciones jerárquicas, donde los registros en un Modelo están relacionados con otros registros en el mismo Modelo. Introduciremos un modelo de categoría de libro para explicar ese caso.

Finalmente, el marco Odoo también admite relaciones flexibles, donde un campo puede apuntar a registros en diferentes tablas. Estos se llaman `Reference` campos.

Relaciones de muchos a uno

Una relación de muchos a uno es una referencia a un registro en otro modelo. Por ejemplo, en el modelo de libro de la biblioteca, el `publisher_id` campo representa al editor del libro y es una referencia a un registro en el modelo asociado:

Dupdo

```
    publisher_id = fields.Many2one(
        'res.partner', string='Publisher')
```

Los `Many2one` campos, primero posicionalEl argumento es el modelo relacionado (el `comodel` argumento de la palabra clave), como es el caso de todos los campos relacionales.

El segundo argumento posicional es la etiqueta del campo (el `string` argumento de la palabra clave), pero este no es el caso para los otros campos relacionales, por lo que la opción preferida es usar siempre `string` como argumento de la palabra clave, como lo hicimos en el código anterior.

Un campo Modelo de varios a uno crea un campo en la tabla de la base de datos, con una clave foránea para la tabla relacionada, y contiene el ID de la base de datos del registro relacionado.

Los siguientes argumentos de palabras clave También se pueden utilizar campos específicos para muchos a uno:

- `ondelete` define lo que sucede cuando se elimina el registro relacionado:
  - `set null` (el valor predeterminado): se establece un valor vacío cuando se elimina el registro relacionado
  - `restricted` genera un error que impide la eliminación
  - `cascade` también eliminará este registro cuando se elimine el registro relacionado
- `context` es un diccionario de datos, significativo para las vistas del cliente web, para transportar información cuando se navega por la relación, por ejemplo, para establecer valores predeterminados. Se explicará mejor en el Capítulo 8

(/book/business/9781789532470/8) , **Lógica empresarial: procesos empresariales de apoyo** .

- `domain` es una expresión de dominio: una lista de tuplas utilizadas para filtrar los registros disponibles para su selección en el campo de relación. Consulte el Capítulo 8 (/book/business/9781789532470/8) , **Lógica empresarial: procesos empresariales de apoyo,** para obtener más detalles.
- `auto_join=True` permite que ORM use combinaciones SQL al realizar búsquedas usando esta relación. Si se usa, se omitirán las reglas de seguridad de acceso y el usuario podría tener acceso a los registros relacionados que las reglas de seguridad no permitirían, pero las consultas SQL serán más eficientes y se ejecutarán más rápido.
- `delegate=True` crea una herencia de delegación con el modelo relacionado. Cuando se usa, también debe establecer `required=True` y `ondelete='cascade'` . Consulte el **Capítulo 4 (/book/business/9781789532470/4) , Módulos de extensión,** para obtener más información sobre la herencia de delegación.

Relaciones inversas uno a muchos

Una relación uno a muchos es la inversa de los muchos a uno. Enumera los registros.del modelo relacionado que tiene una referencia a este registro.

Por ejemplo, en el modelo de libro de la biblioteca, el `publisher_id` campo es una relación de muchos a uno con el modelo asociado. Esto significa que el modelo de socio puede tener una relación inversa de uno a muchos con el modelo de libro, enumerando los libros publicados por cada socio.

Para tener esa relación disponible, podemos agregarla en el modelo de socio. Agregue el archivo con esto: `library_app/models/res_partner.py`

Dupdo

```
from odoo import fields, models

class Partner(models.Model):
    _inherit = 'res.partner'    published_book_ids = fields.One2many(
        'library.book',  # related model        'publisher_id',  # field for "this" on related model       string='Pub
```

Como estamos agregando un nuevo archivo de código al módulo, no debemos olvidar importarlo también en el `library_app/models/__init__.py` archivo:

Dupdo

```
from . import library_book
from . import res_partner
```

Los `One2many` campos aceptan tres argumentos posicionales:

- El modelo relacionado (`comodel_name` argumento de palabra clave).
- El campo en ese modelo que se refiere a este registro (`inverse_name` argumento de palabra clave).
- La etiqueta de campo (`string` argumento de palabra clave).

Los argumentos de palabras clave adicionales disponibles sonlos mismos que para muchos-a-uno campos: `context` , `domain` ,y `ondelete` (en este caso actúa sobre el **que muchos** lado de la relación).

Relaciones de muchos a muchos

Una relación de muchos a muchos se usa cuandoTenemos una relación de muchos en ambos lados. Tomando nuevamente nuestro ejemplo de libros de la biblioteca, podemos encontrar una relación de muchos a muchos entre los libros y autores: cada libro puede tener muchos autores, y cada autor puede tener muchos libros.

En el lado de los libros, tenemos en el `library.book` modelo:

Dupdo

```
class Book(models.Model)
    _name = 'library.book'
author_ids = fields.Many2many(
        'res.partner', string='Authors')
```

Del lado de los autores, podemos agregar la relación inversa al `res.partner` modelo:

Dupdo

```
class Partner(models.Model):
    _inherit = 'res.partner'
    book_ids = fields.Many2many(
        'library.book', string='Authored Books')
```

La `Many2many` firma mínima acepta unoargumento posicional para el modelo relacionado (el `comodel_name` argumento de la palabra clave), y se recomienda encarecidamente proporcionar también el `string` argumento con la etiqueta de campo.

En el nivel de la base de datos, las relaciones de muchos a muchos no agregan columnas a las tablas existentes. En cambio, se crea automáticamente una tabla de relaciones especiales para almacenar las relaciones entre registros. Esta tabla especial tiene solo dos campos ID, con claves foráneas para cada una de las dos tablas relacionadas.

De forma predeterminada, el nombre de la tabla de relación son los dos nombres de tabla unidos con un guión bajo y `_rel` agregados al final. En el caso de nuestra relación de libros o autores, debe nombrarse `library_book_res_partner_rel` .

En algunas ocasiones, es posible que debamos anular estos valores predeterminados automáticos. Uno de esos casos es cuando los modelos relacionadostienen nombres largos y el nombre de la tabla de relaciones generada automáticamente es demasiado largo, excediendo el límite de PostgreSQL de 63 caracteres. En estos casos, debemos elegir manualmente un nombre para que la tabla de relaciones se ajuste al límite de tamaño del nombre de la tabla.

Otro caso es cuando necesitamos una segunda relación de muchos a muchos entre los mismos modelos. En estos casos, debemos proporcionar manualmente un nombre para la tabla de relaciones para que no choque con el nombre de la tabla que ya se está utilizando para la primera relación.

Hay dos alternativas para anular manualmente estos valores: ya sea utilizando argumentos posicionales o argumentos de palabras clave.

Usando argumentos posicionales para la definición de campo, tenemos lo siguiente:

Dupdo

```
# Book <-> Authors relation (using positional args)
author_ids = fields.Many2many(
    'res.partner',      # related model (required)    'library_book_res_partner_rel',  # relation table name to use
    'a_id',             # rel table field for "this" record
    'p_id',             # rel table field for "other" record
    'Authors')          # string label text
```

En cambio, podemos usar argumentos de palabras clave, que pueden ser preferibles para la legibilidad:

Dupdo

```
# Book <-> Authors relation (using keyword args)
author_ids = fields.Many2many(
    comodel_name='res.partner', # related model (required)
    relation='library_book_res_partner_rel', # relation table name
column1='a_id', # rel table field for "this" record
 column2='p_id', # rel table field for "other" record
 string='Authors') # string label text
```

Similar a uno a muchos relacionalescampos, campos de muchos a muchos puedenTambién utilizar los argumentos de palabras clave `context` , `domain` y `auto_join` .

> **Nota**
>
> Al crear modelos abstractos, no use los campos `column1` y `column2` atributos de muchos a muchos . Existe una limitación en el diseño de ORM con respecto a los modelos abstractos, y cuando fuerza los nombres de las columnas de relación, ya no se pueden heredar limpiamente.

Relaciones jerárquicas

Las relaciones de árbol padre-hijo sonrepresentado mediante una relación de muchos a uno con el mismo modelo, utilizado para cada registro para hacer referencia a su padre. La relación inversa uno a muchos corresponde a los hijos directos del registro.

Odoo proporciona un soporte mejorado para estas estructuras de datos jerárquicos, con los operadores adicionales `child_of` y `parent_of` disponibles en las expresiones de dominio. Estos operadores están disponibles siempre que el modelo tenga un `parent_id` campo (o tenga una `_parent_name` definición de Modelo válida).

Podemos habilitar consultas más rápidas en el árbol de jerarquía configurando el `_parent_store=True` atributo Modelo y agregando el `parent_path` campo auxiliar. Este campo almacena información adicional sobre la estructura de árbol de jerarquía que se aprovecha para consultas más rápidas.

> **Nota**
>
> **`Changed on Odoo 12`** El `parent_path` campo auxiliar se introdujo en Odoo 12. Las versiones anteriores usaban los campos `parent_left` y `parent_right` entero para el mismo propósito. Estos están en desuso a partir de Odoo 12.

Tenga en cuenta que estas operaciones adicionales vienen con penalizaciones de almacenamiento y tiempo de ejecución, por lo que se utilizan mejor cuando espera leercon mayor frecuencia que escribir, como en el caso de los árboles de categorías. Esto solo es necesario cuando se optimizan jerarquías profundas con muchos nodos, y se puede utilizar de forma incorrecta para jerarquías pequeñas o poco profundas.

Para mostrar las estructuras jerárquicas, agregaremos un árbol de Categoría a la aplicación Biblioteca, que se utilizará para clasificar nuestros Libros. Para esto, agregaremos el `library_app/models/library_book_category.py` archivo con este código:

Dupdo

```
from odoo import api, fields, models

class BookCategory(models.Model):
    _name = 'library.book.category'
    _description = 'Book Category'
    _parent_store = True

    name = fields.Char(translate=True, required=True)
    # Hierarchy fields
    parent_id = fields.Many2one(
        'library.book.category',
        'Parent Category',
        ondelete='restrict')
    parent_path = fields.Char(index=True)

    # Optional but good to have:
    child_ids = fields.One2many(
        'library.book.category',
        'parent_id',
```

Aquí, tenemos un modelo básico con un `parent_id` campo para hacer referencia al registro padre.

Para habilitar la indexación de la jerarquía, para una búsqueda de árbol más rápida, agregamos el `_parent_store=True` atributo modelo. Al hacerlo, el `parent_path` campo también debededebe agregarse y debe indexarse. Se espera que el campo utilizado para referirse al padre sea nombrado `parent_id` , pero cualquierse puede usar otro nombre de campo siempre que lo declaremos en el `_parent_name` atributo del modelo opcional.

A menudo es conveniente agregar un campo para enumerar los hijos directos. Esta es la relación inversa de uno a muchos vista en el código anterior.

Para que nuestro módulo use el código anterior, recuerde agregar una referencia a su archivo en `library_app/models/__init__.py` :

Dupdo

```
from . import library_book_category
from . import library_book
from . import res_partner
```

Relaciones flexibles usando campos de referencia

Referencia de campos relacionales regularesun comodelo fijo. El `Reference` tipo de campo no tiene esta limitación. y admite relaciones flexibles, de modo que el mismo campo no esté restringido para que siempre apunte al mismo modelo de destino.

As an example, we will use it in our book category model to add a reference to a highlighted book or author. So, the field could refer to either a book or a partner:

Copy

```
# class BookCategory(models.Model):
    highlighted_id = fields.Reference(
        [('library.book', 'Book'), ('res.partner', 'Author')],
        'Category Highlight',
    )
```

The field definition is similar to a `selection` field, but here the selection list holds the models available to be used on the field. In the user interface, the user will first pick a model from the available list, and then pick a specific record from that model.

Note

`Changed in Odoo 12` The referenceable models configuration table was removed. In previous Odoo versions, it could be used to configure the models that can be used in `Reference fields` . It was available in the **`Settings`** | **`Technical`** | **`Database Structure`** menu. These configuration could be leveraged in `Reference field` using the `odoo.addons.res.res_request.referenceable_models` function in place of the model selection list.

Aquí hay algunos detalles técnicos adicionales sobre los campos de referencia que pueden ser útiles:

- Los campos de referencia se almacenan en la base de datos como una `model,id` cadena
- El `read()` método, destinado para su uso desde aplicaciones externas, los devuelve formateados como una `('model_name', id)` tupla, en lugar del `(id, 'display_name')` par habitual para campos de muchos a uno

Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec64/creating-fields)

siguiente › (/book/business/9781789532470/6/ch06lvl1sec66/computed-fields)

Campos calculados

Los campos pueden tener sus valores.calculado automáticamente por una función, en lugar de simplemente leer una base de datosvalor almacenado. Un campo calculado se declara como un campo normal, pero tiene el `compute` argumento adicional para definir la función utilizada para su cálculo.

En la mayoría de los casos, los campos calculados implican escribir algunoslógica de negocios. Por lo tanto, para aprovechar al máximo esta función, necesitamos aprender los temas explicados en el Capítulo 8 (/book/business/9781789532470/8) , **Lógica empresarial: procesos empresariales de apoyo** . Todavía podemos explicar los campos calculados aquí, pero mantendremos la lógica de negocios lo más simple posible.

Trabajemos en un ejemplo. Los libros tienen una editorial. Nos gustaría tener el país del editor en forma de libro.

Para esto, utilizaremos un campo calculado, basado en el `publisher_id` , que tomará su valor del `country_id` campo del editor .

Deberíamos editar el modelo de libro en el `library_app/models/library_book.py` archivo para agregar lo siguiente:

Dupdo

```
# class Book(models.Model):
    publisher_country_id = fields.Many2one(
        'res.country', string='Publisher Country',
        compute='_compute_publisher_country',
    )

    @api.depends('publisher_id.country_id')
    def _compute_publisher_country(self):
        for book in self:
            book.publisher_country_id = book.publisher_id.country_id
```

El código anterior agrega el `publisher_country_id` campo y el `_compute_publisher_country` método utilizado para calcularlo. El nombre de la función se pasó al campo como un argumento de cadena, pero también se le puede pasar una referencia invocable (el identificador de la función, sin las comillas). En ese caso, debemos asegurarnos de que la función esté definida en el archivo Python antes que el campo.

El `@api.depends` decorador es necesario cuandoel cálculo depende de otros campos, como suele suceder. Le permite al servidor saber cuándo volver a calcular los valores almacenados o en caché. Se aceptan uno o más nombres de campo como argumentos y la notación de puntos se puede utilizar para seguir las relaciones de campo. En nuestro caso, nuestro campo debe ser recalculado cada vez que se cambie `country_id` el del libro `publisher_id` .

Como de costumbre, el `self` argumento es el objeto de conjunto de registros con el que trabajar. Entonces, necesitamos iterar sobre él para actuar en cada registro individual. El valor calculado se establece mediante la operación habitual de asignación (escritura). En nuestro caso, el cálculo es bastante simple, lo asignamos al `publisher_id.country_id` valor del libro actual .

Se puede usar el mismo método de cálculo para más de un campo. En ese caso, se usa el mismo método en varios `compute` argumentos de campo, y el método de cálculo debe asignar valores a todos los campos calculados.

> **Nota**
>
> La función de cálculo debe asignar un valor al campo, o campos, para calcular. Si su método de cálculo tiene `if` condiciones, asegúrese de que todas las rutas de ejecución asignen valores a los campos calculados. De lo contrario, el cálculo producirá un error en los casos en que no pueda asignar un valor a los campos calculados.

Todavía no estaremos trabajando en las vistas para este módulo, pero puede hacer una edición rápida en el formulario de tareas para confirmar que el campo calculado funciona como se esperaba utilizando **`Developer Mode`** , seleccionando la **`Edit View`** opción y agregando el campo directamente en el formulario XML No se preocupe, será reemplazado por la vista de módulo limpio en la próxima actualización.

Buscar y escribir en campos calculados

El campo calculado que acabamos de crear se puede leer, pero no se puede buscaro escrito a. Por defecto, los valores de campo calculados se escriben sobre la marcha y no se almacenan en la base de datos. Es por eso que no podemos buscarlos como podemos hacer campos regulares.

Podemos habilitar estas operaciones de búsqueda y escritura implementando funciones especializadas para ellos. Junto con la `compute` función, también podemos establecer una `search` función para implementar la lógica de búsqueda y la `inverse` función para implementar la lógica de escritura.

Usando estos, nuestro cálculo La declaración de campo tiene el siguiente aspecto:

Dupdo

```
# class Book(models.Model):
    publisher_country_id = fields.Many2one(
        'res.country', string='Publisher Country',
        compute='_compute_publisher_country',
        # store = False,  # Default is not to store in db
inverse='_inverse_publisher_country',
        search='_search_publisher_country',
    )
```

Escribir en un campo calculado es la lógica **inversa** del cálculo. Entonces, la función encargada de manejar la operación de escritura se llama **inversa** . En nuestro caso, la función inversa es simple. El cálculo simplemente copió el `book.publisher_id.country_id` valor a `book.publisher_country_id` . La operación inversa es copiar el valor escrito en `book.publisher_country_id` el `book.publisher_id.country_id` campo:

Dupdo

```
    def _inverse_publisher_country(self):
        for book in self:
            book.publisher_id.country_id = book.publisher_country_id
```

Tenga en cuenta que esto modifica los datos en el registro de socio del editor y, por lo tanto, también cambiará el valor que se ve en todos los libros con el mismo editor. Los controles de acceso regulares se aplican a estas operaciones de escritura, por lo que esta acción solo tendrá éxito si el usuario actual también tiene acceso de escritura al modelo asociado.

Para habilitar las operaciones de búsqueda en un campo calculado, necesitamos implementar su función de **búsqueda** . Para esto, necesitamospara poder convertir un dominio de búsqueda en el campo calculado a un dominio de búsqueda usando campos almacenados regulares. En nuestro caso, la búsqueda real debe realizarse en el `country_id` campo del `publisher_id` registro de socio vinculado :

Dupdo

```
    def _search_publisher_country(self, operator, value):
        return [('publisher_id.country_id', operator, value)]
```

Cuando realizamos una búsqueda en un Modelo, se usa una expresión de dominio como argumento con el filtro para aplicar. Las expresiones de dominio se explican con más detalle en el Capítulo 8 (/book/business/9781789532470/8) , **Lógica empresarial: procesos empresariales de apoyo** , pero, por ahora, debe saber que son una lista de `(field, operator, value)` condiciones.

La `search` función se llama siempre que este campo calculado se encuentra en condiciones de una expresión de dominio. Recibe el `operator` y `value` para la búsqueda y se espera que traduzca el elemento de búsqueda original en una expresión de búsqueda de dominio alternativa. El `country_id` campo se almacena en el modelo de socio relacionado, por lo que nuestra implementación de búsqueda solo altera la expresión de búsqueda original para utilizar el `publisher_id.country_id` campo.

Almacenar campos calculados

Los valores de campo calculados también se pueden almacenar en la base de datos, estableciendo `store = True` en su definición. Se volverán a calcular cuando cambie cualquiera de sus dependencias. Dado que los valores ahora están almacenados, se pueden buscar al igual que los campos normales, y no se necesita una función de búsqueda.

Campos relacionados

El campo calculado que implementamos en la sección anteriorsimplemente copia un valor de un registro relacionado en el propio campo de un modelo. Este es un caso de uso común que Odoo puede manejar automáticamente utilizando la función de **campo relacionada** .

Los campos relacionados ponen a disposición, directamente en un modelo, campos que pertenecen a un modelo relacionado y son accesibles mediante una cadena de notación de puntos. Esto los hace disponibles en situaciones en las que no se puede usar la notación de puntos, como las vistas de formulario de IU.

Para crear un campo relacionado, declaramos un campo del tipo requerido, al igual que con los campos calculados normales, pero en lugar de `compute` usar el `related` atributo, configurándolo con la cadena de campo de notación de puntos para alcanzar el campo deseado.

Podemos usar un campo de referencia para obtener exactamente el mismo efecto que el ejemplo anterior, el `publisher_country_id` campo calculado:

Dupdo

```
# class Book(models.Model):
    publisher_country_related = fields.Many2one(
        'res.country', string='Publisher Country (related)',
        related='publisher_id.country_id',
    )
```

Detrás de escena, los campos relacionados son solo campos calculados que implementan convenientemente `search` y `inverse` métodos. Esto significa que podemos buscarlos y escribirlos directamente, sin tener que escribir ningún código adicional. Por defecto, los campos relacionados son de solo lectura, por lo que la operación de escritura inversa no estará disponible. Para habilitarlo, establezca el `readonly=False` atributo de campo.

> Nota
>
> **`Changed in Odoo 12`** Los `related` campos están ahora de sólo lectura de forma predeterminada: `readonly=True` . En versiones anteriores de Odoo, se podían escribir de forma predeterminada, pero se demostró que era un valor predeterminado peligroso, ya que podía permitir cambios en la configuración o los datos maestros en los casos en que no se esperaba que se permitiera.

También vale la pena señalar que estos campos relacionados también se pueden almacenar en una base de datos utilizando `store=True` , al igual que cualquier otro campo calculado.

Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec65/relationships-between-models)

siguiente Sección (/book/business/9781789532470/6/ch06lvl1sec67/model-constraints)

Restricciones del modelo

A menudo, las aplicaciones deben garantizar la integridad de los datos y aplicar algunas validaciones para garantizar esos datos son completos y correctos.

El administrador de bases de datos PostgreSQL admite muchas validaciones útiles, como evitar duplicados o verificar que los valores cumplan ciertas condiciones simples. El modelo puede declarar y usar restricciones de PostgreSQL para esto.

Algunas comprobaciones requieren una lógica más sofisticada y se implementan mejor como código Python. Para estos casos, podemos usar métodos de modelo específicos que implementan la lógica de restricción de Python.

Restricciones del modelo SQL

Se agregan restricciones SQL a la base de datosdefinición de tabla y se aplican directamente por PostgreSQL. Estan definidosutilizando el `_sql_constraints` atributo de clase

Es una lista de tuplas, y cada tupla tiene el formato `(name, code, error)` :

- **nombre** es el nombre del identificador de restricción
- **el código** es la sintaxis de PostgreSQL para la restricción
- **error** es el mensaje de error que se presentará a los usuarios cuando no se verifique la restricción

Agregaremos dos restricciones SQL al modelo de libro. Una es una restricción única que garantiza que no tengamos libros repetidos con el mismo título y fecha de publicación; el otro es verificar que la fecha de publicación no sea futura:

Dupdo

```
# class Book(models.Model):
    _sql_constraints = [
        ('library_book_name_date_uq',  # Constraint unique identifier
         'UNIQUE (name, date_published)',  # Constraint SQL syntax
         'Book title and publication date must be unique.'),  # Message
        ('library_book_check_date',
         'CHECK (date_published <= current_date)',
         'Publication date must not be in the future.'),
    ]
```

Para obtener más información sobre la sintaxis de restricciones de PostgreSQL, consulte la documentación oficial en https://www.postgresql.org/docs/10/static/ddl-constraints.html (https://www.postgresql.org/docs/10/static/ddl-constraints.html) .

Restricciones del modelo de Python

Las restricciones de Python pueden usar una pieza de código arbitrario para verificarlas condiciones. La función de verificación debe estar decoradacon `@api.constrains` y una indicación de la lista de campos involucrados en la verificación. La validación se activa cuando cualquiera de ellos se modifica y generará una excepción si la condición falla.

En el caso de la aplicación Biblioteca, un ejemplo obvio es evitar la inserción de números ISBN incorrectos. Ya tenemos la lógica para verificar que un ISBN sea correcto, en el `_check_isbn()` método. Podemos usarlo ahora en una restricción de modelo para evitar guardar datos incorrectos:

Dupdo

```
from odoo.exceptions import ValidationError

# class Book(models.Model):
    @api.constrains('isbn')
    def _constrain_isbn_valid(self):
        for book in self:
            if book.isbn and not book._check_isbn():
                raise ValidationError(
                    '%s is an invalid ISBN' % book.isbn)
```

❮ Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec66/computed-fields)

siguiente ❯ (/book/business/9781789532470/6/ch06lvl1sec68/about-the-odoo-base-models)

Acerca de los modelos base de Odoo

En los capítulos anteriores, tuvimos la oportunidad de crearModelos nuevos, como el modelo Libro, pero también hicimos uso de los modelos ya existentes, como el modelo asociado, proporcionado por Odoo de forma inmediata. Tendremos aquí una introducción básica a estos modelos incorporados.

En el núcleo de Odoo, tenemos `base` el módulo adicional. Proporciona las características esenciales necesarias para las aplicaciones Odoo. Luego, tenemos un conjunto de módulos complementarios integrados, que proporcionan las aplicaciones y funciones oficiales disponibles con el producto estándar.

El módulo base proporciona dos tipos de modelos:

- Repositorio de información, `ir.*` modelos
- Recursos, `res.*` modelos

El **repositorio de información** se utiliza para almacenar los datos que necesita Odoopara saber cómo trabajar como una aplicación, como menús, vistas, modelos, acciones, etc. Los datos que encontramos en el **`Technical`** menú generalmente se almacenan en modelos de repositorio de información. Algunos ejemplos relevantes son estos:

- `ir.actions.act_window` para acciones de Windows
- `ir.ui.menu` para elementos de menú
- `ir.ui.view` para vistas
- `ir.model` para modelos
- `ir.model.fields` para campos modelo
- `ir.model.data` para ID de XML

Los recursos contienen datos básicos sobre el mundo, que pueden ser útiles para aplicaciones en general. Estos son los modelos de recursos más importantes:

- `res.partner` para socios comerciales, como clientes, proveedores, etc., y direcciones
- `res.company` para datos de la empresa
- `res.currency` para monedas
- `res.country` para países
- `res.users` para usuarios de la aplicación
- `res.groups` para grupos de seguridad de aplicaciones

Esto debería proporcionarle un contexto útil para comprender mejor de dónde provienen los modelos cada vez que los encuentre en el futuro.

Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec67/model-constraints)

siguiente (/book/business/9781789532470/6/ch06lvl1sec69/summary)

Resumen

A medida que nos acercamos al final de este capítulo, ahora deberíamos estar familiarizados con todas las posibilidades que los modelos nos dan para estructurar modelos de datos.

Hemos visto que los modelos generalmente se basan en la `models.Model` clase, pero también podemos usarlos `models.Abstract` para modelos mixin reutilizables y `models.Transient` para asistentes o diálogos avanzados de interacción con el usuario. Hemos visto los atributos generales del modelo disponibles, como el `_order` orden de clasificación predeterminado y `_rec_name` el campo predeterminado que se utilizará para la representación de registros.

Los campos en un modelo definen todos los datos que almacenarán. También hemos visto los tipos de campo no relacionales disponibles y los atributos que admiten. También aprendimos sobre los diversos tipos de campos relacionales, muchos a uno, uno a muchos y muchos a muchos, y cómo definen las relaciones entre modelos, incluidas las relaciones jerárquicas padre / hijo.

La mayoría de los campos almacenan la entrada del usuario en la base de datos, pero los campos pueden tener valores calculados automáticamente por el código Python. Hemos visto cómo implementar campos calculados y algunas posibilidades avanzadas que tenemos, como hacer que se puedan escribir y buscar.

También parte de las definiciones del modelo son las restricciones, que imponen la coherencia y validación de los datos. Estos se pueden implementar utilizando PostgreSQL o código Python.

Una vez que hayamos creado el modelo de datos, debemos completarlo con algunos datos predeterminados y de demostración. En el próximo capítulo, aprenderemos cómo usar archivos de datos para exportar, importar y cargar datos usando nuestro sistema.

Sección anterior Sección (/book/business/9781789532470/6/ch06lvl1sec68/about-the-odoo-base-models)

siguiente (/book/business/9781789532470/7)