Capítulo 8. Lógica empresarial: procesos empresariales de apoyo

En los capítulos anteriores, aprendimos sobre la capa del modelo, cómo construir las estructuras de datos de la aplicación y luego cómo usar la API ORM para almacenar y explorar esos datos. En este capítulo, aprovecharemos lo que aprendimos sobre modelos y manejo de conjuntos de registros para implementarpatrones de lógica de negocios que son comunes en las aplicaciones.

Discutiremos los siguientes temas en este capítulo:

- Uso de etapas para flujos de trabajo centrados en documentos
- Los decoradores de métodos ORM; `@api.multi ` ` @api.one `y `@api@model`
- Sobre los métodos de cambio, para una reacción inmediata a los usuarios
- Usando el ORM incorporados en los métodos, como por ejemplo `create , write y unlink`
- Las características de mensaje y actividad, proporcionadas por el `mail` módulo adicional
- Crear un asistente para ayudar a los usuarios a realizar acciones no triviales
- Uso de mensajes de registro para una mejor observabilidad del sistema
- Aumento de excepciones para dar retroalimentación al usuario cuando las cosas salen mal
- Pruebas unitarias para automatizar los controles de calidad en su código
- Herramientas de desarrollo, que ayudan al trabajo del desarrollador, como la depuración

Sección anterior Sección (/book/business/9781789532470/7/ch07lvl1sec78/summary)

siguiente (/book/business/9781789532470/8/ch08lvl1sec79/technical-requirements)

Requerimientos técnicos

En este capítulo, se crea un nuevo `library_checkout` módulo adicional que depende de la `library_app` y `library_member` módulos adicionales creados en los capítulos anteriores.

El código para estos módulos adicionales se puede encontrar en el repositorio GitHub del libro, en https://github.com/PacktPublishing/Odoo-12-Development-Essentials-Fourth-Edition (https://github.com/PacktPublishing/Odoo-12-Development-Essentials-Fourth-Edition) :

- El `library_app` módulo se puede encontrar en el `ch06/` subdirectorio.
- El `library_member` módulo se puede encontrar en el `ch04/` subdirectorio.

Ambos módulos de complementos deberían estar disponibles en la ruta de complementos que estamos utilizando, para que puedan instalarse y utilizarse.

Sección anterior Sección (/book/business/9781789532470/8)

siguiente (/book/business/9781789532470/8/ch08lvl1sec80/learning-project-the-library-checkout-module)

Proyecto de aprendizaje: el módulo library_checkout

En capítulos anteriores, preparamos al maestroestructuras de datos para la aplicación de la biblioteca. Ahora queremos agregar la posibilidad de que los miembros de la biblioteca tomen prestados libros. Esto significa que podemos realizar un seguimiento de la disponibilidad y las devoluciones de libros.

Cada solicitud de compra de libros tiene un ciclo de vida, desde el momento en que se está redactando, hasta el momento en que se devuelven los libros. Es un flujo de trabajo simple que se puede representar como un tablero Kanban, donde las diversas etapas se presentan como columnas y los elementos de trabajo y las solicitudes de pago fluyen desde la columna de la izquierda a la derecha, hasta que se completan.

En este capítulo, nos centraremos en el modelo de datos y la lógica empresarial para admitir esta función. La interfaz de usuario se analizará en el Capítulo 10 (/book/business/9781789532470/10) , **Vistas de backend: diseño de la interfaz de usuario** , y las vistas de Kanban en el Capítulo 11 (/book/business/9781789532470/11) , **Vistas de Kanban y Cliente - Qweb lateral** .

El modelo de pago de la biblioteca tendrá lo siguiente:

- El miembro de la biblioteca que toma prestados los libros (requerido)
- La fecha de solicitud (el valor predeterminado es hoy)
- El bibliotecario responsable de la solicitud (predeterminado para el usuario actual)
- Líneas de pago, con los libros solicitados (uno o más)

Para respaldar y documentar el ciclo de vida del pago, también tendremos lo siguiente:

- Etapa de la solicitud: borrador, abierto, prestado, devuelto o cancelado
- Fecha de pago, cuando los libros fueron prestados
- Fecha de cierre, cuando los libros fueron devueltos.

Comenzaremos creando el nuevo `library_checkout` módulo e implementando una versión inicial del modelo de pago de la biblioteca. No introduce nada nuevo en comparación con los capítulos anteriores. Proporciona una base para construir las nuevas características discutidas en este capítulo.

Deberíamos crear un nuevo `library_checkout` directorio en el mismo directorio que los otros módulos adicionales del proyecto de biblioteca:

1 Comience agregando el `__manifest__.py` archivo, con este contenido, de la siguiente manera:

Dupdo

```
{ 'name': 'Library Book Borrowing',
  'description': 'Members can borrow books from the library.',
  'author': 'Daniel Reis',
  'depends': ['library_member'],
 'data': [
 'security/ir.model.access.csv',
 'views/library_menu.xml',
 'views/checkout_view.xml',
 ],
}
```

2 Agregue el `__init__.py` archivo en el `module` directorio, con lo siguiente:

Dupdo

```
from . import models
```

3 Agregue el `models/__init__.py` archivo que contiene lo siguiente:

Dupdo

```
from . import library_checkout
```

4 Agregue el archivo de código real `models/library_checkout.py` , como sigue:

Dupdo

```
from odoo import api, exceptions, fields, models


class Checkout(models.Model):
    _name = 'library.checkout'
    _description = 'Checkout Request'
    member_id = fields.Many2one(
        'library.member',
        required=True)
    user_id = fields.Many2one(
        'res.users',
        'Librarian',
        default=lambda s: s.env.uid)
    request_date = fields.Date(
        default=lambda s: fields.Date.today())
    line_ids = fields.One2many(
        'library.checkout.line',
        'checkout_id',
        string='Borrowed Books',)
```

A continuación, debemos agregar los archivos de datos, la regla de acceso, los elementos del menú y algunas vistas básicas, para que el módulo sea mínimamente utilizable.

5 Lo siguientees el `security/ir.model.access.csv` archivo:

Dupdo

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
checkout_user,Checkout User,model_library_checkout,library_app.library_group_user,1,1,1,0
checkout_line_user,Checkout Line User ,model_library_checkout_line,library_app.library_group_user,1,1,1,1
checkout_manager,Checkout Manager ,model_library_checkout,library_app.library_group_manager,1,1,1,1
```

6 Los elementos del menú se implementan a continuación `views/library_menu.xml` :

Dupdo

```
<odoo>
    <act_window id="action_library_checkout"
      name="Checkouts"
      res_model="library.checkout"
      view_mode="tree,form" />

    <menuitem id="menu_library_checkout"
              name="Checkout"
              action="action_library_checkout"
              parent="library_app.library_menu" />
</odoo>
```

7 Las vistas se implementan en lo siguiente `views/checkout_view.xml` :

Dupdo

```
<?xml version="1.0"?>
<odoo>

  <record id="view_tree_checkout" model="ir.ui.view">
    <field name="name">Checkout Tree</field>
    <field name="model">library.checkout</field>
    <field name="arch" type="xml">
        <tree>
            <field name="request_date" />
            <field name="member_id" />
        </tree>
    </field>
  </record>

  <record id="view_form_checkout" model="ir.ui.view">
    <field name="name">Checkout Form</field>
    <field name="model">library.checkout</field>
    <field name="arch" type="xml">
      <form>
```

Ahora podemos instalar el módulo en nuestra base de datos de trabajo de Odoo, y estamos listos para comenzar a agregar más funciones.

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec79/technical-requirements)

siguiente (/book/business/9781789532470/8/ch08lvl1sec81/using-stages-for-document-centered-workflows)

Uso de etapas para flujos de trabajo centrados en documentos

En Odoo, podemos implementar flujos de trabajo centradosen documentos. Lo que llamamos documentos puede ser cosas como pedidos de ventas, tareas de proyectos o solicitantes de recursos humanos. Se espera que todos estos sigan un cierto ciclo de vida desde que se crean hasta que llegan a una conclusión. Se registran en un documento que avanzará a través de una lista de etapas posibles, hasta que se complete.

Si presentamos las etapas como columnas en un tablero, y los documentos como elementos de trabajo en esas columnas, obtenemos un tablero Kanban, que proporciona una vista rápida sobre el trabajo en progreso.

Hay dos enfoques para implementar estos pasos de progreso que generalmente se denominan **estados** y **etapas** .

Los estados se implementan a través de una lista de selección cerrada de opciones predefinidas. Es conveniente cuando se implementan las reglas de negocio, y los modelos y las vistas tienen un soporte especial para el `state` campo, para tener `required` y `invisible` establecer atributos, de acuerdo con el estado actual. Tiene la desventaja de que la lista de estados está predefinida y cerrada, por lo que no puede adaptarse a las necesidades específicas del proceso.

Las etapas se implementan a través de un modelo relacionado, ya que la lista de etapas está abierta para configurarse según las necesidades del proceso actual: es fácil modificar la lista de etapas disponibles, eliminarlas, agregarlas o reordenarlas. Tiene la desventaja de no ser confiable para la automatización de procesos; Dado que la lista de etapas se puede cambiar, las reglas de automatización no pueden depender de identificadores o descripciones de etapas particulares.

Una forma de obtener lo mejor de ambos enfoques es mapear las etapas a los estados. Los documentos están organizados en una etapa configurable y están indirectamente vinculados a un código de estado confiable que puede usarse con confianza para automatizar la lógica de negocios.

Comenzaremos implementando el `library.checkout.stage` modelo en el `library_checkout/models/library_checkout_stage.py` archivo, que se muestra a continuación:

Dupdo

```
from odoo import fields, models


class CheckoutStage(models.Model):
    _name = 'library.checkout.stage'
    _description = 'Checkout Stage'
    _order = 'sequence,name'

    name = fields.Char()
    sequence = fields.Integer(default=10)
    fold = fields.Boolean()
    active = fields.Boolean(default=True)
    state = fields.Selection(
        [('new','New'),
         ('open','Borrowed'),
         ('done','Returned'),
         ('cancel', 'Cancelled')],
        default='new',
    )
```

Aquí podemos ver el `state` campo, permitiendo para mapear cada etapa con uno de los cuatro estados básicos permitidos.

El `sequence` campo es importante. Para configurar el orden, las etapas deben presentarse en el tablero Kanban y en la lista de selección de etapas.

El `fold` tablero Kanban utilizará el campo booleano en algunas columnas plegadas de forma predeterminada, de modo que su contenido no esté disponible de inmediato. Esto generalmente se usa para las etapas `done` y `Cancelled` .

No olvide agregar el nuevo archivo de código al `models/__init__.py` archivo que contiene lo siguiente:

Dupdo

```
from . import library_checkout_stage
from . import library_checkout
```

A continuación, debemos agregar el `stage` campo al modelo de pago de la biblioteca. Edite el `library_checkout/models/library_checkout.py` archivo y, al final de la `Checkout` clase (después del `line_ids` campo), agregue lo siguiente:

Dupdo

```
    # to add in the class Checkout:
    @api.model
    def _default_stage(self):
        Stage = self.env['library.checkout.stage']
        return Stage.search([], limit=1)

     @api.model
     def _group_expand_stage_id(self, stages, domain, order):
         return stages.search([], order=order)

     stage_id = fields.Many2one(
         'library.checkout.stage',
         default=_default_stage,
         group_expand='_group_expand_stage_id')
     state = fields.Selection(related='stage_id.state')
```

`stage_id` es una relación de muchos a uno con el modelo de etapas. También agregamos el `state` campo. Es un `related` campo que simplemente pone a disposición el `state` campo de la etapa en este modelo, para que pueda usarse en las vistas.

El valor predeterminado de la etapa se calcula mediante la `_default_stage()` función auxiliar que devuelve el primer registro en el modelo de etapas. Dado que el modelo de escenario está ordenado correctamente por secuencia, devolverá el que tenga el número de secuencia más bajo.

The `group_expand` parameter overrides the way grouping on the field works. The default, and expected, behavior for grouping operations is to only see the stages that are being used, and the stages with no checkout document won't be shown. But in this case, we would prefer something different; we would like to see all available stages, even if they have no documents. The `_group_expand_stage_id()` helper function returns the list of group's records that the grouping operation should use. In this case, it returns all existing stages, regardless of having library checkouts in that stage or not.

Note

`Changed in Odoo 10` El `group_expand` atributo de campo se introdujo en Odoo 10 pero no se describe en la documentación oficial. Se pueden encontrar ejemplos de uso en el código fuente de Odoo, por ejemplo, en la aplicación Project: https://github.com/odoo/odoo/blob/12.0/addons/project/models/project.py (https://github.com/odoo/odoo/blob/12.0/addons/project/models/project.py) .

Como agregamos un nuevo modelo, también debemos agregar la seguridad de acceso correspondiente al `security/ir.model.access.csv` archivo, que se muestra a continuación:

Dupdo

```
checkout_stage_user,Checkout Stage User,model_library_checkout_stage,library_app.library_group_user,1,0,0,0
checkout_stage_manager,Checkout Stage Manager,model_library_checkout_stage,library_app.library_group_manager,1,1,1,1
```

Necesitamos algunas etapas para trabajar, así que agreguemosalgunos datos predeterminados para el módulo. Crea el `data/library_checkout_stage.xml` archivo con el siguiente código:

Dupdo

```
<odoo noupdate="1">
  <record id="stage_10" model="library.checkout.stage">
      <field name="name">Draft</field>
      <field name="sequence">10</field>
      <field name="state">new</field>
  </record>
  <record id="stage_20" model="library.checkout.stage">
      <field name="name">Borrowed</field>
      <field name="sequence">20</field>
      <field name="state">open</field>
  </record>
  <record id="stage_90" model="library.checkout.stage">
      <field name="name">Completed</field>
      <field name="sequence">90</field>
      <field name="state">done</field>
  </record>
  <record id="stage_95" model="library.checkout.stage">
      <field name="name">Cancelled</field>
      <field name="sequence">95</field>
```

Antes de que esto surta efecto, necesitapara agregar al `library_checkout/__manifest__.py` archivo, de la siguiente manera:

Dupdo

```
  'data': [
    'security/ir.model.access.csv',
    'views/library_menu.xml',
    'views/checkout_view.xml',
    'data/library_checkout_stage.xml',
  ],
```

Sección anterior (/book/business/9781789532470/8/ch08lvl1sec80/learning-project-the-library-checkout-

Sección siguiente (/book/business/9781789532470/8/ch08lvl1sec82/the-orm-method-decorators)

Los decoradores de métodos ORM

En el código de Odoo Python encontradoHasta ahora, podemos ver que los decoradores, como `@api.multi`, se usan con frecuencia en los métodos modelo. Estos son importantes para el ORM y le permiten darle a esos métodos usos específicos.

Repasemos los decoradores ORM que tenemos disponibles y cuándo se deben usar.

Métodos para conjuntos de registros - @ api.multi

La mayoría de las veces, queremos una costumbremétodo para realizar algunas acciones en un conjunto de registros. Para esto, debemos usar `@api.multi`, y en ese caso, el argumento propio será el conjunto de registros para trabajar. La lógica del método generalmente incluirá un `for` ciclo iterando sobre él. Este es seguramente el decorador más utilizado.

> **Nota**
>
> Si no se utiliza ningún decorador en un método modelo, se establecerá de forma predeterminada en `@api.multi behavior`.

Métodos para registros singleton - @ api.one

En algunos casos, el método está preparado para funcionar con un único registro (un singleton). Aquí, podríamos usar el `@api.one` decorador que todavía está disponible, pero anunciadocomo obsoleto desde Odoo 9. Envuelve el método decorado, realiza la `for` iteración del bucle, llama al método decorado con un registro a la vez y luego devuelve una lista con los resultados. Entonces, dentro de un `@api.one` método decorado, `self` se garantiza que sea un singleton.

> **Nota**
>
> El valor de retorno de `@api.one` puede ser complicado; devuelve una lista, no la estructura de datos devuelta por el método real. Por ejemplo, si el código del método regresa `dict`, el valor de retorno real es una lista de valores dict. Este comportamiento engañoso fue la razón principal por la cual el método fue desaprobado.

En el caso de que se espere que nuestro método funcione en un registro singleton, aún deberíamos usar `@api.multi`, y agregar en la parte superior del código del método una línea `self.ensure_one()`, para asegurar que sea un singleton como se esperaba.

Métodos estáticos de clase - @ api.model

En algunos casos, se espera el métodopara trabajar a nivel de clase y no en registros particulares. En algunos lenguajes orientados a objetos, esto se llamaría un método estático. Estos métodos estáticos de nivel de clase se deben decorar con `@api.model`. En estos casos, `self` debe usarse como referencia para el modelo, sin esperar que contenga registros reales.

> **Nota**

Los métodos decorados con `@api.model` no se pueden usar con botones de interfaz de usuario. En esos casos, `@api.multi` debe usarse en su lugar.

Sobre los métodos de cambio

Los `on change` métodos se activan desdevistas de formulario de interfaz de usuario, cuando el usuario edita un valor en un campo en particular para realizar alguna lógica comercial de inmediato. Esto puede llevar a cabo alguna validación, mostrar un mensaje al usuario o cambiar otros campos en el formulario. El método modelo que respalda esta lógica debe estar decorado con `@api.onchange('fld1', 'fld2', ...)` . Los argumentos del decorador son los nombres de los campos que activarán el método cuando el usuario los edite.

Nota

Una forma particular deshabilita el comportamiento de cambio al agregar el atributo de campo `on_change="0"` , por ejemplo `<field name="fld1" onchange="0" />` „.

Dentro del método, el argumento propio es un registro virtual con los datos del formulario actual. Si los valores se establecen en el registro, estos se cambiarán en el formulario de la interfaz de usuario. Tenga en cuenta que en realidad no escribe en los registros de la base de datos, sino que proporciona información para cambiar los datos en el formulario de la interfaz de usuario.

No se necesita ningún valor de retorno, pero se puede devolver una estructura dict con un mensaje de advertencia para mostrar en la interfaz de usuario.

Como ejemplo, podemos usar esto para realizar cierta automatización en el formulario de Pago: cuando se cambia el miembro de la biblioteca, la fecha de solicitud se establece en hoy y se muestra un mensaje de advertencia al usuario que lo alerta.

Para esto, edite el `library_checkout/models/library_checkout.py` archivo para agregar el siguiente método:

Dupdo

```
@api.onchange('member_id')
def onchange_member_id(self):
    today = fields.Date.today()
    if self.request_date != today:
        self.request_date = fields.Date.today()
        return {
            'warning': {
                'title': 'Changed Request Date',
                'message': 'Request date changed to today.',
            }
        }
```

Aquí, estamos usando el `@api.onchange` decorador para activar algo de lógica para cambiar en el `member_id` campo, cuando se hace a través de la interfaz de usuario. El nombre real del método no es relevante, pero la convención es para comenzar con su nombre `onchange_` . Actualizamos el valor del `request_date` campo y devolvemos un mensaje de advertencia.

Dentro de un `onchange` método, `self` representa un único registro virtual que contiene todos los campos establecidos actualmente en el registro que se está editando, y podemos interactuar con ellos. La mayoría de las veces, esto es lo que queremos hacer para completar automáticamente los valores en otros campos, dependiendo del valor establecido en el campo modificado. En este caso, estamos actualizando el `request_date` campo hasta hoy.

Los `onchange` métodos no necesitan devolver nada, pero pueden devolver un diccionario que contiene una advertencia o una clave de dominio, como se muestra a continuación:

La clave de advertencia debe describir un mensaje.para mostrar en una ventana de diálogo, tales como: `{'title': 'Message Title', 'message': 'Message Body'}` .

- La clave de dominio puede establecer o cambiar el atributo de dominio de otros campos. Esto le permite construir interfaces más fáciles de usar, ya que tener un campo para muchos solo pone a disposición las opciones que tienen sentido en ese momento. El valor de la clave de dominio se ve así: `{'user_id': [('email', '!=', False)]}`

Otros decoradores de métodos modelo

Los siguientes decoradores también son frecuentementeusado. Están relacionados con el comportamiento interno de un modelo y se analizan en detalle en el Capítulo 6 (/book/business/9781789532470/6)**, Modelos: estructuración de los datos de la aplicación.**

Los mencionamos aquí para su referencia:

- `@api.depends(fld1,...)` se utiliza para las funciones de campo calculadas para identificar en qué cambios se debe activar el (re) cálculo. Debe establecer valores en los campos calculados, de lo contrario, se producirá un error.
- `@api.constrains(fld1,...)` se utiliza para las funciones de validación del modelo y realiza comprobaciones para cuando se cambia cualquiera de los campos mencionados. No debe escribir cambios en los datos. Si las verificaciones fallan, se debe hacer una excepción.

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec81/using-stages-for-document-centered-v

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec83/using-the-orm-built-in-methods)

Usando los métodos integrados de ORM

Los decoradores discutidos en la sección anterior . nos permite agregar ciertas características a nuestros modelos, como implementar validaciones y cálculos automáticos.

El ORM proporciona métodos para realizar **operaciones de Crear, Leer, Actualizar y Eliminar** (**CRUD**) en nuestroDatos del modelo Ahora exploraremos estas operaciones de escritura y cómo se pueden extender para admitir la lógica personalizada.

Para leer datos, los principales métodos proporcionados son `search()` y `browse()` serán discutidos en el Capítulo 7 (/book/business/9781789532470/7) , **Conjuntos de registros: trabajo con datos del modelo** .

Métodos para escribir datos del modelo

El ORM proporciona tres métodos para los tres. operaciones básicas de escritura, que se muestran a continuación:

- `<Model>.create(values)` crea un nuevo registro en el modelo. Devuelve el registro creado.
- `<Recordset>.write(values)` actualiza los valores de campo en el conjunto de registros. No devuelve nada.
- `<Recordset>.unlink()` elimina los registros de la base de datos. No devuelve nada.

El `values` argumento es un diccionario que asigna nombres de campos a valores para escribir. Los métodos están decorados con `@api.multi` , excepto el `create()` método, que está decorado con `@api.model` .

Nota

`Changed in Odoo 12` El `create()` método ahora también permite crear registros en lotes. Esto se hace pasando como argumento una lista de objetos de diccionario, en lugar de un solo objeto de diccionario. Esto es compatible con los `create()` métodos que tienen el `@api.model_create_multi` decorador.

En algunos casos, necesitamos extender estos métodos para agregar cierta lógica de negocios que se activará cada vez que se ejecuten estas acciones. Al colocar nuestra lógica en la sección apropiada del método personalizado, podemos ejecutar el código antes o después de que se ejecuten las operaciones principales.

Crearemos un ejemplo utilizando la `Checkout` clase de modelo: agregue dos campos de fecha para registrar cuándo el pago ingresó en un estado abierto y cuándo se movió a un estado cerrado. Esto es algo que no se puede hacer con un campo calculado. También agregaremos un cheque para evitar crear pagos ya realizados.

Entonces, debemos comenzar agregando los dos nuevos campos a la clase de pago, en el siguiente `library_checkout/models/library_checkout.py` archivo:

Dupdo

```
# add to the Checkout Model class:
checkout_date = fields.Date(readonly=True)
close_date = fields.Date(readonly=True)
```

Ahora podemos hacer un `create()` método personalizado , para establecer `checkout_date` , si está en el estado apropiado, y evitar la creación de pagos en el estado hecho, como se muestra a continuación:

Dupdo

```
    @api.model
    def create(self, vals):
        # Code before create: should use the `vals` dict
        if 'stage_id' in vals:
            Stage = self.env['library.checkout.stage']
            new_state = Stage.browse(vals['stage_id']).state
            if new_state == 'open':
                vals['checkout_date'] = fields.Date.today()
        new_record = super().create(vals)
        # Code after create: can use the `new_record` created
        if new_record.state == 'done':
            raise exceptions.UserError(
                'Not allowed to create a checkout in the done state.')
        return new_record
```

Tenga en cuenta que, antes de que el nuevo registro real, no tenemos realregistro disponible, solo el diccionario con los valores a utilizar para la creación del registro. Es por eso que solíamos `browse()` obtener un registro para el nuevo registro `stage_id` , para luego inspeccionar el valor de estado correspondiente. En comparación, una vez que se crea el nuevo registro, la operación se hace más simple equivalente, usando la notación punto objeto, `new_record.state` .

El diccionario de valores se puede cambiar antes de la `super().create(vals)` instrucción, y lo usamos para configurar la `checkout_date` escritura, cuando sea apropiado.

> Nota
>
> **`Changed in Odoo 11`** Cuando se usa Python 3, hay disponible una forma simplificada de usar `super()` y se usó en el ejemplo de código anterior. En Python 2, escribiríamos `super(Checkout, self).create(vals)` , donde la salida es el nombre de la clase de Python estamos en la sintaxis Esto sigue siendo válido para Python 3, pero también tenemos disponible la nueva sintaxis simplificada.: `super().create(vals)` .

Y cuando se modifica un registro, queremos actualizar `checkout_date` y `close_date` si el pago ingresa a los estados apropiados. Para esto tendremos un `write()` método personalizado , que se muestra a continuación:

Dupdo

```
@api.multi
    def write(self, vals):
        # Code before write: can use `self`, with the old values
        if 'stage_id' in vals:
            Stage = self.env['library.checkout.stage']
            new_state = Stage.browse(vals['stage_id']).state
            if new_state == 'open' and self.state != 'open':
                vals['checkout_date'] = fields.Date.today()
            if new_state == 'done' and self.state != 'done':
                vals['close_date'] = fields.Date.today()
        super().write(vals)
 # Code after write: can use `self`, with the updated values
  return True
```

Siempre que sea posible, preferimos cambiar los valores para escribir antes de la `super().write(vals)` instrucción. Para el `write()` método, tener más operaciones de escritura en el mismo modelo conducirá a un ciclo de recursión y terminará con un error cuando se agoten los recursos del proceso de trabajo. Considere si esto es realmente necesario. Si es así, una técnica para evitar el ciclo de recursión es establecer una bandera en el contexto. Por ejemplo, podríamos agregar código como el siguiente:

Dupdo

```
        if not self.env.context.get('_library_checkout_writing'):
            self.with_context(_library_checkout_writing=True).write(
                some_values)
```

Con esta técnica, nuestra lógica específica está protegida por una `if` declaración y se ejecuta solo si no se encuentra un marcador específico en el contexto. Además, nuestras `self.write()` operaciones deberían usarse `with_context` para establecer ese marcador. Esta combinación garantiza que el inicio de sesión personalizado dentro de la `if` instrucción se ejecute solo una vez y no se active en futuras `write()` llamadas, evitando el bucle infinito.

Hacerlo después implica ejecutar `write()` dentro del `write()` método, y esto puede crear un bucle infinito. Para evitar eso, necesitamos establecer un valor de marcador en el contexto y verificarlo en el código para evitar encontrarse en un bucle.

Considere cuidadosamente si realmente necesita usar extensiones para los métodos de creación o escritura. En la mayoría de los casos, solo necesitamos realizar alguna validación, o calcular automáticamente algún valor, cuando se guarda el registro:

- Para los valores de campo que se calculan automáticamente en función de otros campos, debemos usar campos calculados. Un ejemplo de esto es calcular un total de encabezado cuando se cambian los valores de las líneas.
- Para que los valores predeterminados de campo se calculen dinámicamente, podemos usar un valor predeterminado de campo vinculado a una función en lugar de un valor fijo.
- Para establecer valores en otros campos cuando se cambia un campo, podemos usar las funciones de cambio. Un ejemplo de esto es, al elegir un cliente, establecer su moneda como la moneda del documento que luego el usuario puede cambiar manualmente. Tenga en cuenta que en el cambio solo funciona en la interacción de vista de formulario y no en llamadas de escritura directa.
- Para las validaciones, debemos usar la restricciónfunciones decoradas con `@api.constraints(fld1,fld2,...)` . Estos son como campos calculados pero, en lugar de calcular valores, se espera que generen errores.

Métodos para la importación y exportación de datos.

La importación y exportaciónlas operaciones, discutidas en el Capítulo 5 (/book/business/9781789532470/5) , **Importar, Exportar, y Datos del Módulo** , también están disponibles desde la API ORM, a través de los siguientes métodos:

- `load([fields], [data])` se usa para importar datos adquiridos de un archivo CSV. El primer argumento es la lista de campos para importar, y se asigna directamente a una fila superior de CSV. El segundo argumento es una lista de registros, donde cada registro es una lista de valores de cadena para analizar e importar, y se asigna directamente a las filas y columnas de datos CSV. Implementa las características de importación de datos CSV, como el soporte de identificadores externos. Lo utiliza la `Import` función de cliente web .
- `export_data([fields], raw_data=False)` es utilizado por la `Export` función de cliente web . Devuelve un diccionario con una clave de datos que contiene los datos: una lista de filas. Los nombres de campo pueden usar los sufijos `.id` y `/id` utilizados en los archivos CSV, y los datos están en un formato compatible con un archivo CSV importable. El `raw_data` argumento opcional permite que los valores de datos se exporten con sus tipos de Python, en lugar de la representación de cadena utilizada en CSV.

Métodos para soportar la interfaz de usuario

Los siguientes métodos son principalmente utilizado por el cliente web para representar la interfaz de usuario y realizar una interacción básica:

- `name_get()` devuelve una lista de tuplas (ID, nombre) con el texto que representa cada registro. Se usa por defecto para calcular el `display_name` valor, proporcionando la representación de texto de los campos de relación. Se puede ampliar para implementar representaciones de visualización personalizadas, como mostrar el código de registro y el nombre en lugar de solo el nombre.
- `name_search(name='', args=None, operator='ilike', limit=100)` devuelve una lista de tuplas (ID, nombre), donde el nombre para mostrar coincide con el texto en el argumento de nombre. Se utiliza en la interfaz de usuario al escribir en un `related` campo para generar la lista con los registros sugeridos que coinciden con el texto escrito. Por ejemplo, se usa para implementar la búsqueda de productos tanto por nombre como por referencia, mientras se escribe en un campo para elegir un producto.
- `name_create(name)` crea un nuevo registro con solo el nombre del título para usarlo. Se usa en la interfaz de usuario para la función de **creación rápida** , donde puede crear rápidamente un registro relacionado con solo proporcionar su nombre. Se puede ampliar

para proporcionar valores predeterminados específicos para los nuevos registros creados a través de esta función.

- `default_get([fields])` devuelve un diccionario con los valores predeterminados para crear un nuevo registro. Los valores predeterminados pueden depender de variables, como el usuario actual o el contexto de la sesión.
- `fields_get()` se usa para describir las definiciones de campo del modelo, como se ve en la `view` opción de campos del menú del desarrollador.
- `fields_view_get()` es utilizado por el cliente web para recuperar la estructura de la vista de la interfaz de usuario para renderizar. Se le puede dar el ID de la vista como argumento, o el tipo de vista que queremos usar `view_type='form'` . Por ejemplo, es posible que intente esto: `self.fields_view_get(view_type='tree')` .

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec82/the-orm-method-decorators)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec84/the-message-and-activity-features)

El mensaje y las características de la actividad.

Odoo tiene mensajes globales disponiblesy características de planificación de actividades, proporcionadas por el debate aplicación, con nombre técnico de correo.

El módulo de correo proporciona la `mail.thread` clase abstracta que simplifica la adición de funciones de mensajería a cualquier modelo, y la `mail.activity.mixin` que agrega funciones de actividad planificadas. Esto se realizó en el Capítulo 4 (/book/business/9781789532470/4) , **Módulos de extensión** , para explicar cómo heredar características de las clases abstractas mixin.

Para agregar estas características, necesitamos agregar la dependencia de correo al módulo adicional `library_checkout` y luego hacer que la clase de modelo de pago de la biblioteca herede de las clases abstractas que proporcionan las siguientes características.

Edite la `'depends'` clave en el `library_checkout/__manifest__.py` archivo para agregar el módulo de correo, como se muestra a continuación:

Dupdo

```
  'depends': ['library_member', 'mail'],
```

Y edite el `library_checkout/models/library_checkout.py` archivo para heredar de los modelos abstractos mixin, como se muestra a continuación:

Dupdo

```
class Checkout(models.Model):
    _name = 'library.checkout'
    _description = 'Checkout Request'
    _inherit = [''mail.thread', 'mail.activity']
```

Después de esto, entre otras cosas, nuestro modelo tendrá tres nuevos campos disponibles. Para cada registro (a veces también llamado documento), tenemos lo siguiente:

- `mail_follower_ids` almacena los seguidores y las preferencias de notificación correspondientes
- `mail_message_ids` enumera todo lo relacionado `messages.activity_ids` con todas las actividades planificadas relacionadas

Los seguidores pueden ser socios o canales. Un **socio** representa a una persona u organización específica. Un **canal** no es una persona en particular y, en cambio, representa una lista de suscripción.

Cada seguidor también tiene una lista de tipos de mensajes a los que están suscritos. Solo los tipos de mensajes seleccionados generarán notificaciones para ellos.

Subtipos de mensajes

Algunos tipos de mensajesse llaman **subtipos** . Se almacenan en el `mail.message.subtype` modelo y son accesibles.en el **`Technical`** | **`Email`** El | **`Subtypes`** menú.

Por defecto, tenemos los siguientes tres subtipos de mensajes disponibles:

Debates , con `mail.mt_comment XMLID` , utilizados para los mensajes creados con el **`Send message`** enlace. Está destinado a enviar una notificación.

- **Actividades**, con `mail.mt_activities XMLID`, utilizadas para los mensajes creados con el **`Schedule activity`** enlace. No está destinado a enviar una notificación.
- **Nota**, con `mail.mt_note XMLID`, usado para los mensajes creados con el **`Log note`** enlace. No está destinado a enviar una notificación.

Los subtipos tienen la configuración de notificación predeterminada descrita anteriormente, pero los usuarios pueden cambiarla para documentos específicos, por ejemplo, para silenciar una discusión que no les interesa.

Además de los subtipos incorporados, también podemos agregar nuestros propios subtipos para personalizar las notificaciones para nuestras aplicaciones. Los subtipos pueden ser genéricos o destinados a un modelo en particular. Para el último caso, debemos completar el `subtype's res_model` campo con el nombre del modelo al que debe aplicarse.

Publicar mensajes

Nuestra lógica de negocios puede hacer uso de este mensajesistema para enviar notificaciones a los usuarios. Para publicar un mensaje, utilizamos el `message_post()` método. El siguiente es un ejemplo de esto:

Dupdo

```
self.message_post('Hello!')
```

Esto agrega un mensaje de texto simple, pero no envía ninguna notificación a los seguidores. Esto se debe a que, de manera predeterminada, el `mail.mt_note` subtipo se usa para los mensajes publicados. Pero podemos publicar el mensaje con el subtipo particular que queremos. Para agregar un mensaje y hacer que envíe notificaciones a los seguidores, debemos usar el `mt_comment` subtipo. Otro atributo opcional es el tema del mensaje. Aquí, hay un ejemplo usando ambas opciones:

Dupdo

```
self.message_post('Hello again!', subject='Hello', subtype='mail.mt_comment')
```

El cuerpo del mensaje es HTML, por lo que podemos incluir marcas para efectos de texto, como `<b>` texto en negrita o `<i>` cursiva.

Nota

El cuerpo del mensaje se desinfectará por razones de seguridad, por lo que es posible que algunos elementos HTML particulares no lleguen al mensaje final.

Agregar seguidores

También interesante de un negocioEl punto de vista lógico es la capacidad de agregar seguidores automáticamente a un documento, para que luego puedan obtener las notificaciones correspondientes. Para esto, tenemos varios métodos disponibles para agregar seguidores, que se enumeran a continuación:

- `message_subscribe(partner_ids=<list of int IDs>)` agrega socios
- `message_subscribe(channel_ids=<list of int IDs>)` agrega canales
- `message_subscribe_users(user_ids=<list of int IDs>)` agrega usuarios

Los subtipos predeterminados se aplicarán a cada suscriptor. Para forzar la suscripción de una lista específica de subtipos, agregue un atributo adicional `subtype_ids=<list of int IDs>`, enumerando los subtipos específicos para habilitar la suscripción.

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec83/using-the-orm-built-in-methods)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec85/creating-a-wizard)

Crear un asistente

Supongamos que los usuarios de nuestra biblioteca tienen la necesidad de enviar mensajes.a grupos de prestatarios. Por ejemplo, podrían seleccionar las cajas más antiguas con un libro prestado y enviarles a todos un mensaje solicitando la devolución de los libros. Esto se puede implementar a través de un asistente. **Los asistentes** son formularios que obtienen información de los usuarios y luego la utilizan para su posterior procesamiento.

Nuestros usuarios comenzarán seleccionando de la lista de cajas los registros a considerar, luego seleccionarán la **`wizard's`** opción en el menú superior de la vista. Esto abrirá el formulario del asistente, donde podemos escribir el asunto y el cuerpo del mensaje. Una vez que hagamos clic en **`Send`** , este mensaje se publicará en todos los pagos seleccionados.

El modelo mago

Un asistente muestra una vista de formulario para el usuario, generalmente como una ventana de diálogo, con algunos campos para completar. Estos se utilizarán para la lógica del asistente.

Esto se implementa utilizando la misma arquitectura de modelo / vista que para las vistas normales, pero el modelo de soporte se basa en `models.TransientModel` lugar de `models.Model` . Este tipo de modelo también tiene una representación de base de datos y almacena el estado allí, pero se espera que los datos sean útiles solo hasta que el asistente complete su trabajo. Un trabajo programado limpia regularmente los datos antiguos de las tablas de la base de datos del asistente.

El `wizard/wizard/library_checkout_massmessage.py` archivo definirá los campos que necesitamos para interactuar con el usuario: la lista de pagos a notificar, el asunto del mensaje y el cuerpo del mensaje.

Primero, edite `library_checkout/__init__.py` para agregar una importación del código en el `wizard/` subdirectorio, como se muestra a continuación:

Dupdo

```
from . import models
from . import wizard
```

Agregue el `wizard/__init__.py` archivo con la siguiente línea de código:

Dupdo

```
from . import library_checkout_massmessage
```

Luego, cree el `wizard/checkout_mass_message.py` archivo real , como se muestra a continuación:

Dupdo

```
from odoo import api, exceptions, fields, models

class CheckoutMassMessage(models.TransientModel):
    _name = 'library.checkout.massmessage'
    _description = 'Send Message to Borrowers'
    checkout_ids = fields.Many2many(
        'library.checkout',
        string='Checkouts')
    message_subject = fields.Char()
    message_body = fields.Html()
```

Vale la pena señalar que las relaciones uno a muchos con modelos regulares no deberían usarse en modelos transitorios. La razón de esto es que requeriría que el modelo regular tenga la relación inversa de muchos a uno con el modelo transitorio. Pero esto no está permitido, ya que eso podría evitar que los registros transitorios antiguos se limpien, debido a las referencias existentes en los registros del modelo regular. La alternativa a esto es usar una relación de muchos a muchos.

> **Nota**
>
> Las relaciones de muchos a muchos se almacenan en una tabla dedicada, y las filas de esta tabla se eliminan automáticamente cuando se elimina cualquier lado de la relación.

Los modelos transitorios no necesitan reglas de acceso, ya queson solo registros desechables para ayudar a ejecutar un proceso. Por lo tanto, no necesitamos agregar ACL al `security/ir.model.access.csv` archivo.

La forma del mago

Las vistas de formulario del asistente son las mismas. en cuanto a los modelos regulares, excepto por dos elementos específicos:

- Se `<footer>` puede usar una sección para reemplazar los botones de acción
- Hay un `special="cancel"` botón disponible para interrumpir al asistente sin realizar ninguna acción.

El siguiente es el contenido de nuestro `wizard/checkout_mass_message_wizard_view.xml` archivo:

Dupdo

```
<?xml version="1.0"?>
<odoo>

  <record id="view_form_checkout_message" model="ir.ui.view">
    <field name="name">Library Checkout Mass Message Wizard</field>
    <field name="model">library.checkout.massmessage</field>
    <field name="arch" type="xml">

      <form>
        <group>
          <field name="message_subject" />
          <field name="message_body" />
          <field name="checkout_ids" />
        </group>
        <footer>
          <button type="object"
            name="button_send"
            string="Send Messages" />
          <button special="cancel"
```

La `<act_window>` acción de la ventana que vemos en el XML agrega una opción al **`Action`** botón de pago de la biblioteca, utilizando el `src_model` atributo El `target="new"` atributo lo abre como una ventana de diálogo.

Para abrir el asistente, debemos seleccionar uno o más registros de la lista de pago y luego elegir la **`Send Messages`** opción del **`Action`** menú, que aparece en la parte superior de la lista, al lado del **`Filters`** menú.

En este momento, esto abre el formulario del asistente, pero los registros seleccionados en la lista se ignoran. Sería bueno para nuestro asistente mostrarlos preseleccionados en la lista de tareas. Cuando se presenta un formulario, `default_get()` se llama al método para calcular los valores predeterminados que se presentarán. Esto es exactamente lo que queremos, por lo que debemos usar ese método. Tenga en cuenta que, cuando se abre el formulario del asistente, tenemos un registro vacío y el `create()` método aún no se invoca, lo que solo sucederá cuando presionamos un botón, por lo que no se puede usar para lo que queremos.

Las vistas de Odoo agregan algunos elementos al diccionario de contexto, que están disponibles cuando hacemos clic en una acción o saltamos a otra vista. Estos son los siguientes:

- `active_model` , con el nombre técnico del modelo de la vista

- `active_id` , con el ID del registro activo del formulario (o el primer registro, si está en una lista)
- `active_ids` , con una lista de los registros activos en una lista (solo un elemento, si está en un formulario)
- `active_domain` , si la acción se desencadena desde una vista de formulario

En nuestro caso, `active_ids` mantenga las ID de los registros.seleccionado en la **`Task`** lista, y podemos usarlos como los valores predeterminados para el `task_ids` campo del asistente , como se muestra a continuación:

Dupdo

```
@api.model
def default_get(self, field_names):
    defaults = super().default_get(field_names)
    checkout_ids = self.env.context['active_ids']
    defaults['checkout_ids'] = checkout_ids
    return defaults
```

Primero usamos `super()` para llamar al `default_get()` cómputo estándar , y luego agregamos el `checkout__id` a los valores predeterminados, con el `active_ids` valor leído del contexto del entorno.

A continuación, debemos implementar las acciones que se realizarán en el **`Send`** botón del formulario .

El asistente de lógica de negocios

Excluyendo el **`Cancel`** botón, que simplemente cierra el formulario sinAl realizar cualquier acción, tenemos los **`Send`** botones de acción para implementar.

El método llamado por el botón es `button_send` , y debe definirse en el `wizard/checkout_mass_message.py` archivo, como se muestra en el siguiente código:

Dupdo

```
@api.multi
def button_send(self):
    self.ensure_one()
    for checkout in self.checkout_ids:
        checkout.message_post(
            body=self.message_body,
            subject=self.message_subject,
            subtype='mail.mt_comment',
        )
    return True
```

Nuestro código solo necesita manejar una instancia de asistente a la vez, por lo que solíamos `self.ensure_one()` dejar eso claro. Aquí, `self` representa el registro de exploración de los datos en el formulario del asistente.

El método recorre los registros de pago seleccionados y publica un mensaje para cada uno. Se `mt_comment` utiliza el subtipo, de modo que el mensaje genera una notificación a los seguidores del registro.

Nota

Es una buena práctica que los métodos siempre devuelvan algo, al menos el `True` valor. La única razón para esto es que algunos protocolos XML-RPC no admiten `None` valores, por lo que esos métodos no serán utilizables con ese protocolo. En la práctica, es posible que no conozca el problema porque el cliente web utiliza JSON-RPC, no XML-RPC, pero sigue siendo una buena práctica a seguir.

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec84/the-message-and-activity-features)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec86/using-log-messages)

Usar mensajes de registro

Escribir mensajes en el archivo de registro puede ser útilpara monitorear y auditar sistemas en ejecución. También es útil para el mantenimiento del código, ya que facilita la obtención de información de depuración de los procesos en ejecución, sin la necesidad de cambiar el código.

Para que nuestro código pueda usar el registro, primero debemos preparar un registrador. Agregue las siguientes líneas de código en la parte superior de `library_checkout/wizard/checkout_mass_message.py` :

Dupdo

```
import logging
_logger = logging.getLogger(__name__)
```

Se `logging` utiliza el módulo de biblioteca estándar de Python . El `_logger` objeto es inicializado, usando el nombre del archivo de código actual, `__name__` . Con esto, los mensajes de registro llevarán información del archivo que los generó.

Hay varios niveles disponibles para los mensajes de registro. Estos son los siguientes:

Dupdo

```
_logger.debug('A DEBUG message')
_logger.info('An INFO message')
_logger.warning('A WARNING message')
_logger.error('An ERROR message')
```

Ahora podemos usar el registrador para escribir mensajes en el registro. Hagamos esto en el `button_send` método del asistente. Agregue las siguientes instrucciones antes del final `return True` :

Dupdo

```
        _logger.info(
            'Posted %d messages to Checkouts: %s',
            len(self.checkout_ids),
            str(self.checkout_ids),
        )
```

Con esto, cuando use el asistente para enviar mensajes, se mostrará un mensaje similar a este en el registro del servidor:

Dupdo

```
INFO 12-library odoo.addons.library_checkout.wizard.checkout_mass_message: Posted 2 messages to Checkouts: [3, 4]
```

Tenga en cuenta que no utilizamos la interpolación de cadenas de Python para el mensaje de registro. En lugar de algo así , usamos algo así . No usar interpolación significa una tarea menos para que nuestro código realice y hace que el registro sea más eficiente. Por lo tanto, siempre debemos proporcionar las variables como parámetros de registro adicionales. `_logger.info('Hello %s' % 'World')` `_logger.info('Hello %s', 'World')`

> **Nota**
>
> Las marcas de tiempo de los mensajes de registro del servidor siempre usan la hora UTC. Por lo tanto, los mensajes de registro también se imprimen en hora UTC. Esto puede ser una sorpresa y proviene del hecho de que el servidor Odoo maneja internamente todas las fechas en UTC.

Para mensajes de registro de nivel de depuración, utilizamos `_logger.debug()`. Por ejemplo, agregue el siguiente mensaje de registro de depuración justo después de la `checkout.message_post()` instrucción:

Dupdo

```
            _logger.debug(
                'Message on %d to followers: %s',
                checkout.id,
                checkout.message_follower_ids)
```

Esto no mostrará nada en el registro del servidor, ya que elNivel de registro predeterminado es `INFO`. El nivel de registro debe establecerse como DEPURACIÓN para que los `debug` mensajes se impriman en el registro.

La opción de comando Odoo `--log-level=debug`, establece el nivel de registro general. También podemos establecer el nivel de registro para módulos particulares. El módulo Python para nuestro asistente es `odoo.addons.library_checkout.wizard.checkout_mass_message`, como se ve en el `INFO` mensaje de registro. Para habilitar solo los mensajes de depuración del asistente, utilizamos la `--log-handler` opción, que se puede repetir varias veces, hasta el nivel de registro de varios módulos, como se muestra a continuación:

Dupdo

```
--log-handler=odoo.addons.library_checkout.wizard.checkout_mass_message:DEBUG
```

La referencia completa a las opciones de registro del servidor Odoo se puede encontrar en la documentación oficial, en https://www.odoo.com/documentation/12.0/reference/cmdline.html#logging (https://www.odoo.com/documentation/12.0/reference/cmdline.html#logging) .

Si desea conocer los detalles esenciales del registro de Python, la documentación oficial es un buen lugar para comenzar: https://docs.python.org/3.5/howto/logging.html (https://docs.python.org/3.5/howto/logging.html) .

‹ Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec85/creating-a-wizard)

siguiente › (/book/business/9781789532470/8/ch08lvl1sec87/raising-exceptions)

Levantando excepciones

Cuando algo no funciona como se esperaba, es posible que deseemosinformar al usuario e interrumpir el programa con un mensaje de error. Esto se hace planteando una excepción. Odoo proporciona clases de excepción que deberíamos usar para esto.

Las excepciones de Odoo más útiles para usar en módulos adicionales son las siguientes:

Dupdo

```
from odoo import exceptions
raise exceptions.ValidationError('Not valid message')
raise exceptions.UserError('Business logic error')
```

La `ValidationError` excepción debe usarse para validaciones en código Python, como `@api.constrains` métodos decorados.

La `UserError` excepción se debe utilizar en todos los demás casos en los que no se debe permitir alguna acción, ya que va en contra de la lógica empresarial.

> Nota
>
> **`Changed in Odoo 9`** La `UserError` excepción se introdujo, reemplazando a la `Warning` excepción, en desuso debido a la colisión con Python incorporado, pero se mantuvo por compatibilidad con versiones anteriores de Odoo.

Como regla general, toda la manipulación de datos realizada durante la ejecución de un método se realiza en una transacción de base de datos y se revierte cuando ocurre una excepción. Esto significa que, cuando se genera una excepción, todos los cambios de datos anteriores se cancelan.

Veamos un ejemplo usando nuestro `button_send` método de asistente . Si lo pensamos bien, no tiene ningún sentido ejecutar la lógica de envío de mensajes si no se seleccionó ningún documento de pago. Y no tiene sentido enviar mensajes sin cuerpo del mensaje. Avisemos al usuario si ocurre alguna de estas cosas.

Edite el `button_send()` método para agregar lo siguienteinstrucción justo después de la `self.ensure_one()` línea:

Dupdo

```
        if not self.checkout_ids:
            raise exceptions.UserError(
                'Select at least one Checkout to send messages to.')
        if not self.message_body:
            raise exceptions.UserError(
                'Write a message body to send.')
```

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec86/using-log-messages)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec88/unit-tests)

Pruebas unitarias

Las pruebas automatizadas son generalmente aceptadascomo una mejor práctica en software. No solo nos ayudan a garantizar que nuestro código se implemente correctamente, sino que, lo que es más importante, proporcionan una red de seguridad para futuros cambios o reescrituras de código.

En el caso de los lenguajes de programación dinámica, como Python, dado que no hay un paso de compilación, los errores de sintaxis pueden pasar desapercibidos. Esto hace que sea aún más importante hacer que las pruebas unitarias pasen por tantas líneas de código como sea posible.

Los dos objetivos descritos pueden proporcionar una luz de guía al escribir pruebas. El primer objetivo para sus pruebas debe ser proporcionar una buena cobertura de prueba; diseñando casos de prueba que atraviesan todas sus líneas de código. Por lo general, esto solo hará un buen progreso en el segundo objetivo, mostrar la corrección funcional del código, ya que después de esto, seguramente tendremos un gran punto de partida para construir casos de prueba adicionales para casos de uso no obvios.

> Nota
>
> **`Changed in Odoo 12`** En versiones anteriores, Odoo también soportaba pruebas descritas usando archivos de datos YAML. Desde Odoo 12, el motor de archivos de datos YAML se eliminó de Odoo, y este tipo de archivo ya no es compatible. La última documentación está disponible en https://doc.odoo.com/v6.0/contribute/15_guidelines/coding_guidelines_testing/ (https://doc.odoo.com/v6.0/contribute/15_guidelines/coding_guidelines_testing/) .

Agregar pruebas unitarias

Las pruebas de Python se agregan a los módulos adicionalesen un `tests/` subdirectorio El corredor de pruebas descubrirá automáticamente las pruebas en los subdirectorios con este nombre en particular.

Para agregar pruebas para la lógica del asistente creada en el `library_checkout` módulo adicional, podemos comenzar creando el `tests/test_checkout_mass_message.py` archivo. Como de costumbre, también necesitaremos agregar el `tests/__init__.py` archivo, de la siguiente manera:

Dupdo

```
from . import test_checkout_mass_message
```

El siguiente sería el esqueleto básico para el `tests/test_checkout_mass_message.py` código de prueba:

Dupdo

```
from odoo.tests.common import TransactionCase

class TestWizard(TransactionCase):

    def setUp(self, *args, **kwargs):
        super(TestWizard, self).setUp(*args, **kwargs)
        # Add test setup code here...

    def test_button_send(self):
        """Send button should create messages on Checkouts"""
        # Add test code
```

Odoo proporciona algunas clases para usar en las pruebas, que se enumeran a continuación:

- La `TransactionCase` prueba utiliza una transacción diferente para cada prueba, que se revierte automáticamente al final.
- El `SingleTransactionCase` , que ejecuta todas las pruebas en una sola transacción, se revierte solo al final de la última prueba. Esto puede ser útil cuando desea que el estado final de cada prueba sea el estado inicial de la siguiente prueba.

El `setUp()` método es donde preparamos los datos y las variables que se utilizarán. Por lo general, los almacenaremos como atributos de clase, de modo que estén disponibles para usarse en los métodos de prueba.

Las pruebas deben implementarse como métodos de clase, como `test_button_send()` . Los nombres de los métodos de casos de prueba deben comenzar con un `test_` prefijo. Se descubren automáticamente, y este prefijo es lo que identifica los métodos que implementan casos de prueba. Los métodos se ejecutarán en el orden de los nombres de las funciones de prueba.

Al usar la `TransactionCase` clase, se realizará una reversión al final de cada caso de prueba. El método `docstring` se muestra cuando se ejecutan las pruebas y debe usarse para proporcionar una breve descripción de la prueba realizada.

Nota

Estas clases de prueba son envoltorios alrededor de `unittest` casos de prueba, parte de la biblioteca estándar de Python. Para obtener más detalles sobre esto, puede consultar la documentación oficial en https://docs.python.org/3/library/unittest.html (https://docs.python.org/3/library/unittest.html) .

Ejecutando pruebas

Las pruebas están escritas, así que ahora es el momentopara ejecutarlos. Para eso, solo necesitamos agregar la `--test-enable` opción al comando de inicio del servidor Odoo, mientras instalamos o actualizamos (`-i` o `-u`) el módulo adicional.

El comando se verá así:

Dupdo

```
$ ./odoo-bin -c 12-library.conf --test-enable -u library_checkout --stop-after-init
```

Solo se probarán los módulos instalados o actualizados, por eso `-u` se utilizó la opción. Si es necesario instalar algunas dependencias, sus pruebas también se ejecutarán. Si desea evitar esto, puede instalar el módulo que desea probar de la manera habitual, y luego, ejecutar las pruebas mientras actualiza (`-u`) el módulo para probar.

Las pruebas que tenemos todavía no prueban nada, pero deberían ejecutarse sin fallar. Al observar de cerca el registro del servidor, deberíamos poder ver los `INFO` mensajes que informan la ejecución de la prueba, como los siguientes:

Dupdo

```
INFO 12-library odoo.modules.module: odoo.addons.library_checkout.tests.test_checkout_mass_message running tests.
```

Configurar pruebas

Deberíamos comenzar preparando los datos que se utilizarán en las pruebas, en el `setUp` método. Aquí, necesitamos crear un registro de pago para usar en el asistente.

Es conveniente realizar la prueba.acciones bajo un usuario específico para probar también que el control de acceso está configurado correctamente. Esto se logra utilizando el `sudo(<user>)` método modelo. Los conjuntos de registros llevan esa información con ellos, por lo que después de crearse utilizando `sudo()` , las operaciones posteriores en el mismo conjunto de registros se realizarán utilizando ese mismo contexto.

Esto es el código para el `setUp` método:

Dupdo

```
from odoo.tests.common import TransactionCase

class TestWizard(TransactionCase):

    def setUp(self, *args, **kwargs):
        super().setUp(*args, **kwargs)
        # Setup test data
        admin_user = self.env.ref('base.user_admin')
        self.Checkout = self.env['library.checkout'].sudo(admin_user)
        self.Wizard = self.env['library.checkout.massmessage']\
.sudo(admin_user)

        a_member = self.env['library.member'].create({'name': 'John'})
        self.checkout0 = self.Checkout.create({
            'member_id': a_member.id})
```

Ahora, podemos usar el `self.checkout0` registro y el `self.Wizard` modelo para nuestras pruebas.

Escribir casos de prueba

Ahora, expandamos el `test_button_test()` método visto en nuestro esqueleto inicial. Las pruebas más simples ejecutan algo de código en el objeto probado, obtienen un resultado y luego use una declaración de aserción para compararlo con un resultado esperado.

Para verificar el método de publicación de un mensaje, la prueba cuenta el número de mensajes antes y después de ejecutar el asistente para confirmar si se agregó el nuevo mensaje. Para ejecutar el asistente, necesitamos establecerlo `active_ids` en el contexto, como lo hacen los formularios de IU, crear el registro del asistente con los valores completados en el formulario del asistente (al menos el mensaje del cuerpo) y luego ejecutar el `button_send` método.

El código completo tiene el siguiente aspecto:

Dupdo

```
    def test_button_send(self):
        "Send button creates messages on Checkouts"
        # Add test code
        msgs_before = len(self.checkout0.message_ids)

        Wizard0 = self.Wizard.with_context(active_ids=self.checkout0.ids)
        wizard0 = Wizard0.create({'message_body': 'Hello'})
        wizard0.button_send()

        msgs_after = len(self.checkout0.message_ids)
        self.assertEqual(
            msgs_after,
            msgs_before +1,
            'Expected one additional message in the Checkout.')
```

`docstring`, en la primera línea de la definición del método, es útil para describir la prueba y se imprime cuando se ejecutan las pruebas.

La verificación que verifica si la prueba fue exitosa o falló es la `self.assertEqual` declaración. Compara el número de mensajes antes y después de ejecutar el asistente; Esperamos encontrar un mensaje más que el que teníamos antes. El último parámetro proporciona un mensaje más informativo cuando falla la prueba. Es opcional, pero recomendado.

La `assertEqual` función es solo uno de los métodos de aserción disponibles. Deberíamos usar la función de aserción que sea apropiada para cada caso, ya que esto facilitará la comprensión de la causa de la falla de las pruebas. La `unittest` documentación proporciona una buena referencia para todos los métodos y está disponible en https://docs.python.org/3/library/unittest.html#test-cases (https://docs.python.org/3/library/unittest.html#test-cases) .

Para agregar un nuevo caso de prueba, agregue otro método a la clase con su implementación. Recuerde que, con las `TransactionCase` pruebas, se realiza una reversión al final de cada prueba. Por lo tanto, las operaciones realizadas en la prueba anterior se revertieron, y debemos volver a llenar la lista de tareas pendientes del asistente. A continuación, simulamos al usuario, completando el nuevo campo de fecha límite y realizando la actualización masiva. Al final, verificamos si ambas tareas pendientes terminaron con la misma fecha.

Probar excepciones

A veces, necesitamos nuestras pruebas para verificar si una excepciónfue generado. Un común case está probando si algunas validaciones se están haciendo correctamente.

En nuestro caso, el asistente realiza algunas validaciones que podemos probar. Por ejemplo, podemos probar que un mensaje de cuerpo vacío genera un error. Para verificar si se genera una excepción, colocamos el código correspondiente dentro de un `with self.assertRaises()` bloque.

Primero necesitamos importar las excepciones de Odoo en la parte superior del archivo, como se muestra a continuación:

Dupdo

```
from odoo import exceptions
```

Luego, agregamos otro método con un caso de prueba a la clase de prueba, que se muestra a continuación:

Dupdo

```
    def test_button_send_empty_body(self):
        "Send button errors on empty body message"
        wizard0 = self.Wizard.create({})
        with self.assertRaises(exceptions.UserError) as e:
            wizard0.button_send()
```

Si el `button_send()` método no genera una excepción, la verificación fallará. Si genera esa excepción, la verificación se realiza correctamente y la excepción generada se almacena en la `e` variable. Podemos usar eso para inspeccionarlo más.

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec87/raising-exceptions)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec89/development-tools)

Herramientas de desarrollo

Hay algunas técnicas para desarrolladoresdeben aprender a ayudarlos en su trabajo. Anteriormente en este libro, presentamos la **`Developer Mode`** interfaz de usuario. También tenemos una opción de servidor disponible, que proporciona algunas características amigables para los desarrolladores. Estaremos describiendo esto con más detalle en esta sección. Después de eso, discutiremos otro tema relevante para los desarrolladores: cómo depurar el código del lado del servidor.

Opciones de desarrollo del servidor

El servidor Odoo ofrece la `--dev` opción de habilitar algunos características del desarrollador para acelerar nuestro ciclo de desarrollo, como las siguientes:

- Introducir el depurador cuando se encuentra una excepción en un módulo adicional
- Recargando el código Python automáticamente una vez que se guarda un archivo Python, evitando un reinicio manual del servidor
- Lectura de definiciones de vista directamente desde archivos XML, evitando actualizaciones manuales de módulos

La `--dev` opción acepta una lista de opciones separadas por comas, y la `all` opción será adecuada la mayor parte del tiempo. También podemos especificar el depurador que preferimos usar. Por defecto, `pdb` se usa el depurador de Python,, . Algunas personas pueden preferir instalar y usar depuradores alternativos. Es bueno tener en cuenta eso `ipdb` y `pudb` también son compatibles aquí.

Nota

`Changed in Odoo 9` En las versiones de Odoo anteriores a Odoo 9, la `--debug` opción estaba disponible, permitiéndole abrir el depurador en una excepción de módulo adicional. Desde Odoo 9, esta opción ya no está disponible y fue reemplazada por la `--dev=all` opción.

Cuando se trabaja en código Python, el servidor debe reiniciarse cada vez que se cambia el código para que se vuelva a cargar. La `--dev` opción de línea de comandos se ocupa de esa recarga. Cuando el servidor detecta que se modificó un archivo Python, repite automáticamente la secuencia de carga del servidor, haciendo que el cambio de código sea efectivo de inmediato.

Para usarlo, simplemente agregue la `--dev=all` opción al comando del servidor, que se muestra a continuación:

Dupdo

```
$ ./odoo-bin -c 12-library --dev=all
```

Para que esto funcione, `watchdog` se requiere el paquete Python. Se puede instalar usando lo siguiente `pip` :

Dupdo

```
$ pip install watchdog
```

Tenga en cuenta que esto es útil solo para cambios en el código de Python y para ver arquitecturas en archivos XML. Para otros cambios, como la estructura de datos del modelo, se necesita una actualización del módulo, y la recarga no es suficiente.

Depuración

Todos sabemos que una gran parte del trabajo de un desarrollador es depurar código. Para hacer esto, a menudo utilizamos un editor de código que puede establecer puntos de interrupción y ejecutar nuestro programa paso a paso.

Si está utilizando Microsoft Windows como suestación de trabajo de desarrollo, configurar un entorno capaz de ejecutar código Odoo desde una fuente no es una tarea trivial. Además, el hecho de que Odoo es un servidor que espera las llamadas de los clientes, y solo luego actúa sobre ellas, hace que la depuración sea bastante diferente en comparación con los programas del lado del cliente.

El depurador de Python

Si bien puede parecer un poco intimidante para los recién llegados, el enfoque más pragmático para Odoo depuración es utilizar el depurador integrado Python, `pdb` . Nosotros también introduce extensiones que proporcionan una interfaz de usuario más rica, similar a lo que suelen proporcionar los IDE sofisticados.

Para usar el depurador, el mejor enfoque es insertar un punto de interrupción en el código que queremos inspeccionar, generalmente un método modelo. Esto se hace insertando la siguiente línea en el lugar deseado:

Dupdo

```
import pdb; pdb.set_trace()
```

Ahora, reinicie el servidor para que se cargue el código modificado. Tan pronto como la ejecución del programa llegue a esa línea, `(pdb)` se mostrará un mensaje de Python en la ventana de Terminal donde se está ejecutando el servidor, esperando nuestra entrada.

Este indicador funciona como un shell de Python, donde puede ejecutar cualquier expresión o comando en el contexto de ejecución actual. Esto significa que las variables actuales se pueden inspeccionar e incluso modificar. Estos son los comandos de acceso directo más importantes disponibles:

- `h` (ayuda) muestra un resumen de los `pdb` comandos disponibles
- `p` (imprimir) evalúa e imprime una expresión
- `pp` (impresión bonita) es útil para imprimir estructuras de datos, como diccionarios o listas
- `l` (lista) enumera el código alrededor de la instrucción que se ejecutará a continuación
- `n` (siguiente) pasa a la siguiente instrucción
- `s` (paso) pasos en la instrucción actual
- `c` (continuar) continúa la ejecución normalmente
- `u` (arriba) se mueve hacia arriba en la pila de ejecución
- `d` (abajo) se mueve hacia abajo en la pila de ejecución
- `bt` (retroceso) muestra la pila de ejecución actual

Si la `dev=all` opción se usa para comenzarlos servidores, cuando se genera una excepción, el servidor ingresa en un modo **post mortem** en la línea correspondiente. Este es un `pdb` aviso, como el descrito anteriormente, que nos permite inspeccionar el estado del programa en el momento en que se encontró el error.

Una sesión de depuración de muestra

Veamos qué depuración simplese parece a la sesión. Podemos comenzar agregando un punto de interrupción del depurador en la primera línea del `button_send` método del asistente, que se muestra a continuación:

Dupdo

```
    def button_send(self):
        import pdb; pdb.set_trace()
        self.ensure_one()
        # ...
```

Ahora, reinicie el servidor, abra un **`Send Message`** formulario de asistente y haga clic en el **`Send`** botón. Esto activará el `button_send` método en el servidor, y el cliente web permanecerá en un **`Loading…`** estado, esperando la respuesta del servidor. Al mirar la ventana de Terminal donde se ejecuta el servidor, verá algo similar a lo siguiente:

Dupdo

```
> /mnt/c/Users/daniel/Documents/library-app/library_checkout/wizard/checkout_mass_message.py(27)button_send()
-> self.ensure_one()
(Pdb)
```

Este es el `pdb` indicador del depurador, y las dos primeras líneas le brindan información sobre dónde se encuentra en la ejecución del código Python. La primera línea indica el archivo, el número de línea y el nombre de la función en la que se encuentra, y la segunda línea es la siguiente línea de código que se ejecutará.

Durante una sesión de depuración, los mensajes de registro del servidor pueden aparecer. Estos no dañarán nuestra depuración, pero pueden molestarnos. Podemos evitar eso reduciendo la verbosidad de los mensajes de registro. La mayoría de las veces, estos mensajes de registro serán del `werkzeug` módulo. Podemos silenciarlos usando la `--log-handler=werkzeug:CRITICAL` opción. Si esto no es suficiente, podemos bajar el nivel de registro general usando `--log-level=warn` . Otra opción es habilitar la `--logfile=/path/to/log` opción para que los mensajes de registro se redirijan desde la salida estándar a un archivo de disco.

> **Nota**
>
> Si obtiene un Terminal que aparentemente no responde, donde lo que escribe no se presenta en el terminal, puede estar relacionado con problemas de visualización en su sesión de terminal. Existe la posibilidad de que esto se pueda resolver escribiendo `<enter>reset<enter>` .

Si escribimos `h` ahora, veremos una referencia rápida de los comandos disponibles. La escritura `l` muestra la línea de código actual y las líneas de código circundantes.

Al escribir, `n` se ejecutará la línea de código actual y se pasará a la siguiente. Si solo presionamos **Enter** , se repetirá el comando anterior. Haga eso tres veces, y deberíamos estar en la declaración de devolución del método.

Podemos inspeccionar el contenido de cualquier variable o atributo, como el `checkout_ids` campo utilizado en el asistente, que se muestra a continuación:

Dupdo

```
(pdb) p self.checkout_ids
```

Se permiten todas las expresiones de Python, incluso las asignaciones variables.

Podemos continuar depurando línea por línea. En cualquierpunto, continuamos la ejecución normal escribiendo `c` .

Depuradores alternativos de Python

Si bien `pdb` tiene la ventaja de estar disponible listo para usar, puede ser bastante breve y existen algunas opciones más cómodas.

El depurador Iron Python `ipdb`, es una opción popular que utiliza los mismos comandos que `pdb`, pero agrega mejoras como la finalización de tabulación y el resaltado de sintaxis para un uso más cómodo. Se puede instalar con lo siguiente:

Dupdo

```
$ sudo pip install ipdb
```

Y se agrega un punto de interrupción con la siguiente línea:

Dupdo

```
import ipdb; ipdb.set_trace()
```

Otro depurador alternativo es `pudb`. También admite los mismos comandos `pdb` y funciona en terminales de solo texto, pero utiliza una pantalla gráfica similar a la que puede encontrar en un depurador IDE. La información útil, como las variables en el contexto actual y sus valores, está fácilmente disponible en la pantalla en sus propias ventanas.

Se puede instalar a través del administrador de paquetes del sistema o `pip`, como se muestra aquí:

Dupdo

```
$ sudo apt-get install python-pudb  # using Debian OS packages
$ sudo pip install pudb  # or using pip, possibly in a virtualenv
```

Agregar un `pudb` punto de interrupción se realiza de la manera que esperaría, como se muestra a continuación:

Dupdo

```
import pudb; pudb.set_trace()
```

Pero lo siguiente es más corto y fácil de recordar. alternativa que está disponible:

Dupdo

```
import pudb; pu.db
```

Impresión de mensajes y registro

A veces, solo necesitamos inspeccionar los valoresde algunas variables o verifique si se están ejecutando algunos bloques de código. Una `print()` función de Python puede hacer el trabajo perfectamente sin detener el flujo de ejecución. Como estamos ejecutando el servidor en una ventana de terminal, el texto impreso se mostrará en la salida estándar, pero no se almacenará en el registro del servidor si se está escribiendo en un archivo.

La `print()` función solo se está utilizando como ayuda para el desarrollo y no debe llegar al código final, listo para implementarse. Si sospecha que necesita más detalles sobre la ejecución del código, utilice mensajes de registro de nivel de depuración en su lugar.

Agregar registro de nivel de depuraciónLos mensajes en puntos sensibles de nuestro código nos permiten investigar problemas en una instancia implementada. Solo necesitaríamos elevar ese nivel de registro del servidor para depurar y luego inspeccionar los archivos de registro.

Inspeccionar y matar procesos en ejecución

También hay algunos trucos quenos permite inspeccionar una carrera Proceso Odoo.

Para eso, primero necesitamos encontrar el **ID de proceso** (**PID**) correspondiente. Para encontrar el PID, ejecute otra ventana de Terminal y escriba lo siguiente:

Dupdo

```
$ ps ax | grep odoo-bin
```

La primera columna en la salida es el PID para ese proceso. Tome nota del PID para el proceso de inspección, ya que lo necesitaremos a continuación.

Ahora, queremos enviar una señal al proceso. El comando utilizado para hacer eso es `kill`. Por defecto, envía una señal para finalizar un proceso, pero también puede enviar otras señales más amigables.

Conociendo el PID para nuestro proceso de servidor Odoo en ejecución, podemos imprimir los rastros del código que se está ejecutando actualmente enviando una señal SIGQUIT al proceso en ejecución, con lo siguiente:

Dupdo

```
$ kill -3 <PID>
```

Después de esto, si miramos la ventana de terminal o el archivo de registro donde se está escribiendo la salida del servidor, veremos la información sobre varios subprocesos que se están ejecutando, así como rastros detallados de la pila en qué línea de código están ejecutando. Esto es utilizado por algunos enfoques de creación de perfiles de código, para rastrear dónde está gastando el tiempo el servidor y perfilar la ejecución del código. En la documentación oficial, en https://www.odoo.com/documentation/12.0/howtos/profilecode.html (https://www.odoo.com/documentation/12.0/howtos/profilecode.html) , se proporciona información útil sobre la creación de perfiles de código .

Otras señales que podemos enviar al proceso del servidor Odoo son `HUP`, para recargar el servidor y / `INT` o `TERM` forzar al servidor a apagarse, como se muestra a continuación:

Dupdo

```
$ kill -HUP <PID>
$ kill -TERM <PID>
```

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec88/unit-tests)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec90/summary)

Resumen

Revisamos las características de la API de ORM y cómo usarlas para crear aplicaciones dinámicas que reaccionen ante los usuarios, lo que les ayuda a evitar errores y automatizar tareas tediosas.

Las validaciones del modelo y los campos calculados pueden abarcar muchos casos de uso, pero no todos. Aprendimos a extender los métodos de creación, escritura y desvinculación de API para cubrir otros casos de uso.

Para una interacción de usuario enriquecida, utilizamos los `mail` mixins de complementos principales para agregar características para que los usuarios se comuniquen alrededor de los documentos y planifiquen actividades en ellos. Los asistentes permiten a la aplicación dialogar con el usuario y recopilar los datos necesarios para ejecutar procesos particulares. Las excepciones permiten que la aplicación anule operaciones incorrectas, informando al usuario del problema y revocando los cambios intermedios, manteniendo el sistema consistente.

También discutimos las herramientas disponibles para que los desarrolladores creen y mantengan sus aplicaciones: mensajes de registro, herramientas de depuración y pruebas unitarias.

En el próximo capítulo, seguiremos trabajando con el ORM, pero analizándolo desde el punto de vista de una aplicación externa, trabajando con el servidor Odoo como un back-end para almacenar datos y ejecutar procesos comerciales.

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec89/development-tools)

siguiente Sección (/book/business/9781789532470/8/ch08lvl1sec91/further-reading)

Otras lecturas

Estos son los materiales de referencia más relevantes para los temas tratados en este capítulo:

- Referencia de ORM: https://www.odoo.com/documentation/12.0/reference/orm.html#common-orm-methods (https://www.odoo.com/documentation/12.0/reference/orm.html#common-orm-methods)
- Características de mensajes y actividades: https://www.odoo.com/documentation/12.0/reference/mixins.html (https://www.odoo.com/documentation/12.0/reference/mixins.html)
- Referencia de pruebas de Odoo: https://www.odoo.com/documentation/12.0/reference/testing.html (https://www.odoo.com/documentation/12.0/reference/testing.html)
- `unittest` Referencia de Python : https://docs.python.org/3/library/unittest.html#module-unittest (https://docs.python.org/3/library/unittest.html#module-unittest)

Sección anterior Sección (/book/business/9781789532470/8/ch08lvl1sec90/summary)

siguiente Sección (/book/business/9781789532470/9)