4
Application Models

In this chapter, we will cover the following topics:

- Defining the Model representation and order
- Adding data fields to a Model
- Using a float field with configurable precision
- Adding a monetary field to a Model
- Adding relational fields to a Model
- Adding a hierarchy to a Model
- Adding constraint validations to a Model
- Adding computed fields to a Model
- Exposing Related fields stored in other models
- Adding dynamic relations using Reference fields
- Adding features to a Model using inheritance
- Using Abstract Models for reusable Model features
- Using Delegation inheritance to copy features to another Model

Introduction

In order to concisely get the point through, the recipes in this chapter make small additions to an existing addon module. We chose to use the module created by the recipes in *Chapter 3, Creating Odoo Modules*. To better follow the examples here, you should have that module created and ready to use.

Defining the Model representation and order

Models have structural attributes defining their behavior. These are prefixed with an underscore and the most important is `_name`, which defines the internal global identifier for the Model.

There are two other attributes we can use. One to set the field used as a representation, or title, for the records, and another one to set the order they are presented in.

Getting ready

This recipe assumes that you have an instance ready with `my_module`, as described in *Chapter 3*, *Creating Odoo Modules*.

How to do it...

The `my_module` instance should already contain a Python file called `models/library_book.py`, which defines a basic model. We will edit it to add a new class-level attribute after `_name`:

1. To add a human-friendly title to the model, add the following:

   ```
   _description = 'Library Book'
   ```

2. To have records sorted first by default from newer to older and then by title, add the following:

   ```
   _order = 'date_release desc, name'
   ```

3. To use the `short_name` field as the record representation, add the following:

   ```
   _rec_name = 'short_name'
   short_name = fields.Char('Short Title')
   ```

When we're done, our `library_book.py` file should look like this:

```
# -*- coding: utf-8 -*-
from openerp import models, fields
class LibraryBook(models.Model):
    _name = 'library.book'
    _description = 'Library Book'
    _order = 'date_release desc, name'
    _rec_name = 'short_name'
    name = fields.Char('Title', required=True)
    short_name = fields.Char('Short Title')
    date_release = fields.Date('Release Date')
    author_ids = fields.Many2many('res.partner', string='Authors')
```

We should then upgrade the module to have these changes activated in Odoo.

How it works...

The first step adds a friendlier description title to the model's definition. This is not mandatory, but can be used by some addons. For instance, it is used by the tracking feature in the `mail` addon module for the notification text when a new record is created. For more details, see *Chapter 12*, *Automation and Workflows*.

By default, Odoo orders the records using the internal `id` value. But this can be changed to use the fields of our choice by providing a `_order` attribute with a string containing a comma-separated list of field names. A field name can be followed with the `desc` keyword to have it sorted in reverse order.

Only fields stored in the database can be used. Non-stored computed fields can't be used to sort records.

[The syntax for the `_sort` string is similar to SQL `ORDER BY` clauses, although it's stripped down. For instance, special clauses such as `NULLS FIRST` are not allowed.]

Model records have a representation used when they are referenced from other records. For example, a `user_id` field with the value 1 represents the **Administrator** user. When displayed in a form view, we will be shown the user name rather than the database ID. By default, the `name` field is used. In fact, that is the default value for the `_rec_name` attribute, and that's why it's convenient to have a `name` field in our Models.

If no `name` field exists in the model, a representation is generated with the model and record identifiers, similar to **(library.book, 1)**.

There's more...

Record representation is available in a magic `display_name` computed field added automatically to all models since version 8.0. Its values are generated using the Model method `name_get()`, which was already in existence in previous Odoo versions.

Its default implementation uses the `_rec_name` attribute. For more sophisticated representations, we can override its logic. This method must return a list of tuples with two elements: the ID of the record and the Unicode string representation for the record.

For example, to have the title and its release date in the representation, such as **Moby Dick (1851-10-18)**, we could define the following:

```
def name_get(self):
    result = []
    for record in self:
        result.append(
            (record.id,
             u"%s (%s)" % (record.name, record.date_released)
            ))
    return result
```

Do notice that we used a Unicode string while building the record representation, `u"%s (%s)"`. This is important to avoid errors, in case we find non-ASCII characters.

Adding data fields to a model

Models are meant to store data, and this data is structured in fields. Here, we will learn about the several types of data that can be stored in fields, and how to add them to a model.

Getting ready

This recipe assumes that you have an instance ready with the `my_module` addon module available, as described in *Chapter 3*, *Creating Odoo Modules*.

How to do it...

The `my_module` addon module should already have a `models/library_book.py` defining a basic Model. We will edit it to add new fields:

1. Use the minimal syntax to add fields to the Library Books model:

```
from openerp import models, fields
class LibraryBook(models.Model):
    # ...
    short_name = fields.Char('Short Title')
    notes = fields.Text('Internal Notes')
    state = fields.Selection(
        [('draft', 'Not Available'),
         ('available', 'Available'),
         ('lost', 'Lost')],
        'State')
    description = fields.Html('Description')
    cover = fields.Binary('Book Cover')
    out_of_print = fields.Boolean('Out of Print?')
    date_release = fields.Date('Release Date')
    date_updated = fields.Datetime('Last Updated')
    pages = fields.Integer('Number of Pages')
    reader_rating = fields.Float(
        'Reader Average Rating',
        (14, 4),  # Optional precision (total, decimals),
    )
```

2. All these fields support a few common attributes. As an example, we could edit the preceding `pages` field to add them:

```
    pages = fields.Integer(
        string='Number of Pages',
        default=0,
        help='Total book page count',
        groups='base.group_user',
        states={'cancel': [('readonly', True)]},
        copy=True,
        index=False,
        readonly=False,
        required=False,
        company_dependent=False,
        )
```

3. The `Char` fields support a few specific attributes. As an example, we can edit the `short_name` field to add them:

```
    short_name = fields.Char(
        string='Short Title',
        size=100,  # For Char only
        translate=False,  # also for Text fields
        )
```

4. The HTML fields also have specific attributes:

```
    description = fields.Html(
        string='Description',
        # optional:
        sanitize=True,
        strip_style=False,
        translate=False,
    )
```

Upgrading the module will make these changes effective in the Odoo model.

How it works...

Fields are added to models by defining an attribute in its Python class. The non-relational field types available are as follows:

- `Char` for string values.
- `Text` for multi-line string values.
- `Selection` for selection lists. This has a list of values and description pairs. The value that is selected is what gets stored in the database, and it can be a string or an integer.
- `Html` is similar to the Text field, but is expected to store rich text in the HTML format.
- `Binary` fields store binary files, such as images or documents.
- `Boolean` stores `True`/`False` values.
- `Date` stores date values. The ORM handles them in the string format, but they are stored in the database as dates. The format used is defined in `openerp.fields.DATE_FORMAT`.
- `Datetime` for date-time values. They are stored in the database in a naive date time, in UTC time. The ORM represents them as a string and also in UTC time. The format used is defined in `openerp.fields.DATETIME_FORMAT`.
- `Integer` fields need no further explanation.
- `Float` fields store numeric values. The precision can optionally be defined with a total number of digits and decimal digits pairs.
- `Monetary` can store an amount in a certain currency; it is also explained in another recipe.

The first step in the recipe shows the minimal syntax to add each field type. The field definitions can be expanded to add other optional attributes, as shown in step 2.

Here is an explanation for the field attributes used:

- `string` is the field's title, used in UI view labels. It actually is optional; if not set, a label will be derived from the field name by adding title case and replacing underscores with spaces.
- `size` only applies to `Char` fields and is the maximum number of characters allowed. In general, it is advised not to use it.
- `translate` when set to `True`, makes the field translatable; it can hold a different value depending on the user interface language.
- `default` is the default value. It can also be a function that is used to calculate the default value. For example, `default=_compute_default`, where `_compute_default` is a method defined on the model before the field definition.
- `help` is an explanation text displayed in the UI tooltips.

- `groups` makes the field available only to some security groups. It is a string containing a comma-separated list of XML IDs for security groups. This is addressed in more detail in *Chapter 10*, *Access Security*.
- `states` allows the user interface to dynamically set the value for the `readonly`, `required`, and `invisible` attributes, depending on the value of the `state` field. Therefore, it requires a `state` field to exist and be used in the form view (even if it is invisible).
- `copy` flags if the field value is copied when the record is duplicated. By default, it is `True` for non-relational and `Many2one` fields and `False` for `One2many` and computed fields.
- `index`, when set to `True`, makes for the creation of a database index for the field, allowing faster searches. It replaces the deprecated `select=1` attribute.
- The `readonly` flag makes the field read-only by default in the user interface.
- The `required` flag makes the field mandatory by default in the user interface.
- The `sanitize` flag is used by HTML fields and strips its content from potentially insecure tags.
- `strip_style` is also an HTML field attribute and has the sanitization to also remove style elements.
- The `company_dependent` flag makes the field store different values per company. It replaces the deprecated `Property` field type.

There's more...

The `Selection` field also accepts a function reference instead of a list, as its "selection" attribute. This allows for dynamically generated lists of options. You can see an example of this in the *Add dynamic relations using Reference fields* recipe, where a selection attribute is also used.

The `Date` and `Datetime` field objects expose a few utility methods that can be convenient.

For `Date`, we have the following:

- `fields.Date.from_string(string_value)` parses the string into a date object.
- `fields.Date.to_string(date_value)` represents the `Date` object as a string.
- `fields.Date.today()` returns the current day in string format.
- `fields.Date.context_today(record)` returns the current day in string format according to the timezone of the record's (or recordset) context.

For `Datetime`, we have the following:

- `fields.Datetime,from_string(string_value)` parses the string into a `datetime` object.
- `fields.Datetime,to_string(datetime_value)` represents the `datetime` object as a string.
- `fields.Datetime,now()` returns the current day and time in string format. This is appropriate to use for default values.
- `fields.Datetime,context_timestamp(record, value)` converts a `value` naive date-time into a timezone-aware date-time using the timezone in the context of `record`. This is not suitable for default values.

Other than basic fields, we also have relational fields: `Many2one`, `One2many`, and `Many2many`. These are explained in the *Add relational fields to a model* recipe.

It's also possible to have fields with automatically computed values, defining the computation function with the `compute` field attribute. This is explained in the *Adding computed fields to a model* recipe.

A few fields are added by default in Odoo models, so we should not use these names for our fields. These are the `id` field, for the record's automatically generated identifier, and a few audit log fields, which are as follows:

- `create_date` is the record creation timestamp
- `create_uid` is the user that created the record
- `write_date` is the last recorded edit timestamp
- `write_uid` is the user that last edited the record

The automatic creation of these log fields can be disabled by setting the `_log_access=False` model attribute.

Another special column that can be added to a model is `active`. It should be a Boolean flag allowing for mark records as inactive. Its definition looks like this:

```
active = fields.Boolean('Active', default=True)
```

By default, only records with `active` set to `True` are visible. To have them retrieved, we need to use a domain filter with `[('active', '=', False)]`. Alternatively, if the `'active_test': False` value is added to the environment Context, the ORM will not filter out inactive records.

Using a float field with configurable precision

When using float fields, we may want to let the end user configure the precision that is to be used. The **Decimal Precision Configuration** module addon provides this ability.

We will add a **Cost Price** field to the Library Book model, with a user-configurable number of digits.

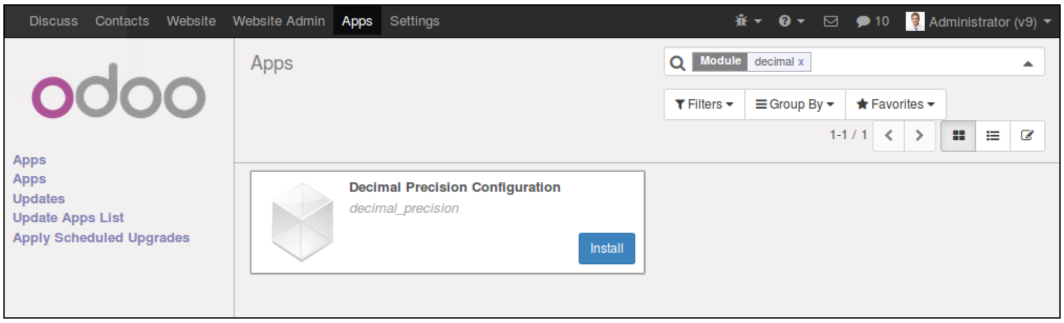
Getting ready

We will reuse the `my_module` addon module from *Chapter 3*, *Creating Odoo Modules*.

How to do it...

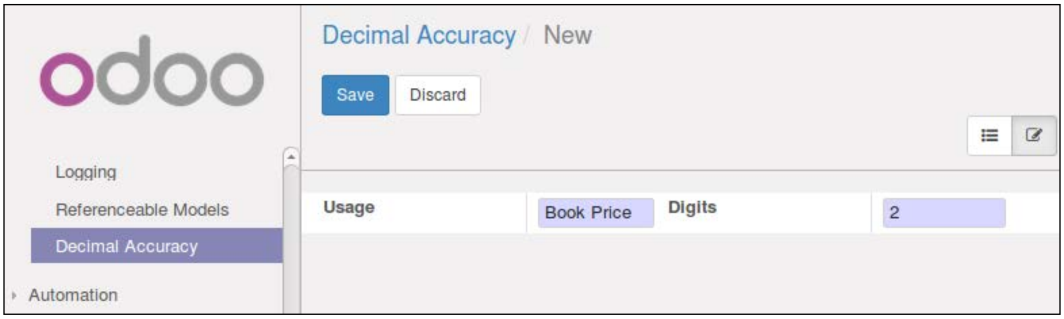
We need to install the `decimal_precision` module, add a "Usage" entry for our configuration, and then use it in the model field:

1. Make sure the Decimal Accuracy module is installed; select **Apps** from the top menu, remove the default filter, search for the **Decimal Precision Configuration** app, and install it if it's not already installed:



2. Activate the **Developer Mode** in the **About** dialog box, available within the **?** icon in the menu bar at the top. This will enable the **Settings** | **Technical** menu.
3. Access the Decimal Precision configurations. To do this, open the **Settings** top menu and select **Technical** | **Database Structure** | **Decimal Accuracy**. We should see a list of the currently defined settings.

4. Add a new configuration, setting **Usage** to **Book Price** and choosing the **Digits** precision:



Usage: Book Price
Digits: 2

5. Add the new dependency to the `__openerp__.py` manifest file. It should be similar to this:

```
{   'name': 'Chapter 03 code',
    'depends': ['base', 'decimal_precision],
    'data': ['views/library_book.xml'] }
```

6. To add the model field using this decimal precision setting, edit the `models/library_book.py` file by adding the following:

```
from openerp.addons import decimal_precision as dp
# ...
class LibraryBook(models.Model):
    # ...
    cost_price = fields.Float(
        'Book Cost', dp.get_precision('Book Price))
```

How it works...

The `get_precision()` function looks up the name in the Decimal Accuracy **Usage** field and returns a tuple representing 16-digit precision with the number of decimals defined in the configuration.

Using this function in the field definition, instead of having it hardcoded, allows the end user to configure it according to his needs.

Adding a monetary field to a Model

Odoo has special support for monetary values related to a currency. Let's see how to use it in a Model.

> The Monetary field was introduced in Odoo 9.0 and is not available in previous versions. If you are using Odoo 8.0, the float field type is your best alternative.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3, Creating Odoo Modules*.

How to do it...

The monetary field needs a complementary currency field to store the currency for the amounts.

The `my_module` already has a `models/library_book.py` defining a basic Model. We will edit this to add the needed fields:

1. Add the field to store the currency that is to be used:

```
class LibraryBook(models.Model):
    # ...
    currency_id = fields.Many2one(
        'res.currency', string='Currency')
```

2. Add the monetary field to store our amount:

```
class LibraryBook(models.Model):
    # ...
    retail_price = fields.Monetary(
        'Retail Price',
        # optional: currency_field='currency_id',
        )
```

Now, upgrade the addon module, and the new fields should be available in the Model. They won't be visible in views until they are added to them, but we can confirm their addition by inspecting the Model fields in **Settings** | **Technical** | **Database Structure** | **Models**.

How it works...

Monetary fields are similar to Float fields, but Odoo is able to represent them correctly in the user interface since it knows what their currency is through a second field for that purpose.

This currency field is expected to be named `currency_id`, but we can use whatever field name we like as long as it is indicated using the `currency_field` optional parameter. You might like to know that the decimal precision for the amount is taken from the currency definition (the `decimal_precision` field of the `res.currency` model).

Adding relational fields to a Model

Relations between Odoo Models are represented by relational fields. We can have three different types of relations: many-to-one, one-to-many, and many-to-many. Looking at the Library Books example, we see that each book can have one Publisher, so we can have a many-to-one relation between books and publishers.

Looking at it from the Publishers point of view, each Publisher can have many Books. So, the previous many-to-one relation implies a one-to-many reverse relation.

Finally, there are cases where we can have a many-to-many relation. In our example, each book can have several (many) Authors. And inversely, each Author can have written many books. Looking at it from either side, this is a many-to-many relation.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3, Creating Odoo Modules*.

How to do it...

Odoo uses the Partner model, `res.partner`, to represent persons, organizations, and addresses. So, we should use it for authors and publishers. We will edit the `models/library_book.py` file to add these fields:

1. Add to `Library Books` the many-to-one field for the book's publisher:

```
class LibraryBook(models.Model):
    # ...
    publisher_id = fields.Many2one(
        'res.partner', string='Publisher',
        # optional:
        ondelete='set null',
        context={},
        domain=[],
        )
```

2. To add the one-to-many field for a publisher's books, we need to extend the partner model. For simplicity, we will add that to the same Python file:

```
class ResPartner(models.Model):
    _inherit = 'res.partner'
    book_ids = fields.One2many(
        'library.book', 'publisher_id',
        string='Published Books')
```

3. The many-to-many relation between books and authors was already created, but let's revisit it:

```
class LibraryBook(models.Model):
    # ...
    author_ids = fields.Many2many(
        'res.partner', string='Authors')
```

4. The same relation, but from authors to books, should be added to the Partner model:

```
class ResPartner(models.Model):
    # ...
    book_ids = fields.Many2many(
        'library.book',
        string='Authored Books',
        # relation='library_book_res_partner_rel'  # optional
        )
```

Now, upgrade the addon module, and the new fields should be available in the Model. They won't be visible in views until they are added to them, but we can confirm their addition by inspecting the Model fields in **Settings** | **Technical** | **Database Structure** | **Models**.

How it works...

Many-to-one fields add a column to the database table of the model storing the database ID of the related record. At the database level, a foreign key constraint will also be created for it, ensuring that the stored IDs are a valid reference to a record in the related table. No database index is created for these relation fields, but this should often be considered; it can be done by adding the attribute `index=True`.

We can see that there are four more attributes we can use for many-to-one fields:

The `ondelete` attribute determines what happens when the related record is deleted. For example, what happens to Books when their Publisher record is deleted? The default is `'set null'`, setting an empty value on the field. It can also be `'restrict'`, which prevents the related record from being deleted, or `'cascade'`, which causes the linked record to also be deleted.

The last two (`context` and `domain`) are also valid for the other relational fields. They are mostly meaningful on the client side and at the model level act just as default values to be used in the client-side views.

- `context` adds variables to the client context when clicking through the field to the related record's view. We can, for example, use it to set default values on that view.
- `domain` is a search filter used to limit the list of related records available for selection when choosing a value for our field.

Both context and domain are explained in more detail in *Chapter 8*, *Backend Views*.

One-to-many fields are the reverse of many-to-one relations, and although they are added to models just like other fields, they have no actual representation in the database. They are instead programmatic shortcuts and enable views to represent these lists of related records.

Many-to-many relations also don't add columns in the tables for the models. This type of relation is represented in the database using an intermediate relation table, with two columns to store the two related IDs. Adding a new relation between a Book and an Author creates a new record in the relation table with the ID for the Book and the ID for the Author.

Odoo automatically handles the creation of this relation table. The relation table name is, by default, built using the name of the two related models plus a `_rel` suffix. But, we can override it using the `relation` attribute. A case to keep in mind is when the two table names are large enough for the automatically generated database identifiers to exceed the PostgreSQL limit of 63 characters.

As a rule of thumb, if the names of the two related tables exceed 23 characters, you should use the `relation` attribute to set a shorter name. In the next section, we will go into more detail on this.

There's more...

The `Many2one` fields support an additional `auto_join` attribute. It is a flag that allows the ORM to use SQL joins on this field. Because of this, it bypasses the usual ORM control such as user access control and record access rules. On a specific case, it can solve a performance issue, but it is advised to avoid using it.

We have seen the shortest way to define the relational fields. For completeness, these are the attributes specific to this type of field:

The `One2many` field attributes are as follows:

- `comodel_name`: This is the target model identifier and is mandatory for all relational fields, but it can be defined position-wise without the keyword
- `inverse_name`: This applies only to `One2many` and is the field name in the target model for the inverse `Many2one` relation
- `limit`: This applies to `One2many` and `Many2many` and sets an optional limit on the number of records to read that are used at the user interface level

The `Many2many` field attributes are as follows:

- `comodel_name`: (as defined earlier)
- `relation`: This is the name to use for the table supporting the relation, overriding the automatically defined name
- `column1`: This is the name for the `Many2one` field in the relational table linking to this model
- `column2`: This is the name for the `Many2one` field in the relational table linking to the `comodel`

For `Many2many` relations, in most cases the ORM will take perfect care of the default values for these attributes. It is even capable of detecting inverse `Many2many` relations, detecting the already existing `relation` table and appropriately inverting the `column1` and `column2` values.

But there are two cases where we need to step in and provide our own values for these attributes. One is the case where we need more than one `Many2many` relation between the same two models. For this to be possible, we must provide ourselves with a `relation` table name that is different from the first relation. The other case is when the database names of the related tables are long enough for the automatically generated relation name to exceed the 63 character PostgreSQL limit for database object names.

The relation table's automatic name is `<model1>_<model2>_rel`. But this relation table also creates an index for its primary key with the following identifier:

```
<model1>_<model2>_rel_<model1>_id_<model2>_id_key
```

It also needs to meet the 63 characters limit. So, if the two table names combined exceed a total of 63 characters, you will probably have trouble meeting the limits and will need to manually set the `relation` attribute.

Adding a hierarchy to a Model

Hierarchies are represented using model relations with itself; each record has a parent record in the same model and also has many child records. This can be achieved by simply using many-to-one relations between the model and itself.

But Odoo also provides improved support for this type of field using the **Nested set model** (`https://en.wikipedia.org/wiki/Nested_set_model`). When activated, queries using the `child_of` operator in their domain filters will run significantly faster.

Staying with the Library Books example, we will build a hierarchical category tree that could be used to categorize books.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3, Creating Odoo Modules*.

How to do it...

We will add a new Python file, `models/library_book_categ.py`, for the category tree, shown as follows:

1. To have the new Python code file loaded, add this line to `models/__init__.py`:

   ```
   from . import library_book_categ
   ```

2. To create the Book Category model with the parent and child relations, create the `models/library_book_categ.py` file with the following:

   ```
   # -*- coding: utf-8 -*-
   from openerp import models, fields, api
   class BookCategory(models.Model):
       _name = 'library.book.category'
       name = fields.Char('Category')
       parent_id = fields.Many2one(
           'library.book.category',
           string='Parent Category',
           ondelete='restrict',
           index=True)
       child_ids = fields.One2many(
           'library.book.category', 'parent_id',
           string='Child Categories')
   ```

3. To enable the special hierarchy support, also add the following:

   ```
   _parent_store = True
   parent_left = fields.Integer(index=True)
   parent_right = fields.Integer(index=True)
   ```

4. To add a check preventing looping relations, add the following line to the model:

   ```
   @api.constrains('parent_id')
   def _check_hierarchy(self):
       if not self._check_recursion():
           raise models.ValidationError(
               'Error! You cannot create recursive categories.')
   ```

Finally, a module upgrade should make these changes effective.

How it works...

Steps 1 and 2 create the new model with hierarchic relations. The `Many2one` relation adds a field to reference the parent record. For faster child record discovery, this field is indexed in the database using the `index=True` parameter. The `parent_id` field must have `ondelete` set to either `'cascade'` or `'restrict'`.

At this point, we have all that is required to have a hierarchic structure. But there are a few more additions we can make to enhance it.

The `One2many` relation does not add any additional fields to the database but provides a shortcut to access all the records with this record as their parent.

In step 3, we activate the special support for hierarchies. This is useful for high-read but low-write instructions, since it brings faster data browsing at the expense of costlier write operations. It is done by adding two helper fields, `parent_left` and `parent_right`, and setting the model attribute to `_parent_store=True`. When this attribute is enabled, the two helper fields will be used to store data in searches in the hierarchic tree.

By default, it is assumed that the field for the record's Parent is called `parent_id`, but a different name can be used. In that case, the correct field name should be indicated using the additional model attribute `_parent_name`. The default is as follows:

```
_parent_name = 'parent_id'
```

Step 4 is advised in order to prevent cyclic dependencies in the hierarchy— that is, having a record both in the ascending and descending trees. This is dangerous for programs that navigate through the tree, since they can get into an infinite loop. The `models.Model` provides a utility method for this (`_check_recursion`) that we have reused here.

There's more...

The technique shown here should be used for "static" hierarchies, which are read and queried often but seldom updated. Book categories are a good example, since the Library will not be continuously creating new categories, but readers will often be restricting their searches to a category and all its children categories. The reason for this lies in the implementation of the Nested Set Model in the database, which requires an update of the `parent_left` and `parent_right` columns (and the related database indexes) for all records whenever a category is inserted, removed, or moved. This can be a very expensive operation, especially when multiple editions are being performed in parallel transactions.

If you are dealing with a very dynamic hierarchical structure, the standard `parent_id` and `child_ids` relations can result in better performance.

Adding constraint validations to a Model

Models can have validations preventing them from entering undesired conditions. Two different types of constraint can be used: the ones checked at the database level and the ones checked at the server level.

Database level constraints are limited to the constraints supported by PostgreSQL. The most commonly used are the `UNIQUE` constraints, but `CHECK` and `EXCLUDE` constraints can also be used. If these are not enough for our needs, we can use Odoo server level constraints, written in Python code.

We will use the Library Book model created in *Chapter 3*, *Creating Odoo Modules,* and add a couple of constraints to it. We will add a database constraint preventing duplicate book titles, and a Python model constraint preventing release dates in the future.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3*, *Creating Odoo Modules*.

We expect it to contain at least the following:

```
# -*- coding: utf-8 -*-
from openerp import models, fields
class LibraryBook(models.Model):
    _name = 'library.book'
    name = fields.Char('Title', required=True)
    date_release = fields.Date('Release Date')
```

How to do it...

We will edit the `LibraryBook` class in the `models/library_book_categ.py` Python file:

1. To create the database constraint, add a model attribute:

   ```
   class LibraryBook(models.Model):
       # ...
       _sql_constraints = [
           ('name_uniq',
            'UNIQUE (name)',
            'Book title must be unique.')
           ]
   ```

2. To create the Python code constraint, add a `model` method:

   ```
   from openerp import api
   class LibraryBook(models.Model):
       # ...
       @api.constrains('date_release')
       def _check_release_date(self):
           for r in self:
               if r.date_release > fields.Date.today():
                   raise models.ValidationError(
                       'Release date must be in the past')
   ```

After these changes are made to the code file, an addon module upgrade and server restart are needed.

How it works...

The first step has a database constraint created on the model's table. It is enforced at the database level. The `_sql_constraints` model attribute accepts a list of constraints to create. Each constraint is defined by a three element tuple; they are listed as follows:

- A suffix to use for the constraint identifier. In our example, we used `name_uniq` and the resulting constraint name is `library_book_name_uniq`.
- The SQL to use in the PostgreSQL instruction to alter or create the database table.
- A message to report to the user when the constraint is violated.

As mentioned earlier, other database table constraints can be used. Note that column constraints, such as `NOT NULL`, can't be added this way. For more information on PostgreSQL constraints in general and table constraints in particular, take a look at `http://www.postgresql.org/docs/9.4/static/ddl-constraints.html`.

In the second step, we added a method to perform a Python code validation. It is decorated with `@api.constrains`, meaning that it should be executed to run checks when one of the fields in the argument list is changed. If the check fails, a `ValidationError` exception should be raised.

> The `_constraints` model attribute is still available but has been deprecated since version 8.0. Instead, we should use methods with the new `@api.constrains` decorator.

Adding computed fields to a Model

Sometimes, we need to have a field that has a value calculated or derived from other fields in the same record or in related records. A typical example is the total amount that is calculated by multiplying a unit price with a quantity. In Odoo models, this can be achieved using computed fields.

To show how computed fields work, we will add one to the Library Books model to calculate the days since the book's release date.

It is also possible to make computed fields editable. We will also implement this in our example.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3*, *Creating Odoo Modules*.

How to do it...

We will edit the `models/library_book.py` code file to add a new field and the methods supporting its logic:

1. Start by adding the new field to the `Library Books` model:

```
class LibraryBook(models.Model):
    # ...
    age_days = fields.Float(
        string='Days Since Release',
        compute='_compute_age',
        inverse='_inverse_age',
        search='_search_age',
        store=False,
        compute_sudo=False,
    )
```

2. Next, add the method with the value computation logic:

```
# ...
from openerp import api  # if not already imported
from openerp.fields import Date as fDate
# ...
class LibraryBook(models.Model):
    # …
    @api.depends('date_release')
    def _compute_age(self):
        today = fDate.from_string(fDate.today())
        for book in self.filtered('date_release'):
            delta = (fDate.from_string(book.date_release -
                     today)
            book.age_days = delta.days
```

3. To add the method implementing the logic to write on the computed field, use the following code:

```
from datetime import timedelta as td
# ...
class LibraryBook(models.Model):
    # …
    def _inverse_age(self): today =
    fDate.from_string(fDate.today())
        for book in self.filtered('date_release'):
            d = td(days=book.age_days) - today
            book.date_release = fDate.to_string(d)
```

4. To implement the logic allowing you to search on the computed field, use the following code:

```
# from datetime import timedelta as td
class LibraryBook(models.Model):
    # …
    def _search_age(self, operator, value):
        today = fDate.from_string(fDate.today())
        value_days = td(days=value)
        value_date = fDate.to_string(today - value_days)
        return [('date_release', operator, value_date)]
```

An Odoo restart followed by a module upgrade should be needed to correctly activate these new additions.

How it works...

The definition of a computed field is the same as the one for a regular field, except that a `compute` attribute is added to specify the name of the method to use for its computation.

The similarity can be deceptive, since computed fields are internally quite different from regular fields. Computed fields are dynamically calculated at runtime, and unless you specifically add that support yourself, they are not writeable or searchable.

The computation function is dynamically calculated at runtime, but the ORM uses caching to avoid inefficiently recalculating it every time its value is accessed. So, it needs to know what other fields it depends on, using the `@depends` decorator to detect when its cached values should be invalidated and recalculated.

> Make sure that the `compute` function always sets a value on the computed field. Otherwise, an error will be raised. This can happen when you have `if` conditions in your code that sometimes fail to set a value on the computed field, and can be tricky to debug.

Write support can be added by implementing the `inverse` function; it uses the value assigned to the computed field to update the origin fields. Of course, this only makes sense for simpler calculations; nevertheless, there are still cases where it can be useful. In our example, we make it possible to set the book release date by editing the **Days Since Release** computed field.

It is also possible to make searchable a non-stored computed field by setting the `search` attribute to the method name to use (similar to `compute` and `inverse`).

However, this method is not expected to implement the actual search. Instead, it receives as parameters the operator and value used to search on the field and is expected to return a domain with the replacement search conditions to use. In our example, we translate a search of the **Days Since Release** field into an equivalent search condition on the **Release Date** field.

The optional `store=True` flag makes the field stored in the database. In this case, after being computed, the field values are stored in the database, and from there on, they are retrieved like regular fields instead of being recomputed at runtime. Thanks to the `@api.depends` decorator, the ORM will know when these stored values need to be recomputed and updated. You can think of it as a persistent cache. It also has the advantage of making the field usable for search conditions, sorting and grouping by operations, without the need to implement the `search` method.

The `compute_sudo=True` flag is to be used in those cases where the computations need to be done with elevated privileges. This can be the case when the computation needs to use data that may not be accessible to the end user.

Exposing Related fields stored in other models

When reading data from the server, Odoo clients can only get values for the fields available in the model being queried. Client-side code can't use dot notation to access data in the related tables like server-side code can.

But those fields can be made available there by adding them as related fields. We will do this to make the publisher's city available in the Library Book model.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3*, *Create Odoo Modules*.

How to do it...

Edit the `models/library_book.py` file to add the new "related" field:

1. Make sure that we have a field for the book Publisher:

```
class LibraryBook(models.Model):
    # ...
    publisher_id = fields.Many2one(
        'res.partner', string='Publisher')
```

2. Now, add the related field for the Publisher's city:

```
# class LibraryBook(models.Model):
    # ...
    publisher_city = fields.Char(
        'Publisher City',
        related='publisher_id.city')
```

Finally, we need to upgrade the addon module for the new fields to be available in the Model.

How it works...

Related fields are just like regular fields, but they have the additional attribute, `related`, with a string for the separated chain of fields to traverse.

In our case, we access the Publisher related record through `publisher_id`, and then read its `city` field. We can also have longer chains, such as `publisher_id.country_id.country_code`.

There's more...

Related fields are in fact computed fields. They just provide a convenient shortcut syntax to read field values from related models. As a computed field, this means that the `store` attribute is also available to them. As a shortcut, they also have all the attributes from the referenced field, such as `name`, `translatable`, `required`, and so on.

Additionally, they support a `related_sudo` flag similar to `compute_sudo`; when set to `True`, the field chain is traversed without checking user access rights.

Adding dynamic relations using Reference fields

With relational fields, we need to decide beforehand the relation's target model (or comodel). But sometimes, we may need to leave that decision to the user and first choose the model we want and then the record we want to link to.

With Odoo, this can be achieved using Reference fields.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3, Creating Odoo Modules*.

How to do it...

Edit the `models/library_book.py` file to add the new related field:

1. We first add a helper method to dynamically build the list of selectable target models:

```
from openerp import models, fields, api
class LibraryBook(models.Model):
    # …
    @api.model
    def _referencable_models(self):
        models = self.env['res.request.link'].search([])
        return [(x.object, x.name) for x in models]
```

2. Then, we add the Reference field and use the previous function to provide the list of selectable models:

```
ref_doc_id = fields.Reference(
    selection='_referencable_models',
    string='Reference Document')
```

Since we are changing the model's structure, a module upgrade is needed to activate these changes.

How it works...

Reference fields are similar to many-to-one fields, except that they allow the user to select the model to link to.

The target model is selectable from a list provided by the `selection` attribute. The `selection` attribute must be a list of two element tuples, where the first is the model internal identifier, and the second is a text description for it.

Here is an example:

```
[('res.users', 'User'), ('res.partner', 'Partner')]
```

However, rather than providing a fixed list, we can use a model list configurable by end users. That is the purpose of the built-in **Referenceable Models** available in the **Settings** | **Technical** | **Database Structure** menu option. This model's internal identifier is `res.request.link`.

Our recipe started with providing a function to browse all the Model records that can be referenced to dynamically build a list to be provided to the `selection` attribute. Although both forms are allowed, we declared the function name inside quotes, instead of a direct reference to the function without quotes. This is more flexible, and for example, allows for the referenced function to be defined only later in the code, which is something that is not possible when using a direct reference.

The function needs the `@api.model` decorator because it operates on the model level, not on the recordset level.

While this feature looks nice, it comes with a significant execution overhead. Displaying the reference fields for a large number of records, for instance, in a list view, can create heavy database loads as each value has to be looked up in a separate query. It is also unable to take advantage of database referential integrity like regular relation fields can.

Adding features to a Model using inheritance

One of the most important Odoo features is the ability of module addons extending features defined in other module addons without having to edit the code of the original feature. This might be to add fields or methods, or modify existing fields, or extend existing methods to perform additional logic.

It is the most frequently used method of inheritance and is referred to by the official documentation as **traditional inheritance** or **classical inheritance**.

We will extend the built in Partner model to add it to a computed field with the authored book count. This involves adding a field and a method to an existing model.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3*, *Creating Odoo Modules*.

How to do it...

We will be extending the built-in Partner model. We should do this in its own Python code file, but to keep the explanation as simple we can, we will reuse the `models/library_book.py` code file:

1. First, we make sure the `authored_book_ids` inverse relation is in the Partner model and add the computed field:

   ```
   class ResPartner(models.Model):
       _inherit = 'res.partner'
       _order = 'name'
       authored_book_ids = fields.Many2many(
           'library.book', string='Authored Books')
       count_books = fields.Integer(
           'Number of Authored Books',
           compute='_compute_count_books'
           )
   ```

2. Next, add the method needed to compute the book count:

   ```
   # ...
   from openerp import api  # if not already imported
   # class ResPartner(models.Model):
       # ...
       @api.depends('authored_book_ids')
       def _compute_count_books(self):
           for r in self:
               r.count_books = len(r.authored_book_ids)
   ```

Finally, we need to upgrade the addon module for the modifications to take effect.

How it works...

When a model class is defined with the `_inherit` attribute, it adds modifications to the inherited model rather than replacing it.

This means that fields defined in the inheriting class are added or changed on the parent model. At the database layer, it is adding fields on the same database table.

Fields are also incrementally modified. This means that if the field already exists in the super class, only the attributes declared in the inherited class are modified; the other ones are kept as in the parent class.

Methods defined in the inheriting class replace the method in the parent class. So, unless they include a call to the parent's version of the method, we will lose features. Because of this, when we want to add logic to existing methods, they should include a statement with `super` to call its version in the parent class. This is discussed in more detail in *Chapter 5, Basic Server Side Business Log*.

There's more...

With the `_inherit` traditional inheritance, it's also possible to copy the parent model's features into a completely new model. This is done by simply adding a `_name` model attribute with a different identifier. Here is an example:

```
class LibraryMember(models.Model):
    _inherit = 'res.partner'
    _name = 'library.member'
```

The new model has its own database table with its own data totally independent from the `res.partner` parent model. Since it still inherits from the Partner model, any later modifications to it will also affect the new model.

In the official documentation, this is called **prototype inheritance**, but in practice, it is seldom used. The reason is that delegation inheritance usually answers to that need in a more efficient way, without the need to duplicate data structures. For more information on it, you can refer to the *Use Delegation inheritance to copy features to another Model* recipe.

Using Abstract Models for reusable Model features

Sometimes, there is a particular feature that we want to be able to add to several different models. Repeating the same code in different files is bad programming practice, so it would be nice to be able to implement it once and be able to reuse it many times.

Abstract models allow us to just create a generic model that implements some feature that can then be inherited by regular models in order to make that feature available in them.

As an example, we will implement a simple `Archive` feature. It adds the `active` field to the model (if it doesn't exist already) and makes available an archive method to toggle the `active` flag. This works because `active` is a magic field; if present in a model by default, the records with `active=False` will be filtered out from queries.

We will then add it to the Library Book model.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3, Creating Odoo Modules*.

How to do it...

The archive feature would certainly deserve its own addon module, or at least its own Python code file. But to keep the explanation as simple as possible, we will cram it into the `models/library_book.py` file:

1. Add the abstract model for the archive feature. It must be defined in the `Library Book` model, where it will be used:

   ```
   class BaseArchive(models.AbstractModel):
       _name = 'base.archive'
       active = fields.Boolean(default=True)

       def do_archive(self):
           for record in self:
               record.active = not record.active
   ```

2. Now, we will edit the `Library Book` model to inherit the Archive model:

   ```
   class LibraryBook(models.Model):
       _name = 'library.book'
       _inherit = ['base.archive']
       # ...
   ```

An upgrade to the addon module is needed for the changes to be activated.

How it works...

An Abstract model is created by a class based on `models.AbstractModel` instead of the usual `models.Model`. It has all the attributes and capabilities of regular models; the difference is that the ORM will not create an actual representation for it in the database. So, it can have no data stored in it. It serves only as a template for a reusable feature that is to be added to regular models.

Our Archive abstract model is quite simple; it just adds the `active` field and a method to toggle the value of the `active` flag, which we expect to later be used via a button on the user interface.

When a model class is defined with the `_inherit` attribute, it inherits the attribute methods of those classes, and what is defined in our class adds modifications to these inherited features.

The mechanism at play here is the same as that for a regular model extension (as per the *Add features to a Model using inheritance* recipe). You may have noticed that here, `_inherit` uses a list of model identifiers instead of a string with one model identifier. In fact, `_inherit` can have both forms. Using the list form allows us to inherit from multiple (usually Abstract) classes. In this case, we are inheriting just one, so a text string would be fine. A list was used instead to underline that a list can be used.

There's more...

A noteworthy built-in abstract model is `ir.needaction_mixin`. It allows for records to signal that a user action is needed on them and is widely used together with the Social Network messaging features.

Another widely used abstract model is `mail.thread`, provided by the `mail` (Discuss) addon module. It enables, on models, the message features that power the message wall seen at the bottom of many forms.

Other than `AbstractModel`, a third model type is available: `models.TransientModel`.

It has database representation like `models.Model`, but the records created there are supposed to be temporary and regularly purged by a server-scheduled job. Other than that, Transient models work just like regular models.

They are useful for more complex user interactions known as wizards, for example, to request the user some input to then run a process or a report. In *Chapter 6*, *Advanced Server Side Development Techniques*, we explore how to use them for advanced user interaction.

Using Delegation inheritance to copy features to another Model

Traditional inheritance using `_inherit` performs in-place modification to extend the model's features.

But there are cases where rather than modifying an existing model, we want to create a new model based on an existing one to leverage the features it already has. This is one with Odoo's **delegation inheritance** that uses the model attribute, `_inherits` (note the additional `s`).

Traditional inheritance is quite different than the concept in object-oriented programming. Delegation inheritance in turn is similar in that a new model can be created to include the features from a parent model. It also supports polymorphic inheritance, where we inherit from two or more other models.

We have a library with books. It's about time for our library to also have members. For a library member, we need all the identification and address data found in the Partner model, and we also want it to keep some information regarding the membership: a start date, termination date, and card number.

Adding those fields to the Partner model is not the best solution since they will be not be used for Partners that are not members. It would be great to extend the Partner model to a new model with some additional fields.

Getting ready

We will reuse the `my_module` addon module from *Chapter 3*, *Creating Odoo Modules*.

How to do it...

The new Library Member model should be in its own Python code file, but to keep the explanation as simple as possible, we will reuse the `models/library_book.py` file:

1. Add the new model, inheriting from `res.partner`:

```
# from openerp import models, fields  # if not done yet
class LibraryMember(models.Model):
    _name = 'library.member'
    _inherits = {'res.partner': 'partner_id'}
    partner_id = fields.Many2one(
        'res.partner',
        ondelete='cascade')
```

2. Next, we add the fields that are specific to Library Members:

```
# class LibraryMember(models.Model):
    # ...
    date_start = fields.Date('Member Since')
    date_end = fields.Date('Termination Date')
    member_number = fields.Char()
```

Now, we should upgrade the addon module to have the changes activated.

How it works...

The `_inherits` model attribute sets the parent models that we want to inherit from. In this case, just one: `res.partner`. Its value is a key-value dictionary where the keys are the inherited models, and the values are the field names used to link to them. These are `Many2one` fields that we must also define in the model. In our example, `partner_id` is the field that will be used to link with the `Partner` parent model.

To better understand how it works, let's look at what happens on the database level when we create a new Member:

- A new record is created in the `res_partner` table
- A new record is created in the `library_member` table
- The `partner_id` field of the `library_member` table is set to the `id` of the `res_partner` record that is created for it

The Member record is automatically linked to a new Partner record. It's just a many-to-one relation, but the delegation mechanism adds some magic so that the Partner's fields are seen as if belonging to the Member record, and a new Partner record is also automatically created with the new Member.

You might like to know that this automatically created Partner record has nothing special about it. It's a regular Partner, and if you browse the Partner model, you will be able to find that record (without the additional Member data, of course). All Members are at the same time Partners, but only some Partners are also Members.

So, what happens if you delete a Partner record that is also a Member? You decide by choosing the `ondelete` value for the relation field. For `partner_id`, we used `cascade`. This means that deleting the Partner would also delete the corresponding Member. We could have used the more conservative setting `restrict` to forbid deleting the Partner while it has a linked Member. In this case, only deleting the Member would work.

It's important to note that delegation inheritance only works for fields and not for methods. So, if the Partner model has a `do_something()` method, the Members model will not automatically inherit it.

There's more...

A noteworthy case of delegation inheritance is the Users model, `res.users`. It inherits from Partners (`res.partner`). This means that some of the fields that you can see on the User are actually stored in the Partner model (notably the `name` field). When a new User is created, we also get a new automatically created Partner.

We should also mention that traditional inheritance with `_inherit` can also copy features into a new model, although in a less efficient way. This was discussed in the *Add features to a Model using inheritance* recipe.