5
Basic Server Side Development

In this chapter, we will cover the following topics:

- Defining model methods and use the API decorators
- Reporting errors to the user
- Obtaining an empty recordset for a different model
- Creating new records
- Updating values of recordset records
- Searching for records
- Combining recordsets
- Filtering recordsets
- Traversing recordset relations
- Extending the business logic defined in a model
- Extending write() and create()
- Customizing how records are searched

Introduction

In *Chapter 4, Application Models*, we have seen how to declare or extend business models in custom modules. The recipes in that chapter cover writing methods for computed fields as well as methods to constrain the values of fields. This chapter focuses on the basics of server side development in Odoo—method definitions, recordset manipulation, and extending inherited methods.

Defining model methods and use the API decorators

The model classes defining custom data models declare fields for the data processed by the model. They can also define custom behavior by defining methods on the model class.

In this recipe, we will see how to write a method that can be called by a button in the user interface, or by some other piece of code in our application. This method will act on `LibraryBooks` and perform the required actions to change the state of a selection of books.

Getting ready

This recipe assumes you have an instance ready, with the `my_module` addon module available, as described in *Chapter 3*, *Creating Odoo Module*s. You will need to add a state field to the `LibraryBook` model defined as follows:

```
from openerp import models, fields, api
class LibraryBook(models.Model):
    # […]
    state = fields.Selection([('draft', 'Unavailable'),
                              ('available', 'Available'),
                              ('borrowed', 'Borrowed'),
                              ('lost', 'Lost')],
                             'State')
```

Please refer to *Chapter 4*, *Adding data fields to a Model* for more clarity.

How to do it...

For defining a method on `LibraryBook` to allow changing the state of a selection of books, you need to add the following code to the model definition:

1. Add a helper method to check whether a state transition is allowed:

```
@api.model
def is_allowed_transition(self, old_state, new_state):
    allowed= [('draft', 'available'),
              ('available', 'borrowed'),
              ('borrowed', 'available'),
              ('available', 'lost'),
              ('borrowed', 'lost'),
              ('lost', 'available')]
    return (old_state, new_state) in allowed
```

2. Add a method to change the state of some books to a new one passed as an argument:

```
@api.multi
def change_state(self, new_state):
    for book in self:
        if book.is_allowed_transition(book.state,
                                      new_state):
            book.state = new_state
        else:
            continue
```

How it works...

The code in the recipe defines two methods. They are normal Python methods, having `self` as their first argument and can have additional arguments as well.

The methods are decorated with **decorators** from the `openerp.api` module. These decorators are there to ensure the conversion of calls made using the **old** or **traditional** API to the **new** API. The old API is used by the remote procedure call (RPC) protocol of the web client and by modules that have not yet been ported to the new API. A key part of this conversion is the creation of an execution **environment** stored in `self.env`; it contains the following:

- `self.env.cr`: This is a database cursor
- `self.env.user`: This is the user executing the action
- `self.env.context`: This is **context**, which is a Python dictionary containing various information such as the language of the user, his configured time zone, and other specific keys that can be set at run time by the actions of the user interface

When writing a new method, you will generally be using `@api.multi`. In addition to the creation of the environment, this decorator tells the RPC layer to initialize `self` using the record IDs supplied in the RPC argument `ids`. In such methods, `self` is a recordset that can refer to an arbitrary number of database records.

The `@api.model` decorator is similar but is used on methods for which only the model is important, not the contents of the recordset, which is not acted upon by the method. The concept is similar to Python's `@classmethod` decorator (but we generally cannot use this in Odoo models because we need access to `self.env` and other important instance attributes).

The `@api.model` and `@api.multi` decorators also have specific meaning when ensuring the compatibility of your addon module with other modules still using the "old API". You will find more information on this in the *Porting old API code to the new API* recipe in *Chapter 6*, *Advanced Server Side Development Techniques*.

Here is an example code snippet called `change_state()`:

```
# returned_book_ids is a list of book ids to return
books = self.env['library.book']
books.browse(returned_book_ids).change_state('available')
```

When `change_state()` is called, `self` is a (possibly empty) recordset containing records of the `library.book` model. The body of the `change_state()` method loops over `self` to process each book in the recordset. Looping on `self` looks strange at first, but you will get used to this pattern very quickly.

Inside the loop, `change_state()` calls `is_allowed_transition()`. The call is made using the local variable `book`, but it could have been made on any recordset for the `library.book` model, including, for example, `self`, since `is_allowed_transition()` is decorated with `@api.model`. If the transition is allowed, `change_state()` assigns the new state to the book by assigning a value to the attribute of the recordset. This is only valid on recordsets of length `1`, which is guaranteed to be the case when iterating over `self`.

There's more...

There are some important factors that are worth understanding.

Hiding methods from the RPC interface

In the old API, methods with a name prefixed by an underscore are not exposed through the RPC interface and are therefore not callable by the web client. This is still the case with the new API, which also offers another way to make a method unavailable to the RPC interface; if you don't put an `@api.model` or `@api.multi` decorator on it (or on one of the decorators mentioned in the *Porting old API code to the new API* recipe in *Chapter 6*, *Advanced Server Side Development Techniques*), then the method will neither be callable by extensions of the model using the traditional API nor via RPC.

The @api.one decorator

You may encounter the `@api.one` decorator while reading source code. In Odoo 9.0, **this decorator is deprecated** because its behavior can be confusing—at first glance, and knowing of `@api.multi`, it looks like this decorator allows the method to be called only on recordsets of size `1`, but it does not. When it comes to recordset length, `@api.one` is similar to `@api.multi`, but it does a `for` loop on the recordset outside the method and aggregates the returned value of each iteration of the loop in a list, which is returned to the caller. Avoid using it in your code.

See also

In *Chapter 8*, *Backend Views*, refer to the *Adding buttons to form* recipe to learn how to call such a method from the user interface.

You will find additional information on the decorators defined in `openerp.api` in the recipe from *Chapter 6*, *Advanced Server Side Development Techniques: Porting old API code to the new API*.

Reporting errors to the user

During method execution, it is sometimes necessary to abort the processing because an error condition was met. This recipe shows how to do this so that a helpful error message is displayed to the user when a method which writes a file to disk encounters an error.

Getting ready

To use this recipe, you need a method, which can have an abnormal condition. We will use the following one:

```
import os
from openerp import models, fields, api

class SomeModel(models.Model):
    data = fields.Text('Data')

    @api.multi
    def save(self, filename):
        path = os.path.join('/opt/exports', filename)
        with open(path, 'w') as fobj:
            for record in self:
                fobj.write(record.data)
                fobj.write('\n')
```

This method can fail because of permission issues, or a full disk, or an illegal name, which would cause an `IOError` or an `OSError` exception to be raised.

How to do it...

To display an error message to the user when an error condition is encountered, you need to take the following steps:

1. Add the following import at the beginning of the Python file:

   ```
   from openerp.exceptions import UserError
   ```

2. Modify the method to catch the exception raised and raise a `UserError` exception:

   ```
   @api.multi
   def save(self, filename):
       if '/' in filename or '\\' in filename:
   ```

```
            raise UserError('Illegal filename %s' % filename)
        path = os.path.join('/opt/exports', filename)
        try:
            with open(path, 'w') as fobj:
                for record in self:
                    fobj.write(record.data)
                    fobj.write('\n')
        except (IOError, OSError) as exc:
            message = 'Unable to save file: %s' % exc
            raise UserError(message)
```

How it works...

When an exception is raised in Python, it propagates up the call stack until it is processed. In Odoo, the RPC layer that answers the calls made by the web client catches all exceptions and, depending on the exception class, it will trigger different possible behaviors on the web client.

Any exception not defined in `openerp.exceptions` will be handled as an Internal Server Error (**HTTP status 500**), with the stack trace. A `UserError` will display an error message in the user interface. The code of the recipe changes the `OSError` to a `UserError` to ensure the message is displayed in a friendly way. In all cases, the current database transaction is rolled back.

Of course, it is not required to catch an exception with a `try..except,` construct to raise a `UserError` exception. It is perfectly OK to test for some condition, such as the presence of illegal characters in a filename, and to raise the exception when that test is `True`. This will prevent further processing of the user request.

There's more...

There are a few more exception classes defined in `openerp.exceptions`, all deriving the base legacy `except_orm` exception class. Most of them are only used internally, apart from the following:

- `Warning`: In Odoo 8.0, `openerp.exceptions.Warning` played the role of `UserError` in 9.0. It is now deprecated because the name was deceptive (it is an error, not a warning) and it collided with the Python built-in `Warning` class. It is kept for backward compatibility only and you should use `UserError` in 9.0.
- `ValidationError`: This exception is raised when a Python constraint on a field is not respected. In *Chapter 4*, *Application Models*, refer to the *Adding constraint validations to a Model* recipe for more information.

Obtaining an empty recordset for a different model

When writing Odoo code, the methods of the current model are available via `self`. If you need to work on a different model, it is not possible to directly instantiate the class of that model—you need to get a recordset for that model to start working.

This recipe shows how to get an empty recordset for any model registered in Odoo inside a model method.

Getting ready

This recipe will reuse the setup of the library example in the addon module `my_module`.

We will write a small method in the `library.book` model searching for all `library.members`. To do this, we need to get an empty recordset for `library.members`.

How to do it...

To get a recordset for `library.members` in a method of `library.book`, you need to take the following steps:

1. In the `LibraryBook` class, write a method called `get_all_library_members`:

```
class LibraryBook(models.Model):
    # ...
    @api.model
    def get_all_library_members(self):
        # ...
```

2. In the body of the method, use the following code:

```
        library_member_model = self.env['library.member']
        return library_member_model.search([])
```

How it works...

At start up, Odoo loads all the modules and combines the various classes deriving from `Model` and defining or extending a given model. These classes are stored in the Odoo **registry** indexed by name. The **environment** in `self.env` provides a shortcut access to the registry by emulating a Python dictionary; if you know the name of the model you're looking for, `self.env[model_name]` will get you an empty recordset for that model. Moreover, the recordset will share the environment of `self`.

The call to `search()` is explained in the *Searching for records* recipe later.

See also

The *Changing the user performing an action* and *Calling a method with a modified enviroment* recipes in *Chapter 6*, *Advanced Server Side Development Techniques*, deal with modifying `self.env` at runtime.

Creating new records

A frequent need when writing business logic methods is to create new records. This recipe explains how to create records of the `res.partner` model, which is defined in Odoo's `base` addon module. We will create a new partner representing a company, with some contacts.

Getting ready

You need to know the structure of the models for which you want to create a record, especially their names and types as well as any constraints existing on these fields (for example, whether some of them are mandatory). The `res.partner` model defined in Odoo has a very large number of fields, and to keep things simple, we will only use a few of these. Moreover, the model definition in Odoo uses the old API. To help you follow the recipe, here is a port of the model definition we will be using for the new API:

```
class ResPartner(models.Model):
    _name = 'res.partner'
    name = fields.Char('Name', required=True)
    email = fields.Char('Email')
    date = fields.Date('Date')
    is_company = fields.Boolean('Is a company')
    parent_id = fields.Many2one('res.partner', 'Related Company')
    child_ids = fields.One2many('res.partner', 'parent_id',
                                'Contacts')
```

How to do it...

In order to create a partner with some contacts, you need to take the following steps:

1. Inside the method that needs to create a new partner, get the current date formatted as a string expected by `create()`:

```
today_str = fields.Date.context_today()
```

2. Prepare a dictionary of values for the fields of the first contact:

```
val1 = {'name': u'Eric Idle',
        'email': u'eric.idle@example.com'
        'date': today_str}
```

3. Prepare a dictionary of values for the fields of the second contact:

```
val2 = {'name': u'John Cleese',
        'email': u'john.cleese@example.com',
        'date': today_str}
```

4. Prepare a dictionary of values for the fields of the company:

```
partner_val = {
    'name': u'Flying Circus',
    'email': u'm.python@example.com',
    'date': today_str,
    'is_company': True,
    'child_ids': [(0, 0, val1),
                  (0, 0, val2),
                  ]
}
```

5. Call the `create()` method to create the new records:

```
record = self.env['res.partner'].create(partner_val)
```

How it works...

To create a new record for a model, we can call the `create(values)` method on any recordset related to the model. This method returns a new recordset of length `1` containing the new record, with fields values specified in the `values` dictionary.

In the dictionary:

- Text field values are given with Python strings (preferably Unicode strings).
- `Float` and `Integer` field values are given using Python floats or integers.
- `Boolean` field values are given, preferably using Python booleans or integer.
- `Date` (resp. `Datetime`) field values are given as Python strings. Use `fields.Date.to_string()` (resp. `fields.Datetime.to_string()`) to convert a Python `datetime.date` (resp. `datetime.datetime`) object to the expected format.
- `Binary` field values are passed as a Base64 encoded string. The `base64` module from the Python standard library provides methods such as `encodestring(s)` to encode a string in Base64.
- `Many2one` field values are given with an integer, which has to be the database ID of the related record.

- `One2many` and `Many2many` fields use a special syntax. The value is a list containing tuples of three elements, as follows:

Tuple	Effect
`(0, 0, dict_val)`	Create a new record that will be related to the main record
`(6, 0, id_list)`	Create a relation between the record being created and existing records, whose IDs are in the Python list `id_list` Caution: When used on a `One2many`, this will remove the records from any previous relation

In the recipe, we create the dictionaries for two contacts in the company we want to create, and then we use these dictionaries in the `child_ids` entry of the dictionary for the company being created, using the `(0, 0, dict_val)` syntax explained previously.

When `create()` is called in step 5, three records are created:

- One for the main partner company, which is returned by `create`
- One for each of the two contacts, which are available in `record.child_ids`

There's more

If the model defined some **default values** for some fields, nothing special needs to be done; `create()` will take care of computing the default values for the fields not present in the supplied dictionary.

On the other hand, **onchange methods** are *not* called by `create()`, because they are called by the web client during the initial edition of the record. Some of these methods compute default values for fields related to a given field. When creating records by hand you have to do the work yourself, either by providing explicit values or by calling the onchange methods. The *Calling onchange methods on the server side* recipe in *Chapter 6*, *Advanced Server Side Development Techniques* explains how to do this.

Updating values of recordset records

Business logic often means updating records by changing the values of some of their fields. This recipe shows how to add a contact for a partner and modify the date field of the partner as we go.

Getting ready

This recipe will be using the same simplified `res.partner` definition as the *Creating new records* recipe previously. You may refer to this simplified definition to know the fields.

The `date` field of `res.partner` has no defined meaning in Odoo. In order to illustrate our purpose, we will use this to record an activity date on the partner, so creating a new contact should update the date of the partner.

How to do it...

To update a partner, you can write a new method called `add_contact()` defined like this:

```
@api.model
def add_contacts(self, partner, contacts):
    partner.ensure_one()
    if contacts:
        partner.date = fields.Date.context_today()
        partner.child_ids |= contacts
```

How it works...

The method starts by checking whether the partner passed as an argument contains exactly one record by calling `ensure_one()`. This method will raise an exception if this is not the case and the processing will abort. This is needed, as we cannot add the same contacts to several partners at the same time.

Then the method checks that the `contacts` recordset is not empty, because we don't want to update the `date` of the partner if we are not modifying it.

Finally, the method modifies the values of the attributes of the partner record. Since `child_ids` is a `One2many` relation, its value is a recordset. We use `|=` to compute the union of the current contacts of the partner and the new contacts passed to the method. See the following recipe, *Combining recordsets*, for more information on these operators.

Note that no assumption is made on `self`. This method could be defined on any model class.

There's more...

There are three options available if you want to write new values to the fields of records.

Option 1 is the one explained in the recipe, and it works in all contexts—assigning values directly on the attribute representing the field of the record. It is not possible to assign a value to all recordset elements in one go, so you need to iterate on the recordset, unless you are certain that you are only handling a single record.

Option 2 is to use the `update()` method by passing a dictionary mapping field names to the values you want to set. This also only works for recordsets of length `1`. It can save some typing when you need to update the values of several fields at once on the same record. Here is step 2 of the recipe, rewritten to use this option:

```
@api.model
def add_contacts(self, partner, contacts):
    partner.ensure_one()
    if contacts:
        today = fields.Date.context_today()
        partner.update(
            {'date': today,
             'child_ids': partner_child_ids | contacts}
        )
```

Option 3 is to call the `write()` method, passing a dictionary mapping field names to the values you want to set. This method works for recordsets of arbitrary size and will update all records with the specified values in one single database operation when the two previous options perform one database call per record and per field. However, it has some limitations:

- It does not work if the records are not yet present in the database (see *Writing onchange methods* in *Chapter 6*, *Advanced Server Side Development Techniques* for more information on this)
- It requires using a special format when writing relational fields, similar to the one used by the `create()` method

Tuple	Effect
`(0, 0, dict_val)`	This creates a new record that will be related to the main record.
`(1, id, dict_val)`	This updates the related record with the specified ID with the supplied values.
`(2, id)`	This removes the record with the specified ID from the related records and deletes it from the database.
`(3, id)`	This removes the record with the specified ID from the related records. The record is not deleted from the database.
`(4, id)`	This adds an existing record with the supplied ID to the list of related records.
`(5, )`	This removes all the related records, equivalent to calling `(3, id)` for each related `id`.
`(6, 0, id_list)`	This creates a relation between the record being updated and the existing record, whose IDs are in the Python list `id_list`.

At the time of this writing, the official documentation is outdated and mentions that the operation numbers 3, 4, 5, and 6 are not available on `One2many` fields, which is no longer true. However, some of these may not work with `One2many` fields, depending on constraints on the models; for instance, if the reverse `Many2one` relation is required, then operation 3 will fail because it would result in an unset `Many2one` relation.

Searching for records

Searching for records is also a common operation in business logic methods. This recipe shows how to find all the `Partner` companies and their contacts by company name.

Getting ready

This recipe will be using the same simplified `res.partner` definition as the *Creating new records* recipe previously. You may refer to this simplified definition to know the fields.

We will write the code in a method called `find_partners_and_contact(self, name)`.

How to do it...

In order to find the partners, you need to perform the following steps:

1. Get an empty recordset for `res.partner`:

```
@api.model
def find_partners_and_contacts(self, name):
    partner = self.env['res.partner']
```

2. Write the search domain for your criteria:

```
domain = ['|',
          '&',
          ('is_company', '=', True),
          ('name', 'like', name),
          '&',
          ('is_company', '=', False),
          ('parent_id.name', 'like', name)
          ]
```

3. Call the `search()` method with the domain and return the recordset:

```
return partner.search(domain)
```

How it works...

Step 1 defines the method. Since we are not using the contents of `self`, we decorate it with `@api.model`, but this is not linked to the purpose of this recipe. Then, we get an empty recordset for the `res.partner` model because we need it to search `res.partner` records.

Step 2 creates a search domain in a local variable. Often you'll see this creation inlined in the call to search, but with complex domains, it is a good practice to define it separately.

For a full explanation of the **search domain** syntax, please refer to the *Defining filters on record lists: Domain* recipe in *Chapter 8*, *Backend Views*.

Step 3 calls the `search()` method with the domain. The method returns a recordset containing all records matching the domain, which can then be further processed. In the recipe, we call the method with just the domain, but the following keyword arguments are also supported:

- `offset=N`: This is used to skip the `N` first records that match the query. This can be used together with `limit` to implement pagination or to reduce memory consumption when processing a very large number of records. It defaults to `0`.
- `limit=N`: return at most `N` records. By default, there is no limit.
- `order=sort_specification`: This is used to force the order on the returned recordset. By default, the order is given by the `_order` attribute of the model class.
- `count=boolean`: If `True`, this returns the number of records instead of the recordset. It defaults to `False`.

We recommend using the `search_count(domain)` method rather than `search(domain, count=True)`, as the name of the method conveys the behavior in a much clearer way; both will give the same result.

There's more...

We said that the `search()` method returned all the records matching the domain. This is not completely true. The method ensures that only records to which the user performing the search has access are returned. Additionally, if the model has a `boolean` field called **active** and no term of the search domain is specifying a condition on that field, then an implicit condition is added by search to only return `active=True` records. So if you expect a search to return something but you only get empty recordsets, be sure to check the value of the `active` field (if present) and to check for **record rules**.

See the recipe *Calling a method with a different context* in *Chapter 6*, *Advanced Server Side Development Techniques* for a way to not have the implicit `active = True` condition added. See the *Limit record access using record rules* recipe in *Chapter 10*, *Accessing Security* for more information about record level access rules.

If for some reason you find yourself writing raw SQL queries to find record IDs, be sure to use `self.env['record.model'].search([('id', 'in', tuple(ids))).ids` after retrieving the IDs to make sure that security rules are applied. This is especially important in **multicompany** Odoo instances where record rules are used to ensure proper discrimination between companies.

Combining recordsets

Sometimes, you will find that you have obtained recordsets which are not exactly what you need. This recipe shows various ways of combining them.

Getting ready

To use this recipe, you need to have two or more recordsets for the same model.

How to do it...

Here is the way to achieve common operations on recordsets:

1. To merge two recordsets into one while preserving their order, use the following operation:

   ```
   result = recordset1 + recordset2
   ```

2. To merge two recordsets into one ensuring that there are no duplicates in the result, use the following operation:

   ```
   result = recordset1 | recordset2
   ```

3. To find the records that are common to two recordsets, use the following operation:

   ```
   result = recordset1 & recordset2
   ```

How it works...

The class for recordsets implements various Python operator redefinitions, which are used here. Here is a summary table of the most useful Python operators that can be used on recordsets:

Operator	Action performed
`R1 + R2`	This returns a new recordset containing the records from `R1` followed by the records from `R2`. This can generate duplicate records in the recordset.
`R1 - R2`	This returns a new recordset consisting of the records from `R1`, which are not in `R2`. The order is preserved.

Operator	Action performed
`R1 & R2`	This returns a new recordset with all the records that belong to both `R1` and `R2` (intersection of recordsets). The order is *not* preserved here.
`R1 | R2`	This returns a new recordset with the records belonging to either `R1` or `R2` (union of recordsets). The order is *not* preserved, but there are no duplicates.
`R1 == R2`	`True` if both recordsets contain the same records.
`R1 <= R2` `R1 in R2`	`True` if all records in `R1` are also in `R2`. Both syntaxes are equivalent.
`R1 >= R2` `R2 in R1`	`True` if all records in `R2` are also in `R1`. Both syntaxes are equivalent.
`R1 != R2`	`True` if `R1` and `R2` do not contain the same records.

There are also in-place operators `+=`, `-=`, `&=`, and `|=`, which modify the left-hand operand instead of creating a new recordset. These are very useful when updating a record's `One2many` or `Many2many` fields. See the *Updating values of recordset records* recipe previously for an example.

There's more...

The `sorted()` method will sort the records in a recordset. Called without arguments, the `_order` attribute of the model will be used. Otherwise, a function can be passed to compute a comparison key in the same fashion as the Python built-in `sorted(sequence, key)` function. The `reverse` keyword argument is also supported.

Note about performance

When the default `_order` parameter of the model is used, the sorting is delegated to the database and a new `SELECT` function is performed to get the order. Otherwise, the sorting is performed by Odoo. Depending on what is being manipulated, and on the size of the recordsets, there can be some important performance differences.

Filtering recordsets

In some cases, you already have a recordset, but you need to operate only on some records. You can, of course, iterate on the recordset, checking for the condition on each iteration and acting depending on the result of the check. It can be easier, and in some cases, more efficient to construct a new recordset containing only the interesting records and calling a single operation on that recordset.

This recipe shows how to use the `filter()` method to extract a recordset from another one.

Getting ready

We will reuse the simplified `res.partner` model shown in the *Create new records* recipe previously. This recipe defines a method to extract partners having an e-mail address from a supplied recordset.

How to do it...

In order to extract records with an e-mail address from a recordset, you need to perform the following steps:

1. Define the method accepting the original recordset:

```
@api.model
def partners_with_email(self, partners):
```

2. Define an inner predicate function:

```
    def predicate(partner):
        if partner.email:
            return True
        return False
```

3. Call `filter()` as follows:

```
return partners.filter(predicate)
```

How it works...

The implementation of the `filter()` method of recordsets creates an empty recordset in which it adds all the records for which the predicate function evaluates to `True`. The new recordset is finally returned. The order of records in the original recordset is preserved.

The preceding recipe used a named internal function. For such simple predicates, you will often find an anonymous lambda function used:

```
@api.model
def partners_with_email(self, partners):
    return partners.filter(lambda p: p.email)
```

Actually, to filter a recordset based on the fact that one attribute is *truthy* in the Python sense, you can use `partners.filter('email')`.

There's more...

Keep in mind that `filter()` operates in the memory. If you are trying to optimize the performance of a method on the critical path, you may want to use a search domain or even move to SQL, at the cost of readability.

Traversing recordset relations

When working with a recordset of length `1`, the various fields are available as record attributes. Relational attributes (`One2many`, `Many2one`, and `Many2many`) are also available with values that are recordsets too. When working with recordsets with more than one record, the attributes cannot be used.

This recipe shows how to use the `mapped()` method to traverse recordset relations; we will write two methods performing the following operations:

- Retrieving the e-mails of all contacts of a single partner company passed as an argument
- Retrieving the various companies to which some contact partners are related

Getting ready

We will be reusing the simplified Partner model shown in the *Create new records* recipe of this chapter.

How to do it...

To write the partner manipulation methods, you need to perform the following steps:

1. Define a method called `get_email_addresses()`:

```
@api.model
def get_email_addresses(self, partner):
    partner.ensure_one()
```

2. Call `mapped()` to get the e-mail addresses of the contacts of the partner:

```
    return partner.mapped('child_ids.email')
```

3. Define a method called `get_companies()`:

```
@api.model
def get_companies(self, partners):
```

4. Call `mapped()` to get the different companies of the partners:

```
    return partners.mapped('parent_id')
```

How it works...

In step 1, we call `ensure_one()` to make sure we have a single partner. This is not required for the recipe, as `mapped()` works very well on recordsets of arbitrary size. However, it is mandated by the specification of the method we are writing, as we don't want to retrieve contacts from multiple companies by mistake.

In step 2 and step 4, we call the `mapped(path)` method to traverse the fields of the recordset; `path` is a string containing field names separated by dots. For each field in the path, `mapped()` produces a new recordset containing all the records related by this field to all elements in the current recordset and then applies the next element in the path on that new recordset. If the last field in the path is a relational field, `mapped()` will return a recordset; otherwise, a Python list is returned.

The `mapped()` method has two remarkable properties:

- If the path is a single scalar field name, then the returned list is in the same order as the processed recordset.
- If the path contains a relational field, then order is not preserved, but duplicates are removed from the result.

This second property is very useful in a method decorated with `@api.multi` where you want to perform an operation on all the records pointed by a `Many2many` field of all records in `self`, but need to make sure that the action is performed only once (even if two records of self share the same target record).

There's more...

When using `mapped()`, keep in mind that it operates in the memory inside the Odoo server by repeatedly traversing relations and therefore making SQL queries, which may not be efficient; however, the code is terse and expressive. If you are trying to optimize a method on the critical path of the performance of your instance, you may want to rewrite the call to `mapped()` and express it as a `search()` with the appropriate domain, or even move to SQL (at the cost of readability).

The `mapped()` method can also be called with a function as argument. In this case, it returns a list containing the result of the function applied to each record of `self`, or the union of the recordsets returned by the function, if the function returns a recordset.

See also

- The *Search for records* recipe
- The *Executing raw SQL queries* recipe In *Chapter 6*, *Advanced Server Side Development Techniques*

Extending the business logic defined in a Model

When defining a model that extends another model, it is often necessary to customize the behavior of some methods defined on the original model. This is a very easy task in Odoo, and one of the most powerful features of the underlying framework.

We will demonstrate this by extending a method that creates records to add a new field in the created records.

Getting ready

If you want to follow the recipe, make sure you have the `my_module` addon from *Chapter 3*, *Creating Odoo Modules*, with the loan wizard defined in the *Writing a wizard to guide the user* recipe from *Chapter 6*, *Advanced Server Side Development Techniques*.

Create a new addon module called `library_loan_return_date` that depends on `my_module`. In this module, extend the `library.book.loan` model as follows:

```
class LibraryBookLoan(models.Model):
    _inherit = 'library.book.loan'
    expected_return_date = fields.Date('Due for', required=True)
```

Extend the `library.member` model as follows:

```
class LibraryMember(models.Model):
    _inherit = 'library.member'
    loan_duration = fields.Integer('Loan duration',
                                   default=15,
                                   required=True)
```

Since the `expected_return_date` field is required and no default is provided, the wizard to record loans will cease functioning because it does not provide a value for that field when it creates loans.

How to do it...

To extend the business logic in the `library.loan.wizard` model, you need to perform the following steps:

1. In `my_module`, modify the method `record_loans()` in the `LibraryLoanWizard` class. The original code is in the *Writing a wizard to guide the user* recipe in *Chapter 6, Advanced Server Side Development Techniques*. The new version is:

```
@api.multi
def record_loans(self):
    for wizard in self:
        books = wizard .book_ids
        loan = self.env['library.book.loan']
        for book in wizard.book_ids:
            values = self._prepare_loan(book)
            loan.create(values)
 @api.multi
 def _prepare_loan(self, book):
     return {'member_id': self.member_id.id,
             'book_id': book.id}
```

2. In `library_loan_return_date`, create a class which extends `library.loan.wizard` and defines the `_prepare_loan` method as follows:

```
from datetime import timedelta
from openerp import models, fields, api
class LibraryLoanWizard(models.TransientModel):
    _inherit = 'library.load.wizard'

    def _prepare_loan(self, book):
        values = super(LibraryLoanWizard,
                       self
                       )._prepare_loan(book)
        loan_duration = self.member_id.loan_duration
        today_str = fields.Date.context_today()
        today = fields.Date.from_string(today_str)
        expected = today + timedelta(days=loan_duration)
        values.update(
            {'expected_return_date':
                 fields.Date.to_string(expected)}
        )
        return values
```

How it works...

In step 1, we refactor the code from the *Writing a wizard to guide the user* recipe in *Chapter 6*, *Advanced Server Side Development Techniques*, to use a very common and useful coding pattern to create the `library.book.loan` records: the creation of the dictionary of values is extracted in a separate method rather than hardcoded in the method calling `create()`. This eases extending the addon module in case new values have to be passed at creation time, which is exactly the case we are facing.

Step 2 then does the extension of the business logic. We define a model that extends `library.loan.wizard` and redefines the `_prepare_loan()` method. The redefinition starts by calling the implementation from the parent class:

```
values = super(LibraryLoanWizard, self)._prepare_loan(book)
```

In the case of Odoo models, the parent class is not what you'd expect by looking at the Python class definition. The framework has dynamically generated a class hierarchy for our recordset, and the parent class is the definition of the model from the modules on which we depend. So the call to `super()` brings back the implementation of `library.loan.wizard` from `my_module`. In this implementation, `_prepare_loan()` returns a dictionary of values with `'member_id'` and `'book_id'`. We update this dictionary by adding the `'expected_return_date'` key before returning it.

There's more...

In this recipe, we chose to extend the behavior by calling the normal implementation and modifying the returned result *afterwards*. It is also possible to perform some actions *before* calling the normal implementation, and of course, we can also do both.

However, what we saw in this recipe is that it is harder to change the behavior of the middle of a method. We had to refactor the code to extract an extension point to a separate method and override this new method in the extension module.

You may be tempted to completely rewrite a method. Always be very cautious when doing so—if you do not call the `super()` implementation of your method, you are breaking the extension mechanism and potentially breaking addons for which their extension of the same method will never be called. Unless you are working in a controlled environment, in which you know exactly which addons are installed and you've checked that you are not breaking them, avoid doing this. And if you have to, be sure to document what you are doing in a very visible way.

What can you do before and after calling the original implementation of the method? Lots of things, including (but not limited to):

- Modifying the arguments that are passed to the original implementation (before)
- Modifying the context that is passed to the original implementation (before)
- Modifying the result that is returned by the original implementation (after)
- Calling another method (before, after)
- Creating records (before, after)
- Raising a `UserError` to cancel the execution in forbidden cases (before, after)
- Splitting `self` in smaller recordsets, and calling the original implementation on each of the subsets in a different way (before)

Extending write() and create()

The *Extending the business logic defined in a Model* recipe showed how to extend methods defined on a model class. If you think about it, methods defined on the parent class of the model are also part of the model. This means that all the base methods defined on `models.Model` (actually on `models.BaseModel`, which is the parent class of `models.Model`) are also available and can be extended.

This recipe shows how to extend `create()` and `write()` to control access to some fields of the records.

Getting ready

We will extend on the library example from the `my_module` addon module in *Chapter 3*, *Creating Odoo Modules*.

You will also need the security groups defined in *Chapter 10*, *Accessing Security* in the *Creating security Groups and assigning them to Users* recipe and the access rights defined in the *Adding security access to models* recipe from the same chapter.

Modify the file `security/ir.model.access.csv` to give write access to library users to books:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_
unlink
access_library_book_user,library.book.user,model_library_book,base.
group_user,1,1,0,0
access_library_book_admin,library.book.admin,model_library_book,base.
group_system,1,0,0,0
```

Add a field `manager_remarks` to the `library.book` model. We want only members of the `Library Managers` group to be able to write to that field:

```
class LibraryBook(models.Model):
    _name = 'library.book'
    manager_remarks = fields.Text('Manager Remarks')
```

How to do it...

In order to prevent users who are not members of the **Library Managers** group from modifying the value of `manager_remarks`, you need to perform the following steps:

1. Extend the `create()` method like this:

```
@api.model
@api.returns('self', lambda rec: rec.id)
def create(self, values):
    if not self.user_has_groups(
            'library.group_library_manager'):
        if 'manager_remarks' in values:
            raise exceptions.UserError(
                'You are not allowed to modify '
                'manager_remarks'
            )
    return super(LibraryBook, self).create(values)
```

2. Extend the `write()` method as follows:

```
@api.multi
def write(self, values):
    if not self.user_has_groups(
            'library.group_library_manager'):
        if 'manager_remarks' in values:
            raise exceptions.UserError(
                'You are not allowed to modify '
                'manager_remarks'
            )
    return super(LibraryBook, self).write(values)
```

3. Extend the `fields_get()` method as follows:

```
@api.model
def fields_get(self,
               allfields=None,
               write_access=True,
               attributes=None):
    fields = super(LibraryBook, self).fields_get(
```

```
            allfields=allfields,
                write_access=write_access,
                attributes=attributes
            )
            if not self.user_has_groups(
                    'library.group_library_manager'):
                if 'manager_remarks' in fields:
                    fields['manager_remarks']['readonly'] = True
```

How it works...

Step 1 redefines the `create()` method. It uses a decorator we have not seen so far, `@api.returns`.

This decorator maps the returned value from the new API to the old API, which is expected by the RPC protocol. In this case, the RPC calls to create expect the database `id` for the new record to be created, so we pass the `@api.returns` decorator an anonymous function, which fetches the `id` from the new record returned by our implementation. It is also needed if you want to extend the `copy()` method. Do not forget it when extending these methods if the base implementation uses the old API or you will crash with hard to interpret messages.

Before calling the base implementation of `create()`, our method uses the `user_has_groups()` method to check whether the user belongs to the group `library.group_library_manager` (this is the XML ID of the group). If it is not the case and a value is passed for `manager_remarks`, a `UserError` exception is raised preventing the creation of the record. This check is performed before the base implementation is called.

Step 2 does the same thing for the `write()` method; before writing, we check the group and the presence of the field in the values to write and raise `UserError` if there is a problem.

Step 3 is a small bonus; the `fields_get()` method is used by the web client to query for the fields of the model and their properties. It returns a Python dictionary mapping field names to a dictionary of field attributes, such as the `display` string or the `help` string. What interests us is the `readonly` attribute, which we force to `True` if the user is not a library manager. This will make the field read only in the web client, which will avoid unauthorized users from trying to edit it only to be faced with an error message.

Having the field set to read-only in the web client does not prevent RPC calls from writing it. This is why we extend `create()` and `write()`.

There's more...

When extending `write()`, note that before calling the `super()` implementation of `write()`, `self` is still unmodified. You can use this to compare the current values of the fields to the ones in the values dictionary.

In the recipe, we chose to raise an exception, but we could have also removed the offending field from the values dictionary and silently skipped updating that field in the record:

```
    @api.multi
    def write(self, values):
        if not self.user_has_groups(
                'library.group_library_manager'):
            if 'manager_remarks' in values:
                del values['manager_remarks']
        return super(LibraryBook, self).write(values)
```

After calling `super().write()`, if you want to perform additional actions, you have to be wary of anything that can cause another call to `write()`, or you will create an infinite recursion loop. The workaround is to put a marker in the context to be checked to break the recursion:

```
class MyModel(models.Model):
    @api.multi
    def write(self, values):
        super(MyModel, self).write(values)
        if self.env.context.get('MyModelLoopBreaker'):
            return
        self = self.with_context(MyModelLoopBreaker=True)
        self.compute_things() # can cause calls to writes
```

Customizing how records are searched

The *Defining the Model representation and order* recipe in *Chapter 3*, *Creating Odoo Modules* introduced the `name_get()` method, which is used to compute a representation of the record in various places, including in the widget used to display `Many2one` relations in the web client.

This recipe shows how to allow searching for a book in the `Many2one` widget by title, author, or ISBN by redefining `name_search`.

Getting ready

For this recipe, we will be using the following model definition:

```
class LibraryBook(models.Model):
    _name = 'library.book'
    name = fields.Char('Title')
    isbn = fields.Char('ISBN')
    author_ids = fields.Many2many('res.partner', 'Authors')

    @api.model
    def name_get(self):
        result = []
        for book in self:
            authors = book.author_ids.mapped('name')
            name = u'%s (%s)' % (book.title,
                                 u', '.join(authors))
            result.append((book.id, name))
    return result
```

When using this model, a book in a `Many2one` widget is displayed as **Book Title (Author1, Author2...)**. Users expect to be able to type in an author's name and find the list filtered according to this name, but this will not work as the default implementation of `name_search` only uses the attribute referred to by the `_rec_name` attribute of the model class, in our case, `'name'`. As a service to the advanced users, we also want to allow filtering by ISBN number.

How to do it...

In order to allow searching `library.book` either by book title, one of the authors, or ISBN, you need to define the `_name_search()` method in the `LibraryBook` class as follows:

```
    @api.model
    def _name_search(self, name='', args=None, operator='ilike',
                     limit=100, name_get_uid=None):
        args = [] if args is None else args.copy()
        if not(name == '' and operator == 'ilike'):
            args += ['|', '|',
                     ('name', operator, name),
                     ('isbn', operator, name),
                     ('author_ids.name', operator, name)
                     ]
        return super(LibraryBook, self)._name_search(
            name='', args=args, operator='ilike',
            limit=limit, name_get_uid=name_get_uid)
```

How it works...

The default implementation of `name_search()` actually only calls the `_name_search()` method, which does the real job. This `_name_search()` method has an additional argument, `name_get_uid`, which is used in some corner cases to compute the results using `sudo()`.

We pass most of the arguments that we receive unchanged to the `super()` implementation of the method:

- `name` is a string containing the value the user has typed so far
- `args` is either `None` or a search domain used as a prefilter for the possible records (it can come from the domain parameter of the `Many2one` relation for instance)
- `operator` is a string containing the match operator. Generally, you will have `'ilike'` or `'='`
- `limit` is the maximum number of rows to retrieve
- `name_get_uid` can be used to specify a different user when calling `name_get()` in the end to compute the strings to display in the widget

Our implementation of the method does the following:

1. It generates a new empty list if `args` is `None`, or makes a copy of `args` otherwise. We make a copy to avoid our modifications to the list having side effects on the caller.
2. Then, we check whether `name` is not an empty string or if `operator` is not `'ilike'`. This is to avoid generating a dumb domain `[('name', ilike, '')]` that does not filter anything. In that case, we jump straight to the call to the `super()` implementation.
3. If we have a `name`, or if the `operator` is not `'ilike'`, then we add some filtering criteria to `args`. In our case, we add clauses that will search for the supplied name in the title of the books, or in their ISBN, or in the authors' names.
4. Finally, we call the `super()` implementation with the modified domain in `args` and forcing `name` to `''` and `operator` to `ilike`. We do this to force the default implementation of `_name_search()` to not alter the domain it receives, so the one we specified is to be used.

There's more...

We mentioned in the introduction that this method is used in the `Many2one` widget. For completeness, it is also used in the following parts of Odoo:

- Proposals in the search widget
- When using the `in` operator on `One2many` and `Many2many` fields in the domain
- To search for records in the `many2many_tags` widget
- To search for records in the CSV file import

See also

The *Define the Model Representation and Order* recipe in *Chapter 3*, *Creating Odoo Modules* presents how to define the `name_get()` method, which is used to create a text representation of a record.

The *Defining filters on record lists: Domain* recipe in *Chapter 8*, *Backend Views* provides more information about search domain syntax.