6
Advanced Server Side Development Techniques

In this chapter, we will see how to:

- Change the user performing an action
- Call a method with a modified context
- Execute raw SQL queries
- Write a wizard to guide the user
- Define onchange methods
- Call onchange methods on the server side
- Port old API code to the new API

Introduction

In *Chapter 5*, *Basic Server Side Development*, we saw how to write methods on a model class, how to extend methods from inherited models, and how to work with recordsets. This chapter deals with more advanced topics such as working with the environment of a recordset and working with onchange methods.

Change the user performing an action

When writing business logic code, you may have to perform some actions with a different security context. A typical case is performing an action with the rights of the Administrator, who bypasses security checks.

This recipe shows how to let normal users modify the phone number of a company by using `sudo()`.

Getting ready

We will be working on records of the `res.company` model. By default, only members of the **Administration/Access Rights** user group can modify records of `res.company`, but in our case, we need to provide an access point to change only the phone number to users who are not necessarily members of that group.

How to do it...

In order to let normal users modify the phone number of a company, you need to perform the following steps:

1. Define a model extending the `res.company` model:

```
class ResCompany(models.Model):
    _inherit = 'res.company'
```

2. Add a method called `update_phone_number()`:

```
    @api.multi
    def update_phone_number(self, new_number):
```

3. In the method, ensure we are acting on a single record:

```
        self.ensure_one()
```

4. Modify the user of the environment:

```
        company_as_superuser = self.sudo()
```

5. Write the new phone number:

```
        company_as_superuser.phone = new_number
```

How it works...

In step 4, we call `self.sudo()`. This method returns a new recordset with a new **environment** in which the user is not the same as the one in `self`. When called without an argument, `sudo()` will link the Odoo **superuser**, Administrator, to the environment. All method calls via the returned recordset are made with the new environment, and therefore with superuser privileges.

If you need a specific user, you can pass either a recordset containing that user or the database `id` of the user. The following snippet allows you to search books that are visible, using the `public` user:

```
public_user = self.env.ref('base.public_user')
public_book = self.env['library.book'].sudo(public_user)
```

Caution when using sudo()

There is no traceability of the action; the author of the last modification of the company in our recipe will be Administrator, not the user originally calling `update_phone_number`.

The community addon, `base_suspend_security`, found at `https://github.com/OCA/server-tools/` can be used to work around this limitation.

There is more...

When using `sudo()` without an argument, you set the user of the context to the Odoo superuser. This superuser bypasses all the security rules of Odoo, both the **access control lists** and the **record rules**. By default, this user also has a `company_id` field set to the main company of the instance (the one with id `1`). This can be problematic in a **multi company** instance:

- If you are not careful, new records created in this environment will be linked to the company of the superuser
- If you are not careful, records searched in this environment may be linked to any company present in the database, which means that you may be leaking information to the real user, or worse, you may be silently corrupting the database by linking together records belonging to different companies

Using `sudo()` also involves creating a new `Environment` instance. This environment will have an initially empty recordset cache, and that cache will evolve independently from the cache of `self.env`. This can cause spurious database queries. In any case, you should avoid creating new environment inside loops, and try to move these environment creations to the outmost possible scope.

See also

- The *Obtain an empty recordset for a model* recipe in *Chapter 5*, *Basic Server Side Development*, explains what the environment is.

Call a method with a modified context

The context is part of the environment of a recordset. It is used to pass information such as the timezone or the language of the user from the user interface as well as contextual parameters specified in actions. A number of methods in the standard addons use the context to adapt their behavior to these values. It is sometimes necessary to modify the context on a recordset to get the desired results from a method call or the desired value for a computed field.

This recipe shows how to read the stock level for all `product.product` models in a given `stock.location`.

Getting ready

This recipe uses the `stock` and `product` addons. For our purposes, here is a simplified version of the `product.product` model:

```
class product.product(models.Model):
    _name = 'product.product'
    name = fields.Char('Name', required=True)
    qty_available = fields.Float('Quantity on Hand',
                                 compute='_product_available')
    def _product_available(self):
        """if context contains a key 'location' linked to a
        database id, then the stock available is computed within
        that location only. Otherwise the stock of all internal
        locations is computed"""
        pass  # read the real source in addons/stock/product.py :)
```

We intentionally don't provide the implementation of the computation and we skipped a few other keys that are looked for in the context in order to focus on the recipe.

How to do it...

In order to compute the stock levels in a given location for all the products, you need to perform the following steps:

1. Create a model class extending `product.product`:

```
class ProductProduct(models.Model):
    _inherit = 'product.product'
```

2. Add a method called `stock_in_location()`:

```
@api.model
def stock_in_location(self, location):
```

3. In the method, get a `product.product` recordset with a context modified as follows:

```
product_in_loc = self.with_context(
    location=location.id,
    active_test=False
)
```

4. Search all products:

```
all_products = product_in_loc.search([])
```

5. Create an array with the product name and stock level of all products present in the specified location:

```
stock_levels = []
for product in all_products:
    if product.qty_available:
        stock_levels.append((product.name,
                             product.qty_available))
return stock_levels
```

How it works...

Step 3 calls `self.with_context()` with some keyword arguments. This returns a new version of `self` (which is a `product.product` recordset) with the keys added to the current **context**. We are adding two keys:

- `location`: This one is mentioned in the docstring of the `product.product` method computing the `qty_available` field.
- `active_test`: When this key is present and linked to the `False` value, the `search()` method does not automatically add `('active', '=', True)` to the search domain. Using this ensures that in step 4, we get all products, including the disabled ones.

When we read the value of `product.qty_available` in step 5, the computation of that field is made using only the specified stock location.

There's more...

It is also possible to pass a dictionary to `self.with_context()`, in which case the dictionary is used as the new context, overwriting the current one. So step 3 could also have been written like this:

```
new_context = self.env.context.copy()
new_context.update({'location': location.id,
                    'active_test': False})
product_in_loc = self.with_context(new_context)
```

Using `with_context()` involves creating a new `Environment` instance. This **environment** will have an initially empty recordset cache, and that cache will evolve independently of the cache of `self.env`. This can cause spurious database queries. In any case, you should avoid creating new environments inside loops and try to move these environment creations to the outmost possible scope.

See also

- The *Obtain an empty recordset for a model* recipe in *Chapter 5*, *Basic Server Side Development*, explains what the environment is
- The *Passing parameters to forms and actions: Context* recipe in *Chapter 8*, *Backend Views*, explains how to modify the context in action definitions
- The *Search for records* recipe in *Chapter 5*, *Basic Server Side Development*, explains active records

Execute raw SQL queries

Most of the time, you can perform the operations you want using the `search()` method. However, sometimes, you need more—either you cannot express what you want using the domain syntax, for which some operations are tricky if not downright impossible, or your query requires several calls to `search()`, which ends up being inefficient.

This recipe shows you how to use raw SQL queries to read `res.partner` records grouped by country.

Getting ready

We will be using a simplified version of the `res.partner` model:

```
class ResPartner(models.Model):
    _name = 'res.partner'
    name = fields.Char('Name', required=True)
```

```
email = fields.Char('Email')
is_company = fields.Boolean('Is a company')
parent_id = fields.Many2one('res.partner', 'Related Company')
child_ids = fields.One2many('res.partner', 'parent_id',
                            'Contacts')
country_id = fields.Many2one('res.country', 'Country')
```

How to do it...

To write a method that returns a dictionary that contains the mapped names of countries to a recordset of all active partners from that country, you need to perform the following steps:

1. Write a class extending `res.partner`:

```
class ResPartner(models.Model):
    _inherit = 'res.partner'
```

2. Add a method called `partners_by_country()`:

```
    @api.model:
    def partners_by_country(self):
```

3. In the method, write the following SQL query:

```
        sql = ('SELECT country_id, array_agg(id) '
               'FROM res_partner '
               'WHERE active=true AND country_id IS NOT NULL '
               'GROUP BY country_id')
```

4. Execute the query:

```
        self.env.cr.execute(sql)
```

5. Iterate over the results of the query to populate the result dictionary:

```
        country_model = self.env['res.country']
        result = {}
        for country_id, partner_ids in self.env.cr.fetchall():
            country = country_model.browse(country_id)
            partners = self.search(
                [('id', 'in', tuple(partner_ids))]
            )
            result[country] = partners
        return result
```

How it works...

In step 3, we declare an SQL `SELECT` query. It uses the `id` field and the `country_id` foreign key, which refers to the `res_country` table. We use a `GROUP BY` statement so that the database does the grouping by `country_id` for us, and the `array_agg` aggregation function. This is a very useful PostgreSQL extension to SQL that puts all the values for the group in an array, which Python maps to a list.

Step 4 calls the `execute()` method on the database cursor stored in `self.env.cr`. This sends the query to PostgreSQL and executes it.

Step 5 uses the `fetchall()` method of the cursor to retrieve a list of rows selected by the query. From the form of the query we executed, we know that each row will have exactly two values, the first being `country_id` and the other one, the list of `ids` for the partners having that country. We loop over these rows and create recordsets from the values, which we store in the result dictionary.

There's more...

The object in `self.env.cr` is a thin wrapper around a `psycopg2` cursor. The following methods are the ones you will want to use most of the time:

- `execute(query, params)`: This executes the SQL `query` with the parameters marked as `%s` in the query substituted with the values in params, which is a tuple

Warning: never do the substitution yourself, as this can make the code vulnerable to SQL injections.

- `fetchone()`: This returns one row from the database, wrapped in a tuple (even if there is only one column selected by the query)
- `fetchall()`: This returns all the rows from the database as a list of tuples
- `fetchalldict()`: This returns all the rows from the database as a list of dictionaries mapping column names to values

Be very careful when dealing with raw SQL queries:

- You are bypassing all the security of the application. Be sure to call `search([('id', 'in', tuple(ids)])` with any list of `ids` you are retrieving to filter out records to which the user has no access to.
- Any modification you are making is bypassing the constraints set by the addon modules, except the `NOT NULL`, `UNIQUE`, and `FOREIGN KEY` constraints, which are enforced at the database level. So are any computed field recomputation triggers, so you may end up corrupting the database.

See also

- For access rights management, see *Chapter 10*, *Access Security*.

Write a wizard to guide the user

In the *Use Abstract Models for reusable Model features* recipe in *Chapter 4*, *Application Models*, the base class `models.TransientModel` was introduced; this class shares a lot with normal `Models` except that the records of **transient models** are periodically cleaned up in the database, hence the name *transient*. These are used to create **wizards** or dialog boxes, which are filled in the user interface by the users and generally used to perform actions on the persistent records of the database.

This recipe extends the code from *Chapter 3*, *Creating Odoo Modules,* by creating a wizard to record the borrowing of books by a library member.

Getting ready

If you want to follow the recipe, make sure you have the `my_module` addon module from *Chapter 3*, *Creating Odoo Modules*.

We will also use a simple model to record book loans:

```
class LibraryBookLoan(models.Model):
    _name = 'library.book.loan'
    book_id = fields.Many2one('library.book', 'Book',
                              required=True)
    member_id = fields.Many2one('library.member', 'Borrower',
                                required=True)
    state = fields.Selection([('ongoing', 'Ongoing'),
                              ('done', Done')],
                             'State',
                             default='ongoing', required=True)
```

How to do it...

To add a wizard for recording borrowed books to the addon module, you need to perform the following steps:

1. Add a new transient model to the module with the following definition:

```
class LibraryLoanWizard(models.TransientModel):
    _name = 'library.loan.wizard'
    member_id = fields.Many2one('library.member', 'Member')
    book_ids = fields.Many2many('library.book', 'Books')
```

2. Add the callback method performing the action on the transient model. Add the following code to the `LibraryLoanWizard` class:

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

3. Create a form view for the model. Add the following view definition to the module views:

```
<record id='library_loan_wizard_form' model='ir.ui.view'>
  <field name='name'>library loan wizard form view</field>
  <field name='model'>library.loan.wizard</field>
  <field name='arch' type='xml'>
    <form string="Borrow books">
      <sheet>
        <group>
          <field name='member_id'/>
        </group>
        <group>
          <field name='book_ids'/>
        </group>
      <sheet>
      <footer>
        <button name='record_loans'
                string='OK'
                class='btn-primary'
                type='object'/>
        or
        <button string='Cancel'
                class='btn-default'
                special='cancel'/>
      </footer>
    </form>
  </field>
</record>
```

4. Create an action and a menu entry to display the wizard. Add the following declarations to the module menu file:

```
<act_window id="action_wizard_loan_books"
            name="Record Loans"
```

```
            res_model="library.loan.wizard"
            view_mode="form"
            target="new"
            />
    <menuitem id="menu_wizard_loan_books"
              parent="library_book_menu"
              action="action_wizard_loan_books"
              sequence="20"
              />
```

How it works...

Step 1 defines a new model. It is no different from other models, apart from the base class, which is `TransientModel` instead of `Model`. Both `TransientModel` and `Model` share a common base class called `BaseModel`, and if you check the source code of Odoo, you will see that 99 percent of the work is in `BaseModel` and that both `Model` and `TransientModel` are almost empty.

The only things that change for `TransientModel` records are as follows:

- Records are periodically removed from the database so the tables for transient models don't grow up in size over time
- You cannot define access rules on `TransientModels`. Anyone is allowed to create a record, but only the user who created a record can read it and use it.
- You must not define `One2many` fields on a `TransientModel` that refer to a normal model, as this would add a column on the persistent model linking to transient data. Use `Many2many` relations in this case. You can of course define `Many2one` and `One2many` fields for relations between transient models.

We define two fields in the model, one to store the member borrowing the books and one to store the list of books being borrowed. We could add other scalar fields to record a scheduled return date, for instance.

Step 2 adds the code to the wizard class that will be called when the button defined in step 3 is clicked on. This code reads the values from the wizard and creates `library.book.loan` records for each book.

Step 3 defines a view for our wizard. Please refer to the *Document-style forms* recipe in *Chapter 8*, *Backend Views*, for details. The important point here is the button in the footer: the type attribute is set to `'object'`, which means that when the user clicks on the button, the method with the name specified by the name attribute of the button will be called.

Step 4 makes sure we have an entry point for our wizard in the menu of the application. We use `target='new'` in the action so that the form view is displayed as a dialog box over the current form. Please refer to the *Add a Menu Item and Window Action* recipe in *Chapter 8*, *Backend Views*, for details.

There's more...

Here are a few tips to enhance your wizards.

Using the context to compute default values

The wizard we are presenting requires the user to fill in the name of the member in the form. There is a feature of the web client we can use to save some typing. When an action is executed, the **context** is updated with some values that can be used by wizards:

Key	Value
`active_model`	This is the name of the model related to the action. This is generally the model being displayed on screen.
`active_id`	This indicates that a single record is active, and provides the ID of that record.
`active_ids`	If several records were selected, this will be a list with the IDs (this happens when several items are selected in a tree view when the action is triggered. In a form view, you get `[active_id]`).
`active_domain`	An additional domain on which the wizard will operate.

These values can be used to compute default values of the model, or even directly in the method called by the button. To improve on the recipe example, if we had a button displayed on the form view of a `library.member` model to launch the wizard, then the context of the creation of the wizard would contain {`'active_model': 'library.member', 'active_id': <member id>`}. In that case, you could define the `member_id` field to have a default value computed by the following method:

```
def _default_member(self):
    if self.context.get('active_model') == 'library.member':
        return self.context.get('active_id', False)
```

Wizards and code reuse

In step 2, we could have dispensed with the `for wizard in self` loop, and assumed that `len(self)` is `1`, possibly adding a call to `self.ensure_one()` at the beginning of the method, like this:

```
@api.multi
def record_borrows(self):
    self.ensure_one()
```

```
member = self.member_id
books = self.book_ids
member.borrow_books(books)
```

We recommend using the version in the recipe though, because it allows reusing the wizard from other parts of the code by creating records for the wizard, putting them in a single recordset (see the *Combine recordsets* recipe in *Chapter 5*, *Basic Server Side Development*, to see how to do this) and then calling `record_loans()` on the recordset. Granted, here the code is trivial and you don't really need to jump through all those hoops to record that some books were borrowed by different members. However, in an Odoo instance, some operations are much more complex, and it is always nice to have a wizard available that does "the right thing." When using such wizards, be sure to check the source code for any possible use of the `active_model` / `active_id` / `active_ids` keys from the context, in which case, you need to pass a custom context (see the *Call a method with a modified context* recipe previously for how to do this).

Redirecting the user

The method in step 2 does not return anything. This will cause the wizard dialog to be closed after the action is performed. Another possibility is to have the method return a dictionary with the fields of an `ir.action`. In this case, the web client will process the action as if a menu entry had been clicked on by the user. For instance, if we wanted to display the form view of the member who has just borrowed the books, we could have written the following:

```
@api.multi
def record_borrows(self):
    for wizard in self:
        member = wizard.member_id
        books = wizard.book_ids
        member.borrow_books(books)
    member_ids = self.mapped('member_id').ids
    action = {
        'type': 'ir.action.act_window',
        'name': 'Borrower',
        'res_model': 'library.member',
        'domain': [('id', '=', member_ids)],
        'view_mode': 'form,tree',
    }
    return action
```

This builds a list of members who has borrowed books from this wizard (in practice, there will only be one such member, when the wizard is called from the user interface) and creates a dynamic action, which displays the members with the specified IDs.

This trick can be extended by having a wizard (with several steps to be performed one after the other), or depending on some condition from the preceding steps, by providing a **Next** button that calls a method defined on the wizard. This method will perform the step (maybe using a hidden field and storing the current step number), update some fields on the wizard, and return an action that will redisplay the same updated wizard and get ready for the next step.

Define onchange methods

When writing Odoo models, it is often the case that some fields are interrelated. We have seen how to specify constraints between fields in the *Add constraint validations to a Model* recipe in *Chapter 4*, *Application Models*. This recipe illustrates a slightly different concept—**onchange methods** are called when a field is modified in the user interface to update the values of other fields of the record in the web client, usually in a form view.

We will illustrate this by providing a wizard similar to the one defined in the preceding recipe, *Write a wizard to guide the user*, but which can be used to record loan returns. When the member is set on the wizard, the list of books is updated to the books currently borrowed by the member. While we are demonstrating onchange methods on a `TransientModel`, these features are also available on normal `Models`.

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 *Write a wizard to guide the user* recipe's changes applied.

You will also want to prepare your work by defining the following transient model for the wizard:

```
class LibraryReturnsWizard(models.TransientModel):
    _name = 'library.returns.wizard'
    member_id = fields.Many2one('library.member', 'Member')
    book_ids = fields.Many2many('library.book', 'Books')
    @api.multi
    def record_returns(self):
        loan = self.env['library.book.loan']
        for rec in self:
            loans = loan.search(
                [('state', '=', 'ongoing'),
                 ('book_id', 'in', rec.book_ids.ids),
                 ('member_id', '=', rec.member_id.id)]
            )
            loans.write({'state': 'done'})
```

Finally, you will need to define a view, an action, and a menu entry for the wizard. This is left as an exercise.

How to do it...

To automatically populate the list of books to return when the user is changed, you need to add an onchange method in the `LibraryReturnsWizard` step with the following definition:

```
@api.onchange('member_id')
def onchange_member(self):
    loan = self.env['library.book.loan']
    loans = loan.search(
        [('state', '=', 'ongoing'),
         ('member_id', '=', self.member_id.id)]
    )
    self.book_ids = loans.mapped('book_id')
```

How it works...

An onchange method uses the `@api.onchange` decorator, which is passed the names of the fields that change and thus will trigger the call to the method. In our case, we say that whenever `member_id` is modified in the user interface, the method must be called.

In the body of the method, we search the books currently borrowed by the member, and we use an attribute assignment to update the `book_ids` attribute of the wizard.

The `@api.onchange` decorator takes care of modifying the view sent to the web client to add an `on_change` attribute to the field. This used to be a manual operation in the "old API".

There's more...

The basic use of onchange methods is to compute new values for fields when some other fields are changed in the user interface, as we've seen in the recipe.

Inside the body of the method, you get access to the fields displayed in the current view of the record, but not necessarily all the fields of the model. This is because onchange methods can be called while the record is being created in the user interface *before* it is stored in the database! Inside an onchange method, `self` is in a special state, denoted by the fact that `self.id` is not an integer, but an instance of `openerp.models.NewId`. Therefore, you must not make any changes to the database in an onchange method, because the user may end up canceling the creation of the record, which would not roll back any changes made by the onchanges called during the edition. To check for this, you can use `self.env.in_onchange()` and `self.env.in_draft()`—the former returns `True` if the current context of execution is an onchange method and the latter returns `True` if `self` is not yet committed to the database.

Additionally, onchange methods can return a Python dictionary. This dictionary can have the following keys:

- `warning`: The value must be another dictionary with the keys `title` and `message` respectively containing the title and the content of a dialog box, which will be displayed when the onchange method is run. This is useful for drawing the attention of the user to inconsistencies or to potential problems.
- `domain`: The value must be another dictionary mapping field names to domains. This is useful when you want to change the domain of a `One2many` field depending on the value of another field.

For instance, suppose we have a fixed value set for `expected_return_date` in our `library.book.loan` model, and we want to display a warning when a member has some books that are late. We also want to restrict the choice of books to the ones currently borrowed by the user. We can rewrite the onchange method as follows:

```
@api.onchange('member_id')
def onchange_member(self):
    loan = self.env['library.book.loan']
    loans = loan.search(
        [('state', '=', 'ongoing'),
         ('member_id', '=', self.member_id.id)]
    )
    self.book_ids = loans.mapped('book_id')
    result = {
        'domain': {'book_ids': [
                       ('id', 'in', self.book_ids.ids)]
                   }
    }
    late_domain = [
        ('id', 'in', loans.ids),
        ('expected_return_date', '<', fields.Date.today())
    ]
    late_loans = loans.search(late_domain)
    if late_loans:
        message = ('Warn the member that the following '
                   'books are late:\n')
        titles = late_loans.mapped('book_id.name')
        result['warning'] = {
            'title': 'Late books',
            'message': message + '\n'.join(titles)
        }
    return result
```

Call onchange methods on the server side

The *Create new records* and *Update values of a recordset record* recipes in *Chapter 5*, *Basic Server Side Development*, mentioned that these operations did not call onchange methods automatically. Yet in a number of cases it is important that these operations are called because they update important fields in the created or updated record. Of course, you can do the required computation yourself, but this is not always possible as the onchange method can be added or modified by a third-party addon module installed on the instance that you don't know about.

This recipe explains how to call the onchange methods on a record by manually playing the onchange method before creating a record.

Getting ready

We will reuse the settings from the preceding recipe, *Write onchange methods*. The action will take place in a new method of `library.member` called `return_all_books(self)`.

How to do it...

In this recipe, we will manually create a record of the `library.returns.wizard` model, and we want the onchange method to compute the returned books for us. To do this, you need to perform the following steps:

1. Create the method `return_all_books` in the `LibraryMember` class:

```
@api.multi
def return_all_books(self):
    self.ensure_one
```

2. Get an empty recordset for `library.returns.wizard`:

```
wizard = self.env['library.returns.wizard']
```

3. Prepare the values to create a new wizard record:

```
values = {'member_id': self.id, book_ids=False}
```

4. Retrieve the onchange specifications for the wizard:

```
specs = wizard._onchange_spec()
```

5. Get the result of the onchange method:

```
updates = wizard.onchange(values, ['member_id'], specs)
```

6. Merge these results with the values of the new wizard:

```
value = updates.get('value', {})
for name, val in value.iteritems():
```

```
            if isinstance(val, tuple):
                value[name] = val[0]
        values.update(value)
```

7. Create the wizard:

```
        record = wizard.create(values)
```

How it works...

For an explanation of step 1 to step 3, please refer to the recipe *Create new records* in *Chapter 5*, *Basic Server Side Development*.

Step 4 calls the `_onchange_spec` method on the model, passing no argument. This method will retrieve the updates that are triggered by the modification of which other field. It does this by examining the form view of the model (remember, onchange methods are normally called by the web client).

Step 5 calls the `onchange(values, field_name, field_onchange)` method of the model with 3 arguments:

- `values`: The list of values we want to set on the record. You need to provide a value for all the fields you expect to be modified by the onchange method. In the recipe, we set `book_ids` to `False` for this reason.
- `field_name`: A list of fields for which we want to trigger the onchange methods. You can pass an empty list, and it will use the fields defined in values. However, you will often want to specify that list manually to control the order of evaluation, in case different fields can update a common field.
- `field_onchange`: The onchange specifications that were computed in step 4. This method finds out which onchange methods must be called and in what order and returns a dictionary, which can contain the following keys:
  - `value`: This is a dictionary of newly computed field values. This dictionary only features keys that are in the `values` parameter passed to `onchange()`. Note that `Many2one` fields are mapped to a tuple containing `(id, display_name)` as an optimization for the web client.
  - `warning`: This is a dictionary containing a warning message that the web client will display to the user.
  - `domain`: This is a dictionary mapping field names to new validity domains.

Generally, when manually playing onchange methods, we only care about what is in `value`.

Step 6 updates our initial values dictionary with the values computed by the onchange. We process the values corresponding to `Many2one` fields to only keep the `id`. To do so, we take advantage of the fact that these fields are only those whose value is returned as a tuple.

Step 7 finally creates the record.

There's more...

If you need to call an onchange method after modifying a field, the code is the same. You just need to get a dictionary for the values of the record, which can be obtained by using `values = dict(record._cache)` after modifying the field.

See also

- The *Create new records* and *Update values of recordset records* recipes in *Chapter 5*, *Basic Server Side Development*

Port old API code to the new API

Odoo has a long history and the so-called "traditional" or "old" API has been in use for a very long time. When designing the "new" API in Odoo 8.0, time was taken to ensure that the APIs would be able to coexist, because it was foreseen that porting the huge codebase to the new API would be a huge effort. So you will probably come across addon modules using the traditional API. When migrating them to the current version of Odoo, you may want to port them to the new API.

This recipe explains how to perform this translation. It can also serve as an aide memoire when you need to extend a module that uses the traditional API with the new API.

Getting ready

Let's port the following addon module code developed with the traditional API:

```
from datetime import date, timedelta
from openerp.osv import orm, fields
from openerp.tools import _, DEFAULT_SERVER_DATE_FORMAT as DATE_FMT

class library_book(orm.Model):
    _name = 'library.book'
    _columns = {
        'name': fields.char('Name', required=True),
```

```
            'author_ids': field.many2many('res.partner'),
        }

    class library_member(orm.Model):
        _name = 'library.member'
        _inherits = {'res.partner': 'partner_id'}

        _columns = {
            'partner_id': fields.many2one('res.partner',
                                          'Partner',
                                          required=True),
            'loan_duration': fields.integer('Loan duration',
                                            required=True),
            'date_start': fields.date('Member since'),
            'date_end': fields.date('Expiration date'),
            'number': fields.char('Number', required=True),
        }

        _defaults = {
            'loan_duration': 10,
        }

        def on_change_date_end(self, cr, uid, date_end, context=None):
            date_end = date.strptime(date_end, DATE_FMT)
            today = date.today()
            if date_end <= today:
                return {
                    'value': {'loan_duration': 0},
                    'warning': {
                        'title': 'expired membership',
                        'message': "This member's membership " \
                                   "has expired",
                    },
                }

        def borrow_books(self, cr, uid, ids, book_ids, context=None):
            if len(ids) != 1:
                raise orm.except_orm(
                    _('Error!'),
                    _('It is forbidden to loan the same books '
                      'to multiple members.'))
            loan_obj = self.pool['library.book.loan']
            member = self.browse(cr, uid, ids[0], context=context)
            for book_id in book_ids:
```

```
            val = self._prepare_loan(
                cr, uid, member, book_id, context=context
            )
            loan_id = loan_obj.create(cr, uid, val,
               context=context)

    def _prepare_loan(self, cr, uid,
                      member, book_id,
                      context=None):
        return {'book_id': book_id,
                'member_id': member.id,
                'duration': member.loan_duration}


class library_book_loan(orm.Model):
    _name = 'library.book.loan'

    def _compute_date_due(self, cr, uid, ids,
                          fields, arg, context=None):
        res = {}
        for loan in self.browse(cr, uid, ids, context=context):
            start_date = date.strptime(loan.date, DATE_FMT)
            due_date = start_date + timedelta(days=loan.duration)
            res[loan.id] = due_date.strftime(DATE_FMT)
        return res

    _columns = {
        'book_id': fields.many2one('library.book', required=True),
        'member_id': fields.many2one('library.member',
                                     required=True),
        'state': fields.selection([('ongoing', 'Ongoing'),
                                   ('done', 'Done')],
                                  'State', required=True),
        'date': fields.date('Loan date', required=True),
        'duration': fields.integer('Duration'),
        'date_due': fields.function(
            fnct=_compute_date_due,
            type='date',
            store=True,
            string='Due for'),
    }

    def _default_date(self, cr, uid, context=None):
```

```
        return date.today().strftime(DATE_FMT)

    _defaults = {
        'duration': 15,
        'date': _default_date,
    }
```

The module, of course, defined some views. Most of them will need no change, so we won't show them. The only one relevant in this recipe is the `library.member` form view. Here is the relevant excerpt for that view:

```
<record id='library.member_form_view' model='ir.ui.view'>
  <field name='model'>library.member</field>
  <field name='arch' type='xml'>
    <!-- [...] -->
    <field name='date_end'
           on_change='on_change_date_end(date_end)'/>
    <!-- [...] -->
  </field>
</record>
```

How to do it...

To port this module code to the new API, you need to perform the following steps:

1. Copy the original source file to make a backup.
2. In the new file, modify the module imports as follows:

```
from datetime import date, timedelta
from openerp import models, fields, api, exceptions
from openerp.tools import _
```

3. Modify the class definitions to the new API base classes, and rename the classes to use CamelCase:

```
class LibraryBook(models.Model):
    _name = 'library.book'

class LibraryMember(models.Model):
     _name = 'library.member'
    _inherits = {'res.partner': 'partner_id'}

class LibraryBookLoan(models.Model):
     _name = 'library.book.loan'
```

4. Migrate the `_columns` definitions to attribute declaration of fields in the `LibraryBook` class:
   - The `_columns` keys become class attributes
   - The names of the field types get capitalized

```
class LibraryBook(models.Model):
    _name = 'library.book'

    name = fields.Char('Name', required=True)
    author_ids = field.Many2many('res.partner')
```

Don't forget to remove the commas at the ends of the lines.

5. Do the same thing for `LibraryMember`, taking care to move `_defaults` declarations to the field definition:

```
class LibraryMember(models.Model):
    # [...]

    partner_id = fields.Many2one('res.partner',
                                 'Partner',
                                 required=True)
    loan_duration = fields.Integer('Loan duration',
                                   default=10,
                                   required=True)
    date_start = fields.Date('Member since')
    date_end = fields.Date('Expiration date')
    number = fields.Char('Number', required=True)
```

6. Migrate the column's definition of the `LibraryBookLoan` class, changing the type of the `function` field to `fields.Date`:

```
class LibraryBookLoan(models.Model):
    # [...]

    book_id = fields.Many2one('library.book', required=True)
    member_id = fields.Many2one('library.member',
                                required=True)
    state = fields.Selection([('ongoing', 'Ongoing'),
                              ('done', 'Done')],
                             'State', required=True)
    date = fields.Date('Loan date', required=True,
                       default=_default_date)
```

```
    duration = fields.Integer('Duration', default=15)
    date_due = fields.Date(
        compute='_compute_date_due',
        store=True,
        string='Due for'
       )
```

7. In the `LibraryBookLoan` class, migrate the definition of the `_compute_date_due` function to the new API:
   - Remove all the arguments except `self`
   - Add an `@api.depends` decorator
   - Change the body of the method to use the new computed field protocol (see the *Add computed fields to a Model* recipe in *Chapter 4*, *Application Models*, for details):

```
# in class LibraryBookLoan
    @api.depends('start_date', 'due_date')
    def _compute_date_due(self):
        for loan in self:
            start_date = fields.Date.from_string(loan.date)
            due_date = start_date + timedelta(days=loan.duration)
            loan.date_due = fields.Date.to_string(due_date)
```

8. In the `LibraryBookLoan` class, migrate the definition of the `_default_date` function:
   - The function definition must be moved before the field declaration
   - The new prototype only has one argument, `self` (see the *Add data fields to a Model* recipe in *Chapter 4*, *Application Models*, for details):

```
# in class LibraryBookLoan, before the fields definitions
    def _default_date(self):
        return fields.Date.today()
```

9. Rewrite the `borrow_books` and `_prepare_loan` methods in the `LibraryMember` class:
   - Add an `@api.multi` decorator
   - Remove the arguments `cr`, `uid`, `ids`, `context`
   - Port the code to the new API (see the various recipes in this chapter for details)
   - Replace the `orm.except_orm` exception with `UserError`:

```
# in class LibraryMember
    @api.multi
    def borrow_books(self, book_ids):
```

```
            if len(self) != 1:
                raise exceptions.UserError(
                    _('It is forbidden to loan the same books '
                      'to multiple members.')
                )
            loan_model = self.env['library.book.loan']
            for book in self.env['library.book'].browse(book_ids):
                val = self._prepare_loan(book)
                loan = loan_model.create(val)
        @api.multi
        def _prepare_loan(self, book)
            self.ensure_one()
            return {'book_id': book.id,
                    'member_id': self.id,
                    'duration': self.loan_duration}
```

10. Port the `on_change_date_end` method in the `LibraryMember` class:
    - Add an `@api.onchange` decorator
    - Port the code to the new API (see the *Write onchange methods* recipe in this chapter for details):

```
# in LibraryMember
    @api.onchange('date_end')
    def on_change_date_end(self):
        date_end = fields.Date.from_string(self.date_end)
        today = date.today()
        if date_end <= today:
            self.loan_duration = 0
            return {
                'warning': {
                    'title': 'expired membership',
                    'message': "Membership has expired",
                },
            }
```

11. Edit the `library.member` form's view definition and remove the `on_change` attribute in the `date_end` field:

```
<record id='library.member_form_view' model='ir.ui.view'>
  <field name='model'>library.member</field>
  <field name='arch' type='xml'>
    <!-- [...] -->
    <field name='date_end'/>
    <!-- [...] -->
  </field>
</record>
```

How it works

Step 1 changes the imports. The old API lives in the `openerp.osv` package, which we change to use `openerp.models`, `openerp.fields`, `openerp.api`, and `openerp.exceptions`. We also drop the import of `DEFAULT_SERVER_DATE_FORMAT`, because we will use the helper methods on `fields.Date` to perform `datetime` / string conversions.

Step 2 changes the base classes of the models. Depending on the age of the code you are migrating, you may need to replace the following:

- `osv.osv`, `osv.Model` or `orm.Model` with `models.Model`
- `osv.osv_memory`, `osv.TransientModel`, or `orm.TransientModel` with `models.TransientModel`

The usual class attributes such as `_name`, `_order`, `_inherit`, `_inherits`, and so on are unchanged.

Steps 3 to 8 deal with the migration of the field's definitions. The `_columns` dictionaries mapping field names to field definitions in the old API are migrated to class attributes.

When doing so, don't forget to remove the commas following each field definition in the dictionary, otherwise the attribute value will be a 1-element tuple, and you will get errors. You also get a warning log line for these.

The field classes in `openerp.fields` usually have the same name as their counterparts in `openerp.osv.fields`, but capitalized (`openerp.osv.fields.char` becomes `openerp.fields.Char`). Default values are moved from the `_defaults` class attribute to a default parameter in the type declaration:

```
_columns = {
    'field1': fields.char('Field1'),
    'field2': fields.integer('Field2'),
}
_defaults = {
    'field2': 42,
}
```

Now the preceding code is modified to the following:

```
field1 = fields.Char('Field1')
field2 = fields.Integer('Field2', default=42)
```

The biggest change is for function fields. The old API uses `fields.function` defined with a `type` parameter giving the type of the column. The new API uses a field of the expected type, defined with a `compute` parameter, which gives the method used to compute the field:

```
_columns = {
    'field3': fields.function(
        _compute_field3,
        arg=None,
        fnct_inv=_store_field3,
        fnct_inv_arg=None,
        type='char',
        fnct_search=_search_field3,
        store=trigger_dict,
        multi=keyword
    ),
}
```

Now the preceding code is modified to the following:

```
field3 = fields.Char('Field1'
                     compute='_compute_field3',
                     inverse='_store_field3',
                     search='_search_field3',
                     store=boolean)
```

The `trigger_dict` in the old API is replaced with `@api.depends` decorators, and there is no need for the `multi` parameter in the new API, as the `compute` method can update several fields sharing the same compute parameter value. See the *Add Computed Fields to a Model* recipe in *Chapter 4*, *Application Models*, for more detailed information.

Step 9 migrates the business logic methods. The decorator to use on the method defines the parameter conversions, which are to be used to map the arguments. You need to choose them carefully because this can break modules using the old API extending the ported module. The same advice is valid if you need to extend a model defined using the old API with the new API. Here are the most common cases, with `<args>` denoting additional arbitrary arguments and keyword arguments:

Old API prototype	New API prototype
`def m(self, cr, uid, ids, <args>, context=None)`	`@api.multi` `def m(self, <args>)`
`def m(self, cr, uid, <args>, context=None)`	`@api.model` `def m(self, <args>)`
`def m(self, cr, <args>)`	`@api.cr` `def m(self, <args>)`

Old API prototype	New API prototype
`def m(self, cr, uid, <args>)`	`@api.cr_uid` `def m(self, <args>)`

Finally, steps 10 and 11 migrate the onchange method. Things have changed quite a bit between the two versions of the API; in the traditional API, onchange methods are declared in the view definitions by using an attribute `on_change` on the `<field>` element with a value describing the call to be performed when the field is edited. In the new API, this declaration is not necessary because the framework dynamically generates it in the view by analyzing the methods decorated with `@api.onchange`.