

3

Creating Odoo Modules

In this chapter, we will cover the following topics:

- ▶ Creating and installing a new addon module
- ▶ Completing the addon module manifest
- ▶ Organizing the addon module file structure
- ▶ Adding Models
- ▶ Adding Menu Items and Views
- ▶ Adding Access Security
- ▶ Using scaffold to create a module

Introduction

All these components that we just mentioned will be addressed in detail in later chapters.

Now that we have a development environment and know how to manage Odoo server instances and databases, you can learn how to create Odoo addon modules.

Our main goal here is to understand how an addon module is structured and the typical incremental workflow to add components to it.

For this chapter, you are expected to have Odoo installed and to follow the recipes in *Chapter 1, Installing the Odoo Development Environment*. You are also expected to be comfortable in discovering and installing extra addon modules, as described in the *Chapter 2, Managing Odoo Server Instances*, recipes.

Creating and installing a new addon module

In this recipe, we will create a new module, make it available in our Odoo instance, and install it.

Getting ready

We will need an Odoo instance ready to use.

If the first recipe in *Chapter 1, Installing the Odoo Development Environment*, was followed, Odoo should be available at `~/odoo-dev/odoo`. For explanation purposes, we will assume this location for Odoo, although any other location of your preference could be used.

We will also need a location for our Odoo modules. For the purpose of this recipe, we will use a `local-addons` directory alongside the `odoo` directory, at `~/odoo-dev/local-addons`.

How to do it...


The following steps will create and install a new addon module:

1. Change the working directory in which we will work and create the addons directory where our custom module will be placed:

```
$ cd ~/odoo-dev
$ mkdir local-addons
```

2. Choose a technical name for the new module and create a directory with that name for the module. For our example we will use `my_module`:

```
$ mkdir local-addons/my_module
```

 A module's technical name must be a valid Python identifier; it must begin with a letter, and only contain letters (preferably lowercase), numbers, and underscore characters.

3. Make the Python module importable by adding an `__init__.py` file:

```
$ touch local-addons/my_module/__init__.py
```

4. Add a minimal module manifest, for Odoo to detect it. Create an `__openerp__.py` file with this line:

```
{'name': 'My module'}
```

5. Start your Odoo instance including our module directory in the addons path:

```
$ odoo/odoo.py --addons-path=odoo/addons/,local-addons/
```



If the `--save` option is added to the Odoo command, the addons path will be saved in the configuration file. Next time you start the server, if no addons path option is provided, this will be used.

6. Make the new module available in your Odoo instance; log in to Odoo using `admin`, enable the **Developer Mode** in the **About** box, and in the Apps top menu select **Update Apps List**. Now Odoo should know about our Odoo module.
7. Select the **Apps** menu at the top and, in the search bar in the top right, delete the default **Apps filter** and search for `my_module`. Click on its **Install** button and the installation will be concluded.

How it works...

An Odoo module is a directory containing code files and other assets. The directory name used is the module's technical name. The `name` key in the module manifest is its title.

The `__openerp__.py` file is the module manifest. It contains a Python dictionary with information about the module, the modules it depends on, and the data files that it will load.

In the example, a minimal manifest file was used, but in real modules, we will want to add a few other important keys. These are discussed in the next recipe, *Completing the module manifest*.

The module directory must be Python-importable, so it also needs to have an `__init__.py` file, even if it's empty. To load a module, the Odoo server will import it. This will cause the code in the `__init__.py` file to be executed, so it works as an entry point to run the module Python code. Because of this, it will usually contain import statements to load the module Python files and submodules.

Modules can be installed directly from the command line using the `--init`, or `-i`, option. In the past, we had to use the **Update Module List** to make it available to the Odoo instance. However, at this moment, this is done automatically when the `--init` or `--update` are used from the command line.

Completing the addon module manifest

The manifest is an important piece for Odoo modules. It contains important information about it and declares the data files that should be loaded.

Getting ready

We should have a module to work with, already containing an `__openerp__.py` manifest file. You may want to follow the previous recipe to provide such a module to work with.

How to do it...

We will add a manifest file and an icon to our addon module:

1. To create a manifest file with the most relevant keys, edit the module `__openerp__.py` file to look like this:

```
# -*- coding: utf-8 -*-
{
    'name': "Title",
    'summary': "Short subtitle phrase",
    'description': """"Long description""",
    'author': "Your name",
    'license': "AGPL-3",
    'website': "http://www.example.com",
    'category': 'Uncategorized',
    'version': '9.0.1.0.0',
    'depends': ['base'],
    'data': ['views.xml'],
    'demo': ['demo.xml'],
}
```

2. To add an icon for the module, choose a PNG image to use and copy it to `static/description/icon.png`.

How it works...

The first line, containing `coding: utf-8`, is necessary for Python to process the file content as Unicode UTF-8. This way, we can use non-ASCII characters in the manifest.

The remaining content is a regular Python dictionary, with keys and values. The example manifest we used contains the most relevant keys:

- ▶ `name`: This is the title for the module.
- ▶ `summary`: This is the subtitle with a one-line description.
- ▶ `description`: This is a long description, written in plain text or **ReStructuredText (RST)** format. It is usually surrounded by triple quotes, used in Python to delimit multiline texts. For an RST quickstart reference, visit <http://docutils.sourceforge.net/docs/user/rst/quickstart.html>.

- ▶ **author:** This is a string with the name of the authors. When there is more than one, it is common practice to use a comma to separate their names, but note that it still should be a string, not a Python list.
- ▶ **license:** This is the identifier for the license under which the module is made available. It is limited to a predefined list, and the most frequent option is `AGPL-3`. Other possibilities include `LGPL-3`, `Other OSI approved license`, and `Other proprietary`.
- ▶ **website:** This is a URL people should visit to know more about the module or the authors.
- ▶ **category:** This is used to organize modules in areas of interest. The list of the standard category names available can be seen at https://github.com/odoo/odoo/blob/master/openerp/addons/base/module/module_data.xml. But it's also possible to define other new category names here.
- ▶ **version:** This is the modules' version numbers. It can be used by the Odoo app store to detect newer versions for installed modules. If the version number does not begin with the Odoo target version (for example, `8.0`), it will be automatically added. Nevertheless, it will be more informative if you explicitly state the Odoo target version, for example, using `8.0.1.0.0` or `8.0.1.0` instead of `1.0.0` or `1.0`.
- ▶ **depends:** This is a list with the technical names of the modules it directly depends on. If none, we should at least depend on the `base` module. Don't forget to include any module defining XML IDs, Views, or Models referenced by this module. That will ensure that they all load in the correct order, avoiding hard-to-debug errors.
- ▶ **data:** This is a list of relative paths to the data files to load with module installation or upgrade. The paths are relative to the module root directory. Usually, these are XML and CSV files, but it's also possible to have YAML data files. These are discussed in depth in *Chapter 9, Module Data*.
- ▶ **demo:** This is the list of relative paths to the files with demonstration data to load. These will only be loaded if the database was created with the **Demonstration Data** flag enabled.

The image that is used as the module icon is the PNG file at `static/description/icon.png`.



Odoo is expected to have significant changes between major versions, so modules built for one major version are likely to not be compatible with the next version without conversion and migration work. Because of this, it's important to be sure about a module's Odoo target version before installing it.

There's more

Instead of having the long description in the module manifest, it's possible to have it in its own file. Since version 8.0, it can be replaced by a README file, with either a `.txt`, `.rst`, or an `.md` (Markdown) extension. Otherwise, include a `description/index.html` file in the module.

This HTML description will override a description defined in the manifest file.

There are a few more keys that are frequently used:

- ▶ `application`: If this is `True`, the module is listed as an application. Usually, this is used for the central module of a functional area.
- ▶ `auto_install`: If this is `True`, it indicates that this is a "glue" module, which is automatically installed when all its dependencies are installed.
- ▶ `installable`: If this is `True` (the default value), it indicates that the module is available for installation.

Organizing the addon module file structure

An addon module contains code files and other assets such as XML files and images. For most of these files, we are free to choose where to place them inside the module directory.

However, Odoo uses some conventions on the module structure, so it is advisable to follow them.

Getting ready

We are expected to have an addon module directory with only the `__init__.py` and `__openerp__.py` files.

How to do it...

To create the basic skeleton for the addon module:

1. Create the directories for code files:

```
$ cd path/to/my-module
$ mkdir models
$ touch models/__init__.py
$ mkdir controllers
```

```

$ touch controllers/__init__.py
$ mkdir views
$ mkdir security
$ mkdir data
$ mkdir demo
$ mkdir i18n
$ mkdir -p static/description

```

2. Edit the module's top `__init__.py` file so that the code in subdirectories is loaded:

```

# -*- coding: utf-8 -*-

from . import models
from . import controllers

```

This should get us started with a structure containing the most used directories, similar to this one:

```

.
├── __init__.py
├── openerp__.py
├──
├── controllers
│   └── __init__.py
├── data
├── i18n
├── models
│   └── __init__.py
├── security
├── static
│   └── description
└── views

```

How it works...

To provide some context, an Odoo addon module can have three types of file:

- ▶ The **Python code** is loaded by the `__init__.py` files, where the `.py` files and code subdirectories are imported. Subdirectories containing code Python, in turn, need their own `__init__.py`
- ▶ **Data files** that are to be declared in the `data` and `demo` keys of the `__openerp__.py` module manifest in order to be loaded. These are usually XML and CSV files for the user interface, fixture data, and demonstration data.
- ▶ **Web assets** such as JavaScript code and libraries, CSS, and QWeb/HTML templates also play an important part. There are declared through an XML file extending the master templates to add these assets to the web client or website pages.

The addon files are to be organized in these directories:

- ▶ `models/` contains the backend code files, creating the Models and their business logic. A file per Model is recommended, with the same name as the model, for example, `library_book.py` for the `library.book` model. These are addressed in depth in *Chapter 4, Application Models*.
- ▶ `views/` contains the XML files for the user interface, with the actions, forms, lists, and so on. As with models, it is advised to have one file per model. Filenames for website templates are expected to end with the `_template` suffix. Backend Views are explained in *Chapter 8, Backend Views*, and website Views are addressed in *Chapter 14, CMS Web Site Development*.
- ▶ `data/` contains other data files with module initial data. Data files are explained in *Chapter 9, Module Data*.
- ▶ `demo/` contains data files with demonstration data, useful for tests, training or module evaluation.
- ▶ `i18n/` is where Odoo will look for the translation `.pot` and `.po` files. See *Chapter 11, Internationalization*, for more details. These files don't need to be mentioned in the manifest file.
- ▶ `security/` contains the data files defining access control lists, usually a `ir.model.access.csv` file, and possibly an XML file to define access Groups and Record Rules for row level security. See *Chapter 10, Access Security*, for more details on this.
- ▶ `controllers/` contains the code files for the website controllers, for modules providing that kind of feature. Web controllers are covered in *Chapter 13, Web Server Development*.

- ▶ `static/` is where all web assets are expected to be placed. Unlike other directories, this directory name is not just a convention, and only files inside it can be made available for the Odoo web pages. They don't need to be mentioned in the module manifest, but will have to be referred to in the web template. This is discussed in more detail in *Chapter 14, CMS Website Development*.



When adding new files to a module, don't forget to declare them either in the `__openerp__.py` (for data files) or `__init__.py` (for code files); otherwise, those files will be ignored and won't be loaded.

Adding models

Models define the data structures to be used by our business applications. This recipe shows how to add a basic model to a module.

We will use a simple book library example to explain this; we want a model to represent books. Each book has a name and a list of authors.

Getting ready

We should have a module to work with. If we follow the first recipe in this chapter, we will have an empty `my_module`. We will use that for our explanation.

How to do it...

To add a new Model, we add a Python file describing it and then upgrade the addon module (or install it, if it was not already done). The paths used are relative to our addon module location (for example, `~/odoo-dev/local-addons/my_module/`):

1. Add a Python file to the module, `models/library_book.py`, with the following code:

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

2. Add a Python initialization file with code files to be loaded by the module `models/__init__.py`, with the following code:

```
from . import library_book
```

3. Edit the module Python initialization file to have the `models/` directory loaded by the module:

```
from . import models
```

4. Upgrade the Odoo module, either from the command line or from the apps menu in the user interface. If you look closely at the server log while upgrading the module, you should see this line:

```
openerp.modules.module: module my_module: creating or updating  
database tables
```

After this, the new `library.book` model should be available in our Odoo instance. If we have the technical tools activated, we can confirm that by looking it up at **Settings | Technical | Database Structure | Models**.

How it works...

Our first step was to create a Python file where our new module was created.

Odoo models are objects derived from the `Odoo Model` Python class.

When a new model is defined, it is also added to a central model registry. This makes it easier for other modules to later make modifications to it.

Models have a few generic attributes prefixed with an underscore. The most important one is `__name__`, providing a unique internal identifier to be used throughout the Odoo instance.

The model fields are defined as class attributes. We began defining the `name` field of the `Char` type. It is convenient for models to have this field, because by default, it is used as the record description when referenced from other models.

We also used an example of a relational field, `author_ids`. It defines a many-to-many relation between `Library Books` and the partners — a book can have many authors and each author can have many books.

There's much more to say about models, and they will be covered in more depth in *Chapter 4, Application Models*.

Next, we must make our module aware of this new Python file. This is done by the `__init__.py` files. Since we placed the code inside the `models/` subdirectory, we need the previous `init` file to import that directory, which should in turn contain another `init` file importing each of the code files there (just one in our case).

Changes to Odoo models are activated by upgrading the module. The Odoo server will handle the translation of the model class into database structure changes.

Although no example is provided here, business logic can also be added to these Python files, either by adding new methods to the Model's class, or by extending existing methods, such as `create()` or `write()`. This is addressed in *Chapter 5, Basic Server Side Development*.

Adding Menu Items and Views

Once we have Models for our data structure needs, we want a user interface for our users to interact with them. This recipe builds on the `Library Book Model` from the previous recipe and adds a menu item to display a user interface featuring list and form Views.

Getting ready

The addon module implementing the `library_book Model`, provided in the previous recipe, is needed. The paths used are relative to our addon module location (for example, `~/odoo-dev/local-addons/my_module/`).

How to do it...

To add a view, we will add an XML file with its definition to the module. Since it is a new Model, we must also add a menu option for the user to be able to access it.

Be aware that the sequence of the following steps is relevant, since some use references to IDs defined in previous steps:

1. Create the XML file to add the data records describing the user interface `views/library_book.xml`:


```
<?xml version="1.0" encoding="utf-8"?>
<openerp>
  <data>
    <!-- Data records go here -->
  </data>
</openerp>
```
2. Add the new data file to the addon module manifest, `__openerp__.py` by adding it to the `views/library_book.xml`:


```
# -*- coding: utf-8 -*-
{
    'name': "Library Books",
    'summary': "Manage your books",
    'depends': ['base'],
    'data': ['views/library_book.xml'],
}
```

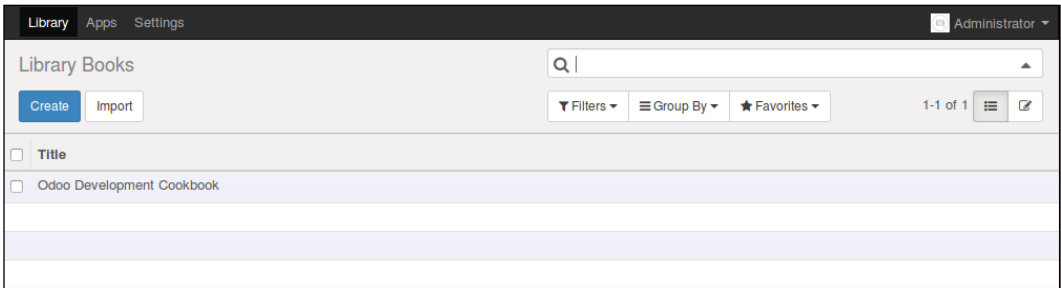
3. Add the Action that opens the Views in the `library_book.xml` file:

```
<act_window
    id="library_book_action"
    name="Library Books"
    res_model="library.book" />
```

4. Add the menu item to the `library_book.xml` file, making it visible to the users:

```
<menuitem
    id="library_book_menu"
    name="Library"
    action="library_book_action"
    parent=""
    sequence="5" />
```

If you try and upgrade the module now, you should be able to see a new top menu option (you might need to refresh your web browser). Clicking on it should work and will open Views for Library Books that are automatically generated by the server.



5. Add a custom form view to the `library_book.xml` file:

```
<record id="library_book_view_form" model="ir.ui.view">
    <field name="name">Library Book Form</field>
    <field name="model">library.book</field>
    <field name="arch" type="xml">

        <form>
            <group>
                <field name="name"/>
                <field name="author_ids" widget="many2many_tags"/>
            </group>
            <group>
                <field name="date_release"/>
            </group>
        </form>

    </field>
</record>
```

6. Add a custom Tree (List) view to the `library_book.xml` file:

```
<record id="library_book_view_tree" model="ir.ui.view">
  <field name="name">Library Book List</field>
  <field name="model">library.book</field>
  <field name="arch" type="xml">

    <tree>
      <field name="name"/>
      <field name="date_release"/>
    </tree>

  </field>
</record>
```

7. Add custom Search options to the `library_book.xml` file:

```
<record id="library_book_view_search" model="ir.ui.view">
  <field name="name">Library Book Search</field>
  <field name="model">library.book</field>
  <field name="arch" type="xml">

    <search>
      <field name="name"/>
      <field name="author_ids"/>
      <filter string="No Authors"
        domain=" [('author_ids', '=', False)] "/>
    </search>

  </field>
</record>
```

How it works...

At the low level, the user interface is defined by records stored in special Models. The first two steps create an empty XML file to define the records to be loaded and then add them to the module's list of data files to be installed.

Data files can be anywhere inside the module directory, but the convention is for the user interface to be defined inside a `views/` subdirectory using file names after the Model the interface is for. In our case, the `library.book` interface is in the `views/library_book.xml` file.

The next step is to define a Window Action to display the user interface in the main area of the web client. The Action has a target Model defined by `res_model` and sets the title to display to the user using `name`. These are just the basic attributes. It supports additional attributes, giving much more control of how the Views are rendered, such as what Views are to be displayed, adding filters on the records available, or setting default values. These are discussed in detail in *Chapter 8, Backend Views*.

In general, data records are defined using a `<record>` tag, but in our example, the Window Action was defined using the `<act_window>` tag. This is a shortcut to create records for the `ir.actions.act_window` Model, where Window Actions are stored.

Similarly, menu items are stored in the `ir.ui.menu` Model, but a convenience `<menuitem>` shortcut tag is available and was used.

These are the menu items' main attributes used here:

- ▶ `name`: This is the menu item text to be displayed.
- ▶ `action`: This is the identifier of the action to be executed. We use the ID of the Window Action created in the previous step.
- ▶ `sequence`: This is used to set the order at which the menu items of the same level are presented.
- ▶ `parent`: This is the identifier for the parent menu item. Our example menu item had no parent, meaning that it is to be displayed at the top of the menu.

At this point, our module can display a menu item, and clicking on it opens Views for the `Library Books` model. Since nothing specific is set on the Window Action, the default is to display a List (or Tree) view and a form view.

We haven't defined any of these Views, so Odoo will automatically create them on the fly. However, we will surely want to control how our Views look, so in the next two steps, a form and a tree view are created.

Both Views are defined with a record on the `ir.ui.view` model. The attributes we used are as follows:

- ▶ `name`: This is a title identifying this view. It is frequent to see the XML ID repeated here, but it can perfectly be a more human readable title.
- ▶ `model`: This is the internal identifier of the target model, as defined in its `_name` attribute.
- ▶ `arch`: This is the view architecture, where its structure is actually defined. This is where different types of View differ from each other.

Form Views are defined with a top `<form>` element, and its canvas is a two-column grid. Inside the form, `<group>` elements are used to vertically compose fields. Two groups result in two columns with fields, that are added using the `<field>` element. Fields use a default widget according to their data type, but a specific widget can be used with the help of the `widget` attribute.

Tree Views are simpler; they are defined with a top `<tree>` element containing `<field>` elements for the columns to be displayed.

Finally, we added a Search view to expand the search option in the box at the top right. Inside the `<search>` top-level tag, we can have `<field>` and `<filter>` elements. Field elements are additional fields that can be searched from the box. Filter elements are predefined filter conditions that can be activated with a click.

Using scaffold to create a module

When creating a new Odoo module, there is some boilerplate that needs to be set up. To help quick starting new modules, Odoo provides the `scaffold` command.

The recipe shows how to create a new module using the `scaffold` command, which will put in place a skeleton of the files directories to use.

Getting ready

We need Odoo installed and a directory for our custom modules.

We will assume that Odoo is installed at `~/odoo-dev/odoo` and our custom modules will be at `~/odoo-dev/local-addons`.

How to do it...

The `scaffold` command is used from the command line:

1. Change the working directory to where we will want our module to be. This can be whatever directory you choose, but within an addons path to be useful. Following the directory choices used in the previous recipe, it should be as follows:

```
$ cd ~/odoo-dev/local-addons
```

2. Choose a technical name for the new module and use the `scaffold` command to create it. For our example, we will choose `my_scaffolded`:

```
$ ~/odoo-dev/odoo/odoo.py scaffold my_scaffolded
```

3. Edit the `__openerp__.py` default module manifest provided and change the relevant values. You will surely want to at least change the module title in the `name` key.

This is how the generated addon module should look like:

```
$ tree my_scaffolded
my_scaffolded
├── controllers.py
├── demo.xml
├── __init__.py
├── models.py
├── openerp.py
├── security
│   └── ir.model.access.csv
└── templates.xml
```

How it works...

The `scaffold` command creates the skeleton for a new module based on a template.

By default, the new module is created in the current working directory, but we can provide a specific directory where to create the module, passing it as an additional parameter. For example:

```
$ ~/odoo-dev/odoo/odoo.py scaffold my_module ~/odoo-dev/local-addons
```

A default template is used but a theme template is also available, for website theme authoring. To choose a specific template, the `-t` option can be used. We are also allowed to use a path for a directory with a template.

This means that we can use our own templates with the `scaffold` command. The built-in themes can be used as a guide, and they can be found in the Odoo subdirectory `./openerp/cli/templates`. To use our own template, we could use something like this:

```
$ ~/odoo-dev/odoo/odoo.py scaffold -t path/to/template my_module
```

There's more...

Unfortunately, the default template does not adhere to the current Odoo guidelines. We could create our own template by copying the default one, at `odoo/openerp/cli/templates/default/`, and modifying to better suit the structure described in the *Organizing the module file structure* recipe. A command similar to this could get us started on that:

```
$ cp -r ~/odoo-dev/odoo/openerp/cli/templates/default ~/odoo-dev/template
```

Later, we can use it with the following command:

```
$ ~/odoo-dev/odoo/odoo.py scaffold -t ~/odoo-dev/template my_module
```


4

Application Models

In this chapter, we will cover the following topics:

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

Introduction

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