

2

Building Your First Odoo Application

Developing in Odoo most of the time means creating our own modules. In this chapter, we will create our first Odoo application, and you will learn the steps needed make it available to Odoo and install it.

Inspired by the notable `todomvc.com` project, we will build a simple to-do application. It should allow us to add new tasks, then mark them as completed, and finally clear the task list of all completed tasks.

You will learn how Odoo follows an MVC architecture, and we will go through the following layers during the to-do application implementation:

- The **model**, defining the structure of the data
- The **view**, describing the user interface
- The **controller**, supporting the business logic of the application

The model layer is defined with Python objects that have their data is stored in the PostgreSQL database. The database mapping is automatically managed by Odoo, and the mechanism responsible for this is the **object relational model**, (ORM).

The view layer describes the user interface. Views are defined using XML, which is used by the web client framework to generate data-aware HTML views.

The web client views perform data persistent actions by interacting with the server ORM. These can be basic operations such as write or delete, but can also invoke methods defined in the ORM Python objects, performing more complex business logic. This is what we refer to as the controller layer.



Note that the concept of controller mentioned here is different from the Odoo web development controllers. Those are program endpoints that web pages can call to perform actions.

With this approach, you will be able to gradually learn about the basic building blocks that make up an application and experience the iterative process of building an Odoo module from scratch.

Understanding applications and modules

It's common to hear about Odoo modules and applications. But what exactly is the difference between them? **Modules** are building blocks of Odoo applications. A module can add or modify Odoo features. It is supported by a directory containing a manifest or descriptor file (named `__openerp__.py`) and the remaining files that implement its features. Sometimes, modules can also be referred to as "add-ons." **Applications** are not different from regular modules, but functionally, they provide a central feature, around which other modules add features or options. They provide the core elements for a functional area, such as accounting or HR, around which other modules add features. Because of this, they are highlighted in the Odoo **Apps** menu.

Modifying and extending modules

In the example that will follow, we will create a new module with as few dependencies as possible.

This will not be the typical case, however. The most frequent situation is where modifications or extensions are needed on an already existing module to fit some specific use cases.

The golden rule is that we shouldn't modify existing modules by changing them directly. It's considered bad practice to modify existing modules. This is especially true for the official modules provided by Odoo. Doing so does not allow a clear separation between the original module code and our modifications, and makes it difficult to apply upgrades.

Instead, we should create new modules to be applied on top of the modules we want to modify, and implement those changes. This is one of Odoo's main strengths: it provides "inheritance" mechanisms that allow custom modules to extend existing modules, either official or from the community. The inheritance is possible at all levels data models, business logic, and user interface layers.

Right now, we will create a completely new module, without extending any existing module, to focus on the different parts and steps involved in module creation. We will just take a brief look at each part, since each will be studied in more detail in the later chapters. Once we are comfortable with creating a new module, we can dive into the inheritance mechanisms, which will be introduced in the next chapter.

Creating a new module

Our module will be a very simple application to keep to-do tasks. These tasks will have a single text field, for the description, and a checkbox to mark them as complete. We will also have a button to clean the to-do list from the old completed tasks.

These are very simple specifications, but throughout the book we will gradually add new features to it, to make it more interesting for the users.

Enough talk, let's start coding and create our new module.

Following the instructions in *Chapter 1, Getting Started with Odoo Development*, we should have the Odoo server at `/odoo-dev/odoo/`. To keep things tidy, we will create a new directory alongside it to host our custom modules:

```
$ mkdir ~/odoo-dev/custom-addons
```

An Odoo module is a directory containing an `__openerp__.py` descriptor file. This is still a legacy from when Odoo was named OpenERP, and in the future is expected to become `__odoo__.py`.

It also needs to be Python importable, so it must also have an `__init__.py` file.

The module's directory name will be its technical name. We will use `todo_app` for it. The technical name must be a valid Python identifier: it should begin with a letter and can only contain letters, numbers, and the underscore character. The following commands create the module directory and create an empty `__init__.py` file in it:

```
$ mkdir ~/odoo-dev/custom-addons/todo_app
$ touch ~/odoo-dev/custom-addons/todo_app/__init__.py
```

Next we need to create the descriptor file. It should contain only a Python dictionary with about a dozen possible attributes, of which only the name attribute is required. A longer description attribute and the author also have some visibility and are advised.

We should now add an `__openerp__.py` file alongside the `__init__.py` file with the following content:

```
{
    'name': 'To-Do Application',
    'description': 'Manage your personal Tasks with this module.',
    'author': 'Daniel Reis',
    'depends': ['mail'],
    'application': True,
}
```

The depends attribute can have a list of other modules required. Odoo will have them automatically installed when this module is installed. It's not a mandatory attribute, but it's advised to always have it. If no particular dependencies are needed, we should depend on the special base module. You should be careful to ensure all dependencies are explicitly set here, otherwise the module may fail to install in a clean database (due to missing dependencies) or have loading errors, if the other needed modules are loaded afterwards. For our application, we want to depend on the **mail** module because that is the module that adds the **Messaging** top menu, and we will want to include our new menu options there.

To be concise, we chose to use very few descriptor keys, but in a real word scenario it is recommended to also use these additional keys, since they are relevant for the Odoo app store:

- `summary` is displayed as a subtitle for the module.
- `version`, by default, is 1.0. Should follow semantic versioning rules (see semver.org for details).
- `license` identifier, by default is AGPL-3.
- `website` is a URL to find more information about the module. This can help people to find more documentation or the issue tracker to file bugs and suggestions.
- `category` is the functional category of the module, which defaults to Uncategorized. The list of existing categories can be found in the security Groups form (Settings | User | Groups menu), in the **Application** field drop-down list.

These other descriptor keys are also available:

- `installable` is by default `True`, but can be set to `False` to disable a module.
- `auto_install` if this is set to `True` this module is automatically installed if all its dependencies are already installed. It is used for **glue** modules.

Since Odoo 8.0, instead of the `description` key we can use a `README.rst` or `README.md` file in the module's top directory.

Adding to the addons path

Now that we have a new module, even if minimal, we want to make it available in Odoo.

For that, we need to make sure the directory the module is in is part of the addons path. And then we need to update the Odoo module list.

Both operations have been explained in detail in the previous chapter, but we will follow here with a brief overview of what is needed.

We will position in our work directory and start the server with the appropriate addons path configuration:

```
$ cd ~/odoo-dev
$ odoo/odoo.py -d v8dev --addons-path="custom-addons,odoo/addons" --save
```

The `--save` option saves the options you used in a config file. This spares you from repeating them the next time you restart the server: just run `./odoo.py` and the last saved options will be used.

Look closely at the server log. It should have an **INFO ? openerp: addons paths: (...)** line, and it should include our `custom-addons` directory.

Remember to also include any other addons directories you might be using. For instance, if you followed the last chapter's instructions to install the department repository, you might want to include it and use the option:

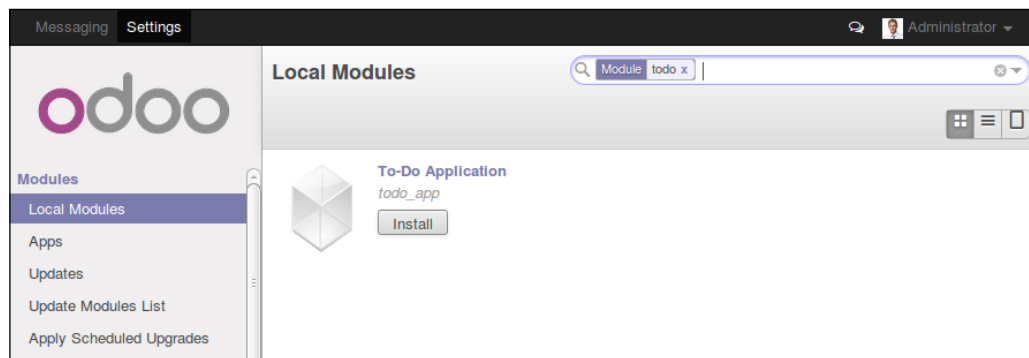
```
--addons-path="custom-addons,departmernt,odoo/addons"
```

Now let's ask Odoo to acknowledge the new module we just added.

For that, in the **Modules** section of the **Settings** menu, select the **Update Modules List** option. This will update the module list adding any modules added since the last update to the list. Remember that we need the Technical Features enabled for this option to be visible. That is done by selecting the **Technical Features** checkbox for our user.

Installing the new module

The **Local Modules** option shows us the list of available modules. By default it shows only **Apps** modules. Since we created an application module we don't need to remove that filter to see it. Type "todo" in the search and you should see our new module, ready to be installed.



Now click on its **Install** button and you're done!

Upgrading a module

Developing a module is an iterative process, and you will want changes made on source files to be applied and visible in Odoo.

In most cases this is done by upgrading the module: look up the module in the **Local Modules** list and, since it is installed, you will see an **Upgrade** button available.

However, when the changes are only in Python code, the upgrade may not have an effect. Instead of a module upgrade, an application server restart is needed.

In some cases, if the module has changed both in data files and Python code, you might need both operations. This is a common source of confusion for newcomer Odoo developers.

But fortunately, there is a better way. The simplest and fastest way to make all our changes to a module effective is to stop (*Ctrl + C*) and restart the server process requesting our modules to be upgraded on our work database.

To start the server upgrading the `todo_app` module in the `v8dev` database, we will use:

```
$ ./odoo.py -d v8dev -u todo_app
```

The `-u` option (or `--update` in the long form) requires the `-d` option and accepts a comma-separated list of modules to update. For example, we could use: `-u todo_app,mail`.

Whenever you need to upgrade work in progress modules throughout the book, the safest way to do so is to go to the terminal window where you have Odoo running, stop the server, and restart it with the command above. Frequently pressing the Up arrow key will be enough, since it should bring you the previous command you used to start the server.

Unfortunately, updating the module list and uninstalling modules are both actions not available through the command line. These have to be done through the web interface, in the **Settings** menu.

Creating an application model

Now that Odoo knows about our new module, let's start by adding to it a simple model.

Models describe business objects, such as an opportunity, a sales order, or a partner (customer, supplier, and so on.). A model has a list of attributes and can also define its specific business.

Models are implemented using a Python class derived from an Odoo template class. They translate directly to database objects, and Odoo automatically takes care of that when installing or upgrading the module.

Some consider it good practice to keep the Python files for models inside a `models` subdirectory. For simplicity we won't be following that here, so let's create a `todo_model.py` file in the `todo_app` module main directory.

Add the following content to it:

```
# -*- coding: utf-8 -*-
from openerp import models, fields
class TodoTask(models.Model):
    _name = 'todo.task'
    name = fields.Char('Description', required=True)
    is_done = fields.Boolean('Done?')
    active = fields.Boolean('Active?', default=True)
```

The first line is a special marker telling the Python interpreter that this file has UTF-8, so that it can expect and handle non-ASCII characters. We won't be using any, but it's safer to use it anyway.

The second line makes available the `models` and `fields` objects from the Odoo core.

The third line declares our new model. It's a class derived from `models.Model`. The next line sets the `_name` attribute defining the identifier that will be used throughout Odoo to refer to this model. Note that the actual Python class name is meaningless to the other Odoo modules. The `_name` value is what will be used as an identifier.

Notice that this and the following lines are indented. If you're not familiar with Python you should know that this is important: indentation defines a nested code block, so these four line should all be equally indented.

The last three lines define the model's fields. It's worth noting that `name` and `active` are names of special fields. By default Odoo will use the `name` field as the record's title when referencing it from other models. The `active` field is used to inactivate records, and by default only active records will be shown. We will use it to clear away completed tasks without actually deleting them from the database.

Right now, this file is not yet used by the module. We must tell Odoo to load it with the module in the `__init__.py` file. Let's edit it to add the following line:

```
from . import todo_model
```

That's it. For our changes to take effect the module has to be upgraded. Locate the **To-Do** application in the **Local Modules** and click on its **Upgrade** button.

Now we can inspect the newly created model in the **Technical** menu. Go to **Database Structure | Models** and search for the **todo.task** model on the list. Then click on it to see its definition:

The screenshot shows the Odoo Technical interface for the `todo.task` model. The left sidebar contains the 'Technical' menu with 'Database Structure' expanded to 'Models'. The main panel displays the model's metadata and a table of fields.

Models / todo.task

Buttons: Edit, Create, Print, More

Page: 2 / 2

Model Description	Model	Type	In Modules	Base Object
todo.task	todo.task			Base Object
Transient Model	<input type="checkbox"/>			todo_app

Tabs: Fields, Access Rights, Notes, Views

Name	Field Label	Field Type	Required	Readonly	Searchable	Type
active	Active?	boolean	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
create_date	Created on	datetime	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
create_uid	Created by	many2one	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
id	ID	integer	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
is_done	Done?	boolean	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
name	Description	char	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
write_date	Last Updated on	datetime	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field
write_uid	Last Updated by	many2one	<input type="checkbox"/>	<input type="checkbox"/>	Not Searchable	Base Field

Buttons: Create a Menu

If everything went right, this will let us confirm that the model and our fields were created. If you made changes and don't see them here, try a server restart, as described before, to force all of the Python code to be reloaded.

We can also see some additional fields we didn't declare. These are the five reserved fields Odoo automatically adds to any model. They are as follows:

- `id`: This is the unique identifier for each record in the particular model.
- `create_date` and `create_uid`: These tell us when the record was created and who created it, respectively.
- `write_date` and `write_uid`: These tell us when the record was last modified and who modified it, respectively.

Adding menu entries

Now that we have a model to store our data, let's make it available on the user interface.

All we need to do is to add a menu option to open the `To-do Task` model so that it can be used. This is done using an XML file. Just as in the case of models, some people consider it good practice to keep the view definitions inside a `views` subdirectory.

We will create a new `todo_view.xml` data file in the module's top directory, and it will declare a menu item and the action performed by it:

```
<?xml version="1.0"?>
<openerp>
  <data>

    <!-- Action to open To-do Task list -->
    <act_window id="action_todo_task"
      name="To-do Task"
      res_model="todo.task"
      view_mode="tree,form" />

    <!-- Menu item to open To-do Task list -->
    <menuitem id="menu_todo_task"
      name="To-Do Tasks"
      parent="mail.mail_feeds"
      sequence="20"
      action="action_todo_task" />

  </data>
</openerp>
```

The user interface, including menu options and actions, is stored in database tables. The XML file is a data file used to load those definitions into the database when the module is installed or upgraded. This is an Odoo data file, describing two records to add to Odoo:

- The `<act_window>` element defines a client-side Window Action to open the `todo.task` model defined in the Python file, with the `tree` and `form` views enabled, in that order.
- The `<menuitem>` defines a menu item under the **Messaging** menu (identified by `mail.mail_feeds`), calling the `action_todo_task` action, which was defined before. The `sequence` lets us set the order of the menu options.

Now we need to tell the module to use the new XML data file. That is done in the `__openerp__.py` file using the `data` attribute. It defines the list of files to be loaded by the module. Add this attribute to the descriptor's dictionary:

```
'data': ['todo_view.xml'],
```

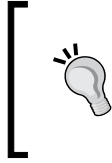
Now we need to upgrade the module again for these changes to take effect. Go to the **Messaging** menu and you should see our new menu option available.



Clicking on it will open an automatically generated form for our model, allowing us to add and edit records.

Views should be defined for models to be exposed to the users, but Odoo is nice enough to do that automatically if we don't, so we can work with our model right away, without having any form or list views defined yet.

So far, so good! Let's improve our user interface now. Try the gradual improvements as shown in the next sections, doing frequent module upgrades, and don't be afraid to experiment.



In case an upgrade fails because of an XML error, don't panic! Comment out the last edited XML portions, or remove the XML file from `__openerp__.py`, and repeat the upgrade. The server should start correctly. Now read the error message in the server log carefully – it should point you to where the problem is.

Creating views – form, tree, and search

As we have seen, if no view is defined, Odoo will automatically generate basic views to get you going. But surely you would like to define the module views yourself, so that's what we'll do next.

Odoo supports several types of views, but the three main ones are: `list` (also called `tree`), `form`, and `search` views. We'll add an example of each to our module.

All views are stored in the database, in the `ir.model.view` model. To add a view in a module, we declare a `<record>` element describing the view in an XML file that will be loaded into the database when the module is installed.

Creating a form view

Edit the XML we just created to add this `<record>` element just after the `<data>` opening tag at the top:

```
<record id="view_form_todo_task" model="ir.ui.view">
  <field name="name">To-do Task Form</field>
  <field name="model">todo.task</field>
  <field name="arch" type="xml">

    <form string="To-do Task">
      <field name="name"/>
      <field name="is_done"/>
      <field name="active" readonly="1"/>
    </form>

  </field>
</record>
```

This will add a record to the model `ir.ui.view` with the identifier `view_form_todo_task`. The view is for the model `todo.task` and named `To-do Task Form`. The name is just for information, does not have to be unique, but should allow one to easily identify what record it refers to.

The most important attribute is `arch`, containing the view definition. Here we say it's a form, and it contains three fields, and we chose to make the `active` field read only.

Formatting as a business document

The above provides a basic form view, but we can make some improvements to make it nicer. For document models Odoo has a presentation style that mimics a paper page. The form contains two elements: a `<head>`, containing action buttons, and a `<sheet>`, containing the data fields:

```
<form>
  <header>
    <!-- Buttons go here-->
  </header>
  <sheet>
    <!-- Content goes here: -->
    <field name="name"/>
    <field name="is_done"/>
  </sheet>
</form>
```

Adding action buttons

Forms can have buttons to run actions. These are able to trigger workflow actions, run Window Actions, such as opening another form, or run Python functions defined in the model.

They can be placed anywhere inside a form, but for document-style forms, the recommended place for them is the `<header>` section.

For our application, we will add two buttons to run methods of the `todo.task` model:

```
<header>
  <button name="do_toggle_done" type="object"
    string="Toggle Done" class="oe_highlight" />
  <button name="do_clear_done" type="object"
    string="Clear All Done" />
</header>
```

The basic attributes for a button are: `string` with the text to display on the button, the `type` of action it performs, and the `name` that is the identifier for that action. The optional `class` attribute can apply CSS styles, just like in regular HTML.

Organizing forms using groups

The `<group>` tag allows organizing the form content. Placing `<group>` elements inside a `<group>` element creates a two column layout inside the outer group. Group elements are advised to have a name to make it easier for other modules to extend on them.

We will use this to better organize our content. Let's change the `<sheet>` content of our form to match this:

```
<sheet>
  <group name="group_top">
    <group name="group_left">
      <field name="name"/>
    </group>
    <group name="group_right">
      <field name="is_done"/>
      <field name="active" readonly="1"/>
    </group>
  </group>
</sheet>
```

The complete form view

At this point, our record in `todo_view.xml` for the `todo.task` form view should look like this:

```
<record id="view_form_todo_task" model="ir.ui.view">
  <field name="name">To-do Task Form</field>
  <field name="model">todo.task</field>
  <field name="arch" type="xml">

    <form>
      <header>
        <button name="do_toggle_done" type="object"
          string="Toggle Done" class="oe_highlight" />
        <button name="do_clear_done" type="object"
          string="Clear All Done" />
      </header>
      <sheet>
        <group name="group_top">
          <group name="group_left">
            <field name="name"/>
          </group>
          <group name="group_right">
            <field name="is_done"/>
          </group>
        </group>
      </sheet>
    </form>
  </field>
</record>
```

```
        <field name="active" readonly="1" />
    </group>
</group>
</sheet>
</form>

</field>
</record>
```

Remember that for the changes to be loaded into our Odoo database, a module upgrade is needed. To see the changes in the web client, the form needs to be reloaded: either click again on the menu option that opens it, or reload the browser page (*F5* in most browsers).

Now, let's add the business logic for the actions buttons.

Adding list and search views

When viewing a model in list mode, a `<tree>` view is used. Tree views are capable of displaying lines organized in hierarchies, but most of the time they are used to display plain lists.

We can add the following tree view definition to `todo_view.xml`:

```
<record id="view_tree_todo_task" model="ir.ui.view">
  <field name="name">To-do Task Tree</field>
  <field name="model">todo.task</field>
  <field name="arch" type="xml">
    <tree colors="gray:is_done==True">
      <field name="name"/>
      <field name="is_done"/>
    </tree>
  </field>
</record>
```

We have defined a list with only two columns, `name` and `is_done`. We also added a nice touch: the lines for done tasks (`is_done==True`) are shown in grey.

At the top right of the list Odoo displays a search box. The default fields it searches for and available predefined filters can be defined by a `<search>` view.

As before, we will add this to the `todo_view.xml`:

```
<record id="view_filter_todo_task" model="ir.ui.view">
  <field name="name">To-do Task Filter</field>
  <field name="model">todo.task</field>
  <field name="arch" type="xml">
    <search>
      <field name="name"/>
      <filter string="Not Done"
        domain=" [('is_done', '=', False)] "/>
      <filter string="Done"
        domain=" [('is_done', '!=', False)] "/>
    </search>
  </field>
</record>
```

The `<field>` elements define fields that are also searched when typing in the search box. The `<filter>` elements add predefined filter conditions, using domain syntax that can be selected with a user click.

Adding business logic

Now we will add some logic to our buttons. Edit the `todo_model.py` Python file to add to the class the methods called by the buttons.

We will use the new API introduced in Odoo 8.0. For backward compatibility, by default Odoo expects the old API, and to create methods using the new API we need to use Python decorators on them. First we need to import the new API, so add it to the import statement at the top of the Python file:

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

The **Toggle Done** button's action will be very simple: just toggle the **Is Done?** flag. For logic on a record, the simplest approach is to use the `@api.one` decorator. Here `self` will represent one record. If the action was called for a set of records, the API would handle that and trigger this method for each of the records.

Inside the `ToDoTask` class add:

```
@api.one
def do_toggle_done(self):
    self.is_done = not self.is_done
    return True
```

As you can see, it simply modifies the `is_done` field, inverting its value. Methods, then, can be called from the client side and must always return something. If they return `None`, client calls using the XMLRPC protocol won't work. If we have nothing to return, the common practice is to just return the `True` value.

After this, if we restart the Odoo server to reload the Python file, the **Toggle Done** button should now work.

For the **Clear All Done** button we want to go a little further. It should look for all active records that are done, and make them inactive. Form buttons are supposed to act only on the selected record, but to keep things simple we will do some cheating, and it will also act on records other than the current one:

```
@api.multi
def do_clear_done(self):
    done_recs = self.search([('is_done', '=', True)])
    done_recs.write({'active': False})
    return True
```

On methods decorated with `@api.multi` the `self` represents a recordset. It can contain a single record, when used from a form, or several records, when used from a list view. We will ignore the `self` recordset and build our own `done_recs` recordset containing all the tasks that are marked as done. Then we set the `active` flag to `False`, in all of them.

The `search` is an API method returning the records meeting some conditions. These conditions are written in a domain, that is a list of triplets. We'll explore domains in more detail later.

The `write` method sets values at once on all elements of the recordset. The values to write are described using a dictionary. Using `write` here is more efficient than iterating through the recordset to assign the value to them one by one.

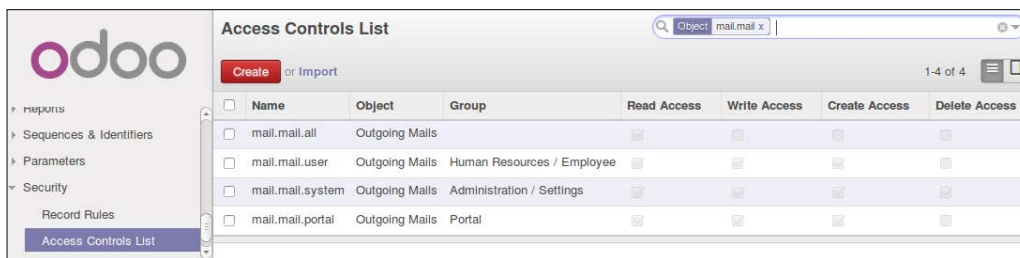
Note that `@api.one` is not the most efficient for these actions, since it will run for each selected record. The `@api.multi` ensures that our code runs only once even if there is more than one record selected when running the action. This could happen if an option for it were to be added on the list view.

Setting up access control security

You might have noticed, upon loading our module is getting a warning message in the server log: **The model `todo.task` has no access rules, consider adding one.**

The message is pretty clear: our new model has no access rules, so it can't be used by anyone other than the admin super user. As a super user the admin ignores data access rules, that's why we were able to use the form without errors. But we must fix this before other users can use it.

To get a picture of what information is needed to add access rules to a model, use the web client and go to: **Settings | Technical | Security | Access Controls List**.



Name	Object	Group	Read Access	Write Access	Create Access	Delete Access
mail.mail.all	Outgoing Mails		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
mail.mail.user	Outgoing Mails	Human Resources / Employee	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
mail.mail.system	Outgoing Mails	Administration / Settings	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
mail.mail.portal	Outgoing Mails	Portal	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Here we can see the ACL for the `mail.mail` model. It indicates, per group, what actions are allowed on records.

This information needs to be provided by the module, using a data file to load the lines into the `ir.model.access` model. We will add full access on the model to the employee group. Employee is the basic access group nearly everyone belongs to.

This is usually done using a CSV file named `security/ir.model.access.csv`. Models have automatically generated identifiers: for `todo.task` the identifier is `model_todo_task`. Groups also have identifiers set by the modules creating them. The employee group is created by the base module and has identifier `base.group_user`. The line's name is only informative and it's best if it's kept unique. Core modules usually use a dot-separated string with the model name and the group. Following this convention we would use `todo.task.user`.

Now we have everything we need to know, let's add the new file with the following content:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_todo_task_group_user,todo.task.user,model_todo_task,base.group_user,1,1,1,1
```

We must not forget to add the reference to this new file in the `__openerp__.py` descriptor's data attribute, so that should look like this:

```
'data': [
    'todo_view.xml',
    'security/ir.model.access.csv',
],
```

As before, upgrade the module for these additions to take effect. The warning message should be gone, and you can confirm the permissions are **OK** by logging in with the user `demo` (password is also `demo`) and trying the to-do tasks feature.

Row-level access rules

Odoo is a multi-user system, and we would like the to-do tasks to be private to each user. Fortunately for us, Odoo also supports row-level access rules. In the **Technical** menu they can be found in the **Record Rules** option, alongside the **Access Control List**.

Record rules are defined in the `ir.rule` model. As usual, we need a distinctive name. We also need the model they operate on and the domain to force access restriction. The domain filter uses the same domain syntax mentioned before, and used across Odoo.

Finally, rules may be either global (the `global` field is set to `True`) or only for particular security groups. In our case, it could perfectly be a global rule, but to illustrate the most common case, we will make it a group-specific rule, applying only to the employees group.

We should create a `security/todo_access_rules.xml` file with this content:

```
<?xml version="1.0" encoding="utf-8"?>
<openerp>
  <data noupdate="1">
    <record id="todo_task_user_rule" model="ir.rule">
      <field name="name">ToDo Tasks only for owner</field>
      <field name="model_id" ref="model_todo_task"/>
      <field name="domain_force">[('create_uid','=',user.id)]
    </field>
      <field name="groups" eval="[(4,ref('base.group_user'))]" />
    </record>
  </data>
</openerp>
```

Notice the `noupdate="1"` attribute. It means this data will not be updated in module upgrades. This will allow it to be customized later, since module upgrades won't destroy user-made changes. But beware that this will also be so while developing, so you might want to set `noupdate="0"` during development, until you're happy with the data file.

In the `groups` field, you will also find a special expression. It's a one-to-many relational field, and they have special syntax to operate with. In this case, the `(4, x)` tuple indicates to append `x` to the records, and `x` is a reference to the `employees` group, identified by `base.group_user`.

As before, we must add the file to `__openerp__.py` before it can be loaded to the module:

```
'data': [
    'todo_view.xml',
    'security/ir.model.access.csv',
    'security/todo_access_rules.xml',
],
```

Adding an icon to the module

Our module is looking good. Why not add an icon to it to make it look even better? For that we just need to add to the module a `static/description/icon.png` file with the icon to use.

The following commands add an icon copied from the core `Notes` module:

```
$ mkdir -p ~/odoo-dev/custom-addons/todo_app/static/description
$ cd ~/odoo-dev/custom-addons/todo_app/static/description
$ cp ../odoo/addons/note/static/description/icon.png ./
```

Now, if we update the module list, our module should be displayed with the new icon.

Summary

We created a new module from the start, covering the most frequently used elements in a module: models, the three base types of views (form, list, and search), business logic in model methods, and access security.

In the process, you got familiar with the module development process, which involves module upgrades and application server restarts to make the gradual changes effective in Odoo.

Always remember, when adding model fields, an upgrade is needed. When changing Python code, including the manifest file, a restart is needed. When changing XML or CSV files, an upgrade is needed; also when in doubt, do both: upgrade the modules and restart the server.

In the next chapter, you will learn about building modules that stack on existing ones to add features.