8
Backend Views

In this chapter, we will cover the following topics:

- Adding a menu item and window action
- Having an action open a specific view
- Adding content and widgets to a form view
- Adding buttons to forms
- Passing parameters to forms and actions: Context
- Defining filters on record lists: Domain
- List views
- Search views
- Changing existing views: View inheritance
- Document-style forms
- Dynamic form elements using attrs
- Embedded views
- Kanban views
- Show kanban cards in columns according to their state
- Calendar and gantt views
- Graph and pivot views
- QWeb reports

Throughout this chapter, we will assume that you have a database with the base addon installed and an empty Odoo addon module where you add XML code from the recipes to a data file referenced in the addon's manifest. Refer to *Chapter 3*, *Creating Odoo Modules* for how to activate changes in your addon.

Introduction

This chapter covers all the UI elements that users are confronted with when they use anything other than the website part of Odoo. Historically, this basically was all of OpenERP, so also in an Odoo context, it is often just referred to as the web client. To be more specific, we will call this the **backend** as opposed to the website frontend.

Adding a menu item and window action

The most obvious way to make a new feature available to the users is by using a menu item. When you click on a menu item, something happens. This recipe walks you through how to define that something.

We will create a top level menu displaying a sub menu in the left hand menu bar, opening a list of all the customers.

This can also be done using the web user interface, via the settings menu, but we prefer to use XML data files since this is what we'll have to use when creating our addon modules.

How to do it...

In an XML data file of our addon module, perform the following steps:

1. Define an action to be executed:

```
<act_window id="action_all_customers"
            name="All customers"
            res_model="res.partner"
            view_mode="tree,form"
            domain="[('customer', '=', True)]"
            context="{'default_customer': True}"
            limit="80" />
```

2. Create the menu structure:

```
<menuitem id="menu_custom_toplevel"
          name="My custom menu" />
<menuitem id="menu_custom_left"
```

```
            parent="menu_custom_toplevel"
            name="This will appear in the left bar" />
```

3. Refer to our action in menu:

```
<menuitem id="menu_all_customers"
            parent="menu_custom_left"
            action="action_all_customers"
            sequence="10"
            groups="" />
```

If we now upgrade the module, we will see a top level menu that opens a sub menu in the left menu bar. Clicking on that menu item will open a list of all the customers.

How it works...

The first XML element, `act_window`, declares a window action to display a list view with all the customers. We used the most important attributes:

- `name`: To be used as the title for the views opened by the action.
- `res_model`: This is the model to be used. We are using `res.partner`, where Odoo stores all the partners and addresses, including customers.
- `view_mode`: This lists the view types to make available. The default value is *tree, view*, making available the list and form views. Other possible choices are *kanban*, *graph*, *calendar*, and *gantt*, explained later in this chapter.
- `domain`: This is optional and allows you to set a filter on the records to be made available in the views. In this case, we want to limit the partners to only those that are customers. We will explain this in more detail in a dedicated recipe later.
- `context`: This can set values made available to the opened views, affecting their behavior. In our example, on new records we want the customer flag's default value to be `True`. This will be covered in more depth in another recipe.
- `limit`: This sets the default amount of records that can be seen on list views. It defaults to `80`.

Next we create the menu item hierarchy; from the top level menu to the clickable end menu item. The most important attributes for the `menuitem` element are:

- `name`: This is used as the text the menu items display. If your menu item links to an action you can leave this out, because in that case the action's name will be used.
- `parent` (`parent_id` if using the `record` element): This is the XML ID referencing the parent menu item. Items with no parent are top level menus.
- `action`: This is the XML ID referencing the action to be called. Only menu items without child elements are clickable, so this option is only effective in those cases.

- `sequence`: This is used to order sibling menu items.
- `groups` (`groups_id` with the `record` tag): This is an optional list of user groups that can access this menu item. If empty, it will be available to all the users.

Window actions automatically determine the view to be used by looking up views for the target model, with the intended type (`form`, `tree`, and so on) and picking the one with the lowest sequence number.

`act_window` and `menuitem` are convenient shortcut XML tags that hide what you're actually doing: You create a record of the models `ir.actions.act_window` and `ir.ui.menu` respectively.

Be aware that names used with the `menuitem` shortcut may not map to the field names used when using a `record` element: `parent` should be `parent_id` and `groups` should be `groups_id`.

To build the menu, the web client reads all the records from `ir.ui.menu` and infers their hierarchy from the `parent_id` field. Menus are also filtered based on user permissions to models and groups assigned to menus and actions. When a user clicks a menu item, its `action` is executed.

There's more...

Window actions also support a `target` attribute to specify how the view is to be presented. Possible choices are:

- **current**: This is the default and opens the view in the web client main content area.
- **new**: This opens in a popup.
- **inline**: This makes most sense for forms. It opens a form in the edit mode but neither with the standard buttons to edit, create, or delete, nor with the **More** actions menu.
- **inlineview**: This is similar to inline, but opens in the read mode instead of edit.

The window action's `view_type` attribute is mostly obsolete by now. The alternative to the default **form** is **tree**, which causes the groups lists to render a hierarchical tree. Don't confuse this attribute with the `view_mode` attribute used and explained earlier, which actually decides which types of views are used.

There are also some additional attributes available for window actions that are not supported by the `act_window` shortcut tag. So to use them, we must use the `record` element with the following fields:

- **res_id**: If opening a form, you can have it open a specific record by setting its ID here. This can be useful for multi step wizards, or in cases when you often have to view/edit a specific record.

- **search_view_id**: This specifies a specific search view to use for the tree and graph views.
- **auto_search**: This is **True** by default. Set this to **False** if searching for your object is very time and/or resource consuming. This way, the user can review the search parameters and press **Search** when satisfied. With the default, the search is triggered immediately when the action is opened.

Keep in mind that the menu bar at the top and the menu to the left are all made up of the same stuff, the menu items. The only difference is that the items in the top bar don't have any parent menus, while the ones on the left bar have the respective menu item from the top bar as parent. In the left bar, the hierarchical structure is more obvious.

Also bear in mind that for design reasons, the first level menus in the left bar are rendered as a kind of header and standard Odoo doesn't assign an action to them very often. So even if you technically can assign an action to them, your users won't be used to click them and will probably be confused if you expect them to do so.

See also

You'll find a more detailed discussion of the XML ID reference mechanism in *Chapter 9, Module Data*. For now, just keep in mind that you can set references this way and – very importantly – that order matters. If the tags above were inverted, the addon containing this XML code wouldn't install because `menuitem` would refer to an unknown `action`.

This can become a pitfall when you add new data files and new elements during your development process, because then the order in which you add those files and elements does not necessarily reflect the order they will be loaded in an empty database. Always check before deployment if your addon installs in an empty database.

The action type `ir.actions.act_window` is the most common one, but a menu can refer to any type of action. Technically, it is just the same if you link to a client action, a server action, or any other model defined in the **ir.actions.*** namespace. It just differs in what the backend makes out of the action.

If you need just a tiny bit more flexibility in the concrete action to be called, look into the server actions that return a window action in turn. If you need complete flexibility on what you present, look into the client actions (**ir.actions.client**) which allow you to have a completely custom user interface. But only do so as last resort as you lose a lot of Odoo's convenient helpers when using them.

Having an action open a specific view

Window actions automatically determine the view to be used, but sometimes we want an action to open a specific view.

We will create a basic form view for the partner model and make the window action specifically open it.

How to do it...

1. Define the partner minimal form view:

```
<record id="form_all_customers" model="ir.ui.view">
    <field name="name">All customers</field>
    <field name="model">res.partner</field>
    <field name="arch" type="xml">
        <form>
            <group>
                <field name="name" />
            </group>
        </form>
    </field>
</record>
```

2. Tell the action from the previous recipe to use it:

```
<record id="action_all_customers_form"
        model="ir.actions.act_window.view">
    <field name="act_window_id" ref="action_all_customers" />
    <field name="view_id" ref="form_all_customers" />
    <field name="view_mode">form</field>
    <field name="sequence">10</field>
</record>
```

Now if you open your menu and click some partner in the list, you should see the very minimal form we just defined.

How it works...

This time, we used the generic XML code for any type of record that is, the `record` element with the required attributes `id` and `model`. As earlier, the `id` attribute is an arbitrary string that must be unique for your addon. The `model` attribute refers to the name of the model you want to create. Given that we want to create a view, we need to create a record of model `ir.ui.view`. Within this element, you set fields as defined in the model you chose via the `model` attribute. For `ir.ui.view`, the crucial fields are `model` and `arch`. The `model` field contains the model you want to define a view for, while the `arch` field contains the definition of the view itself. We'll come to its contents in a short while.

The `name` field, while not strictly necessary, is helpful when debugging problems with views, so set it to some string that tells you what this view is intended to do. This field's content is not shown to the user, so you can fill in all the technical hints to yourself that you deem sensible. If you set nothing here, you'll get a default containing the model name and view type.

ir.actions.act_window.view

The second record we defined works in unison with `act_window` we defined previously. We know already that by setting the field `view_id` there, we can select which view is used for the first view mode. But given we set the field `view_mode` to the default *tree, form*, `view_id` would have to pick a tree view, but we want to set the form view, which comes second here.

If you find yourself in a situation like this, use the model `ir.actions.act_window.view`, which gives you fine grained control over which views to load for which view type. The first two fields defined here are an example of the generic way to refer to other objects; you keep the element's body empty but add an attribute called `ref` which contains the XML ID of the object you want to reference. So what happens here is that we refer to our action from the previous recipe in the `act_window_id` field, and refer to the view we just created in the `view_id` field. Then, though not strictly necessary, we add a sequence number to position this view assignment relatively to the other view assignments for the same action. This is only relevant if you assign views for different view modes by creating multiple `ir.actions.act_window.view` records.

> Once you define the `ir.actions.act_window.view` records, they take precedence over what you filled in the action's `view_mode` field. So with only the above records, you won't see a list at all, but only a form. So you should add another `ir.actions.act_window.view` record pointing to a list view.

Adding content and widgets to a form view

The previous recipe showed how to pick a specific view for an action. Now we'll demonstrate how to make the form we defined previously more useful.

How to do it...

1. Define the form view basic structure:

```
<record id="form_all_customers" model="ir.ui.view">
  <field name="name">All customers</field>
  <field name="model">res.partner</field>
  <field name="arch" type="xml">
    <form>
      <!--form content goes here -->
    </form>
  </field>
</record>
```

2. To add a head bar, usually used for action buttons and stage pipeline, add inside the form:

```
      <header>
        <button type="object"
                name="open_commercial_entity"
                string="Open commercial partner"
                class="oe_highlight" />
      </header>
```

3. Add fields to the form, using group tags to visually organize them:

```
      <group string="Content" name="my_content">
        <field name="name" />
        <field name="category_id" widget="many2many_tags" />
      </group>
```

Now the form should display a top bar with a button and two vertically aligned fields.

How it works...

We'll look at the `arch` field of the `ir.ui.view` model. Here, everything that the user sees happens. First, note that views are defined in XML themselves, so you need to pass the attribute `type="xml"` for the arch field, otherwise the parser will be confused. It is also mandatory that your view definition contains well formed XML, otherwise you'll get in trouble when loading this snippet.

Now, we'll walk through the tags used previously and summarize the others that are available.

Form

When you define a form view, it is mandatory that the first element within the `arch` field is a `form` element. This fact is used internally to derive the record's `type` field, which is why you're not supposed to set this field. You'll see this a lot in legacy code though.

The `form` element can have two legacy attributes itself, which are `string` and `version`. In the previous versions of Odoo, those were used to decide on the title you saw in the breadcrumb and to differentiate between the forms written in pre-7.0 style and afterwards, but both can be considered obsolete by now. The title in the breadcrumb is now inferred from the model's `name_get` function, while the version is assumed to be 7.0 or above.

In addition to the elements listed next, you can use arbitrary HTML within the form tag. The algorithm is that every element unknown to Odoo is considered plain HTML and simply passed through to the browser. Be careful with that, as the HTML you fill in can interact with the HTML code the Odoo elements generate, which might distort the rendering.

Header

This element is a container for the elements that should be shown in a form's header, which is rendered as a gray bar. Usually, as in this example, you place action buttons here or a status bar if your model has a state field.

Button

The `button` element is used to allow the user to trigger an action. See the following recipe *Adding buttons to forms* for details.

Group

The `group` element is Odoo's main means for organizing content. Fields placed within a `group` element are rendered with their title, and all the fields within the same group are aligned so that there's also a visual indicator that they belong together. You can also nest the `group` elements; this causes Odoo to render the contained fields in adjacent columns.

In general, you should use this mechanism for all your fields and only revert to other methods (see next) when necessary.

If you assign the attribute `string` on a group, its content will be rendered as a heading for the group.

You should develop the habit of assigning a `name` to every logical group of fields too. This name is not visible to the user, but very helpful when we override views in the following recipes. Keep the name unique within a form definition to avoid confusion about which group you refer to.

Field

In order to actually show and manipulate data, your form should contain some `field` elements. They have one mandatory attribute `name`, which refers to the field's name in the model. So above, we offer the user to edit the partner's name and categories. If we only want to show one of them, without the user being able to edit the field, we set the attribute `readonly` to `1` or `True`. This attribute actually may contain a small subset of Python code, so `readonly="2>1"` would make the field read only too. The same applies to the attribute `invisible`, that you use to have the value read from the database, but not shown to the user. We'll see later in which situations we want to have that.

Take care not to put the same field twice on your form. Odoo is not designed to support this, and the result will be that all but one of those fields will behave as if they were empty.

You must have noticed the attribute `widget` on the categories field. It defines how the data in the field is supposed to be presented to the user. Every type of field has its standard widget, so you don't have to explicitly choose a widget. But several types provide multiple ways of representation, in which case you might opt for something else than the default. As a complete list of available widgets would exceed the scope of this recipe, you'll have to resort to Odoo's source code to try them out and consult *Chapter 15*, *Web Client Development* for details on how to make your own.

General attributes

On most elements (this includes `group`, `field`, and `button`), you can set the attributes `attrs` and `groups`. While `attrs` is discussed next, the `groups` attribute gives you the possibility to show some elements only to the members of certain groups. Simply put the group's XML ID (separated by commas for multiple groups) in the attribute, and the element will be hidden for everyone who is not a member of at least one of the groups mentioned.

Other tags

There are situations where you might want to deviate from the strict layout groups prescribed. A good example is, if you want the `name` field of a record to be rendered as a heading, the field's label would interfere with the appearance. In this case, don't put your field into a `group` element, but for example, into the plain HTML `h1` element. Then before the `h1` element, put a `label` element with the `for` attribute set to your field name:

```
<label for="name" />
<h1><field name="name" /></h1>
```

This will be rendered with the field's content as a big heading, but the field's name only small above. That's basically what the standard partner form does.

If you need a line break within a group, use the `newline` element. It's always empty.

```
<newline />
```

Another useful element is the `footer` element. When you open a form as a popup, this is the good place to place the action buttons. It will be rendered as a gray bar too, analogous to the `header` element.

There's more...

Since form views are basically HTML with some extensions, Odoo also makes extensive use of CSS classes. Two very useful ones are `oe_read_only` and `oe_edit_only` – they cause elements with these classes applied to be visible only in read/view mode or only in edit mode. So to have the label visible only in edit mode:

```
<label for="name" class="oe_edit_only" />
```

Further, the `form` element can have attributes `create`, `edit`, and `delete`. If you set one of those to `false`, the corresponding action won't be available for this form. Without this being explicitly set, the availability of the action is inferred from the user's permissions. Note that this is purely for straightening the UI; don't use this for security.

See also

The widgets described earlier already offer a lot of functionality, but sooner or later you will encounter cases where they don't do exactly what you want. For defining your own widgets, refer to *Chapter 15*, *Web Client Development*.

Adding buttons to forms

We added a button in the previous form, but there are quite different types of buttons to use. This recipe will add another button; also put the following code in to the previous recipe's `header` element.

How to do it...

Add a button referring to an action:

```
<button type="action" name="%(base.action_partner_category_form)d"
string="Open partner categories" />
```

How it works...

The button's type attribute determines the semantics of the other fields, so we'll first look into the possible values:

- `action`: This makes the button call an action as defined in the **ir.actions.*** namespace. The `name` attribute needs to contain the action's database id, which you can conveniently look up with a python format string containing the XML ID of the action in question.
- `object`: This calls a method of the current model. The `name` attribute contains the function's name. The function should have the signature `@api.multi` and will act on the currently viewed record.
- `workflow`: This sends a workflow signal to the current record. The `name` attribute needs to contain the signal's name. Use the `states` attribute to make the workflow buttons only available in states where they actually do something, that is, in states that have transitions triggered by the signal in question.

The `string` attribute is used to assign the text the user sees.

There's more...

Use CSS classes `btn-primary` to render a button that is highlighted (currently blue) and `btn-default` to render a normal button. This is commonly used for the cancel buttons in wizards or to offer secondary actions in a visually unobtrusive way.

The call with a button of type **object** can return a dictionary describing an action, which will then be executed on the client side. This way you can implement multi screen wizards or just open some other record.

You can also have content within the `button` tag, replacing the `string` attribute. This is commonly used in button boxes as described in the recipe *Document style forms*.

Passing parameters to forms and actions: Context

Internally, every method in Odoo has access to a dictionary called **context** that is propagated from every action to the methods involved in delivering that action. The UI also has access to it and can be modified in various ways by setting values in the context. In this recipe, we'll explore some of the applications of this mechanism by toying with the language, default values, and implicit filters.

Getting ready

While not strictly necessary, this recipe will be more fun if you install the French language, in case you didn't start out with this language in the first place. Consult *Chapter 11, Internationalization*, for how to do this. If you have a French database, change **fr_FR** to some other language; **en_US** will do for English. Also, click the button **Not archived** on one of your customers in order to archive it and verify that this partner doesn't show up any more in the list.

How to do it...

1. Create a new action, very similar to the one from the first recipe:

   ```
   <act_window id="action_all_customers_fr"
               name="Tous les clients"
               res_model="res.partner"
               domain="[('customer', '=', True)]"
               context="{'lang': 'fr_FR', 'default_lang': 'fr_FR',
                   'active_test': False}" />
   ```

2. Add a menu that calls this action. This is left as an exercise for the reader.

When you open this menu, the views will show up in French, and if you create a new partner, she will have French as the preselected language. A less obvious difference is that you will also see the partner you deactivated earlier.

How it works...

The context dictionary is populated from several sources. First, some values from the current user's record (`lang` and `tz`, for the user's language and the user's timezone) are read, then there are addons that add keys for their own purposes. Further, the UI adds keys about which model and which record we're busy with at the moment (`active_id`, `active_ids`, `active_model`). And as seen previously, we can also add our own keys in actions. Those are merged together and passed to the underlying server functions, and also to the client side UI.

So by setting the `lang` context key, we force the display language to be French. You will note that this doesn't change the whole UI language, which is because only the list view that we open lies within the scope of this context. The rest of the UI was loaded already with another context that contained the user's original language. But if you open a record in this list view, it will be presented in French too, and if you open some linked record on the form or press a button that executes an action, the language will be propagated too.

By setting default_lang, we set a default value for every record created within the scope of this context. The general pattern is default_$fieldname: my_default_value, which enables you to set default values for the partners newly created in this case. Given that our menu is about customers, it might have made sense to also set default_customer: True to have the **Customer** field checked by default. But this is a model wide default for res.partner anyway, so this wouldn't have changed anything. For scalar fields, the syntax for this is as you would write it in Python code – string fields go in quotes, numbers just like that, and boolean fields are either True or False. For relational fields, the syntax is slightly more complicated – refer to *Chapter 9*, *Module Data*, for how to write those. Note that the default values set in the context override the default values set in the model definition, so you can have different default values in different situations.

The last key is active_test, which has very special semantics. For every model that has a field called **active**, Odoo automatically filters out the records where this field is False. This is why the partner where you unchecked this field disappeared from the list. By setting this key, we can suppress this behavior.

This is useful for the UI in its own right, but even more useful in your Python code when you need to be sure some operation is applied to all the records, not just the active ones.

There's more...

While defining a context you have access to some variables, the most important one being uid, which evaluates to the current user's ID. You'll need this for setting the default filters (see the next recipe). Further, you have access to the *function* context_today and the *variable* current_date, where the first is a date object representing the current date as seen from the user's time zone and the latter the current date as seen in UTC, formatted as YYYY-MM-DD. For setting a default for a date field to the current date by default, use current_date and, for default filters, use context_today().

Further, you can do some date calculations with a subset of Python's datetime, time, and relativedelta classes.

Why the repeated talk about 'a subset of Python'? For various technical reasons, domains have to be evaluated as well on the client side as on the server side. Server side evaluation existed earlier, there full Python was available (but is restricted by now too for security reasons). When client side evaluation was introduced, the best option in order not to break the whole system was to implement a part of Python in JavaScript. So there is a small JavaScript Python interpreter built into Odoo which works great for simple expressions, and that is usually enough.

Beware about variables in the context in conjunction with the `<act_window />` shortcut. Those are evaluated at installation time, which is nearly never what you want. If you need variables in your context, use `<record />` syntax.

The same way that we added some context keys in our action, we can do with buttons. This causes the function or action the button calls to be run in the context given and by now you know some of the tricks you can pull this way.

Most form element attributes that are evaluated as Python also have access to the context dictionary. The attributes `invisible` and `readonly` are such attributes. So in cases where you want an element to show up in a form sometimes, but not at other times, you set the `invisible` attribute to `context.get('my_key')`, and for actions that lead to the case where the field is supposed to be invisible, you set the context key `my_key: True`. Such a strategy enables you to adapt your form without having to rewrite it for different occasions.

You can also set a context on relational fields, which influences how the field is loaded. By setting the keys `form_view_ref` or `tree_view_ref` to the XML ID of a view, you can select a specific view for this field. This is necessary when you have multiple views of the same type for the same object. Without this key you get the view with the lowest sequence number, which might not always be desirable.

See also

One of the very useful applications of the context is to set default search filters as described in the recipe *Search views*.

Defining filters on record lists: Domain

We've already seen the first example of a domain in the first action, which was `[('customer', '=', True)]`. It is a very common use case when you need to display a subset of all available records from an action, or to allow only a subset of possible records to be the target of a many2one relation. The way to describe these filters in Odoo is called a domain. This recipe illustrates how to use such a domain to display a selection of partners.

How to do it...

To display a subset of partners from your action, you need to perform the following steps:

1. Add an action for non-French speaking customers:

```
<record id="action_my_customers" model="ir.actions.act_window">
    <field name="name">
        All customers who don't speak French
    </field>
    <field name="res_model">res.partner</field>
    <field name="domain">
        [('customer', '=', True), ('user_id', '=', uid), ('lang',
'!=', 'fr_FR')]
    </field>
</record>
```

2. Add an action for the customers who are customers or suppliers:

```
<record id="action_customers_or_suppliers"
        model="ir.actions.act_window">
    <field name="name">Customers or suppliers</field>
    <field name="res_model">res.partner</field>
    <field name="domain">
        ['|', ('customer', '=', True), ('supplier', '=', True)]
    </field>
</record>
```

3. Add menus that call this action. This is left as an exercise for the reader.

How it works...

The simplest form of a domain is a list of 3-tuples that contains a field name of the model in question as string in the first element, an operator as string in the second element, and the value the field is to be checked against as the third element. This is what we did in the first action, and this is interpreted as: All those conditions have to apply to the records we're interested in. This actually is a shortcut, because the domains know the two prefix operators `&` and `|`, where `&` is the default. So, in formal form, the first domain would be written as `['&', '&', ('customer', '=', True), ('user_id', '=', uid), ('lang', '!=', 'fr_FR')]`.

While a bit hard to read for bigger expressions, the advantage of prefix operators is that their scope is rigidly defined, which saves you having to worry about operator precedence and brackets. It's always the following two expressions: The first `&` applies to `'&', ('customer', '=', True), ('user_id', '=', uid)` as the first operand and `('lang', '!=', 'fr_FR')` as the second. Then, the second `&` applies to `('customer', '=', True)` as the first operand and `('user_id', '=', uid)` as the second.