Actions

Actions define the behavior of the system in response to user actions: login, action button, selection of an invoice, ...

Actions can be stored in the database or returned directly as dictionaries in e.g. button methods. All actions share two mandatory attributes:

`type`
the category of the current action, determines which fields may be used and how the action is interpreted

`name`
short user-readable description of the action, may be displayed in the client's interface

A client can get actions in 4 forms:

`False`
if any action dialog is currently open, close it

A string
if a client action matches, interpret as a client action's tag, otherwise treat as a number

A number
read the corresponding action record from the database, may be a database identifier or an external id (../glossary.html#term-external-id)

A dictionary
treat as a client action descriptor and execute

Bindings

Aside from their two mandatory attributes, all actions also share *optional* attributes used to present an action in an arbitrary model's contextual menu:

`binding_model_id`
specifies which model the action is bound to

> For Server Actions, use **`model_id`** .

`binding_type`
specifies the type of binding, which is mostly which contextual menu the action will appear under

`action` (default)

Specifies that the action will appear in the **Action** contextual menu of the bound model.

`report`
Specifies that the action will appear in the **Print** contextual menu of the bound model.

`binding_view_types`
a comma-separated list of view types for which the action appears in the contextual menu, mostly "list" and / or "form". Defaults to `list,form` (both list and form)

Window Actions (`ir.actions.act_window`)

The most common action type, used to present visualisations of a model through views (views.html#reference-views): a window action defines a set of view types (and possibly specific views) for a model (and possibly specific record of the model).

Its fields are:

`res_model`
model to present views for

`views`
a list of `(view_id, view_type)` pairs. The second element of each pair is the category of the view (tree, form, graph, ...) and the first is an optional database id (or `False`). If no id is provided, the client should fetch the default view of the specified type for the requested model (this is automatically done by `fields_view_get()` (orm.html#odoo.models.Model.fields_view_get)). The first type of the list is the default view type and will be open by default when the action is executed. Each view type should be present at most once in the list

`res_id` (optional)
if the default view is `form` , specifies the record to load (otherwise a new record should be created)

`search_view_id` (optional)
`(id, name)` pair, `id` is the database identifier of a specific search view to load for the action. Defaults to fetching the default search view for the model

`target` (optional)
whether the views should be open in the main content area (`current`), in full screen mode (`fullscreen`) or in a dialog/popup (`new`). Use `main` instead of `current` to clear the breadcrumbs. Defaults to `current` .

`context` (optional)
additional context data to pass to the views

`domain` (optional)
filtering domain to implicitly add to all view search queries

`limit` (optional)
number of records to display in lists by default. Defaults to 80 in the web client

For instance, to open customers (partner with the `customer` flag set) with list and form views:

```
{
    "type": "ir.actions.act_window",
    "res_model": "res.partner",
    "views": [[False, "tree"], [False, "form"]],
    "domain": [["customer", "=", true]],
}
```

Or to open the form view of a specific product (obtained separately) in a new dialog:

```
{
    "type": "ir.actions.act_window",
    "res_model": "product.product",
    "views": [[False, "form"]],
    "res_id": a_product_id,
    "target": "new",
}
```

In-database window actions have a few different fields which should be ignored by clients, mostly to use in composing the `views` list:

`view_mode` (default= `tree,form`)
comma-separated list of view types as a string (/!\ No spaces /!\). All of these types will be present in the generated `views` list (with at least a `False` view_id)

`view_ids`
M2M[1] to view objects, defines the initial content of `views`

> Act_window views can also be defined cleanly through `ir.actions.act_window.view` .
> If you plan to allow multiple views for your model, prefer using ir.actions.act_window.view instead of the action `view_ids`
>
> ```
> <record model="ir.actions.act_window.view" id="test_action_tree">
>    <field name="sequence" eval="1"/>
>    <field name="view_mode">tree</field>
>    <field name="view_id" ref="view_test_tree"/>
>    <field name="act_window_id" ref="test_action"/>
> </record>
> ```

`view_id`
specific view added to the `views` list in case its type is part of the `view_mode` list and not already filled by one of the views in `view_ids`

These are mostly used when defining actions from Data Files (data.html#reference-data):

```
<record model="ir.actions.act_window" id="test_action">
    <field name="name">A Test Action</field>
    <field name="res_model">some.model</field>
    <field name="view_mode">graph</field>
    <field name="view_id" ref="my_specific_view"/>
</record>
```

will use the “my_specific_view” view even if that’s not the default view for the model.

The server-side composition of the `views` sequence is the following:

- get each `(id, type)` from `view_ids` (ordered by `sequence`)
- if `view_id` is defined and its type isn’t already filled, append its `(id, type)`
- for each unfilled type in `view_mode`, append `(False, type)`

[1] technically not an M2M: adds a sequence field and may be composed of just a view type, without a view id.

URL Actions (`ir.actions.act_url`)

Allow opening a URL (website/web page) via an Odoo action. Can be customized via two fields:

`url`
the address to open when activating the action

`target`
opens the address in a new window/page if `new`, replaces the current content with the page if `self`. Defaults to `new`

```
{
    "type": "ir.actions.act_url",
    "url": "https://odoo.com",
    "target": "self",
}
```

will replace the current content section by the Odoo home page.

Server Actions (`ir.actions.server`)

Allow triggering complex server code from any valid action location. Only two fields are relevant to clients:

`id`
the in-database identifier of the server action to run

`context` (optional)
context data to use when running the server action

In-database records are significantly richer and can perform a number of specific or generic actions based on their `state`. Some fields (and corresponding behaviors) are shared between states:

`model_id`
Odoo model linked to the action.

`state`

- `code` : Executes python code given through the `code` argument.
- `object_create` : Creates a new record of model `crud_model_id` following `fields_lines` specifications.
- `object_write` : Updates the current record(s) following `fields_lines` specifications
- `multi` : Executes serveral actions given through the `child_ids` argument.

State fields

Depending on its state, the behavior is defined through different fields. The concerned state is given after each field.

`code` (code)
Specify a piece of Python code to execute when the action is called

```
<record model="ir.actions.server" id="print_instance">
    <field name="name">Res Partner Server Action</field>
    <field name="model_id" ref="model_res_partner"/>
    <field name="state">code</field>
    <field name="code">
        raise Warning(record.name)
    </field>
</record>
```

The code segment can define a variable called `action`, which will be returned to the client as the next action to execute:

```
<record model="ir.actions.server" id="print_instance">
    <field name="name">Res Partner Server Action</field>
    <field name="model_id" ref="model_res_partner"/>
    <field name="state">code</field>
    <field name="code">
        if record.some_condition():
            action = {
                "type": "ir.actions.act_window",
                "view_mode": "form",
                "res_model": record._name,
                "res_id": record.id,
            }
    </field>
</record>
```

> will ask the client to open a form for the record if it fulfills some condition

`crud_model_id` (create)(required)
model in which to create a new record

`link_field_id` (create)
many2one to `ir.model.fields` , specifies the current record's m2o field on which the newly created record should be set (models should match)

`fields_lines` (create/write)
fields to override when creating or copying the record. `One2many` (orm.html#odoo.fields.One2many) with the fields:

`col1`
`ir.model.fields` to set in the concerned model (`crud_model_id` for creates, `model_id` for updates)

`value`
value for the field, interpreted via `type`

`type` (value|reference|equation)
If `value` , the `value` field is interpreted as a literal value (possibly converted), if `equation` the `value` field is interpreted as a Python expression and evaluated

`child_ids` (multi)
Specify the multiple sub-actions (`ir.actions.server`) to enact in state multi. If sub-actions themselves return actions, the last one will be returned to the client as the multi's own next action

Evaluation context

A number of keys are available in the evaluation context of or surrounding server actions:

`model` model object linked to the action via `model_id`

`record` / `records` record/recorset on which the action is triggered, can be void.

`env` Odoo Environment

`datetime` , `dateutil` , `time` , `timezone` corresponding Python modules

`log: log(message, level='info')` logging function to record debug information in ir.logging table

`Warning` constructor for the `Warning` exception

Report Actions (`ir.actions.report`)

Triggers the printing of a report.

If you define your report through a `<record>` instead of a `<report>` tag and want the action to show up in the Print menu of the model's views, you will also need to specify `binding_model_id` from Bindings. It's not necessary to set `binding_type` to `report`, since `ir.actions.report` will implicitly default to that.

`name` (mandatory)
used as the file name if `print_report_name` is not specified. Otherwise, only useful as a mnemonic/description of the report when looking for one in a list of some sort

`model` (mandatory)
the model your report will be about

`report_type` (default=qweb-pdf)
either `qweb-pdf` for PDF reports or `qweb-html` for HTML

`report_name` (mandatory)
the name (external id (../glossary.html#term-external-id)) of the qweb template used to render the report

`print_report_name`
python expression defining the name of the report.

`groups_id`
`Many2many` (orm.html#odoo.fields.Many2many) field to the groups allowed to view/use the current report

`multi`
if set to `True`, the action will not be displayed on a form view.

`paperformat_id`
`Many2one` (orm.html#odoo.fields.Many2one) field to the paper format you wish to use for this report (if not specified, the company format will be used)

`attachment_use`
if set to `True`, the report is only generated once the first time it is requested, and re-printed from the stored report afterwards instead of being re-generated every time.

Can be used for reports which must only be generated once (e.g. for legal reasons)

`attachment`
python expression that defines the name of the report; the record is accessible as the variable `object`

Client Actions (`ir.actions.client`)

Triggers an action implemented entirely in the client.

tag
: the client-side identifier of the action, an arbitrary string which the client should know how to react to

params (optional)
: a Python dictionary of additional data to send to the client, alongside the client action tag

target (optional)
: whether the client action should be open in the main content area (`current`), in full screen mode (`fullscreen`) or in a dialog/popup (`new`). Use `main` instead of `current` to clear the breadcrumbs. Defaults to `current` .

```
{
    "type": "ir.actions.client",
    "tag": "pos.ui"
}
```

tells the client to start the Point of Sale interface, the server has no idea how the POS interface works.

> ➡ **See also**
>
> **Tutorial: Client Actions (../howtos/web.html#howtos-web-client-actions)**

Automated Actions (`ir.cron`)

Actions triggered automatically on a predefined frequency.

name
: Name of the automated action (Mainly used in log display)

interval_number
: Number of *interval_type* uom between two executions of the action

interval_type
: Unit of measure of frequency interval (`minutes` , `hours` , `days` , `weeks` , `months` ,

numbercall
: Number of times this action has to be run. If the action is expected to run indefinitely, set to `-1` .

doall
: Boolean precising whether the missed actions have to be executed in case of server restarts.

model_id
: Model on which this action will be called

code
: Code content of the action. Can be a simple call to the model's method :

```
model.<method_name>()
```

`nextcall`
Next planned execution date of this action (date/time format)