odoo
PARTNERS

Partner Training

Introduction to Development

Odoo PS-Tech

odoo
PARTNERS

Chapter 0

Introduction

Welcome

CLICK HERE

- Odoo development
- Odoo Framework
- Customize Module
- Business Objects
- Security Intro
- Building UI
- Other resources

Technical prerequisites

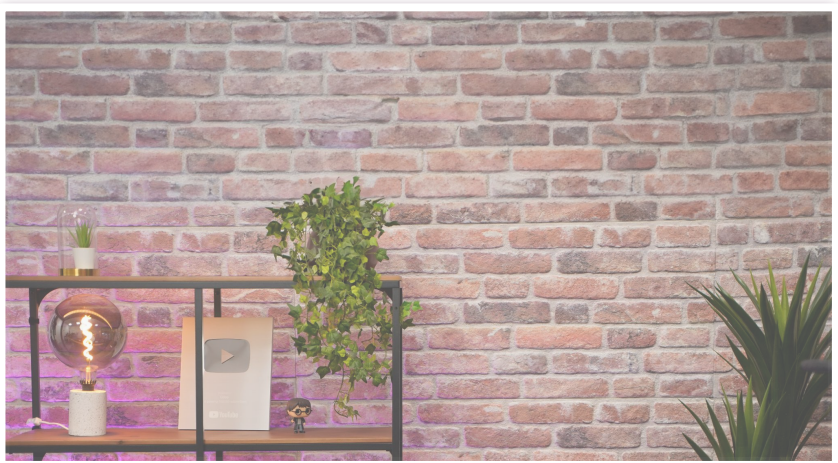
Odoo's ORM is written in Python.

Front-end in Odoo is declared using XML.

For this lesson, the use of JS won't be required.

Odoo uses PostgreSQL. There are some cases where understanding the ORM requires basic knowledge of SQL queries.

Before starting



Installation *prerequisites*

CLICK HERE

- check if the installation was successful

./odoo-bin --addons-path=<odoo-addons>,<enterprise> -d test

The project

Immo. Do Inc.

Mid-size real estate company

odoo standard → Real Estate App

Versatile ERP → Custom app fit standard & client's need

Exercise

Chapter X

Exercise 1

Exercise

Chapter X

~~Exercise 1~~

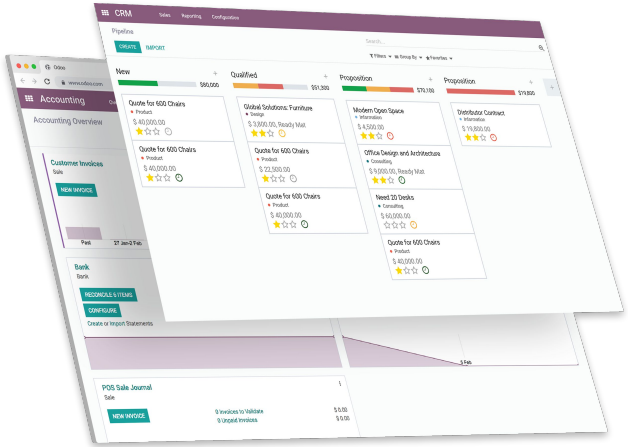
Exercise 2

odoo
PARTNERS

Chapter 1

Module structure

What's a module



- Odoo = modular ERP
- 1 module = 1 set of features

Adaptable

Install standard features to meet your needs

Customizable

Create new modules or extend existing ones easily

Developer Mode

Module composition

Module = Directory

Models (Business Objects)
Declared as Python classes using __init__.py file. Models definition for the data structure and the business logic of the module

Views
Definition of the user interface to display information on screen and interact with end users

Data Files
Data to load at the installation of the module; may contain configuration data, security rules, demo data, etc

```
project_directory/
├── module_a/
│   ├── controllers/
│   │   ├── __init__.py
│   │   └── controller.py
│   ├── data/
│   │   └── data.xml
│   ├── models/
│   │   ├── __init__.py
│   │   └── model.py
│   ├── security/
│   │   └── ir.model.access.csv
│   ├── views/
│   │   ├── templates.xml
│   │   └── views.xml
│   ├── __init__.py
│   └── __manifest__.py
└── module_b/
    ├── ...
    └── ...
```

Manifest structure

__manifest__.py

```
{
    "name": "Module Name",
    "summary": """Small summary about the module""",
    "version": "1X.0.0.0.0",
    "author": "My Company",
    "website": "https://www.mycompany_web.com",
    "license": "XXXX-X",
    "depends": ["module_a","module_b"],
    "data": [
        "views/mymodule_view.xml",
    ],
    "demo": [
        "demo/demo_data.xml",
    ],
}
```

CLICK HERE

- **name** (str, required)
- **version** (str)
- **description** (str)
- **author** (str)
- **website** (str)
- **license** (str, defaults: LGPL-3)
- **category** (str, default: Uncategorized)
- **depends** (list(str))
  Odoo modules which must be loaded before this one.
- **data** (list(str))
  List of paths of data files which must always be installed or updated with the module.
- **demo** (list(str))
  List of data files which are only installed or updated in demonstration mode.
- **auto_install** (bool or list(str), default: False)
  If True, this module will automatically be installed if all of its dependencies are installed.
- **external_dependencies** (dict(key=list(str)))
  A dictionary containing python and/or binary dependencies.
- **assets** (dict) *(v15+)*
- **auto_install** (boolean)
- **application** (boolean)
- ...

Licenses

2 types of available **licences**

Open source

- **GPL-2**
- **GPL-3**
- **AGPL-3**
- **LGPL-3**

Odoo Proprietary

- **OEEL-1** - *Odoo Enterprise Edition License v1.0*
- **OPL-1** - *Odoo Proprietary License v1.0*

CLICK HERE

Exercise

Chapter 1

New Module

1) Create a new module estate in the following folders

```
$HOME/$USER/src/technical-training/estate
```

2) In your new module, add the `__init__.py` and the `__manifest__.py` files.

3) In the `__manifest__.py` dictionary, define a name, a version and a license. Also, in the dependencies, specify for now the module `base`.

4) Use the application key to make your module an App.

5) Run the command line and check your new module appears in the applications list.

odoo
PARTNERS

Chapter 2

Models & Fields

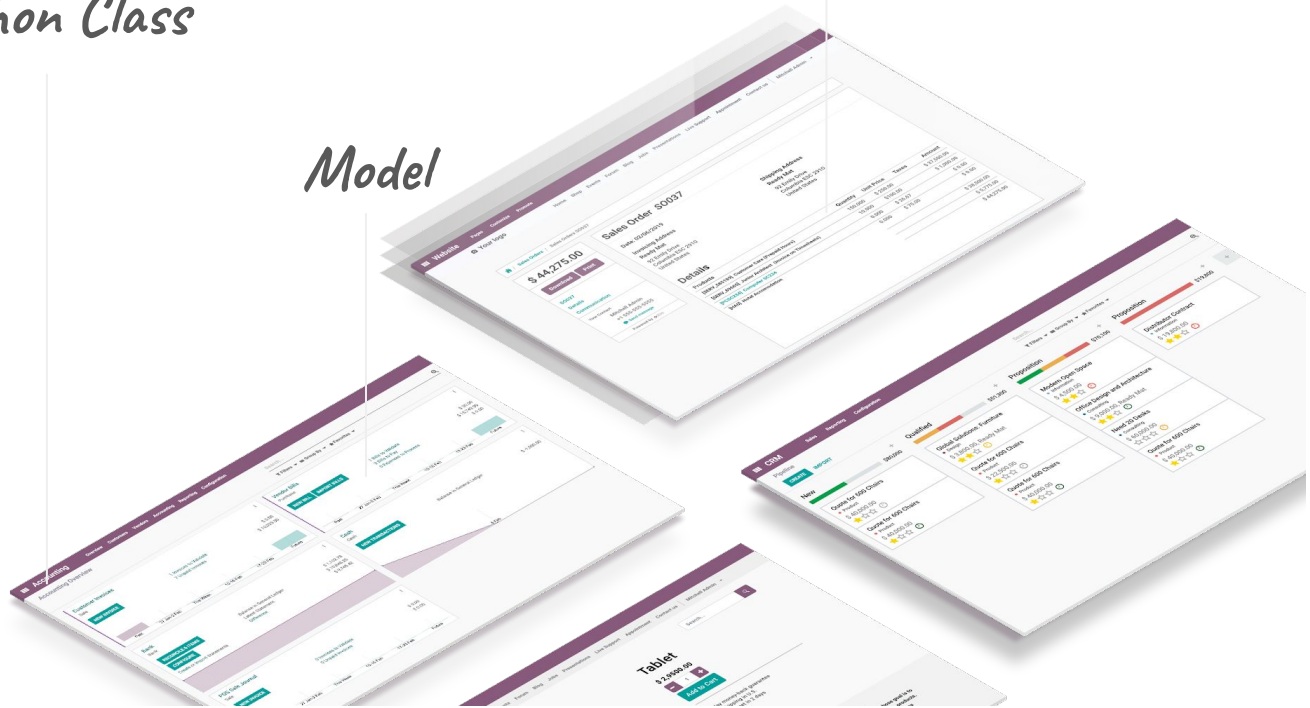
Object-Relational Mapping

SQL table

Python Class

Model

CLICK HERE



Models

model.py

```python
from odoo import models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"
```

CLICK HERE

1. The model is defined in the file `module/models/model.py`

2. The file `model.py` is imported in `module/models/__init__.py`

3. The folder `models` is imported in `module/__init__.py`

```
module/
│
├── models/
│   ├── __init__.py     (2)
│   └── model.py        (1)
│
└── __init__.py         (3)
```

Fields

*Define **what** the **model** can store and **where** it is stored.*

***Fields** are defined as **attributes** in the model.*

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Boolean()
    field_2 = fields.Integer()
```

Fields

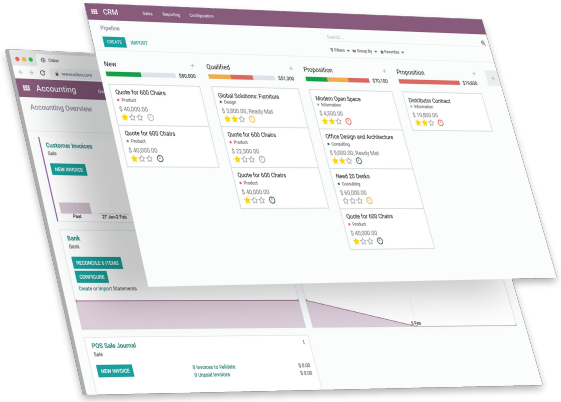
Two categories:

Simple fields *are atomic values stored directly in the model's table*

Relational fields *are link between records (of the same or different models).*

CLICK HERE

Simple fields



Boolean - *True / False*

Char & Text - *character string*

Selection - *list of string options*

Float & Integer - *numbers*

Date & Datetime - *dates*

Binary - *to store files, ...*

HTML - *formatted text*

Fields attributes

- **String** - *field name displayed in views*
- **Invisible** - *True or False, displays the field or not in the views*
- **Readonly** - *True or False, if true then field cannot be edited by user*
- **Required** - *True or False, if true then record cannot be saved in database with an empty/null value*
- **Default** - *Sets value on the field automatically during record creation*
- **...**

```
info = fields.Char(string="Test", required=True, default="test")
```

Exercise

odoo PARTNERS

Chapter 2

New model

Define the real estate properties **model**.

1) In the **estate** module, add a new directory to store our models python files.

2) In this directory, create a new python file for the `estate_property` table (don't forget to add the `__init__` too).

3) When the files are created, add a minimum definition for the `estate.property` model.

4) Add a `_description` to your model to get rid of one of the warnings.

Help: For now, just remember that in the future, we will try to have one model per file in order to keep the project as object-oriented and clean as possible.

Exercise

odoo PARTNERS

Chapter 2

~~New model~~

Add fields

Add basic fields to the Real Estate Property table.

Field	Type
name	Char
description	Text
postcode	Char
date_availability	Date
expected_price	Float
selling_price	Float
bedrooms	Integer
living_area	Integer
facades	Integer

Exercise

odoo PARTNERS

Chapter 2

~~New model~~

~~Add fields~~

Add fields (2)

Add basic fields to the Real Estate Property table.

Field	Type
garage	Boolean
garden	Boolean
garden_area	Integer
garden_orientation	Selection

The `garden_orientation` field must have 4 possible values:
'North', 'South', 'East' and 'West'.

The selection list is defined as a list of tuples
(see link below for example)

CLICK HERE

Exercise

Set **attributes** for the following fields

Chapter 2

~~New model~~

~~Add fields~~

~~Add fields~~ (2)

Fields attributes

Field	Attribute
name	required
expected_price	required

odoo PARTNERS

Chapter 3

Data introduction

Data types

Master Data

VS

Demo Data

CLICK HERE

Master Data

Essential for the module to work properly.

- **Technical** : *Views, actions, groups...*
  (see next chapters)
- **Business** : *countries, currencies, units of measure, ...*

Loaded when you **install/update** the module

CLICK HERE

Demo Data

Used for **testing** or **demonstration** purposes

- *Help* **Sales** *to make quick demos.*
- *Working data for* **Developers** *to* **test** *new features.*
- **Testing** *that loading data doesn't raise an error.*

Loaded when you **start** the server.

CLICK HERE

Data declaration

*In the **manifest** of the module*

***Located** in different **directories** such as data, views, ...*

```
project_directory/
└── module_a/
    ├── data/
    │   └── data.xml
    ├── security/
    │   └── ir.model.access.csv
    ├── views/
    │   ├── templates.xml
    │   └── views.xml
    └── __manifest__.py
```

Essentially XML files, but can also be CSV files or other types of files.

CLICK HERE

Exercise

odoo PARTNERS

Chapter 3

New model data

Create **one demo data** in previously created model.

1) In the **estate** module, add a new directory to store our **demo** data files.

2) In this directory, create a new xml file for the demo data `demo.xml` (don't forget to add the "demo" key in the manifest `__manifest__` too).

3) In the xml, add one record for the `estate.property` model.
Don't forget to add at least all the required fields in the latter.

Help: To populate a DB using the command line, you can also use the following command:

CLICK HERE

odoo-bin populate –addons-path=... -d {database} –size {small|medium|large} –models {model1,model2}

Exercise

odoo PARTNERS

Chapter 3

~~New model data~~

Groups

Create **a new group** of admin users for our model.

1) In the **estate** module, add a new directory to store our **security** data files.

2) In this directory, create a new xml file for the groups `res_groups.xml` (don't forget to add the "data" key in the manifest `__manifest__` too).

3) In the xml, add one record for the `res.groups` model.
The id of this group will be **estate_admin** as it will be used to set the administrators of our model.
The only required field for this model is the **name** (e.g. Estate Manager).

odoo
PARTNERS

Chapter 4

Introduction to Security

Access Rights

The ir.model.access **CSV** *file for is used to create* **security access rules**.

Contains the following columns:

- **id** - *external identifier for the access rule.*
- **name** - *name of the access rule.*
- **model_id:id** - *ID of the model that the access rule applies to.*
- **group_id:id** - *ID of the group that the access rule applies to.*
- **perm_read** - *read permission for the access rule.*
- **perm_write** - *write permission for the access rule.*
- **perm_create** - *create permission for the access rule.*
- **perm_unlink** - *unlink permission for the access rule.*

CLICK HERE

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_test_model,access_test_model,model_test_model,base.group_user,1,0,0,0
```

Exercise

odoo PARTNERS

Chapter 4

Access rights

Create **access rights** for the new custom model.

1) In the **estate** module, in the directory to store our **security** data files created in the last chapter, add a new CSV access file `ir.model.access.csv` .

2) In addition to the header line, add 2 different access rights lines.

→ 1 with readonly permission for **base.group_user**
→ 1 full access for custom admin group **module_name.estate_admin**

3) Don't forget to add the file in the `__manifest__.`

Help: Remember the example given in the warning of in the command line while loading DB. To gain some time, you can copy/paste the warning there in the manifest

odoo
PARTNERS

Chapter 5

Basic actions

Action

Use

- Open different types of window
- Open specific data/set of data

Triggered by

- clicking on menu items
- clicking on buttons in views
- contextual actions on object

CLICK HERE

Action

- **id** - *external identifier*
- **model** - *fixed value of ir.actions.act_window*
- **name** - *name of the action*
- **res_model** - *model which the action applies to*
- **view_mode** - *views that will be available (tree, form, ...)*

```
<record id="test_model_action"
                model="ir.actions.act_window">
    <field name="name">Test action</field>
    <field name="res_model">test.model</field>
    <field name="view_mode">tree,form</field>
</record>
```

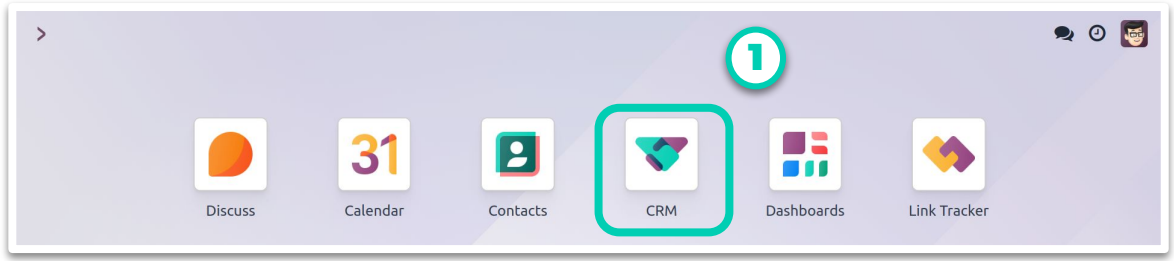
CLICK HERE

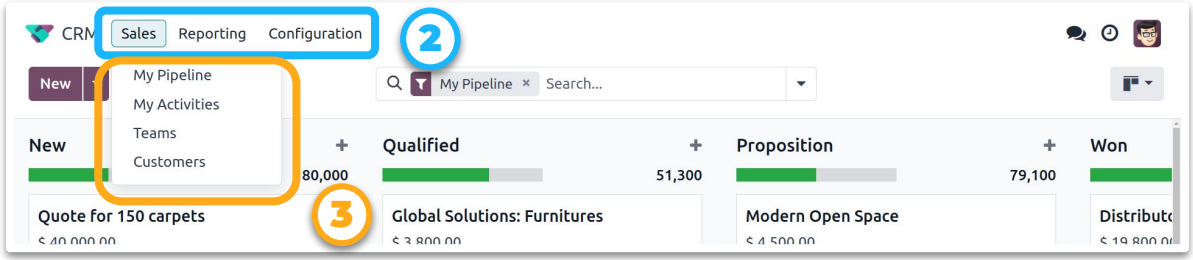
Menus

Using shortcuts, menus declaration will follow a three level architecture:

1. **root** *menu*
2. **first level** *menu, displayed in the top bar.*
3. **actions** *menus*

CLICK HERE






```
<menuitem id="test_menu_root" name="Test">
    <menuitem id="test_app_level_menu" name="App Level">
        <menuitem id="test_model_menu_action" action="test_model_action"/>
    </menuitem>
</menuitem>
```

1 2 3

Exercise

odoo PARTNERS

Chapter 5

Actions

Add **a new action** for the new custom model.

1) In the **estate** module, add a new directory to store our **views** data files.

2) In this directory, create a new xml file `estate_property_views.xml` to store the views and actions for the related model (don't forget to define it the manifest `__manifest__` too).

3) Create an action for the model `estate.property`.
To do so, create a record in the `ir.actions.act_window` model.
Then add a **name**, **res_model** and **view_mode** in the fields.
NB: set "tree,form" in the view modes.

Exercise

odoo PARTNERS

Chapter 5

~~Actions~~

Menu

Add **menus** for the new custom model.

1) In the **estate** module, in the previously created **views** directory create a new xml file `estate_menu.xml` to store the menus of the module (don't forget to define it the manifest `__manifest__` too).

2) Create the three levels of menus for the estate.property action created in the previous exercise.

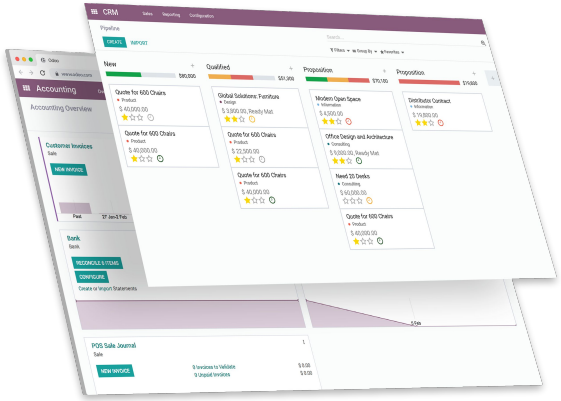
Help: Remember that we create different groups to give different types of access. If you want to be able to create or modify records, you might need to add your user in the right group.

odoo
PARTNERS

Chapter 6

Basic views

Fields attributes (again)



- string
- readonly
- required
- default
- domain

*Modification in the **views** can be done with **python** code using the **fields attributes**.*

```
info = fields.Char(string="Test", required=True, default="test")
```

Exercise

odoo PARTNERS

Chapter 6

Field view attrs

Add new **attributes** to the **fields**.

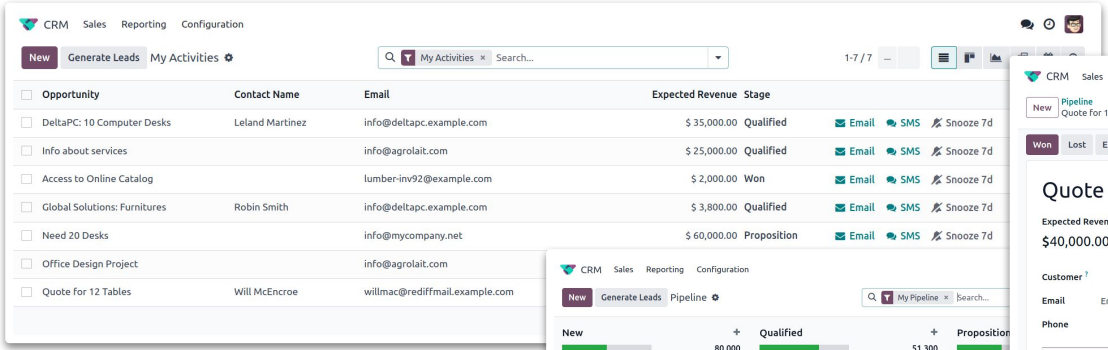
1) set the selling price as **read-only.**
2) prevent **copying** of the availability date and the selling price values.
3) the **default** number of bedrooms is 2
4) the **default** availability date is in 3 months
   → *Tip: this might help you:* `today()`
5) Add fields to the `estate.property` model:

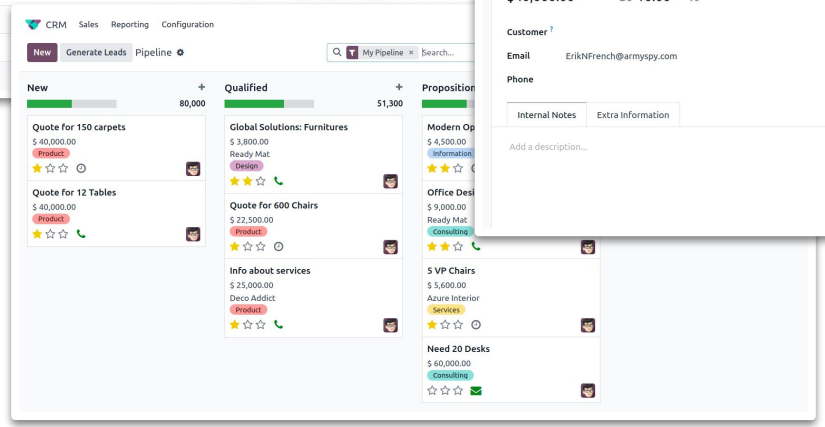
- `active` : Boolean field(default true)
- `state` : Selection field.
  → Possible values: New, Offer Received, Offer Accepted, Sold, Canceled.
  → must be required, should not be copied and default value set to 'New'.

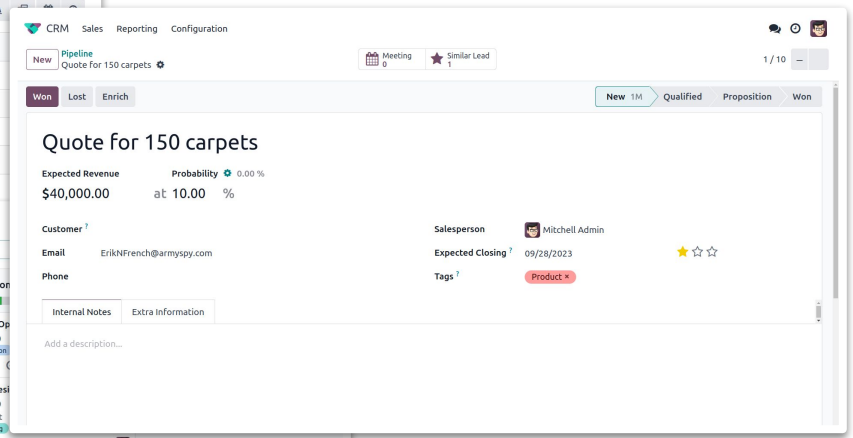
RESERVED FIELDS

Basic views

menu → action → views







CLICK HERE

Basic views

Views = instances of the **ir.ui.view** model.

```
<record id="MODEL_view_TYPE" model="ir.ui.view">
   <field name="name">model.view.type</field>
   <field name="model">model</field>
   <field name="arch" type="xml">
       <VIEW_TYPE>
            <VIEW_SPECIFICATIONS/>
       </VIEW_TYPE>
   </field>
</record>
```

Different types

- list (tree)
- form
- search
- kanban
- pivot
- map
- ...

CLICK HERE

List (or tree) views

CRM Sales Reporting Configuration

New | Generate Leads | My Activities | My Activities × Search... | 1-7 / 7

Opportunity	Contact Name	Email	Expected Revenue	Stage	
DeltaPC: 10 Computer Desks	Leland Martinez	info@deltapc.example.com	$ 35,000.00	Qualified	Email SMS Snooze 7d
Info about services		info@agrolait.com	$ 25,000.00	Qualified	Email SMS Snooze 7d
Access to Online Catalog		lumber-inv92@example.com	$ 2,000.00	Won	Email SMS Snooze 7d
Global Solutions: Furnitures	Robin Smith	info@deltapc.example.com	$ 3,800.00	Qualified	Email SMS Snooze 7d
Need 20 Desks		info@mycompany.net	$ 60,000.00	Proposition	Email SMS Snooze 7d
Office Design Project		info@agrolait.com	$ 24,000.00	New	Email SMS Snooze 7d
Quote for 12 Tables	Will McEncroe	willmac@rediffmail.example.com	$ 40,000.00	New	Email SMS Snooze 7d
			$ 189,800.00		

CLICK HERE

```
<tree string="My activities">
    <field name="opportunity"/>
    <field name="contact_name"/>
    <field name="email"/>
    ...
</tree>
```

Exercise

odoo PARTNERS

Chapter 6

~~Field view attrs~~

List view

Add a custom **list** view to the `estate.property` model.

Check next slide for visual **Goal**

1) do not add the `editable="bottom"` attribute that you can find in the example above. We'll come back to it later.

2) some field labels may need to be adapted to match the reference.

Example

Help: We start using -u **"module_name"** and **--dev xml** in command to restart module

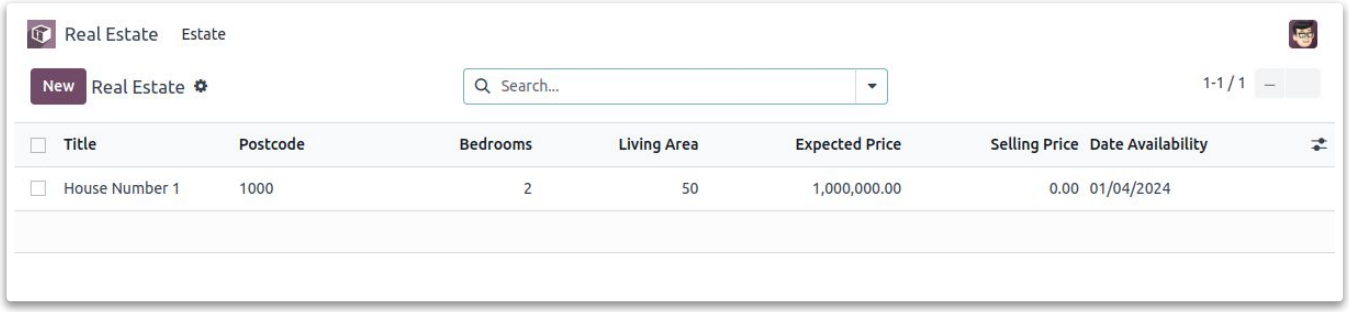
Exercise

Chapter 6

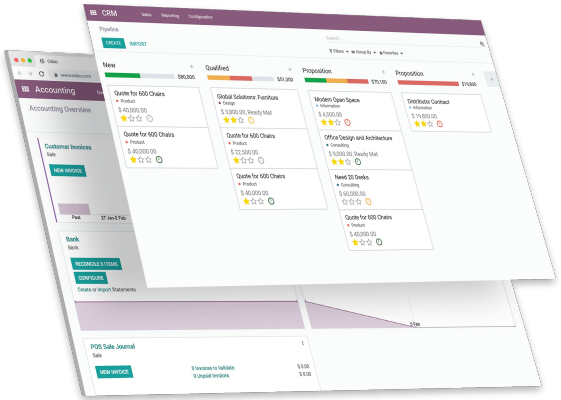
~~Field view attrs~~

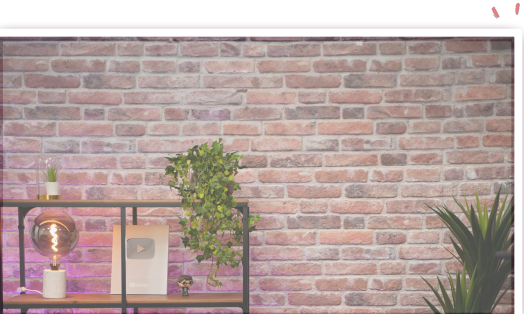
List view

Goal: at the end of this section, the list view should look like this:



Form views





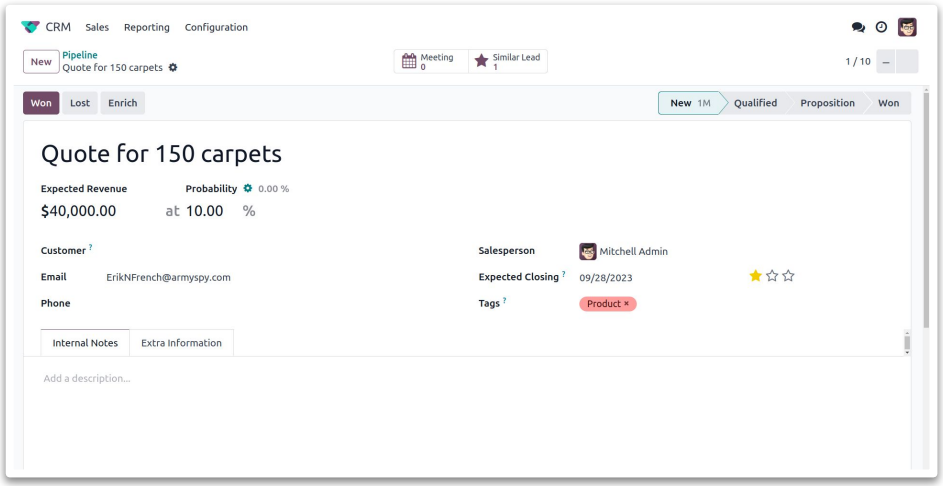
Display, create and edit **single records**

Composed of :

- **high-level structure** *- groups and notebooks*
- **Interactive elements** *- buttons and fields*

CLICK HERE

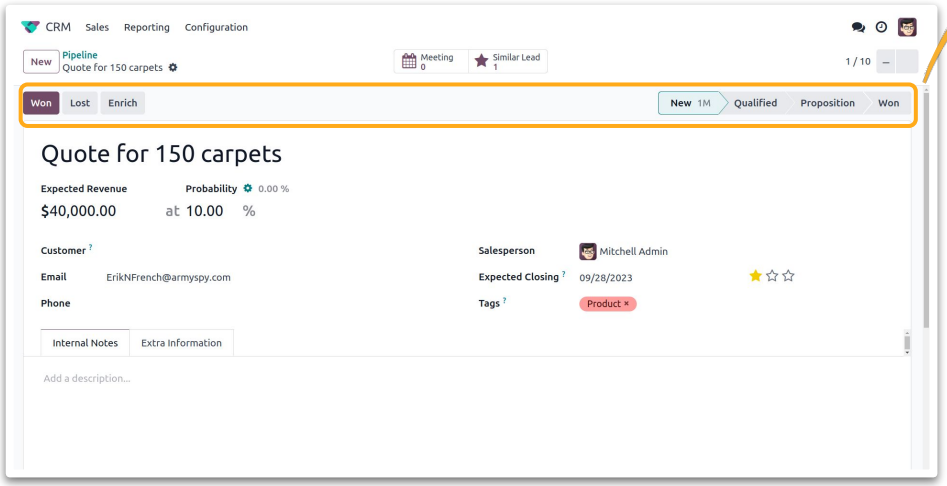
Form views



CLICK HERE

```
<form>
    <header>
        <field name="state"
widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_button_box">
            <BUTTONS/>
        </div>
        <group>
            <group>
                <field name="name"/>
            </group>
        </group>
        <notebook>
            <page string="Page1">
                <group>
                    <CONTENT/>
                </group>
            </page>
            <page string="Page2">
                <group>
                    <CONTENT/>
                </group>
            </page>
        </notebook>
    </sheet>
</form>
```

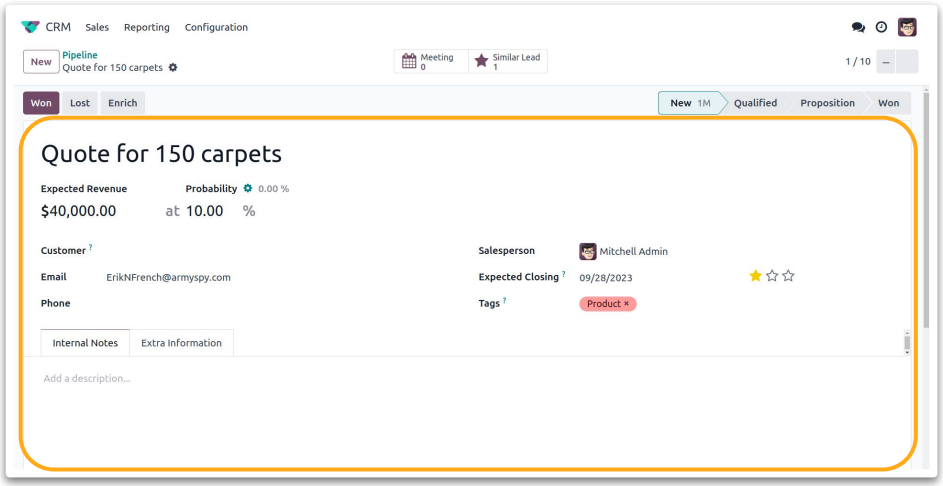
Form views




```xml
<form>
    <header>
        <field name="state"
widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_button_box">
            <BUTTONS/>
        </div>
        <group>
            <group>
                <field name="name"/>
            </group>
        </group>
        <notebook>
            <page string="Page1">
                <group>
                    <CONTENT/>
                </group>
            </page>
            <page string="Page2">
                <group>
                    <CONTENT/>
                </group>
            </page>
        </notebook>
    </sheet>
</form>
```

CLICK HERE

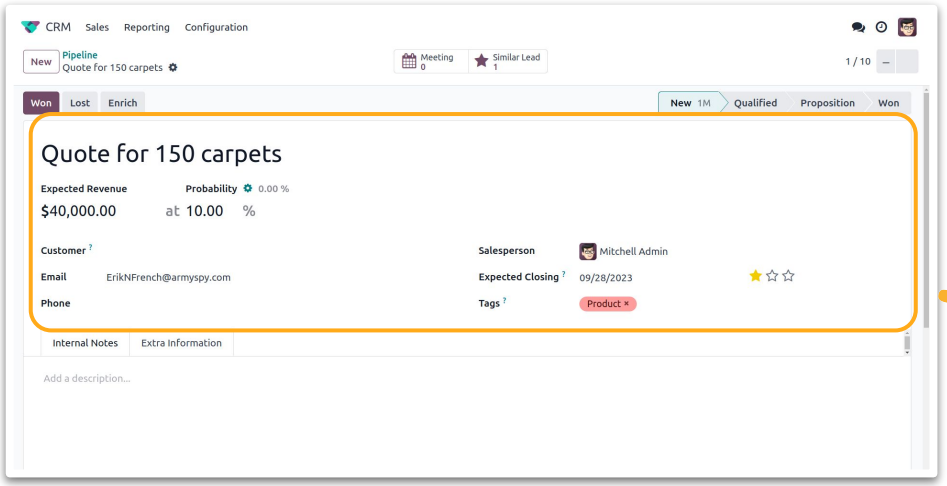
Form views




```
<form>
    <header>
        <field name="state"
widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_button_box">
            <BUTTONS/>
        </div>
        <group>
            <group>
                <field name="name"/>
            </group>
        </group>
        <notebook>
            <page string="Page1">
                <group>
                    <CONTENT/>
                </group>
            </page>
            <page string="Page2">
                <group>
                    <CONTENT/>
                </group>
            </page>
        </notebook>
    </sheet>
</form>
```

CLICK HERE

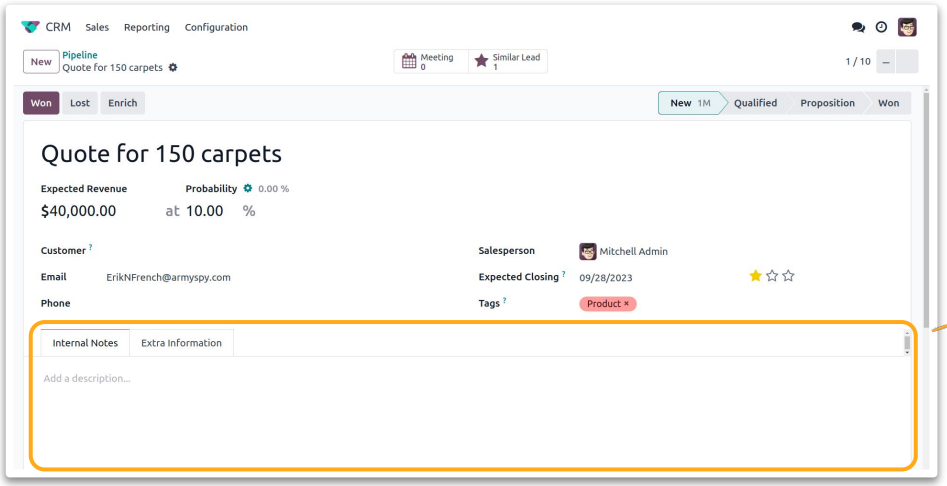
Form views




```
<form>
    <header>
        <field name="state"
widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_button_box">
            <BUTTONS/>
        </div>
        <group>
            <group>
                <field name="name"/>
            </group>
        </group>
        <notebook>
            <page string="Page1">
                <group>
                    <CONTENT/>
                </group>
            </page>
            <page string="Page2">
                <group>
                    <CONTENT/>
                </group>
            </page>
        </notebook>
    </sheet>
</form>
```

CLICK HERE

Form views




```
<form>
    <header>
        <field name="state"
widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_button_box">
            <BUTTONS/>
        </div>
        <group>
            <group>
                <field name="name"/>
            </group>
        </group>
        <notebook>
            <page string="Page1">
                <group>
                    <CONTENT/>
                </group>
            </page>
            <page string="Page2">
                <group>
                    <CONTENT/>
                </group>
            </page>
        </notebook>
    </sheet>
</form>
```

CLICK HERE

Exercise

odoo PARTNERS

Chapter 6

~~Field view attrs~~

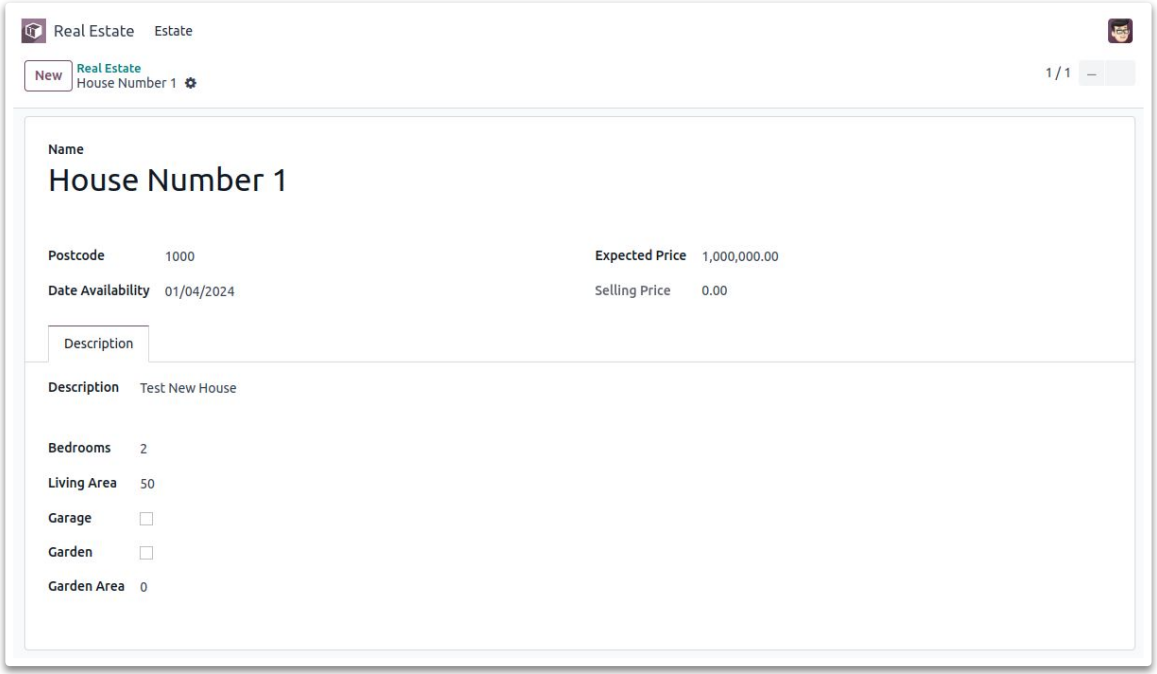
~~List view~~

Form view

Example

Add a custom **form view**.

1) Define a form view for the `estate.property` model in the appropriate XML file. Try to match the following example:

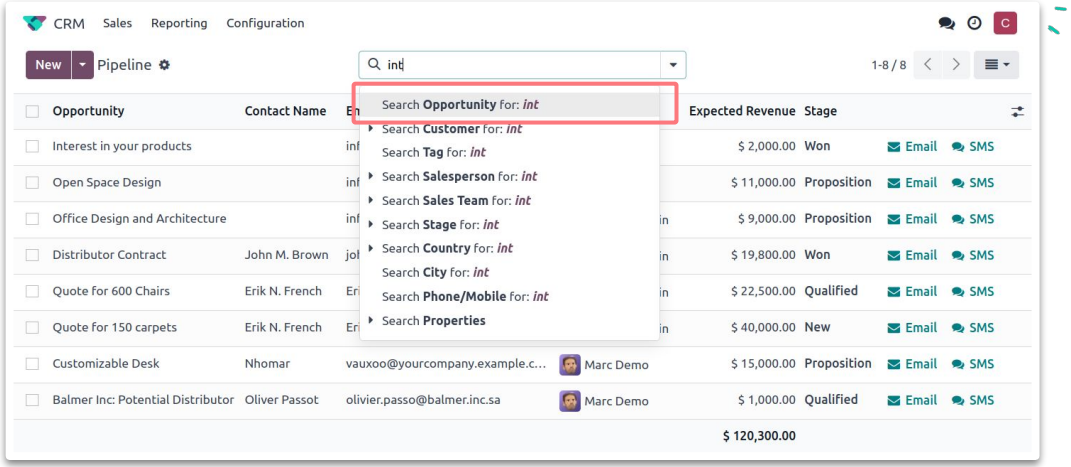


Search views

Doesn't *display content*

Filter *other view's content*

CLICK HERE




```
<search string="Tests">
    <field name="name"/>
    <field name="last_seen"/>
</search>
```

Search domain

Add **complexity** with **list of criteria** to select a subset of a model's records

Each criterion is a **triplet** with a (*'field_name'*, *'operator'*, *'value'*) operators: =, =, <, >, like, ilike, in, ...

```
[('product_type', '=', 'service'), ('unit_price', '>', 1000)]
```

select products 'which types are services ***AND*** with a unit price greater than 1000'

Logical operator & (AND), **|** (OR) and **!** (NOT) can be used **in** and **between** criteria.

```
['|', ('product_type', '=', 'service'), '!', '&', ('unit_price', '>=', 1000), ('unit_price', '<', 2000)]
```

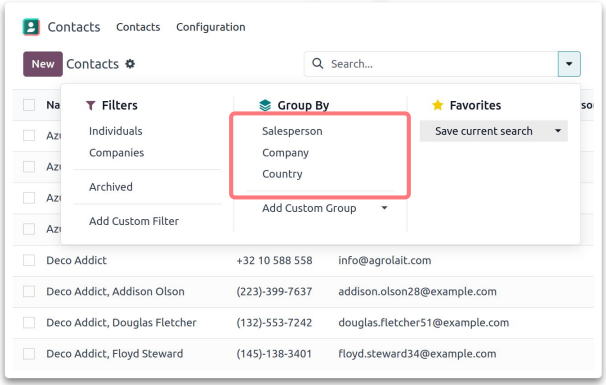
select products 'which are services ***OR*** have a unit price which is ***NOT*** between 1000 and 2000'

CLICK HERE

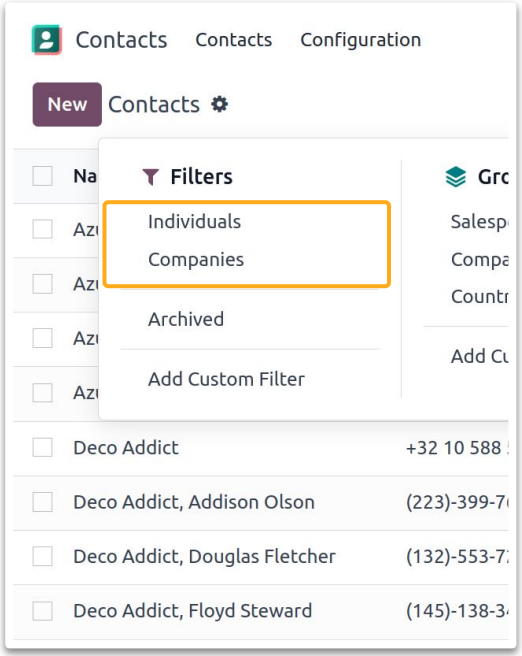
Advanced filter

Search views can contain **<filter>** elements and act as toggles for predefined searches.

Group by



Filters



Advanced filter

```
<search string="Tests">
   <group string="Group By">
        <filter name="group_by_field" context="{'group_by:'field'}"/>
   </group>
</search>
```

Group by

Using the **context** and the key **group_by**

Groups result into subsets based on field.

Filters

Using the **domain** attribute

Records filtered based on domain.

```
<search string="Tests">
    <filter string="New State" name="filter_new_state"
                                domain="[('state','=','new')]"/>
</search>
```

Exercise

odoo PARTNERS

Chapter 6

~~Field view attrs~~

~~List view~~

~~Form view~~

Search view

Add a custom **search view** to the `estate.property` model.

1) The following fields should be searchable:

   title, postcode, expected price, bedrooms, living area, façades

Example

Exercise

Chapter 6

~~Field view attrs~~

~~List view~~

~~Form view~~

Search view (2)

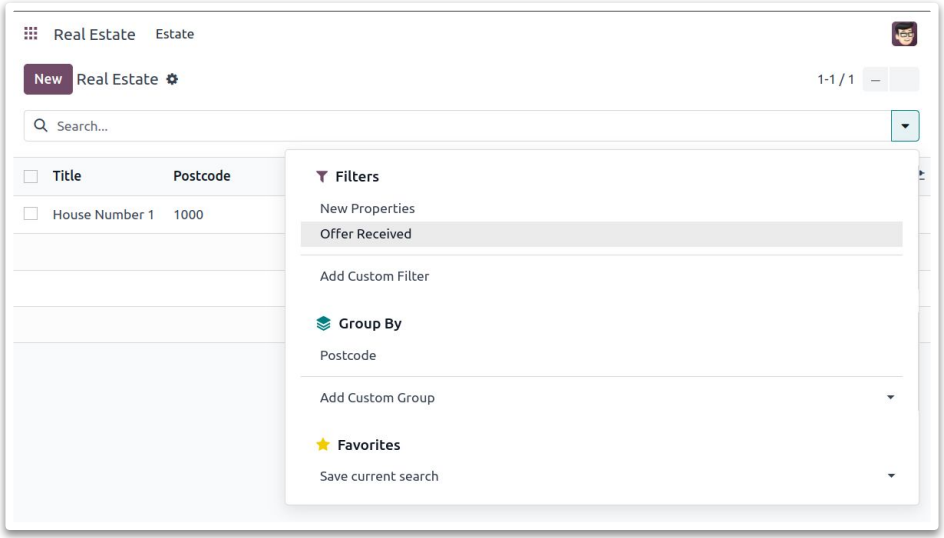
Example

In the **search view** of the `estate.property` model,

Add the following **Filters** and **Group By.**

1) Filters to display available properties,
   i.e. the state should be 'New' or 'Offer Received'.

2) The ability to group results by postcode.

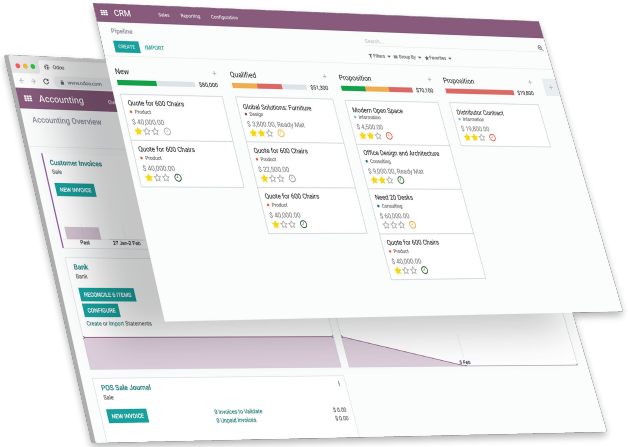


odoo
PARTNERS

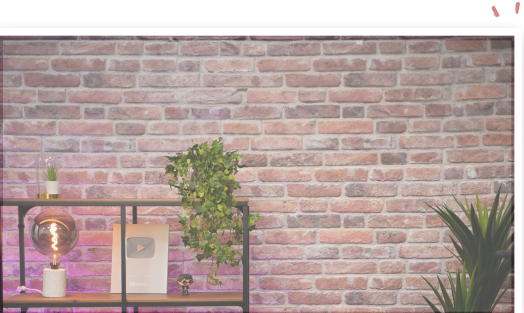
Chapter 7

Relational Fields

Relations between Models



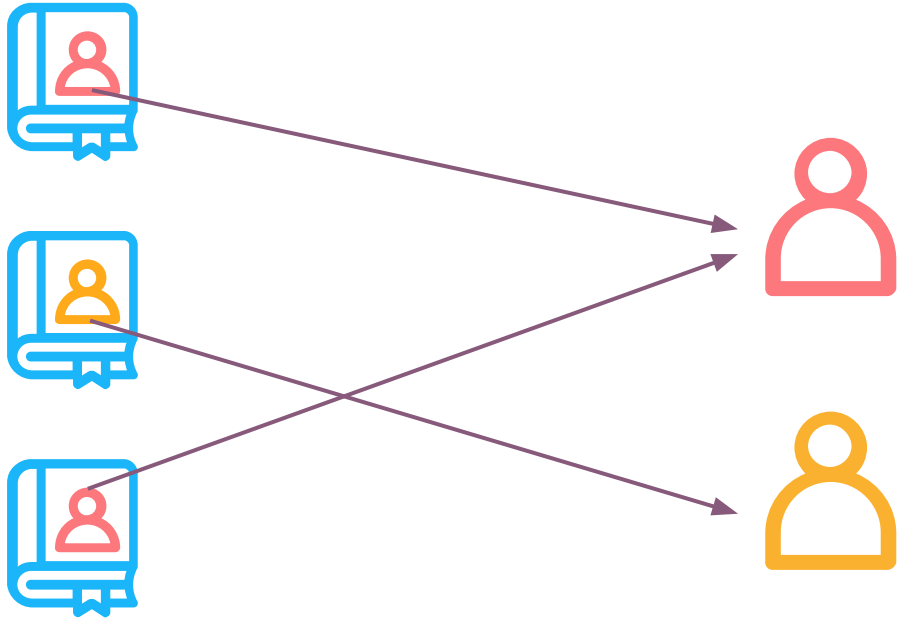
- Many to one (Many2one)
- One to many (One2many)
- Many to many (Many2many)



CLICK HERE

Many to one

- A book can have **one** author
- An author can write **multiple** books



CLICK HERE

```
field_id = fields.Many2one(comodel_name="comodel.name", string="Field")
```

```
field_id = fields.Many2one("comodel.name", string="Field")
```

Exercise

odoo PARTNERS

Chapter 7

New model

Add the Real Estate Property Type table.

- Create the `estate.property.type` model and add the following field:

Field	Type	Attributes
name	Char	required

- Add the setting menus and a submenu redirecting to this new model (as displayed in this section's **Goal** in the next slide)
- Add the field `property_type_id` into your `estate.property` model and its form, tree and search views

This exercise is a good recap of the previous chapters: you need to create a model, set the model, add an action and a menu, and create a view.

Tip: do not forget to import any new Python files in `__init__.py`, add new data files in `__manifest.py__` or add the access rights

Exercise

odoo PARTNERS

Chapter 7

~~New model~~

many2one

Add a buyer (`res.partner`) and a salesperson (`res.users`) to the `estate.property` model.

They should be added in a new tab of the form view, as depicted in this section's **Goal** in the next slide.

The default value for the salesperson must be the current user. The buyer should not be copied.

CLICK HERE

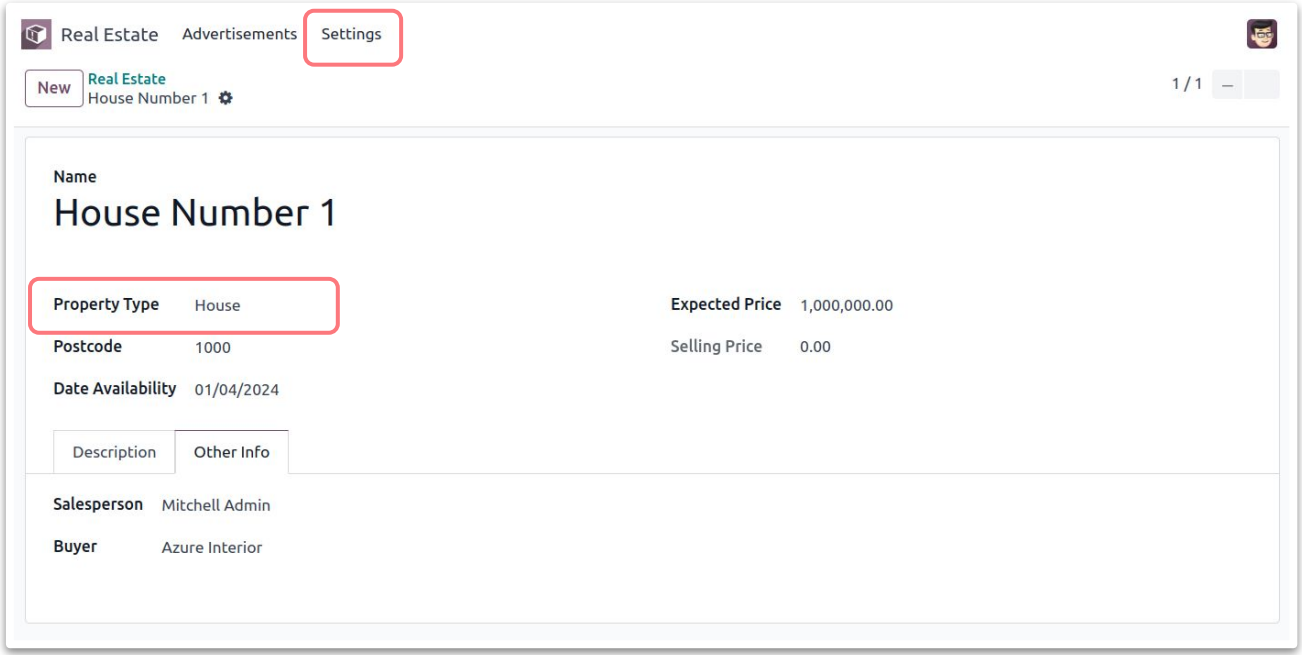
Tip: to get the default value, look at an example here.

Exercise

Chapter 7

~~New model~~

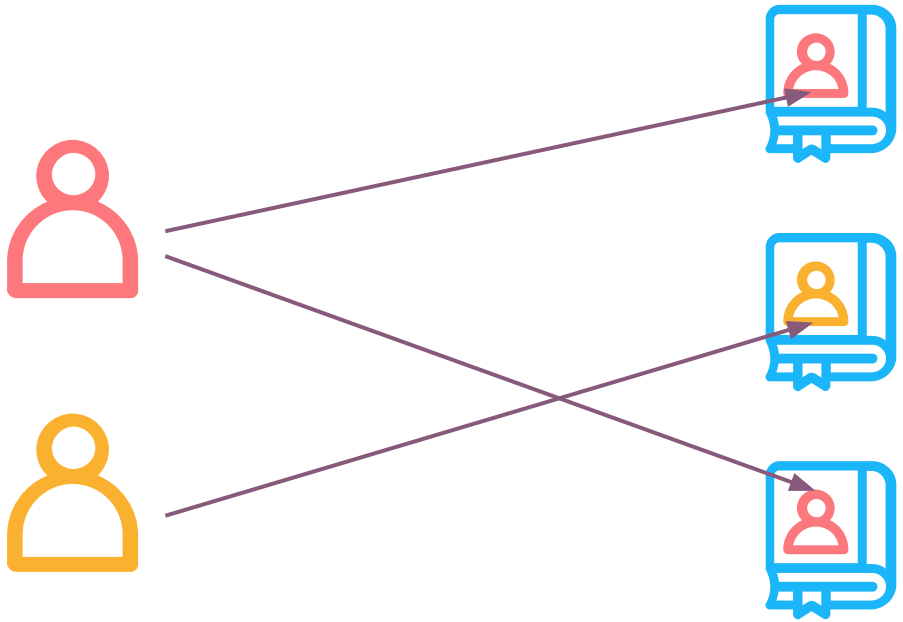
many2one



One to many

one2many is the **inverse** of a many2one

- An author can write **multiple** books
- A book can have **one** author



CLICK HERE

```
field_ids = fields.One2many("comodel.name", "inverse_name_id", string="Field")
```

One2many is a virtual relationship, there must be a Many2one field defined in the comodel

Exercise

odoo PARTNERS

Chapter 7

~~New model~~

~~many2one~~

one2many

Add the **Real Estate Property Offer** table.

- Create the `estate.property.offer` model and add the following fields:
- Create a **tree view** and a **form view** with the `price`, `partner_id` and `status` fields.
  → No need to create an action or a menu.
- Add the field `offer_ids` to your `estate.property` model and in its form view as depicted in this section's **Goal**.

Field	Type	Attributes	Values
price	Float		
status	Selection	no copy	Accepted, Refused
partner_id	Many2one (`res.partner`)	required	
property_id	Many2one (`estate.property`)	required	

Exercise

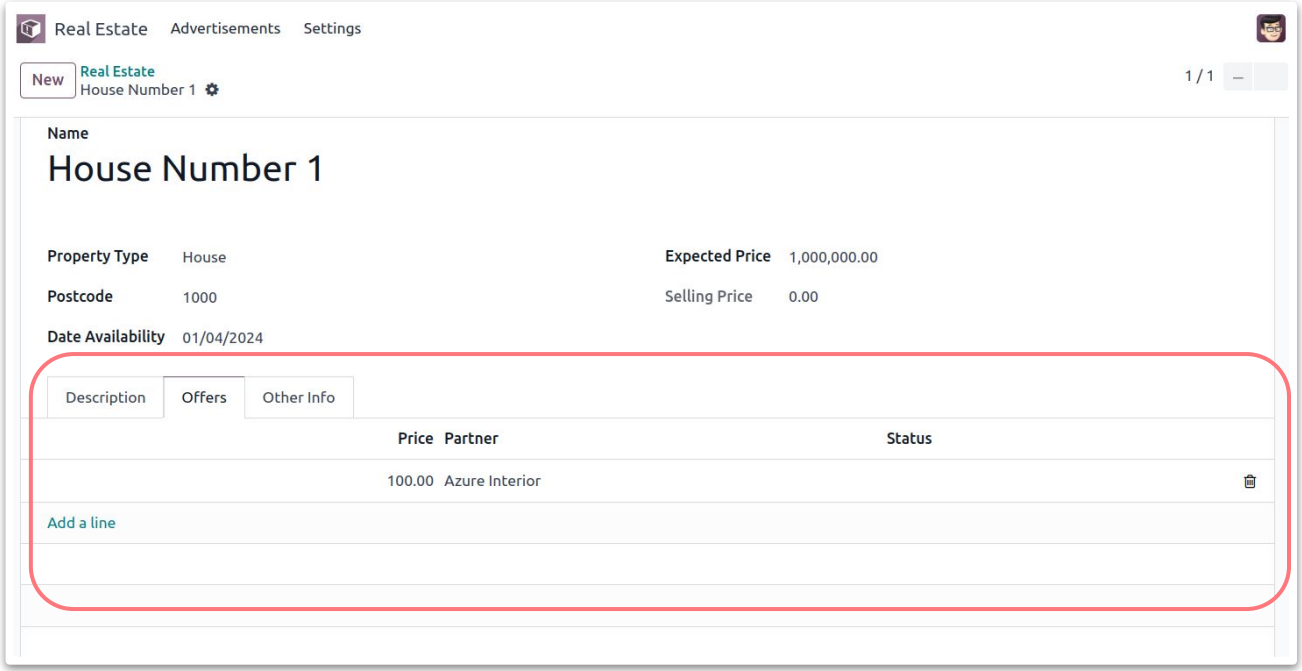
odoo PARTNERS

Chapter 7

~~New model~~

~~many2one~~

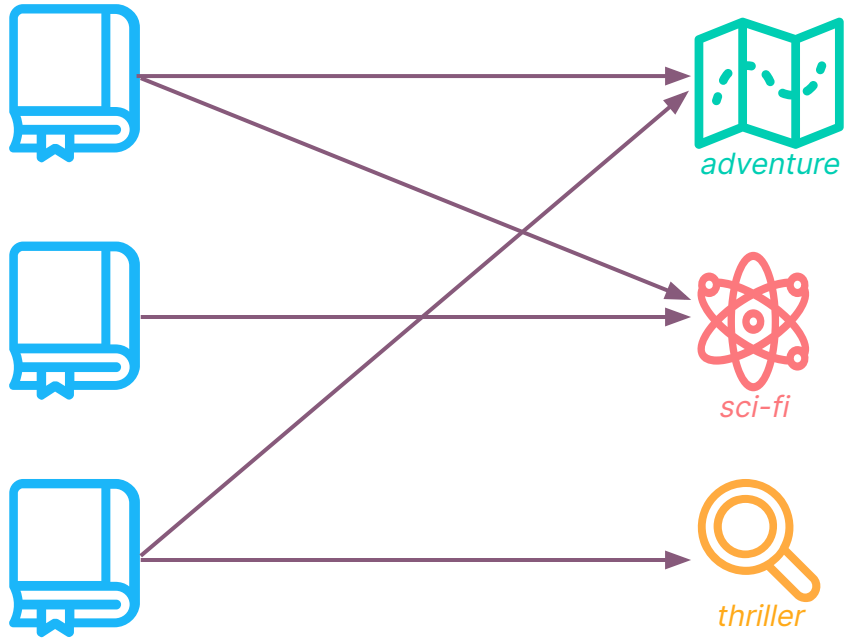
one2many



Many to many

- Books can have **multiple** categories
- Categories can include **multiple** books

CLICK HERE



```
field_ids = fields.Many2many("comodel.name", string="Field")
```

Exercise

odoo PARTNERS

Chapter 7

~~New model~~

~~many2one~~

~~one2many~~

many2many

Add the **Real Estate Property Tag** table.

- Create the `estate.property.tag` model and add the following field:
- Add a menu to get to a tree view of this model in Settings
- Add the field `tag_ids` to your `estate.property` model and in its form and tree views

Tip: in the view, use the `widget="many2many_tags"` attribute as demonstrated here.

Field	Type	Attributes
name	Char	required

The `widget` attribute will be explained in detail in a later chapter of the training.

Exercise

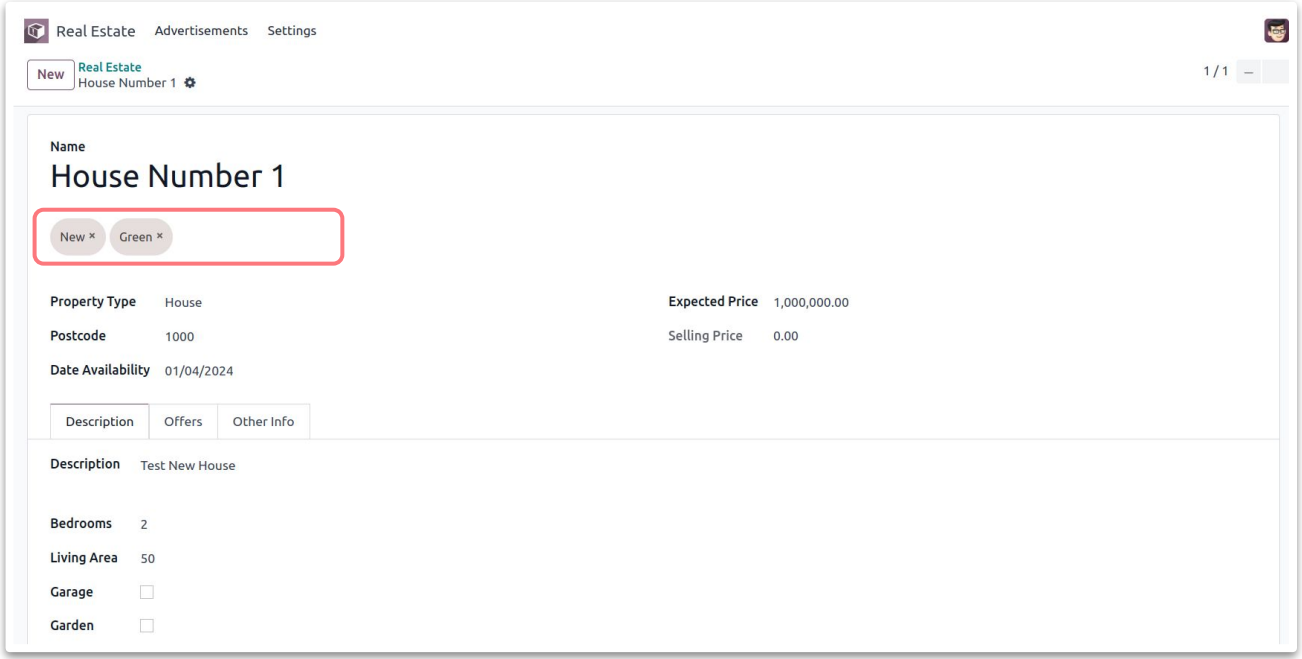
Chapter 7

~~New model~~

~~many2one~~

~~one2many~~

many2many



odoo PARTNERS

Chapter 8

Fields computation

Compute definition

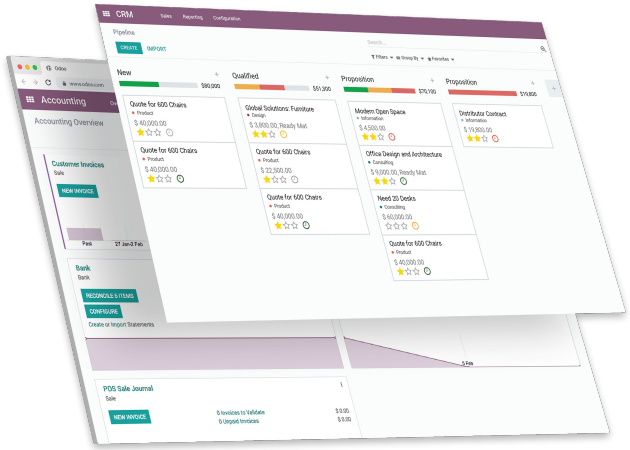
1) Create a python method

2) In the field **attributes**, set **compute** with the name of this method

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    amount = fields.Float()
    total = fields.Float(compute="_compute_total")

    def _compute_total(self):
        for rec in self:
            rec.total = 10 * rec.amount
```

CLICK HERE

Compute information



- **compute methods** are **private**
  - name starts with an underscore

```
def _my_method(self)
```

- **computed fields** are **readonly** by default
- **computed fields** are **not stored** by default.
  - So not possible to search
    - Easy fix → **store value**
    - Other solution exist (search method) but not in this training

CLICK HERE

Recordset (self)

Compute can be triggered for **multiple records** at the same time

↳ **self** will include one **or more than one** object.

danger for the result

```
amount = fields.Float()
total = fields.Float(compute="_compute_total")

def _compute_total(self):
    self.total = 10 * self.amount
```

Recordset (self)

CLICK HERE

Compute can be triggered for **multiple records** at the same time

↳ **self** will include one **or more than one** object.

Solution: iterating over **self** to give values one by one

```
amount = fields.Float()
total = fields.Float(compute="_compute_total")

def _compute_total(self):
    for rec in self:
        rec.total = 10 * rec.amount
```

Dependencies

- Desired result can **depend** on other field(s)
- **Recompute** if those **change**
- Define dependencies using **api decorator "depends"** on method

```python
from odoo import api, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    amount = fields.Float()
    total = fields.Float(compute="_compute_total")

    @api.depends("amount")
    def _compute_total(self):
        for rec in self:
            rec.total = 10 * rec.amount
```

CLICK HERE

Exercise

odoo PARTNERS

Chapter 8

Computation

Compute the **total area** of a property.

- Add the `total_area` field to `estate.property`**.** It is defined as the sum of the `living_area` and the `garden_area`.
- Add the field in the form view.

Related compute

- Result desired can also **depend** on field(s) from **other models**
- Can have **complex relation** in decorator and in compute

```python
from odoo import api, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    partner_total = fields.Float(compute="_compute_partner_total")
    partner_id = fields.Many2one("res.partner")

    @api.depends("partner_id.amount")
    def _compute_total(self):
        for rec in self:
            rec.partner_total = 10 * rec.partner_id.amount
```

Multiple records

- Danger with **multiple relation**
  - **list** of records need to get their data in a **list**

```python
from odoo import api, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    partner_total = fields.Float(compute="_compute_partner_total")
    partner_ids = fields.One2many("res.partner")

    @api.depends("partner_ids.amount")
    def _compute_total(self):
        for rec in self:
            rec.partner_total = sum(rec.partner_ids.amount)
```

CLICK HERE

Multiple records

- Danger with **multiple relation**

  ↳ **list** of records need to get their data in a **list**

- Using ORM **"mapped"** operator

```python
from odoo import api, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    partner_total = fields.Float(compute="_compute_partner_total")
    partner_ids = fields.One2many("res.partner")

    @api.depends("partner_ids.amount")
    def _compute_total(self):
        for rec in self:
            rec.partner_total = sum(rec.partner_ids.mapped("amount"))
```

CLICK HERE

Exercise

odoo PARTNERS

Chapter 8

Computation

Compute the **best offer** of a property.

- Add the `best_price` field to `estate.property`. It is defined as the highest (i.e. maximum) of the offers' `price`.
- Add the field in the form view.

Related fields

```python
partner_id = fields.Many2one("res.partner", string="Partner")
description = fields.Char(compute="_compute_description")

@api.depends("partner_id.name")
def _compute_description(self):
    for record in self:
        record.description = record.partner_id.name
```

is equivalent to

```python
partner_id = fields.Many2one("res.partner", string="Partner")
description = fields.Char(related="partner_id.name")
```

related should have same type as field itself !

CLICK HERE

Exercise

odoo PARTNERS

Chapter 8

~~Computation~~

Related field

Add the **property type** in the offers tree view.

- Add the `property_type_id` field to `estate.property.offer`. We can define it as a related field on `property_id.property_type_id` and set it as stored.
- Add a menu under **Advertisement** linking to the tree view. Display the fields: `property_id,partner_id,price,status,property_type_id`

Inverse function

- We want the **two fields** to have an **inverse relation**

Using compute in both will create **infinite loop**

- Use **inverse attribute** in same field as compute

```python
from odoo import api, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    total = fields.Float(
                compute="_compute_total",
                inverse="_inverse_total"
    )
    amount = fields.Float()

    @api.depends("amount")
    def _compute_total(self):
        for rec in self:
            rec.total = 10 * rec.amount

    def _inverse_total(self):
        for rec in self:
            rec.amount = rec.total / 10
```

Exercise

odoo PARTNERS

Chapter 8

~~Computation~~

~~Related field~~

Inverse

Compute a **validity date** for offers.

- Add the following fields to the `estate.property.offer` model:

Field	Type	Default
validity	Integer	7
date_deadline	Date	

Where `date_deadline` is a computed field which is defined as the sum of two fields from the offer: the `create_date` and the `validity`. Define an appropriate inverse function so that the user can set either the date or the validity.

- Add the fields in the form view and the list view

Note : compute updated after save master record

Tip: the `create_date` is only filled in when the record is created, therefore you will need a fallback to prevent crashing at time of creation.

odoo
PARTNERS

Chapter 9

Fields Onchange

onchange

CLICK HERE

- Field **recomputed instantly** if another field is modified **in the view**
- Defined using **api decorator "onchange"** on method

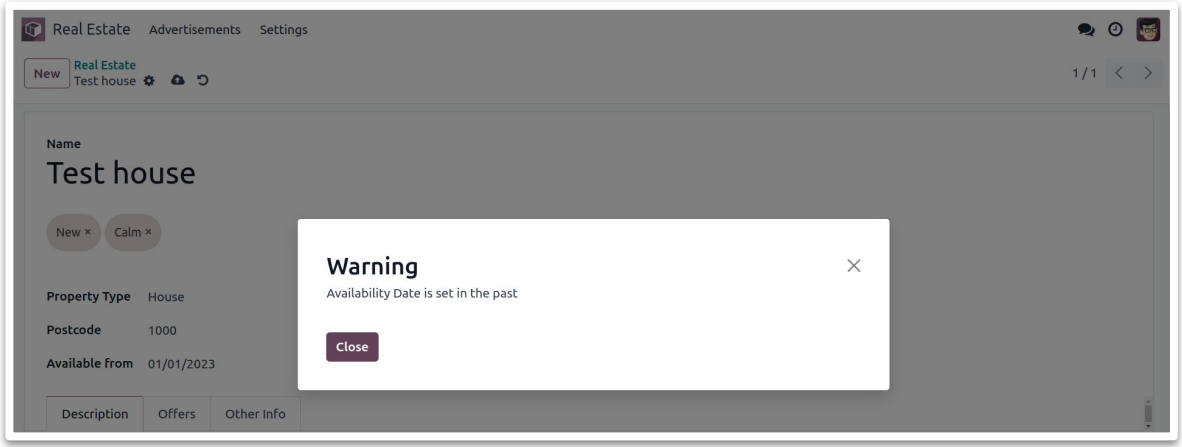
```python
from odoo import api, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    name = fields.Float()
    partner_id = fields.Many2one("res.partner")


    @api.onchange("partner_id")
    def _onchange_partner_id(self):
        self.name = "House - %s" % (self.partner_id.name)
```

Never use onchange for business logic !

onchanges are only triggered **updating** fields in the **form view**

Non blocking warning

Return **non-blocking** pop-up warning in onchange on condition



```python
amount = fields.Float()

@api.onchange("amount")
def _onchange_amount(self):
    if self.amount < 0:
        return {"warning": {"title": _("Warning"), "message": _("Amount is negative")}}
```

Exercise

odoo PARTNERS

Chapter 9

On change

Set values for **garden area** and **orientation**.

Create an `onchange` in the `estate.property` model in order to set values for the garden area and orientation when **garden** is set to **True**.

When unset, clear the fields.

Exercise

odoo PARTNERS

Chapter 9

On change

Raise a **soft warning.**

Create a `soft warning` in the `estate.property` model if the field `date_availability` is set to a date **prior than today's**.

odoo
PARTNERS

Chapter 10

Actions

What is an action ?

System **response** to user **actions**

login, action button, record selection, ...

Success

Open window, modify values, ...

On condition

Raise Error

CLICK HERE

Action

Can be defined in XML records → cfr chapter 5

```xml
<record id="test_model_action"
                model="ir.actions.act_window">
    <field name="name">Test action</field>
    <field name="res_model">test.model</field>
    <field name="view_mode">tree,form</field>
</record>
```

CLICK HERE

In view definition

CLICK HERE

Adding **button** in your different **views**

2 mandatory attributes:

- type="**object**"
- name="**action_do_something**"

```
<button string="Do Something" type="object" name="action_do_something" />
```

OR use type="**action**" and we refer the external identifier in **name** attribute

```
<button string="My Action" type="action" name="%(test.action_do_something)d"/>
```

Business logic

Define a public method *(does not start with "_")*

Make them do what you want !

→ update fields, create/delete records...

```python
from odoo import fields, models

class TestAction(models.Model):
    _name = "test.action"

    name = fields.Char()

    def action_do_something(self):
        for record in self:
            record.name = "Something"
        return True
```

Business logic

Can also be used to open views

(same as actions in chapter 5)

Return a dict with same values

as defined in XML data records

```python
from odoo import fields, models

class TestAction(models.Model):
    _name = "test.action"

    def action_open_something(self):
        self.ensure_one()
        return {
            "type": "ir.actions.act_window",
            "name": "Test action",
            "res_model": "test.model",
            "view_mode": "form",

        }
```

```xml
<button type="object" name="action_open_something" string="Do Something" />
```

Business logic

Can pass more complex arguments

- **res_id**
  (form view) specific record to load
- **context**
  additional data loaded in the context
- **domain**
  filtering domain added in view search queries

→ Many other (see doc)

(args also possible to use in XML data records)

CLICK HERE

```python
from odoo import fields, models

class TestAction(models.Model):
    _name = "test.action"

    def action_open_something(self):
        self.ensure_one()
        return {
            "type": "ir.actions.act_window",
            "name": "Test action",
            "res_model": "test.model",
            "view_mode": "form",
            "res_id": self.child_id.id,
        }
```

```xml
<button type="object" name="action_open_something" string="Do Something" />
```

Raising error

- Under condition, **block** the action
- Different types of **errors**
- **UserError** used if user tries to do something that has no sense given the current state of a record

```python
from odoo import fields, models
from odoo.exceptions import UserError


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    amount = fields.Float()
    total = fields.Float()


    def action_do_something(self):
        for rec in self:
            if rec.total < 0:
                raise UserError(_("Total should be positive"))
            rec.amount = rec.total / 10
        return True
```

Text returned in error can be made translatable using " _(...) "

CLICK HERE

Exercise

odoo PARTNERS

Chapter 10

action

Set a property as **sold** or **cancelled**.

- Add the buttons **'Cancel'** and **'Sold'** to the `estate.property` model. A canceled property cannot be set as sold, and a sold property cannot be canceled.
  Tip: in order to raise an error, you can use the UserError function.

- Add the buttons **'Accept (v)'** and **'Refuse (x)'** to the `estate.property.offer` model list view.
  Tip: to use an icon as a button, have a look at this example.

- When an **offer** is **accepted**, set the **buyer** and the **selling price** for the corresponding **property**.

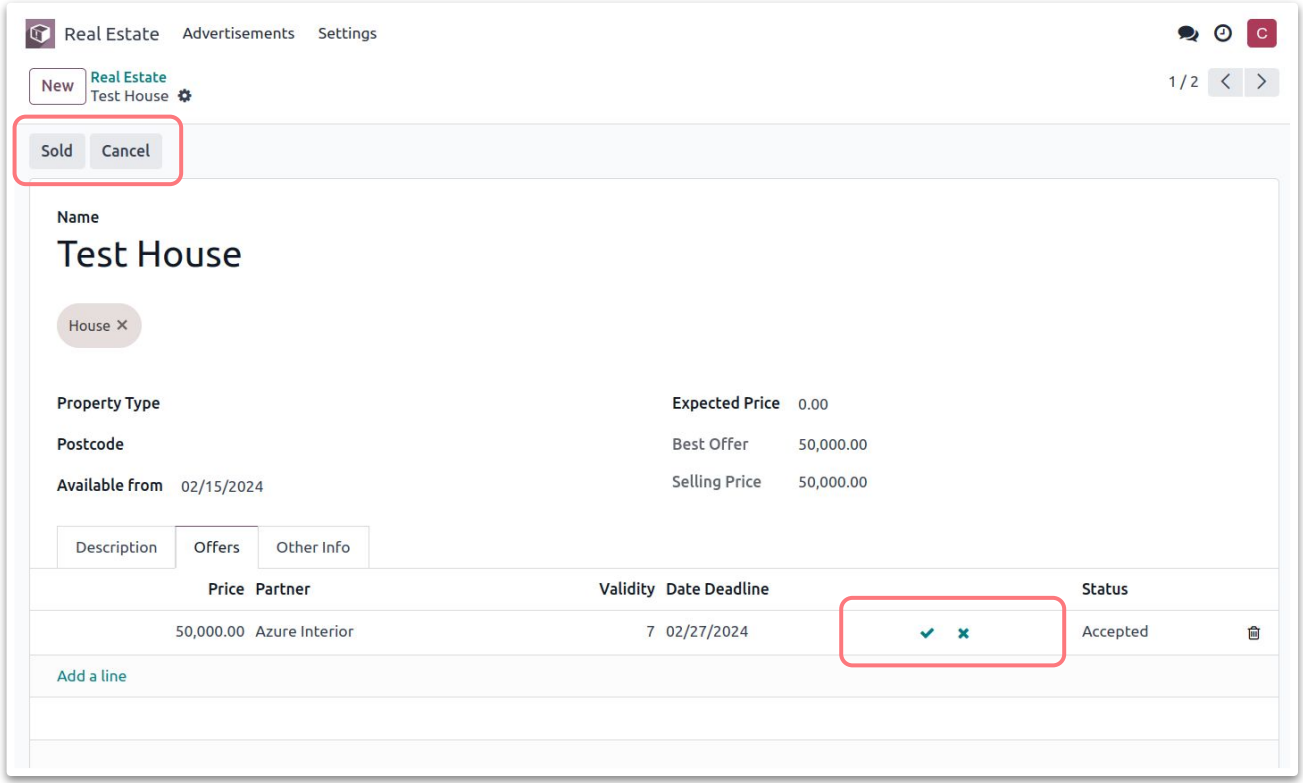
  Pay attention: in real life only one offer can be accepted for a given property!

Exercise

odoo PARTNERS

Chapter 10

action



odoo
PARTNERS

Chapter 11

Constraints

Constraint Types

SQL

VS

Python

SQL constraint

Defined in the model attribute **_sql_constraints** in the python file

```
_sql_constraints = [(name, sql_condition, error_message)]
```

SQL conditions can use different types of constraints:

1) **Check** Constraints
2) **Not-Null** Constraints
3) **Unique** Constraints
4) **Primary Keys**
5) **Foreign Keys**
6) **Exclusion** Constraints

CLICK HERE

SQL constraint

Example :

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    price = fields.Float()
    name = fields.Char()

     _sql_constraints = [
        ("check_price", "CHECK(price > 0)", "Price should be positive."),
        ("check_unique_name", "UNIQUE(name)", "Name must be unique.")
    ]
```

CLICK HERE

Exercise

odoo PARTNERS

Chapter 11

SQL constraints

Add **SQL constraints**.

Add the following constraints to their corresponding **models**:

- A **property** expected price must be strictly positive
- A **property** selling price must be positive
- An **offer** price must be strictly positive
- A **property tag** name and property type name must be unique

Tip: search for the `unique` keyword in the Odoo codebase for examples of unique names.

Python constraint

- For **complex** checks require **Python** code
- Defined using **api decorator "constrains"** on method

```python
from odoo import api, fields, models, ValidationError


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    date = fields.Date()

    @api.constrains("date")
    def _check_date_end(self):
        for record in self:
            if record.date < fields.Date.today():
                raise ValidationError(
                                _("The end date cannot be set in the past")
                                )
```

- Should raise ValidationError if the validation failed.
- Remember about the underscore "_(...)" for the translations

CLICK HERE

CLICK HERE

Technicalities

- **SQL** constraints are usually **more efficient** than **Python** ones
- ***@constrains*** only supports simple field name *(ex: partner_id.customer)*
- ***@constrains*** will be triggered only if fields are in the *create()* or *write()* call.
- Working with **float** : Use *float_compare()* or *float_is_zero()*

(from odoo.tools.float_utils)

Exercise

odoo PARTNERS

Chapter 11

~~SQL constraints~~

Python constraints

Add **Python constraints**.

Add a constraint so that the **selling price** **cannot be lower than 90%** of the **expected price**.

Ensure the constraint is triggered every time the **selling price** or the **expected price** is changed!

Tip: the selling price is zero until an offer is validated. You will need to fine tune your check to take this into account.

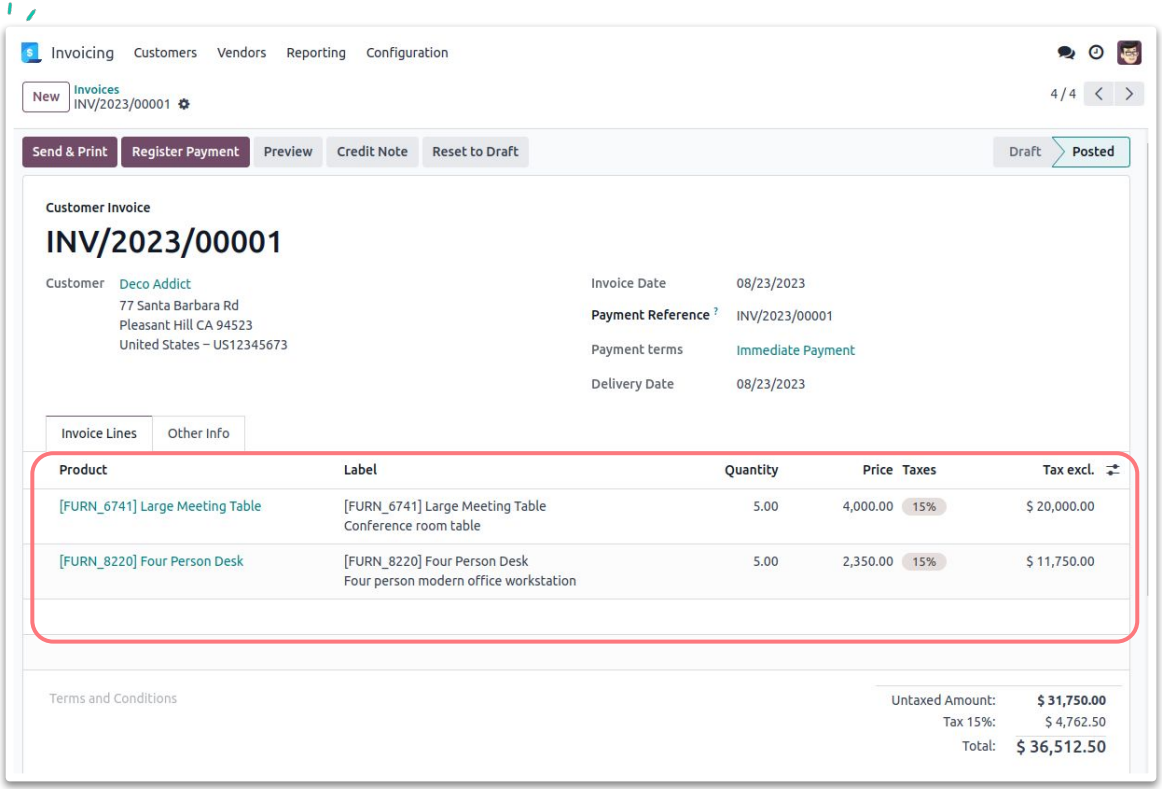
odoo
PARTNERS

Chapter 12

Inline views

Inline view

Possible to have **tree view** of multiple records inside **form view**



Inline view definition

```python
from odoo import fields, models

class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    description = fields.Char()
    line_ids = fields.One2many("test.model.line", "model_id")


class TestModelLine(models.Model):
    _name = "test.model.line"
    _description = "Test Model Line"

    model_id = fields.Many2one("test.model")
    field_1 = fields.Char()
    field_2 = fields.Char()
    field_3 = fields.Char()
```

```xml
<form>
    <field name="description"/>
    <field name="line_ids">
        <tree>
            <field name="field_1"/>
            <field name="field_3"/>
        </tree>
    </field>
</form>
```

Inline view definition

```python
from odoo import fields, models

class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    description = fields.Char()
    line_ids = fields.One2many("test.model.line", "model_id")


class TestModelLine(models.Model):
    _name = "test.model.line"
    _description = "Test Model Line"

    model_id = fields.Many2one("test.model")
    field_1 = fields.Char()
    field_2 = fields.Char()
    field_3 = fields.Char()
```

```xml
<form>
    <field name="description"/>
    <field name="line_ids">
        <tree>
            <field name="field_1"/>
            <field name="field_3"/>
        </tree>
    </field>
</form>
```

Inline view definition

```python
from odoo import fields, models

class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    description = fields.Char()
    line_ids = fields.One2many("test.model.line", "model_id")


class TestModelLine(models.Model):
    _name = "test.model.line"
    _description = "Test Model Line"

    model_id = fields.Many2one("test.model")
    field_1 = fields.Char()
    field_2 = fields.Char()
    field_3 = fields.Char()
```

```xml
<form>
    <field name="description"/>
    <field name="line_ids">
        <tree>
            <field name="field_1"/>
            <field name="field_3"/>
        </tree>
    </field>
</form>
```

CLICK HERE

Exercise

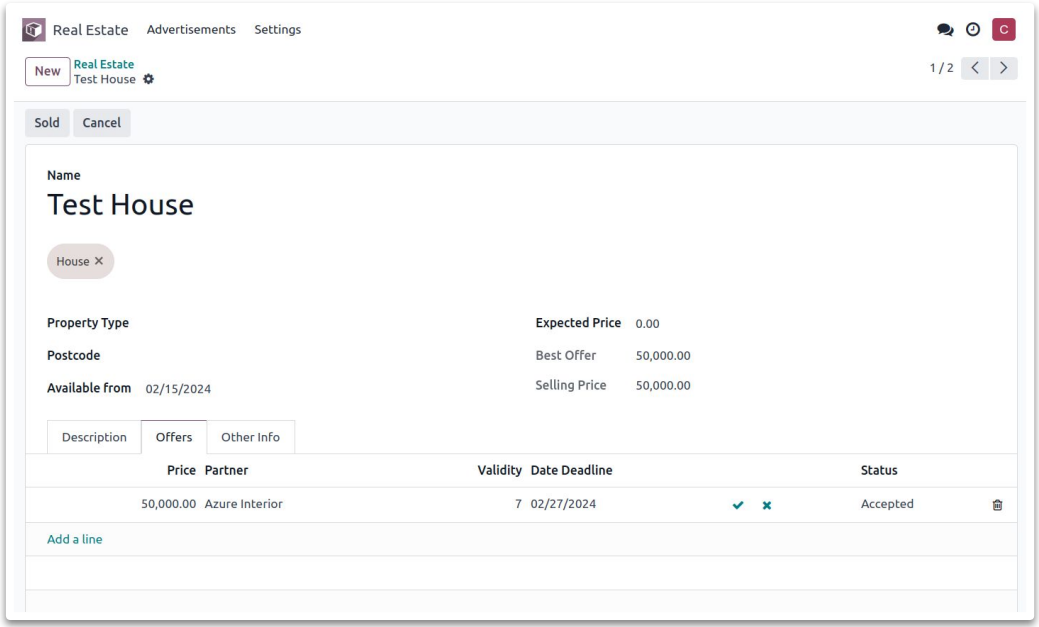
odoo PARTNERS

Chapter 12

Inline tree

CLICK HERE

- In the `estate.property` model form view, adapt the **offers** inline tree view to match the following example:



Exercise

odoo PARTNERS

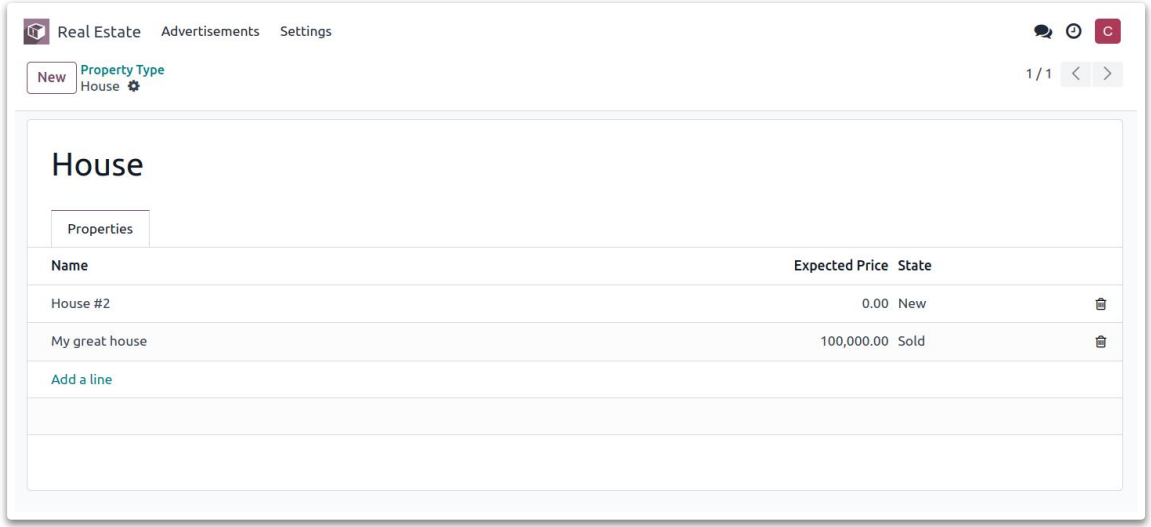
Chapter 12

Inline tree

CLICK HERE

Add an inline list view.

- Add the `One2many` field `property_ids` to the `estate.property.type` model.
- Add the field in the `estate.property.type` form view.



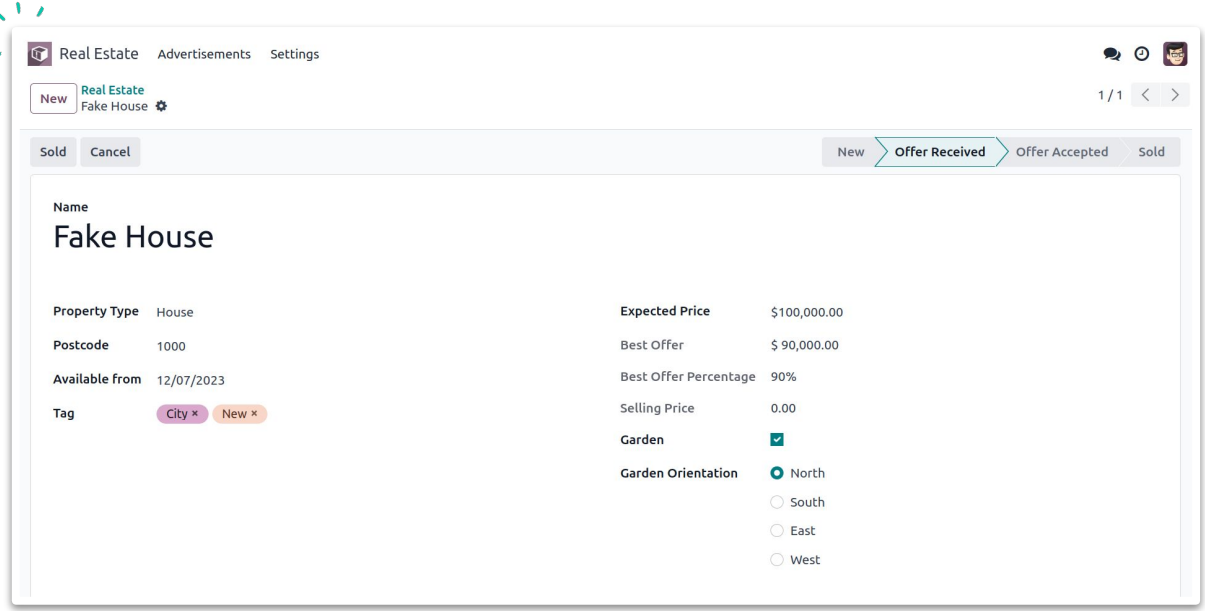
odoo
PARTNERS

Chapter 13

Widgets

Widgets

- statusbar
- monetary
- percentage
- radio
- many2many_tags
- ...



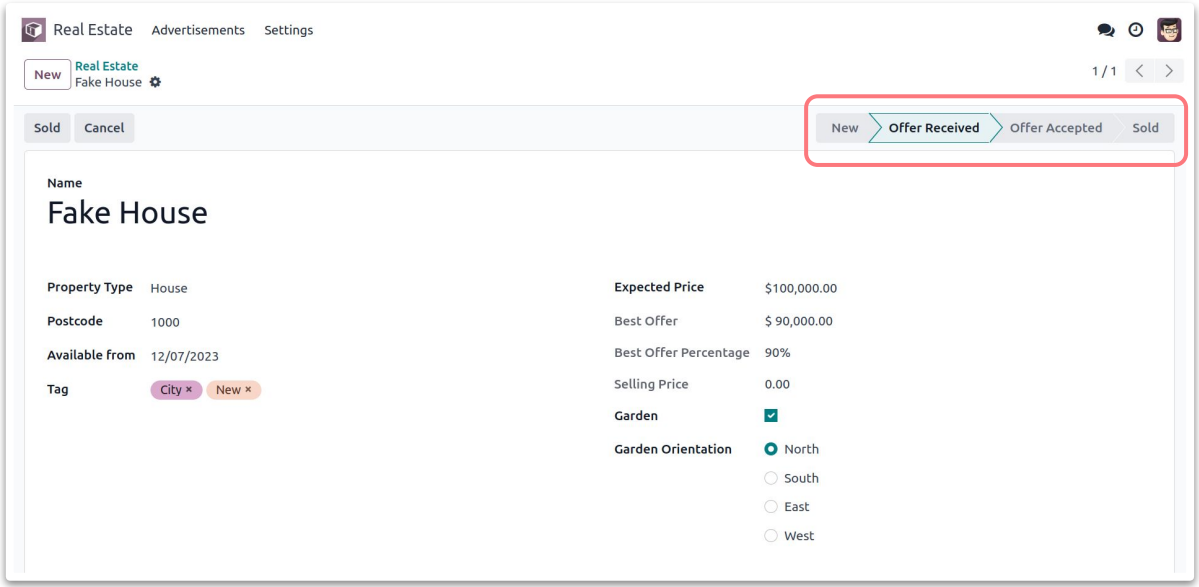
Many different widgets exist !

check in the standard code or in doc to discover them all

CLICK HERE

Widgets

- **statusbar**
- monetary
- percentage
- radio
- many2many_tags
- ...

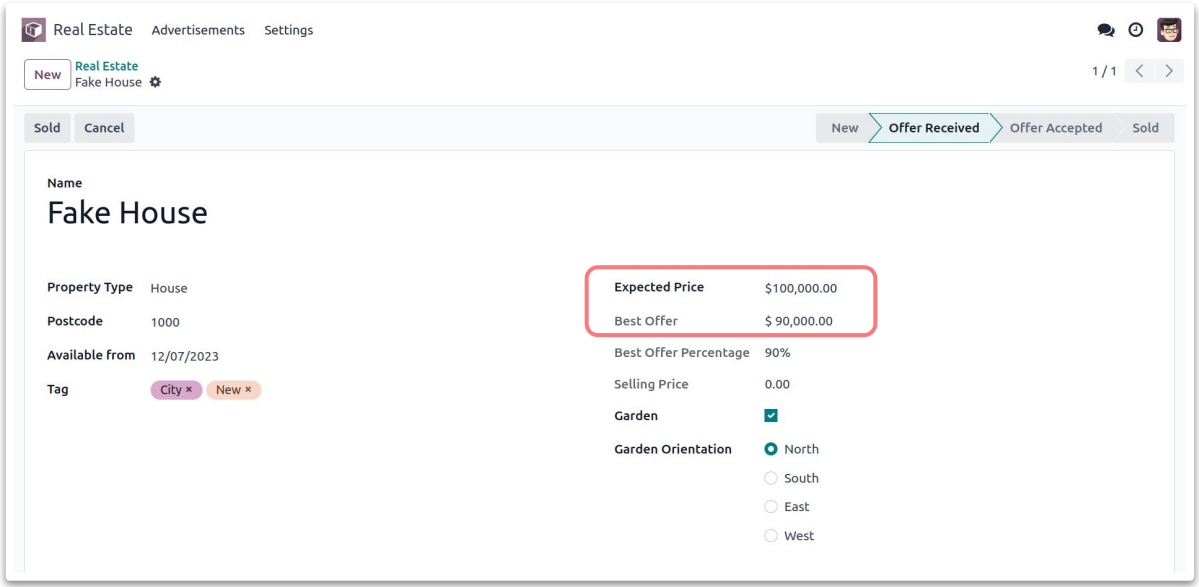


Display selection field as status bar in header

CLICK HERE

Widgets

- statusbar
- **monetary**
- percentage
- radio
- many2many_tags
- ...



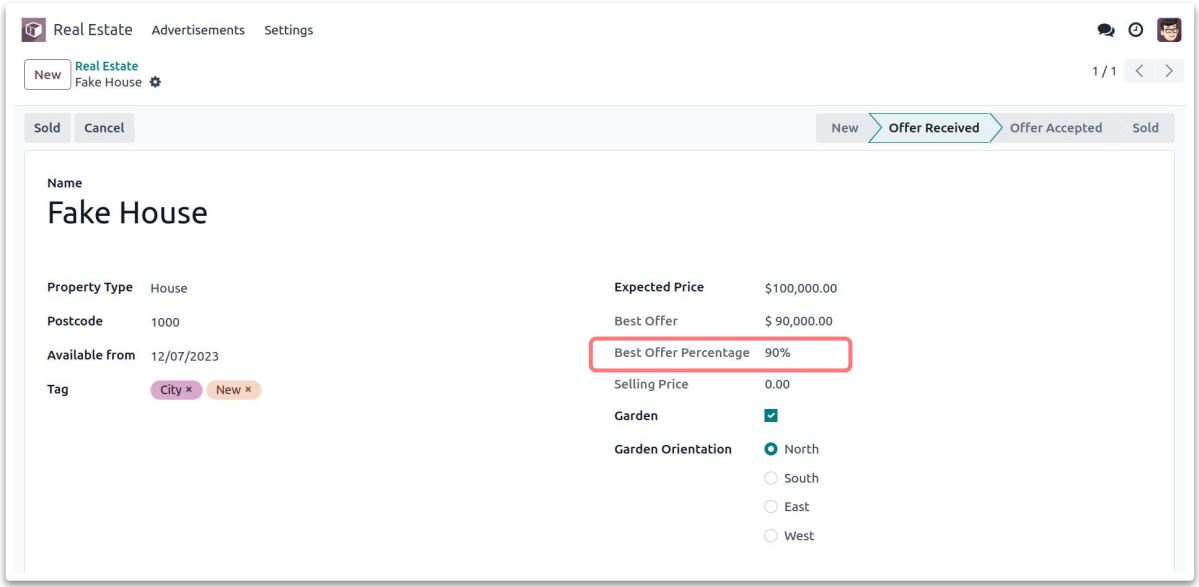
Add currency logo next to field

require currency_id field in model and in view (invisible)

CLICK HERE

Widgets

- statusbar
- monetary
- **percentage**
- radio
- many2many_tags
- ...

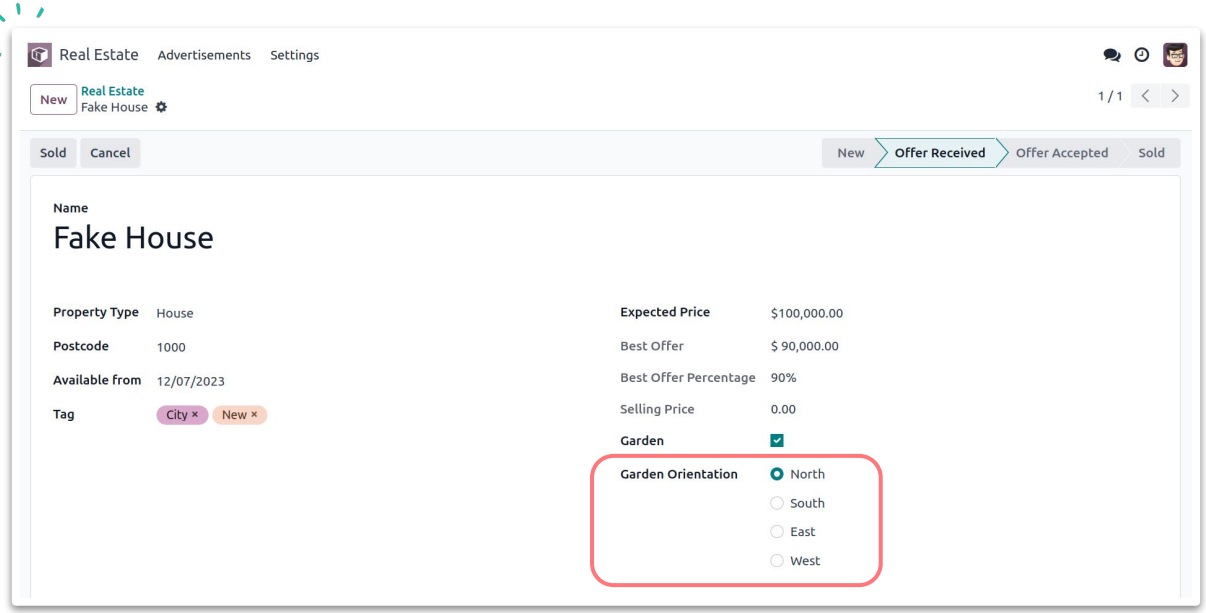


Converts Float field into percentage

CLICK HERE

Widgets

- statusbar
- monetary
- percentage
- **radio**
- many2many_tags
- ...

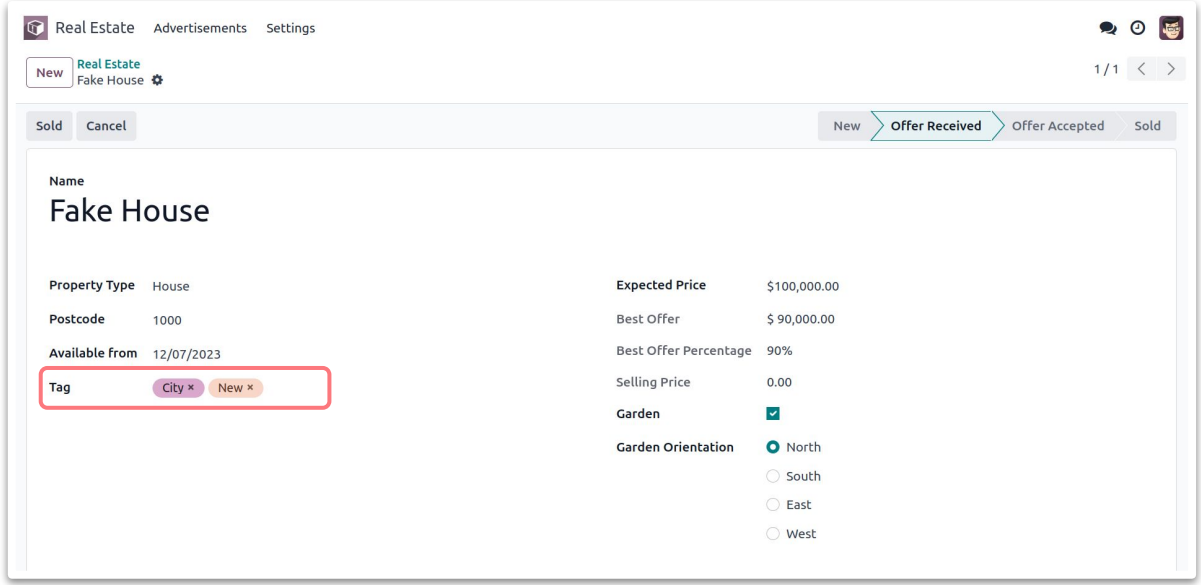


Display selection field in a list of radio buttons

CLICK HERE

Widgets

- statusbar
- monetary
- percentage
- radio
- **many2many_tags**
- ...



Show Many2many fields in a list of tags
(can have colors with option field parameter)

CLICK HERE

Widgets definition

In **each field** defined in the view

→ add the **widget** attribute

→ add some **options** if necessary

```
...
 <field
        name="tag_ids"
        widget="many2many_tags"
        options="{'color_field': 'color'}"
 />
...
```

CLICK HERE

Exercise

odoo PARTNERS

Chapter 13

Simple widgets

Use widgets in the `estate.property` model.

1) Use the `statusbar` widget in order to display the `state` of the as depicted in the **Goal** of this section in the next slide.

2) Make the **Garden Orientation** a **radio button** selection

Tip: a simple example can be found here.

Exercise

odoo PARTNERS

Chapter 13

~~Simple widgets~~

Advanced widgets

Add widget **options**.

- Add the appropriate option to the `property_type_id` field to prevent the creation and the editing of a property type from the property form view.

  Have a look at the Many2one widget documentation for more info.

- Add the appropriate option to the `tag_ids` field (in the model) to add a color picker on the tags.

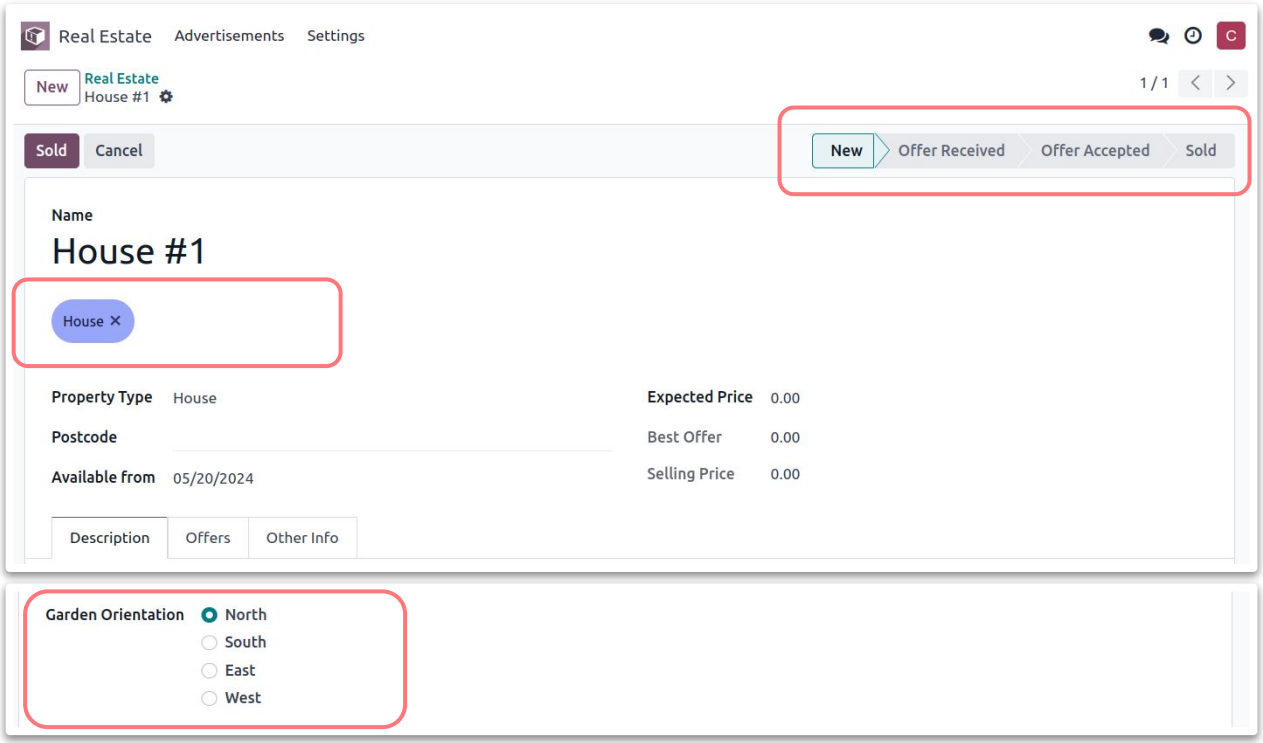
  Have a look at the FieldMany2ManyTags widget documentation for more info.

Model	Field	Type
`estate.property.tag`	Color	Integer

Exercise

Chapter 13

Simple widgets



odoo
PARTNERS

Chapter 14

Ordering

Ordering

2 ways of ordering

- **Model Ordering** - Based on fields values
- **Manual Ordering**

Model Ordering

Using the **_order** attribute in the model

Will be converted to an **order_by** clause in **SQL**

```python
from odoo import fields, models

class TestModel(models.Model):
    _name = "test_model"
    _description = "Test Model"
    _order = "date,price desc"

    date = fields.Date()
    price = fields.Float()
```

(using asc as default sorting direction)

CLICK HERE

CLICK HERE

View Ordering

From tree view attribute

using **default_order**

```
<tree string="My activities" default_order="date desc">
    <field name="opportunity"/>
    <field name="date"/>
    <field name="price"/>
    ...
</tree>
```

In **different** tree views if should be sorted differently

```
<tree string="My activities" default_order="price">
    <field name="opportunity"/>
    <field name="date"/>
    <field name="price"/>
    ...
</tree>
```

CLICK HERE

Exercise

odoo PARTNERS

Chapter 14

Model ordering

CLICK HERE

Add **ordering** in the different models:

Model	Order
`estate.property`	Descending ID
`estate.property.offer`	Descending Price
`estate.property.tag`	Name
`estate.property.type`	Name

Manual Ordering

Create ***sequence*** integer field and use model ordering on it

```python
from odoo import fields, models

class TestModel(models.Model):
    _name = "test_model"
    _description = "Test Model"
    _order = "sequence desc"

    sequence = fields.Integer(default=1)
```

Define in **tree view** and use of widget ***handle***

```xml
<tree string="My activities" >
    <field name="sequence" widget="handle"/>
    <field name="opportunity"/>
    <field name="contact_name"/>
    <field name="email"/>
    ...
</tree>
```

Exercise

odoo PARTNERS

Chapter 14

~~Model ordering~~

Manual ordering

Add **manual ordering**.

- Add the following field:

Model	Field	Type
`estate.property.type`	Sequence	Integer

- Add the sequence to the `estate.property.type` list view with the correct **widget**.

  Tip: you can find an example here: model and view.

odoo
PARTNERS

Chapter 15

Advanced views

IN Views modification

Field Attributes

- **Python Code**: Available to all views
- **XML View**: Only apply on specific view

```python
from odoo import fields, models

class TestModel(models.Model):
    _name = "test_model"
    _description = "Test Model"

    name = fields.Char(required=True, readonly=False)
```

Overriding

Field attributes defined in **XML** can **override** the **Python** Code

```xml
...
 <field name="name" readonly="1" />
...
```

Views before

You might see use of *"attrs"* and *"states"* *in some old views*

```
<field name="name" attrs="{'invisible': [('display_name', '=', False)], 'readonly': [('amount', '!=', 0)]}" />
```

```
<button name="action_payslip_cancel" string="Cancel" type="object" states="draft,done,verify" />
```

Views before

You might see use of *"attrs"* and *"states"* *in some old views*

```
<field name="name" attrs="{'invisible': [('display_name', '=', False)], 'readonly': [('amount', '!=', 0)]}" />
```

```
<button name="action_payslip_cancel" string="Cancel" type="object" states="draft,done,verify" />
```

→ not valid since Odoo 17

Fields attributes

3 Simple in view attributes

- **readonly**
- **invisible**
- **required**

Deal with Python booleans expressions

```
<field name="description" invisible="is_partner" readonly="1" required="state in ('draft', 'confirmed')"/>
```

*To work, fields defined in condition **must** also be present in the same view !*

Exercise

odoo PARTNERS

Chapter 15

Attributes

Use the `invisible` attribute in `estate.property`:

1) Add a **conditional display** for the header buttons (*'Sold'* and *'Cancel'*).

   Both buttons should disappear once the property is either sold or cancelled.

2) Make the **garden area** and orientation invisible when there is no garden.

NB: Do not use `states` or `attrs -> invisible,readonly,required !`

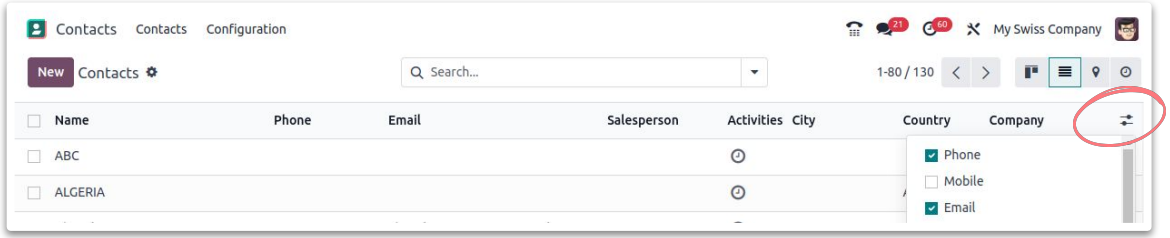
List view attributes

Same simple attributes

- Readonly
- Invisible
- Required

Specific attributes for list views

- ***optional***



- ***column_invisible*** (can use → *parent.xxxx*)

```
<field name="price_description" optional="True" column_invisible="parent.total_amount > 0"/>
```

Use of view attribute can be useful to **prevent** data entry errors, but have **no** level of **security**!

→ There is no server-side check done

Exercise

odoo PARTNERS

Chapter 15

~~Attributes~~

Advanced attributes

Use the attributes in `estate.property.offer`:

1) Make the *'Accept'* and *'Refuse'* buttons `invisible` once the offer state is set.

2) Do not allow adding an offer when the property state is *'Offer Accepted'*, *'Sold'* or *'Canceled'*. To do this use the `readonly` attribute.

NB: Do not use `states` or `attrs -> invisible,readonly,required !`

Exercise

odoo PARTNERS

Chapter 15

~~Attributes~~

~~Advanced attributes~~

List attributes

Make the `estate.property.offer` and `estate.property.tag` list views editable:

→ inside the **<tree>** tag, use the `editable` attribute

CLICK HERE

Views decorators

Color codes are useful to emphasize records

CLICK HERE

decoration-{$name} attribute in view definition

Can be done:

- On line basis (tree view)
- On field basis

```
<tree decoration-success="is_partner">
      <field name="name"/>
      <field name="is_partner"/>
      ...
</tree>
```

```
<field name="name"/>
<field name="is_low" invisible="1"/>
<field name="budget" decoration-danger="is_low"/>
      ...
```

Exercise

odoo PARTNERS

Chapter 15

~~Attributes~~

~~Advanced attributes~~

~~List attributes~~

List decorator

CLICK HERE

Add some **decorations**.

On the `estate.property` list view:

- Properties with an offer received are green
- Properties with an offer accepted are green and bold
- Properties sold are muted
- Add 1 field optional

On the `estate.property.offer` list view:

- Refused offers are red
- Accepted offers are green
- The state should not be visible anymore

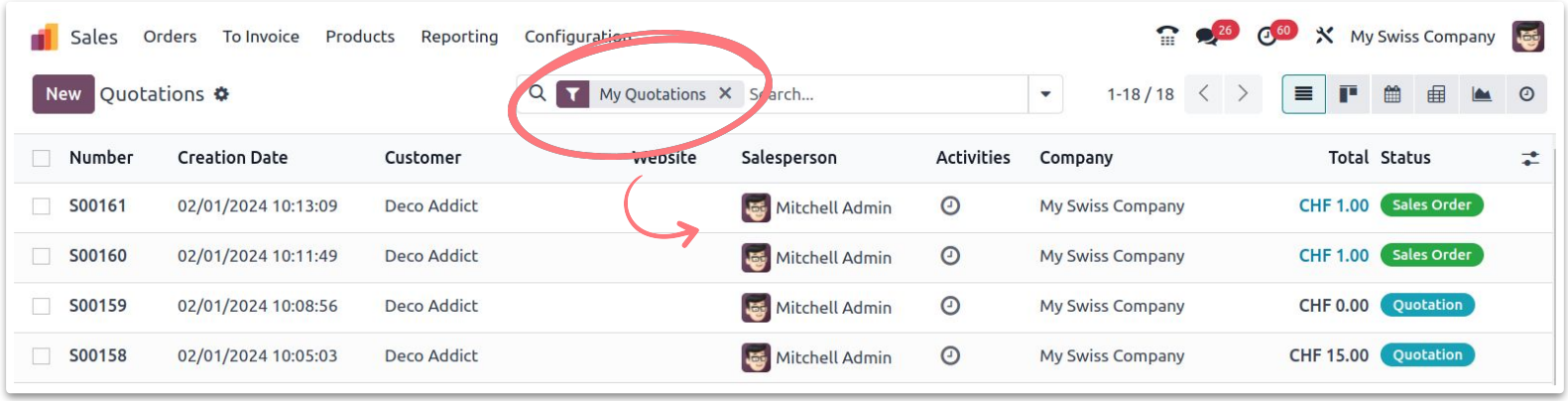
odoo
PARTNERS

Chapter 16

Advanced searches

Default Search

We want to set a **default filter** when we enter a view.



CLICK HERE

Default Search

- Create a **filter** in search view
- Add **context key** in action definition

```
<search string="Tests">
    <filter string="New" name="filter_state" domain="[('state','=','new')]"/>
</search>
```

```
<record id="..." model="ir.actions.act_window">
    <field name="name">Test action</field>
    <field name="res_model">test.model</field>
    <field name="view_mode">...</field>
    <field name="context">{'search_default_filter_new_state': True,}</field>
</record>
```

Exercise

odoo PARTNERS

Chapter 16

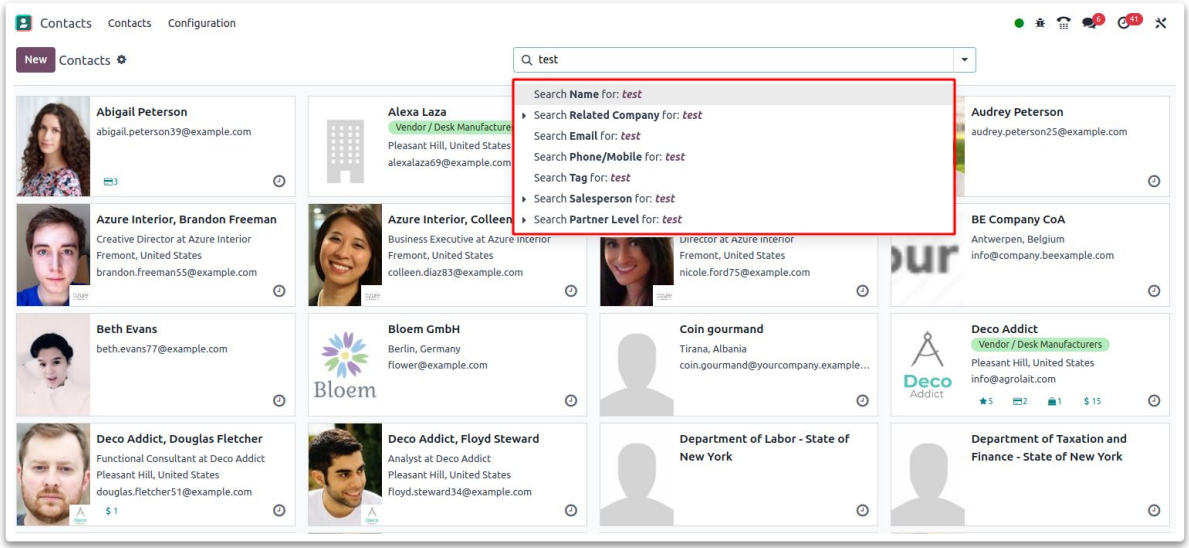
Default filter

Add a **default filter** in `estate.property`.

Make the *'Available'* filter selected by default when opening the views.

Filter domain

Allow users to **search** a record by its field **without an exact match**



CLICK HERE

Filter domain

- Add a **field** in search view
- Add **filter_domain key** attribute

```
<search string="Tests">

    <field name="description" string="Name and description"
  filter_domain="['|', ('name', 'ilike', self), ('description', 'ilike', self)]"/>

</search>
```

→use **self** to reflect the user's input

Exercise

odoo PARTNERS

Chapter 16

~~Default filter~~

Filter Domain

Change the living area **search** in `estate.property`.

- Add a `filter_domain` to the **living area** to include properties with an area equal to or greater than the user's input.

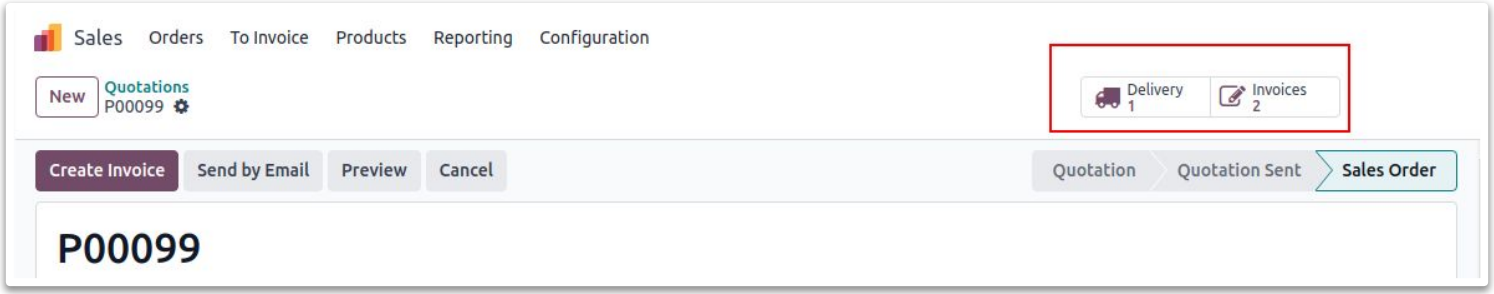
.

odoo
PARTNERS

Chapter 17

More buttons

Stat button



- Allows users to open views to display **related records**
- Can show a **counter** of those records

CLICK HERE

Button definition

- Define a public **method** in the class
- **return** a dictionary

→ keys will be same elements as fields defined in action record defined in the data
(without *id* needed)

```python
from odoo import _, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    line_ids = fields.One2many("related.model", "field_id")

    def action_open_line_ids(self):
        return {
            "name": _("Related Lines"),
            "type": "ir.actions.act_window",
            "view_mode": "tree,form",
            "res_model": "related.model",
            "target": "current",
            "domain": [("id", "=", self.id)],
            "context": {"default_field_id": self.id},
        }
```

- ***domain*** → display all the elements with this field as inverse field
- ***context*** → if new record created then, links it to source record

Button definition

- In the view, add a ** ** with the class *oe_button_box*
- Inside, add a **<button>**

→ *name* attribute must be the same name of the method

→ *type* attribute must *"object"*

→ *class* attribute must be *"oe_stat_button"*

→ chose an **fa-icon**

```python
class TestModel(models.Model):
    ...

    def action_open_line_ids(self):
        ...
```

```xml
<form>
    <sheet>
        <div class="oe_button_box"
             name="button_box">
            <button name="action_open_line_ids"
                    type="object"
                    class="oe_stat_button"
                    icon="fa-icon" />
        </div>
        ...
    </sheet>
</form>
```

Button definition

Add a counter !

- Define new **computed** field in model
- Add the field in the view, inside button tags

```python
from odoo import _, fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    line_count = fields.Integer(compute="_compute_line_count")

    @api.depends("line_ids")
    def _compute_line_count(self):
        for rec in self:
            rec.line_count = len(rec.line_ids)
```

```xml
<div class="oe_button_box" name="button_box" >
    <button name="action_open_line_ids" ... >
        <field name="line_count"
               string="Lines"
               widget="line_count"/>
    </button>
</div>
```

Exercise

odoo PARTNERS

Chapter 17

Button context

Add the field `property_type_id` to `estate.property.offer`.

- We can define it as a related field on `property_id.property_type_id` and set it as stored.

Thanks to this field, an offer will be linked to a property type when it's created. You can add the field to the list view of offers to make sure it works.

- Add the field `offer_ids` to `estate.property.type` which is the One2many inverse of the field defined in the previous step.

Add the field `offer_count` to `estate.property.type`.

- It is a computed field that counts the number of offers for a given property type (use `offer_ids` to do so).

Exercise

odoo PARTNERS

Chapter 17

~~Button context~~

Stat Button

CLICK HERE

Display the **list** of **offers** when clicking on the stat button.

- Create a stat button on `estate.property.type` pointing to the `estate.property.offer` **action**.

  This means you should use the `type="action"` attribute.

At this point, clicking on the stat button should display all offers. We still need to **filter out** the offers.

- On the `estate.property.offer` action, add a domain that defines `property_type_id` as equal to the `active_id` (= the current record)

odoo PARTNERS

Chapter 18

Model Inheritance

Default inheritance

`test.model` inherits from the **Model** class

automatically gives the **CRUD** methods:

- `create()` ~ **C**
- `read()` ~ **R**
- `write()` ~ **U**pdate
- `unlink()` ~ **D**elete

```python
class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"


    @api.model (/create_multi)
    def create(self, vals):
        # Do some business logic, modify vals...
        ...
        # Then call super to execute the parent method
        return super().create(vals)

    def read(self, fields=None, load='_classic_read'):
        # Do some business logic
        return super().read(fields, load)

    def write(self, vals):
        # Do some business logic
        return super().write(vals)

    def unlink(self, vals):
        # Do some business logic
        return super().unlink()
```

→ **always** call `super()`

→ **always** return data consistent in parent method.

Exercise

odoo PARTNERS

Chapter 18

CRUDS

Add business logic to the **CRUD** methods.

- Prevent deletion of a property if its state is not '**New**' or '**Canceled**'

  Tip: create a new method with the `ondelete()` decorator and remember that `self` can be a recordset with more than one record.

- At offer creation, set the property state to '**Offer Received**'. Also raise an error if the user tries to create an offer with a lower amount than an existing offer.

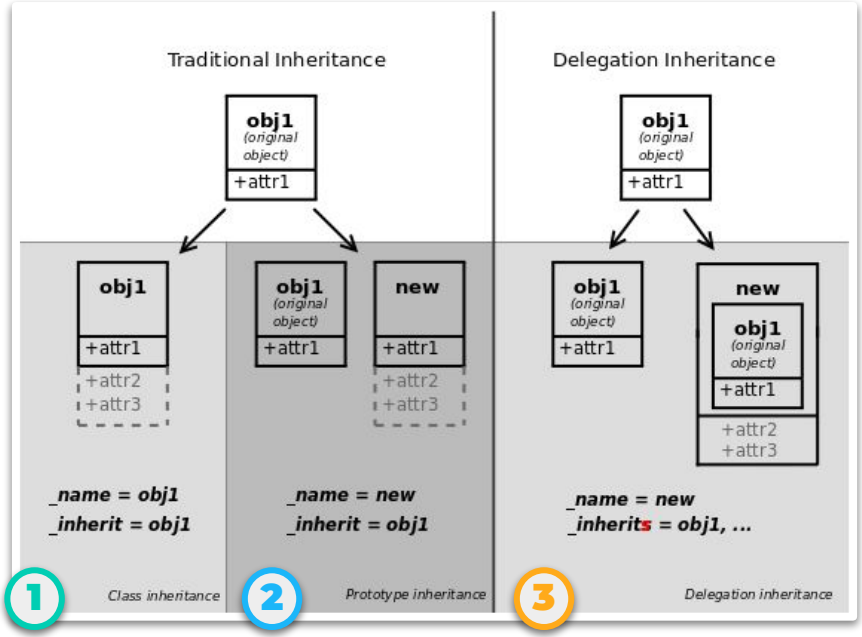
Tip: The `property_id` field is available in the `vals`, but it is an `int`.
To instantiate an `estate.property` object,
use `self.env[model_name].browse(value)`

CLICK HERE

Model Inheritance

3 mechanisms to extend models

1. **Extending** existing model
2. **Creating** new model from existing one and add new information
3. **Delegating** some of the model's fields to records it contains



CLICK HERE

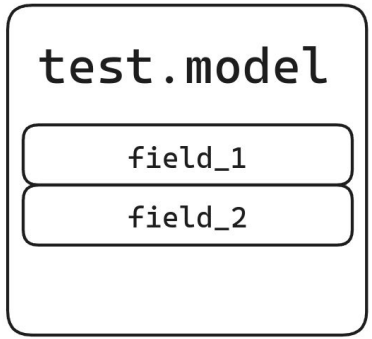
Class inheritance

Module A

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Char()
    field_2 = fields.Char()
```

We have a simple **model** in a module

CLICK HERE

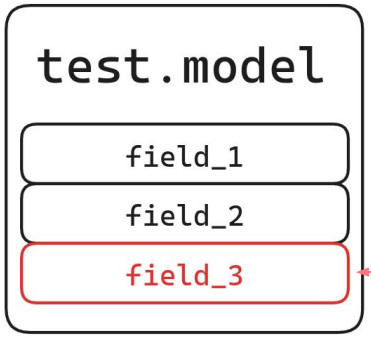


Class inheritance

Define a **new class** that extends an existing model

using `_inherit`

→ `test.model` will have properties defined in both classes



Module A

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Char()
    field_2 = fields.Char()
```

Module B

```python
from odoo import fields, models


class TestModel(models.Model):
    _inherit = "test.model"

    field_3 = fields.Char()
```

Exercise

odoo PARTNERS

Chapter 18

~~CRUDS~~

Class inheritance

Add a field to **Users** (`res.users` model)

- Add the following field to the existing model:

Field	Type
property_ids	One2many inverse of the field that references the salesperson in `estate.property`

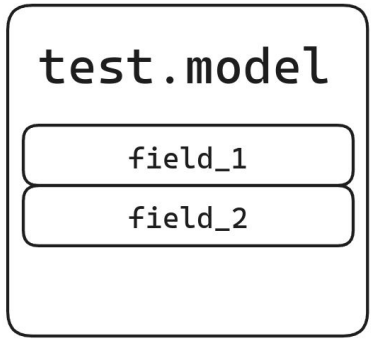
- Add a domain to the field so it only displays the **available** properties.

Prototype inheritance

We have a simple **model** in a module

CLICK HERE

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Char()
    field_2 = fields.Char()
```



Prototype inheritance

New model → **extension** of first model with both:

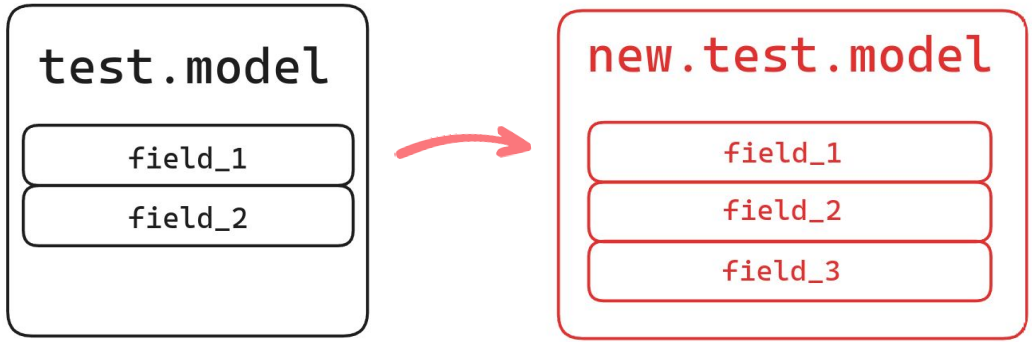
- _name = ...
- _inherit = ...

New model

→ **Possess** first model functionality
(fields, methods and meta-information)

→ **Can add** its own functionality

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Char()
    field_2 = fields.Char()


class TestModel(models.Model):
    _name = "new.test.model"
    _inherit = "test.model"

    field_3 = fields.Char()
```



Exercise

odoo PARTNERS

Chapter 18

~~CRUDS~~

~~Class inheritance~~

Prototype inheritance

Add **chatter** into existing `estate.property` model.

Hint: you might look for `mail.thread` and `mail.activity.mixin`

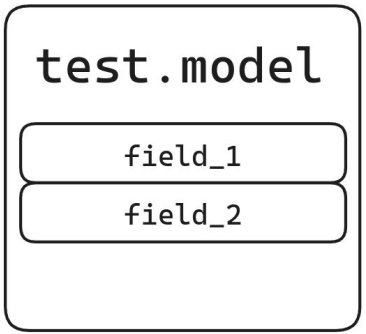
and check in code to add in form view for `class="oe_chatter"`

Delegation inheritance

We have a simple **model** in a module

CLICK HERE

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Char()
    field_2 = fields.Char()
```



Delegation inheritance

New model that **embeds** an object of another model **as a field** by:

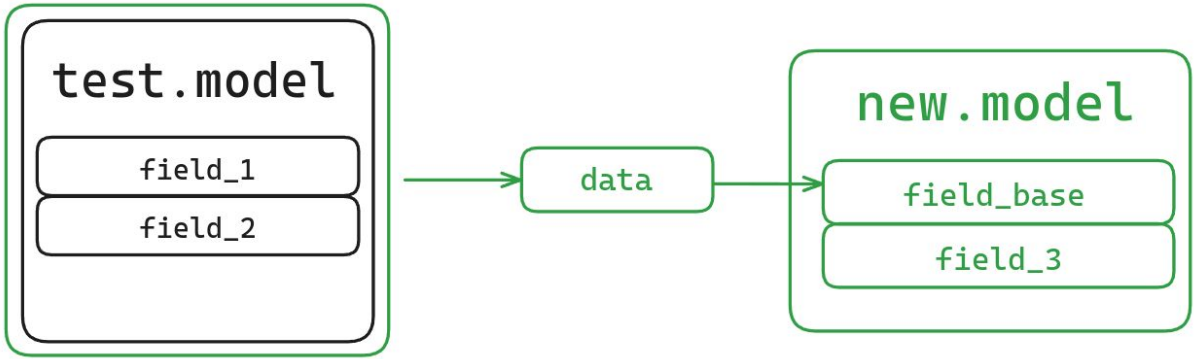
- _inherits = {'model_name': 'field_name'}

In Odoo:

Products (`product.template`)

↳ **Product Variants** (`product.product`)

```python
from odoo import fields, models


class TestModel(models.Model):
    _name = "test.model"
    _description = "Test Model"

    field_1 = fields.Char()
    field_2 = fields.Char()

class TestModel(models.Model):
    _name = "test.model"
    _inherits = {"test.model": "field_base"}

    field_base = fields.Many2one("test.model")
    field_3 = fields.Char()
```



odoo
PARTNERS

Chapter 19

View Inheritance

View Inheritance

- Define a **New** view

  as seen in chapter 6

```xml
<record id="MODEL_view_TYPE" model="ir.ui.view">
  <field name="name">model.view.type</field>
  <field name="model">model</field>
  <field name="arch" type="xml">
    <VIEW_TYPE>
       <VIEW_SPECIFICATIONS/>
    </VIEW_TYPE>
  </field>
</record>
```

- **Extend** an existing one

  new field used : inherid_id

```xml
<record id="inherit_MODEL_view_TYPE" model="ir.ui.view">
  <field name="name">model.view.type.inherit</field>
  <field name="model">model</field>
  <field name="inherit_id" ref="module.inherited_view_id"/>
  <field name="arch" type="xml">
      <xpath expr="..." position="...">
         <!-- do the appropriate modification... -->
      </xpath>
  </field>
</record>
```

CLICK HERE

Elements modification

expr: locating single element in parent view

position: action to do

```
<xpath expr="//field[@name='name']" position="after">
   <!-- do the appropriate modification... -->
</xpath>
```

elements depending on position value

CLICK HERE

Xpath - expr

Locating a node on the view we are inheriting

```
<xpath expr="//div[hasclass('oe_button_box')" position="...">
    <!-- finds the first div element using a class 'oe_button_box' ... -->
</xpath>
```

```
<xpath expr="//field[@name='name']" position="...">
    <!-- finds the first field tax anywhere with name: 'name' ... -->
</xpath>
```

CLICK HERE

View to override

```
<form>
    <sheet>
        <div class="oe_button_box">
            <BUTTONS/>
        </div>
        <group>
            <group>
                <field name="name"/>
            </group>
        </group>
        <notebook>
            <page string="Page1">
                <group>
                    <CONTENT/>
                </group>
            </page>
            <page string="Page2">
                <group>
                    <CONTENT/>
                </group>
            </page>
        </notebook>
    </sheet>
</form>
```

Xpath - position

```
<xpath expr="..." position="KEY">
    <!-- BODY used based on the position value -->
    <group>
        <field name="new_field_1"/>
    </group>
    <group>
        <field name="new_field_2"/>
    </group>
</xpath>
```

Keys

before - inserts BODY before the matched element

after - inserts BODY after the matched element

inside - appends BODY to the end of the matched element

replace - replaces the matched element with BODY

CLICK HERE

Xpath - position - attributes

```
<xpath expr="..." position="attributes">

    <attribute name="invisible">0</attribute>
    <attribute name="required">1</attribute>
    <attribute name="readonly">0</attribute>

</xpath>
```

attributes - alters the attributes of the matched element using the BODY's elements

CLICK HERE

Exercise

odoo PARTNERS

Chapter 19

View inheritance

Add new fields to the **Users** view.

Override the `base.view_users_form` to add the `property_ids` field in a new notebook page.


End