Technical Report

Prepared by:

EL HAJALI Imad Eddine

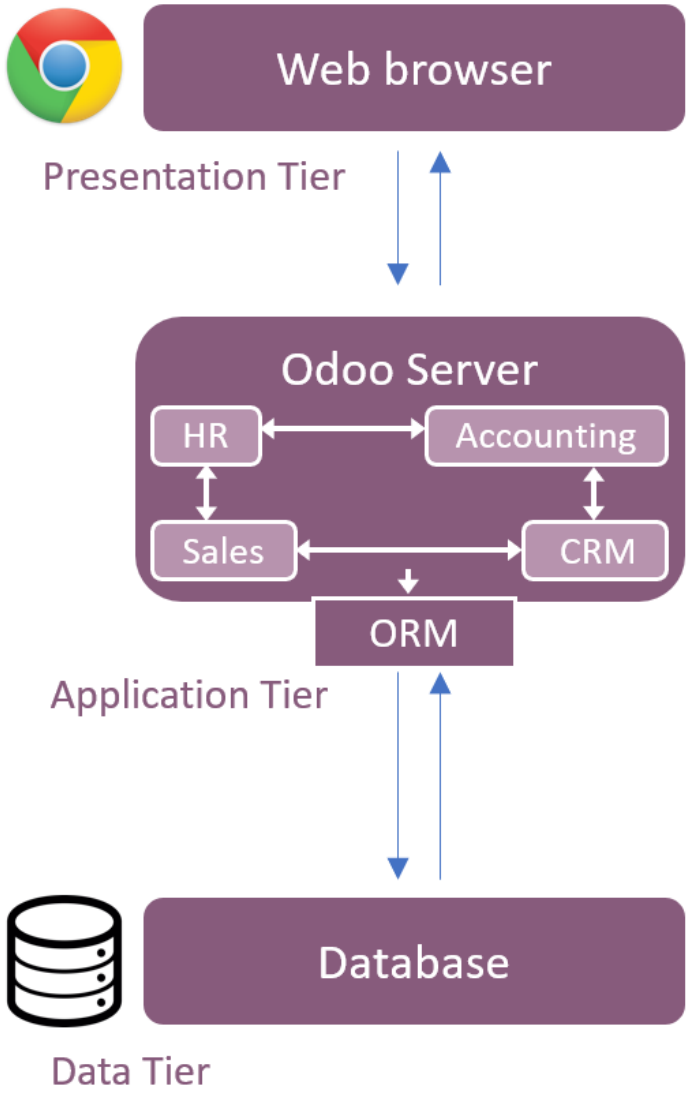
Casablanca, 03 December 2023

Contents

Chapter 1: Architecture Overview

1. Architecture:

Odoo follows a multitier architecture, specifically a three-tier architecture. The presentation tier is comprised of HTML5, JavaScript, and CSS. The logic tier is exclusively written in Python, and the data tier supports PostgreSQL as an RDBMS.



2. Development Tiers:

Odoo development can occur in any of the three tiers, depending on the module's scope. Basic knowledge of HTML and intermediate Python skills are recommended. Advanced topics may require further expertise in these areas.

3. Frameworks:

Starting from version 15.0, Odoo is transitioning to its in-house developed OWL framework for the presentation tier, gradually deprecating the legacy JavaScript framework.

4. Composition of a Module:

A module can include business objects (Python classes mapped to database columns), object views (defining UI display), data files (XML/CSV for model data, views, configuration, demonstration data), web controllers (handling browser requests), and static web data (images, CSS, JavaScript).

5. Module Structure:

Each module is a directory within a module directory. Module directories are specified using *the --addons-path option*. A module is declared by its manifest and organized with a structure containing models, data, and manifest files.

6. Odoo Editions:

Odoo comes in two versions—Odoo Enterprise (licensed & shared sources) and Odoo Community (open-source). The Enterprise version includes additional functionalities installed on top of the Community version modules. Technical differences arise from these added functionalities.

Chapter 2: Development Environment Setup

1. Installation Methods:

For Odoo developers, the recommended installation method is a source install, running Odoo from the source code.

```
$ git clone https://github.com/odoo/odoo.git
$ git clone https://github.com/odoo/enterprise.git
```

2. Requirements:

- ✓ Install Python 3.10 or later.
- ✓ Install PostgreSQL (12.0 or higher) is used as the database system and follow these steps:

1. Ajoutez le dépôt `bin` de PostgreSQL (par défaut : *C:\Program Files\PostgreSQL\<version>\bin*) au `PATH`.
2. Créez un utilisateur postgres avec un mot de passe en utilisant le guide pg admin :
    1. Ouvrez **pgAdmin**.
    2. Double-cliquez sur le serveur pour créer une connexion.
    3. Sélectionnez **Objet ▸ Créer ▸ Rôle Login/Groupe**.
    4. Saisissez le nom d'utilisateur dans le champ **Nom de rôle** (par ex. `odoo`).
    5. Ouvrez l'onglet **Définition**, saisissez un mot de passe (par ex. `odoo`), et cliquez sur **Enregistrer**.
    6. Ouvrez l'onglet **Privilèges** et définissez **Peut se connecter ?** sur `Oui` et **Créer une base de données ?** sur `Oui`.

3. Dependencies:

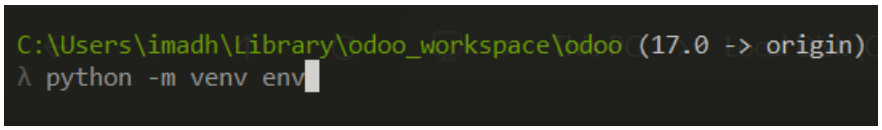
Dependencies for Odoo are listed in the requirements.txt file. Developers are guided to download and install **Visual Studio C++ tools** for Windows.

4. Running Odoo:

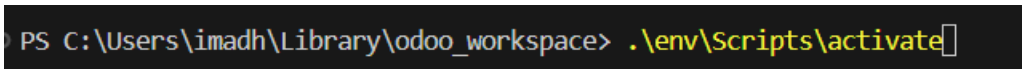
After cloning, you install a tool that helps with creating virtual environments.

```
C:\Users\imadh
λ pip install virtualenv
```

Create a virtual environment inside Odoo folder with the command:



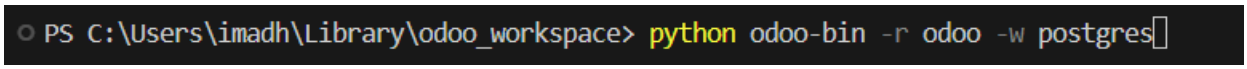
Then, activate the virtual env with the command:



Then, you can install the dependencies with these commands:

```
C:\> pip install setuptools wheel
C:\> pip install -r requirements.txt
```

After installing you can run the Odoo server with the command:



Chapter 3: A New Application

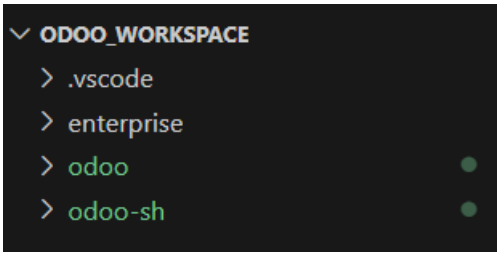
1. Real Estate Advertisement Module:

The new module covers the real estate business area, providing a main list view and form views for property details and offers.

2. Preparing the Addon Directory:

To have Odoo recognize the new module, developers must create its directory in the technical-training-sandbox directory. The module must contain at least two files: manifest.py and init.py. The manifest.py file describes the module, including its name, category, summary, website, and dependencies.

First create a new folder called “odoo-sh” in my case to develop your module inside it.



Then, the first two files you need to add are manifest.py and init.py

3. Making the Module an "App":

The chapter guides developers in adding the appropriate key to the manifest.py file to make the module appear when the "Apps" filter is enabled. The developer mode must be enabled for this process. Although the module is currently an empty shell, developers can install it.

estate_property.py U __manifest__.py U

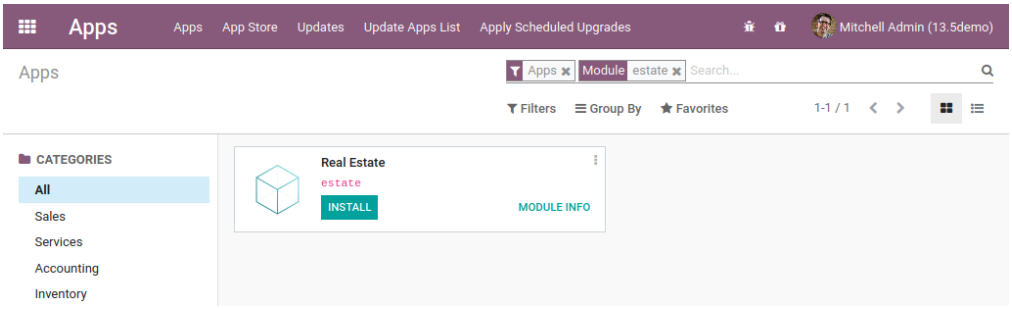
odoo-sh > estate > __manifest__.py

```python
{
    "name": "Real Estate",
    "summary": """
    In summary, this is a real estate module for Berexia's technical training.
    """,
    "description": """
        Module Description : This is a real estate module for Berexia's technical training.
    """,
    "category": "",
    "version": "17.0.0.0.1",
    "author": "Imad Eddine",
    "website": "https://github.com/ImadEddinee",
    "license": "OEEL-1",  # Or any other type of license
    "depends": ['base'],
    "data": [
        # --- Security --
        'security/ir.model.access.csv',
        # --- Views ---
        'views/estate_property_views.xml',
        'views/estate_property_type_views.xml',
        'views/estate_property_tag_views.xml',
        'views/estate_property_offer_views.xml',
        # --- Menus ---
        'views/estate_menu.xml',  # Since Menus load actions they should be defined after
        # --- Data ---
    ],
    "demo": [],
    "installable": True,
    "auto_install": False,
    "application": True
}
```

Then, you can run the project with the command:

```
(env) PS C:\Users\imadh\Library\odoo_workspace\odoo> python odoo-bin -d odoo -r odoo -w postgres --addons-path ..\enterprise,..\addons,..\odoo\addons,..\odoo-sh  -u estate
```

When you run you should see something like:



Chapter 4: Models and Basic Fields

1. Object-Relational Mapping (ORM):

The ORM layer in Odoo is a key component that automates database interactions, eliminating the need to manually write SQL queries. Business objects are declared as Python classes extending the Model class, and these classes are integrated into the automated persistence system. Models are configured by setting attributes in their definition, with _name being a required attribute that defines the model's name in Odoo.

2. Exercise:

Define the "estate.property" model based on the example in the CRM module

estate_property.py U

odoo-sh > estate > models > estate_property.py > ...

```python
from odoo import models, fields, api
from odoo.exceptions import UserError
from odoo.tools.float_utils import float_compare, float_is_zero


class RealEstate(models.Model):
    _name = "real_estate"  # The ORM will replace . by _ for the table name
```

3. Adding Description to the Model:

Adding a description to the model is recommended to provide additional information about the purpose of the model. This involves modifying the Python files and restarting the Odoo server with the appropriate parameters.

4. Exercise:

Add a _description attribute to the estate.property model to eliminate a warning.

estate_property.py U

odoo-sh > estate > models > estate_property.py > ...

```python
from odoo import models, fields, api
from odoo.exceptions import UserError
from odoo.tools.float_utils import float_compare, float_is_zero


class RealEstate(models.Model):
    _name = "real_estate"  # The ORM will replace . by _ for the table name
    _description = "Estate Property"
```

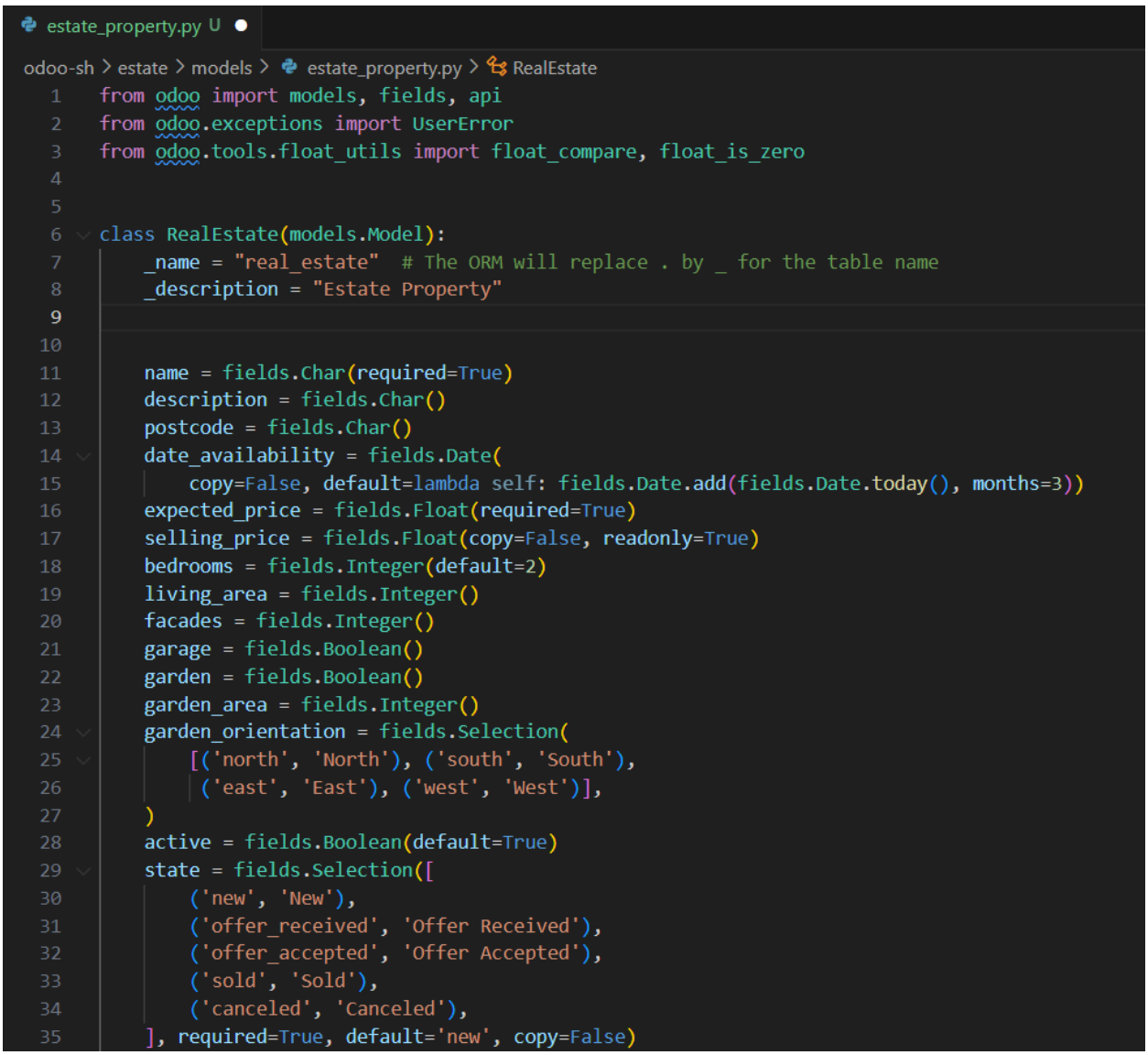
5. Model Fields and Common Attributes:

Fields are used to define what data the model can store and where it is stored. Two broad categories of fields are "simple" fields (Boolean, Float, Char, etc.) and "relational" fields that link records of the same or different models. Examples of simple fields include Char, Text, Date, Float, Integer, and Boolean.

Fields can be configured using attributes like string, required, help, and index. Common attributes like string and required can be set to customize the field's behavior and appearance.

6. Exercise:

Add basic fields to the estate_property table, including Char, Text, Date, Float, Integer, and Boolean types. A Selection field is also added with four possible values. Set attributes such as required for the name and expected_price fields to make them not nullable.

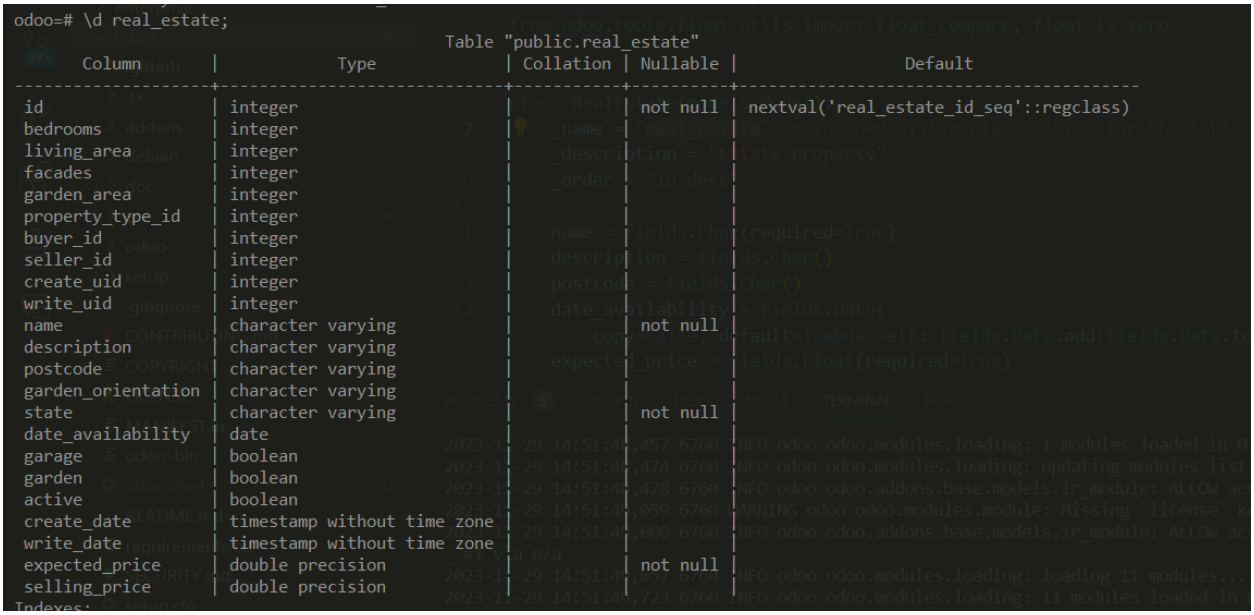


7. Automatic Fields:

Odoo automatically creates certain fields in all models, including id, create_date, create_uid, write_date, and write_uid. These fields are managed by the system and provide information about record creation and modification. Note: It is possible to disable the automatic creation of some fields.

8. Exercise:

Understand and acknowledge the existence of automatic fields in the estate.property model.



```
odoo=# \d real_estate;
                                    Table "public.real_estate"
      Column      |            Type             | Collation | Nullable |                 Default
------------------+-----------------------------+-----------+----------+----------------------------------------
 id               | integer                     |           | not null | nextval('real_estate_id_seq'::regclass)
 bedrooms         | integer                     |           |          |
 living_area      | integer                     |           |          |
 facades          | integer                     |           |          |
 garden_area      | integer                     |           |          |
 property_type_id | integer                     |           |          |
 buyer_id         | integer                     |           |          |
 seller_id        | integer                     |           |          |
 create_uid       | integer                     |           |          |
 write_uid        | integer                     |           |          |
 name             | character varying           |           | not null |
 description      | character varying           |           |          |
 postcode         | character varying           |           |          |
 garden_orientation | character varying         |           |          |
 state            | character varying           |           | not null |
 date_availability | date                       |           |          |
 garage           | boolean                     |           |          |
 garden           | boolean                     |           |          |
 active           | boolean                     |           |          |
 create_date      | timestamp without time zone |           |          |
 write_date       | timestamp without time zone |           |          |
 expected_price   | double precision            |           | not null |
 selling_price    | double precision            |           |          |
Indexes:
```

Chapter 5: Security - A Brief Introduction

1. Overview:

This chapter introduces the concept of security in Odoo, specifically focusing on access rights. Security in Odoo involves controlling which users or groups of users can access and perform operations on data. Access rights are defined using data files, particularly CSV files, and are crucial for ensuring proper data protection and user access control.

2. Data Files (CSV):

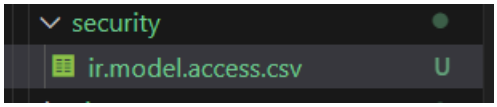
Odoo heavily relies on data-driven systems, and CSV files play a crucial role in loading data during module installation or update. The chapter provides an example of a CSV file for importing country states data, emphasizing the importance of file location conventions (e.g., data for security in the security folder) and proper declaration in the manifest.py file. The sequential loading order of data files is highlighted, stressing the need for careful consideration when files depend on each other.

3. Access Rights:

Access rights are defined using records of the model ir.model.access, associated with a specific model, group (or global access), and permissions (create, read, write, unlink). An example CSV file, ir.model.access.csv, is provided with details on its structure, including external identifiers, model references, group references, and permission flags. The chapter concludes with a practical exercise for adding access rights to a model, emphasizing the elimination of warning messages related to missing access rules.

4. Exercise:

Developers are tasked with creating an ir.model.access.csv file, defining it in the manifest.py file, and assigning read, write, create, and unlink permissions to the base.group_user group. The exercise aims to reinforce the understanding of access rights and eliminate warning messages in the server logs related to missing access rules.



ir.model.access.csv U

odoo-sh > estate > security > ir.model.access.csv

```
1  id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_unlink
2  estate.access_estate_property,access_estate_property,estate.model_real_estate,base.group_user,1,1,1,1
3  estate.access_estate_property_type,estate.property.type,estate.model_estate_property_type,base.group_user,1,1,1,1
4  estate.access_estate_property_tag,estate.property.tag,estate.model_estate_property_tag,base.group_user,1,1,1,1
5  estate.access_estate_property_offer,estate.property.offer,estate.model_estate_property_offer,base.group_user,1,1,1,1
```

Chapter 6: Finally, Some UI To Play With

1. Overview:

This chapter focuses on interacting with the user interface in Odoo, introducing the use of XML files to define actions and menus. It covers the creation of actions and menus for the Real Estate module, providing developers with practical exercises to reinforce their understanding.

2. Data Files (XML):

The chapter distinguishes between CSV and XML file formats for loading data, highlighting the XML format's preference for more complex structures. Developers are encouraged to follow conventions for file locations and declarations in the manifest.py file. The sequential loading order of data files is emphasized, along with a note on performance considerations favoring CSV over XML for speed.

3. Actions:

Actions in Odoo are introduced as records in the database, and a basic action example is provided. Actions link menus to specific models and are triggered by menu items. The chapter guides developers through creating an action for the estate.property model, with a goal to have the file loaded in the system.

4. Menus:

Menus in Odoo follow a hierarchical structure, including root menus, first-level menus, and action menus. The “<menuitem>” shortcut is introduced to simplify the declaration of menus and their connection to actions. Developers are tasked with creating three levels of menus for the estate.property action. The exercise aims to have three menus created, leading to the display of the default form view.

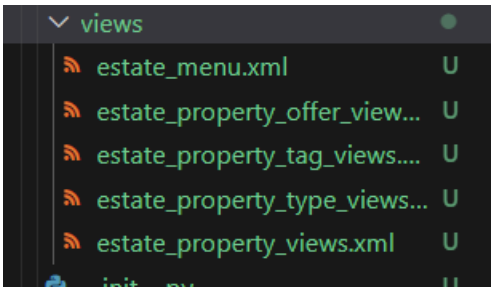
5. Fields, Attributes, and View:

This section delves into model-view interaction, introducing new attributes for fields that impact the view, such as making fields read-only or preventing their values from being copied during duplication. Developers are tasked with adding attributes to the fields to achieve specific behaviors. The concept of default values for fields is covered, and developers are guided through setting default values for specific fields. The chapter concludes with the introduction of reserved fields and exercises related to the "active" field and the addition of a "state" field with specific requirements.

6. Exercise:

a. Add an action:

Create the estate_property_views.xml file.



Define it in the __manifest__.py file.

estate_property.py U __manifest__.py U

odoo-sh > estate > __manifest__.py

```
{
    "name": "Real Estate",
    "summary": """
    In summary, this is a real estate module for Berexia's technical training.
    """,
    "description": """
        Module Description : This is a real estate module for Berexia's technical training.
    """,
    "category": "",
    "version": "17.0.0.0.1",
    "author": "Imad Eddine",
    "website": "https://github.com/ImadEddinee",
    "license": "OEEL-1",  # Or any other type of license
    "depends": ['base'],
    "data": [
        # --- Security --
        'security/ir.model.access.csv',
        # --- Views ---
        'views/estate_property_views.xml',
        'views/estate_property_type_views.xml',
        'views/estate_property_tag_views.xml',
        'views/estate_property_offer_views.xml',
        # --- Menus ---
        'views/estate_menu.xml',  # Since Menus load actions they should be defined after
        # --- Data ---
    ],
    "demo": [],
    "installable": True,
    "auto_install": False,
    "application": True
}

```

Create an action for the model estate.property.

estate_property_views.xml U

odoo-sh > estate > views > estate_property_views.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<odoo>
    <record id="estate_property_action" model="ir.actions.act_window">
        <field name="name">Estate Property</field>
        <field name="res_model">real_estate</field>
        <field name="view_mode">tree,form</field>
        <field name="context">
        {'search_default_filter_available': True}
        </field>
    </record>
</odoo>
```

Restart the server, and the file should be loaded.

b. Add menus:

- ✓ Create the estate_menus.xml file.
- ✓ Define it in the __manifest__.py file.
- ✓ Create three levels of menus for the estate.property action.

estate_menu.xml U

odoo-sh > estate > views > estate_menu.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<odoo>
    <menuitem id="test_menu_root" name="Real Estate">
        <menuitem id="test_first_level_menu" name="Advertisements">
            <menuitem id="test_model_menu_action" name="Property" action="estate_property_action"/>
        </menuitem>
        <menuitem id="test_seconf_level_menu" name="Settings">
            <menuitem id="test_model_action" name="Property Types" action="estate_property_type_action"/>
            <menuitem id="tag_model_action" name="Property Tags" action="estate_property_tag_action"/>
        </menuitem>
    </menuitem>
</odoo>
```

Restart the server and refresh the browser, menus should be visible.

c. Add attributes to fields:

Find appropriate attributes to:

- ✓ Set the selling price as read-only.
- ✓ Prevent copying of availability date and selling price values.

Restart the server and check the expected behaviors.

Set default attributes for:

- ✓ The default number of bedrooms as 2.
- ✓ The default availability date in 3 months.
- ✓ Check that default values are set as expected.

Add the active field:

- ✓ Add the active field to the estate.property model.

Restart the server, create a new property, and observe its behavior.

Set default value for the active field:

- ✓ Check that the default value is set as expected.

Add a state field to the estate.property model with specific requirements:

- ✓ Use the correct type and set it as required with a default value of "New".

estate_property.py U

odoo-sh > estate > models > estate_property.py > RealEstate

```python
from odoo import models, fields, api
from odoo.exceptions import UserError
from odoo.tools.float_utils import float_compare, float_is_zero


class RealEstate(models.Model):
    _name = "real_estate"  # The ORM will replace . by _ for the table name
    _description = "Estate Property"


    name = fields.Char(required=True)
    description = fields.Char()
    postcode = fields.Char()
    date_availability = fields.Date(
        copy=False, default=lambda self: fields.Date.add(fields.Date.today(), months=3))
    expected_price = fields.Float(required=True)
    selling_price = fields.Float(copy=False, readonly=True)
    bedrooms = fields.Integer(default=2)
    living_area = fields.Integer()
    facades = fields.Integer()
    garage = fields.Boolean()
    garden = fields.Boolean()
    garden_area = fields.Integer()
    garden_orientation = fields.Selection(
        [('north', 'North'), ('south', 'South'),
         ('east', 'East'), ('west', 'West')],
    )
    active = fields.Boolean(default=True)
    state = fields.Selection([
        ('new', 'New'),
        ('offer_received', 'Offer Received'),
        ('offer_accepted', 'Offer Accepted'),
        ('sold', 'Sold'),
        ('canceled', 'Canceled'),
    ], required=True, default='new', copy=False)
```

Chapter 7: Basic Views

1. Overview:

This chapter focuses on customizing views in Odoo, specifically addressing list views, form views, and search views. Developers are guided through the process of creating custom views for the Real Estate module, enhancing the presentation and usability of the application.

2. List View:

The list view, also known as the tree view, is introduced as a tabular form displaying records. Developers are provided with an example of a basic list view and tasked with creating a custom list view for the estate.property model.

3. Form View:

Form views, used for creating and editing single records, are explored. The root element is <form>, and developers are presented with an example of a basic form view. The goal is to create a custom form view for the estate.property model, and developers are encouraged to add fields and tags incrementally to understand the process. The **--dev xml** parameter is introduced to facilitate viewing modifications without relaunching the server.

4. Search View:

Search views, different from list and form views, are designed to filter content in other views. The root element is <search>, and developers are provided with a basic example. The goal is to create a custom search view for the estate.property model, allowing users to filter based on specific fields. The concept of filters and the introduction of domains are covered.

5. Domains:

Domains are introduced as encoding conditions on records in Odoo. A domain consists of criteria in the form of triplets with a field name, an operator, and a value. Developers are given examples of domains and logical operators to combine criteria. Entity references are highlighted for special characters in XML. The exercise involves adding filters and a Group By clause to the previously created search view for the estate.property model.

6. Exercise:

a. Add a custom list view:

Define a list view for the estate.property model in the appropriate XML file.

Restart the server, and the list view should reflect the desired organization.

```
<record id="view_real_estate_property_list" model="ir.ui.view">
        <field name="name">real.estate.property.list</field>
        <field name="model">real_estate</field>
        <field name="arch" type="xml">
            <tree   decoration-success="state in ['offer_received','offer_accepted']"
                    decoration-bf="state == 'offer_accepted'"
                    decoration-muted="state == 'sold'"
                    string="Real Estate Properties">
                <field name="name" string='Title'/>
                <field name="state" invisible="1"/>
                <field name="postcode"/>
                <field name="bedrooms"/>
                <field name="living_area" string='Living Area (sqm)'/>
                <field name="expected_price"/>
                <field name="selling_price"/>
                <field name="date_availability" string='Available From' optional="hide"/>
                <field name="property_type_id"/>
                <field name="tag_ids" widget="many2many_tags"/>
            </tree>
        </field>
    </record>
```

b. Add a custom form view:

Define a form view for the estate.property model in the appropriate XML file.

Add fields and tags incrementally for better understanding.

Use the --dev xml parameter for viewing modifications without server relaunch.

estate_property_views.xml U

odoo-sh > estate > views > estate_property_views.xml

```xml
            </field>
        </record>
        <record id="view_real_estate_property_form" model="ir.ui.view">
        <field name="name">real_estate.property.form</field>
        <field name="model">real_estate</field>
<field name="arch" type="xml">
            <form>
                <header>
                    <button name="sold_action" type="object" string="Sold" invisible="state in ['sold','canceled']"/>
                    <button name="cancel_action" type="object" string="Cancel" invisible="state in ['sold','canceled']"/>
                    <field name="state" widget="statusbar" statusbar_visible="new,offer_received,offer_accepted,sold"/>
                </header>
                <sheet>
                <div class="oe_title">
                        <h1>
                            <field name="name" placeholder='Title'/>
                        </h1>
                        <field name="tag_ids" widget="many2many_tags" options="{'color_field': 'color'}"/>
                    </div>
                    <group>
                        <group>
                            <field name="state" string='Status'/>
                            <field name="property_type_id" string='Property Type' options="{'no_create': True}"/>
                            <field name="postcode"/>
                            <field name="date_availability" string='Available From'/>
                        </group>
                        <group>
                            <field name="expected_price"/>
                            <field name="best_price"/>
                            <field name="selling_price"/>
                        </group>
                        <notebook>
                            <page string="Description">
                                <group>
                                    <group>
                                        <field name="description"/>
                                        <field name="bedrooms"/>
                                        <field name="living_area" string='Living Area (sqm)'/>
                                        <field name="facades"/>
                                        <field name="garage"/>
                                        <field name="garden"/>
                                        <field name="garden_area" string='Garden Area (sqm)' invisible="not garden"/>
                                        <field name="garden_orientation" invisible="not garden"/>
                                        <field name="total_area" string='Total Area (sqm)'/>
                                    </group>
                                </group>
                            </page>
                            <page string="Offers">
                                <field name="offer_ids" widget="one2many_list" readonly="state in ['accepted', 'sold', 'canceled']"/>
                            </page>
                            <page string="Other Info">
                                <group>
                                    <field name="seller_id"/>
                                    <field name="buyer_id"/>
                                </group>
                            </page>
                        </notebook>
                        </group>
                </sheet>
            </form>
        </field>
    </record>
```

c. Add a custom search view:

Define a search view for the estate.property model in the appropriate XML file.

After restarting the server, filtering on the specified fields should be possible.

d. Add filter and Group By:

Enhance the previously created search view with:

- ✓ A filter for available properties (state is "New" or "Offer Received").
- ✓ The ability to group results by postcode.

Check that the modifications achieve the desired functionality.

```
<record id="view_real_estate_property_search" model="ir.ui.view">
        <field name="name">real.estate.property.search</field>
        <field name="model">real_estate</field>
        <field name="arch" type="xml">
            <search string="Search Real Estate Properties">
                <field name="name" string='Title'/>
                <field name="postcode"/>
                <field name="expected_price"/>
                <field name="bedrooms"/>
                <field name="living_area" string='Living Area (sqm)' filter_domain="[('living_area', '&gt;', self)]"/>
                 <field name="facades"/>
                 <field name="property_type_id"/>
                <filter name="filter_available" string="Available" domain="[('state', 'in', ['new','received'])]"/>
                <filter name="Postcode" context="{'group_by': 'postcode'}"/>
            </search>
        </field>
</record>
```

Chapter 8: Relations Between Models

1. Overview:

This chapter delves into establishing relationships between models in Odoo, introducing the concepts of Many2one, Many2many, and One2many fields. The Real Estate module is used as a practical example, guiding developers through the creation of related models, menus, actions, and views.

2. Many2one:

The chapter begins by introducing the Many2one concept, which represents a simple link to another object. Developers are tasked with creating a new model, estate.property.type, with a Many2one field, property_type_id, added to the existing estate.property model. This exercise involves creating menus, actions, and views for the new model.

3. Exercise:

Create the estate.property.type model.

estate_property_type.py U

odoo-sh > estate > models > estate_property_type.py > ...

```python
from odoo import models, fields


class EstatePropertyType(models.Model):
    _name = 'estate.property.type'
    _description = 'Real Estate Property Type'
    _order = "sequence, name"

    name = fields.Char(required=True)
    sequence = fields.Integer(default=10)
    _sql_constraints = [
        ('check_name', 'unique (name)',
         'A property type name must be unique.')
    ]
    property_ids = fields.One2many(
        'real_estate', 'property_type_id')

```

Add a Many2one field, property_type_id, to the estate.property model.

Update the form, tree, and search views for the estate.property model.

estate_property.py U

odoo-sh > estate > models > estate_property.py > RealEstate

```python
from odoo import models, fields, api
from odoo.exceptions import UserError
from odoo.tools.float_utils import float_compare, float_is_zero


class RealEstate(models.Model):
    _name = "real_estate"  # The ORM will replace . by _ for the table name
    _description = "Estate Property"
    _order = "id desc"

    name = fields.Char(required=True)
    description = fields.Char()
    postcode = fields.Char()
    date_availability = fields.Date(
        copy=False, default=lambda self: fields.Date.add(fields.Date.today(), months=3))
    expected_price = fields.Float(required=True)
    selling_price = fields.Float(copy=False, readonly=True)
    bedrooms = fields.Integer(default=2)
    living_area = fields.Integer()
    facades = fields.Integer()
    garage = fields.Boolean()
    garden = fields.Boolean()
    garden_area = fields.Integer()
    garden_orientation = fields.Selection(
        [('north', 'North'), ('south', 'South'),
         ('east', 'East'), ('west', 'West')],
    )
    active = fields.Boolean(default=True)
    state = fields.Selection([
        ('new', 'New'),
        ('offer_received', 'Offer Received'),
        ('offer_accepted', 'Offer Accepted'),
        ('sold', 'Sold'),
        ('canceled', 'Canceled'),
    ], required=True, default='new', copy=False)
    property_type_id = fields.Many2one('estate.property.type')
    buyer_id = fields.Many2one('res.partner', string='Buyer', copy=False)
    seller_id = fields.Many2one(
        'res.users', string='Salesman', default=lambda self: self.env.user)
    tag_ids = fields.Many2many('estate.property.tag', string='Tags')
    offer_ids = fields.One2many(
        'estate.property.offer', 'property_id', string='Offers')
    total_area = fields.Float(
        string='Total Area', compute='_compute_total_area')

    best_price = fields.Float(
        string='Best Offer Price', compute='_compute_best_price')
```

4. Many2many:

Next, the Many2many concept is introduced, representing a bidirectional multiple relationship. Developers are tasked with creating a new model, estate.property.tag, with a Many2many field, tag_ids, added to the existing estate.property model. This exercise involves creating menus, actions, and views for the new model, with the introduction of the widget="many2many_tags" attribute in the view.

5. Exercise:

Create the estate.property.tag model.

estate_property_tag.py U

odoo-sh > estate > models > estate_property_tag.py > ...

```python
from odoo import models, fields


class EstatePropertyTag(models.Model):
    _name = 'estate.property.tag'
    _description = 'Real Estate Property Tag'
    _order = "name"

    name = fields.Char(required=True)
    color = fields.Integer()
    _sql_constraints = [
        ('check_name', 'unique (name)',
         'A property tag name must be unique.')
    ]

```

- ✓ Add a Many2many field, tag_ids, to the estate.property model (See Above).
- ✓ Update the form and tree views for the estate.property model, using the widget="many2many_tags" attribute (See Above).

6. One2many:

The chapter concludes with the introduction of the One2many concept, which represents an inverse relationship of Many2one. Developers are tasked with creating a new model, estate.property.offer, with a One2many field, offer_ids, added to the existing estate.property model. This exercise involves creating fields for the new model and updating the form view for the estate.property model(done in the first example).

7. Exercise:

Create the estate.property.offer model.

estate_property_offer.py U

odoo-sh > estate > models > estate_property_offer.py > ...

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


class EstatePropertyOffer(models.Model):
    _name = 'estate.property.offer'
    _description = 'Real Estate Property Offer'
    _order = "price desc"

    price = fields.Float()
    status = fields.Selection([
        ('accepted', 'Accepted'),
        ('refused', 'Refused'),
    ], copy=False)
    validity = fields.Integer(default=7)
    date_deadline = fields.Date(
        string='Validity Date', compute='_compute_date_deadline', inverse='_inverse_date_deadline')
    partner_id = fields.Many2one('res.partner', required=True)
    property_id = fields.Many2one('real_estate', required=True)

```

- ✓ Add a One2many field, offer_ids, to the estate.property model.
- ✓ Create fields for the estate.property.offer model (price, status, partner_id, property_id).
- ✓ Update the form view for the estate.property model. (Done in the first example)

Chapter 9: Computed Fields And Onchanges

1. Overview:

This chapter explores the concepts of computed fields and onchanges in Odoo, providing developers with tools to dynamically compute field values and enhance the user interface by responding to changes in form fields.

2. Computed Fields:

The chapter begins by introducing computed fields, explaining that they are fields whose **values are not directly retrieved from the database** but are computed on-the-fly by calling a method of the model. Developers are guided through creating computed fields by defining the computation method and specifying dependencies using the @api.depends decorator.

3. Exercise:

a. Compute the total area:

- ✓ Add the total_area field to the estate.property model, computed as the sum of living_area and garden_area.
- ✓ Update the form view for the estate.property model.

b. Compute the best offer:

- ✓ Add the best_price field to the estate.property model, computed as the maximum of the offers' prices.
- ✓ Update the form view for the estate.property model.

```python
@api.depends('offer_ids.price')
def _compute_best_price(self):
    for record in self:
        prices = record.offer_ids.mapped('price')
    if prices:
        record.best_price = max(prices)
    else:
        record.best_price = 0

@api.depends('living_area', 'garden_area')
def _compute_total_area(self):
    for record in self:
        record.total_area = record.living_area + record.garden_area
```

4. Exercise:

a. Compute a validity date for offers:

- ✓ Add fields (validity and date_deadline) to the estate.property.offer model.
- ✓ Define date_deadline as a computed field, the sum of create_date and validity, with an appropriate inverse function.
- ✓ Update the form and list views for the estate.property.offer model.

Additional Information on Computed Fields:

Computed fields are not stored in the database by default, and searching on them requires defining a search method or using store=True with performance considerations.

It is mentioned that computed fields can depend on other computed fields, but performance should be considered.

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


class EstatePropertyOffer(models.Model):
    _name = 'estate.property.offer'
    _description = 'Real Estate Property Offer'
    _order = "price desc"

    price = fields.Float()
    status = fields.Selection([
        ('accepted', 'Accepted'),
        ('refused', 'Refused'),
    ], copy=False)
    validity = fields.Integer(default=7)
    date_deadline = fields.Date(
        string='Validity Date', compute='_compute_date_deadline', inverse='_inverse_date_deadline')
    partner_id = fields.Many2one('res.partner', required=True)
    property_id = fields.Many2one('real_estate', required=True)

    @api.depends('create_date', 'validity')
    def _compute_date_deadline(self):
        for record in self:
            record.date_deadline = fields.Date.add(
                record.create_date or fields.Date.today(), days=record.validity)

    def _inverse_date_deadline(self):
        for record in self:
            if record.create_date and record.date_deadline:
                record.validity = (
                    record.date_deadline - fields.Date.to_date(record.create_date or fields.Date.today())).days
```

5. Onchanges:

The chapter moves on to onchanges, which allow the client interface to dynamically update a form without saving to the database when a user fills in a field. Developers learn to define onchange methods, triggered by specific fields, and update other fields within the form.

6. Exercise:

a. Set values for garden area and orientation:

- ✓ Create an onchange in the estate.property model to set values for garden_area (10) and orientation (North) when garden is set to True. Clear the fields when unset.

Additional Information on Onchanges:

Onchange methods can return non-blocking warning messages.

```python
@api.onchange("garden")
def _onchange_garden(self):
    if self.garden:
        self.garden_area = 10
        self.garden_orientation = 'north'
    else:
        self.garden_area = 0
        self.garden_orientation = False
```

Chapter 10: Ready for Some Action?

1. Overview:

Chapter 10 introduces the concept of action buttons in Odoo, enabling developers to link business logic to user interface interactions. The primary goal is to allow users to perform actions like canceling or setting a property as sold and accepting or refusing an offer. The chapter covers the use of action buttons in views and demonstrates how to associate these buttons with Python methods in the model.

2. Action Type:

Developers learn how to add buttons in views and link them to specific business logic using the type="object" attribute. The example illustrates the creation of a button in the view and associating it with a Python method (action_do_something) in the model.

3. Exercise:

a. Cancel and set a property as sold

- ✓ Add "Cancel" and "Sold" buttons to the estate.property model.
- ✓ Implement logic to prevent a canceled property from being set as sold, and vice versa.

```python
def sold_action(self):
    for offer in self:
        if offer.state == 'canceled':
            raise UserError("Cannot mark a canceled property as sold.")
        offer.state = "sold"
    return True

def cancel_action(self):
    for offer in self:
        if offer.state == 'sold':
            raise UserError("Cannot cancel a sold property.")
        offer.state = "canceled"
    return True
```

4. Exercise:

a. Accept or refuse an offer

- ✓ Add "Accept" and "Refuse" buttons to the estate.property.offer model.
- ✓ Implement logic to handle the acceptance of an offer, setting the buyer and selling price for the corresponding property.

```python
def action_accept(self):
    for offer in self:
        if "accepted" in (offer.property_id.offer_ids - offer).mapped("status"):
            raise UserError(
                "You cannot accept an offer if another is already been accepted.")
        offer.status = "accepted"
        offer.property_id.buyer_id = offer.partner_id
        offer.property_id.selling_price = offer.price
    return True

def action_refuse(self):
    for offer in self:
        offer.status = "refused"
    return True
```

Chapter 11: Constraints

1. Overview:

In the previous chapter, we introduced the ability to add business logic to our model, allowing us to link buttons to business code. However, ensuring data integrity is crucial. In this chapter, we'll explore two ways to establish automatically verified invariants in Odoo: Python constraints and SQL constraints.

2. SQL Constraints:

SQL constraints are implemented through the model attribute _sql_constraints. This attribute is assigned a list of triples containing strings (name, sql_definition, message). Here, name is a valid SQL constraint name, sql_definition is a table_constraint expression, and message is the error message.

3. Exercise:

Add the following constraints to their corresponding models:

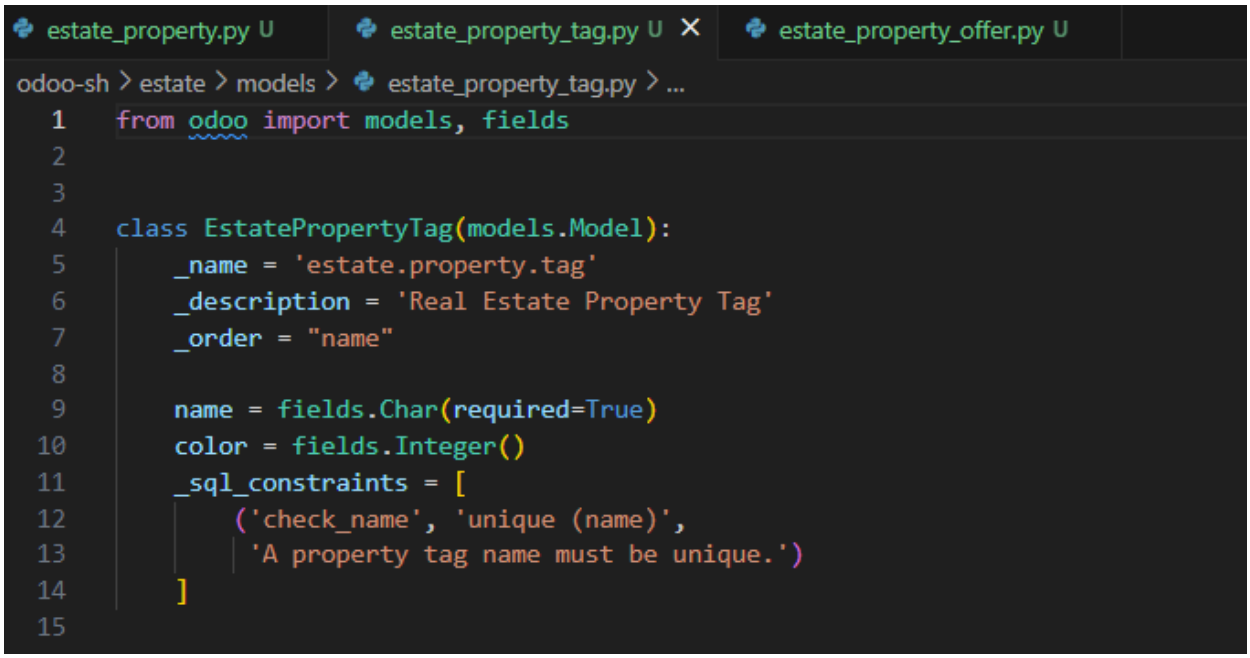
- ✓ A property expected price must be strictly positive.
- ✓ A property selling price must be positive.

```
_sql_constraints = [
    ('check_expected_price', 'CHECK(expected_price > 0)',
     'A property expected price must be strictly positive.'),
    ('check_selling_price', 'CHECK(selling_price >= 0)',
     'A property selling price must be positive.')
]
```

- ✓ An offer price must be strictly positive.

```
_sql_constraints = [
    ('check_price', 'CHECK(price > 0)',
     'An offer price must be strictly positive.')
]
```

- ✓ A property tag name and property type name must be unique.



Restart the server with the -u estate option to see the result.

4. Python constraint:

While SQL constraints efficiently ensure data consistency, there are cases where more complex checks, requiring Python code, are necessary. In this case, we use a Python constraint.

A Python constraint is defined as a method decorated with @api.constrains('field_name') and is invoked on a recordset. The decorator specifies which fields are involved in the constraint. The constraint is automatically evaluated when any of these fields are modified. The method is expected to raise an exception if its invariant is not satisfied.

5. Exercise:

Add Python constraints.

- ✓ Add a constraint so that the selling price cannot be lower than 90% of the expected price.
- ✓ The selling price is zero until an offer is validated. Fine-tune your check to take this into account.
- ✓ Always use the float_compare() and float_is_zero() methods from odoo.tools.float_utils when working with floats!
- ✓ Ensure the constraint is triggered every time the selling price or the expected price is changed!

- ✓ SQL constraints are usually more efficient than Python constraints. When performance matters, always prefer SQL over Python constraints.

```python
@api.constrains('selling_price', 'expected_price')
def _check_selling_price(self):
    for record in self:
        if (
            not float_is_zero(record.selling_price, precision_digits=2) and
                float_compare(record.selling_price, record.expected_price * 0.9, precision_digits=2) == -1):
            raise UserError(
                "The selling price cannot be lower than 90% of the expected price")
```