Odoo ERP Documentation Examples and Explanations

Chapter 1: Architecture Overview

- **Odoo is a multitier application**, meaning that it is composed of several modules. Each module is responsible for a specific part of the application. For example, the `base` module is responsible for the core of the application, while the `sale` module is responsible for sales.
- **Odoo is composed of 3 main layers:**
    - **The Data Layer**: The data layer is the main entry point to the database. It is responsible for the creation, modification, and deletion of data.
    - **The Logic Layer**: The Logic Layer coordinates the application, process commands, and performs calculations. It also is responsible for the creation, modification, and deletion of data.
    - **The User Interface Layer**: The User Interface Layer is responsible for the display of data. It translates the data into a format that is easy to read and easy to use.

Chapter 2: Developer Environment setup

- Odoo's installer includes all the required dependencies to run the application and its accompanying modules and services, including but not limited to

- Python 3.7
- PostgreSQL
- Nginx
- Odoo
- Odoo Server
- Odoo Client
- Odoo Server Admin
- Odoo Server Worker
- Odoo Server Web
- Odoo's installer also includes a configuration file, `odoo.conf`, that contains the necessary information to run the application.

Chapter 3: Creating a new application/module

> **- To create a new application, you must first create a new directory at C:\Program Files\odoo15\server\odoo\addons**
>
> **- Then, you must create two new files in the new directory, and name them `__init__.py` and `__manifest__.py` accordingly**

> **- The `__init__.py` file is required to call the base module, and the `__manifest__.py` file is required to contain the following information**
>
> - **name**: The name of the module.
> - **version**: The version of the module.
> - **author**: The author of the module.
> - **description**: The description of the module.
> - **depends**: The modules that the module depends on.
> - **category**: The category of the module.
> - **sequence**: The sequence of the module.
> - **summary**: The summary of the module.
> - **data**: The data files that the module contains.
> - **application**: The application that the module contains.
> - **installable**: Whether the module is installable or not.
> - **auto_install**: Whether the module is auto-installable or not.

> Or you can do the following command in the terminal to create the basic skeleton for module creation
>
> ```
> > cd C:\Program Files\odoo15\server\python
>
> > python.exe "C:\Program Files\odoo15\server\odoo" scaffold <module_name> "C:\Program Files\odoo15\server\odoo\addons"
> ```

> ***note: When changing functions in the model it is more than enough to restart the Odoo service, but if you were editing the model data structure thereby editing the database it creates, then you must uninstall and reinstall the module since the previous records are now incompatible with the new model fields where it has null where the new columns have been added.***
>
> ***Editing XML views can be seen after a simple upgrade to the already installed app module***

Chapter 4: Models and Basic Fields

- Models are the main data structure in Odoo. They are used to create and store data in tables in the database.

  > ***note: Odoo uses PostgreSQL for its database due to it being a relational database***

  > - Models are created by creating a new file in the `models` directory.
  > - **The `models` directory must include an `__init__.py` file.**
  > - Each model has model fields that are the actual columns in the database table.
  > - Each field has a type, which is a string that corresponds to the type of data that is stored in the field. For example, the `char` type is used to store strings, the `integer` type is used to store integers, and the `boolean` type is used to store booleans. These types directly correspond to SQL's data types.

Start off by importing from Odoo the required dependencies

```
> from odoo import exceptions, api, fields, models
```

> ***The `exceptions` module is used to throw exceptions.***

> ***The `api` module is used to call Odoo's API. The `fields` module is used to create model fields.***

> ***The `models` module is used to create models.***

Creating the module

Create the Class for the Model

```
class <model_name>(models.Model):
```

Add the fields to the model

```
_name = '<model_name>'>
_description = '<model_description>'>
_inherit = '<model_inherit>'>
_order = '<model_order>'>
_sql_constraints = [('<model_constraint_name>',
'<model_constraint_type>', '<model_constraint_message>')]> _inherits =
{'<model_inherit_name>': '<model_inherit_field>'}>
```

note: The `_inherit` attribute is used to inherit fields from another model.

Also, `inherit` results in a different config than `inherits`.

Create the data fields depending on your requirements and needs, but the general format is

```
<name_of_field> = fields.<type_of_field>(<field_attributes>)
```

Types of fields include but are not limited to

- **fields.Char**: Stores strings.
- **fields.Text**: Stores strings.
- **fields.Integer**: Stores integers.
- **fields.Float**: Stores floats.
- **fields.Boolean**: Stores booleans.
- **fields.Date**: Stores dates.
- **fields.Datetime**: Stores datetimes.
- **fields.Time**: Stores times.
- **fields.Binary**: Stores binary data.
- **fields.Many2one**: Stores a many-to-one relationship.
- **fields.One2many**: Stores a one-to-many relationship.
- **fields.Many2many**: Stores a many-to-many relationship.
- **fields.Reference**: Stores a reference to another model.

- **fields.Selection**: Stores a selection of values.

> ***note:*** *`fields.Many2One` **require a corresponding** `fields.One2Many` **field in the other model and vice versa.***

Attributes for each field include but are not limited to

- ***default*** = "(default value)" # default value for the field
- ***string*** = "(String)" # name of the field that will be visible to user
- ***help*** = "(String)" # help message for the field when it is displayed
- ***readonly*** = "(Boolean)" # whether the field is readonly or not
- ***required*** = "(Boolean)" # whether the field can be null or not
- ***selection*** = "([Array])" # selection of values for the field
- ***copy*** = "(Boolean)" # whether the field is copied or not
- ***compute*** = "(lambda self: expression)" # compute function for the field
- ***store*** = "(Boolean)" # whether the field is stored in the database or not

---

Chapter 5: Security - A Brief Introduction

Odoo supports securing your module using CSV files that include the security rules for each user and group for the current module.

- The CSV file named `ir.model.access.csv` is stored in the `security` directory.

here is a sample of the CSV file for `test.model`:

```
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,1,1,1,1
```

-`id` is an external identifier.

- `name` is the name of the `ir.model.access`.

- `model_id/id` refers to the model which the access right applies to. The standard way to refer to the model is `model_<model_name>`, where `<model_name>` is the `_name` of the model with the . replaced by _.

- `group_id/id` refers to the group which the access right applies to. We will cover the concept of groups in the advanced topic dedicated to the security.

- `perm_read`,`perm_write`,`perm_create`,`perm_unlink`: read, write, create and unlink permissions

---

Chapter 6: Finally - Some UI To Play With

Now that we've created our new model and its corresponding access rights, it is time to interact with the user interface.

> **- The `view` directory contains the XML files that define the user interface.**

> **- The XML file for the module is named `<module_name>_menus.xml`.**

> **A basic action for our test.model is:**

```
<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>```
```

- **id** is an external identifier. It can be used to refer to the record (without knowing its in-database identifier).
- **model** has a fixed value of **ir.actions.act_window** (Window Actions (ir.actions.act_window)).
- **name** is the name of the action.
- **res_model** is the model which the action applies to.
- **view_mode** are the views that will be available; in this case they are the list (tree) and form views. We'll see later that there can be other view modes.

> **A basic menu for our test_model_action is:**
>
> `<menuitem id="test_model_menu_action" action="test_model_action"/>` The menu `test_model_menu_action` is linked to the `action test_model_action`, and the action is linked to the model `test.model`. As previously mentioned, the action can be seen as the link between the menu and the model.

The easiest way to define the structure is to create it in the XML file. A basic structure for our test_model_action is:

```
<menuitem id="test_menu_root" name="Test">
    <menuitem id="test_first_level_menu" name="First Level">
        <menuitem id="test_model_menu_action" action="test_model_action"/>
    </menuitem>
</menuitem>
```

Chapter 7: Basic Views

Odoo provides defualt views for each model, but those views are never acceptable for a business application. Instead, we need to define our own views that are more organized.

- The XML file for the model is named `<model_name>_views.xml`**.**

Supported views include: **kanban**, **list**, **form**, **graph**, **pivot**, **cohort**, **dashboard**.

The views that are supported include but are not limited to:

- List View:

- List views, also called tree views, display records in a tabular form.
- Their root element is `<tree>`. The most basic version of this view simply lists all the fields to display in the table (where each field is a column):

```
<tree>
    <field name="id"/>
    <field name="name"/>
    <field name="description"/>
</tree>
```

- Form View:

- Form views display a single record in a single page called a form. The root element is . The most basic version of this view simply lists all the fields to display in the form:

```
<form string="Test">
   <sheet>
    <group>
    <group>
              <field name="name"/>
       </group>
        <group>
            <field name="last_seen"/>
        </group>
        <notebook>
            <page string="Description">
                <field name="description"/>
            </page>
         </notebook>
      </group>
  </sheet>
</form>
```

note: It is possible to use regular HTML tags such as `div` **and** `h1` **as well as the the class attribute (Odoo provides some built-in classes) to fine-tune the look.**

- Search View:

- Search views are slightly different from the list and form views since they don't display content. Although they apply to a specific model, they are used to filter other views' content (generally aggregated views such as List). Beyond the difference in use case, they are defined the same way.
- Their root element is `<search>`. The most basic version of this view simply lists all the fields for which a shortcut is desired:

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

- **Search views can also contain `<filter>` elements, which act as toggles for predefined searches. Filters must have one of the following attributes:**

domain: adds the given domain to the current search.

For instance, when used on the Product model the following domain selects all services with a unit price greater than 1000:

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

note: By default criteria are combined with an implicit AND.

For instance, to select products 'which are services OR have a unit price which is NOT between 1000 and 2000':

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

context: adds some context to the current search; uses the `key group_by` to group results on the given field name.

Chapter 8: Relations Between Models

Relations between models are used to link records from different models together in a way similar to links between database tables using foreign keys such as:

- **Many2one Links:**

  A many2one link is a link between a record and another record that results in a many to one relationship between two tables:

  - Basic form: `partner_id = fields.Many2one("res.partner", string="Partner")`

    > Here, we defined a many to one link from the res.partner model using partner_id as the link or the pathway to that model.

  note: By convention, many2one fields have the _id suffix.

  Accessing the data can be done with the `partner_id.name` attribute.

- **One2many Links:**

  A one2many link is a link between a record and a collection of records that results in a one to many relationship between two tables:

  - Basic form: `partner_ids = fields.One2many("res.partner", "partner_id", string="Partners")`

    > Here, we defined a one2many link from the res.partner model using partner_ids as the link or the pathway to that model.

  note: By convention, one2many fields have the _ids suffix.

  Accessing the data can be done with the `partner_ids.name` attribute.

  Warning: Because a `One2many` is a virtual relationship, there must be a `Many2one` field defined in the comodel.

- **Many2many Links:**

  A many2many link is a link between a record and a collection of records that results in a many to many relationship between two tables:

  - Basic form: `partner_ids = fields.Many2many("res.partner", string="Partners")`

    > Here, we defined a many2many link from the res.partner model using partner_ids as the link or the pathway to that model.

  note: By convention, many2many fields have the _ids suffix.

  Accessing the data can be done with the `partner_ids.name` attribute.

Chapter 9: Computed Fields And Onchanges

Computed fields are fields that are automatically computed based on other fields. Computed fields are defined in the XML file using the `compute` attribute.

> ***note: Computed fields are not stored in the database. but can be set to do so.***

> **note: By convention, compute methods are private, meaning that they cannot be called from the presentation tier, only from the business tier. Private methods have a name starting with an underscore _.**

> Sample Model with defined computed fields:

```
from odoo import api, fields, models

class TestComputed(models.Model):
    _name = "test.computed"

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

    @api.depends("amount")
    def _compute_total(self):
        for record in self:
            record.total = 2.0 * record.amount
```

> **note: `self` is a collection.**
>
> **The object `self` is a recordset, i.e. an ordered collection of records. It supports the standard Python operations on collections, e.g. `len(self)` and `iter(self)`, plus extra set operations such as `recs1 | recs2`.**
>
> **Iterating over self gives the records one by one, where each record is itself a collection of size 1. You can access/assign fields on single records by using the dot notation, e.g. `record.name`.**

> **For relational fields it's possible to use paths through a field as a dependency:**

```
description = fields.Char(compute="_compute_description")
partner_id = fields.Many2one("res.partner")

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

> **The example is given with a Many2one, but it is valid for Many2many or a One2many.**

Inverse Computed Functions:

In some cases, it might be useful to still be able to set a value directly. In our real estate example, we can define a validity duration for an offer and set a validity date. We would like to be able to set either the duration or the date with one impacting the other.

To support this Odoo provides the ability to use an inverse function:

```
from odoo import api, fields, models

class TestComputed(models.Model):
   _name = "test.computed"

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

   @api.depends("amount")
   def _compute_total(self):
       for record in self:
           record.total = 2.0 * record.amount

   def _inverse_total(self):
       for record in self:
           record.amount = record.total / 2.0
```

OnChanges:

OnChanges are triggered when a field is modified.

```
from odoo import api, fields, models
class TestOnchange(models.Model):
   _name = "test.onchange"

   name = fields.Char(string="Name")
   description = fields.Char(string="Description")
   partner_id = fields.Many2one("res.partner", string="Partner")

   @api.onchange("partner_id")
   def _onchange_partner_id(self):
       self.name = "Document for %s" % (self.partner_id.name)
       self.description = "Default description for %s" %
(self.partner_id.name)
```

Chapter 10: Ready For Some Action?

Action Type:

To link logic to buttons we use the `action` attribute.

```
<form>
   <header>
       <button name="action_do_something" type="object" string="Do
Something"/>
   </header>
   <sheet>
       <field name="name"/>
   </sheet>
</form>
```

And then we define the action in the `action` attribute:

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

name = fields.Char()

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

By assigning `type="object"` to our button, the Odoo framework will execute a Python method with `name="action_do_something"` on the given model.

Object Type:

You may be wondering if it is possible to link an action to a button. Good news, it is! One way to do it is:

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

We use `type="action"` and we refer to the external identifier in the `name`.

Chapter 11: Constraints:

Odoo provides two ways to set up automatically verified invariants:

- Python constraints
- SQL constraints

SQL Constraints:

SQL constraints are defined through the model attribute _sql_constraints. This attribute is assigned a list of triples containing strings in the following form:

```
(name, sql_definition, message)
```

And a sample SQL Constraint definition is:

```
_sql_constraints = [('<model_constraint_name>', '<model_constraint_type>',
'<model_constraint_message>')]> _inherits = {'<model_inherit_name>':
'<model_inherit_field>'}>
```

Python Constraints:

SQL constraints are an efficient way of ensuring data consistency. However it may be necessary to make more complex checks which require Python code. In this case we need a Python constraint.

The method that is decorated with `constraints()` is expected to raise an exception if its invariant is not satisfied:

```
from odoo.exceptions import ValidationError

@api.constrains('date_end')
def _check_date_end(self):
   for record in self:
       if record.date_end < fields.Date.today():
           raise ValidationError("The end date cannot be set in the past")
   # all records passed the test, don't return anything
```

Chapter 12: Add The Sprinkles

This chapter covers a very small subset of what can be done in the views. Do not hesitate to read the reference documentation for a more complete overview.

Inline Views:

In the real estate module we added a list of offers for a property. We simply added the field `offer_ids` with:

```
<field name="offer_ids"/>
```

The field uses the specific view for `estate.property.offer`.

We would like to display the list of properties linked to a property type. However, we only want to display 3 fields for clarity: name, expected price and state.

To do this, we can define inline list views. An inline list view is defined directly inside a form view. For example:

```
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()
```

And to follow that in the XML View as:

```
<form>
   <field name="description"/>
   <field name="line_ids">
       <tree>
           <field name="field_1"/>
           <field name="field_2"/>
       </tree>
   </field>
</form>
```

In the form view of the `test.model`, we define a specific list view for `test.model.line` with fields `field_1` and `field_2`.

Widgets:

In some cases, we want a specific representation of a field which can be done thanks to the widget attribute:

```
<field name="state" widget="statusbar"
statusbar_visible="open,posted,confirm"/>
```

List Order:

Example:

```
_order = "id desc"
```

Manual Order:

To do so, a `sequence` field is used in combination with the `handle` widget. Obviously the `sequence` field must be the first field in the `_order` attribute.

Field Attributes and Options:

Example:

```
<form>
   <field name="description" attrs="{'invisible': [('is_partner', '=',
False)]}"/>
   <field name="is_partner" invisible="1"/>
</form>
```

List Decoration:

Example:

```
<tree decoration-success="is_partner==True">
   <field name="name">
   <field name="is_partner" invisible="1">
</tree>
```

Search Attributes:

Example:

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

Stat Buttons:

The easiest way to understand it is to consider it as a specific case of a computed field. The following definition of the description field:

```
...

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

is equivalent to:

```
...

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
```

Chapter 13: Inheritance

A powerful aspect of Odoo is its modularity. A module is dedicated to a business need, but modules can also interact with one another. This is useful for extending the functionality of an existing module.

Python Inheritance

The Odoo framework provides the necessary tools to do them. In fact, such actions are already included in our model thanks to classical Python inheritance:

```
from odoo import fields, models

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

   ...
```

Our class `TestModel` inherits from `Model` which provides `create()`, `read()`, `write()` and `unlink()`.

These methods (and any other method defined on `Model`) can be extended to add specific business logic:

```
from odoo import fields, models

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

   ...

   @api.model
   def create(self, vals):
       # Do some business logic, modify vals...
       ...
```

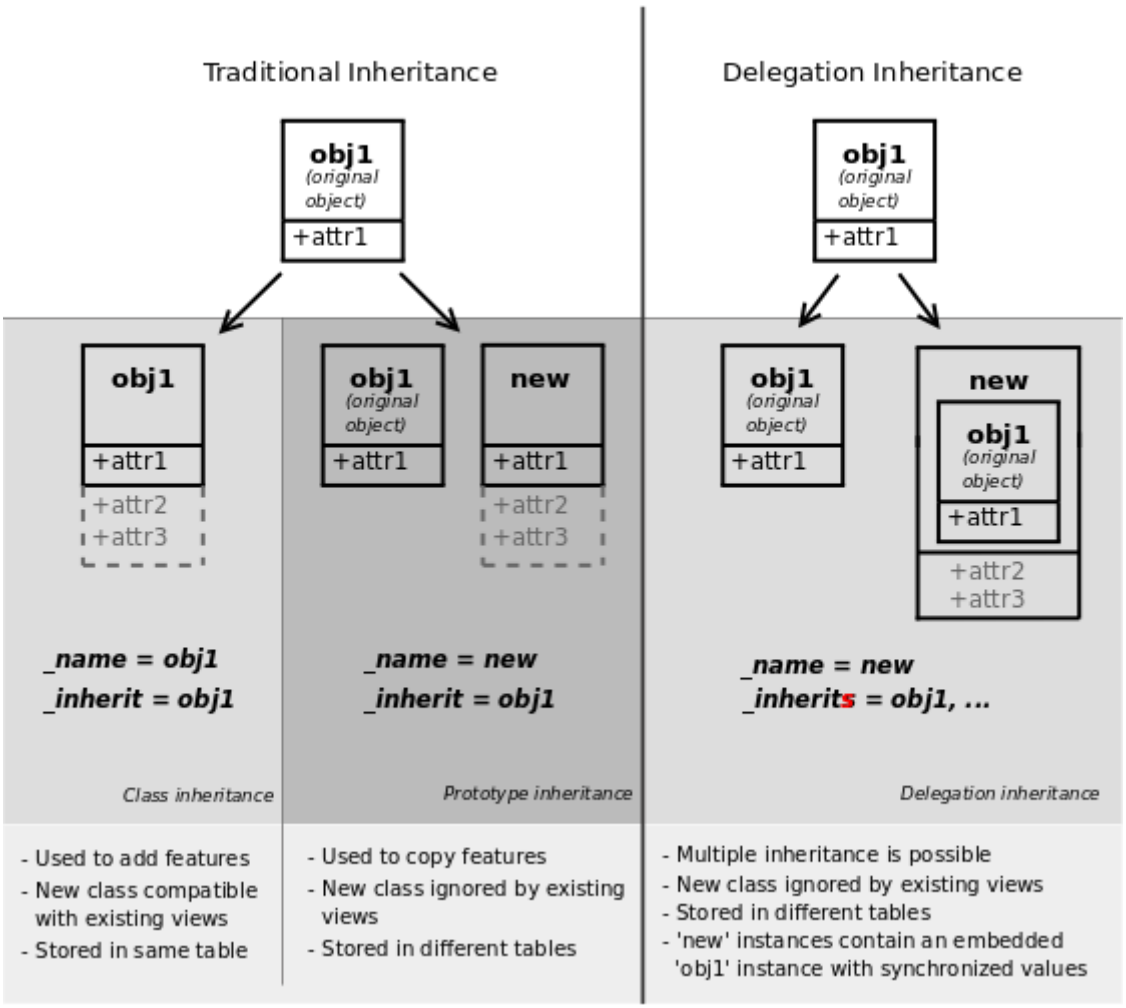
```
        # Then call super to execute the parent method
        return super().create(vals)
```

The decorator `model()` is necessary for the `create()` method because the content of the recordset `self` is not relevant in the context of creation, but it is not necessary for the other CRUD methods.

note: even though we can directly override the `unlink()` method, you will almost always want to write a new method with the decorator `ondelete()` instead. Methods marked with this decorator will be called during `unlink()` and avoids some issues that can occur during uninstalling the model's module when `unlink()` is directly overridden.

> **It is very important to always call super() to avoid breaking the flow. There are only a few very specific cases where you don't want to call it.**
>
> **Make sure to always return data consistent with the parent method. For example, if the parent method returns a dict(), your override must also return a dict().**

Model Inheritance



In Odoo, the first mechanism is by far the most used. In our case, we want to add a field to an existing model, which means we will use the first mechanism. For example:

```
from odoo import fields, models
```

```
class InheritedModel(models.Model):
   _inherit = "inherited.model"

   new_field = fields.Char(string="New Field")
```

note: the `_inherit` attribute is the name of the model to inherit from. It can be a string or a list of strings.

note: By convention, each inherited model is defined in its own Python file. In our example, it would be `models/inherited_model.py`.

View Inheritance

Instead of modifying existing views in place (by overwriting them), Odoo provides view inheritance where children 'extension' views are applied on top of root views. These extension can both add and remove content from their parent view.

An extension view references its parent using the `inherit_id` field. Instead of a single view, its `arch` field contains a number of `xpath` elements that select and alter the content of their parent view:

```
<record id="inherited_model_view_form" model="ir.ui.view">
   <field name="name">inherited.model.form.inherit.test</field>
   <field name="model">inherited.model</field>
   <field name="inherit_id" ref="inherited.inherited_model_view_form"/>
   <field name="arch" type="xml">
       <!-- find field description and add the field
            new_field after it -->
       <xpath expr="//field[@name='description']" position="after">
         <field name="new_field"/>
       </xpath>
   </field>
</record>
```

- **expr** An XPath expression selecting a single element in the parent view. Raises an error if it matches no element or more than one
- **position** Operation to apply to the matched element:
- **inside** appends xpath's body to the end of the matched element
- **replace** replaces the matched element with the xpath's body, replacing any $0 node occurrence in the new body with the original element
- **before** inserts the xpath's body as a sibling before the matched element
- **after** inserts the xpaths's body as a sibling after the matched element
- **attributes** alters the attributes of the matched element using the special attribute elements in the xpath's body

When matching a single element, the position attribute can be set directly on the element to be found. Both inheritances below have the same result.

```
<xpath expr="//field[@name='description']" position="after">
   <field name="idea_ids" />
</xpath>

<field name="description" position="after">
   <field name="idea_ids" />
</field>
```

Chapter 14: Interact With Other Modules

Link Module

The common approach for such use cases is to create a 'link' module. In our case, the module would depend on `estate` and `account` and would include the invoice creation logic of the estate property. This way the real estate and the accounting modules can be installed independently. When both are installed, the link module provides the new feature.

```
from odoo import models

class InheritedModel(models.Model):
   _inherit = "inherited.model"

   def inherited_action(self):
       return super().inherited_action()
```

```
from odoo import Command

def inherited_action(self):
   self.env["test.model"].create(
       {
           "name": "Test",
           "line_ids": [
               Command.create({
                   "field_1": "value_1",
                   "field_2": "value_2",
               })
           ],
       }
   )
   return super().inherited_action()
```

Advanced C: Master and Demo Data

Master Data is technical data that data that must be created on install of the module that is often necessary for the module to work properly. **Demo Data**, on the other hand, is data that is initialized on install and is used for demonstration purposes.

> Demo data is automatically loaded when you start the server if you don't explicitly say you don't want it. This can be done in the database manager or with the command line. `$ ./odoo-bin --addons-path=... -d db -i account --without-demo=all`

For convention purpouses, the name of the demo data folder is `demo` and the name of the master data folder is `data`, and the location and name of each file must be declared in the `__manifest__.xml` file.

Data Entry Methods:

> - **CSV:**
>
> > The easiest way to declare simple data is by using the CSV format. This is however limited in terms of features: use it for long lists of simple models.
> >
> > ```
> > id,field_a,field_b,related_id:id
> > id1,valueA1,valueB1,module.relatedid
> > id2,valueA2,valueB2,module.relatedid```
> > ```
>
> - **XML:**
>
> > The XML format is the most flexible way to declare data. It is also the most complex, but it is also the most powerful.
> >
> > ```
> > <odoo>
> >  <record id="id1" model="tutorial.example">
> >    <field name="field_a">valueA1</field>
> >    <field name="field_b">valueB1</field>
> >  </record>
> >
> > <record id="id2" model="tutorial.example">
> >   <field name="field_a">valueA2</field>
> >   <field name="field_b">valueB2</field>
> > </record>
> > </odoo>```
> > ```
> >
> > - For example, you can create records for the types of properties in the `real-estate` module using **CSV** but creating records for the properties themselves and adding offers to each property must be done using **XML**.
> >
> > **Ref:**

> Related fields can be set using the `ref` key. The value of that key is the `xml_id` of the record you want to link. Remember the `xml_id` is composed of the name of the module where the data is first declared, followed by a dot, followed by the id of the record .
>
> ```
> <odoo>
>   <record id="id1" model="tutorial.example">
>     <field name="related_id" ref="module.relatedid"/>
>   </record>
> </odoo>
> ```
>
> **eval:**
>
> `eval` is used to compute values and it can also be used to optimize the insertion of related values, or because a constraint forces you to add the related values in batch.
>
> ```
> <odoo>
>  <record id="id1" model="tutorial.example">
>    <field name="year" eval="datetime.now().year+1"/>
>  </record>
> </odoo>
> ```
>
> **function:**
>
> You might also need to execute python code when loading data.
>
> ```
> <function model="tutorial.example" name="action_validate">
>    <value eval="[ref('demo_invoice_1')]"/>
> </function>
> ```

QWeb Templates

QWeb is the primary templating engine used by Odoo. It is an XML templating engine and used mostly to generate HTML fragments and pages.

Template directives are specified as XML attributes prefixed with `t-`, for instance `t-if` for Conditionals, with elements and other attributes being rendered directly.

Clearest explanation of QWeb Templates can be found here

https://www.odoo.com/documentation/15.0/developer/reference/frontend/qweb.html

---

Advanced J: PDF Reports

Now we will expand on one of QWeb's other main uses: creating PDF reports. A common business requirement is the ability to create documents to send to customers and to use internally. These reports can be used to summarize and display information in an organized template to support the business in different ways.

For convention purposes, the name of the report folder is `report` and the location and name of each file must be declared in the `__manifest__.xml` file.

Report Templates

We can create templates for the report to specify the format required by the user for the exported report in order to see the result he wants. A Report Template is required to define the fields that will be displayed in the report in the format that the user wants.

Start off with a sample template

```
<?xml version="1.0" encoding="UTF-8" ?>
<odoo>
   <template id="report_property_offers">
       <t t-foreach="docs" t-as="property">
           <t t-call="web.html_container">
               <t t-call="web.external_layout">
                   <div class="page">
                       <h2>
                           <span t-field="property.name"/>
                       </h2>
                       <div>
                           <strong>Expected Price: </strong>
                           <span t-field="property.expected_price"/>
                       </div>
                       <table class="table">
                           <thead>
                               <tr>
                                   <th>Price</th>
                               </tr>
                           </thead>
                           <tbody>
                               <t t-set="offers" t-
value="property.mapped('offer_ids')"/>
                               <tr t-foreach="offers" t-as="offer">
                                   <td>
                                       <span t-field="offer.price"/>
                                   </td>
                               </tr>
                           </tbody>
                       </table>
                   </div>
               </t>
           </t>
       </t>
   </template>
</odoo>
```

The use of `t-set`, `t-value`, `t-foreach` and `t-as` so that we can loop over all of the `offer_ids`.

Report Action

Now that we have a template, we need to make it accessible in our app via a `ir.actions.report`. An `ir.actions.report` is primarily used via the Print menu of a model's view. In the practical example, the `binding_model_id` specifies which model's views the report should show in and Odoo will auto-magically add it for you.

```
 <action model="ir.actions.report" name="report_property_offers"
binding_model_id="tutorial.example"/>
```

Sub-Templates

Sometimes you have to make multiple templates for the same report. For example, you might have a template for the header and footer of the report, and a template for the body of the report. And you might have multiple reports sharing the same header and footer or the same set of data. For that we can utilize sub-templates.

Sub-templates are separate templates that are not callable as a report by themselves but are callable using the function `t-call` to be implemented in an actual report.

A quick Sub-Report I made is this example:

```
    <template id="property_and_its_offers">
        <div class="page">
            <div t-if="who == 'salesman'">
                <h3>
                    <span t-field="property.name" />
                </h3>
            </div>
            <div>
                <strong>Expected Price: </strong>
                <span t-field="property.expected_price" />
                $
            </div>
            <div>
                <strong>Status: </strong>
                <span t-field="property.state" />
            </div>
            <table class="table" t-if="property.property_offer_ids">
                <thead>
                    <tr>
                        <th>Price</th>
                        <th>Partner Name</th>
                        <th>Validity(days)</th>
```

```
                        <th>Offer Deadline</th>
                        <th>State</th>
                    </tr>
                </thead>
                <tbody>
                    <t t-set="offers" t-
value="property.mapped('property_offer_ids')" />
                    <tr t-foreach="offers" t-as="offer">
                        <td>
                            <span t-field="offer.price" />
                        </td>
                        <td>
                            <span t-field="offer.partner_name" />
                        </td>
                        <td>
                            <span t-field="offer.validity" />
                        </td>
                        <td>
                            <span t-field="offer.date_deadline" />
                        </td>
                        <td>
                            <span t-field="offer.status" />
                        </td>
                    </tr>
                </tbody>
            </table>
            <table t-else="">
                <div class="text-center" t-attf-style="display: -
webkit-box; -webkit-box-pack: center; -webkit-box-orient: vertical;">
                    <h4 style="color: gray;">No offers have been made
yet :(</h4>
                </div>
            </table>
        </div>
    </template>
</odoo>
```

Report Inheritance

Inheritance in QWeb uses the same xpath elements as views inheritance. A QWeb template refers to its parent template in a different way though. It is even easier to do by just adding the `inherit_id` attribute to the template element and setting it equal to the `module.parent_template_id`.

> For example, we know that any “Sold” properties will already have an invoice created for them, so we can add this information to our report.
>
> **We can do this by adding the following to our template:**

```
<div t-if="property.state == 'sold'">
    <strong>Invoice has been already created for this property.
```

```
</strong>
</div>
```

Additional Features

- **Translations**

> We all know Odoo is used in multiple languages thanks to automated and manual translating. QWeb reports are no exception!

- **Reports as webpages**

> One of the neat features about reports being written in QWeb is they can be viewed within the web browser. This can be useful if you want to embed a hyperlink that leads to a specific report.

- **Barcodes**

> Odoo has a built-in barcode image creator that allows for barcodes to be embedded in your reports.

This Documentation was created by Ali Dandan with the help of Github Copilot.