Odoo Notes

▼ Technical

▼ What are the different types of Odoo licenses ?

- **Community Edition:** Open-source and free, providing basic features and functionalities.
- **Enterprise Edition:** Paid subscription-based license offering advanced features, support, and premium modules.
- **On-premise license:** For installing Odoo on your own servers, allowing full control and customization.
- **Cloud hosting license:** For using Odoo hosted services, simplifying deployment and maintenance.

▼ odoo online vs sh vs on premise

Odoo online:

Odoo online only allows the installation of apps available on Odoo's online platform. It is not possible to add custom/third-party applications in odoo online and does not support customizations. In odoo online if you are using one application it is free for unlimited users or if you want additional applications you can buy it by subscription.

Odoo.sh

provides a managed cloud environment for hosting Odoo applications. This removes the need for users to set up and maintain their own servers. This supports all the odoo modules, third party custom apps as well as highly customizable. This includes hosting, automated backups, upgrades, security features, and access to the

integrated development environment (IDE). Pricing is influenced by factors such as the number of users, the number of Odoo instances, and the level of resources required.

Odoo on-premise:

Odoo On-Premises is a self-hosted deployment option that allows organizations to host the software on their own servers.

▼ What are the different types of actions in odoo ?

- Server actions(ir.actions.server)

```
<record id="action_production_order_merge" model="ir.actions.server">
        <field name="name">Merge</field>
        <field name="model_id" ref="mrp.model_mrp_production"/>
        <field name="binding_model_id" ref="mrp.model_mrp_production"/>
        <field name="binding_view_types">list</field>
        <field name="state">code</field>
        <field name="code">action = records.action_merge()</field>
</record>
```

- Window actions(ir.actions.act_window)
- URL actions(ir.actions.act_url)

```
    def action_preview_sale_order(self):
        self.ensure_one()
        return {
            'type': 'ir.actions.act_url',
            'target': 'self',
            'url': self.get_portal_url(),
        }
```

- Client actions(ir.actions.client)

```
{
            'type': 'ir.actions.client',
            'tag': 'display_notification',
            'params': {
                'type': 'danger',
                'message': _('Private tasks cannot be converted into sub-tasks. Plea
            }
        }
```

- Report actions(ir.actions.report)
- Automated actions(ir.cron)

```
<record id="ir_cron_account_move_send" model="ir.cron">
    <field name="name">Send invoices automatically</field>
    <field name="model_id" ref="model_account_move"/>
```

```
        <field name="state">code</field>
        <field name="code">model._cron_account_move_send(job_count=20)</fiel
        <field name="user_id" ref="base.user_root"/>
        <field name="interval_number">1</field>
        <field name="interval_type">days</field>
        <field name="numbercall">-1</field>
    </record>
```

▼ Difference between @api.model and @api.multi.

@api.model: This decorator indicates that the method operates on the model itself, not on a specific record. It is used for methods that perform actions on the model level, such as creating records, searching for records, or performing global operations.

The decorator method should work on a model level rather than the record level, such as methods for creating records or performing model-wide tasks. Methods decorated with `@api.model` are not bound to any specific record. Instead, they operate at the model level. This means that they don't require a record to be called and don't have access to any record-specific data (like `self.id` or `self.name` in a recordset). They are typically used for operations that apply to the model as a whole or when you need to perform a task that doesn't involve manipulating or querying individual records.

@api.multi: This decorator indicates that the method operates on a recordset, which is a collection of records. It is used for methods that perform actions on one or more records, such as updating records, deleting records, or performing actions specific to each record in the recordset.

▼ Explain the use of automated actions.

- **Sending emails:** Automatically send emails to customers or employees based on specific events, such as order confirmations or task assignments.
- **Creating records:** Automatically create records in other modules based on events in one module, such as creating a customer invoice when a sales order is confirmed.
- **Updating fields:** Automatically update fields in records based on specific conditions, such as updating the status of a task when it is completed.
- **Executing custom code:** Execute custom Python code to perform complex actions or integrate with external systems.

▼ Types of Inheritance in Odoo

1. Python Inheritance

   In Odoo, Python inheritance allows you to extend or modify existing model methods without rewriting them completely. Every model automatically inherits from the base models.Model class, providing access to basic CRUD methods (Create, Read, Update, Delete).

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


    @api.model
    def create(self, vals):
        # Add custom logic before creating the record
        return super(TestModel, self).create(vals)
```

2. Model Inheritance

   a. Classical Inheritance: This type of inheritance allows a new model to inherit fields, methods, and metadata from an existing model, adding or modifying functionality. It uses both name and inherit.

```
class Dog(models.Model):
    _name = 'zoo.dog'
    _inherit = 'zoo.animal'  # Classical inheritance
    _description = 'Dog'

    def sound(self):
        return f"{self.name} barks!"
```

   b. Extension Inheritance: This is used to extend an existing model without creating a new one by adding fields or modifying methods using *inherit without* name.

```
from odoo import fields, models
class ExtendedBook(models.Model):
    _inherit = 'library.book'
```

```
author = fields.Char(string='Author', default="Unknown Author")
```

c. Delegation Inheritance: Delegation inheritance allows a model to have fields from another model via a relationship. The child model doesn't inherit methods but can access fields through delegation using _inherits.

Delegation inheritance inherits only fields and methods are not inherited. It can be useful, when we need to embed a model in our current model without affecting the existing views, but we want to have the fields of inherited objects.

Any field not found in current model will be taken from the inherited models. You can have multiple inheritance, so that the new table created in DB contains your new fields and fields which delegates the inherited object fields (fields storing IDs from inherited tables.

Use Cases

1. **Embedding Models**: Add functionality to a model without altering its original structure (e.g., linking `hr.employee` to `resource.resource` for shared resource fields like availability)1.
2. **Preserving Original Views**: Inherit fields without modifying the parent model's views, requiring explicit view inheritance for customizations.
3. **Data Synchronization**: Changes to delegated fields automatically sync with the parent model1.

```
class ProductProduct(models.Model):
    _name = "product.product"
    _inherits = {'product.template': 'product_tmpl_id'}  # Delegation inheritance

product_tmpl_id = fields.Many2one('product.template', required=True, ondelete=
```

3. View Inheritance: View inheritance allows you to modify existing views in Odoo without altering them directly. This is done using XPath expressions to target specific elements in the original view and either modify or extend them

```
<record id="inherited_model_view_form" model="ir.ui.view">
  <field name="name">inherited.model.form.inherit</field>
  <field name="model">inherited.model</field>
  <field name="inherit_id" ref="base.model_view_form"/>
  <field name="arch" type="xml">
    <xpath expr="//field[@name='description']" position="after">
      <field name="new_field"/>
    </xpath>
  </field>
</record>
```

▼ What is Wkhtmltopdf ?

The Wkhtmltopdf is an open-source utility that enables users to convert any given HTML page to a PDF document of an Image. It is an additional package that we can use with Odoo. It comes in handy when generating PDF reports

▼ _inherit vs _inherits

- _inherit: If we want to **extend or customize** anything for particular **object/class** `_inherit` is used. It is a single **object/class**inheritance.

  When you use `_inherit` and add new fields, those fields will be added inside inherited object. No other table will be created in database.

  For example I want to add one or more field(s) or I want to add/override method in `sale.order` then I will use:

  ```
  class SaleOrder(models.Model):
    _inherit = 'sale.order'
    _columns = {
      'my_field': fields.char('My New Field', size=50),
    }
  ```

- _inherits: If we want to use anything (fields or methods) of any object and I want to achieve multiple inheritance, then I will use `_inherits` which allows you to use features of any object. That means you can directly use fields and methods of inherited object.

  When you use `_inherits` and new table is created in database. That will have your own fields and field ID of inherited object.

```
class ProductProduct(models.Model):
    _name = "product.product"
    _inherits = {'product.template': 'product_tmpl_id'}  # Delegation inheritance

product_tmpl_id = fields.Many2one('product.template', required=True, ondelete=
```

Separate Tables: product.product has its own table for variant-specific data, keeping it separate from the template's data.

Use Case: This structure is ideal for products with multiple variants, as it avoids duplicating common data (e.g., one name for all T-Shirt variants) while allowing variant-specific attributes (e.g., unique barcodes).

Database Impact: In the product_product table, a column named product_tmpl_id is created as a foreign key to the id column of the product_template table.

product_template table:

```
id | name     | type    | list_price
---+----------+---------+------------
1  | T-Shirt  | product | 20.0
```

product_product table:

```
id | product_tmpl_id | barcode         | default_code
---+-----------------+-----------------+--------------
1  | 1               | TSHIRT-SMALL    | TS01
2  | 1               | TSHIRT-MEDIUM   | TS02
```

Delegation: The _inherits mechanism makes it appear as though product.product has all the fields of product.template, but these fields are fetched from the product.template table via the product_tmpl_id relationship.

The product_tmpl_id field acts as the link, allowing product.product to reuse fields from product.template while storing variant-specific data in its own table.

- **When to use _inherit**:

        - To add a new field to an existing model (e.g., adding a custom field to res.partner).
        - To override or extend methods of an existing model.
        - To modify views, constraints, or other attributes of an existing model.
    - **When to use _inherits**:
        - When you need a new model with its own identity but want to reuse fields and behaviors from an existing model.
        - When you want to keep the parent model's data separate from the new model's data.

▼ Explain the Purpose of the 'context' Parameter in Odoo.

The 'context' parameter is used to pass additional information between different methods and views in Odoo. It allows developers to influence the behavior of operations.

▼ What is the Purpose of the 'Odoo.sh' Service?

Odoo.sh is a cloud-based platform for hosting and managing Odoo instances. It provides a simplified deployment process and facilitates collaboration among development teams.

▼ What is the ORM in Odoo?

Odoo uses an Object-Relational Mapping (ORM) system to interact with the database. It allows developers to manipulate data in Python code without directly interacting with SQL.

▼ Context & Domain

You can pass any information in **context** by using XML or python. And **domain** is any condition that helps to filter the data when searching.

***Context*:** Context is actually a python dictionary that helps to pass needful data to a function in odoo. In odoo, you can see almost all functions had the context parameter to pass data.

Examples for using context in different ways:

1. To pass default values for fields:

```
<field name="work_location_id" context="{'default_address_id': address_id}" />
```

2. Setting default filters and groups by records

Inside the context you can add the group_by field.

```
<group expand="0" string="Group By">
    <filter string="Department" name="department" domain="[]" context="{'group
    <filter string="Status" name="status" domain="[]" context="{'group_by': 'state
    <filter string="Company" name="company" domain="[]" context="{'group_by':
    groups="base.group_multi_company"/>
</group>
```

3. In Window actions

You can use context in window actions for setting default values for new records. In this example the default type is set as opportunity.

```
<record id="crm_lead_action_my_activities" model="ir.actions.act_window">
        <field name="name">My Activities</field>
        <field name="res_model">crm.lead</field>
        <field name="view_mode">tree,kanban,graph,pivot,calendar,form,activity</f
        <field name="view_id" ref="crm_lead_view_list_activities"/>
        <field name="domain">[('activity_ids','!=',False)]</field>
        <field name="search_view_id" ref="crm.view_crm_case_my_activities_filter"/
        <field name="context">{'default_type': 'opportunity',
        'search_default_assigned_to_me': 1}
        </field>
        <field name="help" type="html">
        <p class="o_view_nocontent_smiling_face">
        Looks like nothing is planned.
        </p><p>
        Schedule activities to keep track of everything you have to do.
        </p>
        </field>
</record>
```

4. In python function

You can also use context inside a python function. If you pass any value from XML, then you can access it from your python function as self.env.context.get('partner_id'), Here partner_id is the value that passes from the XML.

Domain:You can use domains for searching or filtering data or records based on specific conditions. A domain contains a field, operator and value. You can use

different operators for filtering the data.

Th syntax for domain is: **domain="[(field_name, 'operator', ‘value’)]"**

You can use domains in different situations:-

1. In a search view filter

```
<search string="Search for mrp workcenter">
<field name="name" string="Work Center" filter_domain="['|', ('name', 'ilike', self), ('
<filter name="archived" string="Archived" domain="[('active', '=', False)]"/>
</search>
```

2. In record rule

```
<record model="ir.rule" id="res_users_log_rule">
        <field name="name">res.users.log per user</field>
        <field name="model_id" ref="model_res_users_log"/>
        <field name="domain_force">[('create_uid','=', user.id)]</field>
        <field name="perm_read" eval="False"/>
</record>
```

3. fields_view_get() method

To set dynamic values for domain filters, you can use the domain in the method fields_view_get().

4. To filter relational object fields records

```
partner_id = fields.Many2one('res.partner', 'Account Holder', domain=['|', ('is_compa
```

5. To display specific records

```
<record id="action_bank_statement_tree" model="ir.actions.act_window">
    <field name="name">Bank Statements</field>
    <field name="res_model">account.bank.statement</field>
    <field name="view_mode">tree,form,pivot,graph</field>
    <field name="domain">[('journal_id.type', '=', 'bank')]</field>
    <field name="context">{'journal_type':'bank'}</field>
    <field name="search_view_id" ref="view_bank_statement_search"/>
</record>
```

▼ Explain the Difference Between Server Actions and Automated Actions.

Server actions are used for one-time server-side operations, while automated actions are triggered based on predefined conditions, automating repetitive tasks.

▼ What is the Purpose of the 'Super()' Method in Odoo?

The `super()` method is used to call a method from a parent class, allowing developers to extend the functionality of that method in a child class.

▼ What is a Wizard in Odoo, and How is it Different from a Form ?

A wizard in Odoo is a transient interface used for specific operations or data input. It differs from a form as it does not create a new record in the database.

▼ What is recordset ?

It is a set of records which represent a collection of data for specific odoo module.

▼ What is cursor ?

It is an object useful to execute queries & manipulate data in the database

▼ What is env ?

stores various contextual data used by the ORM: the database cursor (for database queries), the current user (for access rights checking) and the current context (storing arbitrary metadata). The environment also stores caches.

All recordsets have an environment, which is immutable, can be accessed using `env` and gives access to:

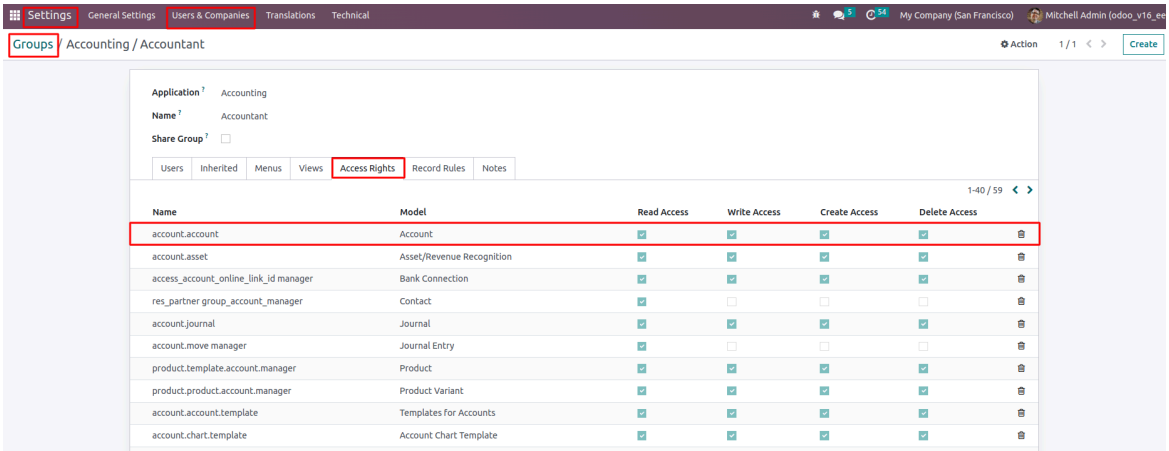
- the current user (`user`)
- the cursor (`cr`)
- the superuser flag (`su`)
- or the context (`context`)

```
>>> records.env
<Environment object ...>
>>> records.env.user
res.user(3)
>>> records.env.cr
<Cursor object ...)
```

▼ Access right vs record rule

Access rights define the access a user can have to a particular object. Those global rights are defined per document type or model. rights follow the CRUD model: create, read (search), update (write), delete (unlink).

For example, you can define access rights for Accounting / Accountant group users to read, write, create, and delete access to accounts/charts of accounts records.



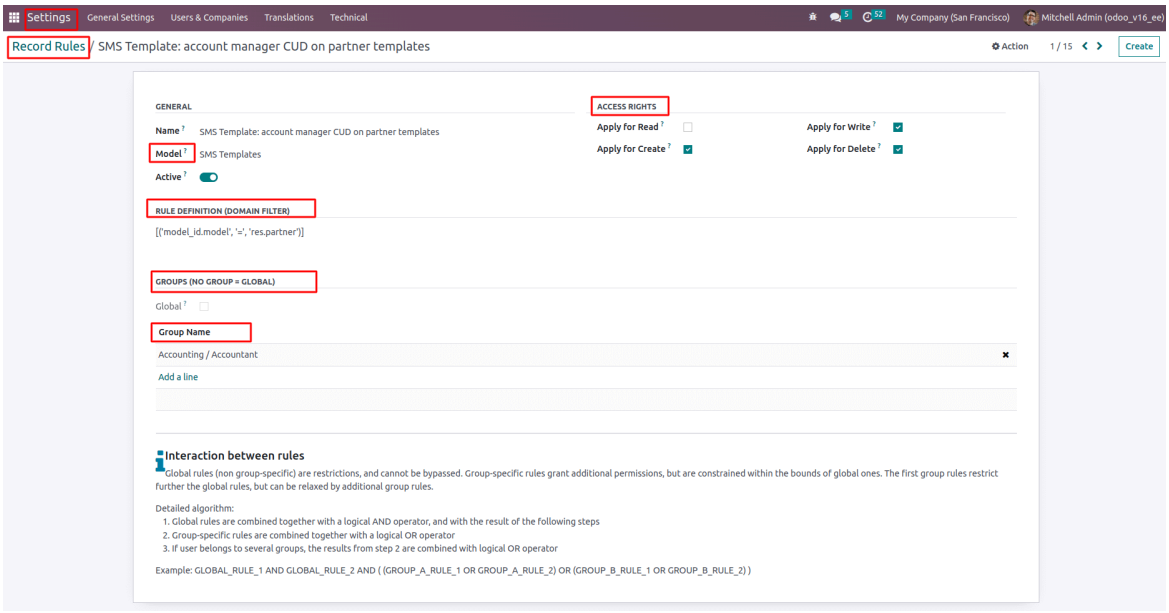
Record rules are conditions that records must satisfy for an operation (create, read, update or delete) to be allowed. It is applied record-by-record after access control has been applied.

Model: Define the Model on which we have to apply the Record Rule. for example, SMS Templates.

Access Rights: As we can see in the screenshots we can assign access rights like, Read, Write, Create, and Delete as needed.

Groups / Group Name: If we assign Groups in Record rules, it means the record rule will apply only to those specific group users, not to all users.If we do not apply any Group on record rule, the record rule becomes Global. It means the record rule will apply to all users.

Rule Definition: Rule definition is domain filter, where we can define some conditions to restrict Or to show some specific documents/records to users.



▼ Incoming vs Outgoing mail server

▼ How to Configure Incoming Mail Server in Odoo 18 ?

Gmail

i. **Name:** Add mail server name i.e. Gmail Incoming Mail Server

ii. **Server Type:** POP Server

iii. **Server Type:** smtp.gmail.com

iv. **Port: 995**

v. **SSL/TLS:** To enhance message security, make sure to enable this option for encrypting the messages.

vi. Add Username & app password of gmail

▼ How to Configure Outgoing Mail Server in Odoo 18 ?

1. Mailtrap.io

   i. **Name:** Add mail server name i.e. MailTrap Outgoing Server

   ii. **Authenticate With:** Select **User Name** as the authentication method.

   iii. **SMTP Server:** smtp.mailtrap.io

   iv. **SMTP Port:** 2525

   v. **Connection Encryption:** TLS (STARTTLS)

   vi. Add Username & Password

2. Gmail

i. **Authenticate With:** Select **User Name** as the authentication method.

ii. **SMTP Server:** smtp.gmail.com

iii. **SMTP Port: 465**

iv. **Connection Encryption:** SSL/TLS

v. Add Username & app password of gmail

▼ How to create & manage wizard ?

A wizard is a model that extends the class Transient Model rather than a model. The class TransientModel extends models and reuses all of its existing mechanisms in the following ways:

1. Wizard data is not permanent; it is automatically deleted from the database after a certain period.

2. All users have complete access to wizard data.

3. Wizard records can refer to either regular or wizard data via many2one fields, but regular data cannot refer to wizard data via a many2one field.

To get started, we need to create a transient model class for the wizard. Create a wizard directory in your module as well.

```
from odoo import fields, models
class WhatsappSendMessage(models.TransientModel):
  """This model is used for sending WhatsApp messages through Odoo."""
  _name = 'whatsapp.send.message'
  _description = "Whatsapp Wizard"
  user_id = fields.Many2one('res.partner', string="Recipient")
  mobile = fields.Char(related='user_id.mobile', required=True)
  message = fields.Text(string="message", required=True)
```

Next, we must create a view for that wizard to open. Wizards should always be opened as ir.actions—act_window data. A menu item or button action defined in Python or XML code can initiate the wizard action.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<odoo>
  <!-- Define a new view to extend the res.partner form view -->
  <record id="view_partner_form" model="ir.ui.view">
    <field name="name">res.partner.view.form.inherit.whatsapp.redirect</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
      <!-- Add a WhatsApp button to the partner's form view -->
      <xpath expr="//div[@name='button_box']"
          position="inside">
        <div class="oe_button_box" name="button_box">
          <button name="action_send_msg" string="Whatsapp"
              type="object"
              class="oe_stat_button" icon="fa-whatsapp"/>
        </div>
      </xpath>
    </field>
  </record>
</odoo>
```

After that, we can add a function to the Button.

```python
def action_send_msg(self):
  """This function is called when the user clicks the
   'Send WhatsApp Message' button on a partner's form view. It opens a
    new wizard to compose and send a WhatsApp message."""
  return {'type': 'ir.actions.act_window',
        'name': _('Whatsapp Message'),
        'res_model': 'whatsapp.send.message',
        'target': 'new',
        'view_mode': 'form',
        'view_type': 'form',
        'context': {'default_user_id': self.id}, }
```

type: Specifies the type of action, which is opening a window.

name: Sets the name (title) of the window.

res_model: Specifies the model associated with the window.

target: Indicates that the window should be opened in a new tab or window.

view_mode: Specifies the view mode for the window.

view_type: Specifies the type of view.

context: Passes a context to the new window.

Next, we need to create a view to display the wizard. We can apply the following code to do this.

```
<?xml version="1.0" encoding="UTF-8" ?>
<odoo>
  <!-- Odoo View Definition for Whatsapp Message Form -->
  <record id="whatsapp_send_message_view_form" model="ir.ui.view">
    <field name="name">whatsapp.send.message.view.form</field>
    <field name="model">whatsapp.send.message</field>
    <field name="priority" eval="8"/>
    <field name="arch" type="xml">
      <form string="Whatsapp Message">
        <group>
          <field name="user_id"/>
          <field name="mobile"/>
        </group>
        <group>
          <field name="message"/>
        </group>
        <footer>
          <button name="action_send_message" string="Send" type="object"/
>
          <button name="cancel" string="Cancel" special="cancel"/>
        </footer>
      </form>
    </field>
  </record>
</odoo>
```

▼ SQL v/s Python Constraints

SQL Constraints:

1- SQL constraints are enforced at the database level.

2- They ensure data integrity by defining rules and conditions directly in the database schema.

3- SQL constraints include primary key constraints, foreign key constraints, unique constraints, check constraints, etc.

4- SQL constraints are defined in the sql_constraints attribute of the database table definition in the Odoo model.

```
_sql_constraints = [
      # email notification: partner is required
      ('notification_partner_required',
       "CHECK(notification_type NOT IN ('email', 'inbox') OR res_partner_id IS NOT NU
       'Customer is required for inbox / email notification'),
  ]
```

Python Constraints:

1- Python constraints are enforced at the application level.

2- They provide a way to define custom rules and conditions using Python code.

3- Python constraints are defined as methods in the Odoo model, and they are executed whenever the corresponding fields are modified.

4 -Python constraints can have complex logic and can access other fields and records in the model.

5- Python constraints are defined using the @api.constrains decorator in the Odoo model.

```
@api.constrains('exam_date', 'deadline')
    def _check_date(self):
        if self.exam_date < self.deadline:
            raise ValidationError(_('Deadline must be greater than exam date'))
```

When to use SQL Constraints:

1- Use SQL constraints when you need to enforce data integrity rules that are fundamental to the database schema.

2- SQL constraints are generally faster and more efficient because they are enforced at the database level.

When to use Python Constraints:

1- Use Python constraints when you need to enforce data integrity rules that involve complex logic or require access to other fields and records.

2- Python constraints provide more flexibility and control since they are executed at the application level.

3- Python constraints can also be used to perform additional validation or trigger specific actions based on field values.

In general, it's recommended to use SQL constraints for basic data integrity rules that can be expressed directly in the database schema, and use Python constraints for more complex rules and logic that require application-level processing.

▼ Model Types in Odoo: Model vs. TransientModel vs. AbstractModel

Model

The Model class in Odoo is the most commonly used type for defining persistent data structures. It corresponds directly to a database table, with instances of the model representing records in that table. Here's an example of defining a Product model in Odoo:

```
from odoo import models, fields


class Product(models.Model):
    _name = 'my_module.product'  # Database table name

    name = fields.Char(string='Name', required=True)
  description = fields.Text(string='Description')
  price = fields.Float(string='Price')
```

In this example, the `Product` model represents products in our system, with fields such as name, description, and price. Instances of this model will be stored in the database.

TransientModel

TransientModel, on the other hand, is designed for handling temporary data that doesn't need to be persisted in the database. Instances of TransientModel exist only in memory during the session and are not stored in the database. Let's consider an example of a transient model for a wizard in Odoo:

```
from odoo import models, fields

class Wizard(models.TransientModel):
    _name = 'my_module.wizard'

    name = fields.Char(string='Name', required=True)
    description = fields.Text(string='Description')
    date = fields.Date(string='Date')
```

In this example, the `Wizard` model is used for creating a transient wizard in Odoo. The data entered in this wizard won't be stored in the database; it's transient and exists only during the session.

AbstractModel

AbstractModel serves as a blueprint for other models and cannot be instantiated on its own. It's designed for defining common fields and methods that can be shared among multiple models. Let's define an abstract model with common fields:

```
from odoo import models, fields

class CommonFields(models.AbstractModel):
    _name = 'my_module.common_fields'

    name = fields.Char(string='Name', required=True)
    description = fields.Text(string='Description')
```

In this example, `CommonFields` is an AbstractModel defining common fields like name and description. Other models can inherit from this abstract model to reuse these fields:

```
class Product(models.Model):
    _name = 'my_module.product'
    _inherit = 'my_module.common_fields'

    price = fields.Float(string='Price')
```

Here, the `Product` model inherits from `CommonFields`, automatically gaining the fields defined in `CommonFields`.

▼ name_search vs name_get

name_search

It is an ORM method responsible for searching some specific record by some field values in a relational field.

@api.model decorator: This decorator indicates that the method can be called without creating an instance of the model (i.e., directly on the model class).

Arguments:

name (str): This parameter represents the search pattern (partial value) to match against the relevant fields.

args (list, optional): This optional parameter allows you to specify additional domain criteria for the search.

operator (str, optional): This parameter defines the search operator used for comparison. By default, it's set to 'ilike' (case-insensitive like operator).

limit (int, optional): This parameter determines the maximum number of records to return in the search results.

Returns:

List of tuple

```
@api.model
def _name_search(self, name, domain=None, operator='ilike', limit=None, order=N
    domain = domain or []
    if name:
        email_domain = [('email', 'ilike', name)]
```

```
        domain = expression.AND([email_domain, domain])
    return self._search(domain, limit=limit, order=order)
```

name_get

Note:- It is deprecated in odoo17 instead you can use display_name field. If needed to change display_name then override compute_display_name method.

It is an ORM method used to get display_name of records.It returns a list of tuples, where each tuple contains a record's ID and its display name, which is what will be shown in the interface.

```
def name_get(self):
    result = []
    for record in self:
    display_name = f"{record.name} - {record.code}"  # Customize this
    result.append((record.id, display_name))
    return result
```

▼ display_name

It is the textual representation of a record. ie, the name being displayed in the relational fields defined based on that model and on top of the form view of a record.

- By default, object name field will be displayed as the name of the record by default.
- If we want to show using other field then we need to specify _rec_name attribute for that model.
- When we need the display name to be a combination of multiple fields or to be in some other pattern, need to override name_get function(Upto v16)/_compute_display_name function (V17).

▼ Docker

Docker is a container **platform that allows developers to package applications and their dependencies into units called "containers."** Each container is a virtualized,

isolated instance of an application that includes everything necessary for it to run independently.

Images: A Docker **image** is a read-only template used to build containers.

- It is a lightweight, standalone, executable package of software that includes everything needed to run an application, such as code, runtime, system tools, system libraries, and settings.
- Images are created using a `Dockerfile` which contains instructions for building the image.
- Images are used to distribute an application consistently across different environments.

Containers: A Docker container is a running instance of a Docker image.

- They are created from images and can be started, stopped, moved, or deleted
- They are lightweight because they share the host's operating system kernel.
- They are used to run an application in an isolated environment.

Docker Daemon: That background service running on the host that listens to API calls (via the Docker client), manages images and building, running and distributing containers. The Deamon is the process that runs in the operating system which the client talks to – playing the role of the broker.

Docker Client: The command line tool that allows the user to interact with the daemon. There are other forms of clients too.

Docker Hub: A registry of Docker images containing all available Docker images. A user can have their own registry, from which they can pull images or upload ("push") images that they have built.

Dockerfile: A Dockerfile is a simple text file that specifies the steps needed to create a Docker image.

▼ Decorators

1) api.model

Methods decorated with `@api.model` are not bound to any specific record. Instead, they operate at the model level rather than the record level, such as methods for creating records or performing model-wide tasks.This means that they don't require a record to be called and don't have access to any record-specific data (like `self.id` or `self.name` in a recordset).

They are typically used for operations that apply to the model as a whole or when you need to perform a task that doesn't involve manipulating or querying individual records.

```
@api.model
def default_get(self, fields):
    result = super(ProductTemplate, self).default_get(fields)
    if self.env.context.get('default_can_be_expensed'):
        result['supplier_taxes_id'] = False
    return result
```

2) api.autovacuum

The ‘ @api.autovacuum’ decorator in Odoo schedules methods so that it is called by the daily vacuum cron job (model `ir.autovacuum`) used for executing garbage collection-like tasks that require daily cleanup without needing a dedicated cron job entry.

```
@api.autovacuum
def remove_inactive_users(self):
    """Remove inactive user records."""
    self.env['res.users'].search([('active', '=', False)]).unlink()
```

3) api.constrains

This decorator is used for model validation methods and these methods run whenever any of the mentioned fields are changed. These methods should only validate the data and raise an exception if a check fails. The decorator will only be triggered if the specified fields are included in a create or write call, or if they are present in the view.

```
@api.constrains('amount_total')
 def _check_amount_total(self):
    for order in self:
            if order.amount_total <= 0:
```

```
            raise ValidationError("The total amount of the order must be greater tha
n ze
```

4) api.depends

The ‘@api.depends’ decorator in Odoo is used to specify field dependencies for computed fields. It informs Odoo that the computed field's value depends on the values of other fields, ensuring that the computed field is recalculated whenever any of its dependencies change.

```
@api.depends('cost_price', 'sale_price')
    def _compute_profit_margin(self):
        for product in self:
            if product.cost_price and product.sale_price:
                product.profit_margin = ((product.sale_price - product.cost_price) / pro
duct.cost_price) * 100
            else:
```

6) api.onchange

Contrary to the `api.depends` decorator, this decorator is used for changes in the user interface. This decorator is triggered if a user edits a value in a particular field in the user interface. This method can change other field values, validate the field input, display a message to the user, or set a domain filter which limits the available options for relational fields. Note that it does not write store data directly, but it provides feedback to the user, which allows a user to change the data in the user interface.

```
@api.onchange('partner_id')
    def _onchange_partner_id(self):
        if self.partner_id:
            self.payment_term_id = self.partner_id.payment_term_id
```

7) api.model_create_multi

The Odoo @api.multi decorator creates multiple records from a single or a list of dictionaries. It's useful for bulk record creation, improving efficiency and productivity. The values are conveyed as parameters to the method.

```
@api.model_create_multi
  def create(self, vals):
      events = super(CalendarEvent, self).create(vals)
      for event in events:
          if event.opportunity_id and not event.activity_ids:
              event.opportunity_id.log_meeting(event)
      return events
```

8) api.ondelete

The ‘@api.ondelete’ decorator in Odoo marks a method to be called when records are being unlinked (deleted). It's often used to prevent the deletion of records, such as lines on posted entries, during module uninstallation, which cannot be achieved by overriding the unlink() method.

Parameters:

- **at_uninstall:** A boolean parameter indicating whether the decorated method should be called during module uninstallation. If False, errors raised by the method won't prevent module uninstallation.

```
@api.ondelete(at_uninstall=False)
def _unlink_if_not_done(self):
    if any(batch.state == 'done' for batch in self):
        raise UserError(_("You cannot delete Done batch transfers."))
```

9) api.returns

The ‘@api.returns’ decorator is used to specify the return type of methods that return instances of a particular model. It adapts the method output to the Odoo API style, either traditional style (returning ids or False) or record style (returning recordset).

Parameters:

- **model:** Specifies the model name or 'self' for the current model.
- **downgrade:** Optional function to convert the record-style output to traditional-style output.
- **upgrade:** Optional function to convert the traditional-style output to record-style output.

```
  @api.model
   @api.returns('res.partner')
   def find_partner(self, arg):
      # Return some record
      return self.env['res.partner'].search([('name', '=', arg)])

# Traditional style call
partner_id = model.find_partner(arg)
# Record style call
partner_record = recs.find_partner(arg)
```

10) api.depends_context

The depends_context decorator is used with non-stored “compute” methods to specify the context dependencies for the methods. The arguments passed will be the key in the context's dictionary and the dependencies must be hashable.

In this decorator, these keys have special support:

company: value in context or current company id

uid: current user id and superuser flag

active_test: value in env.context or value in field.context

```
tz_mismatch = fields.Boolean(compute='_compute_tz_mismatch')
@api.depends('tz')
@api.depends_context('uid')
def _compute_tz_mismatch(self):
   for leave in self:
      leave.tz_mismatch = leave.tz != self.env.user.tz
```

▼ ORM Methods

1. create()
   - Used to create new records in the database.
   - **Model.create(vals_list)**
   - Accepts a dictionary of values in a list.
   - It returns a recordset (a list of new records created).

```
@api.model
    def create(self, vals):
        if vals.get('circular_number', 'New') == 'New':
            vals['circular_number'] = self.env['ir.sequence'] \
                                      .next_by_code('op.circular.german') or 'New'
        result = super(OpCircular, self).create(vals)
        return result
```

2. write()
    - Updates existing records with the given data.
    - **Model.write(vals)**
    - In the example, the field email of the partner record gets updated by the given value.
    - Accepts a dictionary with fields to update and the value to set on them.

```
def write(self, vals):
        res = super(OpFaculty, self).write(vals)
        if vals.get('email') and self.emp_id:
            hr_emp = self.env['hr.employee'].search([('id', '=', self.emp_id.id)])
            hr_emp.write({'work_email': vals.get('email')})
        return res
```

3. browse()
    - Takes a database id or a list of ids and returns a recordset
    - **Model.browse([ids]) ? records**
    - In the example, browsing inside the ‘res.partner’ model for the record with id, partner_id.

```
partner_id = self.env['res.partner'].sudo().browse(partner_id)
```

4. exists()

    The exists() method returns the subset of records that exists in self. It returns a recordset which contains only the records that exist in the database. It can also be used to check whether a particular record still exists.

    Syntax: **Model.exists() ? records**

```
if not record.exists():
    raise Exception("The record has been deleted")
```

5. search()

   Takes a *search domain*, returns a recordset of matching records. Can return a subset of matching records (offset and limit parameters) and be ordered (order parameter):

   - **Model.search(domain[, offset=0][, limit=None][, order=None])**
   - Use an empty list to match all records.

```
search_res = self.env['crm.lead'].search([('id', 'in', test_leads.ids)], limit=5, offse
```

6. search_count()

- Returns the number of records that match the given domain.
- **Model.search_count(domain[, limit=None])**
- In the example, returns the count of active records in the ‘res.partner’ model.

```
count = self.env['res.partner'].search_count([('active', '=', True)])
```

7. unlink()

- Delete records from the database.
- In the example, the row corresponding to the partner record will be removed from the database.
- This method permanently deletes records, so it should be used carefully.

```
def unlink(self):
    moves = self.with_context(force_delete=True).move_id
    res = super().unlink()
    moves.unlink()
    return res
```

8. ensure_one()

   checks that the recordset is a singleton (only contains a single record), raises an error otherwise:

```
records.ensure_one()
```

9. copy()

   The copy() method is used to duplicate a recordset in the current model. This method accepts only the default argument.

```
# Get the original customer record
original_customer = env['res.partner'].browse(customer_id)

# Create a copy with a modified discount
new_customer = original_customer.copy({'supplier_rank': 1})

print(new_customer.name)  # Output: Same name as original customer
print(new_customer.email)  # Output: Same email as original customer
print(new_customer.supplier_rank)  # Output: 1 (modified value)
```

10. search_read()

    The search_read() method combines the search() and read() methods to fetch specific fields of records that meet certain criteria. This method reduces the number of database calls, making it more efficient for retrieving specific data points.

    - **Model.search_read(domain,fields,order, limit=limit)**

```
self.env['hr.employee'].sudo().search_read(domain=[('company_id', '=', company
                                    fields=["id",
                                        "name",
                                        "avatar_1024",
                                        "job_id",
                                        "department_id"],
                                    order='id asc',
                                    limit=10
                                    )
```

    Use search_read when you need to fetch specific fields from records matching a domain, especially in API calls, reports, or when you want to minimize memory usage by avoiding full recordset loading.

11. default_get()

The default_get() method fetches default values for fields in a model, allowing default data to be calculated based on conditions.

Method Signature: `default_get(self, fields)`

```
@api.model
 def default_get(self, fields):
    res = super(SessionPopup, self).default_get(fields)
    context = dict(self.env.context)
    active_id = context.get('active_id', False)
    active_session_id = self.env['op.session'].browse(active_id)
    res.update({
        'session_id': active_session_id.id
    })
    return res
```

12. flush()

    The flush method is an ORM function that forces the immediate writing of any pending changes in the current transaction to the database.

    - The given example is a method used in Odoo to manually flush the SQL query cache and ensure that any pending database operations or changes in the Odoo ORM (Object-Relational Mapping) are committed to the database.

    ```
    self.env.cr.flush()
    ```

▼ Monkey Patching

Monkey Patching is a technique that allows us to modify or extend the behavior of existing code at runtime. It makes changes to the code without changing the source code.

It is useful when we want to add new features, fix bugs, or to override the existing code.

```
def prepare_procurement_values(self, group_id=False):
           Order_date = self.date_order
           Order_id = self.order_id
           deadline_date = self.delivery_date or (
                           order_id.order_date +
                           timedelta(Days= self.customer_lead or 0.0)
                           )
        planned_date = deadline_date - timedelta(
            days=self.order_id.companyid.security_lead)
        values = {
            'group_id': group_id,
            'sale_line_id': self.id,
            'date_planned': planned_date,
            'date_deadline': deadline_date,
            'route_ids': self.route_id,
            'warehouse_id': order_id.warhouse_id or False,
            'product_description_variants': self.with_context(
                lang=order_id.partner_id.lang).
            _get_sale_order_line_multiline_description_variants(),
            'company_id': order_id.company_id,
            'product_packaging_id': self.product_packaging_id,
            'sequence': self.sequence,
 }
         return values
    SaleOrderLine._prepare_procurement_values = _prepare_procurement_values
```

▼ RecordSet Operations

1. filtered()

   returns a recordset containing only records satisfying the provided predicate function. The predicate can also be a string to filter by a field being true or false.

   Syntax: **Model.filtered(func) ? records**

   ```
   # only keep records whose company is the current user's
   records.filtered(lambda r: r.company_id == user.company_id)

   # only keep records whose partner is a company
   records.filtered("partner_id.is_company")
   ```

2. mapped()

   applies the provided function to each record in the recordset, returns a recordset if the results are recordsets:

   Syntax: **Model.mapped(func) ? recordset / list**

   The provided function can be a string to get field values:

   ```
   # returns a list of names
   records.mapped('name')

   # returns a recordset of partners
   ```

```
record.mapped('partner_id')

# returns the union of all partner banks, with duplicates removed
record.mapped('partner_id.bank_ids')
```

3. sorted()

   The sorted() method returns the records inside the self, ordered by the key passed as the argument where key can be field name or function. If no key is provided, use the model's default sort order

   Syntax: **Model.sorted(key=None, reverse=False)**

```
# sort records by name
records.sorted(key=lambda r: r.name)

# sort records by id
self.env['res.partner'].search([('is_company','=',True)]).sorted('id')
```

▼ Compute v/s Onchange

Compute Method

A compute method in Odoo is a function that calculates the value of a field based on other fields' values rather than directly reading from the user.

Defining Computer Methods

To define a compute method, you use the @api.depends decorator to specify the fields on which the computation depends. This ensures that the compute method is triggered whenever any dependent fields change.

```
from odoo import models, fields

class VehicleRental(models.Model):
  _name = "vehicle.rental"
  _description = "Vehicle Rental"
  hour_rate = fields.Monetary(string="Hour Rate",)
```

```
hours = fields.Integer(string="Hours")
total_rent = fields.Monetary(string='Total Rent',
                        compute='_compute_total_rent')
 @api.depends('hour_rate','hours')
def _compute_total_rent(self):
   for record in self:
      record.total_rent = record.hour_rate*record.hours
```

Use Cases for Compute Methods

Compute methods are ideal for scenarios where the value of a field is derived from other fields. Everyday use cases include:

I am calculating totals or aggregates.

I am deriving dates or periods.

We are generating dynamic text or descriptions.

- computed fields are not stored by default, they are computed and returned when requested. Setting store=True will store them in the database and automatically enable searching.
- to allow *setting* values on dependencies of a computed field, use the inverse parameter. It is the name of a function reversing the computation and setting the relevant fields.
- the "inverse" attribute is used in the definition of computed fields to specify a method that will be called when the value of the computed field is set.The purpose of the "inverse" method is to update the dependency field that contribute to the computed field based on the new value assigned to it.
- The "inverse" method provides a way to update those dependencies when the value of the computed field is changed. Note that for the inverse method to work correctly, the computed field should be defined with the store=True attribute. This tells Odoo to store the computed field's value in the database, allowing it to be set explicitly and triggering the inverse method appropriately.

  ```
  email_from = fields.Char(
        compute='_compute_email_from', inverse='_inverse_email_from', readonly=

  @api.depends('partner_id.phone')
   def _compute_phone(self):
  ```

```
    for lead in self:
        if lead.partner_id.phone and lead._get_partner_phone_update():
            lead.phone = lead.partner_id.phone

def _inverse_phone(self):
    for lead in self:
        if lead._get_partner_phone_update():
            lead.partner_id.phone = lead.phone
```

Onchange

An on-change method in Odoo is used to perform specific actions when the value of a field changes in the user interface. Unlike compute methods, which update field values in the database, on-change methods are triggered in the form view and can be used to update other fields or provide immediate feedback to the user.

```
@api.onchange('multi_faculty')
    def _onchange_faculty(self):
        if self.multi_faculty:
            self.faculty_id = False
        else:
            self.faculty_ids = False
```

Use Cases for Onchange Methods

Onchange methods suit scenarios where user interaction requires immediate feedback or additional logic. Everyday use cases include:

I am updating related fields based on user input.

It validates input and provides warnings or errors.

Choosing Between Compute and Onchange

In this example, compute methods are used to dynamically calculate the subtotal for each order line and the amount total for the entire order. Onchange methods are used to update the price unit and taxied when the product changes and to calculate the tax amount when the tax or subtotal changes.

▼ XMLRPC v/s JSONRPC

A Web service, in very broad terms, is a method of communication between two applications or electronic devices over the World Wide Web (WWW).

What is RPC ?

RPC (Remote Procedure Call) is used to make a remote call to a python method.

Odoo web services can be of two types → **XML-RPC and JSON-RPC**

XML RPC

XML-RPC allows connecting programs running on different programming languages and systems working on different operating systems. It's a remote procedure call where HTTP is used for the transport and XML for encoding.

To access data in Odoo, first, we need to create a connection with odoo. There are two types of xmlrpc endpoints in Odoo.

1) **xmlrpc/2/common** This endpoint is used to authenticate the user or fetch the version of odoo. The authenticate method returns the id of the user.

```
from xmlrpc.client import ServerProxy
url = 'http://localhost:8069' # odoo instance url
database = 'data' # database name
user = 'admin' # username
# we can generate api key from user or use password
password = '75d90f11aff5b58768c433cac87ab915bf71116c' # api key
common = ServerProxy('{}/xmlrpc/2/common'.format(url))
uid = common.authenticate(database, user, password, {})
```

Calling Methods:

2) The second endpoint is xmlrpc/2/object which will be used to access data from odoo or create data in odoo. execute_kw method of xmlrpc/2/object will help us to access the data.

```
model = xmlrpc.client.ServerProxy('{}/xmlrpc/2/object'.format(url))

# search orm
partners_ids = model.execute_kw(database, uid, password, 'res.partner', 'search', [[
```

JSON RPC

It is an RPC encoded with JSON. It is similar to XML-RPC, but it's much lighter than XML-RPC and does not require a response for sending data to the server.

Connection to Odoo:

We can connect to Odoo using the json-rpc method.

```
import json
import random
import urllib.request

host = 'localhost'
port = 8069
database = 'test_docker'
user = 'admin'
password = 'admin'


def json_rpc(url, method, params):
    data = {
        "jsonrpc": "2.0",
        "method": method,
        "params": params,
        "id": random.randint(0, 1000000000),
    }
    req = urllib.request.Request(url=url, data=json.dumps(data).encode(), headers={
        "Content-Type": "application/json",
    })
    reply = json.loads(urllib.request.urlopen(req).read().decode('UTF-8'))
    if reply.get("error"):
        raise Exception(reply["error"])
    return reply["result"]


def call(url, service, method, *args):
    return json_rpc(url, "call", {"service": service, "method": method, "args": args})
```

```
url = "http://%s:%s/jsonrpc" % (host, port)
uid = call(url, "common", "login", database, user, password)
print(uid)


order_ids = call(url, "object", "execute", database, uid, password, "sale.order", "read
print("Order: ", order_ids)
```

▼ Graph Calendar Search Pivot Gantt Activity View

1) Graph

A Graph View in Odoo is a unique view that helps display data in bar charts, line graphs, or pie charts. It is built using the Odoo graph tag, making visualizing records easier than viewing raw data in list or form views.

Odoo supports three types of graphs:

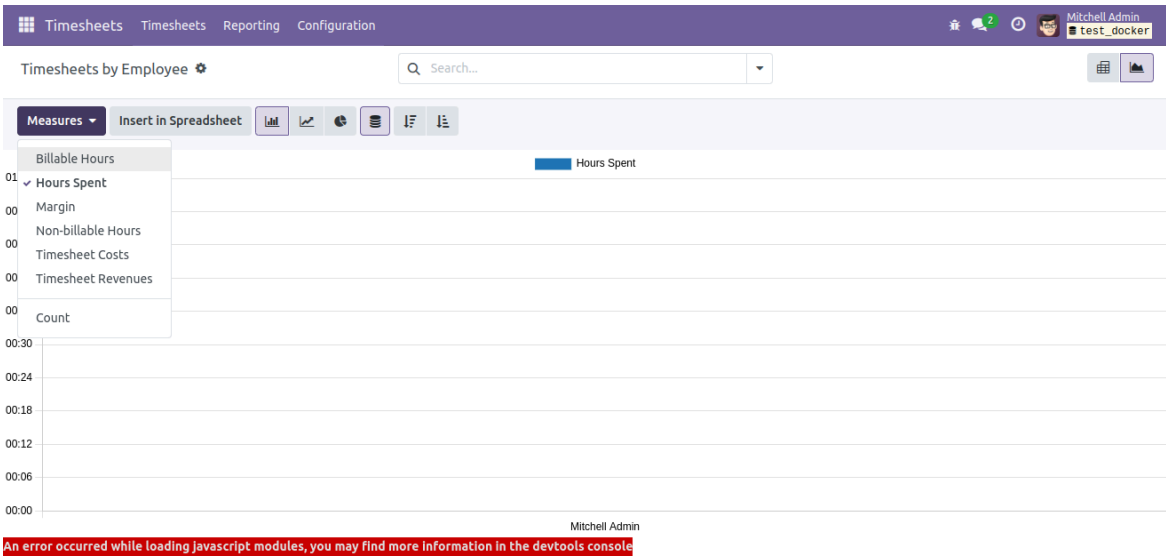
- **Bar Chart** – Best for comparing categories or groups
- **Line Chart** – Ideal for showing trends over time
- **Pie Chart** – Used to display proportions or percentages

How Does This Work?

- <graph string="Custom Model Graph"> Defines the graph's title.
- <field name="employee_id" type="row"/> Groups data by the date field.
- <field name="unit_amount" string="Hours Spent" type="measure"/> Uses value as the numeric measure for the graph.

```
<record id="timesheets_analysis_report_graph_employee" model="ir.ui.view">
        <field name="name">timesheets.analysis.report.graph</field>
        <field name="model">timesheets.analysis.report</field>
        <field name="arch" type="xml">
          <graph string="Timesheets" sample="1" js_class="hr_timesheet_graphview
            <field name="employee_id" type="row"/>
            <field name="amount" string="Timesheet Costs"/>
            <field name="unit_amount" type="measure" widget="timesheet_uom"/>
            <field name="billable_time" widget="timesheet_uom"/>
              <field name="non_billable_time" widget="timesheet_uom"/>
```

```
            </graph>
        </field>
    </record>
```



2) Calendar

In the calendar view, we can see multiple attributes like date_start, date_end. Let's explain one by one,

date_start - It indicates the starting date of the calendar event. The beginning date of the calendar event will be used if we supply a date field from the model for which we are building the calendar view in this field.

date_stop - It indicates the ending date of the calendar event.

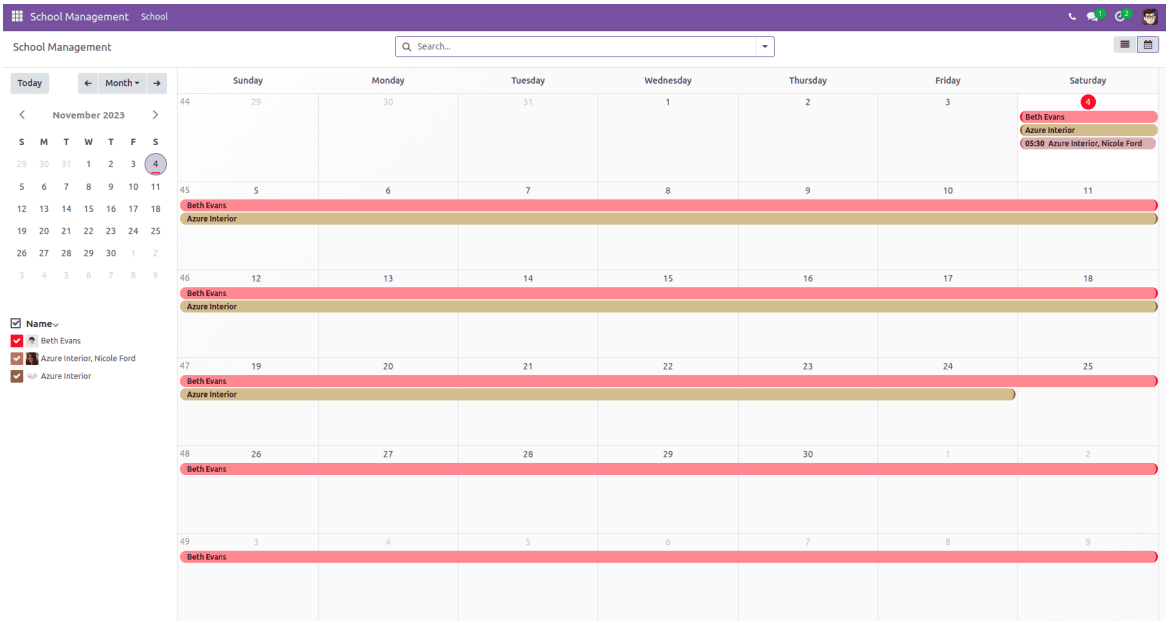
mode - A view is available on a daily, weekly, and monthly basis with the calendar view. It displays the Calendar view's default view mode as soon as the page loads. The values for this attribute can be Day, Week, or Month.

color - This gives color for different events based on the specified field (generally many2one). Records with the same value for this field will show the same color.

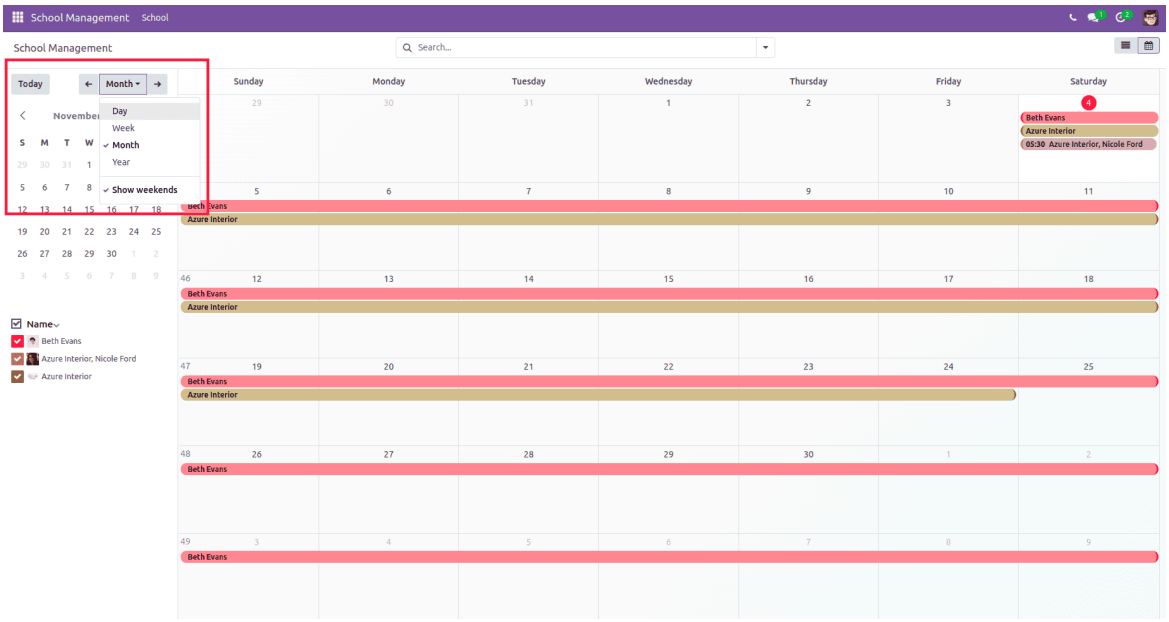
event_open_popup - If this is set, then the event will open as a popup. That is, if the event_open_popup="true" then the view that extends from the calendar will open as a popup. Otherwise, it will open events in a form.

quick_add - This field can do instant event creation default value is true.

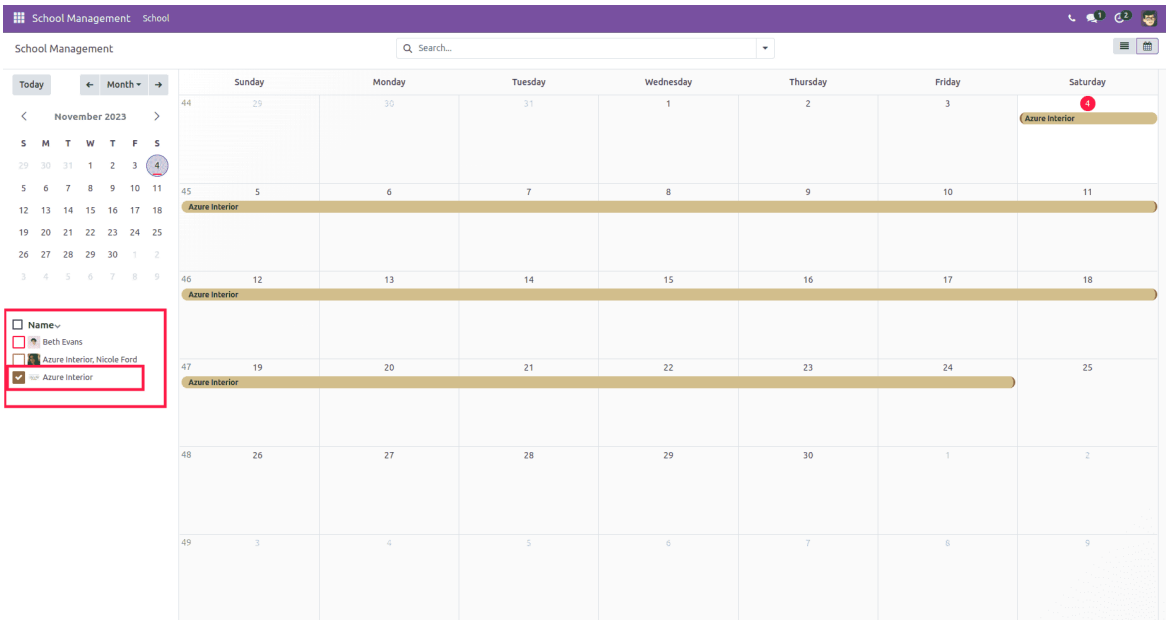
```xml
<record i="school_management_view_calendar" model="ir.ui.view">
  <field name="name">school.management.view.calendar</field>
  <field name="model">school.management</field>
  <field name="arch" type="xml">
    <calendar date_start="date" string="School Management Calendar"
        mode="month" color="name" event_open_popup="true"
       quick_add="true" date_stop="end_date">
      <field name="dob"/>
      <field name="name" avatar_field="avatar_128"/>
      <field name="st_class"/>
      <field name="division"/>
    </calendar>
  </field>
</record>
```



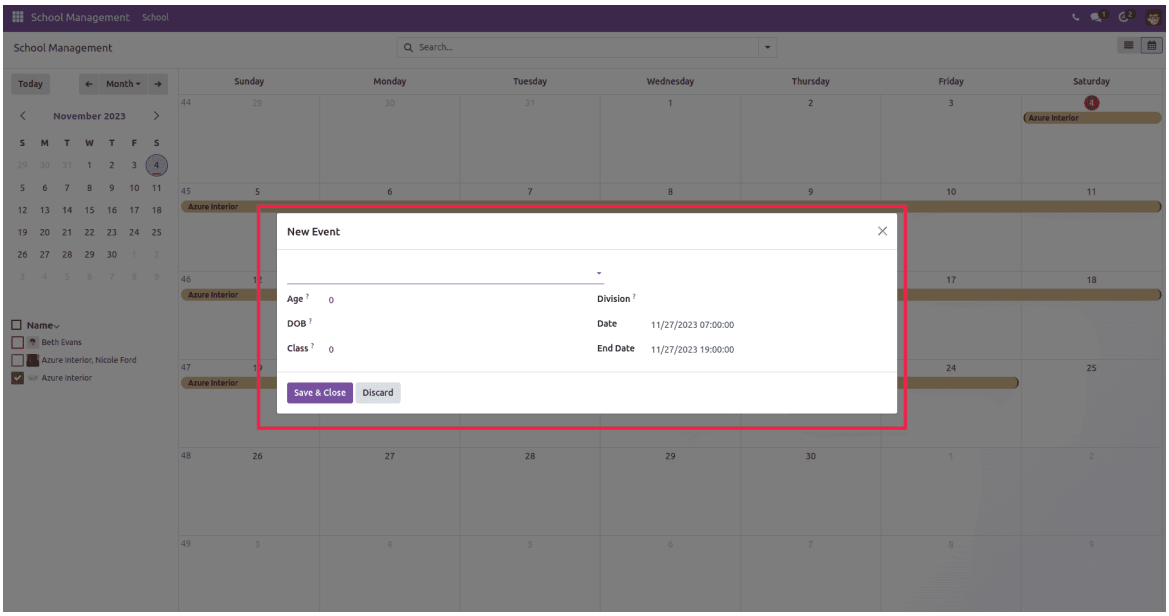
The colored area indicates that records are within that date. In the picture, we can see the records are filtered by month, and we can change this to day, week, and year.



Also, we can filter our records by the many2one field that we set as the avatar_field.



We can quickly create a new record by clicking on the date that we need to create the record.



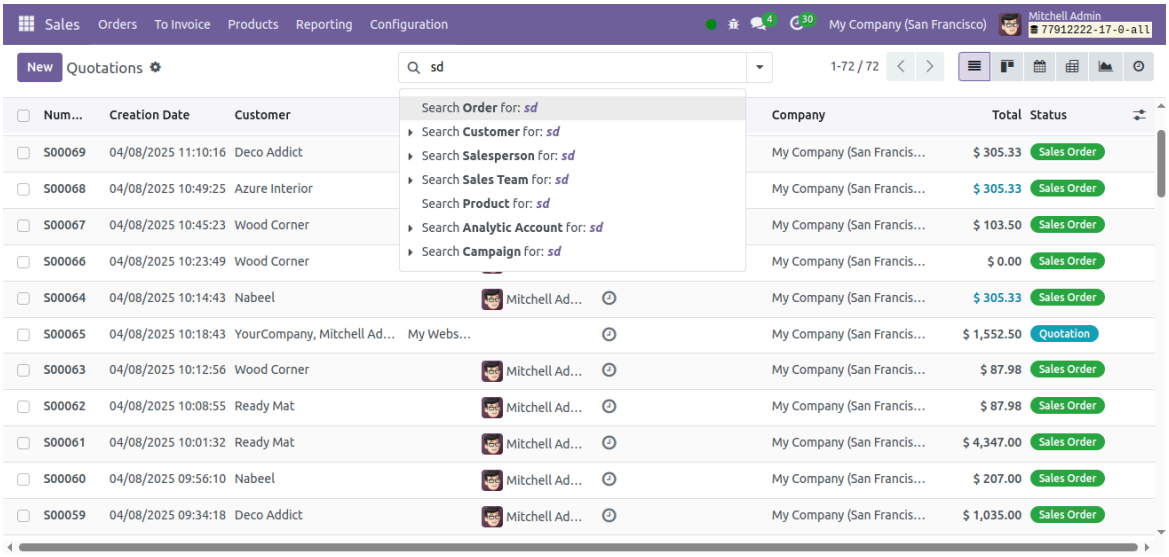
3) Search

In the search view, filters and grouping are utilized to refine records. Filtering is applied through the filter command based on specific conditions, and grouping is achieved using the group by command based on a designated field.

```
<record id="view_sales_order_filter" model="ir.ui.view">
      <field name="name">sale.order.list.select</field>
      <field name="model">sale.order</field>
      <field name="priority" eval="15"/>
      <field name="arch" type="xml">
        <search string="Search Sales Order">
          <field name="name" string="Order"
            filter_domain="['|', '|', ('name', 'ilike', self), ('client_order_ref', 'ilike', self),
          <field name="partner_id" operator="child_of"/>
          <field name="user_id"/>
          <field name="team_id" string="Sales Team"/>
        </search>
      </field>
    </record>
```

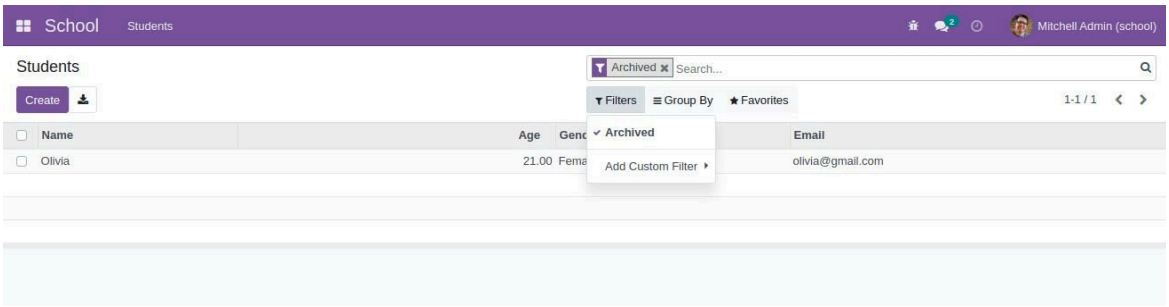
We can define the required fields to be searched inside the <**search**> tag as shown above. Here we have added ‘name’, ‘partner_id’, ‘user_id’, and ‘team_id’ under the <**search**> tag.

The above code will allow you to search for the required fields from the UI interface as shown in the image below. As you type in the search bar, you will see a dropdown menu appear with different options available to help you quickly locate your desired information.



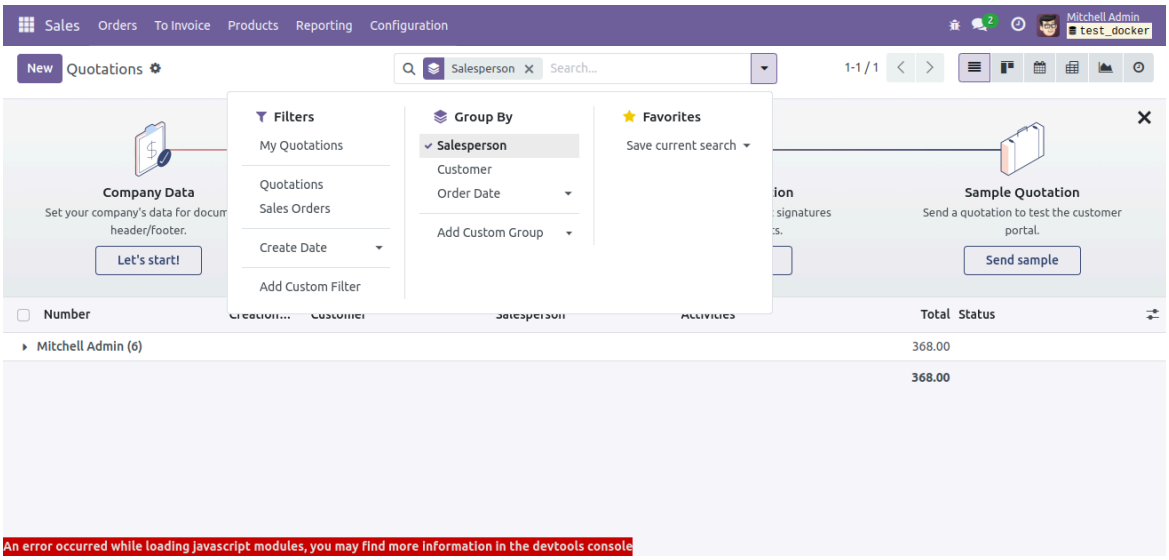
Here we have added a filter for getting all archived records. A **domain** should be given for the filter because filtering will be done based on this domain, i.e. here the filter will fetch all the records if the field ‘active’ is ‘False’.

```
<filter string="Archived" name="inactive" domain="[('active', '=', False)]"/>
```

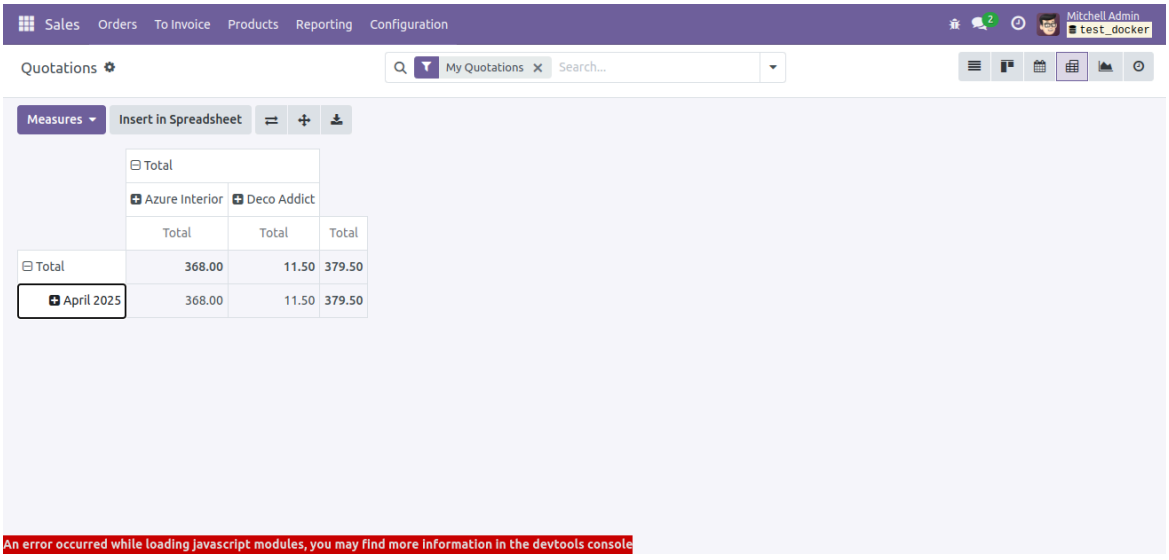


```
<group expand="0" string="Group By">
    <filter string="Salesperson" name="salesperson" domain="[]" context="{'group_
</group>
```

In the above code, you can see that we are using ‘SalesPerson’ as a ‘Group By’ option. Also, the grouping parameter is passed as a **context** as shown in the image.



4) Pivot



```
<record id="view_sale_order_pivot" model="ir.ui.view">
      <field name="name">sale.order.pivot</field>
      <field name="model">sale.order</field>
      <field name="arch" type="xml">
        <pivot string="Sales Orders" sample="1">
          <field name="date_order" type="row"/>
          <field name="partner_id" type="col"/>
          <field name="amount_total" type="measure"/>
        </pivot>
      </field>
  </record>
```

Inside the <pivot> element, you define the structure of the pivot table:

```
<field name="date_order" type="row"/>
```

This field will be used for rows in the pivot table. It implies that the pivot table will group data based on the "date_order" field of the sale order.

```
<field name="partner_id" type="col"/>
```

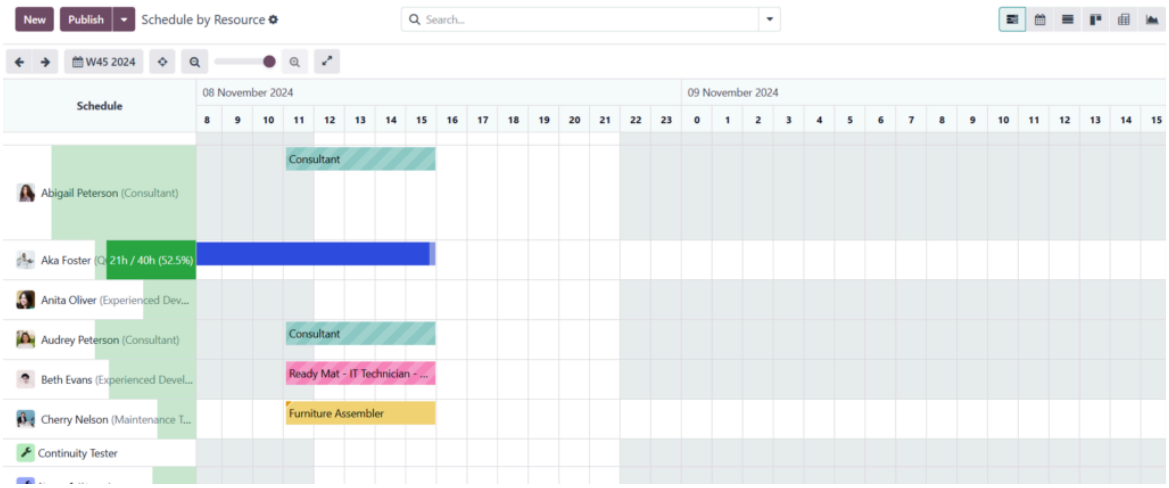
This field will be used for columns in the pivot table. It implies that the pivot table will organize data horizontally based on the "partner_id" field of the sale order.

```
<field name="amount_total" type="measure"/>
```

This field represents the numerical values to be aggregated in the pivot table. It implies that the pivot table will calculate the sum or other measures of the "amount_total" field for each combination of date_order and partner_id.

5) Gantt

Gantt view is only supported in the Odoo enterprise edition because it requires the installation of the ‘web_gantt’ module, which is available in the enterprise edition.




```
<record id="view_gantt_view_gantt" model="ir.ui.view">
    <field name="name">view.gantt.view.gantt</field>
    <field name="model">view.gantt</field>
    <field name="arch" type="xml">
      <gantt date_start="start_date" date_stop="end_date"
          default_group_by="partner_id" string="Customer">
        <field name="partner_id"/>
        <field name="start_date"/>
        <field name="end_date"/>
      </gantt>
    </field>
 </record>
```

6) Activity

From the activity view of Odoo we get information on all activities that are linked to our records. The view is like a chart, It contains rows and columns and all the records are in rows and the activities are in columns.

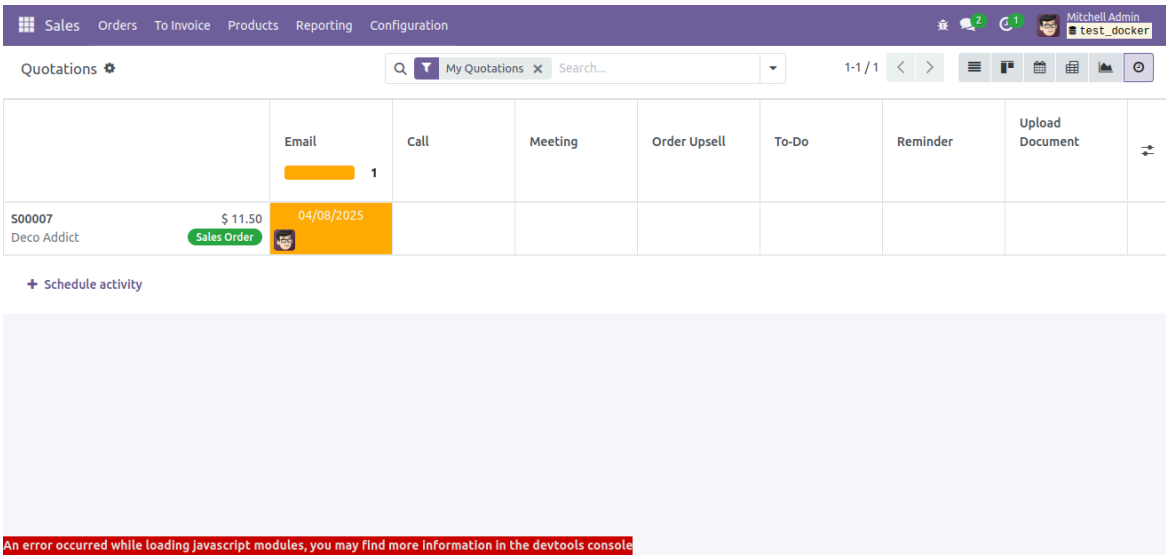
First, we must add "mail" to the dependencies - 'depends': ['mail'],

We need to inherit the mail.thread and mail.activity.mixin models after adding dependencies to the manifest.

Next we can define the activity view of the model sale order. Where the <activity> is the root element and inside the template tag we can give fields that want to be shown

in the activity view as shown in the following image:

```
<record id="sale_order_view_activity" model="ir.ui.view">
      <field name="name">sale.order.activity</field>
      <field name="model">sale.order</field>
      <field name="arch" type="xml">
        <activity string="Sales Order">
          <field name="currency_id"/>
          <templates>
            <div t-name="activity-box" class="d-block">
              <div class="d-flex justify-content-between">
                <field name="name" display="full" class="o_text_block o_text_bold
                <div class="m-1"/>
                <field name="amount_total" widget="monetary"/>
              </div>
              <div class="d-flex justify-content-between">
                <field name="partner_id" muted="1" display="full" class="o_text_bl
                <div class="m-1"/>
                <field name="state" widget="badge"
                  decoration-info="state == 'draft'" decoration-success="state ==
              </div>
            </div>
          </templates>
        </activity>
      </field>
    </record>
```



▼ Controller

Controller is a Python class that manages the interactions between the backend logic (models) and the frontend views (web pages). Controllers handle incoming requests from the user's browser, process the requests by interacting with models or other components, and generate responses to be displayed on the web pages.

```
from odoo import http

@http.route('/product', type='http', auth="user", website=True)
def product_details(self):
  product_id =request.env['product.template'].sudo().search([])
  values = {
     'product': product_id,
  }
  return request.render('product_details_template', values)
```

Controllers in Odoo are inherited from the "Controller" class, as evident from the provided code snippet. The @http.route() decorator within the controller defines various details, including the URL to redirect, request type (HTTP or JSON), authentication parameters to specify access, and the 'website' parameter to link the controller with a page (accepting True or False values).

Moreover, within the controller function definition, utilizing request.render() allows us to specify the template for rendering when redirecting to the designated URL.

▼ Pros & Cons of Odoo Studio

Pros

Drag-and-Drop Interface:

Odoo Studio offers a simple interface that allows you to edit existing apps or develop new ones easily. You can easily add and rearrange objects such as fields and views by dragging them into place. This eliminates the requirement for coding by lowering the technical barrier to modification.

Report Customization & Dashboard Design: Users can modify pre-made reports or make brand-new ones with Odoo Studio. Reports can be customized to meet the requirements of your company by using filters and changing layouts. You may also create dynamic dashboards with live KPI metrics and visualizations.

Cons

Compatibility Issues: Changes made with Odoo Studio may cause issues when upgrading to new Odoo versions, as there are no migration scripts for custom modifications.

Risk from Non-Technical Users: By enabling non-technical users to make changes, there's a risk of introducing errors or incorrect configurations that could impact the system.

If you uninstall the Studio application from your Odoo platform by accident or with sense the entire application as well as the customization that has been configured by you will be lost.

▼ Webhooks

Webhooks are a powerful tool that allows us to receive real-time updates from external systems and perform or trigger some actions within Odoo.

To set up a webhook, we need a URL that external software systems can send data to when a specific event occurs. This URL, along with the triggering event, is shared with the external system. When the event happens, the system sends data to the URL. In Odoo, this URL should point to a controller that can process the received data.

Once an event happens in the external system, it sends a response to an Odoo controller via a specific URL. The data, usually in JSON format, is received and

processed in the controller. From there, we can parse the data and perform various actions based on the information provided.

▼ Functional

▼ Product Types

Storable Products

Physical, tangible goods stored in inventory with a lasting presence, tracked by stock levels, and often managed with serial or lot numbers.

Examples: Electronics (e.g., laptops, smartphones), raw materials (e.g., metal, plastic), finished goods (e.g., cars, appliances).

Consumable Products

Physical goods consumed quickly or used in processes, not requiring detailed inventory tracking. Odoo assumes sufficient stock, allowing deliveries regardless of actual inventory levels.

Examples: Office supplies (e.g., pens, paper), packaging materials (e.g., wrapping paper, tape), small production items (e.g., screws, nails).

Service Products

Intangible, non-physical products or activities provided to customers, not stored or tracked in inventory.

Examples: Consulting services, subscription plans (e.g., software subscriptions), maintenance packages.

▼ How to Create PO from SO ?

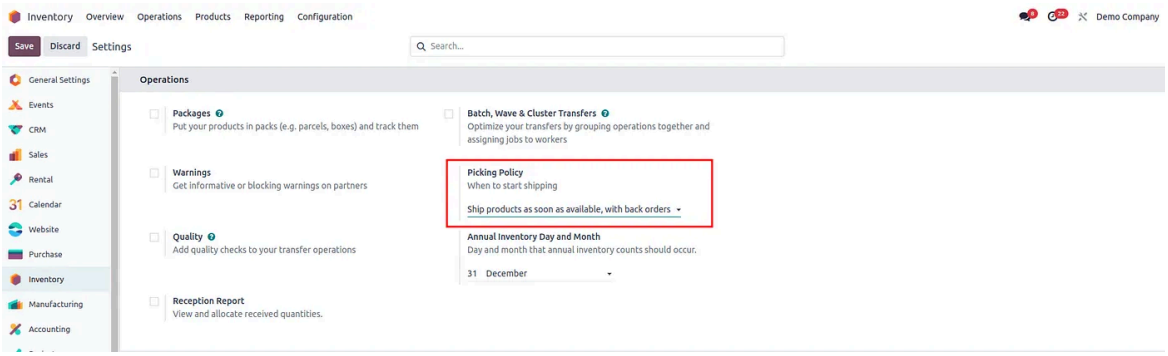
1. Goto → Settings → Inventory → **Multi-Step Routes**
2. Goto → Inventory → Configuration → Routes → Unarchive route Replenish on Order (MTO).
3. Goto → Inventory → Products → Inventory Tab → Check Buy & MTO. Goto Purchase → Add vendor when system generates the purchase order, it will select the first vendor in the list by default.

4. Create a sale order for that product. Once you have created a sale order go to the delivery part of the sale order You can see there, the delivery is waiting for **Another Operation.**
5. Go to **Purchase → Requests for Quotation.** There you can find the generated Purchase quotation against respective sale order. **The** purchase order will be created against the first vendor in vendors tab of the product. The **Source Document** field indicates the related sale order.
6. Confirm purchase order → Receive the products → Validate picking
7. After completing the purchase order, go the corresponding sale order. The state of the sale order will be changed from **Waiting for Another operation** to **Available** state. Without confirming PO it is not possible to create SO.

▼ Picking Policy & Shipping Policy

Picking Policy: How the product is being taken from inventory when customer places order

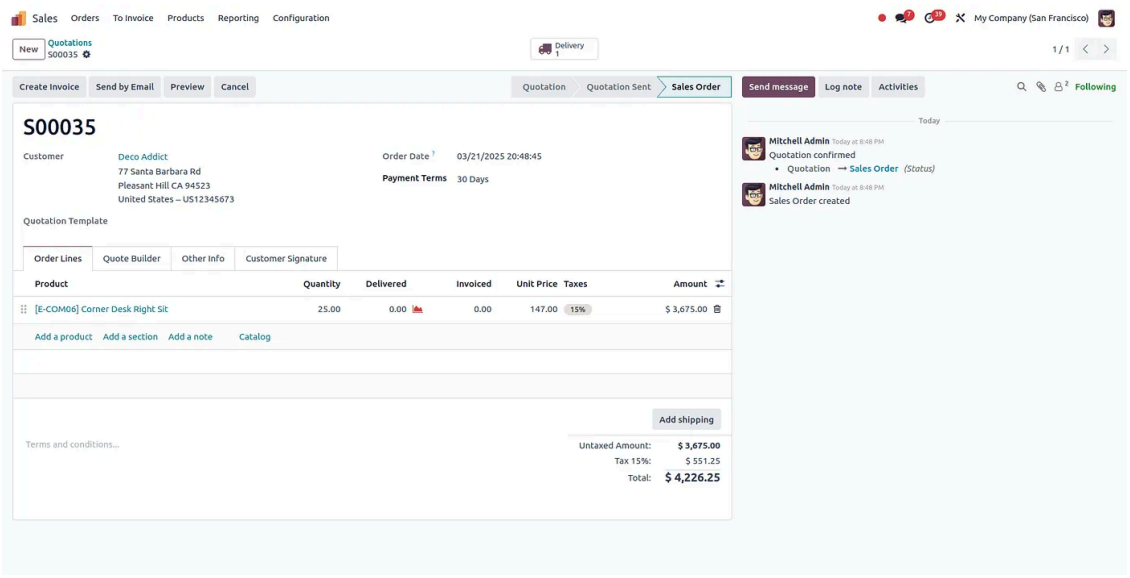
Configuring Picking Policies



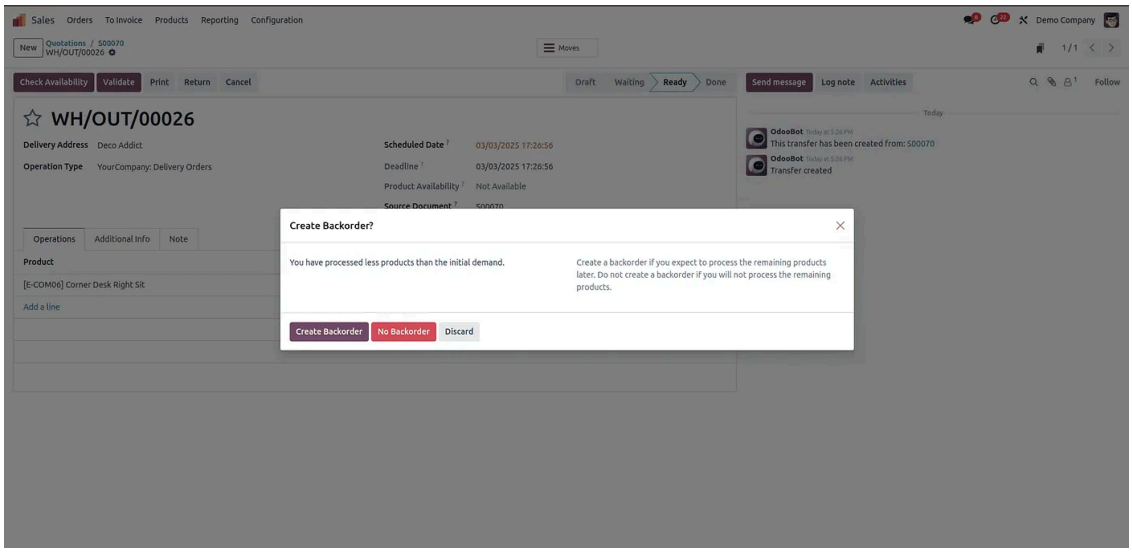
1) Ship Products as Soon as Possible, with Backorders

With this **picking policy enabled**, products are shipped as soon as they become available. For example, if an order is placed for **10 units**, but only **7 units** are in stock, the available **7 units** will be shipped immediately, while the remaining **3 units** will be placed on **backorder**.

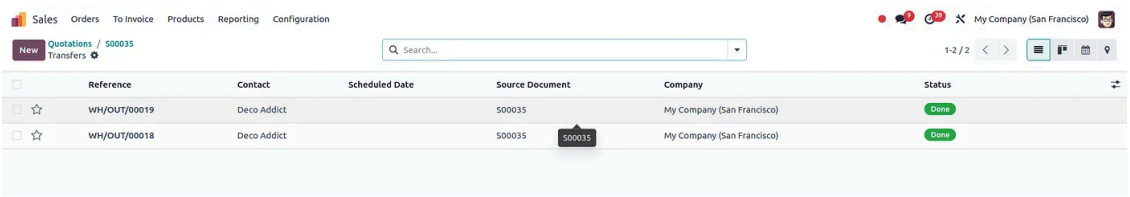
In the **Ask** scenario, for example, if an order is placed for **25 units** of the product '**Corner Desk Right Sit'**, but **20 units available of the product Corner Desk Right Sit** the user will be prompted to decide whether to create a backorder for the unavailable quantity.



During the delivery process, the transfer status changes to **'READY'** once the available items are prepared for shipment. After validating the delivery, a pop-up appears prompting the user to decide whether to create a backorder for the remaining products.



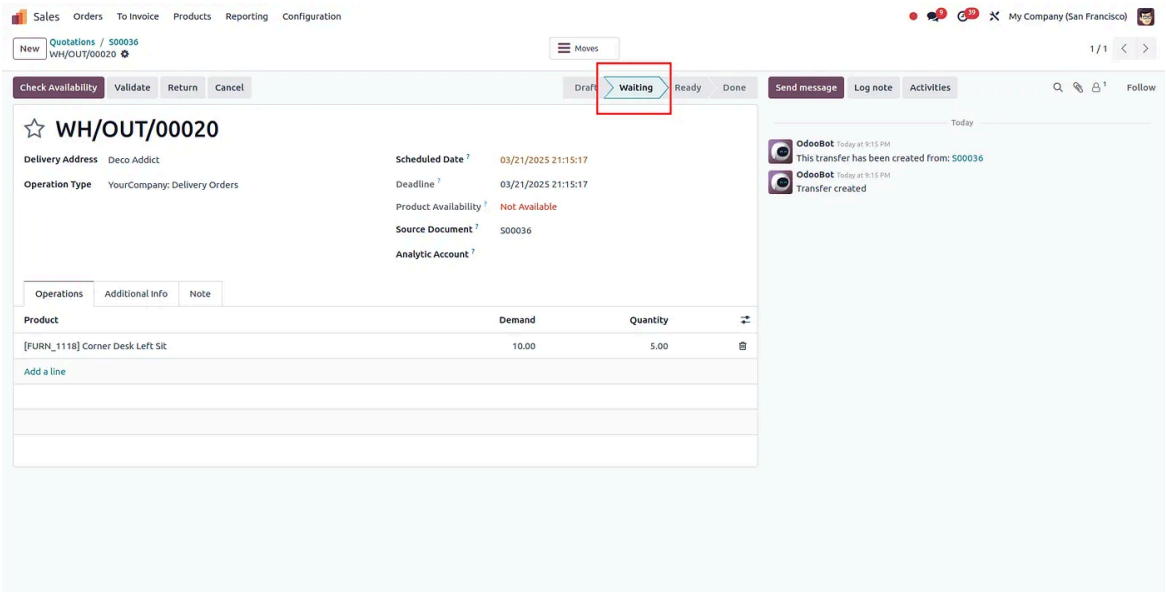
Once the product quantity is restocked, the backorder will be processed, and the remaining items will be shipped.



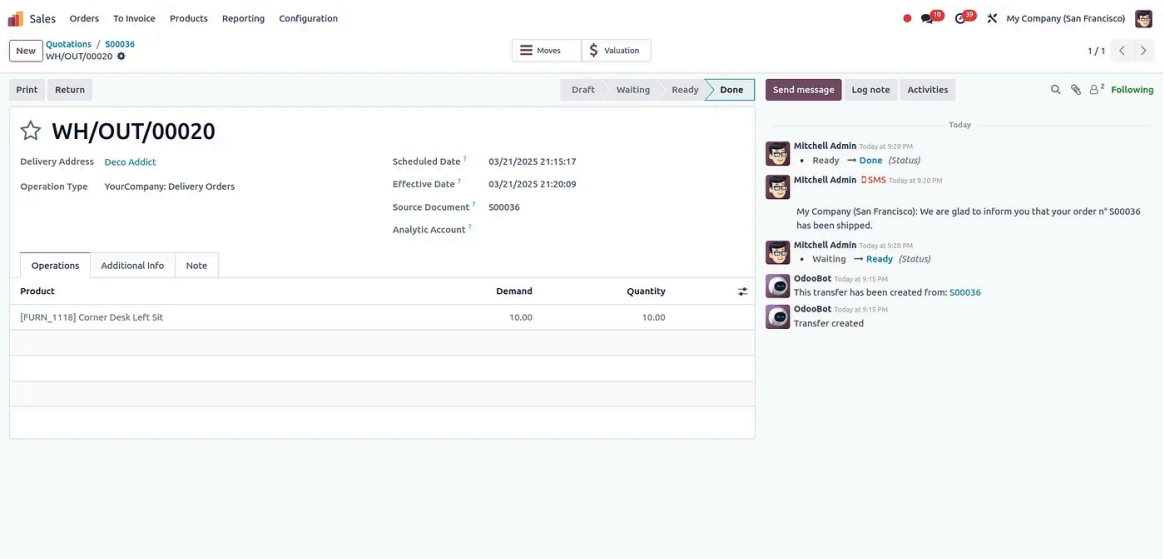
Ship all products at once

When a customer orders more products than are available in stock, the "**Ship all products at once**" option holds the entire order until the required stock is replenished. This ensures that the customer receives all items in a single shipment, avoiding partial deliveries. The order will only be processed once inventory is fully restocked to fulfill the requested quantity, guaranteeing a complete delivery.

On the **Delivery** page, the status will be '**Waiting**', and it will only change to '**Ready**' once the products are restocked.

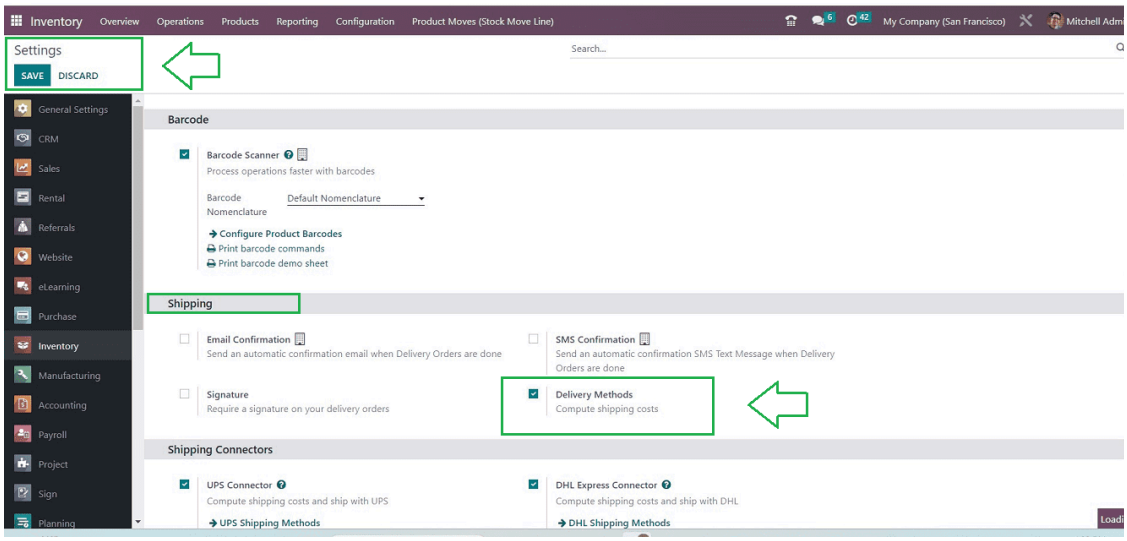


Since the **picking policy** is set to '**Ship all products at once**,' the sale order will remain on hold until the full requested quantity is available in stock. Since the ordered quantity exceeds the available stock, the order cannot proceed. Once the stock is replenished, all products will be shipped together as per the configured policy. After the stock update, the **delivery status** will change to '**Ready**,' allowing validation and shipment of the entire order at once.

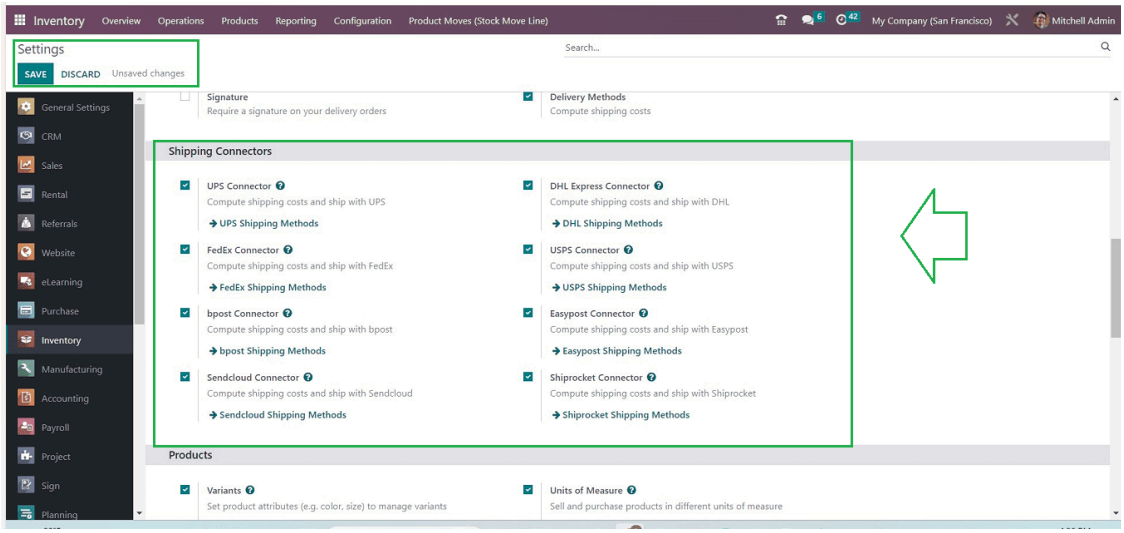


▼ Shipping Policies(Delivery Methods)

In the Inventory module, configuring and controlling Delivery methods/ Shipping Policies is simple to perform. Products can be shipped directly to customers, or you can use a third-party delivery service.



After enabling the “Delivery Methods” option available in the “Shipping” tab, you can enable the required shipping connector services from the “Shipping Connectors” tab, as illustrated below.



▼ Describe the workflow of the purchase module & it’s features.

1. **Configuration (Initial Setup):**

- **Vendor Setup**: Before starting, you need to configure vendors in the system. Go to the *Purchase* module, select *Vendors* under the *Purchase* menu, and create vendor profiles with details like name, contact info, and payment terms.
- **Product Setup**: Define the products you intend to purchase. Navigate to *Products* under the *Purchase* menu, create product records, and specify details like product type, cost, and vendor (if specific to a supplier).
- **Purchase Settings**: Adjust settings (e.g., Purchase Agreements, 3 way matching, etc) under *Configuration > Purchase > Settings* to align with your business needs.

2. **Request for Quotation (RFQ) Creation**

- The process begins with creating a *Request for Quotation (RFQ)*.
- Navigate to *Purchase > Purchase Orders* and click *Create*.
- Select a vendor, add products (with quantities and prices), and set delivery dates or terms.
- Save the RFQ. At this stage, it’s in the "Draft" state and can be modified.

3. **Send RFQ to Vendor**

- Once the RFQ is ready, click *Send by Email* to share it with the vendor. Odoo generates an email template with the RFQ details (customizable if needed).

- Alternatively, you can print the RFQ and send it manually.
- The status changes to "RFQ Sent."

4. **Vendor Response and Confirmation**

- After the vendor responds (e.g., with a price confirmation or adjustments), update the RFQ in Odoo if necessary.
- When ready, click *Confirm Order* to convert the RFQ into a *Purchase Order (PO)*. The status changes to "Purchase Order," and it becomes a binding commitment.

5. **Receipt of Goods**

- Once the vendor delivers the goods, go to *Purchase > Receipts* (or find the linked receipt under the PO).
- Open the incoming shipment document tied to the PO, verify the delivered quantities against ordered quantities, and click *Validate* to confirm receipt.
- If partial delivery occurs, Odoo tracks the remaining quantities automatically for backorders.

6. **Vendor Bill Creation**

- After validating the receipt, Odoo can generate a *Vendor Bill* automatically (depending on settings).
- Go to *Purchase > Vendor Bills*, open the draft bill linked to the PO, and verify details like quantities, prices, and taxes.
- Adjust if necessary (e.g., if the vendor invoice differs), then click *Confirm* to post the bill to your accounting system.

7. **Payment Processing**

- Once the bill is confirmed, proceed to *Accounting > Vendor Payments* or use the *Register Payment* option on the bill.
- Specify the payment method, amount, and date, then validate the payment.
- Odoo reconciles the payment with the bill, marking it as "Paid."

8. **Tracking and Reporting**

- Throughout the process, you can track the status of RFQs, POs, receipts, and bills via dashboards or smart buttons on the PO (e.g., *Delivery*, *Invoices*).
- Use *Reporting > Purchase Analysis* to generate insights on spending, vendor performance, or delays.

Key Features of Purchase

- **Convert RFQ to Order:** Easily convert quotations into purchase orders with a single click.
- **Email Order:** Enables users to send orders directly to vendors via email.
- **Product Pricing and Tax :** The purchase module facilitates the addition of multiple products to purchase orders, allowing users to specify prices and apply relevant taxes.
- **Variants Grid Entry:** Add product variants to your purchase orders with a grid, or matrix, displaying all the possible combinations of a product's attributes (e.g. sizes, colors).
- **Manage incoming products:** Keep track of your stock and determine quantity and locations for each batch of items you receive.
- **Purchase Dashboard:** The Purchase Dashboard in Odoo offers a centralized hub for monitoring procurement activities, providing real-time insights into purchasing performance and vendor relationships.
- **Vendors:** Odoo vendor management system provides a centralized hub for storing and managing vendor information, facilitating personalized.
- **Inventory forecasts:** Get forecasts of product availabilities based on confirmed sales orders, purchase orders or manufacturing orders as well as internal moves.

▼ Describe the workflow of manufacturing module & it’s features.

1. **Setup and Configuration:**

   Before starting the manufacturing process, foundational elements must be configured:

- **Products**: Define the products you manufacture (finished goods) and their components (raw materials) in the "Products" module. Assign attributes like name, type (storable product), unit of measure, and cost.

- **Bill of Materials (BoM)**: Create a BoM for each manufactured product. The BoM lists the raw materials, quantities, and operations required to produce the finished product. You can specify whether it's a simple production or involves sub-assemblies.
- **Work Centers**: Set up work centers (e.g., machines, workstations) where production tasks occur. Define their capacity, working hours, and costs.
- **Routing**: Define the sequence of operations (e.g., cutting, assembling, packing) and link them to work centers. This is optional and used for advanced manufacturing processes.
- **Inventory**: Ensure raw materials are tracked in the Inventory module and stocked in warehouses or locations.

2. **Manufacturing Order Creation**

- **Manual Creation**: Go to the Manufacturing module and create a "Manufacturing Order" (MO). Select the product to manufacture, specify the quantity, and assign a BoM. Odoo automatically populates the required raw materials and operations based on the BoM.
- **Triggered by Sales**: If integrated with the Sales module, a manufacturing order can be automatically generated when a sales order is confirmed (e.g., for make-to-order products).
- **Planned Orders**: Use the Material Requirements Planning (MRP) feature to generate manufacturing orders based on demand forecasts, stock levels, or reorder rules.

3. **Planning and Scheduling**

- **Scheduling**: Odoo schedules the manufacturing order based on the availability of raw materials, work center capacity, and deadlines. You can manually adjust the planned date or let Odoo optimize it.
- **Work Orders**: If routing is defined, Odoo breaks the manufacturing order into work orders (specific tasks) assigned to work centers. These can be scheduled sequentially or in parallel.
- **Resource Availability**: Check the availability of raw materials in the warehouse. If stock is insufficient, Odoo can trigger purchase orders or suggest production of sub-assemblies.

4. **Production Execution**

- **Start Production**: Once the manufacturing order is confirmed, move to the "Produce" stage. Odoo provides a tablet-friendly interface for shop floor workers to record progress.
- **Consume Materials**: Raw materials listed in the BoM are consumed from inventory. You can manually adjust quantities if there's scrap or overconsumption.
- **Record Operations**: For each work order, log the time spent, progress, and any additional details (e.g., machine issues). This step is optional if no routing is defined.
- **Quality Checks**: Integrate with the Quality module to perform quality control at specific stages (e.g., inspecting raw materials or finished goods).

5. **Completion and Inventory Update**

- **Finish Production**: Once all operations are completed and the product is manufactured, mark the manufacturing order as "Done."
- **Inventory Adjustment**: Odoo automatically deducts the consumed raw materials from inventory and adds the finished goods to stock. This is tracked in real-time using the Inventory module.
- **Scrap Management**: Record any defective items or waste as scrap, which adjusts inventory accordingly.

6. **Analysis and Reporting**

- **Traceability**: Use Odoo's lot/serial number tracking to trace raw materials and finished products through the production process.
- **Cost Analysis**: Review production costs, including material costs, labor (based on time spent), and overheads (from work centers).
- **Performance Reports**: Generate reports like Overall Equipment Efficiency (OEE), production times, or delays to optimize future processes.

Example Workflow

1. A customer orders 100 chairs (Sales Order).
2. Odoo generates a Manufacturing Order for 100 chairs based on the BoM (e.g., 200

wooden legs, 100 seats, 400 screws).
3. The system checks inventory: 150 legs and 80 seats are in stock, so it triggers a Purchase Order for 50 legs and 20 seats.
4. Once materials arrive, production begins at the "Assembly" work center.
5. Workers assemble the chairs, consume materials, and mark the order as "Done."
6. Inventory updates: 100 chairs added, raw materials deducted.
7. The chairs are shipped to the customer, and costs are recorded in Accounting.

Key Features of Manufacturing

1. Manage:

With Odoo Manufacturing implementation, it helps the manufacturers to access the business information in one click.

Manufacturing orders: Manage your products into assembly lines or manual assembly.

Work orders: Launch the production of items needed in the final assembly of your products.

Repair orders: Manage repairs of items under warranty or as a service.

2. Schedule & plan:

Plan manufacturing: Get a clear view of your whole planning and easily reschedule manufacturing.

Organize work orders: Have access to all available resources and plan ahead for your production.

Manage Bill of Materials: Keep track of the availability of items in stock and production time.

WorkCentre Capacity: MRP II scheduler using capacities and schedules of WorkCentre.

3. Define Flexible Master Data

Before manufacturing takes place, routing needs to be properly defined. Routing is basically the method to create the product using the materials outlined in the BOM. This may involve multiple stages and multiple work centers (the groups of

individuals/locations where production activity takes place). Odoo manufacturing does the following:

Create multi-level Bills of Materials: Set a Bill of Materials within another in order to manufacture components of a product in another Bill of Materials.

Optional routing: Create new routines for work orders in order to sequence your production depending on the routing used.

Version changes: Allow your products to evolve and add configurable options when creating orders.

Phantom of Bill of Materials: Create phantom BoM to manufacture and sell products in kits or to build replacement parts.

4. Quality

Odoo manufacturing inspects quality checks for purchased materials before production, in the middle of manufacturing stages, or as final inspections. On account of any problems, the quality check systems in manufacturing ERP helps narrow down the issue, thereby improving the product quality and customer satisfaction.

Control Points: Automatically trigger quality checks for the manufacturing department.

Quality Checks: Deploy your statistical process control easily with checks.

Quality Alerts: Organize your work using the Kanban view of quality alerts.

5. Maintenance

A manufacturing ERP can also allow maintenance orders to take place for repairs and other fixes. Such orders will organize requests and be assigned to the maintenance team. With ERP systems such as Odoo, these systems can also automate preventive maintenance (regularly scheduled maintenance on items so that they’re less likely to fail) by computing statistics such as average time to failure.

Preventive Maintenance: Trigger maintenance requests automatically based on KPIs.

Corrective Maintenance: Trigger corrective maintenance directly from the control center panel.

Calendar: Schedule maintenance operations with a calendar.

Statistics: Get all maintenance statistics computed for you: *MTBF*

6. WorkCentre Control Panel

Tablets: Set tablets at every work center to organize their work efficiently.

Record production: Register productions, scan products, lots, or serial numbers.

Worksheets: Display worksheets directly on the WorkCentre with instructions for an operator.

Misc. Operations: Scrap products, create quality alerts, and perform checks, right from the WorkCentre.

Alerts: Use alerts to show changes or quality checks to the operator.

▼ CRM

▼ Workflow of CRM & it’s features:-

CRM in Odoo can be explained as a series of events that start with identifying a Lead (a future sale possibility) and passes through different stages like Opportunity, Quotation, Sale Order and actual sale (invoice generation and payment). Odoo integrates Customer Management module along with these processes to accomplish effective Customer Relationship
Management. The basic steps involved in tracking a sale can be listed as below

Lead: - A possible future sale, it may be created just because a user enquired a product

Opportunity: - More possibility of sale. Here onwards the organization may appoint a person to
follow-up the customer.

Pipeline: - it is a convenient mechanism provided by Odoo to track leads & Opportunities. You can create
many stages based on the possibility of the Sale and track more effectively.

Quotation: - once the opportunity is Won, then next level is to make a quotation and send to
the customer.

Next, Quotation changes to Sale Order and then to Invoice Generation and Payment.

▼ Notes:-

1. How do I schedule an activity for an opportunity from the main CRM dashboard (in Kanban view) ?

   Click the ‘clock’ icon on the Kanban card of the opportunity

2. In the ‘Extra Information’ tab of an opportunity form, what sections of information can be found ?

   ‘Customer Information’, ‘Marketing’, and ‘Tracking’

3. What happens to the probability score of an opportunity as it moves through the pipeline?

   It automatically adjusts accordingly.

4. If the probability has been manually changed, how can the probability return to the Odoo probability prediction?

   By clicking the ‘gear’ icon beside the probability field.

5. In order to add a single salesperson to multiple sales teams, what feature *must* be activated in the CRM settings?

   Multi Teams

6. Can I edit opportunities after they have been marked as Won ?

   Yes, at any time, for any reason.

7. Can I restore and edit opportunities that have been marked as Lost ?

   Yes, I can click into opportunities at any time to edit them. To restore opportunities, I can click the ‘Restore’ button.

8. How do I enable Leads in Odoo CRM ?

   CRM > Configuration > Settings > Leads > Tick Checkbox.

9. What is absolutely required to save a lead in Odoo CRM?

   Lead title

10.

Key Features of CRM

- **Lead management.** The tracking and nurturing of leads are becoming more practical.

- **Sales pipeline management.** Visually represents the sales stages.
- **Automated lead generation.** Automatic capture of lead from all sources.
- **Activity tracking.** Tracking of all the lead interactions and tasks done over a period.
- **Customizable sales stages.** Allows you to customize the sales pipeline for your specific business operations.
- **Reporting and analytics.** In-depth sales reporting and performance analytics.
- **Email Integration.** Easily links with emails towards the communication of clients.
- **Contact management.** It encompasses the structuring and maintenance of comprehensive customer profiles.
- **Opportunity management.** It is about surfacing and ranking potential sales opportunities.
- **Marketing campaign integration.** It leads to the confluence of the marketing with the sales systems.
- **Lead Enrichment: I**t is the procedure to collect a customer's contact details from their email ID alone.
- **Lead mining:- It** is a feature that *allows CRM users to generate new leads directly into their Odoo database*

Example Workflow in Practice

1. A website form captures a lead for a product inquiry.
2. The lead is assigned to a salesperson, who schedules a call and qualifies the lead.
3. The lead is converted into an opportunity and moved to the "Proposal" stage.
4. A quotation is created and sent to the prospect, who approves it via the customer portal.
5. The opportunity is marked as "Won," and the sales order is processed.
6. Post-sale, the customer receives a satisfaction survey, and the CRM tracks future upselling opportunities.
7. The sales manager reviews pipeline reports to assess team performance.

Full Example Flow:

1. **Lead**: John visits your website and signs up for a demo.

2. **Opportunity**: You qualify John — he has a need, budget, and decision power.
3. **Pipeline**: John moves through your sales stages: Contacted → Demo Scheduled → Proposal Sent.
4. **Quotation**: You send him a formal quote.
5. **Closed Won**: John accepts, and the deal is closed. Congratulations!

▼ Sales

▼ BackOrder

If we decide to process the delivery order partially (e.g., a part of the products can't be shipped yet due to an unexpected event), Odoo automatically asks if you wish to create a **backorder**.

- Create a backorder if you expect to process the remaining products later.
- Do not create a backorder if you will not supply/receive the remaining products.

1. Goto Inventory > Configuration > Operation Types > Delivery Orders > Create BackOrder.

   Three options for managing backorders in a field:

   - **Ask**: where users are asked whether they wish to place a backorder for remaining goods
   - **Always**: The backorder will be automatically generated for the remaining goods
   - **Never**: Remaining products will be canceled

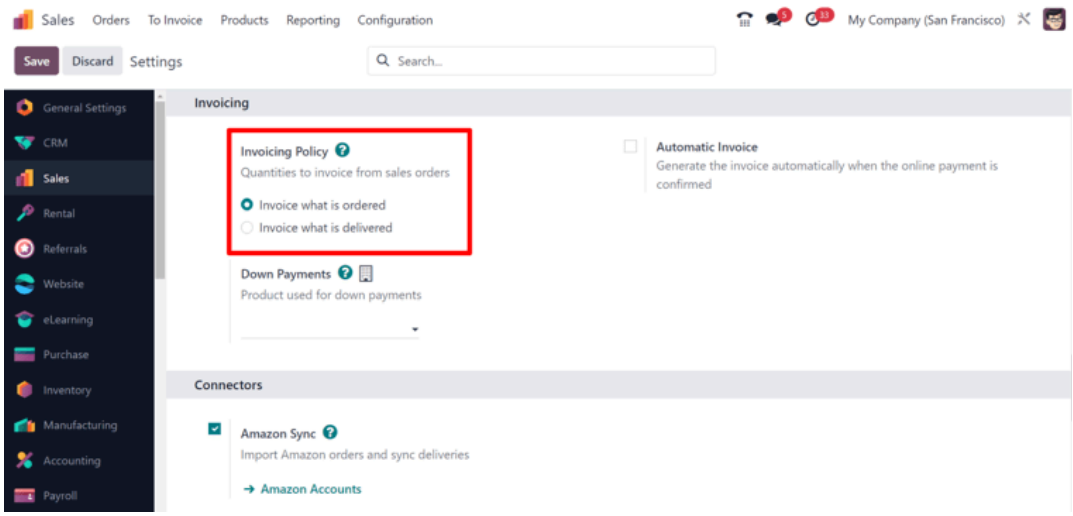
▼ What is Delivery Lead Time in odoo ?

The number of days from the date the sales order (SO) is confirmed to the date the products are shipped from the warehouse

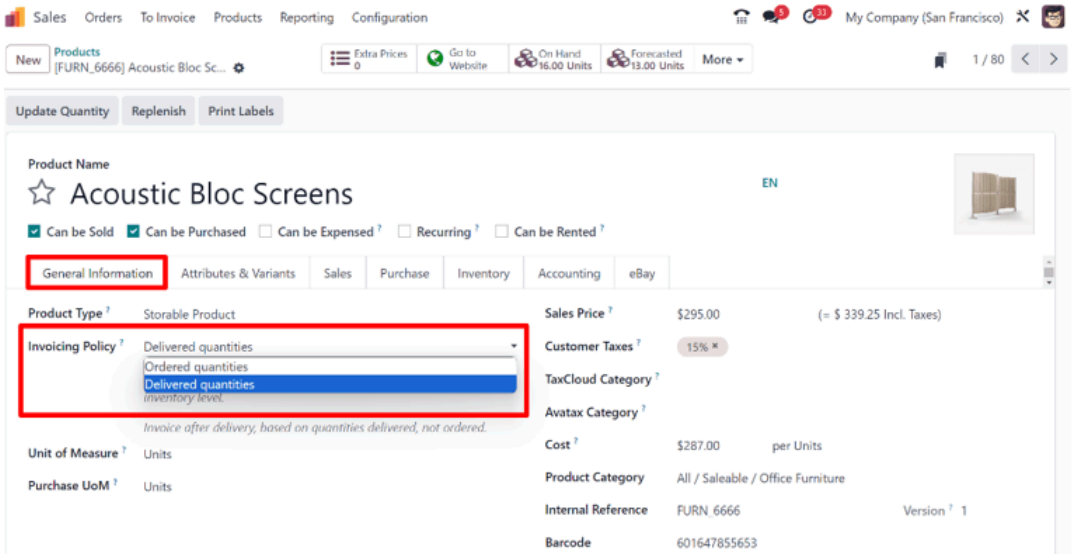
▼ Invoicing policy

- **Invoice what is ordered**: No impact on the basic sales flow. An invoice is created as soon as a sale is confirmed.
- **Invoice what is delivered**: Minor impact on sales flow, because the delivered quantity needs to be manually entered on the sales order. Or, the *Inventory* app can be installed and used to confirm the delivered quantity before creating an invoice with the *Sales*app.

1. Go to **Sales app ▸ Configuration ▸ Settings**, and under the **Invoicing** heading, select an**Invoicing Policy** rule: **Invoice what is ordered** or **Invoice what is delivered**.



2. **Sales app ▸ Products ▸ Products dashboard**, locate the **Invoicing Policy** option located under the **General Information** tab. It can be changed manually using the drop-down menu.



▼ Workflow & it’s features:-

- **Lead generation:** Identify potential customers and capture their information as leads.

- **Quotation:** Create a quotation for the customer, outlining the proposed products or services, their prices, and terms.
- **Sales order:** Convert the quotation into a sales order once the customer confirms the purchase.
- **Delivery:** Deliver the products or services to the customer, updating the sales order with delivery information.
- **Invoicing:** Generate an invoice based on the sales order, detailing the amounts owed by the customer.
- **Payment:** Register the payment received from the customer, reconciling it with the invoice.
- **Delivery Lead Time**: the number of days from the date the sales order (SO) is confirmed to the date the products are shipped from the warehouse

▼ Key Features of Sales

- **Mobile Capabilities :** Use Odoo's mobile interface to sell on the road.
- **Data Entry Reduction:** Easily generate quotes, sales orders, and invoices from a single screen with seamless integration with the CRM app, streamlining your sales pipeline management from initial qualification to final closure.
- **Sales Warnings:** Get warnings before sending quotations to specific customers or for specific products.
- **Quotation Builder:** Accelerate the quote creation process by leveraging pre-defined products, price lists, and templates, enabling sales teams to work more efficiently.
- **Quotation Template:** Design custom quotation templates in just a few clicks and reuse them to save time.
- **Electronic Signature:** Sell faster with electronic signatures, allowing your customers to review and sign their quotations online.
- **Variant Grid Entry:** Add product variants to your sales orders using a grid, or matrix, displaying all the possible combinations of a product's attributes (e.g. sizes, colors).
- **Modern User Interface:** A user-friendly interface optimized for salespeople enables quick access to all relevant information in the right places.

- **Customer Portal:** Provide customers with access to an online portal to view their quotes and sales orders, and to track the status of their deliveries in real-time. Customers can modify their orders and confirm them with an electronic signature or online payment.

▼ Purchase

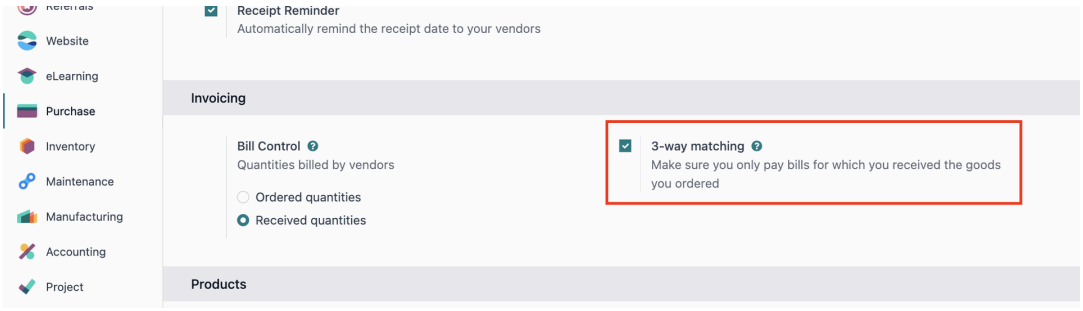
▼ What is 3-way matching ?

3-way matching ensures you pay vendor bills only after receiving the items listed in the purchase order.

▼ How does 3-way matching work in Odoo 17 ?

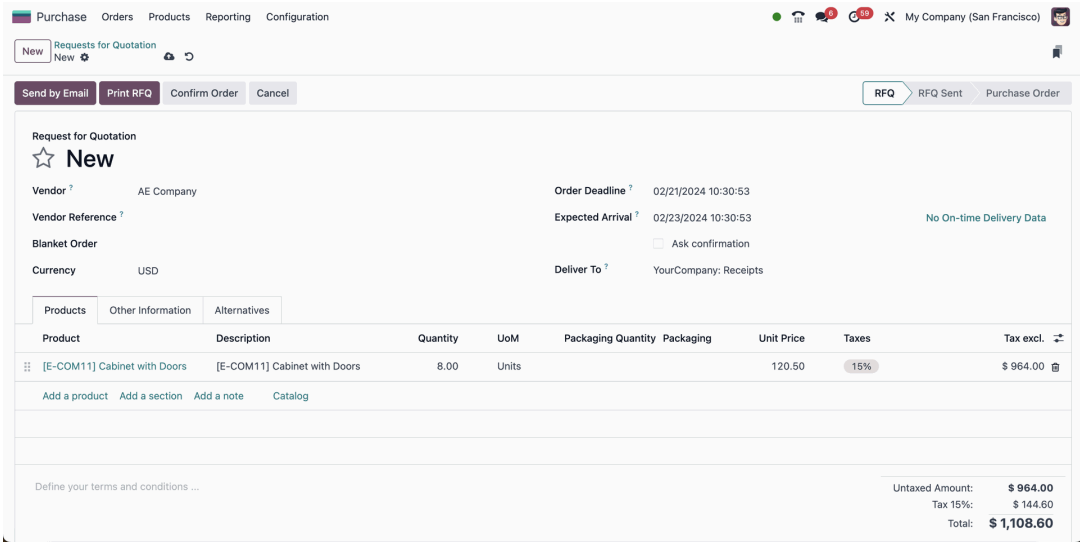
Step 1: Enable 3-way matching in Settings

Go to the Settings of your Purchase App and ensure that the 3-way matching feature is enabled.



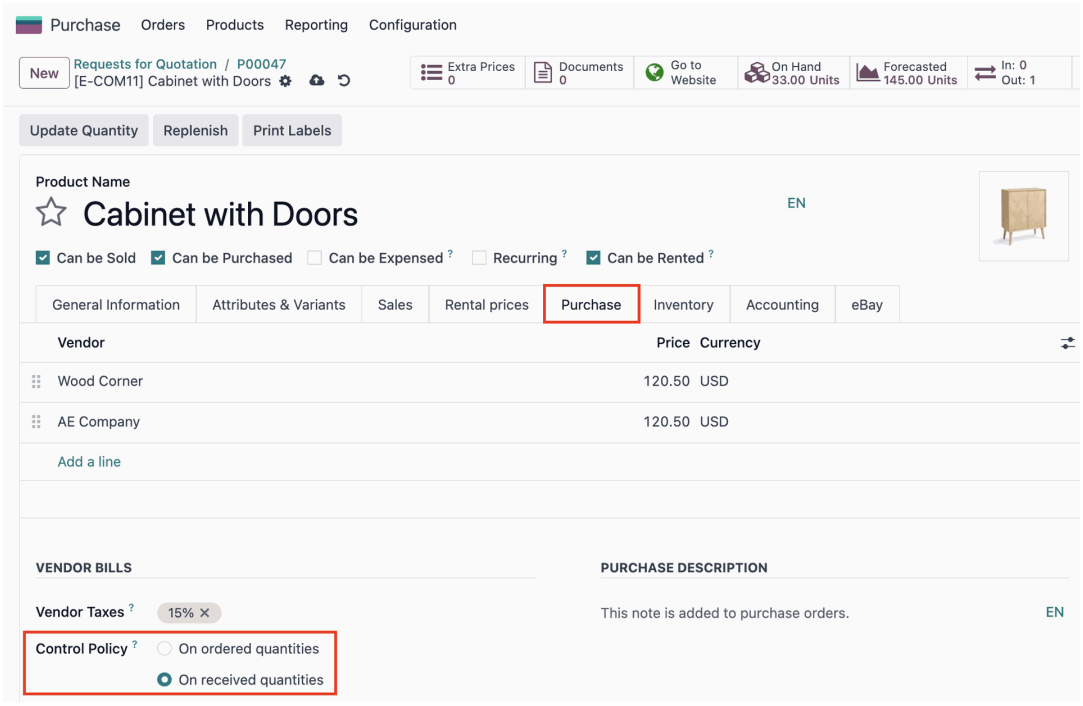
Step 2: Create a purchase order

This is the first step in the 3-way matching process. The purchase order should include details such as the supplier, the items to be purchased, and the quantities and prices.



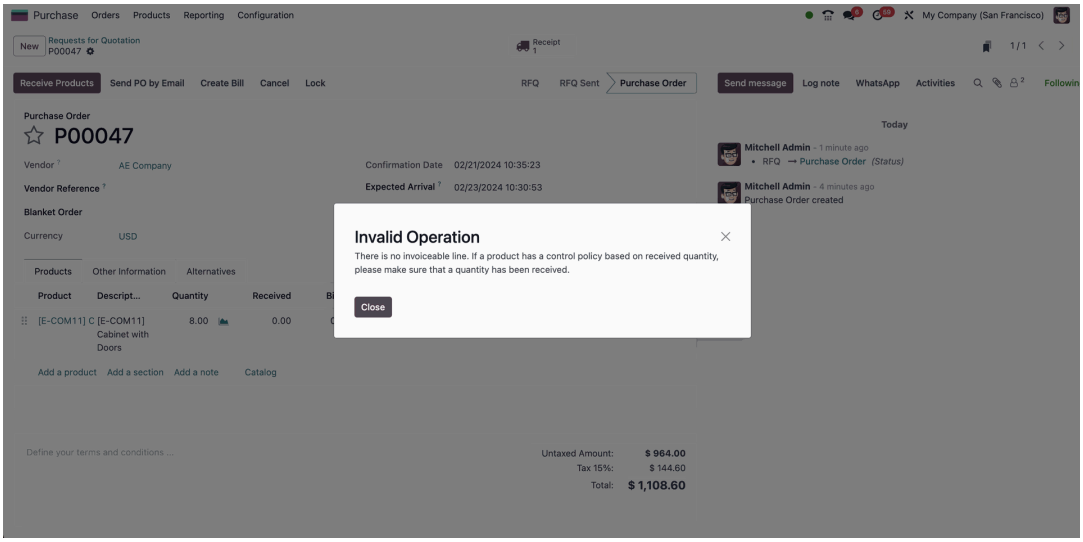
Step 3: Check product Control policy

Once you added the product, click on the internal link to go to the product settings. Choose the “Purchase” tab & make sure that the correct control policy “On received quantities” is selected.



Step 4: Confirm Order & Test 3-way matching

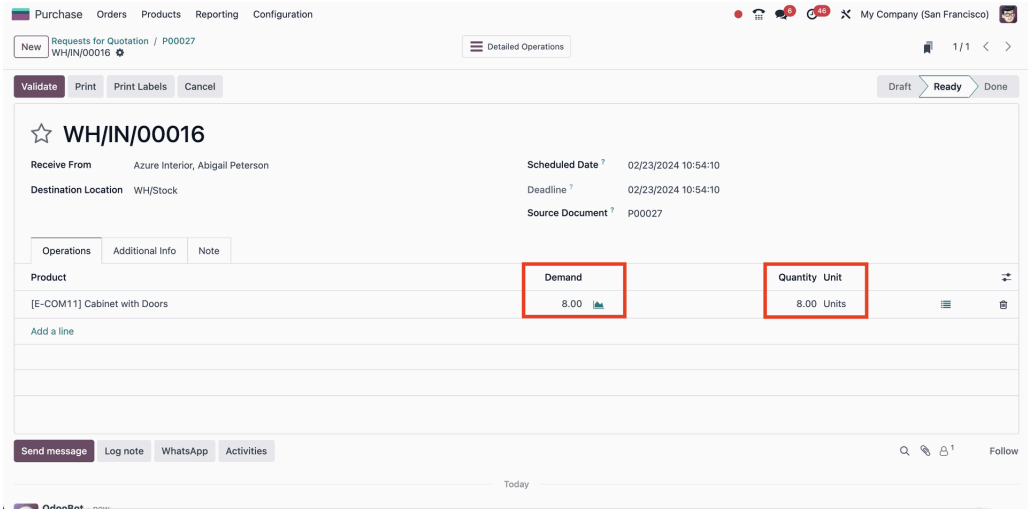
Go back to your Purchase Order and select “Confirm the Order”. Odoo 17 will now sent your RFQ automatically to the vendor. However, since you still have not received the products, you will not be able to create a vendor bill yet thanks to the 3-way matching. Instead, if you try to select “Create Bill” it will show you the following error message.



Step 5: Receive goods

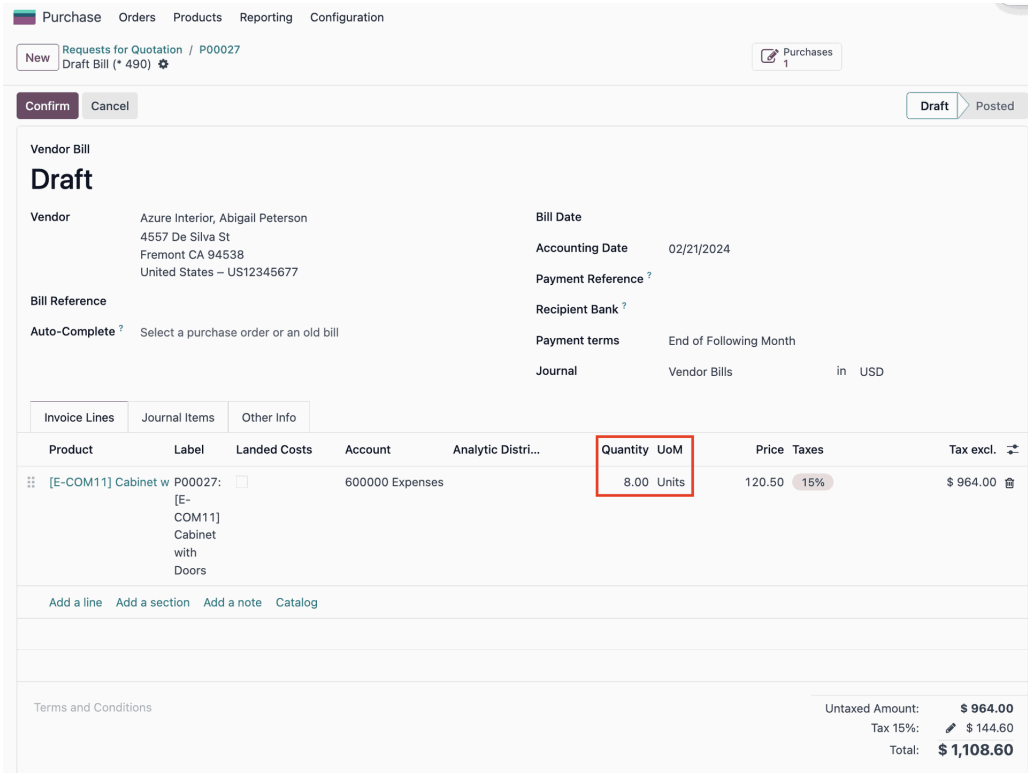
When the goods are received, a receipt should be created in the system to confirm that the correct items were received and in the correct quantities.

- Select “Receive Products”. Here you can check that the demanded quantity and the quantity you will receive match.
- Select “Validate”



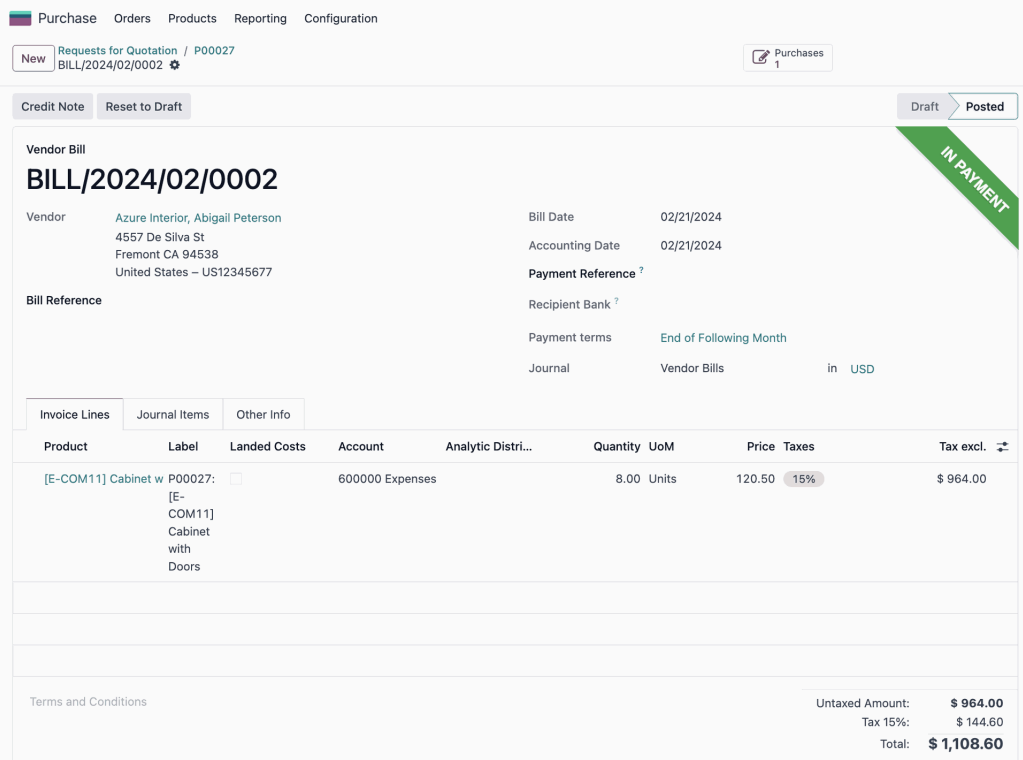
Step 6: Invoice receipt

The supplier will then send an invoice for the goods received. The invoice should be entered into the system and matched to the purchase order and the receipt.



Step 7: Approve & Pay

If there are no discrepancies, the matched data can be approved and the payment can be processed.



▼ What is Blanket Order ?

There are two main types of purchase agreements preconfigured in Odoo: blanket orders and call for tenders.

- Blanket Orders: Blanket orders are long-term purchase agreements between a company and a vendor to deliver products on a recurring basis with predetermined pricing.

  Blanket orders are helpful when products are consistently purchased from the same vendor, but in different quantities, and at different times.

- Call for Tender: A Call for Tender is a special procedure to request offers from multiple vendors to obtain the most interesting price.

▼ Workflow & it’s features:-

1. **Configuration (Initial Setup):**

- **Vendor Setup**: Before starting, you need to configure vendors in the system. Go to the *Purchase* module, select *Vendors* under the *Purchase* menu, and create vendor profiles with details like name, contact info, and payment terms.
- **Product Setup**: Define the products you intend to purchase. Navigate to *Products* under the *Purchase* menu, create product records, and specify details like product type, cost, and vendor (if specific to a supplier).
- **Purchase Settings**: Adjust settings (e.g., Purchase Agreements, 3 way matching, etc) under *Configuration > Purchase > Settings* to align with your business needs.

2. **Request for Quotation (RFQ) Creation**

- The process begins with creating a *Request for Quotation (RFQ)*.
- Navigate to *Purchase > Purchase Orders* and click *Create*.
- Select a vendor, add products (with quantities and prices), and set delivery dates or terms.
- Save the RFQ. At this stage, it’s in the "Draft" state and can be modified.

3. **Send RFQ to Vendor**

- Once the RFQ is ready, click *Send by Email* to share it with the vendor. Odoo generates an email template with the RFQ details (customizable if needed).
- Alternatively, you can print the RFQ and send it manually.
- The status changes to "RFQ Sent."

4. **Vendor Response and Confirmation**

- After the vendor responds (e.g., with a price confirmation or adjustments), update the RFQ in Odoo if necessary.
- When ready, click *Confirm Order* to convert the RFQ into a *Purchase Order (PO)*. The status changes to "Purchase Order," and it becomes a binding commitment.

5. **Receipt of Goods**

- Once the vendor delivers the goods, go to *Purchase > Receipts* (or find the linked receipt under the PO).
- Open the incoming shipment document tied to the PO, verify the delivered quantities against ordered quantities, and click *Validate* to confirm receipt.
- If partial delivery occurs, Odoo tracks the remaining quantities automatically for backorders.

6. **Vendor Bill Creation**

- After validating the receipt, Odoo can generate a *Vendor Bill* automatically (depending on settings).
- Go to *Purchase > Vendor Bills*, open the draft bill linked to the PO, and verify details like quantities, prices, and taxes.
- Adjust if necessary (e.g., if the vendor invoice differs), then click *Confirm* to post the bill to your accounting system.

7. **Payment Processing**

- Once the bill is confirmed, proceed to *Accounting > Vendor Payments* or use the *Register Payment* option on the bill.
- Specify the payment method, amount, and date, then validate the payment.
- Odoo reconciles the payment with the bill, marking it as "Paid."

8. **Tracking and Reporting**

- Throughout the process, you can track the status of RFQs, POs, receipts, and bills via dashboards or smart buttons on the PO (e.g., *Delivery*, *Invoices*).
- Use *Reporting > Purchase Analysis* to generate insights on spending, vendor performance, or delays.

Key Features of Purchase

- **Convert RFQ to Order:** Easily convert quotations into purchase orders with a single click.
- **Email Order:** Enables users to send orders directly to vendors via email.

- **Product Pricing and Tax :** The purchase module facilitates the addition of multiple products to purchase orders, allowing users to specify prices and apply relevant taxes.
- **Variants Grid Entry:** Add product variants to your purchase orders with a grid, or matrix, displaying all the possible combinations of a product's attributes (e.g. sizes, colors).
- **Manage incoming products:** Keep track of your stock and determine quantity and locations for each batch of items you receive.
- **Purchase Dashboard:** The Purchase Dashboard in Odoo offers a centralized hub for monitoring procurement activities, providing real-time insights into purchasing performance and vendor relationships.
- **Vendors:** Odoo vendor management system provides a centralized hub for storing and managing vendor information, facilitating personalized.
- **Inventory forecasts:** Get forecasts of product availabilities based on confirmed sales orders, purchase orders or manufacturing orders as well as internal moves.

▼ Inventory

Notes:-

1. What happens if you try to validate a receipt without entering lot numbers?

   Odoo shows an error message

2. What is a traceability report used for ?

   Viewing the movement of lots and serials

3. What product type must an item be to enable lot or serial tracking ?

   Goods or storable

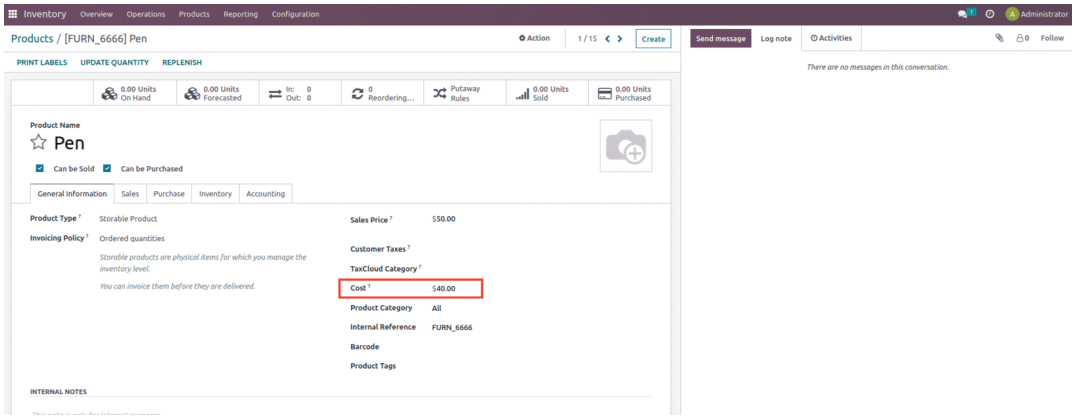
▼ Product Availability in Picking & it's values

1. Available: Products are reserved.
2. Available: Quantities available but not reserved.
3. Not Available: Insufficient quantities to fulfill order.
4. Exp [arrival date]: Expected to arrive today or earlier to delivery due date.
5. Exp [arrival date]: Expected to arrive after the delivery due date.

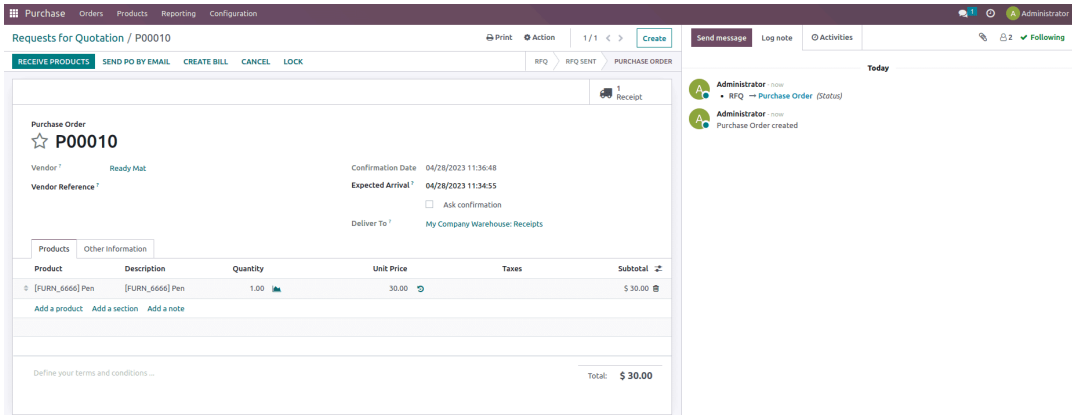
▼ Costing Method in Odoo

- ***Standard Price***: This method is very basic and easy. What we have to do is, just put the cost price on product master. The inventory valuation will not consider the amount for which you bought this item. It directly takes the cost price you provided on product master.

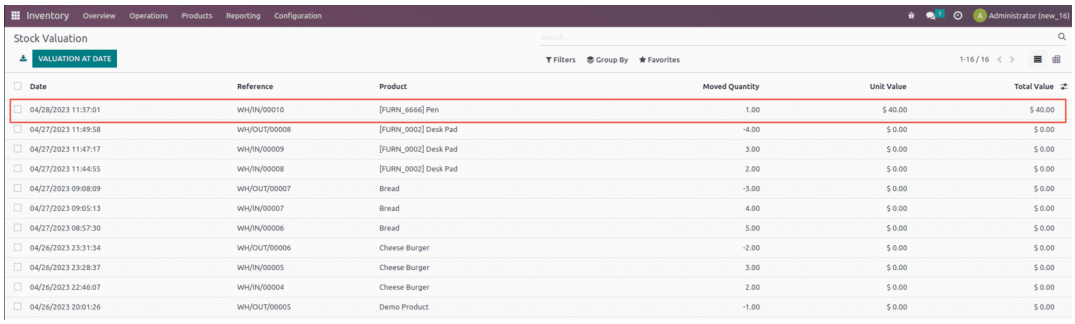
  Example: let's say for product pen we have set cost price as 40



Let's save it and create a purchase order for the product & validate the receipt.

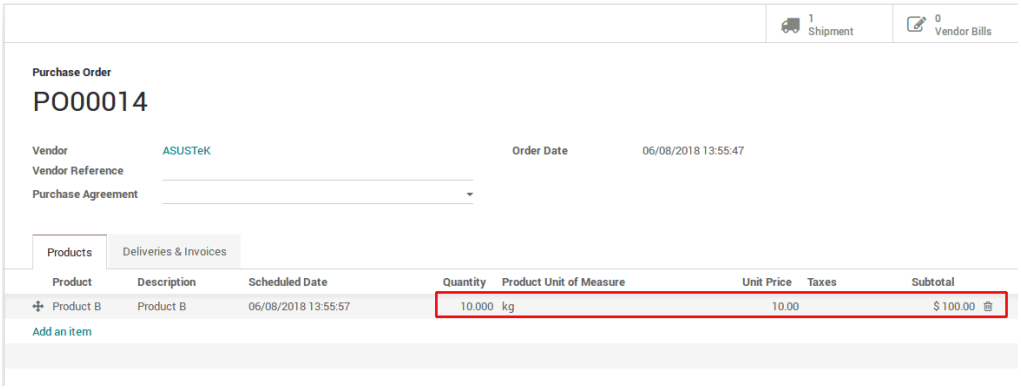


Once the purchase order is confirmed and validated, let's check the inventory valuation generated for the product Pen.

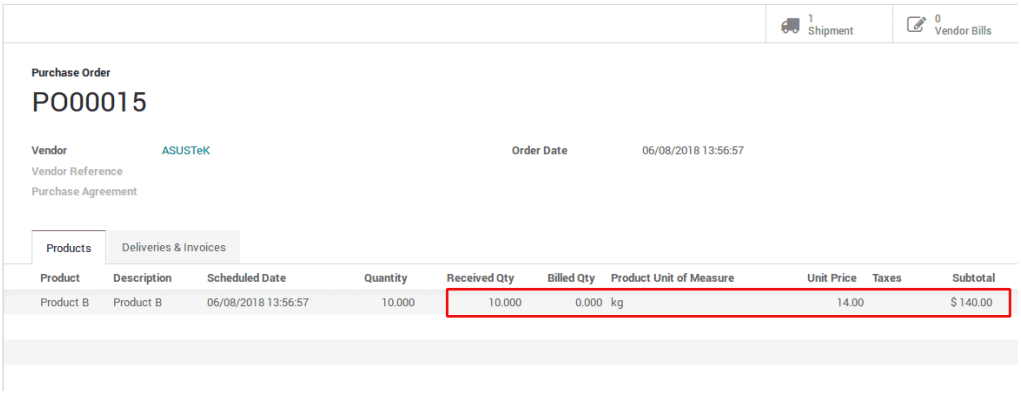


In a theoretical sense, we paid $30 for the product Pen, and when we check the Inventory Valuation report, we can see that for one quantity, the value added was $40 rather than the $30.

- ***Average Costing Method:*** The average costing approach calculates the cost of inventory items based on the average cost of all available similar goods in inventory. This is the cost of an item in inventory divided by the number of items in stock.
- Example: We bought 10 Kg of Product B for 100$. So, 10$/Kg



Again we bought Product B 10 kg for 140$. So 14$/Kg



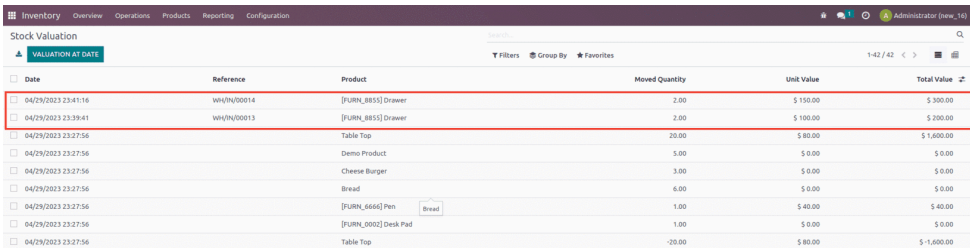
Now we have 20Kg of Product B in our inventory. Let's check the inventory value

You can see Valuation says the Inventory value of Product B is 240$ i.e, 12$/Kg (240$/20Kg)

Therefore the cost price of Product B is now $12

- ***First In First Out (FIFO):*** This technique adjusts a product's cost price based on the last existing product in the inventory. As a result, removal tactics such as FIFO have an impact on price.

  here the last existing product cost price Drawer is 150.



then the product cost will be updated to $150

▼ What is landed cost ? What are the different split methods available in Odoo?

Landed cost refers to the total cost incurred to get a product to its final destination. This includes not only the purchase price of the product but also additional costs such as shipping, handling, customs duties, insurance, and other associated fees.

Odoo provides the following split methods to distribute landed costs:

- Equal (Manual Distribution)
- By Quantity
- By Current Cost (Value-Based)
- By Weight
- By Volume

▼ Stock Removal Strategies

1. First In First Out (FIFO) :- Products/Lots that were stocked first will be moved out first.

2. Last In First Out (LIFO) :- Products/Lots that were stocked last will be moved out first.
3. Closest Location :- Products/Lots with the closest to the target location will be moved out first.
4. First Expiry First Out (FEFO) :- Products/Lots with the closest removal date will be moved out first(The availability of this method depends on the "Expiration Dates" Settings.)
5. Least Packages :- FIFO but with the least number of packages possible when there are several packages containing the same product.

▼ Location Types

In an Inventory, we usually manage three types of locations. They are Physical locations, partner locations, and virtual locations.

Physical Location

Physical location is part of the internal location. The internal locations will be located in the warehouse itself and that is why we have given a short name for an internal location as WH representing Warehouse.

Internal locations are the areas where we store our products within our odoo inventory. The loading of the products from the warehouse and unloading of materials to the warehouse also take place in this area.

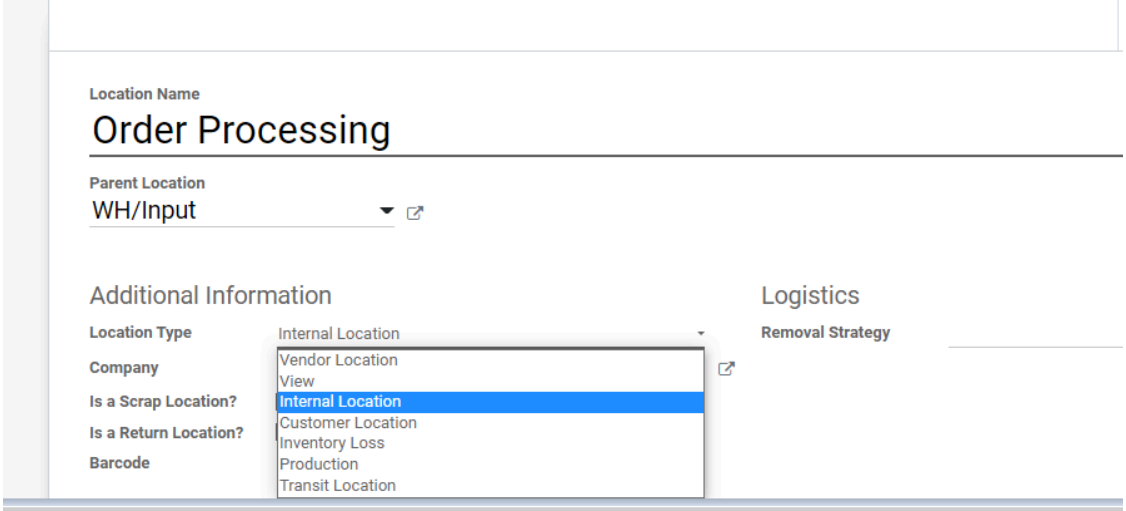
Partner location

Partner location is the location of the partners. Partner could be a customer or vendor. These locations would be outside your warehouse and the customer or vendor would be the owner of these locations. However, these locations operate the same way our physical locations work.

Virtual Locations

Apart from a physical location, companies manage virtual locations to represent a warehouse that really doesn't exist. Virtual location, as the name indicates, is a location that exists only virtually. The products which are not physically in the inventory can be placed in this location.

This kind of warehouse is used to show the movement of the product or to trace the history of the product. This is why we use virtual locations to place lost products and mark inventory loss.



Customer location: Customer location can be termed as the virtual location that will represent the location of the customer or the destination point of the product.

Inventory Loss: We use virtual locations to represent inventory loss. Manage correct stock levels with the support of this.

View: Virtual locations are used to manage the hierarchical structure of the locations. It cannot be used to store products directly.

Production: This is a virtual location or virtual counterpart location that will help us to manage production-related operations. It is at this location that the finished products are produced using materials.

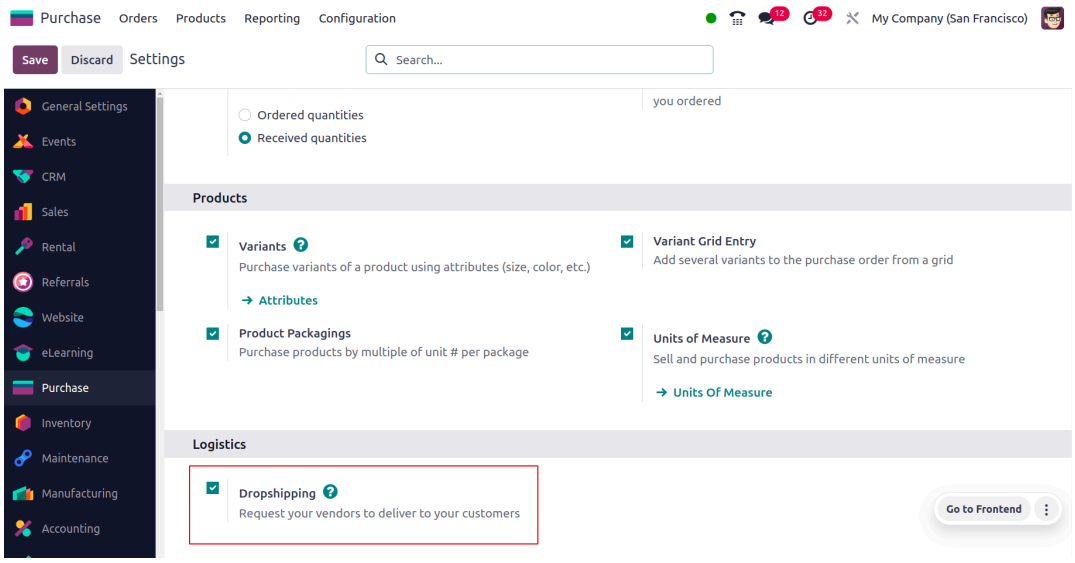
Transit Location: These are counterpart locations that have to be used to manage inter-warehouse operations and inter-company operations smoothly.

Company: It is also required to give the details of the company.

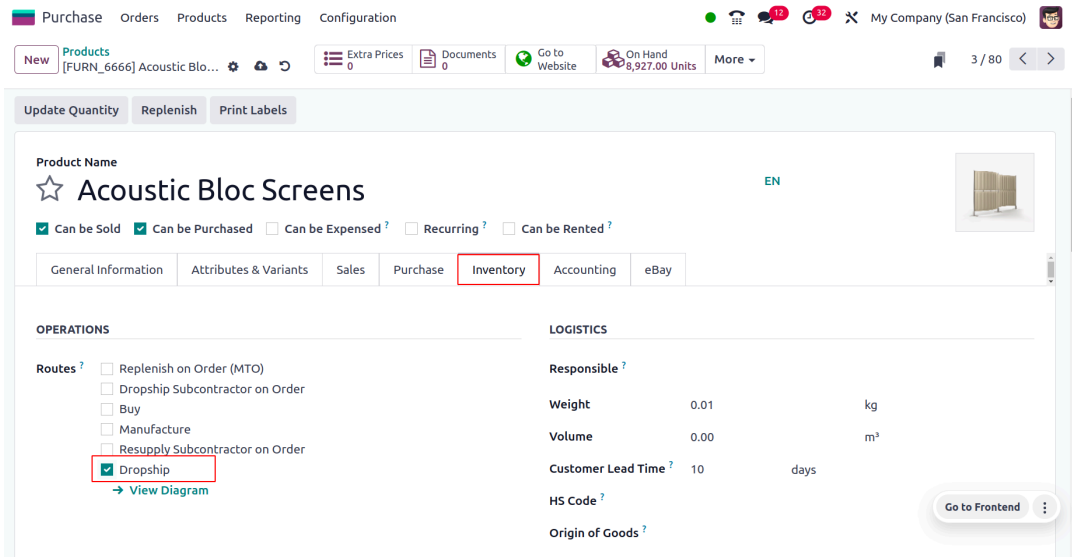
▼ Dropshipping in odoo

Dropshipping is one of the effective shipping methods that allow shipping of products directly from suppliers to customers. There will be no intermediary storage of products in the case of drop shipping. If you are running a retailing business, you don’t need to store the ordered product in your warehouse by selecting the drop shipping method.

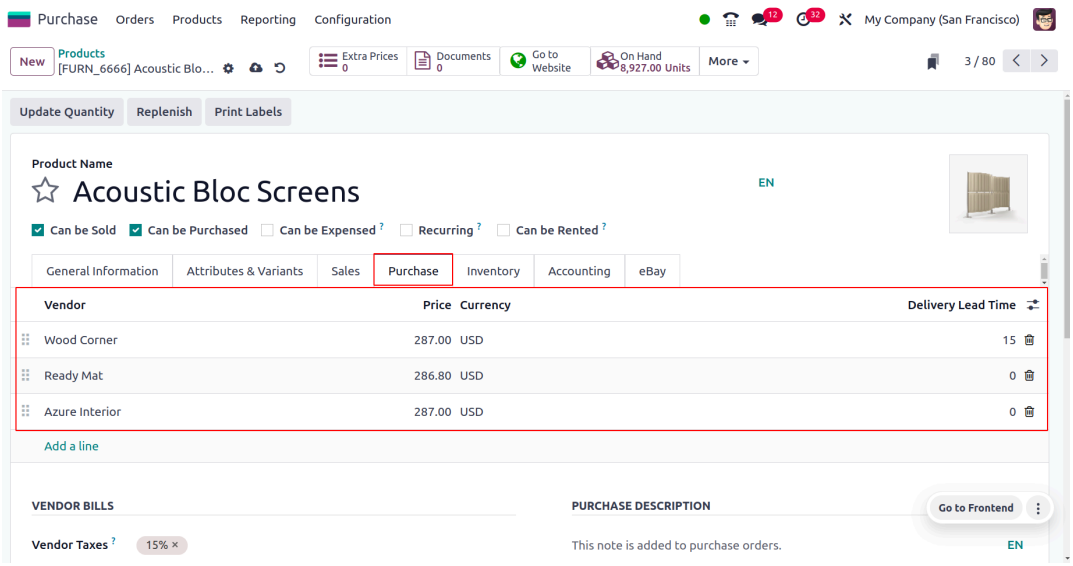
1. Goto Settings → Purchase → Activate the Drop Shipping feature under the Logistics tab.



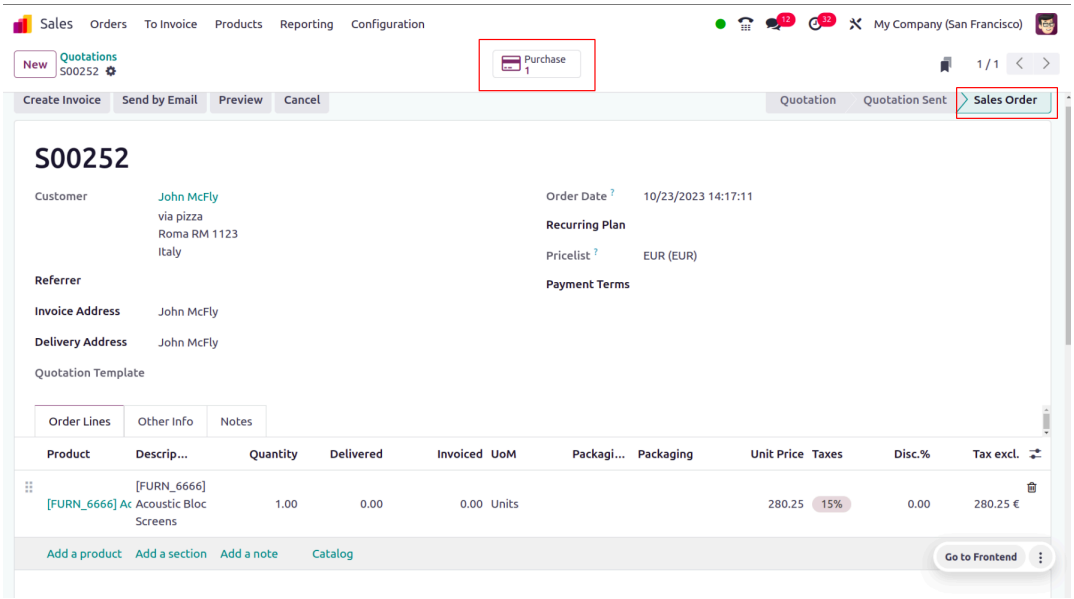
2. The product configuration provides the tab for an inventory of the ability to specify the operational routes. The routes of a product can be specified using this parameter.



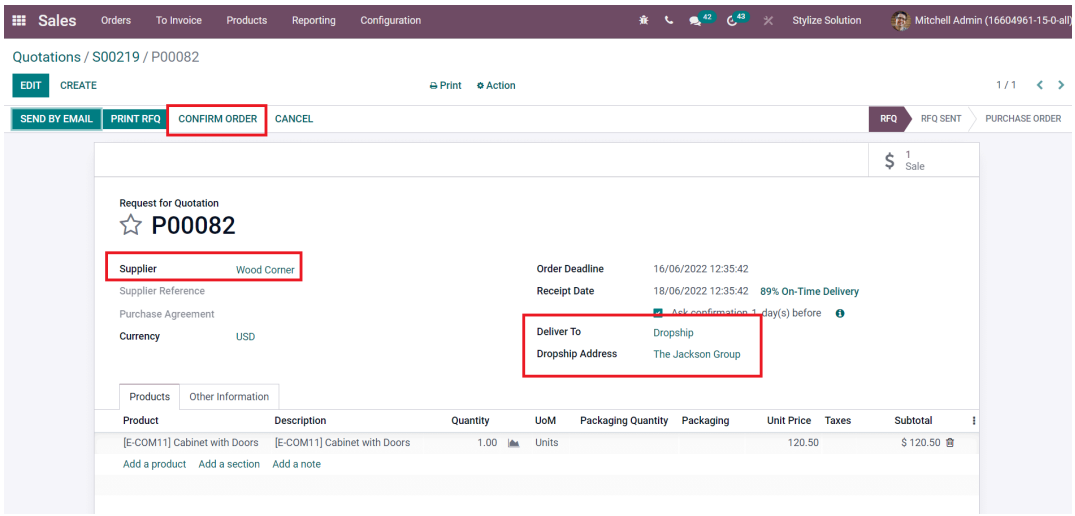
3. Additionally, don't forget to add a vendor from the product form's purchasing tab. This vendor will be the product's seller when a purchase order is created.



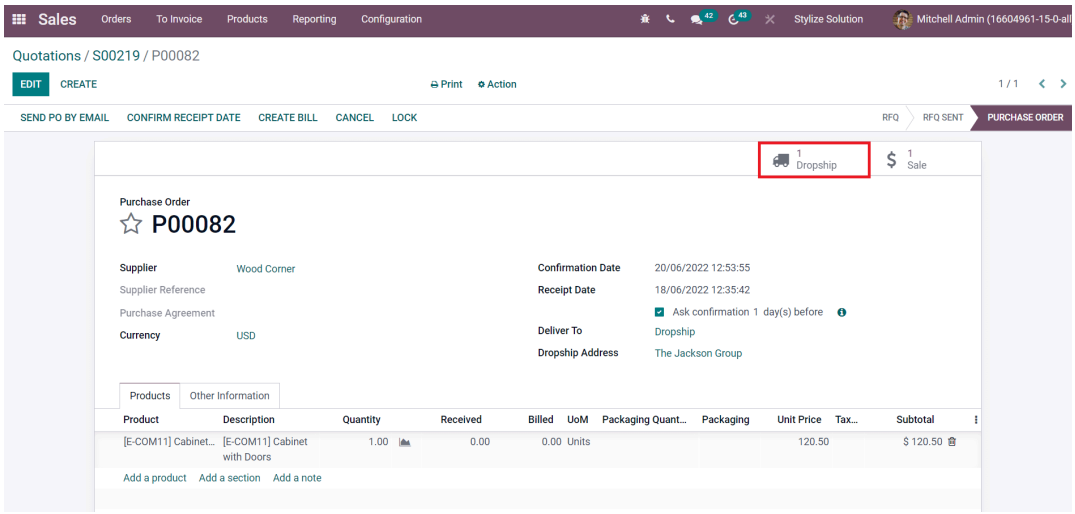
4. As you can see in the image , a newly generated Purchase Order can be seen on a smart button.



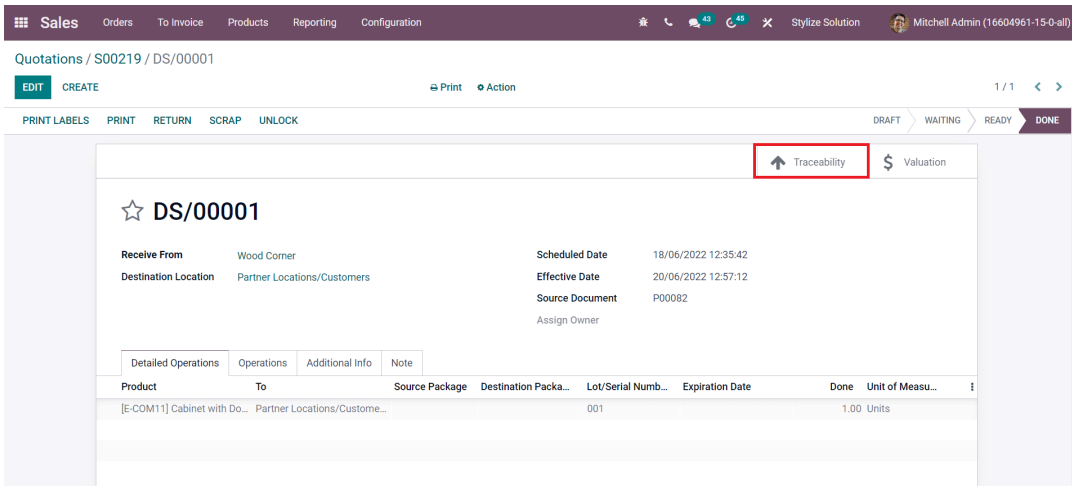
5. The image shows the Purchase details of the product from the respective vendor. The name of the Supplier can be seen here. The operation type of the incoming shipment and the Dropship Address will be shown in the corresponding fields. Once you confirm the Purchase Order, the product will be directly delivered to the customer (mentioned in the Sales Order) from the supplier (mentioned in the Purchase order).



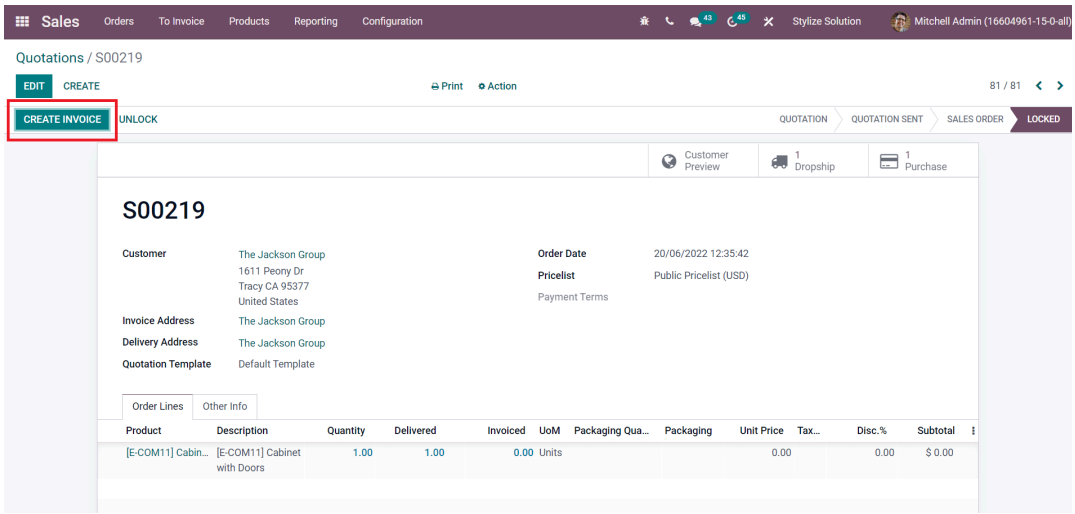
6. The smart button highlighted in the screenshot will show the Dropship details to Validate.



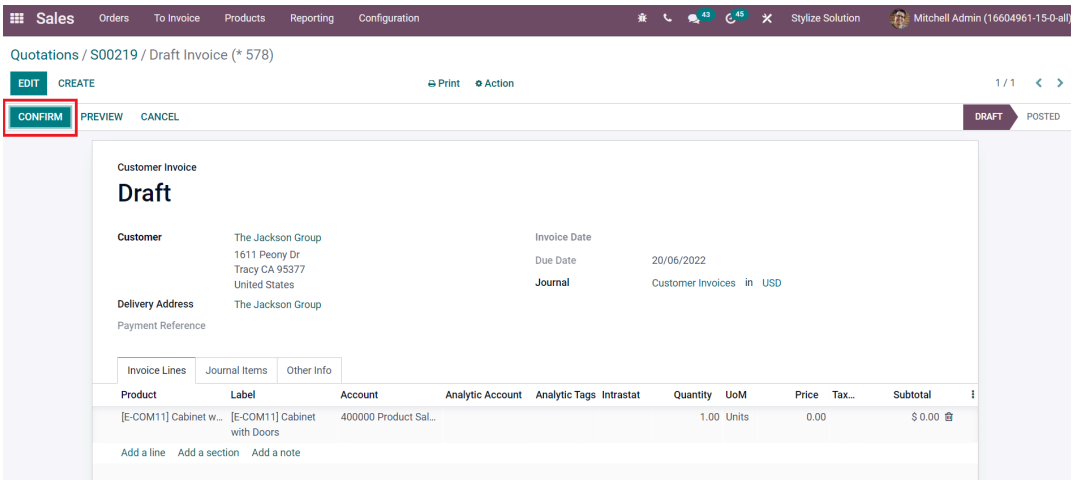
7. Validate the Dropship using the Validate button.



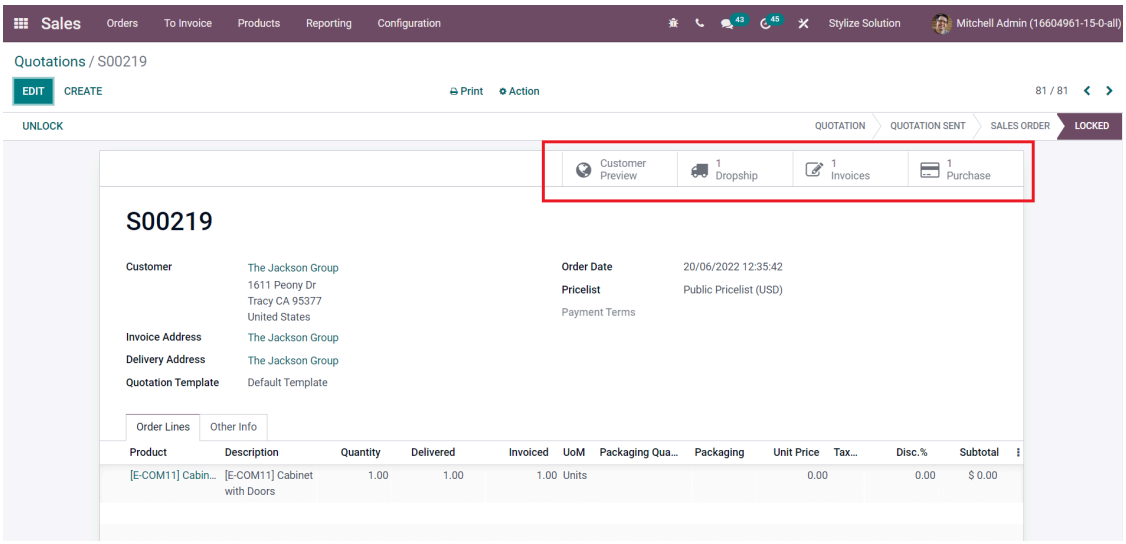
8. After validating the dropship, you are allowed to Create Bill for the purchase order and register payment. Coming back to the Sales Order, you will get the Create Invoice button in the window in order to generate an invoice for the product ‘Cabinet with Doors’ to the respective customer.



9. Clicking on this button will lead you to a new window where you will be able to see a draft form of the customer invoice which can be confirmed using the Confirm button.



10. The smart button in the Sales Order will show the records of Customer Preview, Dropshipping, Invoices, and Purchase Orders.



▼ reordering rule vs replenishment

Reordering Rules are used to automate the reordering of stock when it falls below a minimum level, ensuring that your stock is replenished based on predefined thresholds. It actually define when and how much of a product should be replenished.

On the other hand, Replenishment is the process of automatically or manually refilling the stocks when the quantity of product falls below a certain level.

Characteristics	Replenishment	Reordering Rules
Action	Immediate	Scheduled
Trigger	User-defined	Stock level
Flexibility	One-click	Automated
Use case	Emergency fill	Planned control

There are two methods for restocking inventory in Odoo: Reordering Rules and the Make to Order (MTO) route.

Reordering Rule

Automatic Reordering Rule

→ Since the **Trigger** is set to **Auto**, Odoo automatically generates a **Purchase Order** when the stock falls below the **Minimum Quantity**.

The scheduler is set to run once a day, by default.

To manually trigger a reordering rule before the scheduler runs, ensure developer mode is enabled, and select **Inventory app ▸ Operations ▸ Run Scheduler**.

Manual Reordering Rule

→ We need to manually click on **Order Once.**

Before adding a new reordering rule, ensure that the product has a vendor or a bill of materials specified on the product form. Additionally, confirm that the product form's Product Type is set to **Storable Product**. A consumable product cannot be included in the stock valuation because, by definition, its inventory levels are not recorded.

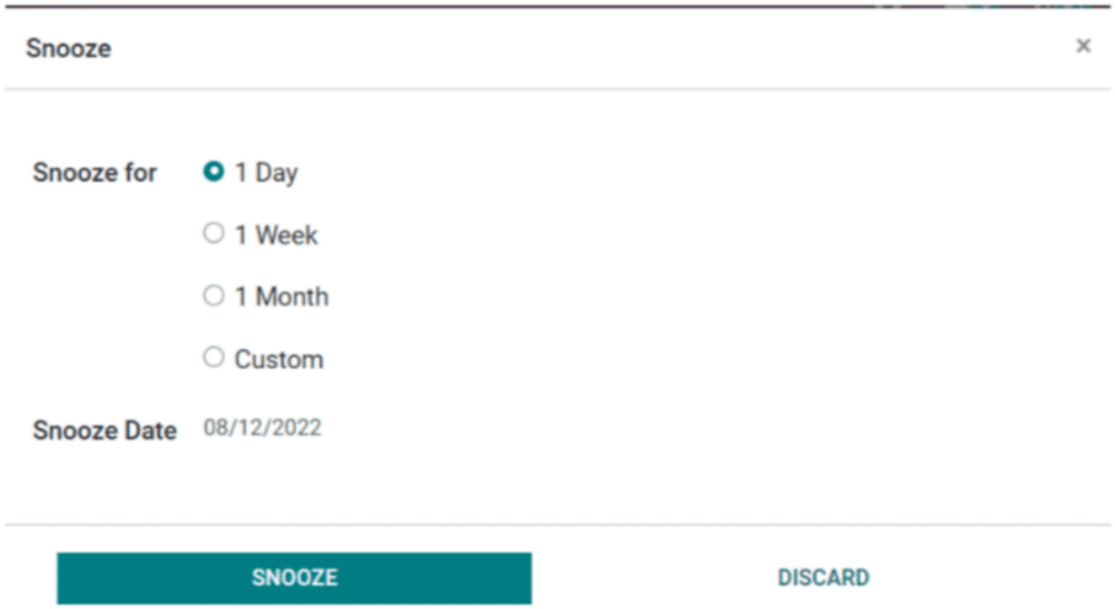
Go to **Inventory > Operations > Replenishment**, click **Create**, and set the **Product** to create a new reordering rule from the replenishment report. Set a **minimum** and **maximum** quantity, if desired. Lastly, select **Save**.

To create a new reordering rule from the product form, navigate to **Inventory > Products > Products**, select a product to open its product form, click the **Reordering Rules** smart button, and then click **Create**. After completing the fields, save the new reordering rule.

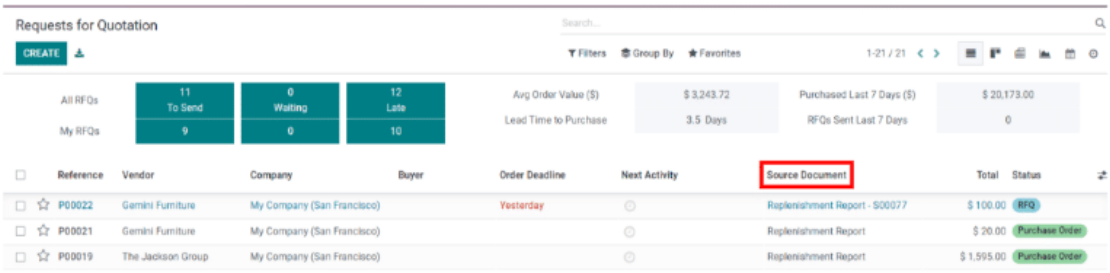
The quantity entered in the **To Order** field by default is the amount needed to reach the **Max Quantity** that has been set. However, by selecting the field, altering the value, and then selecting **Save**, the **To Order** quantity can be changed. Click **Order Once** to manually replenish a product.

Click **Automate Orders** to start automating replenishment. Every time the anticipated stock level drops below the specified **Min Quantity** of the reordering rule, Odoo will automatically generate a draft PO/MO when this button is pressed.

By using the **Snooze** button, a reordering rule can be momentarily turned off for a specific amount of time.



The Replenishment Report will be the source document for any PO or MO produced by a manual replenishment. The sales order(s) reference number(s) that triggered the rule will be listed as the source document on any PO or MO created by an automatic reordering rule.



Make to Order

Make to Order (MTO) is a purchasing strategy that, regardless of the quantity of stock on hand, generates a preliminary purchase order or manufacture order each time a sales order is approved.

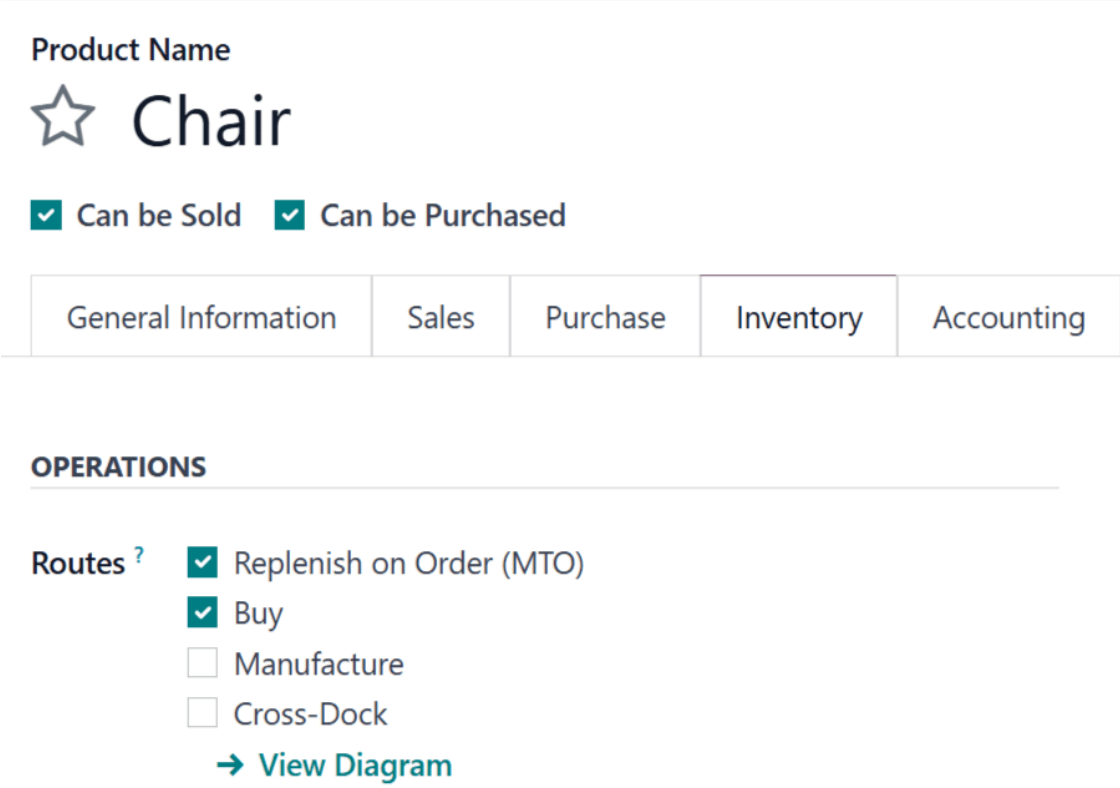
Odoo automatically connects the sales order to the purchase order (PO) or manufacturing order (MO) produced by the MTO route, unlike products supplied using reordering rules. Another distinction between MTO and reordering rules is that with MTO, Odoo generates a draft PO or MO right away following the confirmation of the sales order. When the product's expected stock falls below the predetermined minimum quantity, reordering rules in Odoo generate a draft PO or MO.

To activate the Make to Order (MTO) route in Odoo:

1. To activate the **Make to Order (MTO)** route in Odoo:
2. Go to **Inventory ▸ Configuration ▸ Settings ▸ Warehouse**.
3. Activate **Multi-Step Routes** setting and click **Save**.
4. Go to **Inventory ▸ Configuration ▸ Routes**.
5. Click on **Filters ▸ Archived** to show archived routes.
6. Select the checkbox next to **Replenish on Order (MTO)**, and click on **Action ▸ Unarchive**.

Go to **Inventory > Products > Products**, click on a product to view the product form, then click **Edit** to set a product's procurement route to MTO. Next, select Replenish on Order under the **Routes** choices on the **Inventory** tab (MTO). Make

sure the **Buy** route is chosen in addition to the MTO route for products that are purchased directly from a vendor, and that a vendor is set up in the **Purchase** tab. Make sure a bill of materials is configured for internal products and that the **Manufacture** route is chosen in addition to the MTO route. Lastly, select **Save**.



Replenish using MTO

After configuring a product to use the MTO route, a replenishment order is created for it every time an SO or MO including the product is confirmed. The type of order created depends on the second route selected in addition to MTO.

For example, if *Buy* was the second route selected, then a PO is created upon confirmation of an SO.

While the MTO route can be used in union with the *Buy* or *Manufacture* routes, the *Buy* route is used as the example for this workflow. Begin by navigating to the **Sales** app, then click **New**, which opens a blank quotation form.

On the blank quotation form, add a **Customer**. Then, click **Add a product** under the **Order Lines** tab, and enter a product configured to use

the *MTO* and *Buy* routes. Click **Confirm**, and the quotation is turned into an SO.

A **Purchase** smart button now appears at the top of the page. Clicking it opens the RFQ associated with the SO.

Click **Confirm Order** to confirm the RFQ, and turn it into a PO. A purple **Receive Products** button now appears above the PO. Once the products are received, click **Receive Products** to open the receipt order, and click **Validate** to enter the products into inventory.

Return to the SO by clicking the **SO** breadcrumb, or by navigating to **Sales app** ▸ **Orders** ▸ **Orders**, and selecting the SO.

Finally, click the **Delivery** smart button at the top of the order to open the delivery order. Once the products have been shipped to the customer, click **Validate** to confirm the delivery.

▼ Lot vs serial number

Lots correspond to a certain number of products you received and store altogether in one single pack.

Serial numbers are identification numbers given to one product in particular, to allow to track the history of the item from reception to delivery and after-sales.

When to use

Lots are interesting for products you receive in great quantity and for which a lot number can help in reportings, quality controls, or any other info. Lots will help identify a number of pieces having for instance a production fault. It can be useful for a batch production of clothes or food.

Serial numbers are interesting for items that could require after-sales service, such as smartphones, laptops, fridges, and any electronic devices. You could use the manufacturer's serial number or your own, depending on the way you manage these products

▼ Putaway Rules

Putaway is the process of ***routing products to appropriate storage locations upon shipment arrival***. It means where the products should be stored within the warehouse or inventory once they are received based on predefined criteria such as product category, weight, volume, or specific storage requirements.

How can I set Putaway Rules based on closest location or last location used ?

How can I set Putaway Rules based on closest location or last location used | Odoo 17.4

I would like to receive products automatically in either the last location that was used to store that product or in the closes location. Is that possible?

https://www.odoo.com/fr_FR/forum/aide-1/how-can-i-set-putaway-rules-based-on-closest-location-or-last-location-used-odoo-17-4-259844

▼ How to configure inventory settings in Odoo ?

1. **Define products:** Create product records, specifying their names, descriptions, units of measure, and other relevant attributes.
2. **Configure inventory valuation:** Choose an inventory valuation method, such as FIFO (First-In, First-Out) or LIFO (Last-In, First-Out), to determine the cost of goods sold.
3. **Set up warehouses:** Define warehouses and locations to represent your physical storage spaces. While choosing, we also need to choose a replenishment method. This method determines how your inventory will be replenished when it reaches a certain threshold. You can opt for manual replenishment, where you manually create purchase orders or manufacturing orders, or you can set up automatic replenishment based on predefined rules and forecasts.
4. **Putaway Rules:** Putaway is the process of taking products off the receiving shipments and directly putting them into the most appropriate location.

   For instance, if you run a retail store that stores whiteboard pens and customizable desks. While in the same warehouse, you would want to ensure pens are stored in the pen location and desks in the desk location. Putaway Rules is what you use to arrange products even before they get to the warehouse.
5. **Set up Operation Types**: Operation types are the types of operation that are used in terms of transferring goods from one location to another.

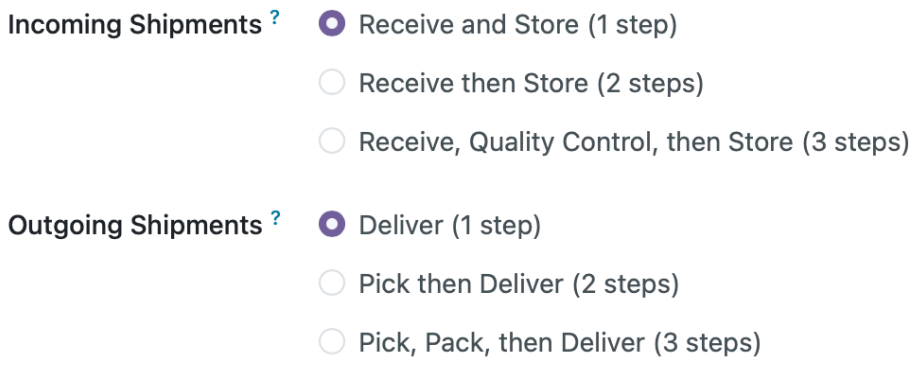
   There are mainly 3 types of operation: 1. Receipt, 2. Delivery, and 3. Internal Transfer.
6. **Define routes:** Create routes to define the paths that goods take through your warehouse and supply chain.

7. **Configure shipping methods:** Set up shipping methods to integrate with shipping carriers and calculate shipping costs.

Key Features of Odoo Inventory Management

1. Real-Time Inventory Tracking: Odoo provides real-time updates on stock availability, allowing businesses to make informed decisions and prevent stockouts or overstocking.
2. Warehouse Organization: With Odoo, you can define your warehouse layout, including racks, bins, and locations. This enables efficient storage management and streamlined order picking processes.
3. Barcode Scanning: By integrating barcode scanners with Odoo, businesses can automate and enhance their inventory processes, improving accuracy and efficiency.
4. Automated Reordering: Odoo Inventory Management offers automatic reordering capabilities, ensuring that you never run out of stock by generating purchase orders based on predefined stock levels.
5. Batch and Serial Number Tracking: Businesses dealing with serialized or batch-tracked products can easily track and manage these items using Odoo. This feature helps ensure product traceability and compliance.
6. Inventory Reporting and Analytics: Odoo provides comprehensive reporting and analytics capabilities, allowing businesses to gain insights into their inventory performance, identify trends, and make data-driven decisions.
7. Integration with Other Modules: Odoo Inventory Management seamlessly integrates with other modules within the Odoo ecosystem, such as sales, purchasing, and manufacturing, providing end-to-end visibility and control over your business processes.
8. Multi-Location Management: With Odoo, businesses can efficiently manage stock across multiple locations, whether it's different warehouses, stores, or even virtual locations. This ensures accurate inventory tracking and efficient stock transfers.
9. Quality Control: Odoo allows businesses to define quality control processes and perform quality checks on incoming and outgoing products. This helps maintain product integrity and customer satisfaction.

▼ Incoming & Outgoing Shipments Steps



One-step receipt and delivery

In Odoo *Inventory*, both incoming and outgoing shipments are configured to process in one step, by default. This means purchases will be received directly into stock, and deliveries will be moved directly from stock to customers.

Receive goods directly (1 step)

When products are received in one step, they will move from the vendor location to warehouse stock in the database immediately upon validation of a purchase order (PO).

Deliver goods directly (1 step)

When products are delivered in one step, they will move from warehouse stock to the customer location in the database immediately upon validation of a sales order (SO).

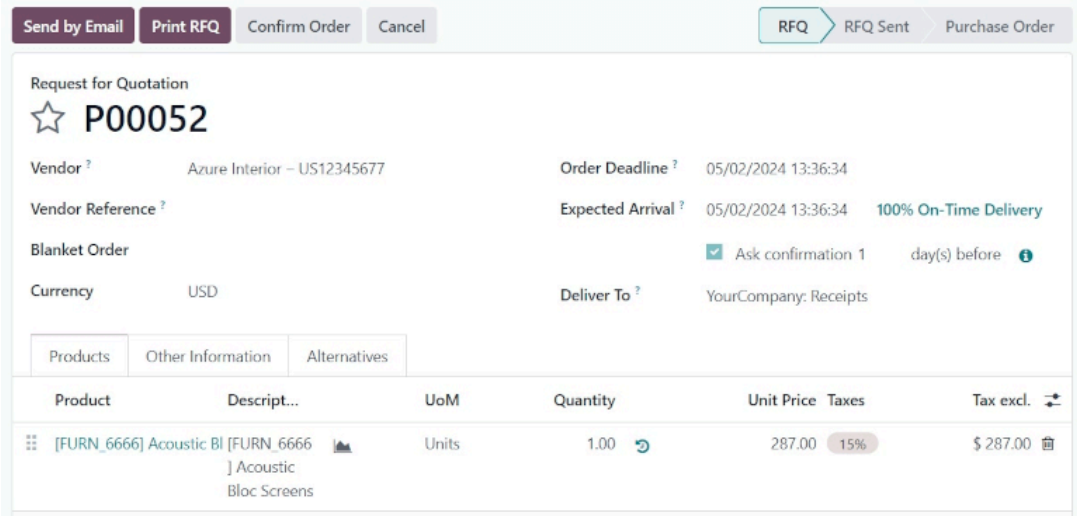
Two-step receipt and delivery

When products are received in two steps, they first move from the vendor location to an input location.Then, they move from the input location to warehouse stock in the database, upon validation of a purchase order (PO), and a subsequent internal transfer.

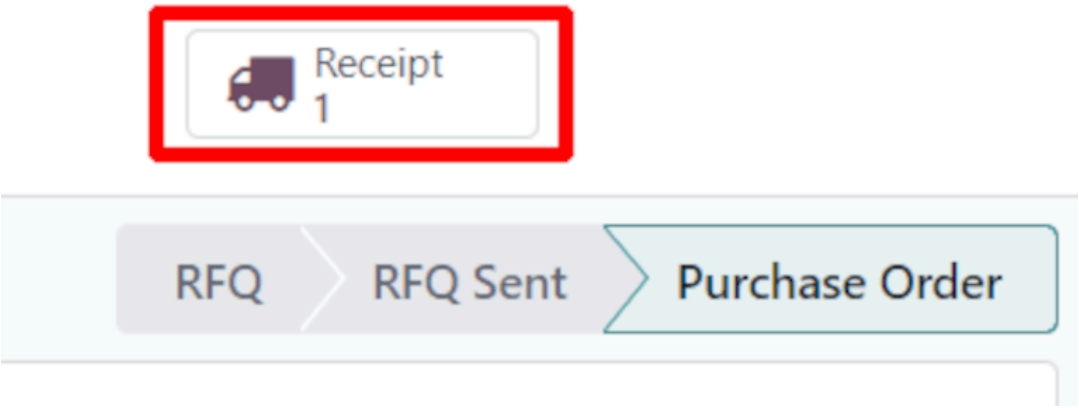
▼ Process receipt in two steps (input + stock)

When products are received in two steps, they first move from the vendor location to an input location. Then, they move from the input location to warehouse stock in the database, upon validation of a purchase order (PO), and a subsequent internal transfer.

1. Create PO(Purchase Order)

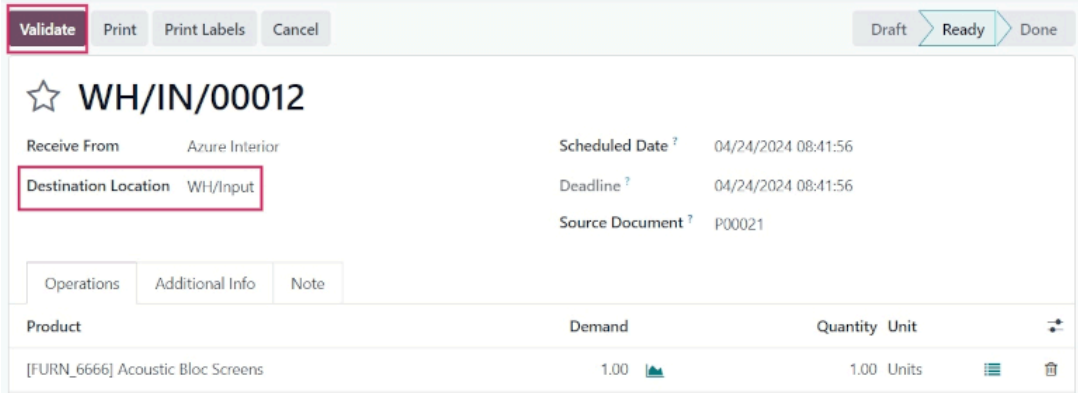


Once the PO is confirmed, a **Receipt** smart button appears at the top of the form. Clicking the smart button opens the warehouse receipt (WH/IN) form.



Process receipt

From the warehouse receipt form, the products ordered can be received into the warehouse. To receive the products, click **Validate**. Once validated, the receipt moves to the **Done** stage, and the products move to the **WH/Input** location.

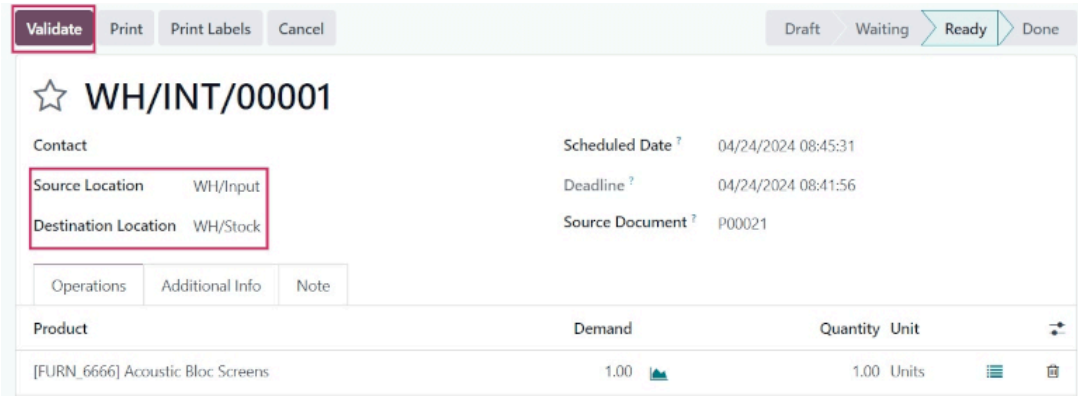


Process internal transfer

Once the receipt is validated, an internal transfer is created and ready to process.

Once ready, click **Validate** to complete the transfer, and move the product from **WH/Input** to **WH/Stock**.

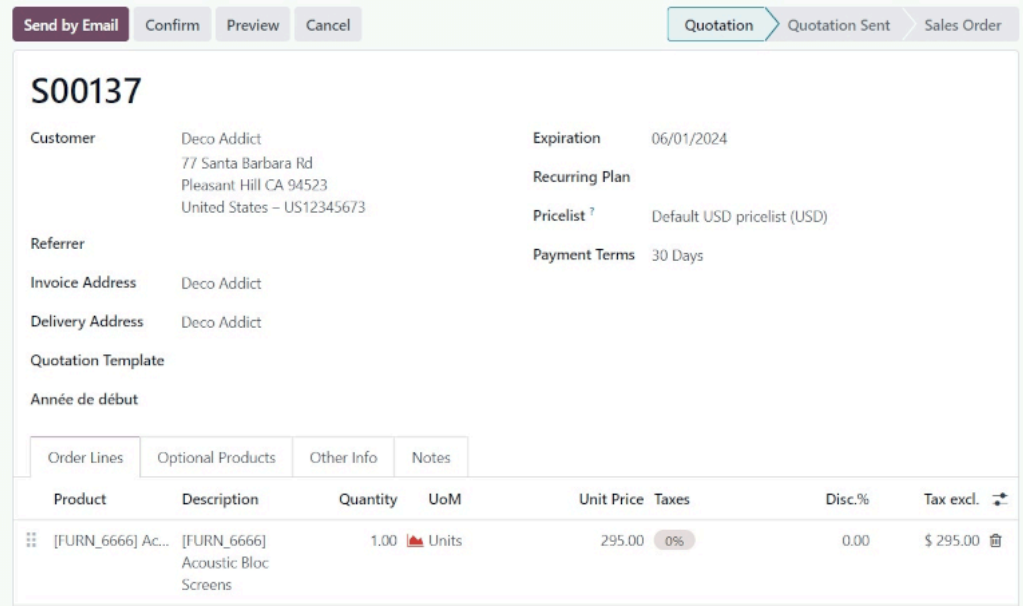
Once the transfer is validated, the products enter inventory, and are available for customer deliveries or manufacturing orders.



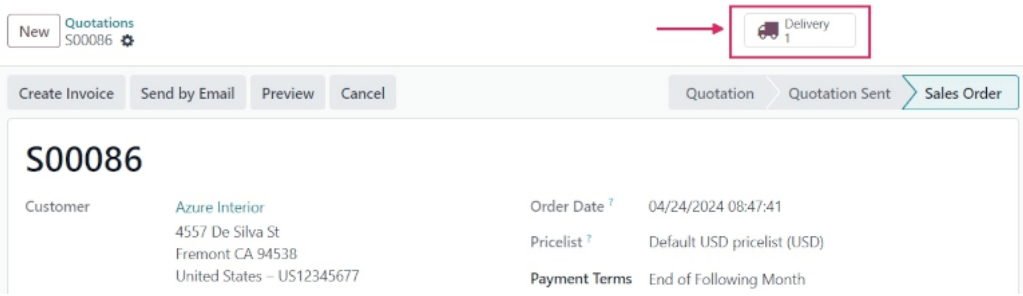
▼ Process delivery order in two steps (pick + ship)

When products are delivered in two steps, they move from warehouse stock to an output location. Then, they move from the output location to a customer location in the database, upon validation of a picking order, and a subsequent delivery order (DO).

1. Create Sale Order(SO)



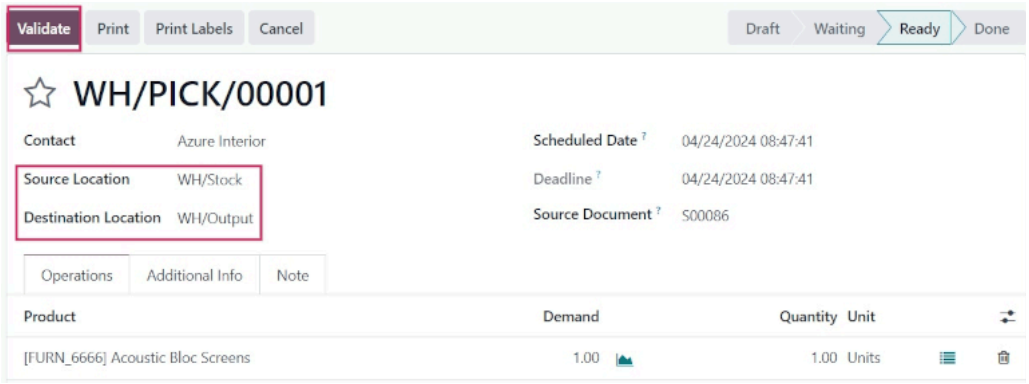
Once the SO is confirmed, a **Delivery** smart button appears at the top of the form. Clicking the smart button opens the warehouse delivery (WH/OUT) form.



Process picking

Once the sales order is confirmed, a picking order is generated and ready to process.

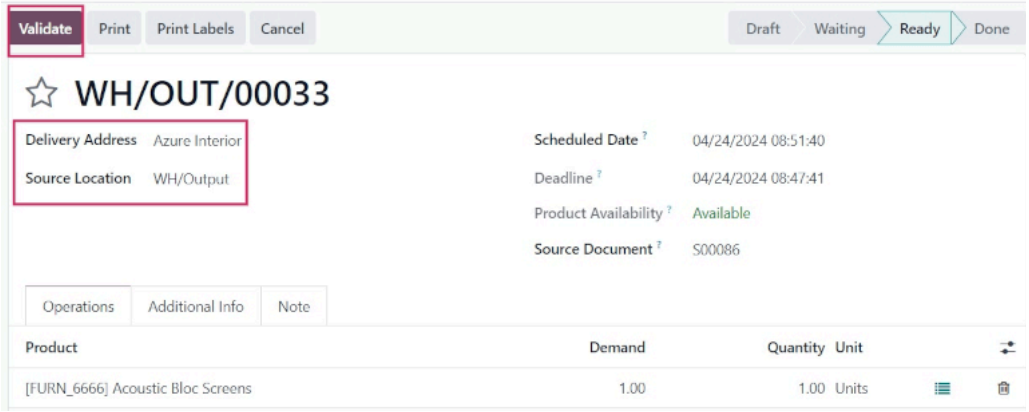
Click on the picking (WH/PICK) operation associated with the sales order to reveal the picking order.



Once ready, click **Validate** to complete the picking, and move the product from **WH/Stock** to **WH/Output.**

Process delivery

Once the picking is validated, a delivery order is created, and ready to process. Clicking the **Delivery** smart button on the sales order form reveals the newly created delivery order.



Three-step receipt

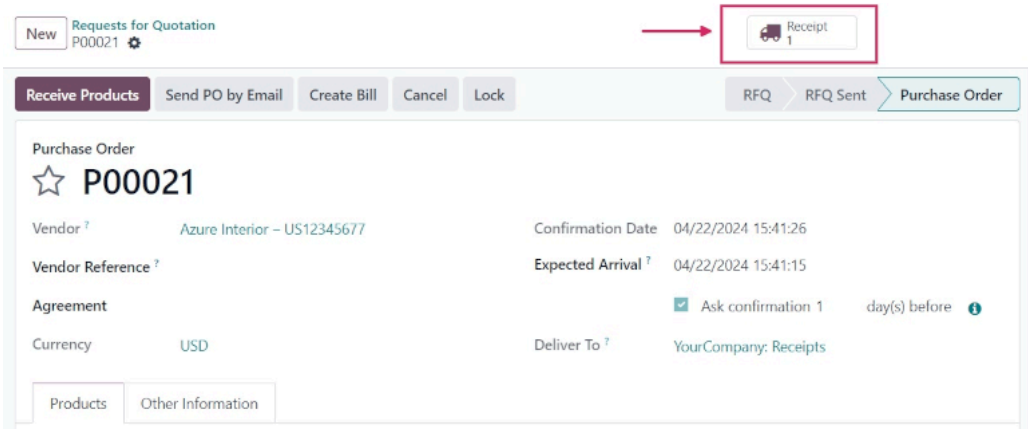
In the three-step receipt process, products are received in an input area, then transferred to a quality area for inspection. Products that pass the quality inspection are then transferred into stock.

▼ Receive in three steps (input + quality + stock)

1. Create PO(Purchase Order)

   A **Receipt** smart button will appear in the top right, and the receipt will be associated with the purchase order. Clicking on the **Receipt** smart button

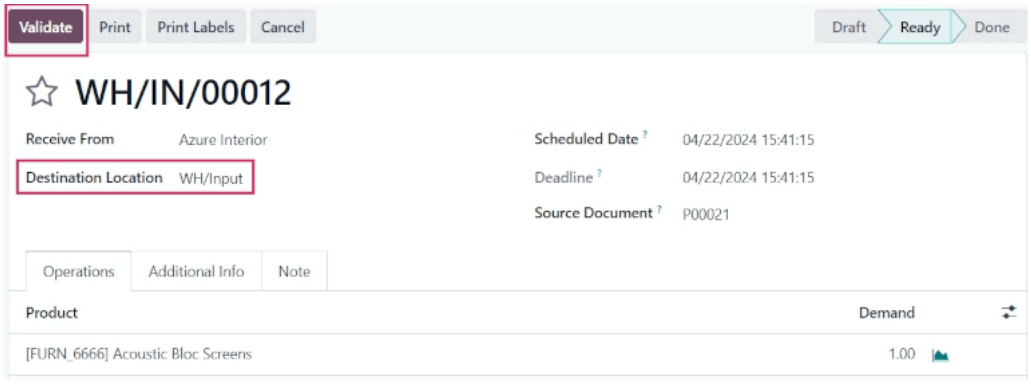
will show the receipt order.



Process a receipt

Once the purchase order (PO) is confirmed, a receipt (WH/IN) operation is generated and ready to process.

Click **Validate** to validate the receipt, and move the product to the destination location, **WH/Input**.

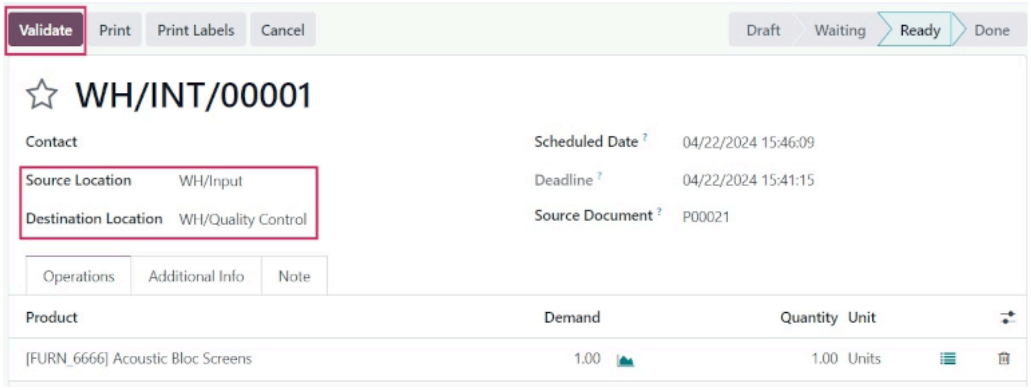


Process a transfer to Quality Control

Click **Inventory Overview** in the breadcrumbs to navigate back to the dashboard, and locate the **Internal Transfers** task card.

Select the **# To Process** button to reveal all internal transfers to process. Then, choose the internal transfer associated with the validated receipt.

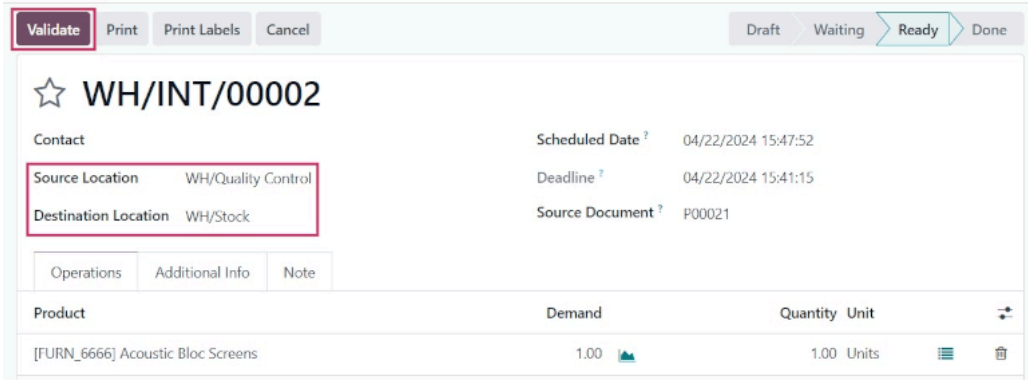
Once ready, click **Validate** to complete the transfer, and move the product from **WH/Input** to **WH/Quality Control**.



Process a transfer to stock

Once the internal transfer to move the product to quality control has been validated, another internal transfer operation to move the product into warehouse stock is ready to process.

Once ready, click **Validate** to complete the transfer, and move the product from **WH/Quality Control** to **WH/Stock.**



Three-step delivery

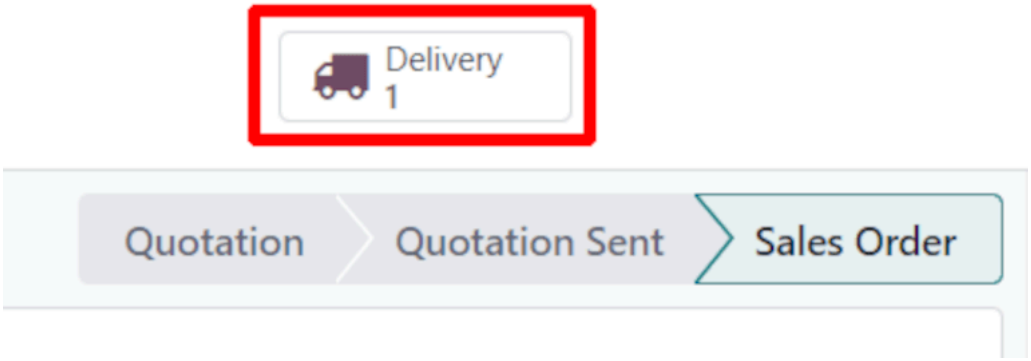
In the default three-step delivery process, products that are part of a delivery order are picked in the warehouse according to their removal strategy, and brought

to a packing zone. After the items have been packed into the different shipments in the packing zone, they are brought to an output location before being shipped.

▼ Deliver in three steps (pick + pack + ship)

1. Create a SO

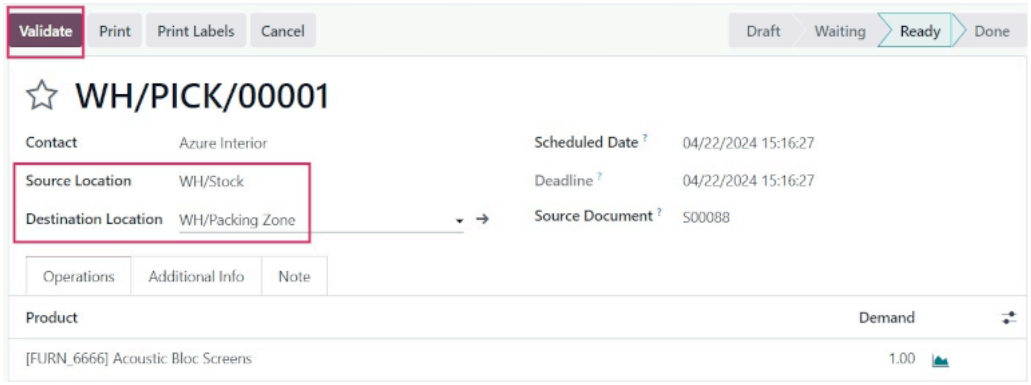
   A **Delivery** smart button appears in the top right of the quotation form. Clicking it opens the picking order to move the ordered product from `WH/Stock` to `WH/Packing Zone`.



Process a picking

The picking order will be created once the sales order is confirmed.

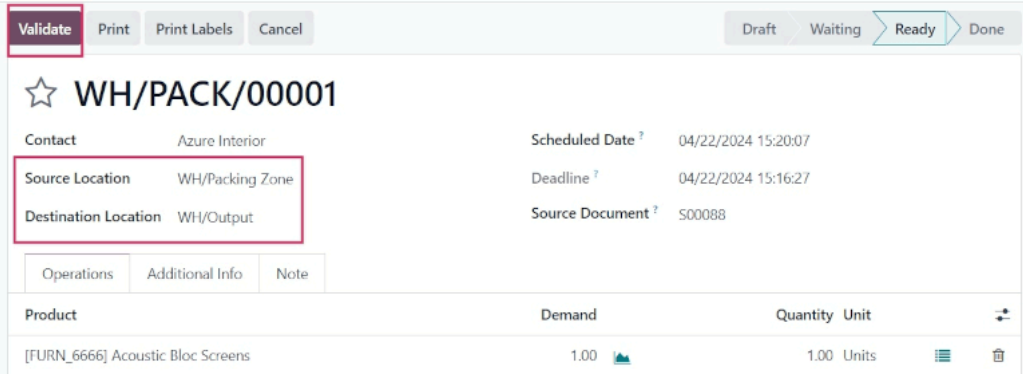
Click on the picking to process. If the product is in stock, Odoo will automatically reserve the product. Click **Validate** to mark the picking as done, and complete the transfer to the **Packing Zone**.



Process a packing

After validating the picking, the packing order is ready to process.

Click on the packing order associated with the sales order, then click on **Validate** to complete the packing.



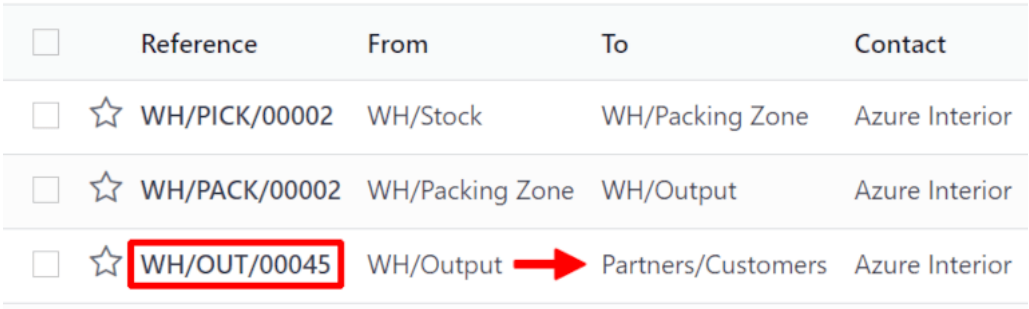
Process a delivery

Once the packing order has been validated, the delivery order is ready to process.

Navigate back to the original sales order to process the delivery by going to **Sales app**, and selecting the sales order created previously.

The **Delivery** smart button now indicates there are 3 transfers, instead of one. Clicking the **Delivery** smart button shows the three operations for this sales order: the picking, the packing, and the delivery.

Click the delivery (WH/OUT) transfer to open the delivery order. Then, click **Validate**.



Once the delivery order is validated, the product leaves the **WH/Output** location and moves to the **Partners/Customers** location.

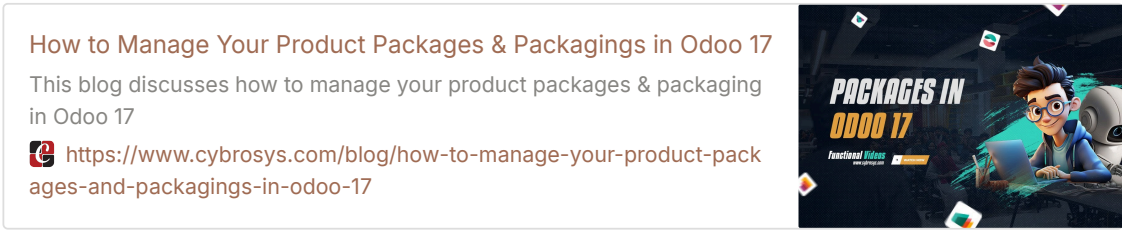
Then, the status of the document will change to **Done**.

▼ Packages & Packagings

In Odoo, there are two options for packages:

1. Packages, which are packs of different products that allow tracking and placement of different products

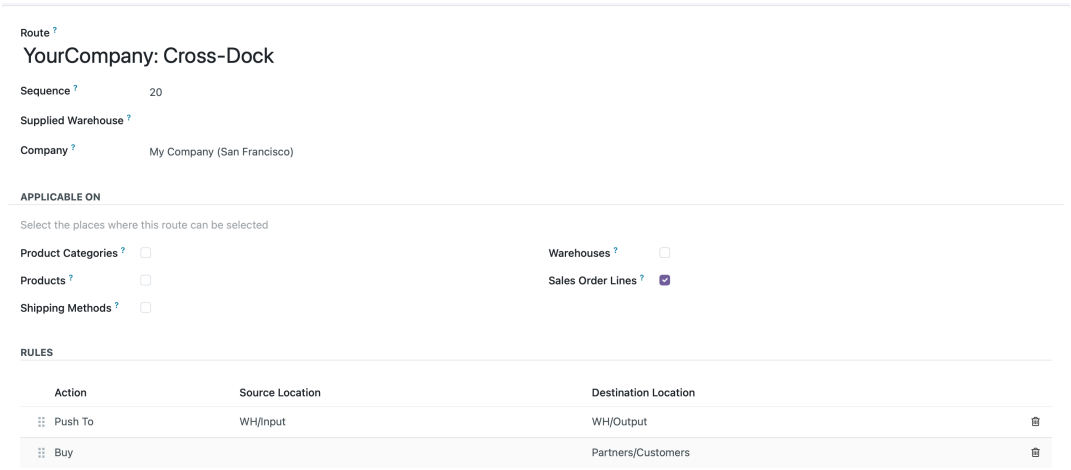
2. Product packaging is used to package varied amounts of the same product in a single pack that contains several units of the same product.



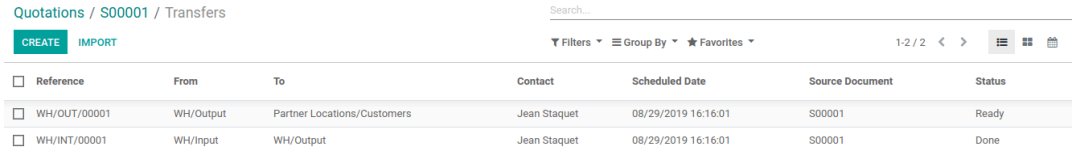
▼ Cross Docking

Cross-docking is the process of sending products that are received directly to the customers, without making them enter the stock. The trucks are simply unloaded in a *Cross-Dock* area in order to reorganize products and load another truck.

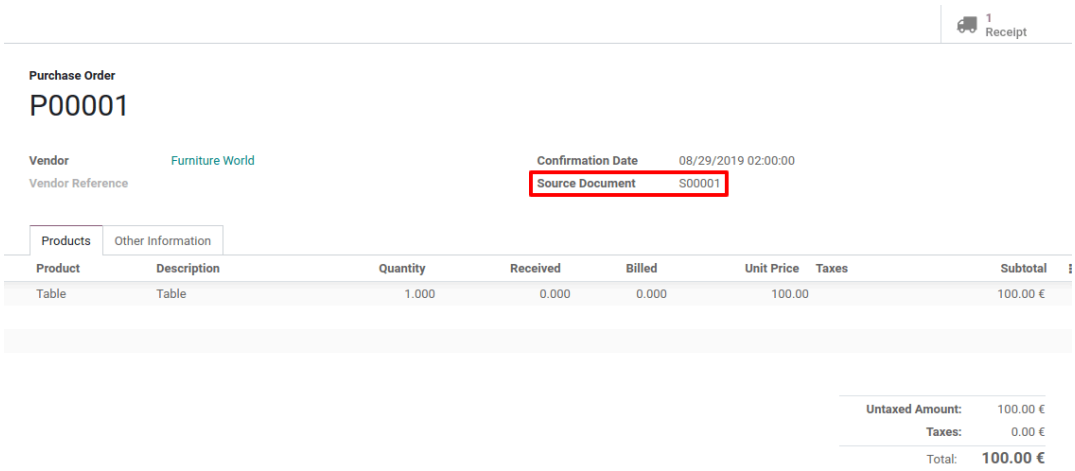
1. open **Configuration ▸ Settings** and activate the *Multi-Step Routes*.
2. both *Incoming* and *Outgoing* shipments should be configured to work with 2 steps. To adapt the configuration, go to **Inventory ▸ Configuration ▸ Warehouses** and edit your warehouse.
3. This modification will lead to the creation of a *Cross-Docking* route that can be found in **Inventory ▸ Configuration ▸ Routes**.



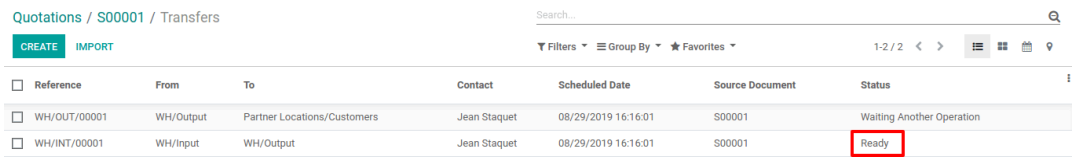
4. Create the product that uses the *Cross-Dock Route* and then, in the inventory tab, select the routes *Buy* and *Cross-Dock*. Now, in the purchase tab, specify the vendor to who you buy the product and set a price for it.
5. Create a sale order for the product and confirm it. Odoo will automatically create two transfers which will be linked to the sale order. The first one is the transfer from the *Input Location* to the *Output Location*, corresponding to the move of the product in the *Cross-Dock* area. The second one is the delivery order from the *Output Location* to your *Customer Location. Both are in state *Waiting Another Operation* because we still need to order the product to our supplier.



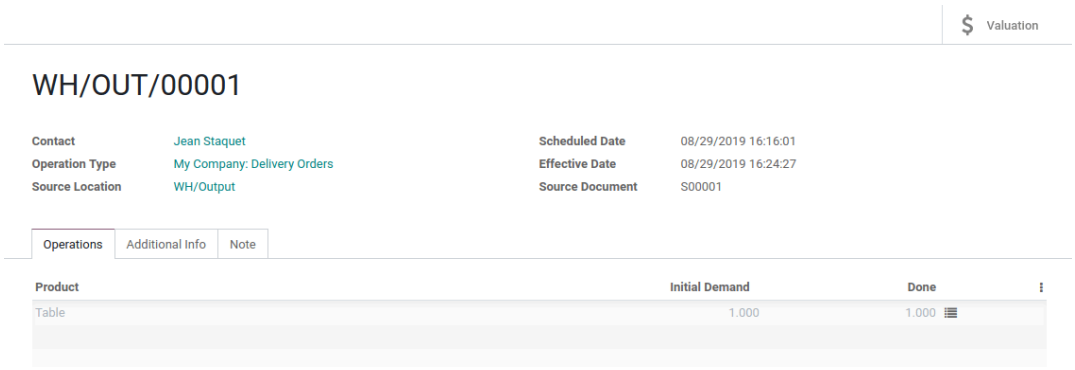
6. Now, go to the *Purchase* app. There, you will find the purchase order that has been automatically triggered by the system. Validate it and receive the products in the *Input Location*.



7. When the products have been received from the supplier, you can go back to your initial sale order and validate the internal transfer from *Input* to *Output*.



8. The delivery order is now ready to be processed and can be validated too.



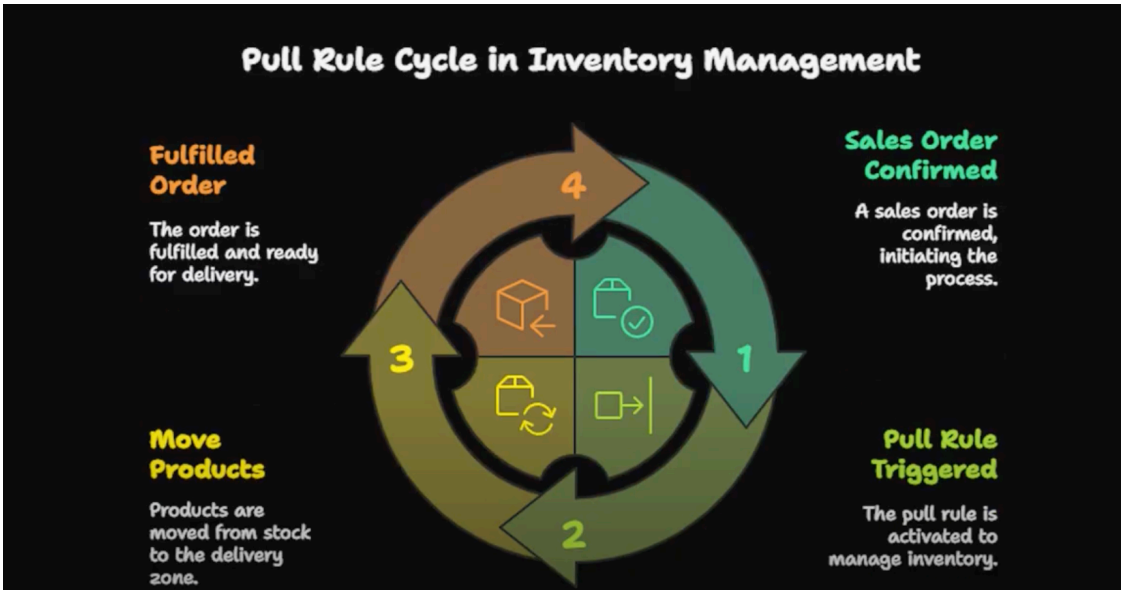
▼ Push & Pull rule

Routes are paths in which products can move.

Routes are defined by rules which guide products from one location to another.

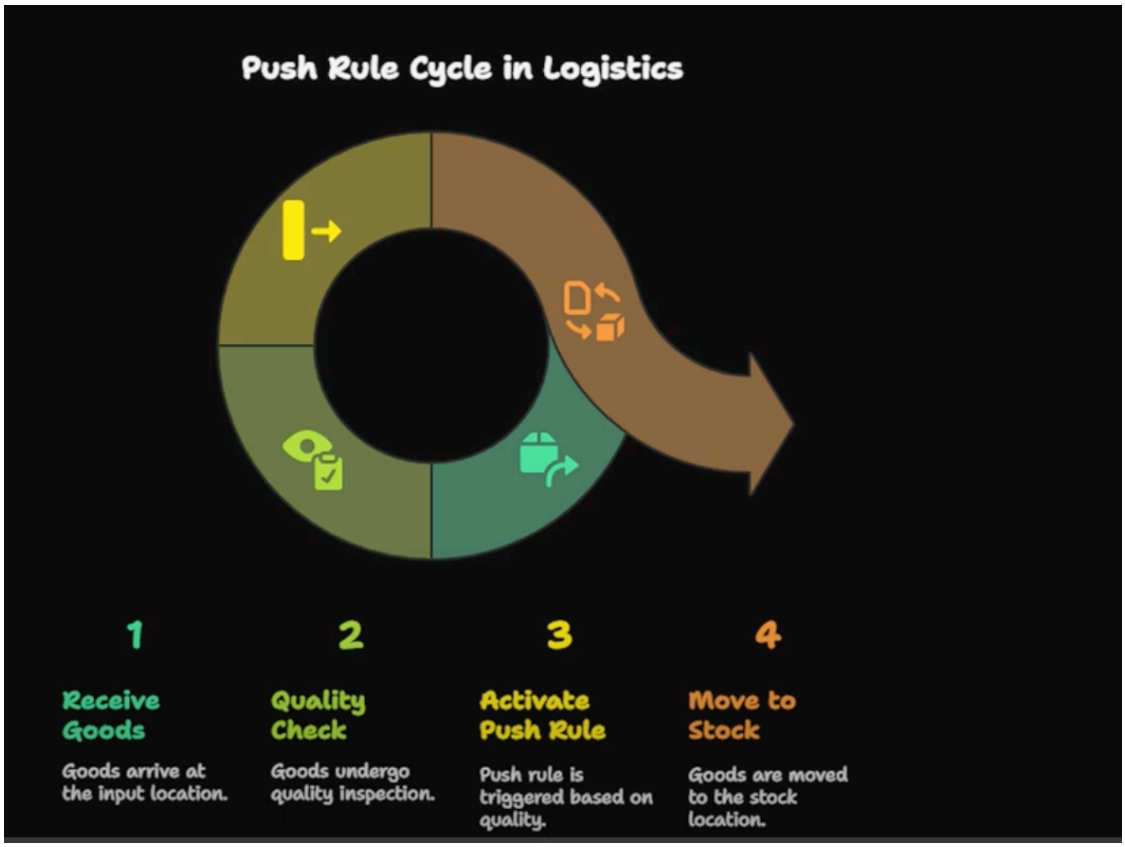
Pull Rule

Pull rule is triggered when there is a demand for a product in a specific location. It pulls stock from source location to fulfil that demand.For example, when SO is confirmed a pull rule can move products from stock location to the delivery zone to fulfil the order.



Push Rule

It is activated after a product arrives a location and needs to be moved to another. It pushes the product forward to the next location automatically. For instance when goods are received in input location a push rule can move them automatically to quality check area or stock location.



Configuration

Inventory > Warehouses > Select Warehouse > Enable three steps in Incoming & Outgoing shipments.

YourCompany: Receive in 3 steps (input + quality + stock)

YourCompany: Deliver in 3 steps (pick + pack + ship)

RULES

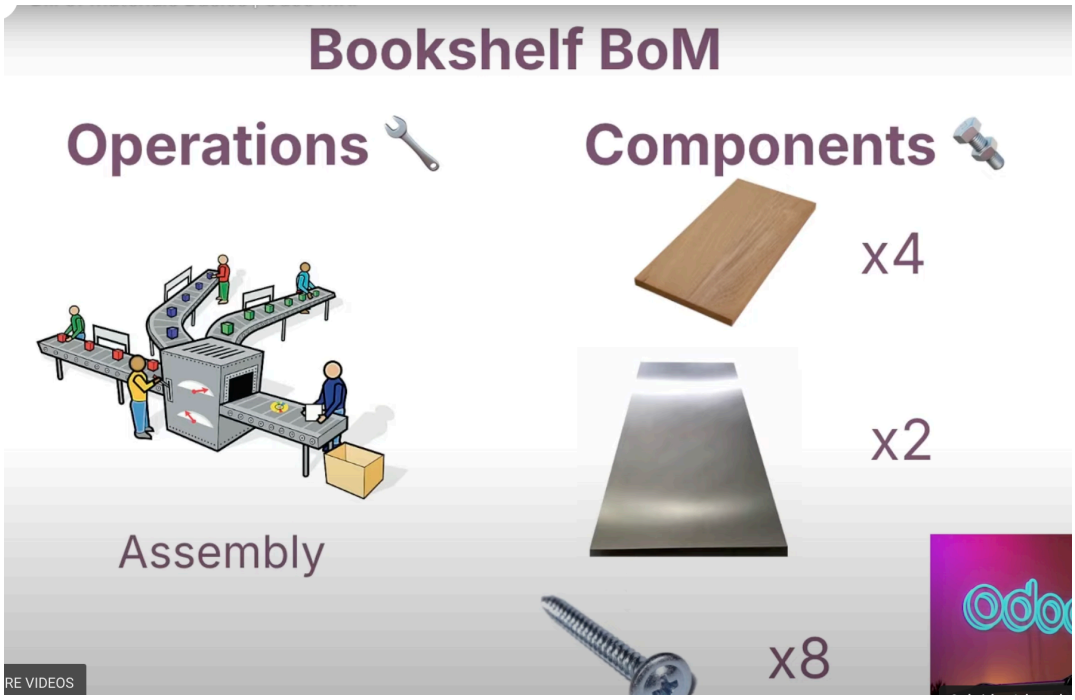
Action	Source Location	Destination Location
Pull From	WH/Stock	Partners/Customers
Push To	WH/Packing Zone	WH/Output
Push To	WH/Output	Partners/Customers
Add a line		

▼ Manufacturing

▼ Terms:-

1. Bill of Materials(BOM)

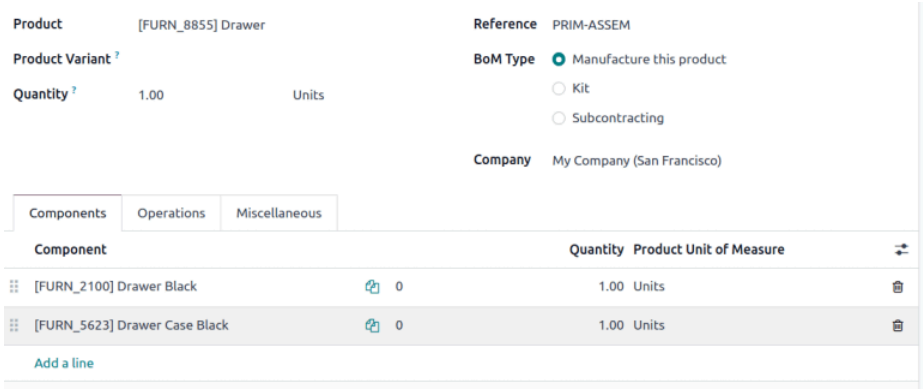
   A *bill of materials* (or *BoM* for short) is a document that defines the quantity of each component required to make or deliver a finished product. It can also include various operations and the individual step guidelines needed to complete a production procedure. This Bill of Material can be used while creating a manufacturing order and the components mentioned in the respective BoM will automatically appear in the Components tab of the manufacturing order.



BoM setup

To create a BoM, go to **Manufacturing app ▸ Products ▸ Bills of Materials** and click **New**.

Next, set the **BoM Type** to **Manufacture this Product**.



Components

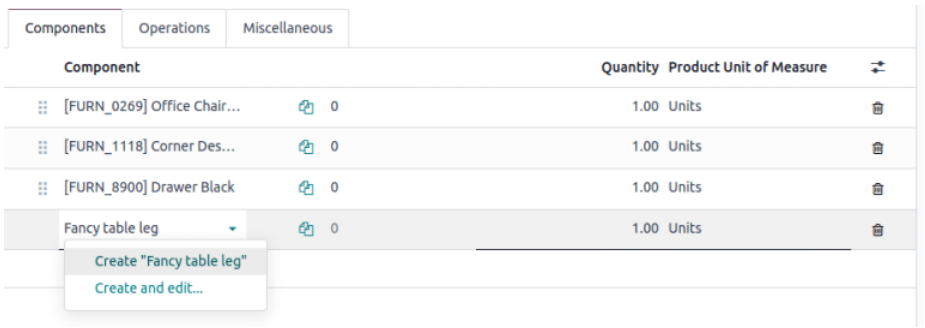
Components are individual parts or items that are used to build a product.

It is a distinct item that serves a specific function within the final product.

Examples of Components:

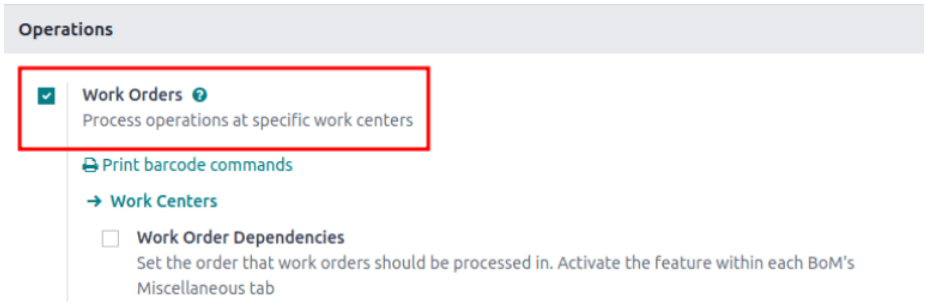
- In a car: Engine, wheels, battery, or dashboard.
- In electronics: Resistors, microchips, or display screens.
- In furniture: Legs, cushions, or hinges.

In the **Components** tab of a BoM, specify components used to manufacture the product by clicking **Add a line**. From the **Components** drop-down menu, select from existing products or create a new product by typing the name and selecting either the **Create ” “** option to quickly add the line item, or the **Create and edit...** option to add the component and continue to its configuration form.



Operations

Add an *operation* to a BoM to specify instructions for production and register time spent on an operation. To use this feature, first enable the *Work Orders* feature by going to **Manufacturing app ‣ Configuration ‣ Settings**. In the **Operations** section, tick the **Work Orders** checkbox to enable the feature.



Next, navigate to the BoM by going to **Manufacturing app ‣ Products ‣ Bill of Materials** and selecting the desired BoM. To add a new operation, go to the **Operations** tab, and click **Add a line**.

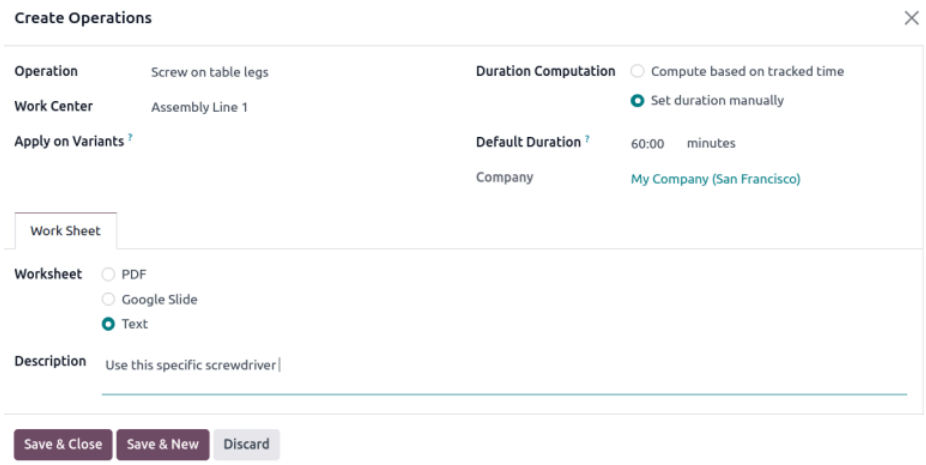
Doing so opens the **Create Operations** pop-up window, where the various fields of the operation are configured:

- **Operation**: name of the operation.
- **Work Center**: select existing locations to perform the operation, or create a new work center by typing the name and selecting the **Create ” “** option.
- **Apply on Variants**: specify if this operation is only available for certain product variants. If the operation applies to all product variants, leave this field blank.
- **Duration Computation**: choose how time spent on the operation is tracked. Opt for**Compute based on tracked time** to use the operation’s time tracker

or **Set duration manually** if operators can record and modify time themselves.

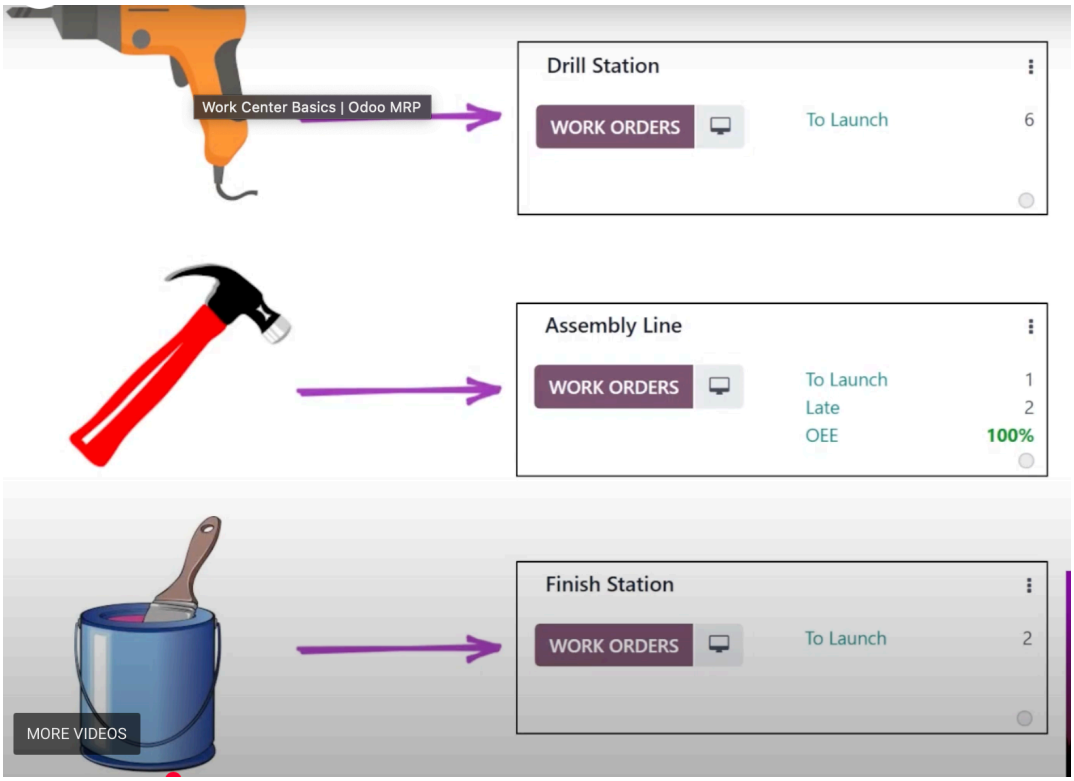
Choosing the **Compute based on tracked time** option enables the **Based on last __ work orders** option, which automatically estimates the time to complete this operation based on the last few operations. Choosing **Set duration manually** enables the **Default Duration**field instead.

- **Default Duration**: estimated amount of time to complete the operation; used for planning manufacturing orders and determining work center availability.
- **Company**: specify the company the BoM is available in.



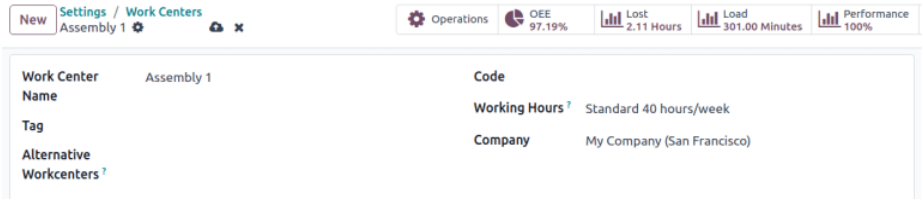
2. Work Center

   In Odoo, *work centers* are used to carry out manufacturing operations at specific locations. *Work centers* are where **Manufacturing** work orders are processed, and can be used to track costs, make schedules, plan capacity, organize equipment, and track efficiency.



Create a new work center

In the **Manufacturing** app, select **Configuration ▸ Work Centers** and click the **New** button to open a new work center form.

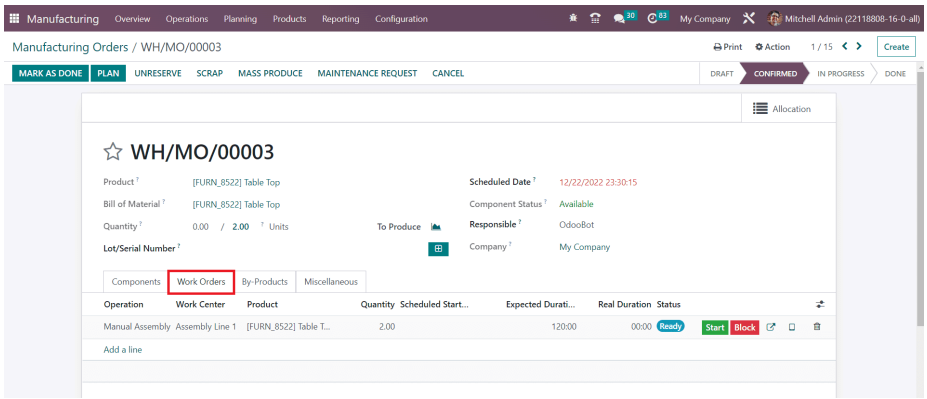


- **Work Center Name**: the label for the work center used to select it on a work order or on the reporting dashboards
- **Tag**: reusable labels that can be used to sort work centers in list view
- **Alternative Workcenters**: where a work order should be carried out if this work center is not available
- **Code**: reference id for the work center that is displayed in the list view
- **Working Hours**: the hours that work center can be used during the week

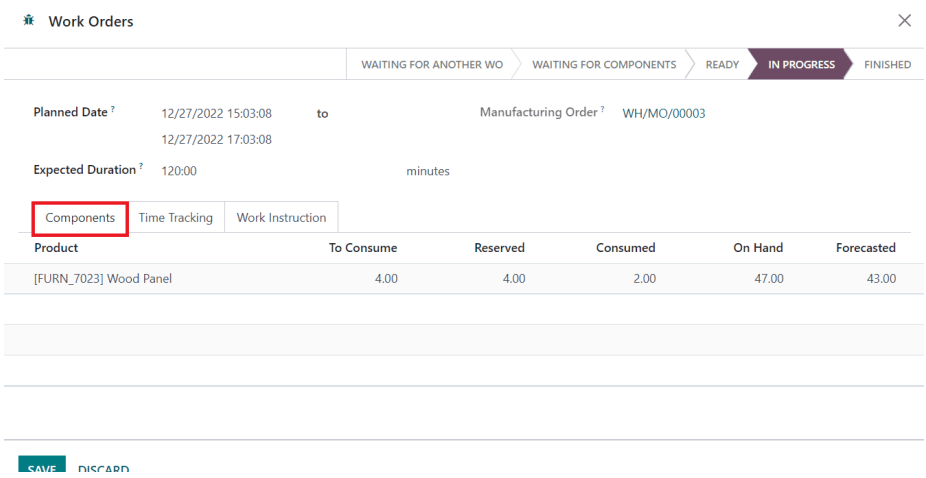
3. Work Orders

A manufacturing order comprises various operations to get a perfectly finished product. In the Odoo Manufacturing module, you can specify all these operations as Work Orders while configuring a manufacturing order. These work orders will process operations at specific work centers. In order to get this option in the Manufacturing module, activate the **Work Orders** feature from the **Settings** menu.

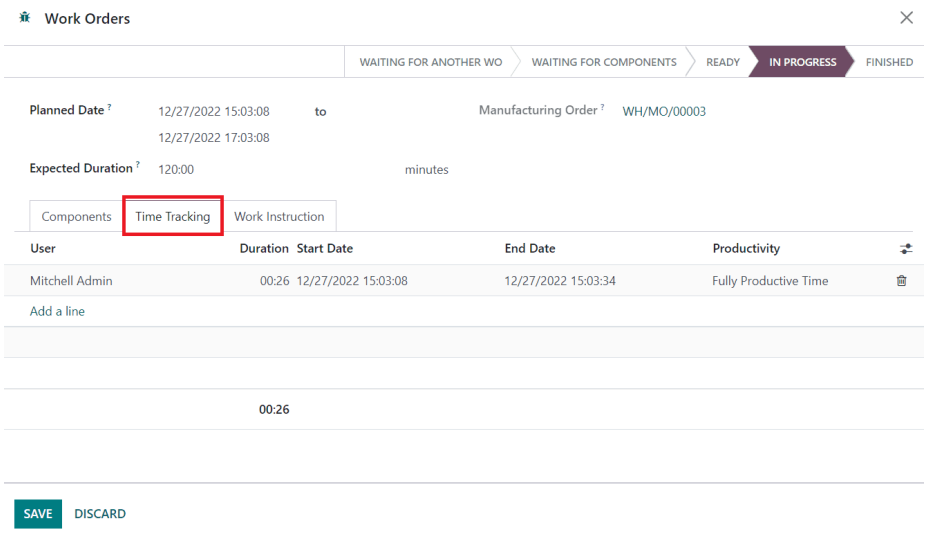
All operations related to a manufacturing process can be added under the **Work Orders** tab of the respective manufacturing order.



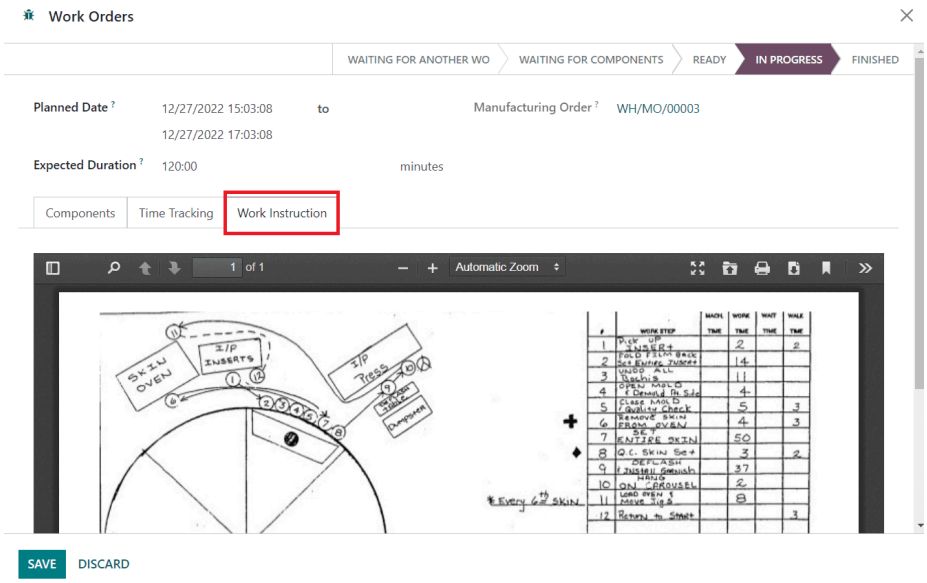
The external link available in the operation line can be used to open the respective work order.



It shows the Planned Date, Manufacturing Order, and Expected Duration of the work. Under the **Components** tab, you can observe the components required in this work order.



The **Time Tracking** tab will show the tracked time of this work order with the details of the User, Duration, Start Date, End Date, and Productivity.



You can mention the instructions for the work order under the **Work Instructions** tab as shown in the image above.

▼ BOM Types:-

Subcontracting

It is the process of a company engaging a third-party manufacturer, or subcontractor, to manufacture products that are then sold by the contracting company.

This means that the contracting company only has to worry about what happens to subcontracted products once they are produced.

Kit

a *kit* is a type of bill of materials (BoM) that can be manufactured and sold. Kits are sets of unassembled components sold to customers.

▼ WorkFlow:-

Prerequisites

- **Modules**: Ensure the "Manufacturing" and "Inventory" modules are installed in Odoo.
- **Configuration**:
  - Set up product. Select Product > Inventory Tab > Enable Manufacture.
  - Set up products with Bills of Materials (BoMs) in **Manufacturing > Products > Bills of Materials**.
  - Define work centers and routings in **Manufacturing > Configuration > Work Centers** and **Routings** if using multi-step processes.
  - Configure warehouse settings for manufacturing steps (1, 2, or 3 steps) in **Inventory > Configuration > Warehouses**.

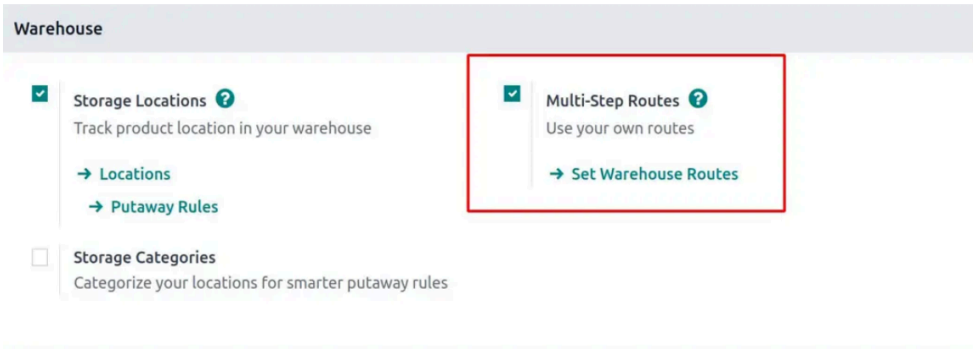
1. **Create a Manufacturing Order (MO)**
   - Navigate to **Manufacturing > Operations > Manufacturing Orders**.
   - Click **New** to create a new MO.
   - Select the **Product** to manufacture from the dropdown.
   - The **Bill of Materials** field auto-populates based on the product. If multiple BoMs exist, choose the appropriate one.
   - Specify the **Quantity to Produce**.

- Add a **Deadline** and **Planned Date** for production scheduling.
- Click **Save**.

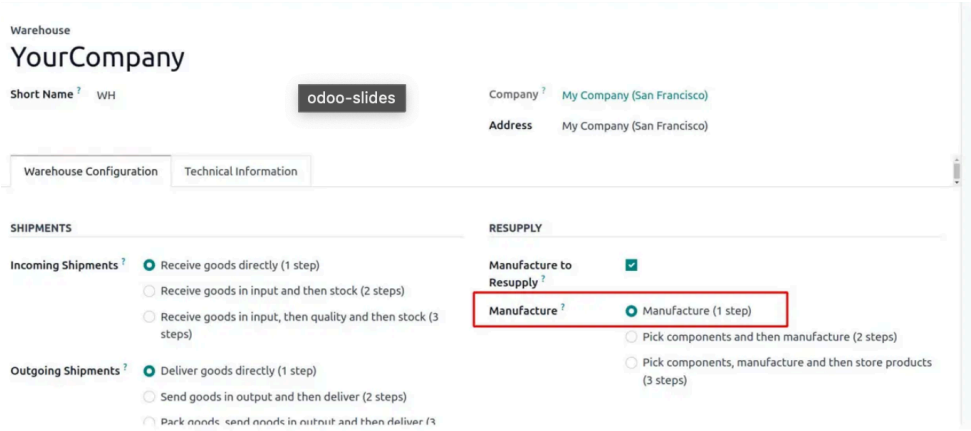
2. **Check Component Availability**
   - After saving, click **Check Availability** to verify if raw materials (components) listed in the BoM are in stock.
3. **Process Work Orders (Optional)**
   - If using routings and work centers (multi-step manufacturing):
     - Go to the **Work Orders** tab on the MO.
     - For each work order, click **Start** to begin, triggering a timer.
     - Complete each step and click **Done** to finish the work order.
4. **Produce the Product**
   - Once all components are ready and work orders (if any) are completed, click **Produce** on the MO.

▼ Manufacturing in One Step

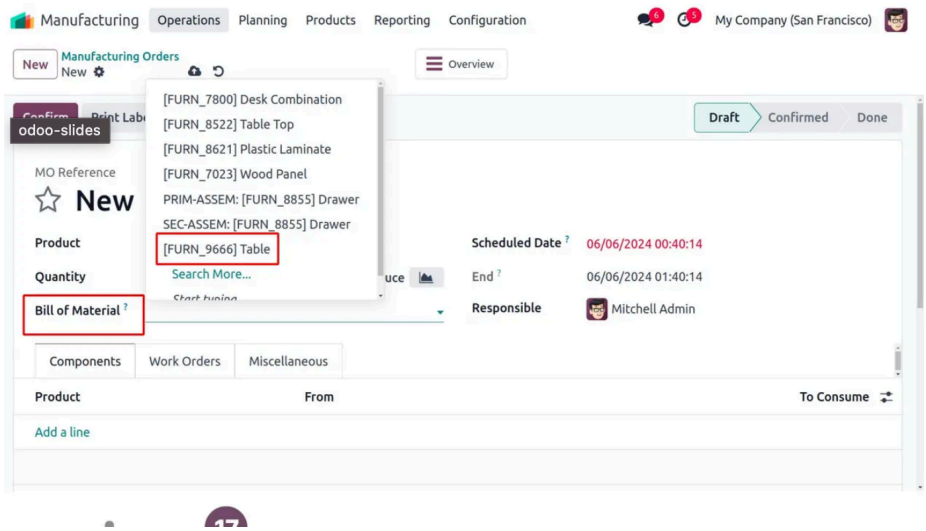
1. Enable multi step routes for warehouse.



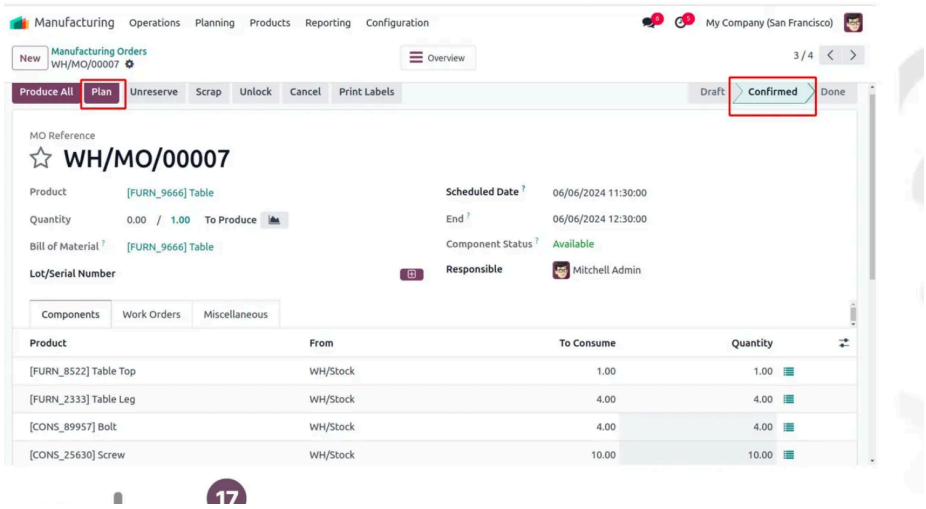
2. In the warehouse choose the manufacture in one step.



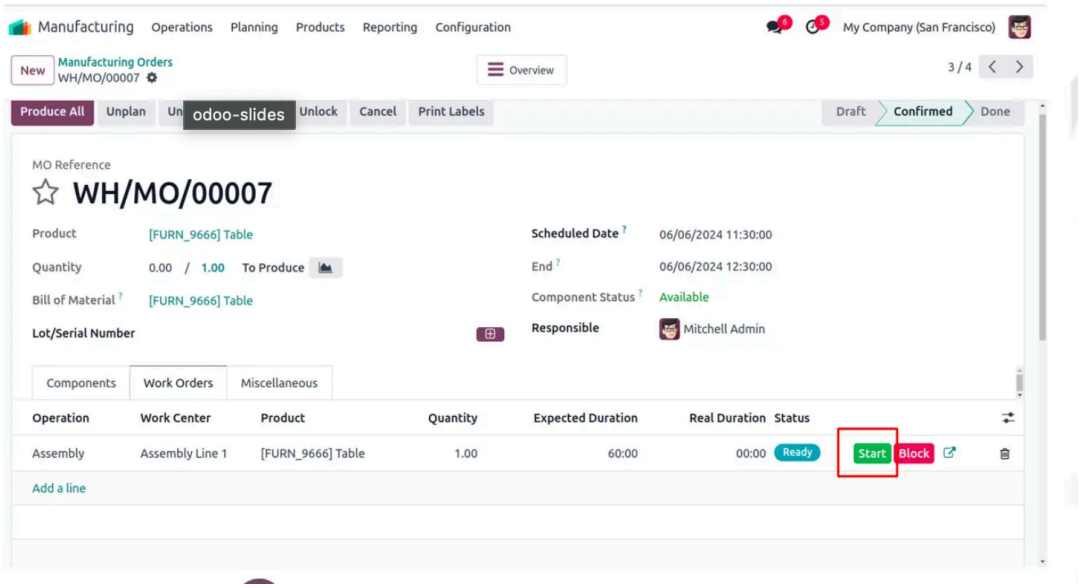
3. Create New MO, choose the BOM of the product which we need to manufacture the product.



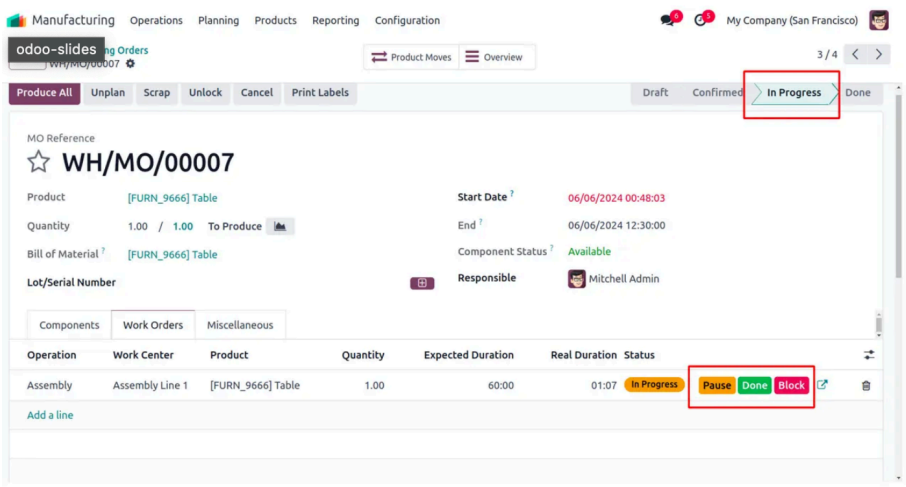
4. Select Product from Components, then click on Confirm, we can plan the manufacture by clicking on the Plan button.



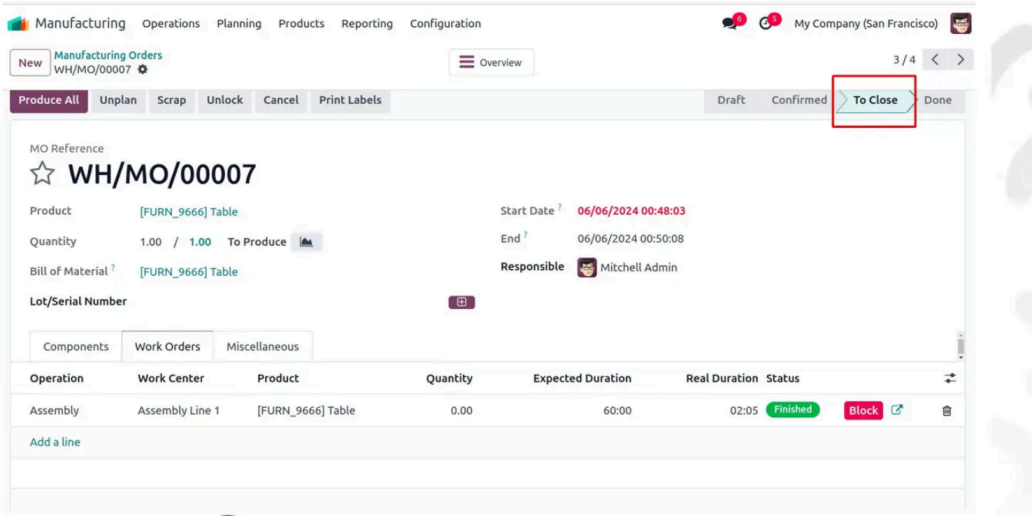
5. Start the manufacturing process by clicking on the start button in operations tab.



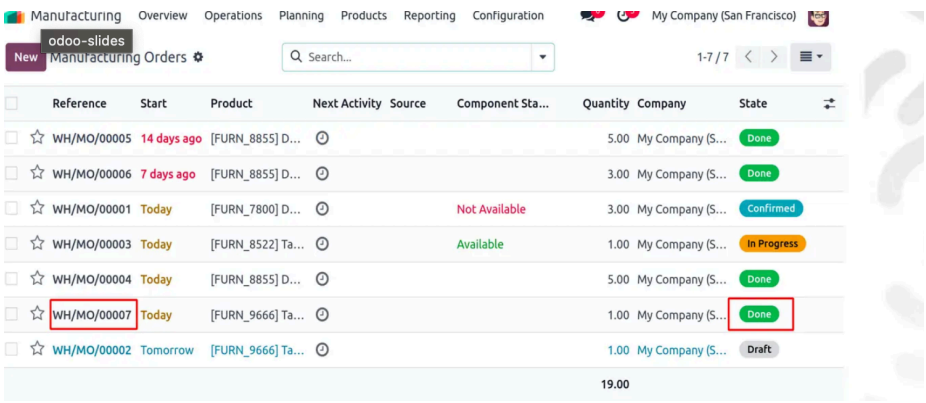
6. Now the MO will be in Progress state & we can either pause or done or block the manufacturing process.



7. Click on Done and the status change to Finished operation. Now the MO will be in Close Stage.

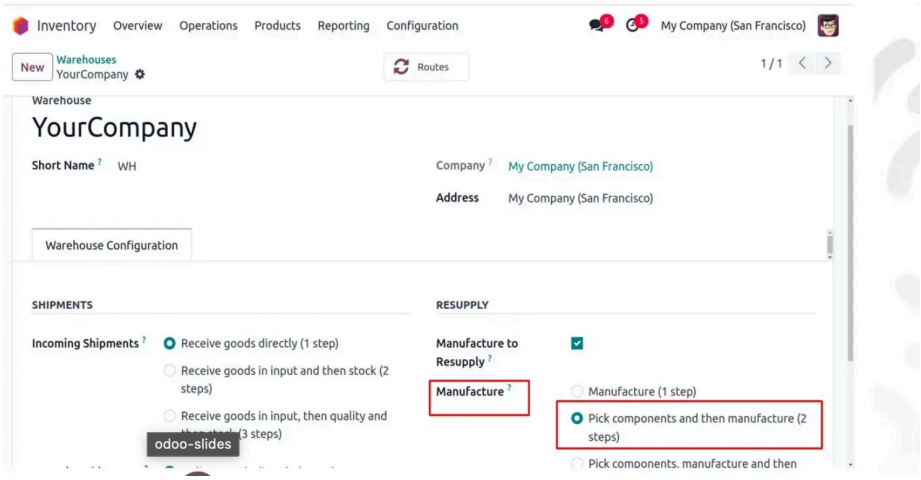


8. Click on Produce All and MO is in done state.

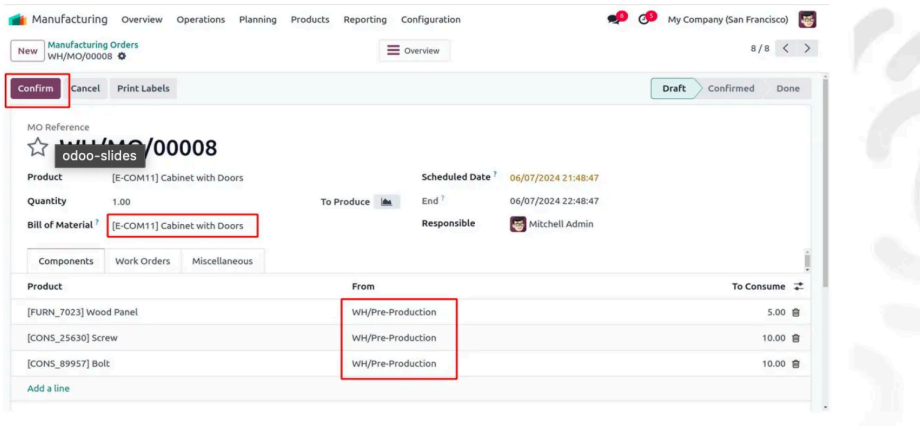


▼ Manufacturing in Two Step

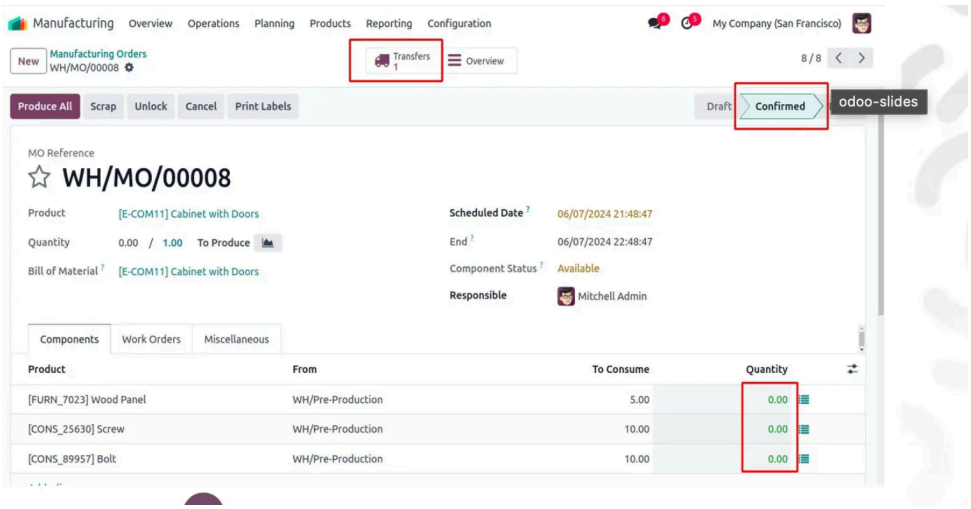
1. In the warehouse choose the manufacture in two step i.e first pick the components and then manufacture.



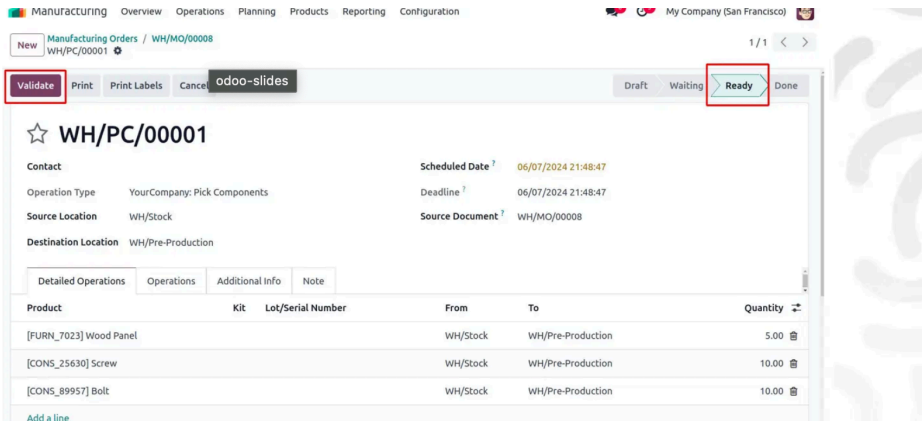
2. Create New MO, we can confirm the MO by clicking on Confirm button. Here the from location is pre-production as we have chosen two step manufacturing process.



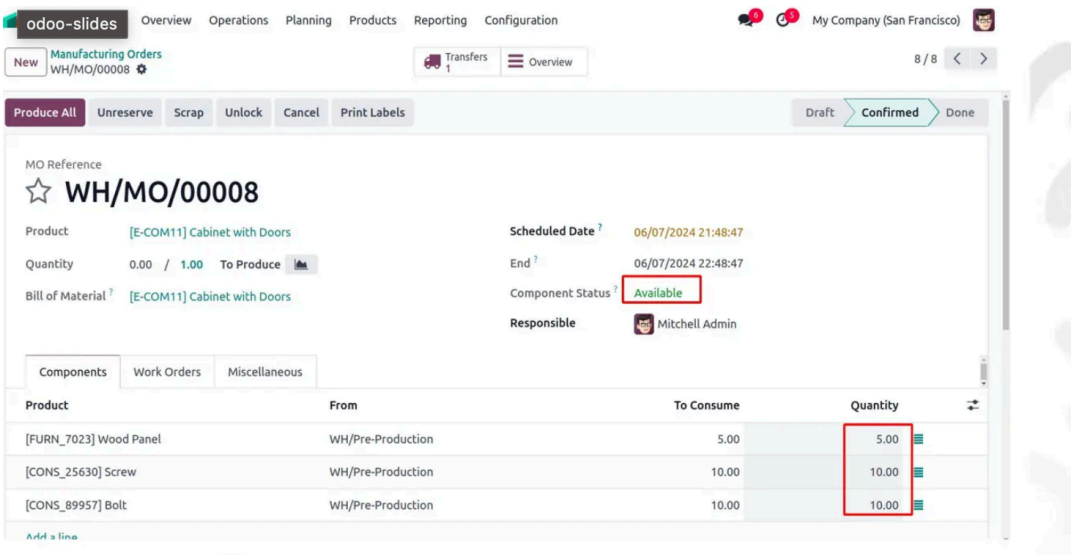
3. After confirming a transfer will be created and only after validating the transfer the operation will be in ready stage.



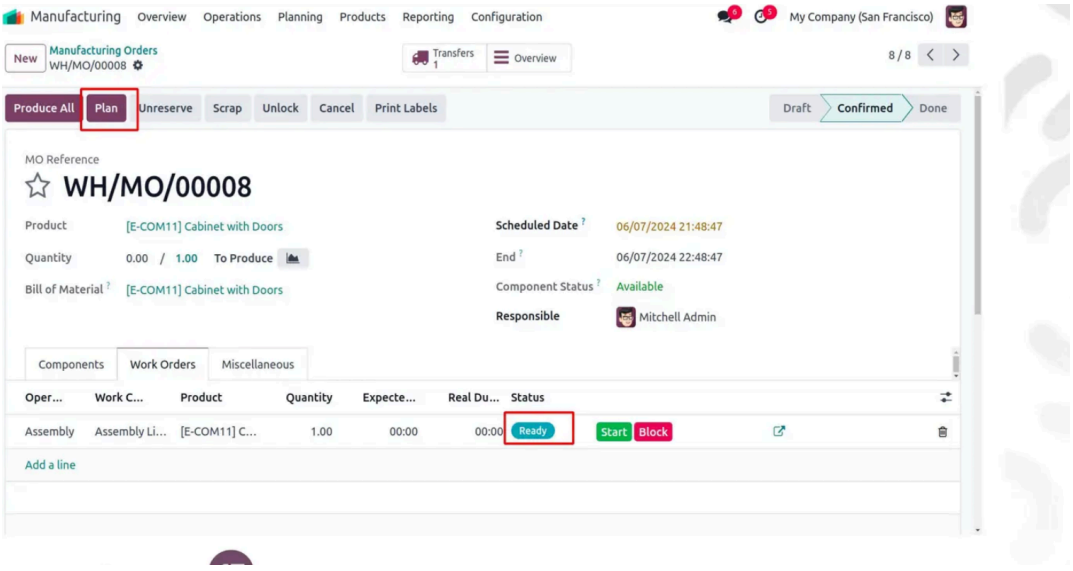
4. We can validate the transfer by clicking on Validate button in the transfer.



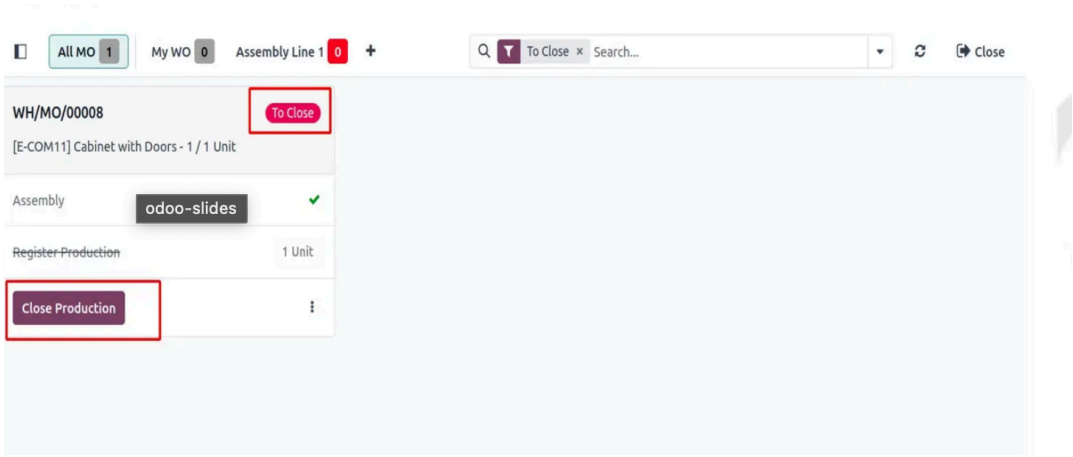
5. Now the quantity is changed from zero to respective quantities in BOM after validating it.



6. Next we can plan the MO so that the MO will be in progress and then start the manufacturing by clicking on Start button.

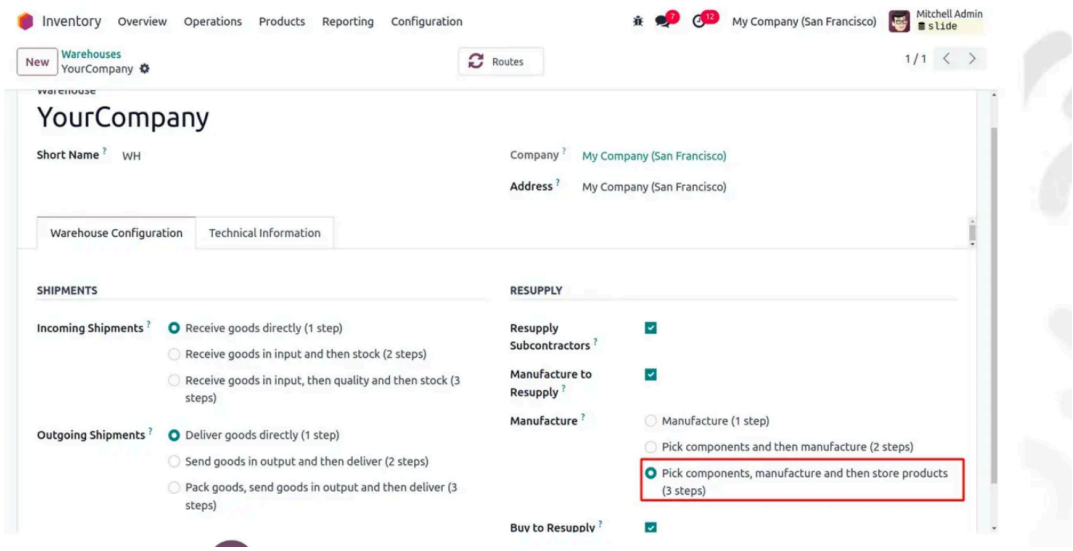


7. To close the MO we can move to shop floor module. We can close the production by clicking on Close button. MO will be in Done state.

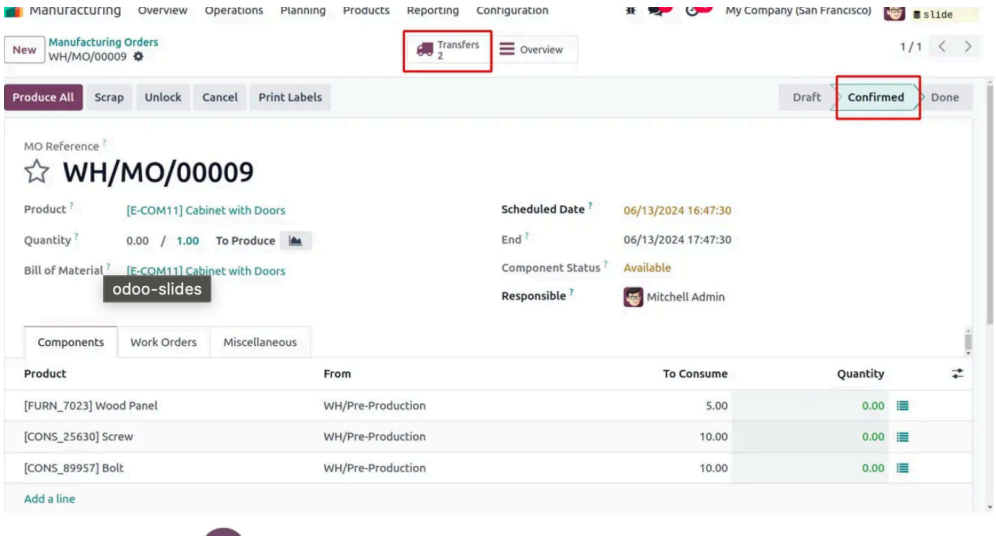


▼ Manufacturing in Three Step

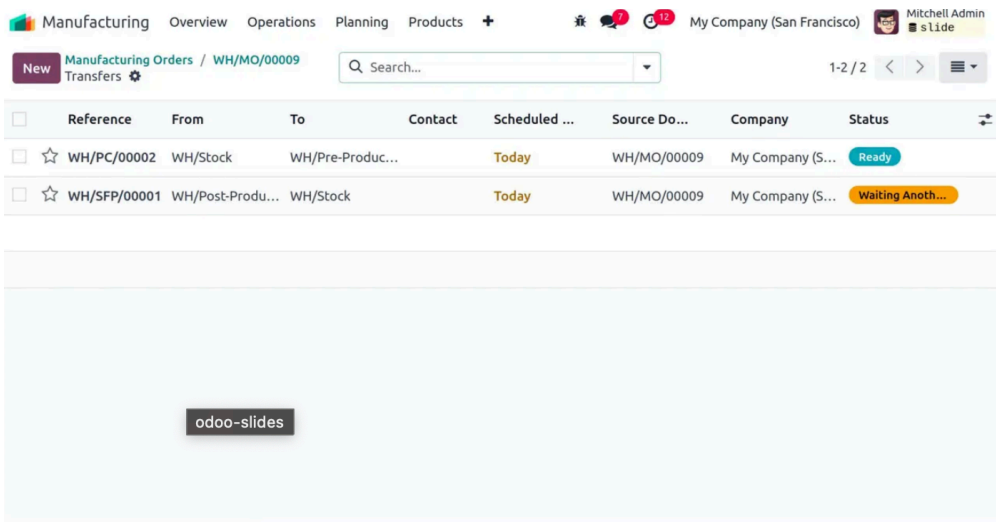
1. In the warehouse choose the manufacture in three step i.e first pick the components, manufacture and then store products.



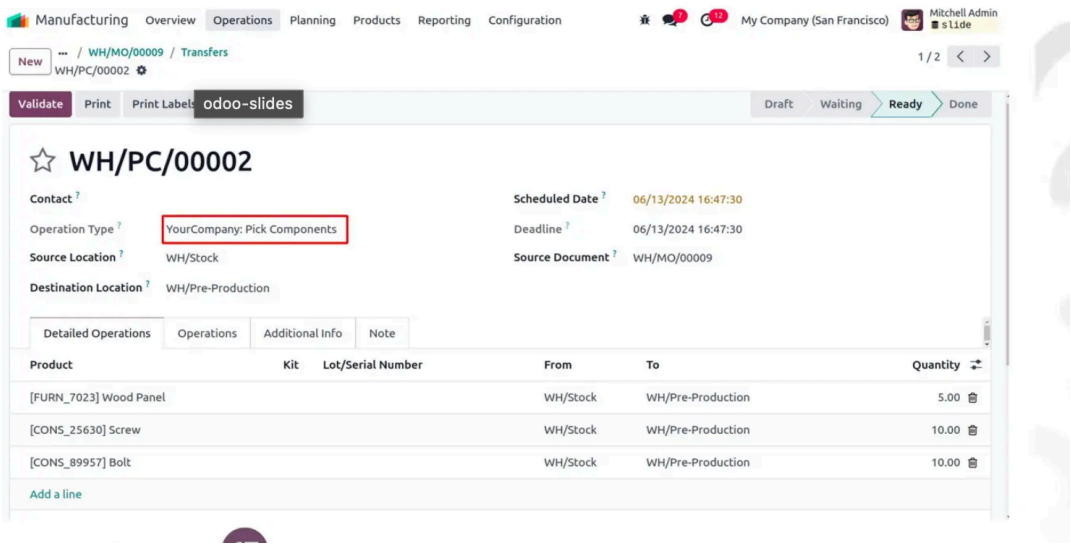
2. Create New MO, we can confirm the MO by clicking on Confirm button. Since this is a 3 step process we can see two transfers created for this.



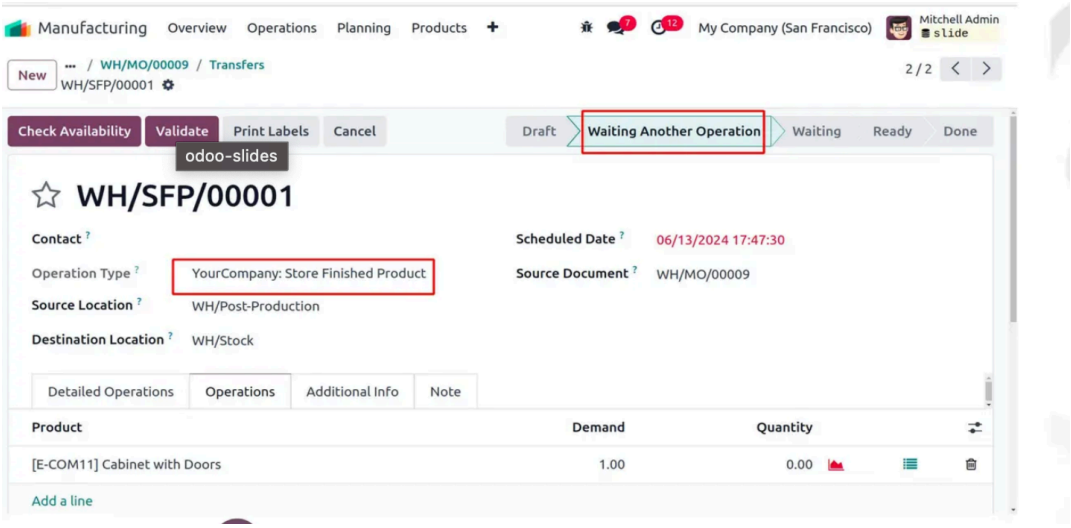
3. These are the transfers created in ready and waiting for another operation for the manufacturing order.



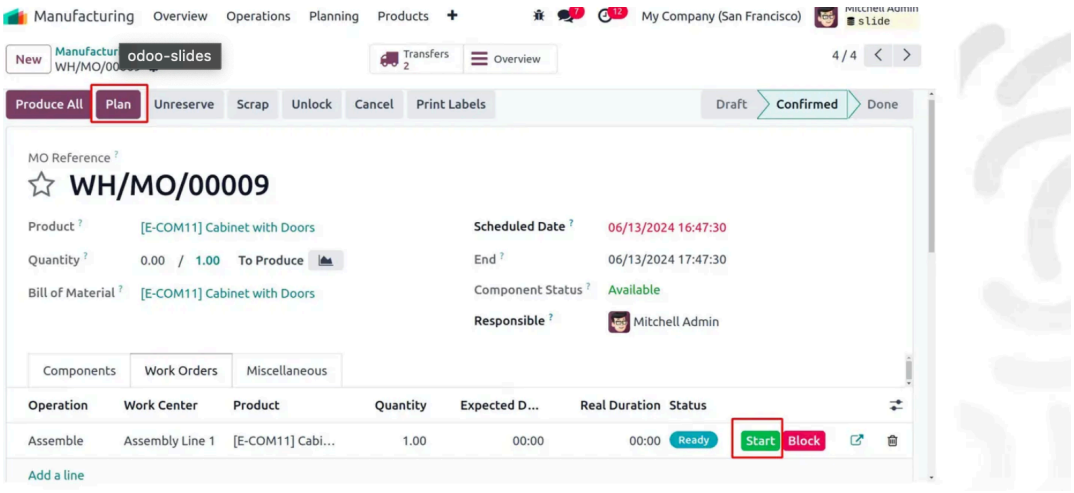
4. This is the transfer created for picking the components. We can validate the transfer by clicking on validate button so that components will move to production area.



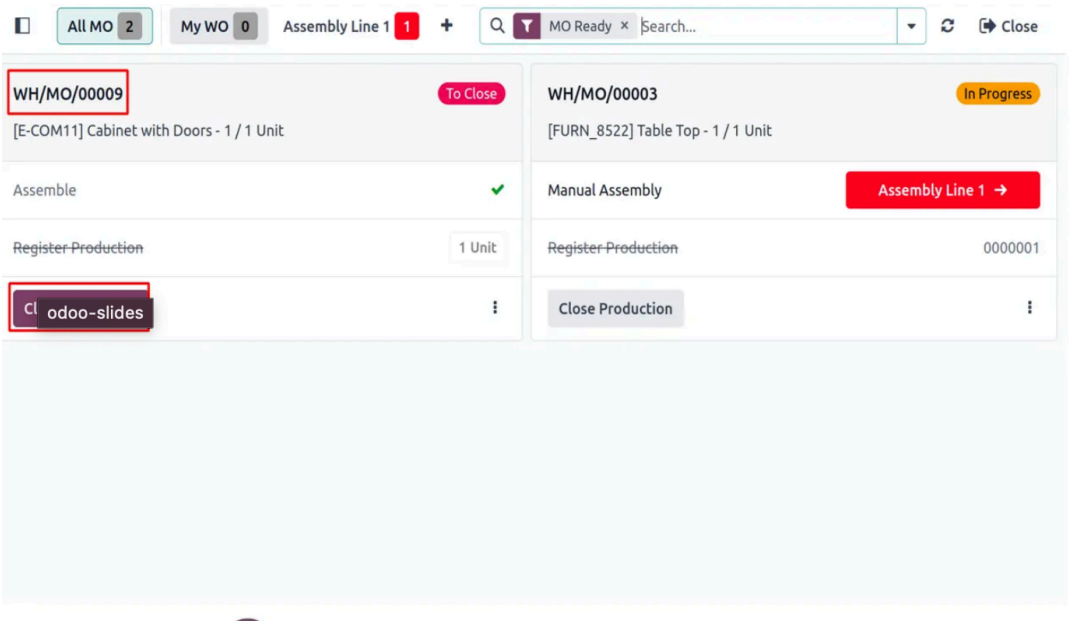
5. Other transfer is created for moving the finished product to the stock. But this will be still in waiting another operation stage, because we can complete this transfer only when manufacturing is done.



6. To complete the manufacturing process, we can go back to MO and plan the MO so that the MO will be in progress and then start the manufacturing by clicking on Start button.



7. After completing the process we can click on Done button and the work order will be in finished status and the MO will be in To close state. To close the MO we can move to shop floor module. We can close the production by clicking on Close button. MO will be in Done state.



8. After moving the MO to Done state. If we go to the transfers, we can see the second transfer will be in ready state and we can complete the transfer by clicking on validate button.

▼ Accounting

▼ Credit Note & Debit Note

Credit Note

A credit note is a document issued by a seller to a buyer to correct a mistake on an order or an invoice, or to refund an amount paid for products for damaged goods.

Debit Note

Debit notes are typically issued by the buyer to the seller. A debit note is needed where the ***value of the invoice changes*** due to additional goods being shipped.

▼ Fiscal Position

Fiscal positions are used to adapt taxes and accounts for particular customers or sales orders/invoices

▼ QWEB

▼ Internal vs external layout

- **internal_layout** is used on documents **intra company**
- **external_layout** is used on external documents available for partners (logo + full info about company)

▼ Remove header & footer

If you need to remove header and footer you can comment this line ***<t t-call="web.external_layout">*** also don't forget to comment the closing tag of this t element in the last.

▼ OWL

- A JavaScript framework (similar to React/Vue)
- Basic building block is Component
- A templating system (extension of QWeb)
- Some hooks/utility functions

▼ Self vs this

this: references the object executing the current function or refer to an object in which a method is called.

Function/global scope: window object

arrow function: retains the value of context from the surrounding scope where the arrow function is defined.

method: object itself

strict mode: undefined in regular functions

self: Browser defines window.self as window. Self can be redefined within an enclosing scope

let self = this.

▼ Class Include vs extends

When you `extend`, instances from the parent class remain untouched, but instances from the new child class will have the extended features.

```
odoo.define('my_module.MyClassExtension', function (require) {
    'use strict';
    var core = require('web.core');
    var MyClass = require('some.module.MyClass');
    var _t = core._t;
    MyClass.extend({

        // New methods
        myNewMethod: function () {
            // New functionality added to MyExtendedClass
        },

        myMethod: function () {
            // Calling parent class method
            this._super();
            // Adding new functionality to the parent method
            // ...
        },
    });
    return MyClass;
});
```

OTOH, when you `include`, you are adding the new features to the prototype of the parent class, which means that automatically all instances of such class include the extended behavior. The include method is used to add additional behavior or properties to an existing class without overriding any existing methods. It is primarily used for mixins or to add generic behavior to multiple classes.

▼ DOM

HTML DOM (Document Object Model) is a programming interface that represents the structure of a web page as a tree of objects, where each object corresponds to an element, attribute, or piece of text in the HTML or XML document allowing you to dynamically change or interact with the content and structure of the page.

Virtual DOM

It is a light weight in memory representation of the real DOM. When changes occur, Owl updates only the necessary parts of the real DOM, instead of re-rendering everything.

▼ Widget

- It is a basic building block of user interface in odoo framework. It can be considered as an element which helps to display information or provides a specific way for the user to interact with the interface.
- class in web app (web.widget)

Features of Widget Class:

1. Parent or child relationships between Widgets.

2. Extensive lifecycle management with safety features.

3. Automatic rendering with Qweb.

4. Various utility functions for interaction with the outside environment.

```
odoo.define('customer_ebz.notification', function (require) {
    "use strict";
    var core = require('web.core');
    var Widget = require('web.Widget');
    var websiteRootData = require('website.root');

    var NotificationSelect = Widget.extend({
//    selector: '.option_select',
    events:{
    'change .option_select': '_onchangeOptions',
    },
    init: function () {
            this._super.apply(this, arguments);
    },
    start: function (editable_mode) {
```

```
        this._super.apply(this, arguments);
        this.multi_select();
        this._onchangeOptions();
    },
     multi_select: function(e){
          $('option').mousedown(function(e) {
          e.preventDefault();
          $(this).prop('selected', !$(this).prop('selected'));
          return false;
});
    },
    });
        websiteRootData.websiteRootRegistry.add(NotificationSelect, '.select-checkbo
    })
```

Types of Widget:-

1. **Ribbon widget**

```
<widget name="web_ribbon" text="PAID" bg_color="bg-success" />
```

2. Color Picker

```
<field name="color" widget="color_picker"/>
```

3. **Badge**

This is generally used in list view. Field types – Many2one, Selection

```
<field name="state_id" widget="badge" />
```

4. **Many2one_avatar_user**

This widget will show the name of the user along with their avatar photos.

```
<field name="user_id" widget="many2one_avatar_user"/>
```

Widget Lifecycle:-

The lifecycle of the widget is as the following

1. init

2. willstart

3. rendering

4. start

5. destroy

init()

The init() function is the constructor. They initialize the widget.

```
init: function (parent) {
    this._super(parent);
    // stuff that you want to init before the rendering
},
```

willstart()

The willstart method is a hook that has to return a deferred value. This method will be called when the widget has been created and appending it to the DOM is under progress. This method is mainly used to fetch data from the server.

```
    willStart: function () {
        // async work that need to be done before the widget is ready
        // this method should return a deferred
    },
```

rendering()

rendering() of a widget is done automatically by the framework. The framework checks for a defined template in the widget. If a template is defined then widget key in the Qweb template is used to read the value from the widget. If no template is defined then the tag name key is read and a DOM element is created.

start()

After the rendering process is complete, the start method will be called automatically. This method is useful when we have to perform some specialized post_rendering works such as setting up a library.

```
start: function() {
    this.$(".my_button").click(* an example of event binding * );
},
```

destroy()

The destroy method would destroy the widget. This means the following operations would be performed.

1. Clean up operations

2. remove widget from component tree

3. unbinding events

▼ Web Client

In Odoo, the web client is the browser-based interface that allows users to interact with Odoo's business applications and modules. It is a Single Page Application (SPA), meaning the page does not reload entirely during use; instead, it dynamically fetches and updates data from the server as needed.

Example:

When a user logs into Odoo and navigates to the “Sales” module, the web client renders a kanban or list view of sales orders. Clicking an order triggers a client action to display a form view, all handled dynamically within the SPA without reloading the page.

▼ Component

- Basic building block of OWL
- Component are async by default.
- XML template
- props

> https://github.com/odoo/owl/blob/master/doc/reference/component.md

An Owl component is a small class which represents some part of the user interface. It is part of a component tree, and has an environment (`env`), which is extended from

a parent to its children.

OWL components are defined by subclassing the `Component` class. For example, here is how a `Counter` component could be implemented:

```
const { Component, xml, useState } = owl;

class Counter extends Component {
  static template = xml`
    <button t-on-click="increment">
      Click Me! [<t t-esc="state.value"/>]
    </button>`;

  state = useState({ value: 0 });

  increment() {
    this.state.value++;
  }
}
```

▼ Registries

- Global key-value mapping
- It is the primary extensibility mechanism (all addons can register something ⇒ web client will pick it up)

```
import { Registry } from "@web/core/registry";

const myRegistry = new Registry();
myRegistry.add("some key", "some value");

console.log(myRegistry.get("some key"));
```

- Each registry has sub-registries
- There is a global registry
- registry can maintain an order (with sequence)

```
// .category method gets a sub registry
const fieldRegistry = registry.category("fields");
```

```
fieldRegistry.add("account.SomeField", ...);
```

Examples

- actions
- fields
- services
- views
- error_handlers
- parsers
- systray

▼ Services

- Piece of code that stay alive for the duration of web client
- It provide some functionality to the rest of web client
- Can be useful to hold some state
- e.g. notification, orm , action, bus, etc.
- kept in a category on the registry

Service API

- contains start() method :- the return value is the value of service
- optionally : list of dependencies
- can be utilised with useState() hook
- must be registered in services category

```
import { registry } from "@web/core/registry";

const myService = {
  dependencies: ["notification"],
  start(env, { notification }) {
    function notify() {
        let counter = 1;
        setInterval() => {
            notification.add(`Tick Tock ${counter++}`);
```

```
        }, 5000);
    }
    return { notify };
};

registry.category("services").add("myService", myService);
```

▼ Props

- Props are used to pass data from a parent component to a child component, ensuring a unidirectional flow of information and render the dynamic content.They are immutable, means a child component cannot modify the props it receives.
- is a static key, this.props is a component instance.
- we can access props in js using this.props & in template props

```
class Child extends Component {
  static template = xml`<div><t t-esc="props.a"/><t t-esc="props.b"/></div>`;
}

class Parent extends Component {
  static template = xml`<div><Child a="state.a" b="'string'"/></div>`;
  static components = { Child };
  state = useState({ a: "fromparent" });
}
```

In this example, the `Child` component receives two props from its parent: `a` and `b`. They are collected into a `props` object by Owl, with each value being evaluated in the context of the parent. So, `props.a` is equal to `'fromparent'` and `props.b` is equal to `'string'`.

Note that `props` is an object that only makes sense from the perspective of the child component.

Definition

The `props` object is made of every attributes defined on the template, with the following exceptions:
• every attribute starting with
`t-` are not props (they are QWeb directives),

```
<div>
    <ComponentA a="state.a" b="'string'"/>
    <ComponentB t-if="state.flag" model="model"/>
</div>
```

the `props` object contains the following keys:

- for `ComponentA` : `a` and `b`,
- for `ComponentB` : `model`,

▼ Hooks

https://github.com/odoo/owl/blob/master/doc/reference/hooks.md

- It help reusing stateful logic between components and use state in functional components, without writing a class.
- Starts with on or use
- Note:- need to be define in setup()

Lifecycle Hooks

Hook	Description
onWillStart	async, before first rendering
onWillRender	just before component is rendered
onRendered	just after component is rendered
onMounted	just after component is rendered and added to the DOM
onWillUpdateProps	async, before props update
onWillPatch	just before the DOM is patched
onPatched	just after the DOM is patched
onWillUnmount	just before removing component from DOM
onWillDestroy	just before component is destroyed
onError	catch and handle errors

Other Hooks

`useState`

The `useState` hook is certainly the most important hook for Owl components: this is what allows a component to be reactive, to react to state change.

The `useState` hook has to be given an object or an array, and will return an observed version of it (using a `Proxy`).

```
const { useState, Component } = owl;

class Counter extends Component {
  static template = xml`
    <button t-on-click="increment">
       Click Me! [<t t-esc="state.value"/>]
    </button>`;

  state = useState({ value: 0 });

  increment() {
    this.state.value++;
  }
}
```

It is important to remember that `useState` only works with objects or arrays. It is necessary, since Owl needs to react to a change in state.

useRef

The `useRef` hook is useful when we need a way to interact with some inside part of a component, rendered by Owl. It only work on a html element tagged by the `t-ref` directive:

```
<div>
    <input t-ref="someInput"/>
    <span>hello</span>
</div>


class Parent extends Component {
  inputRef = useRef("someInput");

  someMethod() {
    // here, if component is mounted, refs are active:
```

```
    // - this.inputRef.el is the input HTMLElement
  }
}
```

▼ Environment

shared object that contains stuff (like services)

An environment is a shared object given to all components in a tree. It is not used by Owl itself, but it is useful for application developers to provide a simple communication channel between components (in addition to the props).

`useSubEnv` and `useChildSubEnv`

The environment is sometimes useful to share some common information between all components. But sometimes, we want to *scope* that knowledge to a subtree.

For example, if we have a form view component, maybe we would like to make some `model` object available to all sub components, but not to the whole application. This is where the `useChildSubEnv` hook may be useful: it lets a component add some information to the environment in a way that only its children can access it:

```
class FormComponent extends Component {
  setup() {
    const model = makeModel();
    // model will be available on this.env for this component and all children
    useSubEnv({ model });
    // someKey will be available on this.env for all children
    useChildSubEnv({ someKey: "value" });
  }
}
```

The `useSubEnv` and `useChildSubEnv` hooks take one argument: an object which contains some key/value that will be added to the current environment. These hooks will create a new env object with the new information:

- `useSubEnv` will assign this new `env` to itself and to all children components
- `useChildSubEnv` will only assign this new `env` to all children components.

As usual in Owl, environments created with these two hooks are frozen, to prevent unwanted modifications.

▼ Network Request(RPC/ORM)

RPC Service

> Note that the `rpc` service is considered a low-level service. It should only be used to interact with Odoo controllers. To work with models (which is by far the most important usecase), one should use the `orm` service instead

RPC allows a client application to request services or functions from a server application located on a different machine over a network. The client and server interact as if they were on the same system, abstracting the complexity of network communication and providing a seamless way to distribute tasks and processing across multiple nodes

```
import { rpc } from "@web/core/network/rpc";

// somewhere else, in an async function:
const result = await rpc("/my/route", { some: "value" });
```

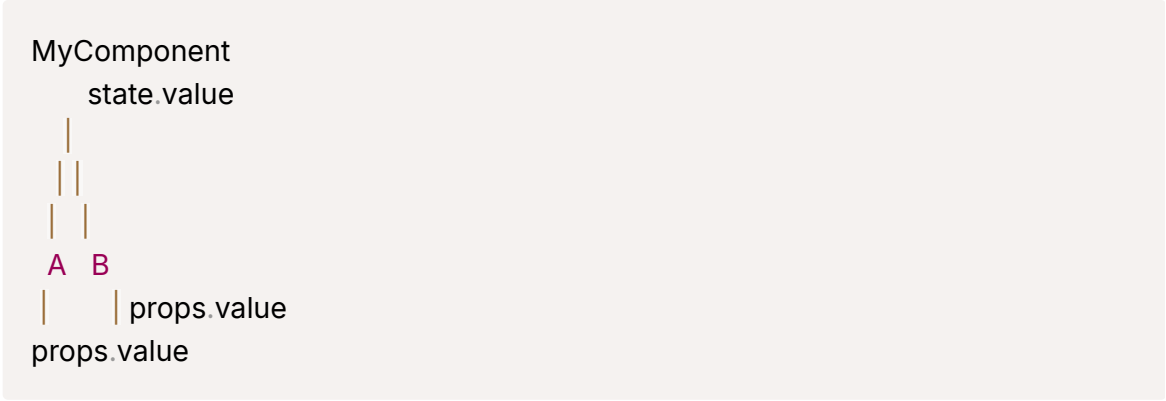
ORM Service

- facilitate easy access to backend ORM
- call public methods of model

```
this.orm = useService('orm')
const resPartner = await
this.orm.call('res.partner','search_read',[[]])
```

▼ State Management

- State generally refers to application data or properties that need to be tracked.
- state should flow from parent to children
- each “piece of state” is owned by 1 component/entity
- only owner should update state
- propagate state to children through props



▼ Events

The heart of event handling in Owl.js is the t-on directive. It's your bridge between user interactions (clicks, keystrokes, etc.) and your component's logic.

```
<button t-on-click="someMethod">Do something</button>
```

The value of the `t-on` expression should be a valid javascript expression that evaluates to a function in the context of the current component. So, one can get a reference to the event, or pass some additional arguments. For example, all the following expressions are valid:

```
<button t-on-click="someMethod">Do something</button>
<button t-on-click="() ⇒ this.increment(3)">Add 3</button>
<button t-on-click="ev ⇒ this.doStuff(ev, 'value')">Do something</button>
```

Modifiers

In order to remove the DOM event details from the event handlers (like calls to `event.preventDefault`) and let them focus on data logic, *modifiers* can be specified as additional suffixes of the `t-on` directive.

Modifier	Description
`.stop`	calls `event.stopPropagation()` before calling the method
`.prevent`	calls `event.preventDefault()` before calling the method
`.self`	calls the method only if the `event.target` is the element itself
`.capture`	bind the event handler in capture mode.
`.synthetic`	define a synthetic event handler (see below)

```
<button t-on-click.stop="someMethod">Do something</button>
```

This will simply stop the propagation of the event.

▼ Assets & bundles

Assets in Odoo

Assets are static files that define the behavior and appearance of Odoo applications. They include:

- **JavaScript (JS)**: Code for client-side logic, such as interactive features or Odoo module systems (e.g., boot.js).
- **CSS/SCSS**: Stylesheets for visual design and layout.
- **XML Templates**: Static files used for rendering QWeb templates, which define the structure of views in Odoo.

These assets are processed differently:

- **JavaScript and CSS/SCSS** are minified (to reduce file size by removing comments and extra spaces) and concatenated into a single file unless in debug mode (debug=assets).
- **XML Templates** are read from the file system on demand, concatenated, and fetched via the /web/webclient/qweb/ controller when the browser loads Odoo.

Bundles in Odoo

Bundles are logical groupings of assets (lists of file paths for JS, CSS, SCSS, or XML) defined in a module's __manifest__.py file under the assets key. Each bundle serves a specific purpose, ensuring only relevant assets are loaded for a given application or context, which optimizes performance. Bundles are declared using **glob syntax** to include multiple files in a single line.

Common Asset Bundles

1. web.assets_common: this bundle contains most assets that are common to the web client, website & the point of sale.
2. web.assets_backend: this bundle contains the code specific to the web client (notably the web client/action manager/ views)
3. web.assets_frontend: this bundle is about all that is specific to the public website: ecommerce, blog, forum....

Operations

1. append: default
2. prepend: add to the beginning
3. before: Add one or multiple file(s) before a specific file.
4. after: Add one or multiple file(s) after a specific file.
5. remove: remove one or multiple files.
6. replace: Replace an asset file with one or multiple file(s).
7. include: nested bundles

▼ Patch

Patching allows you to change an existing method's behavior without totally replacing it. Patching in OWL can be accomplished by using the framework's "patch" method. Two arguments are needed for the patch method:

What class or object needs to be patched

- The Object or class that has to be patched
- An object with the newly added method or properties

```
/** @odoo-module */
import { OrderWidget } from "@point_of_sale/app/generic_components/order_widge
import { patch } from "@web/core/utils/patch";
patch(OrderWidget.prototype, {
  get TotalQuantity(){
     var totalQuantity = 0;
     this.props.lines.forEach(line => totalQuantity += line.quantity);
     return totalQuantity
  }
});
```

▼ API

▼ SQL Questions

▼ SQL Query Order Phrase vs SQL Order Execution Phrase

Order Phrase

> Silly Ducks Fly, Jumping Walls, Going Home Over Lakes.

Execution Phrase

> Find Joy With Good Habits, Stay Determined On Learning.

▼ SQL Commands with Query

▼ DDL Commands(CREATE,ALTER,DROP,TRUNCATE)

CREATE

```
-- Create a new database
CREATE DATABASE company_db;

-- Create a new table
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    hire_date DATE,
    salary DECIMAL(10, 2),
    department_id INT
);

-- Create an index
CREATE INDEX idx_employee_name ON employees (last_name, first_name);
```

ALTER

```
-- Alter a table to add a new column
ALTER TABLE employees
ADD email VARCHAR(100);
```

```
-- Alter a table to modify a column
ALTER TABLE employees
MODIFY COLUMN salary DECIMAL(12, 2);

-- Alter a table to drop a column
ALTER TABLE employees
DROP COLUMN email;

-- Alter a table to add a foreign key
ALTER TABLE employees
ADD CONSTRAINT fk_department
FOREIGN KEY (department_id)
REFERENCES departments (department_id);
```

DROP

```
-- Drop a table
DROP TABLE employees;

-- Drop a database
DROP DATABASE company_db;

-- Drop an index
DROP INDEX idx_employee_name ON employees;
```

TRUNCATE

```
TRUNCATE TABLE employees;
```

▼ DML Commands(INSERT, UPDATE, DELETE)

INSERT

```
INSERT INTO EMPLOYEES(employee_id, first_name, last_name, email)
VALUES(2, 'Jane', 'Smith', 'jane.smith@example.com'),
        (3, 'Alice', 'Johnson', 'alice.johnson@example.com')
```

UPDATE

```
UPDATE employees
SET salary = 80000.00, department_id = 15
WHERE employee_id = 1
```

DELETE

```
DELETE FROM employees
WHERE employee_id = 3;
```

▼ DQL Commands(SELECT)

SELECT

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE department_id = 10;
```

▼ DCL Commands(grant,revoke)

GRANT

```
-- Grant SELECT and INSERT permissions on the employees table to a user name
GRANT SELECT, INSERT ON employees TO app_user;

-- Grant all privileges on the employees table to a role 'hr_role'
GRANT ALL PRIVILEGES ON employees TO hr_role;
```

REVOKE

```
REVOKE INSERT ON employees FROM app_user;

-- Revoke all privileges on the employees table from 'hr_role'
REVOKE ALL PRIVILEGES ON employees FROM hr_role;
```

▼ TCL Commands(grant,revoke)

COMMIT

It is used to permanently save all changes made during the **current transaction**. After executing a COMMIT statement, the changes are **irreversible**, and the database **cannot revert** to its **previous state**

```
COMMIT;
```

ROLLBACK

It is used to **undo changes** made during the **current transaction** that have not yet been committed. This command is particularly useful when errors occur, or the **transaction** is **aborted**. The **ROLLBACK** command ensures that the **database** returns to its **previous state** by undoing **uncommitted changes**.

```
ROLLBACK;
```

SAVEPOINT

The SAVEPOINT command sets a point within a transaction to which you can later roll back without affecting the entire transaction.

```
SAVEPOINT savepoint_name;
```

ROLLBACK TO SAVEPOINT

This query reverts the database to the state at savepoint_name, discarding changes made after the savepoint but preserving those before it.

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

▼ What is SQL?

SQL (Structured Query Language) is a standard programming language used to communicate with **relational databases**. It allows users to create, read, update, and delete data, and provides commands to define **database schema** and manage database security.

▼ What are the main types of SQL commands?

SQL commands are broadly classified into:

- **DDL (Data Definition Language):** to define and modify the structure of a database. e.g., CREATE, ALTER, DROP, TRUNCATE.
- **DML (Data Manipulation Language):** to access, manipulate, and modify data in a database.e.g., INSERT, UPDATE, DELETE.
- **DCL (Data Control Language):** to control user access to the data in the database and give or revoke privileges to a specific user or a group of users. e.g., GRANT, REVOKE.
- **TCL (Transaction Control Language):** to control transactions in a database. e.g., COMMIT, ROLLBACK, SAVEPOINT.
- **Data Query Language (DQL)**: to perform queries on the data in a database to retrieve the necessary information from it. e.g., SELECT

▼ What is a primary key ?

A **primary key** is a unique identifier for each record in a table. It ensures that no two rows have the same value in the primary key column(s), and it does not allow NULL values.

▼ What is a foreign key ?

A **foreign key** is a column (or set of columns) in one table that refers to the primary key in another table. It establishes and enforces a relationship between the two tables, ensuring data integrity.

▼ What is a unique key ?

A column (or multiple columns) of a table to which the `UNIQUE` constraint was imposed to ensure unique values in that column, including a possible `NULL` value (the only one).

▼ What are the different operators available in SQL ?

- **Arithmetic Operators:** +, -, *, /, %
- **Comparison Operators:** =, !=, <>, >, <, >=, <=
- **Logical Operators:** AND, OR, NOT
- **Set Operators:** UNION, INTERSECT, EXCEPT
- **Special Operators:** BETWEEN, IN, LIKE, IS NULL

▼ What is a constraint & types of constraint ?

A set of conditions defining the type of data that can be input into each column of a table. Constraints ensure data integrity in a table and block undesired actions.

- `DEFAULT` – provides a default value for a column.
- `UNIQUE` – allows only unique values.
- `NOT NULL` – allows only non-null values.
- `PRIMARY KEY` – allows only unique and strictly non-null values (`NOT NULL` and `UNIQUE`).
- `FOREIGN KEY` – provides shared keys between two or more tables.
- ChECK - Ensures that all values in a column satisfy a specific condition.

▼ What is the purpose of the DEFAULT constraint ?

The **DEFAULT constraint** assigns a default value to a column when no value is provided during an **INSERT operation**. This helps maintain consistent data and simplifies data entry.

▼ What is the purpose of the UNIQUE constraint ?

The **UNIQUE constraint** ensures that all values in a column (or combination of columns) are **distinct**. This prevents duplicate values and helps maintain data integrity.

▼ What is a join & types of join ?

A JOIN is ***used to combine rows from two or more tables***, based on a related column between them.

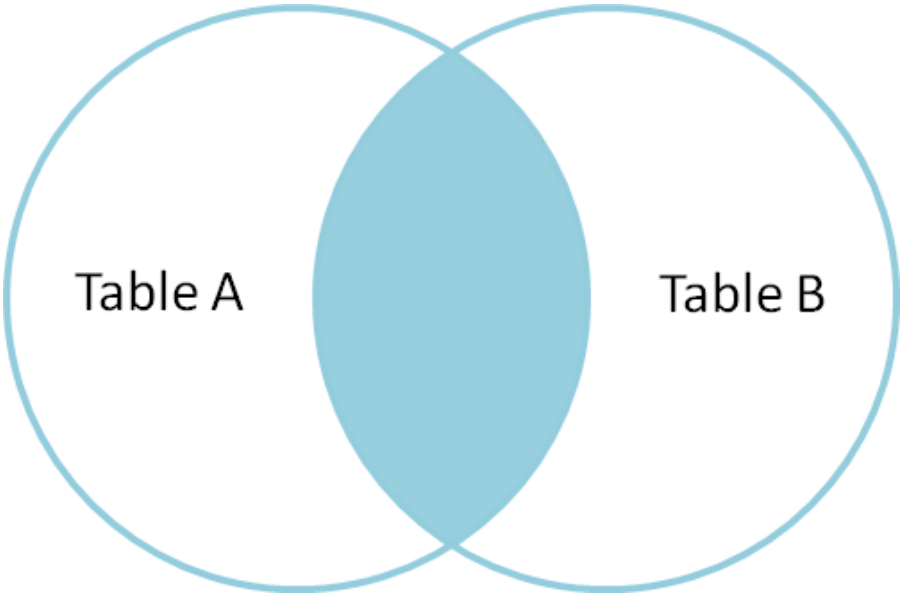
General Syntax:-

```
SELECT table1.col, table2.col from table1
left join table2 // any join
on table1.col = table2.col;
```

`(INNER) JOIN` – returns only those records that satisfy a defined join condition in both (or all) tables. It's a default SQL join.

Syntax:-

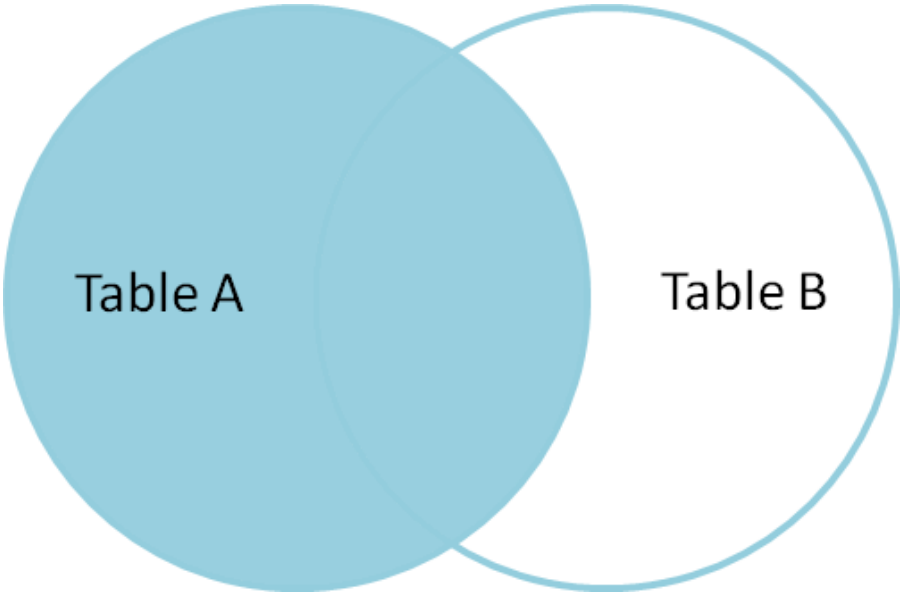
```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```



`LEFT (OUTER) JOIN` – returns all records from the left table and those records from the right table that satisfy a defined join condition.

Syntax:-

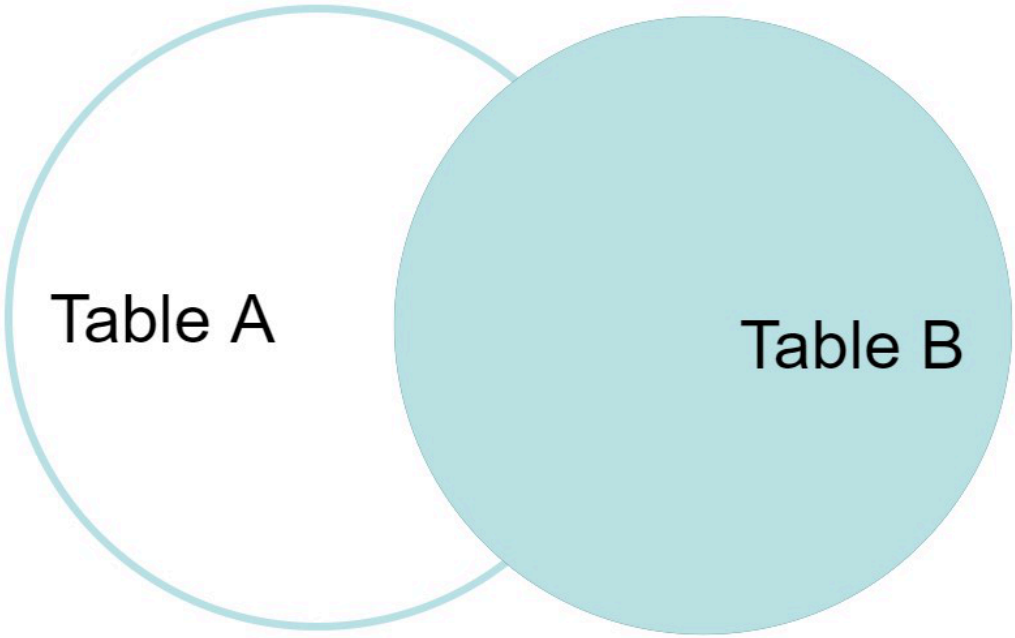
```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```



`RIGHT (OUTER) JOIN` – returns all records from the right table and those records from the left table that satisfy a defined join condition.

Syntax:-

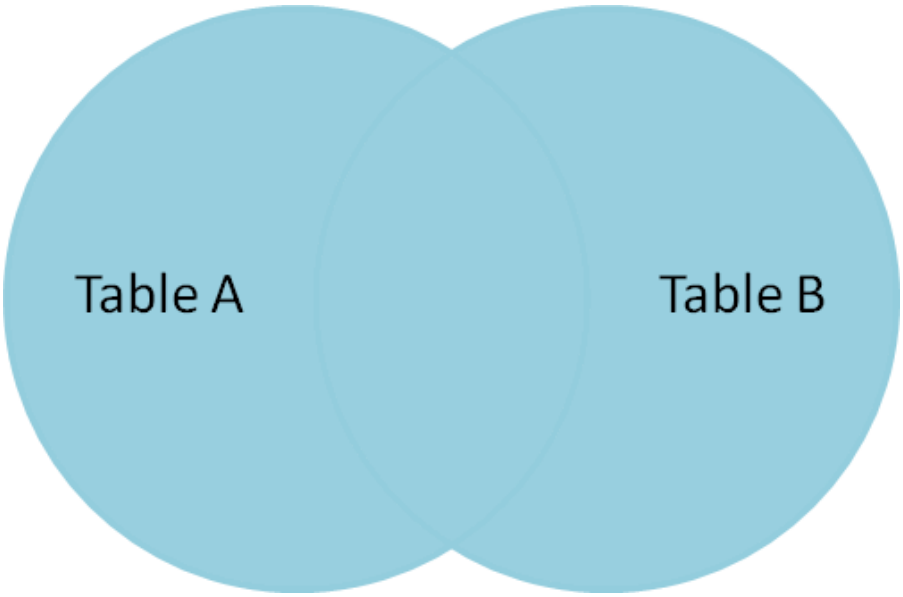
```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```



`FULL (OUTER) JOIN` – returns all records from both (or all) tables. It can be considered as a combination of left and right joins.

Syntax:-

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```



▼ What is a clause ?

A condition imposed on a SQL query to filter the data to obtain the desired result. Some examples are `WHERE` , `LIMIT` , `HAVING` , `LIKE` , `AND` , `OR` , `ORDER BY` , etc.

▼ What is the purpose of the GROUP BY clause ?

The **GROUP BY** clause is used to arrange **identical data** into **groups**. It is typically used with aggregate functions (such as COUNT, SUM, AVG) to perform calculations on each group rather than on the entire dataset.

▼ What is the purpose of the SQL ORDER BY clause ?

The **ORDER BY** clause sorts the result set of a query in either **ascending** (default) or **descending order**, based on one or more columns. This helps present the data in a more meaningful or readable sequence.

▼ What is the difference between the WHERE and HAVING clauses ?

- **WHERE:** Filters rows before any grouping takes place.
- **HAVING:** Filters grouped data after the GROUP BY clause has been applied.In short, WHERE applies to individual rows, while HAVING applies to groups.

▼ What are aggregate functions in SQL ?

Aggregate functions perform calculations on a set of values and return a single value. Common aggregate functions include:

- COUNT(): Returns the number of rows.
- SUM(): Returns the total sum of values.
- AVG(): Returns the average of values.

- MIN(): Returns the smallest value.
- MAX(): Returns the largest value.

▼ What is a subquery?

A **subquery** is a query nested within another query. It is often used in the **WHERE clause** to filter data based on the results of another query, making it easier to handle complex conditions.

▼ What is a view in SQL ?

A **view** is a **virtual table** created by a **SELECT query**. It does not store data itself, but presents data from one or more tables in a structured way. Views simplify complex queries, improve readability, and enhance security by restricting access to specific rows or columns.

▼ What is a query in SQL?

A query is a SQL statement used to retrieve, update, or manipulate data in a **database**. The most common type of query is a **SELECT statement**, which fetches data from one or more tables based on specified conditions.

▼ What is a table in SQL ?

A table is a **structured collection** of related data organized into rows and columns. Columns define the type of data stored, while rows contain individual records.

▼ What is a cursor in SQL ?

A **cursor** is a database object used to **retrieve**, **manipulate**, and traverse through rows in a result set one row at a time. Cursors are helpful when performing operations that must be processed sequentially rather than in a set-based manner.

▼ What are indexes, and why are they used and types of index ?

Indexes are **database objects** that improve query performance by allowing **faster retrieval of rows**. They function like a book's index, making it quicker to find specific data without scanning the entire table.

- **Unique index** – doesn't allow duplicates in a table column and hence helps maintain data integrity.
- **Clustered index** – defines the physical order of records of a database table and performs data searching based on the key values. A table can have only one clustered index.

- **Non-clustered index** – keeps the order of the table records that don't match the physical order of the actual data on the disk. It means that the data is stored in one place and a non-clustered index – in another one. A table can have multiple non-clustered indexes.

▼ What is a transaction in SQL?

A transaction in SQL is a sequence of one or more SQL operations treated as a single unit of work. Transactions ensure that database operations are either completed successfully or rolled back entirely in case of failure.

▼ What is an alias ?

A temporary name given to a table (or a column in a table) while executing a certain SQL query. Aliases are used to improve the code readability and make the code more compact. An alias is introduced with the `AS` keyword:

```
SELECT col_1 AS column
FROM table_name;
```

▼ What is the purpose of the SQL SELECT statement ?

The **SELECT** statement retrieves data from one or more tables. It is the most commonly used command in SQL, allowing users to filter, sort, and display data based on specific criteria.

▼ What are NULL values in SQL ?

NULL represents a missing or unknown value. It is different from zero or an empty string. NULL values indicate that the data is not available or applicable.

▼ What is the purpose of the ALTER command in SQL ?

The `ALTER` command is used to **modify the structure** of an existing database object. This command is essential for adapting our **database schema** as requirements evolve.

- Add or drop a column in a table.
- Change a column's data type.
- Add or remove constraints.
- Rename columns or tables.
- Adjust indexing or storage settings.

▼ What is a composite primary key ?

A c**omposite primary key** is a primary key made up of two or more columns. Together, these columns must form a unique combination for each row in the table. It's used when a single column isn't sufficient to uniquely identify a record.

▼ What is a UNION operation, and how is it used ?

The `UNION` operator combines the result sets of two or more `SELECT` **queries** into a single result set, removing **duplicate rows**. The result sets must have the same number of columns and compatible data types for corresponding columns.

```
SELECT Name FROM Customers
UNION
SELECT Name FROM Employees;
```

▼ What is the difference between UNION and UNION ALL ?

- **UNION:** Combines result sets from two queries and removes **duplicate rows**, ensuring only unique records are returned.
- **UNION ALL:** Combines the result sets without removing duplicates, meaning all records from both queries are included.

▼ What is the purpose of the COALESCE function?

The `COALESCE` **function** returns the first non-NULL value from a list of expressions. It's commonly used to provide default values or handle missing data gracefully.

```
SELECT COALESCE(NULL, NULL, 'Default Value') AS Result;
```

▼ DELETE v/s TRUNCATE

- **DELETE:** Removes rows one at a time and records each deletion in the transaction log, allowing rollback. It can have a WHERE clause.
- **TRUNCATE:** Removes all rows at once without logging individual row deletions. It cannot have a WHERE clause and is faster than DELETE for large data sets.

▼ DROP v/s TRUNCATE

DROP: It is used to completely remove a database object (e.g., a table, schema, or database) from the database, including its structure, data, indexes, constraints, and permissions.

TRUNCATE: It removes all rows from a table but retains the table's structure, indexes, constraints, and permissions.

▼ Aggregate functions vs Scalar functions

Aggregate functions in SQL perform calculations on a set of values and return a single result.

- SUM: To calculate the sum of values in a column.
- COUNT: To count a column's number of rows or non-null values.
- AVG: To calculate the average of values in a column.
- MIN: To retrieve the minimum value in a column.
- MAX: To retrieve the maximum value in a column

Scalar functions

Scalar Functions are built-in functions that operate on a single value and return a single value.

- **LEN()** - Calculates the total length of the given field (column).
- **UCASE()** - Converts a collection of string values to uppercase characters.
- **LCASE()** - Converts a collection of string values to lowercase characters.
- **MID()** - Extracts substrings from a collection of string values in a table.
- **CONCAT()** - Concatenates two or more strings.
- **RAND()** - Generates a random collection of numbers of a given length.
- **ROUND()** - Calculates the round-off integer value for a numeric field (or decimal point values).
- **NOW()** - Returns the current date & time.
- **FORMAT()** - Sets the format to display a collection of values.

▼ Explain ACID properties in SQL.

ACID - Atomicity, Consistency, Isolation, and Durability. They are essential properties that ensure the reliability and integrity of database transactions:

- Atomicity (single, indivisible unit of transactions)
- Consistency (transactions bring the DB from one consistent state to another)
- Isolation (transactions are isolated from each other)
- Durability (committed transactions are permanent and survive system failures.)

▼ Third Party Integration

▼ Stripe

1. **Create or Log in to Your Stripe Account**

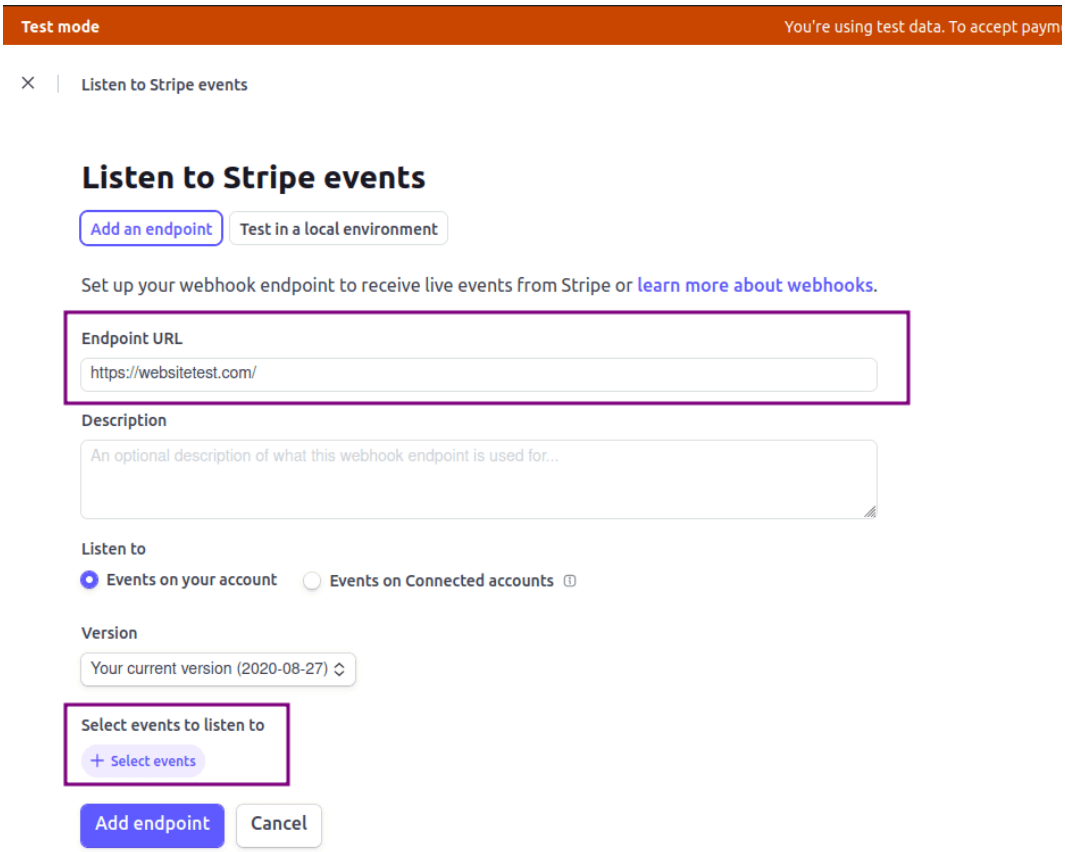
- If you don’t have a Stripe account, register at dashboard.stripe.com/register.
- Log in to your Stripe dashboard to access API keys and webhook settings.

2. **Configure API Credentials (Manual Setup)**

- From Stripe dashboard, go to **Developers > API Keys > Standard Keys**, copy the **Publishable key**.
- **Secret Key**: Copy the **Secret key** from the same section.
- Save the credentials.
- In Odoo, activate **Developer Mode** (Settings > Activate the Developer Mode).
- Go to **Invoicing > Configuration > Payment Providers > Stripe.**
- In the **Credentials** tab, enter:
  • Publish Key & Secret key
- Set State Enabled.

3. **Set Up Webhooks**

- Webhooks ensure Odoo receives payment status updates from Stripe.
- **Automatic Webhook Creation**:
  - In Odoo’s Stripe configuration, click **Generate your Webhook**. This automatically sets up the webhook in Stripe.
- **Manual Webhook Creation**:
  - In Stripe, go to **Developers > Webhooks > Add endpoint**.



- Enter your Odoo database URL followed by /payment/stripe/webhook (e.g., https://your-odoo-domain.com/payment/stripe/webhook).
- Select the event **checkout.session.completed** under the **Checkout** section. Other events are not processed by Odoo.
- Select the following events to listen to:
- **Charge**: charge.refunded, charge.refund.updated
- **Payment Intent**: payment_intent.amount_capturable_updated, payment_intent.payment_failed, payment_intent.processing, payment_intent.succeeded
- **Setup Intent**: setup_intent.succeeded
- Click **Add endpoint**, then click **reveal** to copy the **Webhook Signing Secret**.
- In Odoo, paste the Webhook Signing Secret into the **Credentials** tab's **Webhook Signing Secret** field.

        - Save the configuration

4. **Configure Payment Methods and Settings**

- In Odoo's Stripe configuration, go to the **Configuration** tab.
- Add or edit **Supported Payment Icons** to enable local payment methods (e.g., Apple Pay, Google Pay) supported by Stripe. If a method is listed in Odoo's supported methods, it's automatically enabled unless removed.
- Set the **Payment Flow**:
    - **Payment From Odoo**: Customers complete payments on your Odoo website without redirection.
    - **Redirection to Acquirer Website**: Redirects customers to Stripe's payment page.
- Select a **Payment Journal** (e.g., Bank or a dedicated Stripe journal) for accounting purposes.
- Optionally, restrict availability by selecting specific **Countries** in the **Availability** section.

5. **Test the Integration**

- Test in a safe environment first:
    - In Stripe, switch to **Test Mode** in the dashboard.
    - In Odoo, activate **Developer Mode**, navigate to Stripe configuration, set **State** to **Test Mode**, and use test API keys (available in Stripe's **Developers > API Keys** under Test Mode).
    - Recommended: Use a test Odoo database to avoid affecting live data.
- Create a test transaction (e.g., place an order on your Odoo website or create an invoice in the Invoicing app).
- Verify that the Stripe payment option appears at checkout and processes correctly. Use Stripe's test card numbers (e.g., 4242 4242 4242 4242) for testing.
- Verify that the transaction appears in Odoo under Invoicing > Payments > Payment Transactions (visible in Debug mode)

▼ Modules made by me

Sale Return

Allow user to return products from sale order with stock evaluation along with their delivery returns.

Export Stock Information

Whether you need to analyze stock levels, track product movements, or review inventory valuations, this app allows you to easily produce detailed reports in both PDF and Excel formats.

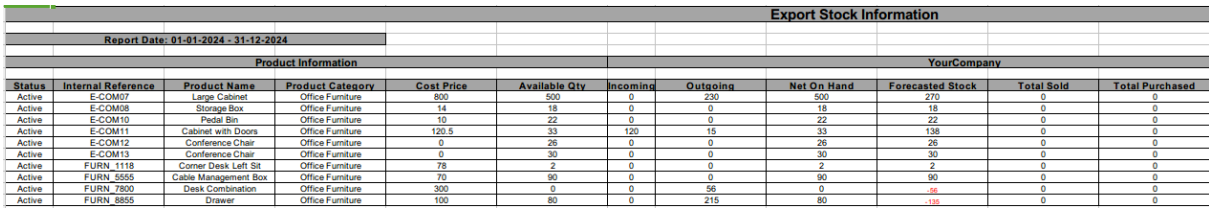
PDF

Export Stock Information

Report Date	Warehouse
01-01-2024 - 31-12-2024	YourCompany

Internal Ref.	Product Name	Cost Price	Available Qty	Incoming Qty	Outgoing Qty	Net on Hand	Forecasted Qty	Total Sold Qty	Total Purchase Qty	Valuation
E-COM07	Large Cabinet	800	500	0	230	500	270	0	0	400000
E-COM08	Storage Box	14	18	0	0	18	18	0	0	252
E-COM10	Pedal Bin	10	22	0	0	22	22	0	0	220
E-COM11	Cabinet with Doors	120.5	33	120	15	33	138	0	0	3976.5
E-COM12	Conference Chair	0	26	0	0	26	26	0	0	0
E-COM13	Conference Chair	0	30	0	0	30	30	0	0	0
FURN_1118	Corner Desk Left Sit	78	2	0	0	2	2	0	0	156
FURN_5555	Cable Management Bux	70	90	0	0	90	90	0	0	6300
FURN_7800	Desk Combination	300	0	0	56	0	-56	0	0	0

Excel



Export Stock Information

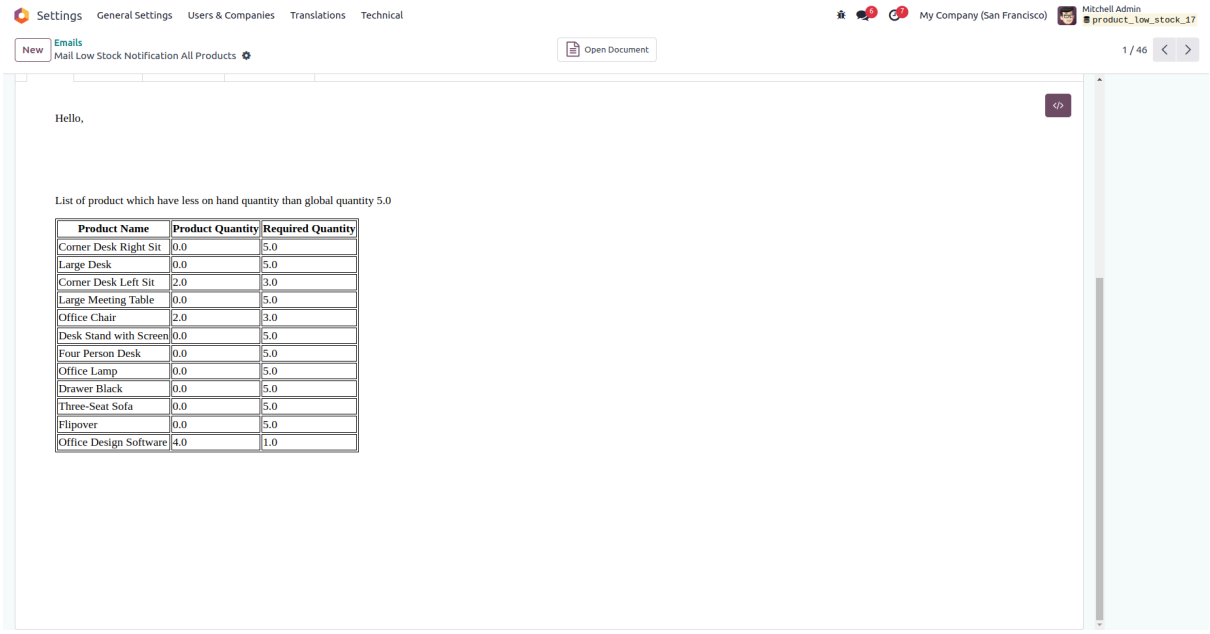
Report Date: 01-01-2024 - 31-12-2024

Product Information						YourCompany					
Status	Internal Reference	Product Name	Product Category	Cost Price	Available Qty	Incoming	Outgoing	Net On Hand	Forecasted Stock	Total Sold	Total Purchased
Active	E-COM07	Large Cabinet	Office Furniture	800	500	0	230	500	270	0	0
Active	E-COM08	Storage Box	Office Furniture	14	18	0	0	18	18	0	0
Active	E-COM10	Pedal Bin	Office Furniture	10	22	0	0	22	22	0	0
Active	E-COM11	Cabinet with Doors	Office Furniture	120.5	33	120	15	33	138	0	0
Active	E-COM12	Conference Chair	Office Furniture	0	26	0	0	26	26	0	0
Active	E-COM13	Conference Chair	Office Furniture	0	30	0	0	30	30	0	0
Active	FURN_1118	Corner Desk Left Sit	Office Furniture	78	2	0	0	2	2	0	0
Active	FURN_5555	Cable Management Box	Office Furniture	70	90	0	0	90	90	0	0
Active	FURN_7800	Desk Combination	Office Furniture	300	0	0	56	0	-56	0	0
Active	FURN_8855	Drawer	Office Furniture	100	80	0	215	80	-135	0	0

Product Low Stock Notification

Configured user will get low stock product notification based on on hand or forecast quantity of product by email with attachment. User can get notification as global for all

the products, individual for every product or based on reordering rules. Additionally, user can also set minimum stock quantity for product and product variant.



Your logo

My Company (San Francisco)
250 Executive Park Blvd, Suite 3400
San Francisco 94134
California CA
India

Product Low Stock Report

Product Name	Minimum Quantity	Product Quantity	Required Quantity
Corner Desk Right Sit	5.0	0.0	5.0
Large Desk	5.0	0.0	5.0
Corner Desk Left Sit	5.0	2.0	3.0
Large Meeting Table	5.0	0.0	5.0
Office Chair	5.0	2.0	3.0
Desk Stand with Screen	5.0	0.0	5.0
Four Person Desk	5.0	0.0	5.0
Office Lamp	5.0	0.0	5.0
Drawer Black	5.0	0.0	5.0
Three-Seat Sofa	5.0	0.0	5.0
Flipover	5.0	0.0	5.0
Office Design Software	5.0	4.0	1.0

▼ Bizzappdev

▼ Sale Order to Delivery Order

For Inherit the sale order and add the field:
from odoo import models, fields

```
class SaleOrder(models.Model):
    _inherit = 'sale.order'

    description = fields.Char(string='Description')
```

For Inherit the stock move object and add the field:

```
from odoo import models, fields

class StockMove(models.Model):
    _inherit = 'stock.move'

    description = fields.Char(string='Description')

    def _get_new_picking_values(self):
        """Inherit method for pass value from sale order to delivery order."""
        res = super(StockMove, self)._get_new_picking_values()
        res["description"] = self.group_id.sale_id.description
        return res
```

These are the root methods for finding the original method for passing value from the sale order to the delivery order:

step1: def action_confirm(self): [module:sale, object:sale.order]
step2: def _action_confirm(self, merge=True, merge_into=False): [module:stock, object:stock.move]
step3: def _assign_picking(self): [module:stock, object:stock.move]
step4: def _get_new_picking_values(self): [module:stock, object:stock.move]

▼ Sale to Invoice

For Inherit the sale order and add the field:

```
from odoo import models, fields

class SaleOrder(models.Model):
    _inherit = 'sale.order'

        invoice_description = fields.Char(string='Invoice Description')
```

For Inherit the account move and add the field:

```
from odoo import models, fields

class AccountMove(models.Model):
   _inherit = 'account.move'

       invoice_description = fields.Char(string='Invoice Description')
```

Now passing the value from the sale order to the invoice, Here there are two meth
ods for passing value.
one is for the regular invoice and another is for downpayment invoices.

1. Method for passing value from sale order to invoice(regular).

```
from odoo import models, fields

class SaleOrder(models.Model):
    _inherit = 'sale.order'

    invoice_description = fields.Char(string='Invoice Description')


def _prepare_invoice(self):
       """ Method for passing value from sale order to regular invoice. """
        res = super(SaleOrder, self)._prepare_invoice()
        res["invoice_description"] = self.invoice_description
        return res
```

These are the root methods for finding the original method for passing value from
sale order to invoice(Regular):

step1: def create_invoices(self): [module:sale,object:sale.advance.payment.inv]
step2: def _create_invoices(self, grouped=False, final=False, date=None): [modul
e:sale, object:sale.order]
step3: def _prepare_invoice(self): [module:sale, object:sale.order]

2. Method for passing value from sale order to invoice(downpayment invoices).

```
from odoo import models, fields

class SaleAdvancePaymentInv(models.TransientModel):
    _inherit = 'sale.advance.payment.inv'
```

```
def _prepare_invoice_values(self, order, name, amount, so_line):
        """ Method for passing value from sale order to regular invoice. """
        res = super(SaleAdvancePaymentInv, self)._prepare_invoice_values(
            order, name, amount, so_line
        )
        res.update({"invoice_description": order.invoice_description})
        return res
```

These are the root methods for finding the original method for passing value from sale order to invoice(Downpayment):

step1: def create_invoices(self): [module:sale,object:sale.advance.payment.inv]
step2: def _create_invoices(self, grouped=False, final=False, date=None): [module:sale,object:sale.advance.payment.inv]
step3: def _prepare_invoice_values(self, order, name, amount, so_line): [module:sale,object:sale.advance.payment.inv]

▼ Sale order to project and tasks

For Inherit the sale order and add the field:

```
from odoo import models, fields

class SaleOrder(models.Model):
    _inherit = 'sale.order'

        project_description = fields.Char(string='Project Description')
    task_description = fields.Char(string='Task Description')
```

Now, added fields at the project level:

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

```
class Project(models.Model):
    _inherit = 'project.project'

        project_description = fields.Char(string='Project Description')
```

Now, added fields at the task level:

```
from odoo import models, fields

class ProjectTask(models.Model):
    _inherit = 'project.task'

    task_description = fields.Char(string='Task Description')
```

1.Method for passing value from sale order to project.

```
from odoo import models, fields

class SaleOrderLine(models.Model):
    _inherit = 'sale.order.line'

        def _timesheet_create_project_prepare_values(self):
        """ Inherit method for passing value from sale order to project. """
         vals = super(SaleOrderLine, self)._timesheet_create_project_prepare_values
()
         vals["project_description"] = self.order_id.project_description
         return vals
```

These are the root methods for finding the original method for passing value fro m the sale order to the project:

step1: def action_confirm(self): [module:sale, object:sale.order]
step2: def _action_confirm(self): [module:sale_project, object:sale.order]
step3: def _timesheet_service_generation(self): [module:sale_project, object:sale.

order.line]
step4: def _timesheet_create_project(self): [module:sale_project, object:sale.orde
r.line]
step5: def _timesheet_create_project_prepare_values(self): [module:sale_project,
object:sale.order.line]

2.Method for passing value from sale order to task.

```
from odoo import models, fields

class SaleOrderLine(models.Model):
    _inherit = 'sale.order.line'

        def _timesheet_create_task_prepare_values(self, project):
        """ Inherit method for passing value from sale order to task. """
         vals = super(SaleOrderLine, self)._timesheet_create_task_prepare_values(pr
oject)
         vals["project_description"] = self.order_id.project_description
         return vals
```

These are the root methods for finding the original method for passing value from the sale order to the task:

step1: def action_confirm(self): [module:sale, object:sale.order]
step2: def _action_confirm(self): [module:sale_project, object:sale.order]
step3: def _timesheet_service_generation(self): [module:sale_project,object:sale.
order.line]
step4: def _timesheet_create_task(self, project):: [module:sale_project,object:sal
e.order.line]
step5: def _timesheet_create_task_prepare_values(self, project): [module:sale_pro
ject,object:sale.order.line]

▼ Sale order to manufacturing order

For Inherit the sale order and add the field:

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

```
class SaleOrder(models.Model):
    _inherit = 'sale.order'

        mrp_description = fields.Char(string='Manufacturing Description')
```

Now Inherit mrp production and add the field:

```
from odoo import models, fields

class SaleOrder(models.Model):
    _inherit = 'mrp.production'

        mrp_description = fields.Char(string='Manufacturing Description')
```

For updating or passing the value in mrp, we need to inherit the two methods for t he stock move and stock rule.

Inherit the method of stock move object:

```
from odoo import models, fields

class StockMove(models.Model):
    _inherit = 'stock.move'

def _prepare_procurement_values(self):
        val = super(StockMove, self)._prepare_procurement_values()
        val["mrp_description"] = self.group_id.sale_id.mrp_description
        return val
```

Inherit the method of stock rule object:

```
from odoo import models, fields

class StockRule(models.Model):
    _inherit = 'stock.rule'

def _prepare_mo_vals(
        self,
        product_id,
        product_qty,
```

```
        product_uom,
        location_id,
        name,
        origin,
        company_id,
        values,
        bom,
    ):
        """ Method for passing value from sale order to manufacturing order. """

        vals = super(StockRule, self)._prepare_mo_vals(
            product_id,
            product_qty,
            product_uom,
            location_id,
            name,
            origin,
            company_id,
            values,
            bom,
        )
        vals["mrp_description"] = values.get("mrp_description")
        return vals
```

These are the root methods for finding the original method for passing value from the sale order to the manufacturing order:

step1: def action_confirm(self): [module:sale, object:sale.order]
step2: def _action_confirm(self): [module:sale_project, object:sale.stock]
step3: def _action_launch_stock_rule(self, previous_product_uom_qty=False):
step4: def _run_manufacture(self, procurements): [module:mrp, object:sale.rule]
step5: def _prepare_mo_vals(self, product_id, product_qty, product_uom, location_id, name, origin, company_id, values, bom): [module:mrp, object:sale.rule]

▼ Sale order to purchase order

```
from odoo import models, fields

class SaleOrder(models.Model):
    _inherit = 'sale.order'

    purchase_description = fields.Char(string='Purchase Description')
```

Now Inherit purchase order and add the field:

```
from odoo import models, fields

class SaleOrder(models.Model):
    _inherit = 'purchase.order'

    purchase_description = fields.Char(string='Purchase Description')
```

To update or pass the value in the purchase order, we need to inherit the two metho

Inherit the method of stock move object:

```
from odoo import models, fields

class StockMove(models.Model):
   _inherit = 'stock.move'

def _prepare_procurement_values(self):
    val = super(StockMove, self)._prepare_procurement_values()
    val["sale"] = self.group_id.sale_id
return val
```

Inherit the method of stock rule object:

```
from odoo import models, fields

class StockRule(models.Model):
_inherit = 'stock.rule'

def _prepare_purchase_order(self, company_id, origins, values):
```

```
""" Method for passing value from sale order to purchase order. """

res = super()._prepare_purchase_order(company_id=company_id, origins=origins, va
res['purchase_description'] = values[0].get("sale").purchase_description
return res

These are the root methods for finding the original method for passing value from th

step1: def action_confirm(self): [module:sale, object:sale.order]
step2: def _action_confirm(self): [module:sale_project, object:sale.stock]
step3: def _action_launch_stock_rule(self, previous_product_uom_qty=False):
step4: def _run_buy(self, procurements): [module:purchase_stock, object:sale.rule]
step5: def _prepare_purchase_order(self, company_id, origins, values): [module:purc
```