Module structure

Directories

A module is organised in a few directory :

- *data/* : demo and data xml
- *models/* : models definition
- *controllers/* : contains controllers (http routes).
- *views/* : contains the views and templates
- *static/* : contains the web assets, separated into *css/, js/, img/, lib/, ...*

File naming

For views declarations, split backend views from (frontend) templates in 2 differents files.

For models, split the business logic by sets of models, in each set select a main model, this model gives its name to the set. If there is only one model, its name is the same as the module name. For each set named <main_model> the following files may be created:

- **models/<*main_model*>.py**
- **models/<*inherited_main_model*>.py**
- **views/<*main_model*>_templates.xml**
- **views/<*main_model*>_views.xml**

For instance, *sale* module introduces **sale_order** and **sale_order_line** where **sale_order** is dominant. So the **<main_model>**files will be named **models/sale_order.py** and **views/sale_order_views.py**.

For *data*, split them by purpose : demo or data. The filename will be the main_model name, suffixed by *_demo.xml* or *_data.xml*.

For *controller*, the only file should be named *main.py*.

For *static files*, the name pattern is <*module_name*>*.ext* (i.e. : static/js/im_chat.js, static/css/im_chat.css, static/xml/im_chat.xml, ...). Don't link data (image, libraries) outside Odoo : don't use an url to an image but copy it in our codebase instead.

The complete tree should look like

```
addons/<my_module_name>/
```

```
|-- __init__.py
|-- __openerp__.py
|-- controllers/
|   |-- __init__.py
|   `-- main.py
|-- data/
|   |-- <main_model>_data.xml
|   `-- <inherited_main_model>_demo.xml
|-- models/
|   |-- __init__.py
|   |-- <main_model>.py
|   `-- <inherited_main_model>.py
|-- security/
|   |-- ir.model.access.csv
|   `-- <main_model>_security.xml
|-- static/
|   |-- img/
|   |   |-- my_little_kitten.png
|   |   `-- troll.jpg
|   |-- lib/
|   |   `-- external_lib/
|   `-- src/
|       |-- js/
|       |   `-- <my_module_name>.js
|       |-- css/
|       |   `-- <my_module_name>.css
|       |-- less/
|       |   `-- <my_module_name>.less
|       `-- xml/
|           `-- <my_module_name>.xml
`-- views/
    |-- <main_model>_templates.xml
    |-- <main_model>_views.xml
    |-- <inherited_main_model>_templates.xml
    `-- <inherited_main_model>_views.xml
```

Note

File names should only contain **[a-z0-9_]** (lowercase alphanumerics and _)

Warning

Use correct file permissions : folder 755 and file 644.

XML files

Format

When declaring a record in XML,

- Place **id** attribute before **model**
- For field declaration, **name** attribute is first. Then place the *value* either in the **field** tag, either in the **eval** attribute, and finally other attributes (widget, options, ...) ordered by importance.
- Try to group the record by model. In case of dependencies between action/menu/views, the convention may not be applicable.
- Use naming convention defined at the next point
- The tag <*data*> is only used to set not-updatable data with **noupdate=1**

```
<record id="view_id" model="ir.ui.view">
   <field name="name">view.name</field>
   <field name="model">object_name</field>
   <field name="priority" eval="16"/>
   <field name="arch" type="xml">
      <tree>
         <field name="my_field_1"/>
         <field name="my_field_2" string="My Label" widget="statusbar" statusbar_vis
ible="draft,sent,progress,done" statusbar_colors='{"invoice_except":"red","waiting_
date":"blue"}' />
      </tree>
   </field>
</record>
```

Naming xml_id

Security, View and Action

Use the following pattern :

- For a menu: **<*model_name*>_menu**
- For a view: **<*model_name*>_view_<*view_type*>**,
  where *view_type* is **kanban**, **form**, **tree**, **search**, ...

- For an action: the main action respects **<*model_name*>_action**. Others are suffixed with **_<*detail*>**, where *detail* is a lowercase string briefly explaining the action. This is used only if multiple actions are declared for the model.
- For a group: **<*model_name*>_group_<*group_name*>** where *group_name* is the name of the group, generally 'user', 'manager', ...
- For a rule: **<*model_name*>_rule_<*concerned_group*>** where *concerned_group* is the short name of the concerned group ('user' for the 'model_name_group_user', 'public' for public user, 'company' for multi-company rules, ...).

```
<!-- views and menus -->
<record id="model_name_menu" model="ir.ui.menu">
    ...
</record>

<record id="model_name_view_form" model="ir.ui.view">
    ...
</record>

<record id="model_name_view_kanban" model="ir.ui.view">
    ...
</record>

<!-- actions -->
<record id="model_name_action" model="ir.actions.act_window">
    ...
</record>

<record id="model_name_action_child_list" model="ir.actions.act_window">
    ...
</record>

<!-- security -->
<record id="model_name_group_user" model="res.groups">
    ...
</record>

<record id="model_name_rule_public" model="ir.rule">
    ...
</record>

<record id="model_name_rule_company" model="ir.rule">
    ...
</record>
```

Note

View names use dot notation **my.model.view_type** or **my.model.view_type.inherit** instead of *"This is the form view of My Model"*.

Inherited XML

The naming pattern of inherited view is **<*base_view*>_inherit_<*current_module_name*>**. A module may only extend a view once. Suffix the orginal name with **_inherit_<*current_module_name*>** where *current_module_name* is the technical name of the module extending the view.

```
<record id="inherited_model_view_form_inherit_my_module" model="ir.ui.view">
    ...
</record>
```

Python

PEP8 options

Using a linter can help show syntax and semantic warnings or errors. Odoo source code tries to respect Python standard, but some of them can be ignored.

- E501: line too long
- E301: expected 1 blank line, found 0
- E302: expected 2 blank lines, found 1
- E126: continuation line over-indented for hanging indent
- E123: closing bracket does not match indentation of opening bracket's line
- E127: continuation line over-indented for visual indent
- E128: continuation line under-indented for visual indent
- E265: block comment should start with '# '

Imports

The imports are ordered as

1. External libraries (one per line sorted and split in python stdlib)
2. Imports of **openerp**
3. Imports from Odoo modules (rarely, and only if necessary)

Inside these 3 groups, the imported lines are alphabetically sorted.

```
# 1 : imports of python lib
import base64
import re
import time
# 2 :  imports of openerp
import openerp
from openerp import api, fields, models # alphabetically ordered
from openerp.tools.safe_eval import safe_eval as eval
from openerp.tools.translate import _
# 3 :  imports from odoo modules
from openerp.addons.website.models.website import slug
from openerp.addons.web.controllers.main import login_redirect
```

Idioms

- Prefer **%** over **.format()**, prefer **%(varname)** instead of position (This is better for translation)
- Try to avoid generators and decorators
- Always favor *Readability* over *conciseness* or using the language features or idioms.
- Use list comprehension, dict comprehension, and basic manipulation using **map**, **filter**, **sum**, ... They make the code easier to read.
- The same applies for recordset methods : use **filtered**, **mapped**, **sorted**, ...
- Each python file should have **# -*- coding: utf-8 -*-** as first line
- Use the **UserError** defined in **openerp.exceptions** instead of overriding **Warning**, or find a more appropriate exception in*exceptions.py*
- Document your code (docstring on methods, simple comments for the tricky part of the code)

- Use meaningful variable/class/method names

Symbols

- Odoo Python Class : use camelcase for code in api v8, underscore lowercase notation for old api.

```
class AccountInvoice(models.Model):
    ...

class account_invoice(osv.osv):
    ...
```

- **Variable name :**
    - use camelcase for model variable
    - use underscore lowercase notation for common variable.
    - since new API works with record or recordset instead of id list, don't suffix variable name with *_id* or *_ids* if they not contain id or list of id.

```
ResPartner = self.env['res.partner']
partners = ResPartner.browse(ids)
partner_id = partners[0].id
```

- **One2Many** and **Many2Many** fields should always have *_ids* as suffix (example: sale_order_line_ids)
- **Many2One** fields should have *_id* as suffix (example : partner_id, user_id, ...)
- **Method conventions**
    - Compute Field : the compute method pattern is *_compute_<field_name>*
    - Search method : the search method pattern is *_search_<field_name>*
    - Default method : the default method pattern is *_default_<field_name>*
    - Onchange method : the onchange method pattern is *_onchange_<field_name>*
    - Constraint method : the constraint method pattern is *_check_<constraint_name>*
    - Action method : an object action method is prefix with *action_*. Its decorator is **@api.multi**, but since it use only one record, add **self.ensure_one()** at the beginning of the method.
- **In a Model attribute order should be**
    1. Private attributes (**_name**, **_description**, **_inherit**, ...)
    2. Default method and **_default_get**
    3. Field declarations

4. Compute and search methods in the same order as field declaration
5. Constrains methods (`@api.constrains`) and onchange methods (`@api.onchange`)
6. CRUD methods (ORM overrides)
7. Action methods
8. And finally, other business methods.

```
class Event(models.Model):
    # Private attributes
    _name = 'event.event'
    _description = 'Event'

    # Default methods
    def _default_name(self):
        ...

    # Fields declaration
    name = fields.Char(string='Name', default=_default_name)
    seats_reserved = fields.Integer(oldname='register_current', string='Reserved Seats
',
        store=True, readonly=True, compute='_compute_seats')
    seats_available = fields.Integer(oldname='register_avail', string='Available Seats',
        store=True, readonly=True, compute='_compute_seats')
    price = fields.Integer(string='Price')

    # compute and search fields, in the same order that fields declaration
    @api.multi
    @api.depends('seats_max', 'registration_ids.state', 'registration_ids.nb_register')
    def _compute_seats(self):
        ...

    # Constraints and onchanges
    @api.constrains('seats_max', 'seats_available')
    def _check_seats_limit(self):
        ...

    @api.onchange('date_begin')
    def _onchange_date_begin(self):
        ...

    # CRUD methods
    def create(self):
        ...
```

```
    # Action methods
    @api.multi
    def action_validate(self):
        self.ensure_one()
        ...

    # Business methods
    def mail_user_confirm(self):
        ...
```

Javascript and CSS

For javascript :

- `use strict;` is recommended for all javascript files
- Use a linter (jshint, ...)
- Never add minified Javascript Libraries
- Use camelcase for class declaration

For CSS :

- Prefix all your class with *o_<module_name>* where *module_name* is the technical name of the module ('sale', 'im_chat', ...) or the main route reserved by the module (for website module mainly, i.e. : 'o_forum' for website_forum module). The only exception for this rule is the webclient : it simply use *o_* prefix.
- Avoid using id
- Use bootstrap native class
- Use underscore lowercase notation to name class

Build an Odoo module

Both server and client extensions are packaged as *modules* which are optionally loaded in a *database*.

Odoo modules can either add brand new business logic to an Odoo system, or alter and extend existing business logic: a module can be created to add your country's accounting rules to Odoo's generic accounting support, while the next module adds support for real-time visualisation of a bus fleet.

Everything in Odoo thus starts and ends with modules.

Composition of a module

An Odoo module can contain a number of elements:

Business objects

declared as Python classes, these resources are automatically persisted by Odoo based on their configuration

Data files

XML or CSV files declaring metadata (views or workflows), configuration data (modules parameterization), demonstration data and more

Web controllers

Handle requests from web browsers

Static web data

Images, CSS or javascript files used by the web interface or website

Module structure

Each module is a directory within a *module directory*. Module directories are specified by using the `--addons-path` option.

Tip

most command-line options can also be set using **a configuration file**

An Odoo module is declared by its **manifest**. See the **manifest documentation** information about it.

A module is also a **Python package** with a `__init__.py` file, containing import instructions for various Python files in the module.

For instance, if the module has a single `mymodule.py` file `__init__.py` might contain:

```
from . import mymodule
```

Odoo provides a mechanism to help set up a new module, **odoo.py** has a subcommand **scaffold** to create an empty module:

```
$ odoo.py scaffold <module name> <where to put it>
```

The command creates a subdirectory for your module, and automatically creates a bunch of standard files for a module. Most of them simply contain commented code or XML. The usage of most of those files will be explained along this tutorial.

Exercise

Module creation

Use the command line above to create an empty module Open Academy, and install it in Odoo.

1. Invoke the command `odoo.py scaffold openacademy addons`.
2. Adapt the manifest file to your module.
3. Don't bother about the other files.

openacademy/__openerp__.py

```
# -*- coding: utf-8 -*-
{
    'name': "Open Academy",

    'summary': """Manage trainings""",

    'description': """
        Open Academy module for managing trainings:
            - training courses
            - training sessions
            - attendees registration
    """,

    'author': "Your Company",
    'website': "http://www.yourcompany.com",

    # Categories can be used to filter modules in modules listing
    # Check https://github.com/odoo/odoo/blob/master/openerp/addons/base/module/module_data
.xml
    # for the full list
    'category': 'Test',
    'version': '0.1',

    # any module necessary for this one to work correctly
    'depends': ['base'],
```

```
    # always loaded
    'data': [
        # 'security/ir.model.access.csv',
        'templates.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
        'demo.xml',
    ],
}
```

openacademy/__init__.py

```
# -*- coding: utf-8 -*-
from . import controllers
from . import models
```

openacademy/controllers.py

```
# -*- coding: utf-8 -*-
from openerp import http

# class Openacademy(http.Controller):
#     @http.route('/openacademy/openacademy/', auth='public')
#     def index(self, **kw):
#         return "Hello, world"

#     @http.route('/openacademy/openacademy/objects/', auth='public')
#     def list(self, **kw):
#         return http.request.render('openacademy.listing', {
#             'root': '/openacademy/openacademy',
#             'objects': http.request.env['openacademy.openacademy'].search([]),
#         })

#     @http.route('/openacademy/openacademy/objects/<model("openacademy.openacademy"):obj>/
', auth='public')
#     def object(self, obj, **kw):
#         return http.request.render('openacademy.object', {
#             'object': obj
#         })
```

openacademy/demo.xml

```
<openerp>
    <data>
        <!--  -->
        <!--   <record id="object0" model="openacademy.openacademy"> -->
        <!--     <field name="name">Object 0</field> -->
        <!--   </record> -->
        <!--  -->
        <!--   <record id="object1" model="openacademy.openacademy"> -->
        <!--     <field name="name">Object 1</field> -->
        <!--   </record> -->
        <!--  -->
        <!--   <record id="object2" model="openacademy.openacademy"> -->
        <!--     <field name="name">Object 2</field> -->
        <!--   </record> -->
        <!--  -->
        <!--   <record id="object3" model="openacademy.openacademy"> -->
        <!--     <field name="name">Object 3</field> -->
        <!--   </record> -->
```

```
        <!--  -->
        <!--   <record id="object4" model="openacademy.openacademy"> -->
        <!--     <field name="name">Object 4</field> -->
        <!--   </record> -->
        <!--  -->
    </data>
</openerp>
```

openacademy/models.py

```
# -*- coding: utf-8 -*-

from openerp import models, fields, api

# class openacademy(models.Model):
#     _name = 'openacademy.openacademy'

#     name = fields.Char()
```

openacademy/security/ir.model.access.csv

```
id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_unlink
access_openacademy_openacademy,openacademy.openacademy,model_openacademy_openacademy,,1,0,0
,0
```

openacademy/templates.xml

```
<openerp>
    <data>
        <!-- <template id="listing"> -->
        <!--   <ul> -->
        <!--     <li t-foreach="objects" t-as="object"> -->
        <!--       <a t-attf-href="{{ root }}/objects/{{ object.id }}"> -->
        <!--         <t t-esc="object.display_name"/> -->
        <!--       </a> -->
        <!--     </li> -->
        <!--   </ul> -->
        <!-- </template> -->
        <!-- <template id="object"> -->
        <!--   <h1><t t-esc="object.display_name"/></h1> -->
        <!--   <dl> -->
        <!--     <t t-foreach="object._fields" t-as="field"> -->
        <!--       <dt><t t-esc="field"/></dt> -->
        <!--       <dd><t t-esc="object[field]"/></dd> -->
        <!--     </t> -->
        <!--   </dl> -->
        <!-- </template> -->
    </data>
</openerp>
```

Object-Relational Mapping

A key component of Odoo is the ORM layer. This layer avoids having to write most SQL by hand and provides extensibility and security services[2].

Business objects are declared as Python classes extending `Model` which integrates them into the automated persistence system.

Models can be configured by setting a number of attributes at their definition. The most important attribute is `_name` which is required and defines the name for the model in the Odoo system. Here is a minimally complete definition of a model:

```
from openerp import models
class MinimalModel(models.Model):
    _name = 'test.model'
```

Model fields

Fields are used to define what the model can store and where. Fields are defined as attributes on the model class:

```
from openerp import models, fields

class LessMinimalModel(models.Model):
    _name = 'test.model2'

    name = fields.Char()
```

Common Attributes

Much like the model itself, its fields can be configured, by passing configuration attributes as parameters:

```
name = field.Char(required=True)
```

Some attributes are available on all fields, here are the most common ones:

`string` (`unicode`, default: field's name)
The label of the field in UI (visible by users).
`required` (`bool`, default: `False`)
If `True`, the field can not be empty, it must either have a default value or always be given a value when creating a record.
`help` (`unicode`, default: `''`)
Long-form, provides a help tooltip to users in the UI.
`index` (`bool`, default: `False`)
Requests that Odoo create a **database index** on the column

Simple fields

There are two broad categories of fields: "simple" fields which are atomic values stored directly in the model's table and "relational" fields linking records (of the same model or of different models).

Example of simple fields are `Boolean`, `Date`, `Char`.

Reserved fields

Odoo creates a few fields in all models[1]. These fields are managed by the system and shouldn't be written to. They can be read if useful or necessary:

`id` (`Id`)
the unique identifier for a record in its model
`create_date` (`Datetime`)
creation date of the record
`create_uid` (`Many2one`)
user who created the record
`write_date` (`Datetime`)
last modification date of the record
`write_uid` (`Many2one`)
user who last modified the record

Special fields

By default, Odoo also requires a `name` field on all models for various display and search behaviors. The field used for these purposes can be overridden by setting `_rec_name`.

Data files

Odoo is a highly data driven system. Although behavior is customized using **Python** code part of a module's value is in the data it sets up when loaded.

Module data is declared via **data files**, XML files with `<record>` elements. Each `<record>` element creates or updates a database record.

```
<openerp>
    <data>
        <record model="{model name}" id="{record identifier}">
```

```
            <field name="{a field name}">{a value}</field>
        </record>
    </data>
<openerp>
```

- `model` is the name of the Odoo model for the record
- `id` is an **external identifier**, it allows referring to the record (without having to know its in-database identifier)
- `<field>` elements have a `name` which is the name of the field in the model (e.g. `description`). Their body is the field's value.

Data files have to be declared in the manifest file to be loaded, they can be declared in the `'data'` list (always loaded) or in the `'demo'` list (only loaded in demonstration mode).

Exercise

Define demonstration data

Create demonstration data filling the *Courses* model with a few demonstration courses.

Edit the file `openacademy/demo.xml` to include some data.

openacademy/demo.xml

```
<openerp>
    <data>
        <record model="openacademy.course" id="course0">
            <field name="name">Course 0</field>
            <field name="description">Course 0's description

Can have multiple lines
            </field>
        </record>
        <record model="openacademy.course" id="course1">
            <field name="name">Course 1</field>
            <!-- no description for this one -->
        </record>
        <record model="openacademy.course" id="course2">
            <field name="name">Course 2</field>
            <field name="description">Course 2's description</field>
        </record>
    </data>
</openerp>
```

Actions and Menus

Actions and menus are regular records in database, usually declared through data files. Actions can be triggered in three ways:

1. by clicking on menu items (linked to specific actions)
2. by clicking on buttons in views (if these are connected to actions)
3. as contextual actions on object

Because menus are somewhat complex to declare there is a **`<menuitem>`** shortcut to declare an **`ir.ui.menu`** and connect it to the corresponding action more easily.

```
<record model="ir.actions.act_window" id="action_list_ideas">
    <field name="name">Ideas</field>
    <field name="res_model">idea.idea</field>
    <field name="view_mode">tree,form</field>
</record>
<menuitem id="menu_ideas" parent="menu_root" name="Ideas" sequence=
"10"
          action="action_list_ideas"/>
```

Exercise

Define new menu entries

Define new menu entries to access courses and sessions under the OpenAcademy menu entry. A user should be able to

- display a list of all the courses
- create/modify courses

1. Create **`openacademy/views/openacademy.xml`** with an action and the menus triggering the action
2. Add it to the **`data`** list of **`openacademy/__openerp__.py`**

openacademy/__openerp__.py

```
    'data': [
        # 'security/ir.model.access.csv',
        'templates.xml',
        'views/openacademy.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
```

openacademy/views/openacademy.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
    <data>
        <!-- window action -->
        <!--
            The following tag is an action definition for a "window action",
            that is an action opening a view or a set of views
        -->
        <record model="ir.actions.act_window" id="course_list_action">
```

```
            <field name="name">Courses</field>
            <field name="res_model">openacademy.course</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form</field>
            <field name="help" type="html">
                <p class="oe_view_nocontent_create">Create the first course
                </p>
            </field>
        </record>

        <!-- top level menu: no parent -->
        <menuitem id="main_openacademy_menu" name="Open Academy"/>
        <!-- A first level in the left side menu is needed
             before using action= attribute -->
        <menuitem id="openacademy_menu" name="Open Academy"
                  parent="main_openacademy_menu"/>
        <!-- the following menuitem should appear *after*
             its parent openacademy_menu and *after* its
             action course_list_action -->
        <menuitem id="courses_menu" name="Courses" parent="openacademy_menu"
                  action="course_list_action"/>
        <!-- Full id location:
             action="openacademy.course_list_action"
             It is not required when it is the same module -->
    </data>
</openerp>
```

Basic views

Views define the way the records of a model are displayed. Each type of view represents a mode of visualization (a list of records, a graph of their aggregation, …). Views can either be requested generically via their type (e.g. *a list of partners*) or specifically via their id. For generic requests, the view with the correct type and the lowest priority will be used (so the lowest-priority view of each type is the default view for that type).

View inheritance allows altering views declared elsewhere (adding or removing content).

Generic view declaration

A view is declared as a record of the model `ir.ui.view`. The view type is implied by the root element of the `arch` field:

```
<record model="ir.ui.view" id="view_id">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
```

```
    <field name="arch" type="xml">
        <!-- view content: <form>, <tree>, <graph>, ... -->
    </field>
</record>
```

Tree views

Tree views, also called list views, display records in a tabular form.

Their root element is `<tree>`. The simplest form of the tree view simply lists all the fields to display in the table (each field as a column):

```
<tree string="Idea list">
    <field name="name"/>
    <field name="inventor_id"/>
</tree>
```

Form views

Forms are used to create and edit single records.

Their root element is `<form>`. They composed of high-level structure elements (groups, notebooks) and interactive elements (buttons and fields):

```
<form string="Idea form">
    <group colspan="4">
        <group colspan="2" col="2">
            <separator string="General stuff" colspan="2"/>
            <field name="name"/>
            <field name="inventor_id"/>
        </group>

        <group colspan="2" col="2">
            <separator string="Dates" colspan="2"/>
            <field name="active"/>
            <field name="invent_date" readonly="1"/>
        </group>

        <notebook colspan="4">
            <page string="Description">
                <field name="description" nolabel="1"/>
```

```
            </page>
        </notebook>

        <field name="state"/>
    </group>
</form>
```

Exercise

Customise form view using XML

Create your own form view for the Course object. Data displayed should be: the name and the description of the course.

openacademy/views/openacademy.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
    <data>
        <record model="ir.ui.view" id="course_form_view">
            <field name="name">course.form</field>
            <field name="model">openacademy.course</field>
            <field name="arch" type="xml">
                <form string="Course Form">
                    <sheet>
                        <group>
                            <field name="name"/>
                            <field name="description"/>
                        </group>
                    </sheet>
                </form>
            </field>
        </record>

        <!-- window action -->
        <!--
            The following tag is an action definition for a "window action",
```

Exercise

Notebooks

In the Course form view, put the description field under a tab, such that it will be easier to add other tabs later, containing additional information.

Modify the Course form view as follows:

openacademy/views/openacademy.xml

```
                    <sheet>
                        <group>
                            <field name="name"/>
                        </group>
                        <notebook>
                            <page string="Description">
                                <field name="description"/>
                            </page>
                            <page string="About">
```

```
                            This is an example of notebooks
                        </page>
                    </notebook>
                </sheet>
            </form>
        </field>
```

Form views can also use plain HTML for more flexible layouts:

```
<form string="Idea Form">
    <header>
        <button string="Confirm" type="object" name="action_confirm
"
                states="draft" class="oe_highlight" />
        <button string="Mark as done" type="object" name="action_do
ne"
                states="confirmed" class="oe_highlight"/>
        <button string="Reset to draft" type="object" name="action_
draft"
                states="confirmed,done" />
        <field name="state" widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_title">
            <label for="name" class="oe_edit_only" string="Idea Nam
e" />
            <h1><field name="name" /></h1>
        </div>
        <separator string="General" colspan="2" />
        <group colspan="2" col="2">
            <field name="description" placeholder="Idea description
..." />
        </group>
    </sheet>
</form>
```

Search views

Search views customize the search field associated with the list view (and other aggregated views). Their root element is `<search>`and they're composed of fields defining which fields can be searched on:

```xml
<search>
    <field name="name"/>
    <field name="inventor_id"/>
</search>
```

If no search view exists for the model, Odoo generates one which only allows searching on the `name` field.

Exercise

Search courses

Allow searching for courses based on their title or their description.

openacademy/views/openacademy.xml

```xml
            </field>
        </record>

        <record model="ir.ui.view" id="course_search_view">
            <field name="name">course.search</field>
            <field name="model">openacademy.course</field>
            <field name="arch" type="xml">
                <search>
                    <field name="name"/>
                    <field name="description"/>
                </search>
            </field>
        </record>

        <!-- window action -->
        <!--
            The following tag is an action definition for a "window action",
```

Relations between models

A record from a model may be related to a record from another model. For instance, a sale order record is related to a client record that contains the client data; it is also related to its sale order line records.

Exercise

Create a session model

For the module Open Academy, we consider a model for *sessions*: a session is an occurrence of a course taught at a given time for a given audience.

Create a model for *sessions*. A session has a name, a start date, a duration and a number of seats. Add an action and a menu item to display them. Make the new model visible via a menu item.

1. Create the class *Session* in `openacademy/models.py`.
2. Add access to the session object in `openacademy/view/openacademy.xml`.

openacademy/models.py

```
    name = fields.Char(string="Title", required=True)
    description = fields.Text()


class Session(models.Model):
    _name = 'openacademy.session'

    name = fields.Char(required=True)
    start_date = fields.Date()
    duration = fields.Float(digits=(6, 2), help="Duration in days")
    seats = fields.Integer(string="Number of seats")
```

openacademy/views/openacademy.xml

```
        <!-- Full id location:
             action="openacademy.course_list_action"
             It is not required when it is the same module -->

        <!-- session form view -->
        <record model="ir.ui.view" id="session_form_view">
            <field name="name">session.form</field>
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <form string="Session Form">
                    <sheet>
                        <group>
                            <field name="name"/>
                            <field name="start_date"/>
                            <field name="duration"/>
                            <field name="seats"/>
                        </group>
                    </sheet>
                </form>
            </field>
        </record>
```

```
        <record model="ir.actions.act_window" id="session_list_action">
            <field name="name">Sessions</field>
            <field name="res_model">openacademy.session</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form</field>
        </record>

        <menuitem id="session_menu" name="Sessions"
                  parent="openacademy_menu"
                  action="session_list_action"/>
    </data>
</openerp>
```

Note

`digits=(6, 2)` specifies the precision of a float number: 6 is the total number of digits, while 2 is the number of digits after the comma. Note that it results in the number digits before the comma is a maximum 4

Relational fields

Relational fields link records, either of the same model (hierarchies) or between different models.

Relational field types are:

`Many2one(other_model, ondelete='set null')`

A simple link to an other object:

```
print foo.other_id.name
```

See also

foreign keys

`One2many(other_model, related_field)`

A virtual relationship, inverse of a `Many2one`. A `One2many` behaves as a container of records, accessing it results in a (possibly empty) set of records:

```
for other in foo.other_ids:
    print other.name
```

`Many2many(other_model)`

Bidirectional multiple relationship, any record on one side can be related to any number of records on the other side. Behaves as a container of records, accessing it also results in a possibly empty set of records:

```
for other in foo.other_ids:
    print other.name
```

Exercise

Many2one relations

Using a many2one, modify the *Course* and *Session* models to reflect their relation with other models:

- A course has a *responsible* user; the value of that field is a record of the built-in model `res.users`.
- A session has an *instructor*; the value of that field is a record of the built-in model `res.partner`.
- A session is related to a *course*; the value of that field is a record of the model `openacademy.course` and is required.
- Adapt the views.

1. Add the relevant `Many2one` fields to the models, and
2. add them in the views.

openacademy/models.py

```
    name = fields.Char(string="Title", required=True)
    description = fields.Text()

    responsible_id = fields.Many2one('res.users',
        ondelete='set null', string="Responsible", index=True)


class Session(models.Model):
    _name = 'openacademy.session'
    start_date = fields.Date()
    duration = fields.Float(digits=(6, 2), help="Duration in days")
    seats = fields.Integer(string="Number of seats")

    instructor_id = fields.Many2one('res.partner', string="Instructor")
    course_id = fields.Many2one('openacademy.course',
        ondelete='cascade', string="Course", required=True)
```

openacademy/views/openacademy.xml

```
                    <sheet>
                        <group>
                            <field name="name"/>
                            <field name="responsible_id"/>
                        </group>
                        <notebook>
                            <page string="Description">
            </field>
        </record>

        <!-- override the automatically generated list view for courses -->
        <record model="ir.ui.view" id="course_tree_view">
            <field name="name">course.tree</field>
            <field name="model">openacademy.course</field>
            <field name="arch" type="xml">
                <tree string="Course Tree">
```

```
                <field name="name"/>
                <field name="responsible_id"/>
            </tree>
        </field>
    </record>

    <!-- window action -->
    <!--
        The following tag is an action definition for a "window action",
            <form string="Session Form">
                <sheet>
                    <group>
                        <group string="General">
                            <field name="course_id"/>
                            <field name="name"/>
                            <field name="instructor_id"/>
                        </group>
                        <group string="Schedule">
                            <field name="start_date"/>
                            <field name="duration"/>
                            <field name="seats"/>
                        </group>
                    </group>
                </sheet>
            </form>
        </field>
    </record>

    <!-- session tree/list view -->
    <record model="ir.ui.view" id="session_tree_view">
        <field name="name">session.tree</field>
        <field name="model">openacademy.session</field>
        <field name="arch" type="xml">
            <tree string="Session Tree">
                <field name="name"/>
                <field name="course_id"/>
            </tree>
        </field>
    </record>

    <record model="ir.actions.act_window" id="session_list_action">
        <field name="name">Sessions</field>
        <field name="res_model">openacademy.session</field>
```

Exercise

Inverse one2many relations

Using the inverse relational field one2many, modify the models to reflect the relation between courses and sessions.

1. Modify the `Course` class, and
2. add the field in the course form view.

openacademy/models.py

```
    responsible_id = fields.Many2one('res.users',
        ondelete='set null', string="Responsible", index=True)
    session_ids = fields.One2many(
        'openacademy.session', 'course_id', string="Sessions")


class Session(models.Model):
```

openacademy/views/openacademy.xml

```
                            <page string="Description">
                                <field name="description"/>
                            </page>
                            <page string="Sessions">
                                <field name="session_ids">
                                    <tree string="Registered sessions">
                                        <field name="name"/>
                                        <field name="instructor_id"/>
                                    </tree>
                                </field>
                            </page>
                        </notebook>
                    </sheet>
```

Exercise

Multiple many2many relations

Using the relational field many2many, modify the *Session* model to relate every session to a set of *attendees*. Attendees will be represented by partner records, so we will relate to the built-in model `res.partner`. Adapt the views accordingly.

1. Modify the `Session` class, and
2. add the field in the form view.

openacademy/models.py

```
    instructor_id = fields.Many2one('res.partner', string="Instructor")
    course_id = fields.Many2one('openacademy.course',
        ondelete='cascade', string="Course", required=True)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")
```

openacademy/views/openacademy.xml

```
                                <field name="seats"/>
                            </group>
                        </group>
                        <label for="attendee_ids"/>
                        <field name="attendee_ids"/>
                    </sheet>
                </form>


            </field>
```

Inheritance

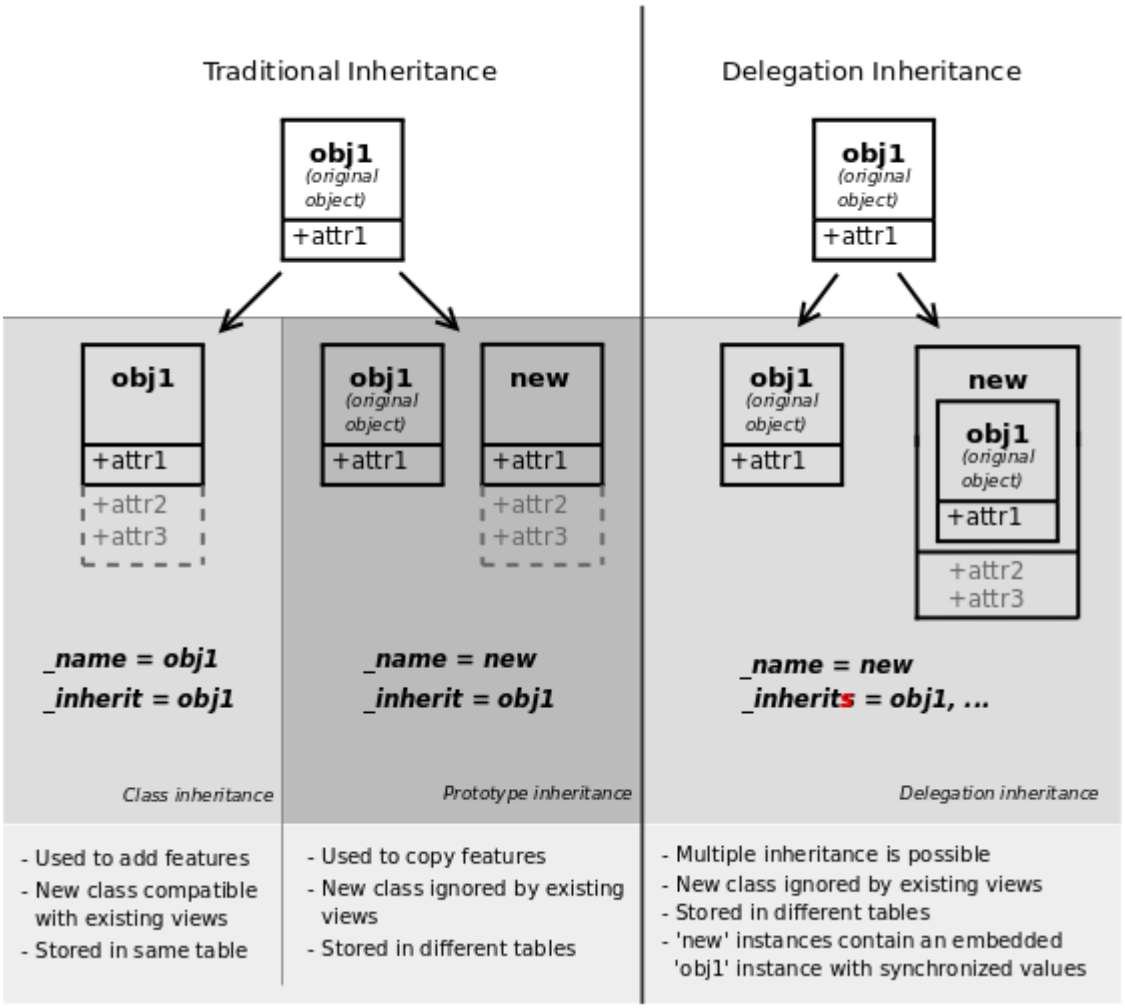
Model inheritance

Odoo provides two *inheritance* mechanisms to extend an existing model in a modular way.

The first inheritance mechanism allows a module to modify the behavior of a model defined in another module:

- add fields to a model,
- override the definition of fields on a model,
- add constraints to a model,
- add methods to a model,
- override existing methods on a model.

The second inheritance mechanism (delegation) allows to link every record of a model to a record in a parent model, and provides transparent access to the fields of the parent record.



View inheritance

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

An extension view references its parent using the `inherit_id` field, and instead of a single view its `arch` field is composed of any number of `xpath` elements selecting and altering the content of their parent view:

```
<!-- improved idea categories list -->
<record id="idea_category_list2" model="ir.ui.view">
    <field name="name">id.category.list2</field>
```

```
    <field name="model">idea.category</field>
    <field name="inherit_id" ref="id_category_list"/>
    <field name="arch" type="xml">
        <!-- find field description and add the field
             idea_ids after it -->
        <xpath expr="//field[@name='description']" position="after"
>
          <field name="idea_ids" string="Number of ideas"/>
        </xpath>
    </field>
</record>
```

`expr`

An XPath expression selecting a single element in the parent view. Raises an error if it matches no element or more than one

`position`

Operation to apply to the matched element:

`inside`

appends `xpath`'s body at the end of the matched element

`replace`

replaces the matched element by the `xpath`'s body

`before`

inserts the `xpath`'s body as a sibling before the matched element

`after`

inserts the `xpaths`'s body as a sibling after the matched element

`attributes`

alters the attributes of the matched element using special `attribute` elements in the `xpath`'s body

Tip

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

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

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

Exercise

Alter existing content

- Using model inheritance, modify the existing *Partner* model to add an `instructor` boolean field, and a many2many field that corresponds to the session-partner relation

- Using view inheritance, display this fields in the partner form view

> **Note**
>
> This is the opportunity to introduce the developer mode to inspect the view, find its external ID and the place to put the new field.

1. Create a file `openacademy/partner.py` and import it in `__init__.py`
2. Create a file `openacademy/views/partner.xml` and add it to `__openerp__.py`

openacademy/__init__.py

```
# -*- coding: utf-8 -*-
from . import controllers
from . import models
from . import partner
```

openacademy/__openerp__.py

```
        # 'security/ir.model.access.csv',
        'templates.xml',
        'views/openacademy.xml',
        'views/partner.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
```

openacademy/partner.py

```
# -*- coding: utf-8 -*-
from openerp import fields, models

class Partner(models.Model):
    _inherit = 'res.partner'

    # Add a new column to the res.partner model, by default partners are not
    # instructors
    instructor = fields.Boolean("Instructor", default=False)

    session_ids = fields.Many2many('openacademy.session',
        string="Attended Sessions", readonly=True)
```

openacademy/views/partner.xml

```
<?xml version="1.0" encoding="UTF-8"?>
 <openerp>
    <data>
        <!-- Add instructor field to existing view -->
        <record model="ir.ui.view" id="partner_instructor_form_view">
            <field name="name">partner.instructor</field>
            <field name="model">res.partner</field>
            <field name="inherit_id" ref="base.view_partner_form"/>
            <field name="arch" type="xml">
                <notebook position="inside">
                    <page string="Sessions">
                        <group>
                            <field name="instructor"/>
                            <field name="session_ids"/>
                        </group>
```

```
                    </page>
                </notebook>
            </field>
        </record>

        <record model="ir.actions.act_window" id="contact_list_action">
            <field name="name">Contacts</field>
            <field name="res_model">res.partner</field>
            <field name="view_mode">tree,form</field>
        </record>
        <menuitem id="configuration_menu" name="Configuration"
                  parent="main_openacademy_menu"/>
        <menuitem id="contact_menu" name="Contacts"
                  parent="configuration_menu"
                  action="contact_list_action"/>
    </data>
</openerp>
```

Domains

In Odoo, **Domains** are values that encode conditions on records. A domain is a list of criteria used to select a subset of a model's records. Each criteria is a triple with a field name, an operator and a value.

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

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

By default criteria are combined with an implicit AND. The logical operators `&` (AND), `|` (OR) and `!` (NOT) can be used to explicitly combine criteria. They are used in prefix position (the operator is inserted before its arguments rather than between). For instance to select products "which are services *OR* have a unit price which is *NOT* between 1000 and 2000":

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

A `domain` parameter can be added to relational fields to limit valid records for the relation when trying to select records in the client interface.

Exercise

Domains on relational fields

When selecting the instructor for a *Session*, only instructors (partners with `instructor` set to `True`) should be visible.

openacademy/models.py

```
    duration = fields.Float(digits=(6, 2), help="Duration in days")
    seats = fields.Integer(string="Number of seats")

    instructor_id = fields.Many2one('res.partner', string="Instructor",
        domain=[('instructor', '=', True)])
    course_id = fields.Many2one('openacademy.course',
        ondelete='cascade', string="Course", required=True)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")
```

Note

A domain declared as a literal list is evaluated server-side and can't refer to dynamic values on the right-hand side, a domain declared as a string is evaluated client-side and allows field names on the right-hand side

Exercise

More complex domains

Create new partner categories *Teacher / Level 1* and *Teacher / Level 2*. The instructor for a session can be either an instructor or a teacher (of any level).

1. Modify the *Session* model's domain
2. Modify `openacademy/view/partner.xml` to get access to *Partner categories*:

openacademy/models.py

```
    seats = fields.Integer(string="Number of seats")

    instructor_id = fields.Many2one('res.partner', string="Instructor",
        domain=['|', ('instructor', '=', True),
                     ('category_id.name', 'ilike', "Teacher")])
    course_id = fields.Many2one('openacademy.course',
        ondelete='cascade', string="Course", required=True)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")
```

openacademy/views/partner.xml

```
        <menuitem id="contact_menu" name="Contacts"
                  parent="configuration_menu"
                  action="contact_list_action"/>

        <record model="ir.actions.act_window" id="contact_cat_list_action">
            <field name="name">Contact Tags</field>
            <field name="res_model">res.partner.category</field>
            <field name="view_mode">tree,form</field>
        </record>
        <menuitem id="contact_cat_menu" name="Contact Tags"
                  parent="configuration_menu"
                  action="contact_cat_list_action"/>

        <record model="res.partner.category" id="teacher1">
            <field name="name">Teacher / Level 1</field>
        </record>
        <record model="res.partner.category" id="teacher2">
```

```
            <field name="name">Teacher / Level 2</field>
        </record>
    </data>
</openerp>
```

Computed fields and default values

So far fields have been stored directly in and retrieved directly from the database. Fields can also be *computed*. In that case, the field's value is not retrieved from the database but computed on-the-fly by calling a method of the model.

To create a computed field, create a field and set its attribute `compute` to the name of a method. The computation method should simply set the value of the field to compute on every record in `self`.

```
import random
from openerp import models, fields

class ComputedModel(models.Model):
    _name = 'test.computed'

    name = fields.Char(compute='_compute_name')

    @api.multi
    def _compute_name(self):
        for record in self:
            record.name = str(random.randint(1, 1e6))
```

Our compute method is very simple: it loops over `self` and performs the same operation on every record. We can make it slightly simpler by using the decorator `one()` to automatically loop on the collection:

```
@api.one
def _compute_name(self):
    self.name = str(random.randint(1, 1e6))
```

Dependencies

The value of a computed field usually depends on the values of other fields on the computed record. The ORM expects the developer to specify those dependencies on the compute method with the decorator `depends()`. The given dependencies are used by the ORM to trigger the recomputation of the field whenever some of its dependencies have been modified:

```
from openerp import models, fields, api

class ComputedModel(models.Model):
    _name = 'test.computed'

    name = fields.Char(compute='_compute_name')
    value = fields.Integer()

    @api.one
    @api.depends('value')
    def _compute_name(self):
        self.name = "Record with value %s" % self.value
```

Exercise

Computed fields

- Add the percentage of taken seats to the *Session* model
- Display that field in the tree and form views
- Display the field as a progress bar

1. Add a computed field to *Session*
2. Show the field in the *Session* view:

openacademy/models.py

```
    course_id = fields.Many2one('openacademy.course',
        ondelete='cascade', string="Course", required=True)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")

    taken_seats = fields.Float(string="Taken seats", compute='_taken_seats')

    @api.one
    @api.depends('seats', 'attendee_ids')
    def _taken_seats(self):
        if not self.seats:
            self.taken_seats = 0.0
        else:
            self.taken_seats = 100.0 * len(self.attendee_ids) / self.seats
```

openacademy/views/openacademy.xml

```
                                <field name="start_date"/>
                                <field name="duration"/>
                                <field name="seats"/>
                                <field name="taken_seats" widget="progressbar"/>
                            </group>
                        </group>
```

```
                    <label for="attendee_ids"/>
                <tree string="Session Tree">
                    <field name="name"/>
                    <field name="course_id"/>
                    <field name="taken_seats" widget="progressbar"/>
                </tree>
            </field>
        </record>
```

Default values

Any field can be given a default value. In the field definition, add the option `default=X` where `X` is either a Python literal value (boolean, integer, float, string), or a function taking a recordset and returning a value:

```
name = fields.Char(default="Unknown")
user_id = fields.Many2one('res.users', default=lambda self: self.en
v.user)
```

Note

The object `self.env` gives access to request parameters and other useful things:

- `self.env.cr` or `self._cr` is the database *cursor* object; it is used for querying the database
- `self.env.uid` or `self._uid` is the current user's database id
- `self.env.user` is the current user's record
- `self.env.context` or `self._context` is the context dictionary
- `self.env.ref(xml_id)` returns the record corresponding to an XML id
- `self.env[model_name]` returns an instance of the given model

Exercise

Active objects – Default values

- Define the start_date default value as today (see `Date`).
- Add a field `active` in the class Session, and set sessions as active by default.

openacademy/models.py

```
    _name = 'openacademy.session'

    name = fields.Char(required=True)
    start_date = fields.Date(default=fields.Date.today)
    duration = fields.Float(digits=(6, 2), help="Duration in days")
    seats = fields.Integer(string="Number of seats")
    active = fields.Boolean(default=True)

    instructor_id = fields.Many2one('res.partner', string="Instructor",
        domain=['|', ('instructor', '=', True),
```

openacademy/views/openacademy.xml

```
                                <field name="course_id"/>
                                <field name="name"/>
                                <field name="instructor_id"/>
                                <field name="active"/>
                            </group>
                            <group string="Schedule">
                                <field name="start_date"/>
```

Note

Odoo has built-in rules making fields with an **`active`** field set to **`False`**invisible.

Onchange

The "onchange" mechanism provides a way for the client interface to update a form whenever the user has filled in a value in a field, without saving anything to the database.

For instance, suppose a model has three fields **`amount`**, **`unit_price`** and **`price`**, and you want to update the price on the form when any of the other fields is modified. To achieve this, define a method where **`self`** represents the record in the form view, and decorate it with **`onchange()`** to specify on which field it has to be triggered. Any change you make on **`self`** will be reflected on the form.

```
<!-- content of form view -->
<field name="amount"/>
<field name="unit_price"/>
<field name="price" readonly="1"/>
# onchange handler
@api.onchange('amount', 'unit_price')
def _onchange_price(self):
    # set auto-changing field
    self.price = self.amount * self.unit_price
    # Can optionally return a warning and domains
    return {
        'warning': {
            'title': "Something bad happened",
            'message': "It was very bad indeed",
        }
    }
```

For computed fields, valued **`onchange`** behavior is built-in as can be seen by playing with the *Session* form: change the number of seats or participants, and the **`taken_seats`** progressbar is automatically updated.

Exercise

Warning

Add an explicit onchange to warn about invalid values, like a negative number of seats, or more participants than seats.

openacademy/models.py

```
            self.taken_seats = 0.0
        else:
            self.taken_seats = 100.0 * len(self.attendee_ids) / self.seats

    @api.onchange('seats', 'attendee_ids')
    def _verify_valid_seats(self):
        if self.seats < 0:
            return {
                'warning': {
                    'title': "Incorrect 'seats' value",
                    'message': "The number of available seats may not be negative",
                },
            }
        if self.seats < len(self.attendee_ids):
            return {
                'warning': {
                    'title': "Too many attendees",
                    'message': "Increase seats or remove excess attendees",
                },
            }
```

Model constraints

Odoo provides two ways to set up automatically verified invariants: `Python constraints` and `SQL constraints`.

A Python constraint is defined as a method decorated with `constrains()`, and invoked on a recordset. The decorator specifies which fields are involved in the constraint, so that the constraint is automatically evaluated when one of them is modified. The method is expected to raise an exception if its invariant is not satisfied:

```
from openerp.exceptions import ValidationError

@api.constrains('age')
def _check_something(self):
    for record in self:
        if record.age > 20:
            raise ValidationError("Your record is too old: %s" % re
cord.age)
    # all records passed the test, don't return anything
```

Exercise

Add Python constraints

Add a constraint that checks that the instructor is not present in the attendees of his/her own session.

openacademy/models.py

```
# -*- coding: utf-8 -*-

from openerp import models, fields, api, exceptions

class Course(models.Model):
    _name = 'openacademy.course'
                    'message': "Increase seats or remove excess attendees",
                },
            }

    @api.one
    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        if self.instructor_id and self.instructor_id in self.attendee_ids:
            raise exceptions.ValidationError("A session's instructor can't be an attendee")
```

SQL constraints are defined through the model attribute `_sql_constraints`. The latter is assigned to a list of triples of strings`(name, sql_definition, message)`, where `name` is a valid SQL constraint name, `sql_definition` is a table_constraintexpression, and `message` is the error message.

Exercise

Add SQL constraints

With the help of **PostgreSQL's documentation** , add the following constraints:

1. CHECK that the course description and the course title are different
2. Make the Course's name UNIQUE

openacademy/models.py

```
    session_ids = fields.One2many(
        'openacademy.session', 'course_id', string="Sessions")

    _sql_constraints = [
        ('name_description_check',
         'CHECK(name != description)',
         "The title of the course should not be the description"),

        ('name_unique',
         'UNIQUE(name)',
         "The course title must be unique"),
    ]


class Session(models.Model):
    _name = 'openacademy.session'
```

Exercise

Exercise 6 - Add a duplicate option

Since we added a constraint for the Course name uniqueness, it is not possible to use the "duplicate" function anymore (**Form ‣ Duplicate**).

Re-implement your own "copy" method which allows to duplicate the Course object, changing the original name into "Copy of [original name]".

openacademy/models.py

```
    session_ids = fields.One2many(
        'openacademy.session', 'course_id', string="Sessions")

    @api.one
    def copy(self, default=None):
        default = dict(default or {})

        copied_count = self.search_count(
            [('name', '=like', u"Copy of {}%".format(self.name))])
        if not copied_count:
            new_name = u"Copy of {}".format(self.name)
        else:
            new_name = u"Copy of {} ({})".format(self.name, copied_count)

        default['name'] = new_name
        return super(Course, self).copy(default)

    _sql_constraints = [
        ('name_description_check',
         'CHECK(name != description)',
```

Advanced Views

Tree views

Tree views can take supplementary attributes to further customize their behavior:

`colors`

mappings of colors to conditions. If the condition evaluates to `True`, the corresponding color is applied to the row:

```
<tree string="Idea Categories" colors="blue:state=='draft';red:stat
e=='trashed'">
    <field name="name"/>
    <field name="state"/>
</tree>
```

Clauses are separated by `;`, the color and condition are separated by `:`.

`editable`

Either `"top"` or `"bottom"`. Makes the tree view editable in-place (rather than having to go through the form view), the value is the position where new rows appear.

Exercise

List coloring

Modify the Session tree view in such a way that sessions lasting less than 5 days are colored blue, and the ones lasting more than 15 days are colored red.

Modify the session tree view:

openacademy/views/openacademy.xml

```
            <field name="name">session.tree</field>
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <tree string="Session Tree" colors="#0000ff:duration&lt;5;red:dura
tion&gt;15">
                    <field name="name"/>
                    <field name="course_id"/>
                    <field name="duration" invisible="1"/>
                    <field name="taken_seats" widget="progressbar"/>
                </tree>
            </field>
```

Calendars

Displays records as calendar events. Their root element is `<calendar>` and their most common attributes are:

`color`

The name of the field used for *color segmentation*. Colors are automatically distributed to events, but events in the same color segment (records which have the same value for their `@color` field) will be given the same color.

date_start
record's field holding the start date/time for the event
date_stop (optional)
record's field holding the end date/time for the event
field (to define the label for each calendar event)

```
<calendar string="Ideas" date_start="invent_date" color="inventor_i
d">
    <field name="name"/>
</calendar>
```

Exercise

Calendar view

Add a Calendar view to the *Session* model enabling the user to view the events associated to the Open Academy.

1. Add an **`end_date`** field computed from **`start_date`** and **`duration`**

   Tip

   the inverse function makes the field writable, and allows moving the sessions (via drag and drop) in the calendar view

2. Add a calendar view to the *Session* model
3. And add the calendar view to the *Session* model's actions

openacademy/models.py

```
# -*- coding: utf-8 -*-

from datetime import timedelta
from openerp import models, fields, api, exceptions

class Course(models.Model):
    attendee_ids = fields.Many2many('res.partner', string="Attendees")

    taken_seats = fields.Float(string="Taken seats", compute='_taken_seats')
    end_date = fields.Date(string="End Date", store=True,
        compute='_get_end_date', inverse='_set_end_date')

    @api.one
    @api.depends('seats', 'attendee_ids')
            }

    @api.one
    @api.depends('start_date', 'duration')
    def _get_end_date(self):
        if not (self.start_date and self.duration):
            self.end_date = self.start_date
            return

        # Add duration to start_date, but: Monday + 5 days = Saturday, so
        # subtract one second to get on Friday instead
        start = fields.Datetime.from_string(self.start_date)
        duration = timedelta(days=self.duration, seconds=-1)
        self.end_date = start + duration

    @api.one
    def _set_end_date(self):
        if not (self.start_date and self.end_date):
            return

        # Compute the difference between dates, but: Friday - Monday = 4 days,
        # so add one day to get 5 days instead
        start_date = fields.Datetime.from_string(self.start_date)
        end_date = fields.Datetime.from_string(self.end_date)
        self.duration = (end_date - start_date).days + 1
```

```
    @api.one
    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        if self.instructor_id and self.instructor_id in self.attendee_ids:
```

openacademy/views/openacademy.xml

```
            </field>
        </record>

        <!-- calendar view -->
        <record model="ir.ui.view" id="session_calendar_view">
            <field name="name">session.calendar</field>
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <calendar string="Session Calendar" date_start="start_date"
                          date_stop="end_date"
                          color="instructor_id">
                    <field name="name"/>
                </calendar>
            </field>
        </record>

        <record model="ir.actions.act_window" id="session_list_action">
            <field name="name">Sessions</field>
            <field name="res_model">openacademy.session</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form,calendar</field>
        </record>

        <menuitem id="session_menu" name="Sessions"
```

Search views

Search view `<field>` elements can have a `@filter_domain` that overrides the domain generated for searching on the given field. In the given domain, `self` represents the value entered by the user. In the example below, it is used to search on both fields `name`and `description`.

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

`domain`

add the given domain to the current search

`context`

add some context to the current search; use the key `group_by` to group results on the given field name

```
<search string="Ideas">
    <field name="name"/>
    <field name="description" string="Name and description"
```

```
            filter_domain="['|', ('name', 'ilike', self), ('descript
ion', 'ilike', self)]"/>
    <field name="inventor_id"/>
    <field name="country_id" widget="selection"/>

    <filter name="my_ideas" string="My Ideas"
            domain="[('inventor_id', '=', uid)]"/>
    <group string="Group By">
        <filter name="group_by_inventor" string="Inventor"
                context="{'group_by': 'inventor_id'}"/>
    </group>
</search>
```

To use a non-default search view in an action, it should be linked using the `search_view_id` field of the action record.

The action can also set default values for search fields through its `context` field: context keys of the form`search_default_`*`field_name`* will initialize *field_name* with the provided value. Search filters must have an optional `@name` to have a default and behave as booleans (they can only be enabled by default).

Exercise

Search views

1. Add a button to filter the courses for which the current user is the responsible in the course search view. Make it selected by default.
2. Add a button to group courses by responsible user.

openacademy/views/openacademy.xml

```
                <search>
                    <field name="name"/>
                    <field name="description"/>
                    <filter name="my_courses" string="My Courses"
                            domain="[('responsible_id', '=', uid)]"/>
                    <group string="Group By">
                        <filter name="by_responsible" string="Responsible"
                                context="{'group_by': 'responsible_id'}"/>
                    </group>
                </search>
            </field>
        </record>
            <field name="res_model">openacademy.course</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form</field>
            <field name="context" eval="{'search_default_my_courses': 1}"/>
            <field name="help" type="html">
                <p class="oe_view_nocontent_create">Create the first course
                </p>
```

Gantt

Horizontal bar charts typically used to show project planning and advancement, their root element is `<gantt>`.

```
<gantt string="Ideas" date_start="invent_date" color="inventor_id">
    <level object="idea.idea" link="id" domain="[]">
        <field name="inventor_id"/>
    </level>
</gantt>
```

Exercise

Gantt charts

Add a Gantt Chart enabling the user to view the sessions scheduling linked to the Open Academy module. The sessions should be grouped by instructor.

1. Create a computed field expressing the session's duration in hours
2. Add the gantt view's definition, and add the gantt view to the *Session* model's action

openacademy/models.py

```
    end_date = fields.Date(string="End Date", store=True,
        compute='_get_end_date', inverse='_set_end_date')

    hours = fields.Float(string="Duration in hours",
                         compute='_get_hours', inverse='_set_hours')

    @api.one
    @api.depends('seats', 'attendee_ids')
    def _taken_seats(self):
        self.duration = (end_date - start_date).days + 1

    @api.one
    @api.depends('duration')
    def _get_hours(self):
        self.hours = self.duration * 24

    @api.one
    def _set_hours(self):
        self.duration = self.hours / 24

    @api.one
    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        if self.instructor_id and self.instructor_id in self.attendee_ids:
```

openacademy/views/openacademy.xml

```
            </field>
        </record>

        <record model="ir.ui.view" id="session_gantt_view">
```

```
            <field name="name">session.gantt</field>
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <gantt string="Session Gantt" color="course_id"
                       date_start="start_date" date_delay="hours"
                       default_group_by='instructor_id'>
                    <field name="name"/>
                </gantt>
            </field>
        </record>

        <record model="ir.actions.act_window" id="session_list_action">
            <field name="name">Sessions</field>
            <field name="res_model">openacademy.session</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form,calendar,gantt</field>
        </record>

        <menuitem id="session_menu" name="Sessions"
```

Graph views

Graph views allow aggregated overview and analysis of models, their root element is `<graph>`.

Graph views have 4 display modes, the default mode is selected using the `@type` attribute.

Pivot
a multidimensional table, allows the selection of filers and dimensions to get the right aggregated dataset before moving to a more graphical overview
Bar (default)
a bar chart, the first dimension is used to define groups on the horizontal axis, other dimensions define aggregated bars within each group.
By default bars are side-by-side, they can be stacked by using `@stacked="True"` on the `<graph>`
Line
2-dimensional line chart
Pie
2-dimensional pie
Graph views contain `<field>` with a mandatory `@type` attribute taking the values:
`row` (default)
the field should be aggregated by default
`measure`
the field should be aggregated rather than grouped on

```
<graph string="Total idea score by Inventor">
    <field name="inventor_id"/>
    <field name="score" type="measure"/>
</graph>
```

Exercise

Graph view

Add a Graph view in the Session object that displays, for each course, the number of attendees under the form of a bar chart.

1. Add the number of attendees as a stored computed field
2. Then add the relevant view

openacademy/models.py

```
    hours = fields.Float(string="Duration in hours",
                         compute='_get_hours', inverse='_set_hours')

    attendees_count = fields.Integer(
        string="Attendees count", compute='_get_attendees_count', store=True)

    @api.one
    @api.depends('seats', 'attendee_ids')
    def _taken_seats(self):
        self.duration = self.hours / 24

    @api.one
    @api.depends('attendee_ids')
    def _get_attendees_count(self):
        self.attendees_count = len(self.attendee_ids)

    @api.one
    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        if self.instructor_id and self.instructor_id in self.attendee_ids:
```

openacademy/views/openacademy.xml

```
        </record>

        <record model="ir.ui.view" id="openacademy_session_graph_view">
            <field name="name">openacademy.session.graph</field>
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <graph string="Participations by Courses">
                    <field name="course_id"/>
                    <field name="attendees_count" type="measure"/>
                </graph>
            </field>
        </record>

        <record model="ir.actions.act_window" id="session_list_action">
            <field name="name">Sessions</field>
            <field name="res_model">openacademy.session</field>
            <field name="view_type">form</field>
            <field name="view_mode">tree,form,calendar,gantt,graph</field>
        </record>

        <menuitem id="session_menu" name="Sessions"
```

```
                  parent="openacademy_menu"
```

Kanban

Used to organize tasks, production processes, etc… their root element is `<kanban>`.

A kanban view shows a set of cards possibly grouped in columns. Each card represents a record, and each column the values of an aggregation field.

For instance, project tasks may be organized by stage (each column is a stage), or by responsible (each column is a user), and so on.

Kanban views define the structure of each card as a mix of form elements (including basic HTML) and **QWeb**.

Exercise

Kanban view

Add a Kanban view that displays sessions grouped by course (columns are thus courses).

1. Add an integer `color` field to the *Session* model
2. Add the kanban view and update the action

openacademy/models.py

```
    duration = fields.Float(digits=(6, 2), help="Duration in days")
    seats = fields.Integer(string="Number of seats")
    active = fields.Boolean(default=True)
    color = fields.Integer()

    instructor_id = fields.Many2one('res.partner', string="Instructor",
        domain=['|', ('instructor', '=', True),
```

openacademy/views/openacademy.xml

```
        </record>

        <record model="ir.ui.view" id="view_openacad_session_kanban">
            <field name="name">openacad.session.kanban</field>
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <kanban default_group_by="course_id">
                    <field name="color"/>
                    <templates>
                        <t t-name="kanban-box">
                            <div
                                    t-attf-class="oe_kanban_color_{{kanban_getcolo
r(record.color.raw_value)}}
                                                  oe_kanban_global_click_edit oe_s
emantic_html_override
                                                  oe_kanban_card {{record.group_fa
ncy==1 ? 'oe_kanban_card_fancy' : ''}}">
                                <div class="oe_dropdown_kanban">
```

```
                                <!-- dropdown menu -->
                                <div class="oe_dropdown_toggle">
                                    <span class="oe_e">#</span>
                                    <ul class="oe_dropdown_menu">
                                        <li>
                                            <a type="delete">Delete</a>
                                        </li>
                                        <li>
                                            <ul class="oe_kanban_colorpicker"
                                                data-field="color"/>
                                        </li>
                                    </ul>
                                </div>
                                <div class="oe_clear"></div>
                            </div>
                            <div t-attf-class="oe_kanban_content">
                                <!-- title -->
                                Session name:
                                <field name="name"/>
                                <br/>
                                Start date:
                                <field name="start_date"/>
                                <br/>
                                duration:
                                <field name="duration"/>
                            </div>
                        </div>
                    </t>
                </templates>
            </kanban>
        </field>
    </record>

    <record model="ir.actions.act_window" id="session_list_action">
        <field name="name">Sessions</field>
        <field name="res_model">openacademy.session</field>
        <field name="view_type">form</field>
        <field name="view_mode">tree,form,calendar,gantt,graph,kanban</field>
    </record>

    <menuitem id="session_menu" name="Sessions"
              parent="openacademy_menu"
```

Workflows

Workflows are models associated to business objects describing their dynamics. Workflows are also used to track processes that evolve over time.

Exercise

Almost a workflow

Add a **state** field to the *Session* model. It will be used to define a workflow-ish.

A sesion can have three possible states: Draft (default), Confirmed and Done.

In the session form, add a (read-only) field to visualize the state, and buttons to change it. The valid transitions are:

- Draft -> Confirmed
- Confirmed -> Draft
- Confirmed -> Done
- Done -> Draft

1. Add a new **state** field
2. Add state-transitioning methods, those can be called from view buttons to change the record's state
3. And add the relevant buttons to the session's form view

openacademy/models.py

```
    attendees_count = fields.Integer(
        string="Attendees count", compute='_get_attendees_count', store=True)

    state = fields.Selection([
         ('draft', "Draft"),
         ('confirmed', "Confirmed"),
         ('done', "Done"),
    ], default='draft')

    @api.one
    def action_draft(self):
        self.state = 'draft'

    @api.one
    def action_confirm(self):
        self.state = 'confirmed'

    @api.one
    def action_done(self):
        self.state = 'done'

    @api.one
    @api.depends('seats', 'attendee_ids')
    def _taken_seats(self):
```

openacademy/views/openacademy.xml

```
            <field name="model">openacademy.session</field>
            <field name="arch" type="xml">
                <form string="Session Form">
                    <header>
                        <button name="action_draft" type="object"
                                string="Reset to draft"
```

```
                                states="confirmed,done"/>
                        <button name="action_confirm" type="object"
                                string="Confirm" states="draft"
                                class="oe_highlight"/>
                        <button name="action_done" type="object"
                                string="Mark as done" states="confirmed"
                                class="oe_highlight"/>
                        <field name="state" widget="statusbar"/>
                    </header>

                    <sheet>
                        <group>
                            <group string="General">
```

Workflows may be associated with any object in Odoo, and are entirely customizable. Workflows are used to structure and manage the lifecycles of business objects and documents, and define transitions, triggers, etc. with graphical tools. Workflows, activities (nodes or actions) and transitions (conditions) are declared as XML records, as usual. The tokens that navigate in workflows are called workitems.

Exercise

Workflow

Replace the ad-hoc *Session* workflow by a real workflow. Transform the *Session* form view so its buttons call the workflow instead of the model's methods.

openacademy/__openerp__.py

```
        'templates.xml',
        'views/openacademy.xml',
        'views/partner.xml',
        'views/session_workflow.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
```

openacademy/models.py

```
        ('draft', "Draft"),
        ('confirmed', "Confirmed"),
        ('done', "Done"),
    ])

    @api.one
    def action_draft(self):
```

openacademy/views/openacademy.xml

```
            <field name="arch" type="xml">
                <form string="Session Form">
                    <header>
                        <button name="draft" type="workflow"
                                string="Reset to draft"
                                states="confirmed,done"/>
```

```
                        <button name="confirm" type="workflow"
                                string="Confirm" states="draft"
                                class="oe_highlight"/>
                        <button name="done" type="workflow"
                                string="Mark as done" states="confirmed"
                                class="oe_highlight"/>
                        <field name="state" widget="statusbar"/>
```

openacademy/views/session_workflow.xml

```
<openerp>
    <data>
        <record model="workflow" id="wkf_session">
            <field name="name">OpenAcademy sessions workflow</field>
            <field name="osv">openacademy.session</field>
            <field name="on_create">True</field>
        </record>

        <record model="workflow.activity" id="draft">
            <field name="name">Draft</field>
            <field name="wkf_id" ref="wkf_session"/>
            <field name="flow_start" eval="True"/>
            <field name="kind">function</field>
            <field name="action">action_draft()</field>
        </record>
        <record model="workflow.activity" id="confirmed">
            <field name="name">Confirmed</field>
            <field name="wkf_id" ref="wkf_session"/>
            <field name="kind">function</field>
            <field name="action">action_confirm()</field>
        </record>
        <record model="workflow.activity" id="done">
            <field name="name">Done</field>
            <field name="wkf_id" ref="wkf_session"/>
            <field name="kind">function</field>
            <field name="action">action_done()</field>
        </record>

        <record model="workflow.transition" id="session_draft_to_confirmed">
            <field name="act_from" ref="draft"/>
            <field name="act_to" ref="confirmed"/>
            <field name="signal">confirm</field>
        </record>
        <record model="workflow.transition" id="session_confirmed_to_draft">
            <field name="act_from" ref="confirmed"/>
            <field name="act_to" ref="draft"/>
            <field name="signal">draft</field>
        </record>
        <record model="workflow.transition" id="session_done_to_draft">
            <field name="act_from" ref="done"/>
            <field name="act_to" ref="draft"/>
            <field name="signal">draft</field>
        </record>
        <record model="workflow.transition" id="session_confirmed_to_done">
            <field name="act_from" ref="confirmed"/>
            <field name="act_to" ref="done"/>
            <field name="signal">done</field>
        </record>
    </data>
</openerp>
```

Exercise

Automatic transitions

Automatically transition sessions from *Draft* to *Confirmed* when more than half the session's seats are reserved.

openacademy/views/session_workflow.xml

```
            <field name="act_to" ref="done"/>
            <field name="signal">done</field>
        </record>

        <record model="workflow.transition" id="session_auto_confirm_half_filled">
            <field name="act_from" ref="draft"/>
            <field name="act_to" ref="confirmed"/>
            <field name="condition">taken_seats &gt; 50</field>
        </record>
    </data>
</openerp>
```

Exercise

Server actions

Replace the Python methods for synchronizing session state by server actions.

Both the workflow and the server actions could have been created entirely from the UI.

openacademy/views/session_workflow.xml

```
            <field name="on_create">True</field>
        </record>

        <record model="ir.actions.server" id="set_session_to_draft">
            <field name="name">Set session to Draft</field>
            <field name="model_id" ref="model_openacademy_session"/>
            <field name="code">
model.search([('id', 'in', context['active_ids'])]).action_draft()
            </field>
        </record>
        <record model="workflow.activity" id="draft">
            <field name="name">Draft</field>
            <field name="wkf_id" ref="wkf_session"/>
            <field name="flow_start" eval="True"/>
            <field name="kind">dummy</field>
            <field name="action"></field>
            <field name="action_id" ref="set_session_to_draft"/>
        </record>

        <record model="ir.actions.server" id="set_session_to_confirmed">
            <field name="name">Set session to Confirmed</field>
            <field name="model_id" ref="model_openacademy_session"/>
            <field name="code">
model.search([('id', 'in', context['active_ids'])]).action_confirm()
            </field>
        </record>
        <record model="workflow.activity" id="confirmed">
            <field name="name">Confirmed</field>
            <field name="wkf_id" ref="wkf_session"/>
            <field name="kind">dummy</field>
            <field name="action"></field>
```

```
            <field name="action_id" ref="set_session_to_confirmed"/>
        </record>

        <record model="ir.actions.server" id="set_session_to_done">
            <field name="name">Set session to Done</field>
            <field name="model_id" ref="model_openacademy_session"/>
            <field name="code">
model.search([('id', 'in', context['active_ids'])]).action_done()
            </field>
        </record>
        <record model="workflow.activity" id="done">
            <field name="name">Done</field>
            <field name="wkf_id" ref="wkf_session"/>
            <field name="kind">dummy</field>
            <field name="action"></field>
            <field name="action_id" ref="set_session_to_done"/>
        </record>

        <record model="workflow.transition" id="session_draft_to_confirmed">
```