Warning

This tutorial requires **having installed Odoo (../setup/install.html#setup-install)**

Start/Stop the Odoo server

Odoo uses a client/server architecture in which clients are web browsers accessing the Odoo server via RPC.
Business logic and extension is generally performed on the server side, although supporting client features (e.g. new data representation such as interactive maps) can be added to the client.
In order to start the server, simply invoke the command odoo-bin (../reference/cmdline.html#reference-cmdline) in the shell, adding the full path to the file if necessary:

```
odoo-bin
```

The server is stopped by hitting `Ctrl-C` twice from the terminal, or by killing the corresponding OS process.

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` (../reference/cmdline.html#cmdoption-odoo-bin--addons-path) option.

Tip

most command-line options can also be set using **a configuration file (../reference/cmdline.html#reference-cmdline-config)**

An Odoo module is declared by its manifest (../reference/module.html#reference-module-manifest). See the manifest documentation (../reference/module.html#reference-module-manifest) about it.
A module is also a Python package (http://docs.python.org/2/tutorial/modules.html#packages) 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-bin (../reference /cmdline.html#reference-cmdline-server) has a subcommand scaffold (../reference /cmdline.html#reference-cmdline-scaffold) to create an empty module:

```
$ odoo-bin 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-bin scaffold openacademy addons`.
2. Adapt the manifest file to your module.
3. Don't bother about the other files.

openacademy/__manifest__.py

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

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

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

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

    # Categories can be used to filter modules in modules listing
    # Check https://github.com/odoo/odoo/blob/master/odoo/addons/base/mc
    # 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 odoo import http

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

Object-Relational Mapping

A key component of Odoo is the ORM (Object-Relational Mapping) layer. This layer avoids having to write most SQL (Structured Query Language) by hand and provides extensibility and security services2.

Business objects are declared as Python classes extending `Model` (../reference/orm.html#odoo.models.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` (../reference/orm.html#odoo.models.Model._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 odoo 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 odoo 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 (http://use-the-index-luke.com/sql/preface) 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` (../reference/orm.html#odoo.fields.Boolean), `Date` (../reference/orm.html#odoo.fields.Date), `Char` (../reference/orm.html#odoo.fields.Char).

Reserved fields

Odoo creates a few fields in all models1. 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` (../reference/orm.html#odoo.fields.Datetime))
: Creation date of the record.

`create_uid` (`Many2one` (../reference/orm.html#odoo.fields.Many2one))
: User who created the record.

`write_date` (`Datetime` (../reference/orm.html#odoo.fields.Datetime))
: Last modification date of the record.

`write_uid` (`Many2one` (../reference/orm.html#odoo.fields.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` (../reference /orm.html#odoo.models.Model._rec_name).

Exercise

Define a model

Define a new data model *Course* in the *openacademy* module. A course has a title and a description. Courses must have a title.

Edit the file `openacademy/models/models.py` to include a *Course* class.

openacademy/models.py

```
from odoo import models, fields, api

class Course(models.Model):
    _name = 'openacademy.course'

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

Data files

Odoo is a highly data driven system. Although behavior is customized using Python (http://python.org) code part of a module's value is in the data it sets up when loaded.

Tip

some modules exist solely to add data into Odoo

Module data is declared via data files (../reference/data.html#reference-data), XML files with `<record>` elements. Each `<record>` element creates or updates a database record.

```
<odoo>
    <data>
        <record model="{model name}" id="{record identifier}">
            <field name="{a field name}">{a value}</field>
        </record>
    </data>
</odoo>
```

- `model` is the name of the Odoo model for the record.
- `id` is an external identifier (../glossary.html#term-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/demo.xml` to include some data.

openacademy/demo.xml

```
<odoo>
    <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>
</odoo>
```

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"/>
```

Danger

The action must be declared before its corresponding menu in the XML file.

Data files are executed sequentially, the action's `id` must be present in the database before the menu can be created.

Exercise

Define new menu entries

Define new menu entries to access courses 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/__manifest__.py`

openacademy/__manifest__.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"?>
<odoo>
    <data>
        <!-- window action -->
        <!--
            The following tag is an action definition for a "window acti
            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 cou
                </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_m
                  action="course_list_action"/>
        <!-- Full id location:
             action="openacademy.course_list_action"
             It is not required when it is the same module -->
    </data>
</odoo>
```

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 (../reference/views.html#reference-views-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>
```

Danger

The view's content is XML.

The `arch` field must thus be declared as `type="xml"` to be parsed correctly.

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 are 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"?>
<odoo>
    <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 acti
```

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_done"
                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 Name" />
            <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:

```
<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

```
            </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 acti
```

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/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>
```

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')` **(../reference/orm.html#odoo.fields.Many2one)**

A simple link to an other object:

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

See also

foreign keys (http://www.postgresql.org/docs/9.3/static/tutorial-fk.html)

`One2many(other_model, related_field)` **(../reference/orm.html#odoo.fields.One2many)**

A virtual relationship, inverse of a `Many2one` (../reference/orm.html#odoo.fields.Many2one). A `One2many` (../reference/orm.html#odoo.fields.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
```

Danger

Because a `One2many` **(../reference/orm.html#odoo.fields.One2many)** is a virtual relationship, there *must* be a `Many2one` **(../reference /orm.html#odoo.fields.Many2one)** field in the *`other_model`*, and its name *must* be *`related_field`*

`Many2many(other_model)` **(../reference/orm.html#odoo.fields.Many2many)**

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>
```

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

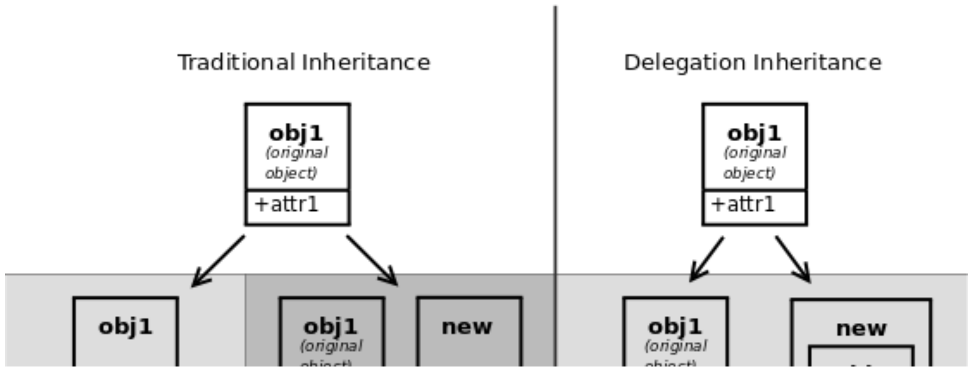
Model inheritance

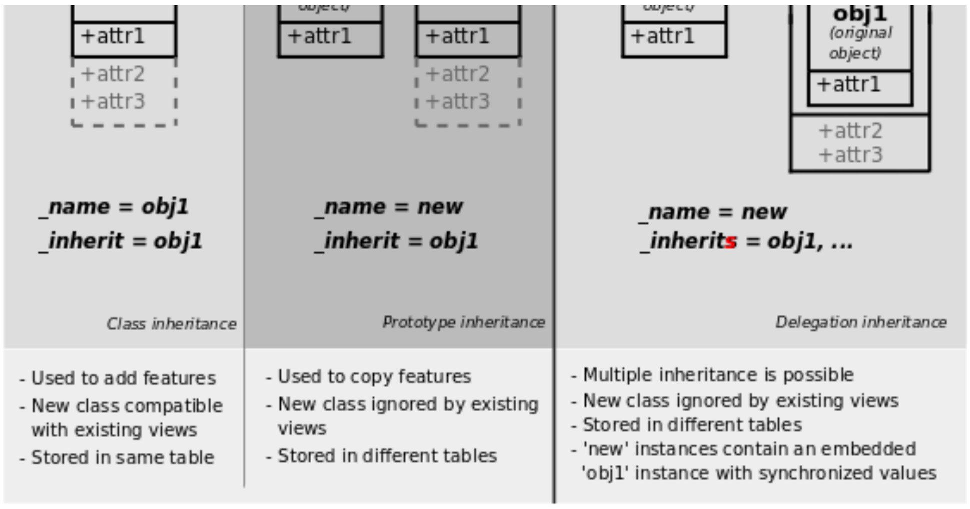
Odoo provides two *inheritance* mechanisms to extend an existing model in a modular way.

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.





See also

- `_inherit` (../reference/orm.html#odoo.models.Model._inherit)
- `_inherits` (../reference/orm.html#odoo.models.Model._inherits)

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 (http://w3.org/TR/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 with the `xpath`'s body, replacing any `$0` node occurrence in the new body with the original element

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

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

`attributes`
: alters the attributes of the matched element using 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/models/partner.py` and import it in `__init__.py`
2. Create a file `openacademy/views/partner.xml` and add it to `__manifest__.py`

openacademy/__init__.py

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

openacademy/__manifest__.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 odoo import fields, models

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

    # Add a new column to the res.partner model, by default partners are
    # 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"?>
 <odoo>
    <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"/>
```

Domains

In Odoo, Domains (../reference/orm.html#reference-orm-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_actic
            <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>
</odoo>
```

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`.

Danger

`self` is a collection

The object `self` is a *recordset*, i.e., an ordered collection of records. It supports the standard Python operations on collections, like `len(self)` and `iter(self)`, plus extra set operations like `recs1 + recs2`.

Iterating over `self` gives the records one by one, where each record is itself a collection of size 1. You can access/assign fields on single records by using the dot notation, like `record.name`.

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

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))
```

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()` (../reference/orm.html#odoo.api.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 odoo import models, fields, api

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

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

    @api.depends('value')
    def _compute_name(self):
        for record in self:
            record.name = "Record with value %s" % record.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_sea

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

openacademy/views/openacademy.xml

```
                                <field name="start_date"/>
                                <field name="duration"/>
                                <field name="seats"/>
                                <field name="taken_seats" widget="progre
                            </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.env.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` **(../reference/orm.html#odoo.fields.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()` (../reference/orm.html#odoo.api.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

```
                r.taken_seats = 0.0
            else:
                r.taken_seats = 100.0 * len(r.attendee_ids) / r.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
                },
            }
        if self.seats < len(self.attendee_ids):
            return {
                'warning': {
                    'title': "Too many attendees",
                    'message': "Increase seats or remove excess attendee
                },
            }
```

Model constraints

Odoo provides two ways to set up automatically verified invariants: `Python constraints` (../reference/orm.html#odoo.api.constrains) and `SQL constraints` (../reference /orm.html#odoo.models.Model._sql_constraints).
A Python constraint is defined as a method decorated with `constrains()` (../reference /orm.html#odoo.api.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 odoo.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" % record.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 odoo import models, fields, api, exceptions

class Course(models.Model):
    _name = 'openacademy.course'


                    'message': "Increase seats or remove excess attendee
                },
            }

    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        for r in self:
            if r.instructor_id and r.instructor_id in r.attendee_ids:
                raise exceptions.ValidationError("A session's instructor
```

SQL constraints are defined through the model attribute `_sql_constraints` (../reference /orm.html#odoo.models.Model._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_constraint (http://www.postgresql.org/docs/9.3/static/ddl-constraints.html) expression, and `message` is the error message.

Exercise

Add SQL constraints

With the help of **PostgreSQL's documentation (http://www.postgresql.org/docs/9.3/static/ddl-constraints.html)** , 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.multi
    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:

`decoration-{$name}`

allow changing the style of a row's text based on the corresponding record's attributes. Values are Python expressions. For each record, the expression is evaluated with the record's attributes as context values and if `true` , the corresponding style is applied to the row. Other context values are `uid` (the id of the current user) and `current_date` (the current date as a string of the form `yyyy-MM-dd`).

`{$name}` can be `bf` (`font-weight: bold`), `it` (`font-style: italic`), or any bootstrap contextual color (http://getbootstrap.com/components/#available-variations) (`danger` , `info` , `muted` , `primary` , `success` or `warning`).

```
<tree string="Idea Categories" decoration-info="state=='draft'"
    decoration-danger="state=='trashed'">
    <field name="name"/>
    <field name="state"/>
</tree>
```

`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" decoration-info="duration&lt
                    <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_id">
    <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 odoo 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_sea
    end_date = fields.Date(string="End Date", store=True,
        compute='_get_end_date', inverse='_set_end_date')

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


                },
            }

    @api.depends('start_date', 'duration')
    def _get_end_date(self):
        for r in self:
            if not (r.start_date and r.duration):
                r.end_date = r.start_date
                continue

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

    def _set_end_date(self):
        for r in self:
            if not (r.start_date and r.end_date):
                continue

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

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), ('description', '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="Responsibl
                                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 cou
                </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"
       date_stop="date_finished"
       progress="progress"
       default_group_by="inventor_id" />
```

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.depends('seats', 'attendee_ids')
    def _taken_seats(self):
        for r in self:


            end_date = fields.Datetime.from_string(r.end_date)
            r.duration = (end_date - start_date).days + 1

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

    def _set_hours(self):
        for r in self:
            r.duration = r.hours / 24

    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        for r in self:
```

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>
```

Graph views

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

Note

Pivot views (element `<pivot>`) a multidimensional table, allows the selection of filers and dimensions to get the right aggregated dataset before moving to a more graphical overview. The pivot view shares the same content definition as graph views.

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

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>
```

Warning

Graph views perform aggregations on database values, they do not work with non-stored computed fields.

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=

    @api.depends('seats', 'attendee_ids')
    def _taken_seats(self):
        for r in self:


        for r in self:
            r.duration = r.hours / 24

    @api.depends('attendee_ids')
    def _get_attendees_count(self):
        for r in self:
            r.attendees_count = len(r.attendee_ids)

    @api.constrains('instructor_id', 'attendee_ids')
    def _check_instructor_not_in_attendees(self):
        for r in self:
```

openacademy/views/openacademy.xml

```
            </field>
        </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</fiel
        </record>

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

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 (../reference/qweb.html#reference-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_{{kanb
                                                  oe_kanban_global_click
                                                  oe_kanban_card {{recor
                                <div class="oe_dropdown_kanban">
                                    <!-- dropdown menu -->
                                    <div class="oe_dropdown_toggle">
                                        <i class="fa fa-bars fa-lg"/>
                                        <ul class="oe_dropdown_menu">
                                            <li>
                                                <a type="delete">Delete<
                                            </li>
                                            <li>
                                                <ul class="oe_kanban_col
                                                    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>
```

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=

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

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

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

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

    @api.depends('seats', 'attendee_ids')
    def _taken_seats(self):
        for r in 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"/>
```

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.

Warning

A workflow associated with a model is only created when the model's records are created. Thus there is no workflow instance associated with session instances created before the workflow's definition

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/__manifest__.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.multi
    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

```
<odoo>
    <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"/>
```

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_hal
            <field name="act_from" ref="draft"/>
            <field name="act_to" ref="confirmed"/>
            <field name="condition">taken_seats &gt; 50</field>
        </record>
    </data>
</odoo>
```

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_confirm
```

Security

Access control mechanisms must be configured to achieve a coherent security policy.

Group-based access control mechanisms

Groups are created as normal records on the model `res.groups` , and granted menu access via menu definitions. However even without a menu, objects may still be accessible indirectly, so actual object-level permissions (read, write, create, unlink) must be defined for groups. They are usually inserted via CSV files inside modules. It is also possible to restrict access to specific fields on a view or object using the field's groups attribute.

Access rights

Access rights are defined as records of the model `ir.model.access` . Each access right is associated to a model, a group (or no group for global access), and a set of permissions: read, write, create, unlink. Such access rights are usually created by a CSV file named after its model: `ir.model.access.csv` .

```
id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_unlink
access_idea_idea,idea.idea,model_idea_idea,base.group_user,1,1,1,0
access_idea_vote,idea.vote,model_idea_vote,base.group_user,1,1,1,0
```

Exercise

Add access control through the Odoo interface

Create a new user "John Smith". Then create a group "OpenAcademy / Session Read" with read access to the *Session* model.

1. Create a new user *John Smith* through **Settings ▸ Users ▸ Users**
2. Create a new group `session_read` through **Settings ▸ Users ▸ Groups**, it should have read access on the *Session* model
3. Edit *John Smith* to make them a member of `session_read`
4. Log in as *John Smith* to check the access rights are correct

Exercise

Add access control through data files in your module

Using data files,

- Create a group *OpenAcademy / Manager* with full access to all OpenAcademy models
- Make *Session* and *Course* readable by all users

1. Create a new file `openacademy/security/security.xml` to hold the OpenAcademy Manager group
2. Edit the file `openacademy/security/ir.model.access.csv` with the access rights to the models
3. Finally update `openacademy/__manifest__.py` to add the new data files to it

openacademy/__manifest__.py

```
    # always loaded
    'data': [
        'security/security.xml',
        'security/ir.model.access.csv',
        'templates.xml',
        'views/openacademy.xml',
        'views/partner.xml',
```

openacademy/security/ir.model.access.csv

```
id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_ur
course_manager,course manager,model_openacademy_course,group_manager,1,1
session_manager,session manager,model_openacademy_session,group_manager,
course_read_all,course all,model_openacademy_course,,1,0,0,0
session_read_all,session all,model_openacademy_session,,1,0,0,0
```

openacademy/security/security.xml

```
<odoo>
    <data>
        <record id="group_manager" model="res.groups">
            <field name="name">OpenAcademy / Manager</field>
        </record>
    </data>
</odoo>
```

Record rules

A record rule restricts the access rights to a subset of records of the given model. A rule is a record of the model `ir.rule`, and is associated to a model, a number of groups (many2many field), permissions to which the restriction applies, and a domain. The domain specifies to which records the access rights are limited.
Here is an example of a rule that prevents the deletion of leads that are not in state `cancel`. Notice that the value of the field `groups` must follow the same convention as the method `write()` (../reference/orm.html#odoo.models.Model.write) of the ORM.

```
<record id="delete_cancelled_only" model="ir.rule">
    <field name="name">Only cancelled leads may be deleted</field>
    <field name="model_id" ref="crm.model_crm_lead"/>
    <field name="groups" eval="[(4, ref('sales_team.group_sale_manager'))]"/>
    <field name="perm_read" eval="0"/>
    <field name="perm_write" eval="0"/>
    <field name="perm_create" eval="0"/>
    <field name="perm_unlink" eval="1" />
    <field name="domain_force">[('state','=','cancel')]</field>
</record>
```

Exercise

Record rule

Add a record rule for the model Course and the group "OpenAcademy / Manager", that restricts `write` and `unlink` accesses to the responsible of a course. If a course has no responsible, all users of the group must be able to modify it.

Create a new rule in `openacademy/security/security.xml` :

openacademy/security/security.xml

```
        <record id="group_manager" model="res.groups">
            <field name="name">OpenAcademy / Manager</field>
        </record>

        <record id="only_responsible_can_modify" model="ir.rule">
            <field name="name">Only Responsible can modify Course</fielc
            <field name="model_id" ref="model_openacademy_course"/>
            <field name="groups" eval="[(4, ref('openacademy.group_manac
            <field name="perm_read" eval="0"/>
            <field name="perm_write" eval="1"/>
            <field name="perm_create" eval="0"/>
            <field name="perm_unlink" eval="1"/>
            <field name="domain_force">
                ['|', ('responsible_id','=',False),
                      ('responsible_id','=',user.id)]
            </field>
        </record>
    </data>
</odoo>
```

Wizards

Wizards describe interactive sessions with the user (or dialog boxes) through dynamic forms. A wizard is simply a model that extends the class `TransientModel` instead of `Model` (../reference /orm.html#odoo.models.Model). The class `TransientModel` extends `Model` (../reference /orm.html#odoo.models.Model) and reuse all its existing mechanisms, with the following particularities:

- Wizard records are not meant to be persistent; they are automatically deleted from the database after a certain time. This is why they are called *transient*.
- Wizard models do not require explicit access rights: users have all permissions on wizard records.
- Wizard records may refer to regular records or wizard records through many2one fields, but regular records *cannot* refer to wizard records through a many2one field.

We want to create a wizard that allow users to create attendees for a particular session, or for a list of sessions at once.

Exercise

Define the wizard

Create a wizard model with a many2one relationship with the *Session* model and a many2many relationship with the *Partner* model.

Add a new file `openacademy/wizard.py` :

openacademy/__init__.py

```
from . import controllers
from . import models
from . import partner
from . import wizard
```

openacademy/wizard.py

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

from odoo import models, fields, api

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

    session_id = fields.Many2one('openacademy.session',
        string="Session", required=True)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")
```

Launching wizards

Wizards are launched by `ir.actions.act_window` records, with the field `target` set to the value `new` . The latter opens the wizard view into a popup window. The action may be triggered by a menu item.
There is another way to launch the wizard: using an `ir.actions.act_window` record like above, but with an extra field `src_model` that specifies in the context of which model the action is available. The wizard will appear in the contextual actions of the model, above the main view. Because of some internal hooks in the ORM, such an action is declared in XML with the tag `act_window` .

```
<act_window id="launch_the_wizard"
            name="Launch the Wizard"
            src_model="context.model.name"
            res_model="wizard.model.name"
            view_mode="form"
            target="new"
            key2="client_action_multi"/>
```

Wizards use regular views and their buttons may use the attribute `special="cancel"` to close the wizard window without saving.

Exercise

Launch the wizard

1. Define a form view for the wizard.
2. Add the action to launch it in the context of the *Session* model.
3. Define a default value for the session field in the wizard; use the context parameter `self._context` to retrieve the current session.

openacademy/wizard.py

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

    def _default_session(self):
        return self.env['openacademy.session'].browse(self._context.get(

    session_id = fields.Many2one('openacademy.session',
        string="Session", required=True, default=_default_session)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")
```

openacademy/views/openacademy.xml

```
                  parent="openacademy_menu"
                  action="session_list_action"/>

        <record model="ir.ui.view" id="wizard_form_view">
            <field name="name">wizard.form</field>
            <field name="model">openacademy.wizard</field>
            <field name="arch" type="xml">
                <form string="Add Attendees">
                    <group>
                        <field name="session_id"/>
                        <field name="attendee_ids"/>
                    </group>
                </form>
            </field>
        </record>

        <act_window id="launch_session_wizard"
                    name="Add Attendees"
                    src_model="openacademy.session"
                    res_model="openacademy.wizard"
                    view_mode="form"
                    target="new"
                    key2="client_action_multi"/>
    </data>
</odoo>
```

Exercise

Register attendees

Add buttons to the wizard, and implement the corresponding method for adding the attendees to the given session.

openacademy/views/openacademy.xml

```
                        <field name="attendee_ids"/>
                    </group>
                    <footer>
                        <button name="subscribe" type="object"
                                string="Subscribe" class="oe_highlight"/
                        or
                        <button special="cancel" string="Cancel"/>
                    </footer>
                </form>
            </field>
        </record>
```

openacademy/wizard.py

```
    session_id = fields.Many2one('openacademy.session',
        string="Session", required=True, default=_default_session)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")

    @api.multi
    def subscribe(self):
        self.session_id.attendee_ids |= self.attendee_ids
        return {}
```

Exercise

Register attendees to multiple sessions

Modify the wizard model so that attendees can be registered to multiple sessions.

openacademy/views/openacademy.xml

```
                <form string="Add Attendees">
                    <group>
                        <field name="session_ids"/>
                        <field name="attendee_ids"/>
                    </group>
                    <footer>
                        <button name="subscribe" type="object"
```

openacademy/wizard.py

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

    def _default_sessions(self):
        return self.env['openacademy.session'].browse(self._context.get(

    session_ids = fields.Many2many('openacademy.session',
        string="Sessions", required=True, default=_default_sessions)
    attendee_ids = fields.Many2many('res.partner', string="Attendees")

    @api.multi
    def subscribe(self):
        for session in self.session_ids:
            session.attendee_ids |= self.attendee_ids
        return {}
```

Internationalization

Each module can provide its own translations within the i18n directory, by having files named LANG.po where LANG is the locale code for the language, or the language and country combination when they differ (e.g. pt.po or pt_BR.po). Translations will be loaded automatically by Odoo for all enabled languages. Developers always use English when creating a module, then export the module terms using Odoo's gettext POT export feature (**Settings ‣ Translations ‣ Import/Export ‣ Export Translation** without specifying a language), to create the module template POT file, and then derive the translated PO files. Many IDE's have plugins or modes for editing and merging PO/POT files.

Tip

The Portable Object files generated by Odoo are published on **Transifex (https://www.transifex.com/odoo/public/)**, making it easy to translate the software.

```
|- idea/ # The module directory
   |- i18n/ # Translation files
      | - idea.pot # Translation Template (exported from Odoo)
      | - fr.po # French translation
      | - pt_BR.po # Brazilian Portuguese translation
      | (...)
```

Tip

By default Odoo's POT export only extracts labels inside XML files or inside field definitions in Python code, but any Python string can be translated this way by surrounding it with the function `odoo._()` (e.g. `_("Label")`)

Exercise

Translate a module

Choose a second language for your Odoo installation. Translate your module using the facilities provided by Odoo.

1. Create a directory `openacademy/i18n/`
2. Install whichever language you want (**Administration** ▸ **Translations** ▸ **Load an Official Translation**)
3. Synchronize translatable terms (**Administration** ▸ **Translations** ▸ **Application Terms** ▸ **Synchronize Translations**)
4. Create a template translation file by exporting (**Administration** ▸ **Translations -> Import/Export** ▸ **Export Translation**) without specifying a language, save in `openacademy/i18n/`
5. Create a translation file by exporting (**Administration** ▸ **Translations** ▸ **Import/Export** ▸ **Export Translation**) and specifying a language. Save it in `openacademy/i18n/`
6. Open the exported translation file (with a basic text editor or a dedicated PO-file editor e.g. **POEdit (http://poedit.net)** and translate the missing terms
7. In `models.py` , add an import statement for the function `odoo._` and mark missing strings as translatable
8. Repeat steps 3-6

openacademy/models.py

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

from datetime import timedelta
from odoo import models, fields, api, exceptions, _

class Course(models.Model):
    _name = 'openacademy.course'


        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_cc

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


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

Reporting

Printed reports

Odoo 8.0 comes with a new report engine based on QWeb (../reference/qweb.html#reference-qweb), Twitter Bootstrap (http://getbootstrap.com) and Wkhtmltopdf (http://wkhtmltopdf.org).
A report is a combination two elements:

- an `ir.actions.report.xml` , for which a `<report>` shortcut element is provided, it sets up various basic parameters for the report (default type, whether the report should be saved to the database after generation,…)

```
<report
    id="account_invoices"
    model="account.invoice"
    string="Invoices"
    report_type="qweb-pdf"
    name="account.report_invoice"
    file="account.report_invoice"
    attachment_use="True"
    attachment="(object.state in ('open','paid')) and
        ('INV'+(object.number or '').replace('/','')+'.pdf')"
/>
```

- A standard QWeb view (../reference/views.html#reference-views-qweb) for the actual report:

```
<t t-call="report.html_container">
    <t t-foreach="docs" t-as="o">
        <t t-call="report.external_layout">
            <div class="page">
                <h2>Report title</h2>
            </div>
        </t>
    </t>
</t>

the standard rendering context provides a number of elements, the most
important being:

``docs``
    the records for which the report is printed
``user``
    the user printing the report
```

Because reports are standard web pages, they are available through a URL and output parameters can be manipulated through this URL, for instance the HTML version of the *Invoice* report is available through http://localhost:8069/report/html/account.report_invoice/1 (http://localhost:8069/report/html/account.report_invoice/1) (if `account` is installed) and the PDF version through http://localhost:8069/report/pdf/account.report_invoice/1 (http://localhost:8069/report/pdf/account.report_invoice/1).

Danger

If it appears that your PDF report is missing the styles (i.e. the text appears but the style/layout is different from the html version), probably your **wkhtmltopdf (http://wkhtmltopdf.org)** process cannot reach your web server to download them.

If you check your server logs and see that the CSS styles are not being downloaded when generating a PDF report, most surely this is the problem.

The **wkhtmltopdf (http://wkhtmltopdf.org)** process will use the `web.base.url` system parameter as the *root path* to all linked files, but this parameter is automatically updated each time the Administrator is logged in. If your server resides behind some kind of proxy, that could not be reachable. You can fix this by adding one of these system parameters:

- `report.url` , pointing to an URL reachable from your server (probably `http://localhost:8069` or something similar). It will be used for this particular purpose only.
- `web.base.url.freeze` , when set to `True` , will stop the automatic updates to `web.base.url` .

Exercise

Create a report for the Session model

For each session, it should display session's name, its start and end, and list the session's attendees.

openacademy/__manifest__.py

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

openacademy/reports.xml

```
<odoo>
<data>
    <report
        id="report_session"
        model="openacademy.session"
        string="Session Report"
        name="openacademy.report_session_view"
        file="openacademy.report_session"
        report_type="qweb-pdf" />

    <template id="report_session_view">
        <t t-call="report.html_container">
            <t t-foreach="docs" t-as="doc">
                <t t-call="report.external_layout">
                    <div class="page">
                        <h2 t-field="doc.name"/>
                        <p>From <span t-field="doc.start_date"/> to <spa
                        <h3>Attendees:</h3>
                        <ul>
                            <t t-foreach="doc.attendee_ids" t-as="attenc
                                <li><span t-field="attendee.name"/></li>
                            </t>
                        </ul>
                    </div>
                </t>
            </t>
        </t>
    </template>
</data>
</odoo>
```

Dashboards

Exercise

Define a Dashboard

Define a dashboard containing the graph view you created, the sessions calendar view and a list view of the courses (switchable to a form view). This dashboard should be available through a menuitem in the menu, and automatically displayed in the web client when the OpenAcademy main menu is selected.

1. Create a file `openacademy/views/session_board.xml`. It should contain the board view, the actions referenced in that view, an action to open the dashboard and a re-definition of the main menu item to add the dashboard action

   Note

   Available dashboard styles are `1`, `1-1`, `1-2`, `2-1` and `1-1-1`

2. Update `openacademy/__manifest__.py` to reference the new data file

openacademy/__manifest__.py

```
    'version': '0.1',

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

    # always loaded
    'data': [


        'views/openacademy.xml',
        'views/partner.xml',
        'views/session_workflow.xml',
        'views/session_board.xml',
        'reports.xml',
    ],
    # only loaded in demonstration mode
```

openacademy/views/session_board.xml

```
<?xml version="1.0"?>
<odoo>
    <data>
        <record model="ir.actions.act_window" id="act_session_graph">
            <field name="name">Attendees by course</field>
            <field name="res_model">openacademy.session</field>
            <field name="view_type">form</field>
            <field name="view_mode">graph</field>
            <field name="view_id"
                   ref="openacademy.openacademy_session_graph_view"/>
        </record>
        <record model="ir.actions.act_window" id="act_session_calendar">
            <field name="name">Sessions</field>
            <field name="res_model">openacademy.session</field>
            <field name="view_type">form</field>
            <field name="view_mode">calendar</field>
            <field name="view_id" ref="openacademy.session_calendar_view
        </record>
        <record model="ir.actions.act_window" id="act_course_list">
            <field name="name">Courses</field>
```

WebServices

The web-service module offer a common interface for all web-services :

- XML-RPC
- JSON-RPC

Business objects can also be accessed via the distributed object mechanism. They can all be modified via the client interface with contextual views.
Odoo is accessible through XML-RPC/JSON-RPC interfaces, for which libraries exist in many languages.

XML-RPC Library

The following example is a Python program that interacts with an Odoo server with the library `xmlrpclib` :

```
import xmlrpclib

root = 'http://%s:%d/xmlrpc/' % (HOST, PORT)

uid = xmlrpclib.ServerProxy(root + 'common').login(DB, USER, PASS)
print "Logged in as %s (uid: %d)" % (USER, uid)

# Create a new note
sock = xmlrpclib.ServerProxy(root + 'object')
args = {
    'color' : 8,
    'memo' : 'This is a note',
    'create_uid': uid,
}
note_id = sock.execute(DB, uid, PASS, 'note.note', 'create', args)
```

Exercise

Add a new service to the client

Write a Python program able to send XML-RPC requests to a PC running Odoo (yours, or your instructor's). This program should display all the sessions, and their corresponding number of seats. It should also create a new session for one of the courses.

```
import functools
import xmlrpclib
HOST = 'localhost'
PORT = 8069
DB = 'openacademy'
USER = 'admin'
PASS = 'admin'
ROOT = 'http://%s:%d/xmlrpc/' % (HOST,PORT)

# 1. Login
uid = xmlrpclib.ServerProxy(ROOT + 'common').login(DB,USER,PASS)
print "Logged in as %s (uid:%d)" % (USER,uid)

call = functools.partial(
    xmlrpclib.ServerProxy(ROOT + 'object').execute,
    DB, uid, PASS)

# 2. Read the sessions
sessions = call('openacademy.session','search_read', [], ['name','seats'
for session in sessions:
    print "Session %s (%s seats)" % (session['name'], session['seats'])
# 3.create a new session
session_id = call('openacademy.session', 'create', {
    'name' : 'My session',
    'course_id' : 2,
})
```

Instead of using a hard-coded course id, the code can look up a course by name:

```
# 3.create a new session for the "Functional" course
course_id = call('openacademy.course', 'search', [('name','ilike','Funct
session_id = call('openacademy.session', 'create', {
    'name' : 'My session',
    'course_id' : course_id,
})
```

JSON-RPC Library

The following example is a Python program that interacts with an Odoo server with the standard Python libraries `urllib2` and `json`:

```
import json
import random
import urllib2

def json_rpc(url, method, params):
    data = {
        "jsonrpc": "2.0",
        "method": method,
        "params": params,
        "id": random.randint(0, 1000000000),
    }
    req = urllib2.Request(url=url, data=json.dumps(data), headers={
        "Content-Type":"application/json",
    })
    reply = json.load(urllib2.urlopen(req))
    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})

# log in the given database
url = "http://%s:%s/jsonrpc" % (HOST, PORT)
uid = call(url, "common", "login", DB, USER, PASS)

# create a new note
args = {
    'color' : 8,
    'memo' : 'This is another note',
    'create_uid': uid,
}
note_id = call(url, "object", "execute", DB, uid, PASS, 'note.note', 'create', args)
```

Here is the same program, using the library jsonrpclib (https://pypi.python.org/pypi/jsonrpclib):

```
import jsonrpclib

# server proxy object
url = "http://%s:%s/jsonrpc" % (HOST, PORT)
server = jsonrpclib.Server(url)

# log in the given database
uid = server.call(service="common", method="login", args=[DB, USER, PASS])

# helper function for invoking model methods
def invoke(model, method, *args):
    args = [DB, uid, PASS, model, method] + list(args)
    return server.call(service="object", method="execute", args=args)

# create a new note
args = {
    'color' : 8,
    'memo' : 'This is another note',
    'create_uid': uid,
}
note_id = invoke('note.note', 'create', args)
```

Examples can be easily adapted from XML-RPC to JSON-RPC.

Note

There are a number of high-level APIs in various languages to access Odoo systems without *explicitly* going through XML-RPC or JSON-RPC, such as:

- **https://github.com/akretion/ooor (https://github.com/akretion/ooor)**
- **https://github.com/syleam/openobject-library (https://github.com/syleam/openobject-library)**
- **https://github.com/nicolas-van/openerp-client-lib (https://github.com/nicolas-van/openerp-client-lib)**
- **http://pythonhosted.org/OdooRPC (http://pythonhosted.org/OdooRPC)**
- **https://github.com/abhishek-jaiswal/php-openerp-lib (https://github.com/abhishek-jaiswal/php-openerp-lib)**

[1] it is possible to `disable the automatic creation of some fields` (../reference/orm.html#odoo.models.Model._log_access)
[2] writing raw SQL queries is possible, but requires care as it bypasses all Odoo authentication and security mechanisms.