Desarrollo de componentes

Sistemas de Gestión Empresarial

Desarrollo de aplicaciones multiplataforma

curso 2023 - 2024

IES Clara del Rey

Índice

Introducción

El entorno de desarrollo.

Servidor de Odoo

- Máquina virtual **odooserver23_24** para VirtualBox (preferible v7)
- Directorio de instalación: */usr/lib/python3/dist-packages/odoo*
- Fichero de configuración de Odoo: */etc/odoo/odoo.conf* (conexión a la BD postgres)
- Fichero de **logs** de Odoo: configurado en */var/log/odoo/odoo-server.log*

Desarrollo de componentes:

- Visual Studio Code
- Ficheros *.py, xml y csv*
- Utilidades para traspaso de ficheros:
  - file.io
  - hackmd.io

Arranque y parada del servicio

El servidor Odoo suele arrancar como un servicio con arranque automático en el inicio del sistema operativo. Está configurado en el archivo */lib/systemd/system/odoo.service*. La instrucción de arranque del servicio es

```
ExecStart=/usr/bin/odoo --config /etc/odoo/odoo.conf
  --logfile /var/log/odoo/odoo-server.log
```

Para arrancar, parar o consultar el estado del servicio *odoo* se utiliza

```
sudo systemctl [stop | start] odoo
sudo systemctl status odoo
```

1. Estructura básica de un módulo de Odoo

Las aplicaciones incorporadas en la instalación *Community* están en el directorio */usr/lib/python3/dist-packages/odoo/addons*

El nombre de la aplicación o módulo se corresponde con el nombre de la carpeta en la carpeta *addons*.

Las aplicaciones y módulos a desarrollar se van a crear en **/var/lib/odoo/modules**, que es el directorio configurado para esta tarea.

Un módulo de Odoo se compone de dos archivos principales y una serie de carpetas con una estructura común.

Estructura básica. Ficheros principales

___init___.py

Es el primer archivo que se lee. Indica los archivos python a importar y ejecutar. Cualquier directorio con archivos python debe incluirlo aunque esté vacío.

___manifest___.py

Manifiesto con metadatos del módulo como autor, dependencias, datos de instalación, seguridad o datos demo. Utiliza un diccionario tipo clave-valor.
Se debe descomentar la línea que contiene la lista de control de acceso en el fichero *'security/ir.model.access.csv'*
Incluye los ficheros XML que deben ser cargados de forma ordenada.
Al agregar nuevos archivos a un módulo deben declararse en estos archivos.

Estructura básica. Directorios principales

- **models**: archivos .py. Se define un ejemplo del modelo de datos y sus campos.
- **views**: describe las vistas (interfaces) de nuestro módulo (formulario, árbol ,menús, etc.)
- **demo**: datos de prueba precargados en ficheros .xml
- **static**: archivos javascript, hojas de estilo e imágenes.
- **security**: archivos .xml y csv con permisos de grupos y reglas de seguridad. Archivos *ir.model.access.csv* y *security.xml*
- **report**: modelos de informe y archivos .py e .xml relacionados
- **controllers**: controladores de rutas http del sitio web
- **views/templates.xml**: ejemplos de vistas "qweb" usado por el controlador de rutas

Estructura básica. Nomenclatura

Se basa en un nombre jerárquico separado por ramas de cada modelo.

Módulo *mimodulo* con dos modelos *modelo1* y *modelo2*

- *mimodulo_modelo1.py* y *mimodulo_modelo2.py*
- Vistas, menús y acciones:
  - *mimodulo.modelo1_view_form*
  - *mimodulo.modelo1_view_tree*
  - *mimodulo.modelo1_view_kanban*
  - *mimodulo.modelo1_action_window*
  - *mimodulo.menu_root*

Guía de referencia de Odoo

2. Generación de un módulo

2.1. Creación de un módulo con *scaffold*

El comando *scaffold* genera los archivos y carpetas mínimas necesarias para que el módulo sea reconocido por Odoo.
Sintaxis:
`$ ./odoo scaffold <nombre_del_modulo> <carpeta de addons>`
El nombre del módulo no debe comenzar por número ni contener el carácter '_'

Permisos de la carpeta del módulo

La carpeta del módulo debe tener los permisos necesarios para que se pueda modificar el contenido de los archivos. En nuestra máquina, el usuario *odoo* es el propietario de la carpeta */var/lib/odoo/modules*.
En nuestro caso, es la carpeta configurada para aplicaciones desarrolladas por usuario.

Árbol de directorios generado con scaffold

```
odoo@linuxserver2324:~/modules$ tree mymodule0
mymodule0
├── controllers
│   ├── controllers.py
│   └── __init__.py
├── demo
│   └── demo.xml
├── __init__.py
├── __manifest__.py
├── models
│   ├── __init__.py
│   └── models.py
├── security
│   └── ir.model.access.csv
└── views
    ├── templates.xml
    └── views.xml

5 directories, 10 files
```

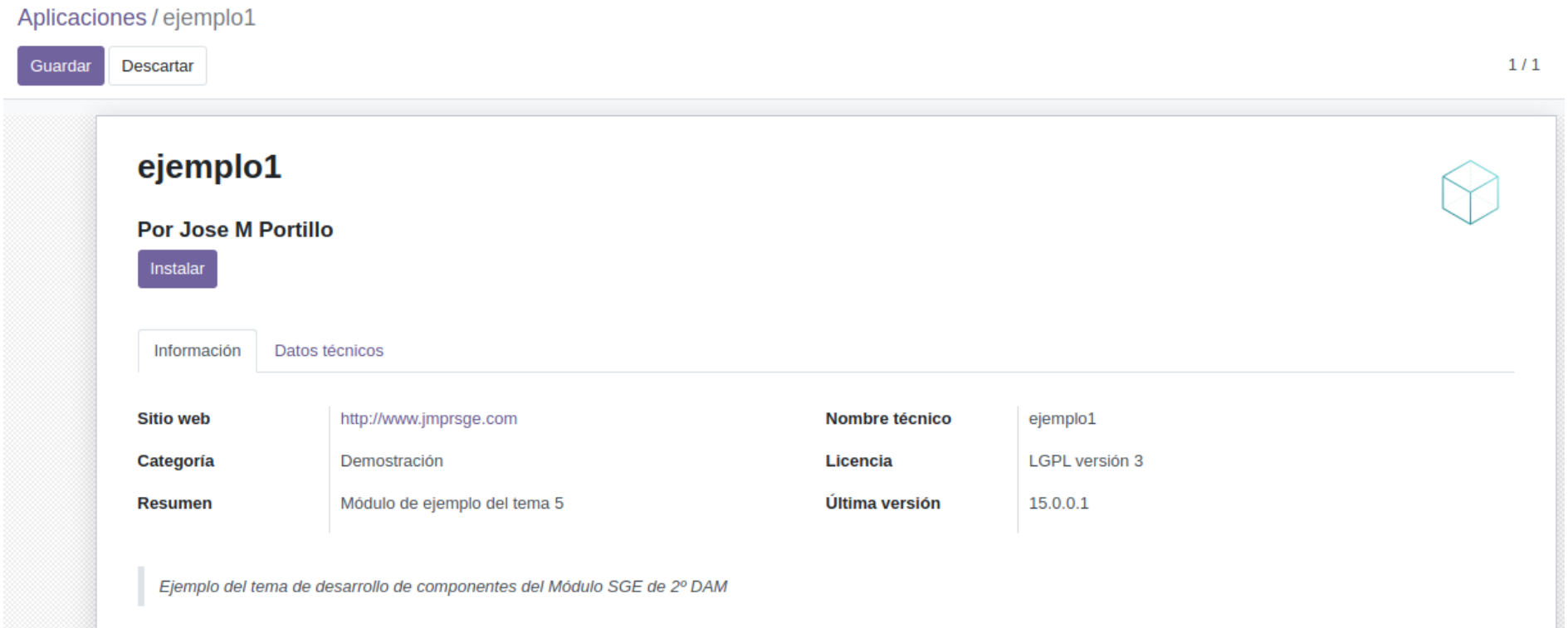
Ficheros generados por scaffold: __init__.py y __manifest__.py

El fichero **__init__.py** tiene el nombre de los ficheros Python o directorios que contienen la lógica del módulo.

```
odoo@linuxserver2324:~/modules$ more mymodule0/__init__.py
# -*- coding: utf-8 -*-

from . import controllers
from . import models
odoo@linuxserver2324:~/modules$ more mymodule0/__manifest__.py
# -*- coding: utf-8 -*-
{
    'name': "mymodule0",

    'summary': """
        Short (1 phrase/line) summary of the module's purpose, used as
        subtitle on modules listing or apps.openerp.com""",

    'description': """
        Long description of module's purpose
    """,

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

    # Categories can be used to filter modules in modules listing
    # Check https://github.com/odoo/odoo/blob/16.0/odoo/addons/base/data/ir_module_category_data.xml
    # for the full list
    'category': 'Uncategorized',
    'version': '0.1',

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

Archivo __manifest__.py

Incluye los datos básicos del módulo que aparecen en la vista kanban de las aplicaciones y la información necesaria para interpretar todos los ficheros que contiene el directorio.



Archivo ___manifest___.py

_manifest__.py

```python
# -*- coding: utf-8 -*-
{
    'name': "ejemplo1",

    'summary': """
        Módulo de ejemplo del tema 5""",

    'description': """
        Ejemplo del tema de desarrollo de componentes del Módulo SGE de 2º DAM
    """,

    'author': "Jose M Portillo",
    'website': "http://www.jmprsge.com",

    # Categories can be used to filter modules in modules listing
    # Check https://github.com/odoo/odoo/blob/14.0/odoo/addons/base/data/ir_module_category_data.xml
    # for the full list
    'category': 'Demostración',
    'version': '0.1',

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

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

2.2. Instalación de módulos

Al instalar o actualizar un módulo de Odoo, el manifest indica al actualizador dónde están los ficheros de datos y otros parámetros para que actualice la base de datos.

Cuando instalamos un módulo, las vistas, datos, etc. (los .xml) se almacenan en la base de datos. La actualización de la base de datos con datos, vistas, etc. sólo se hace al instalar o actualizar el módulo. Así que aunque cambiemos una vista en un fichero XML, no se verá el cambio si no actualizamos el módulo Odoo.

Los ficheros Python de un módulo son cargados de nuevo cada vez que se inicia el servicio Odoo. Por lo que si cambiamos algo en ellos, tenemos dos opciones para observar los cambios: recargar el módulo o reiniciar el servicio Odoo.

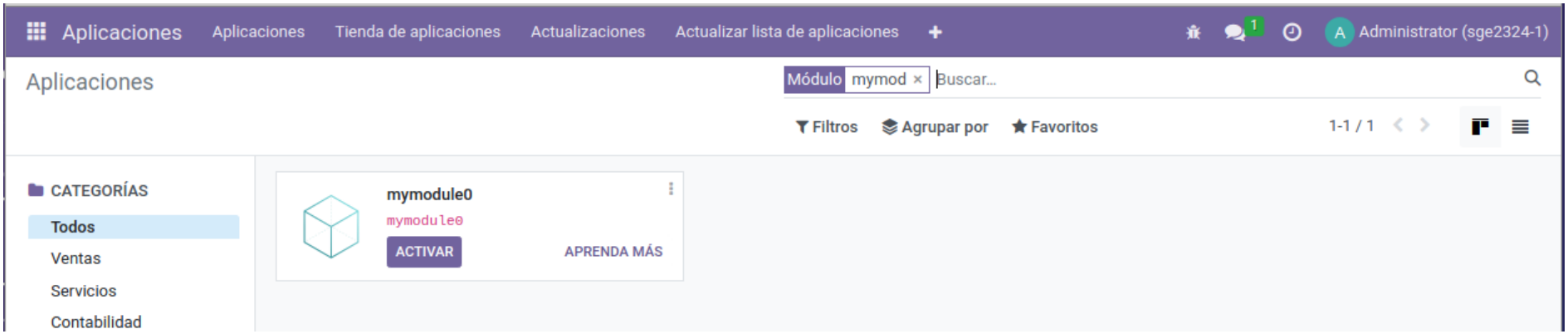
Instalar el módulo no consiste en más que situar la estructura necesaria de archivos en los directorios asignados por Odoo para las aplicaciones y módulos

Instalación de módulos

Para esta tarea es necesario trabajar en el **modo desarrollador**

(Ajustes -> (al final) -> Activar el modo desarrollo)

En *Aplicaciones* aparece el menú *Actualizaciones* que permite recargar las aplicaciones y módulos nuevos. Hay que ajustar los filtros para que aparezcan.



Para ver si va bien, o hay errores, abrir una consola con la instrucción *tail -f /var/log/odoo/odoo-server.log*

2.2. Creación de los modelos

El fichero *models.py* de la carpeta *models* define el modelo o modelos de cada módulo.Un modelo se define como una clase Python que hereda de la clase **“models.Model”**

Cada objeto se corresponde con una clase Python que extiende la clase *Model*. Es buena práctica nombrarlas incluyendo el nombre del módulo.

El nombre de las clases de Python siempre ha de ser en minúscula y con el punto para separar por jerarquía. El nombre de un modelo, por lo tanto, será siempre: **“modulo.modelo”**. Si el modelo tiene un nombre compuesto, se separa por “_”.

En la base de datos, al nombrar el modelo el punto se sustituye por una barra baja.

Código de un modelo en *models.py*

Los campos mínimos obligatorios que deben contener son:

- *_name*: de sintaxis recomendada mimodulo.modelo1. Es el nombre del modelo.
- *name*: tipo *char*. Siempre debe existir para su uso en las vistas de búsquedas y relaciones entre objetos.

Además, se enumeran el resto de campos del modelo con sus tipos y atributos.

```
20  from odoo import models, fields, api
21
22
23  class garaje(models.Model):
24      _name = 'parking.garaje'
        _description = 'Características de los garajes'
6
27      name = fields.Char('Dirección', required=True)
28      plazas = fields.Integer(string='Plazas', required=True)
29
```

Tipos de campos básicos

Los modelos pueden tener atributos internos de la clase, propio del lenguaje Python. Pero los atributos que se guardan en la base de datos se han de definir cómo "fields". Un "field" es una instancia de la clase **"fields.Field"**, y tiene sus propios atributos y funciones.

- fields.Char(string='Nombre', required=True, size=7)
- fields.Date(string='Fecha de fabricación')
- fields.Float('Precio', (4, 1), default=0.0, help='Precio en euros €')
- fields.Boolean(string='Defectuoso', default=False)
- fields.Text('Descripción')
- fields.Selection(string='Tipo', selection=[('l','Lavado'),('r','Revisión'),('m','Mecánica'),('p','Pintura')], default='l')
- fields.Binary('Foto de perfil', help='Introduzca la fotografía')

Campos relacionales

Many2one

Indica que el modelo en el que está tiene una relación muchos a uno con otro modelo.
Esto significa que un registro tiene relación con un único registro del otro modelo, mientras que el otro registro puede tener relación con muchos registros del modelo que tiene el "Many2one".
En la tabla de la base de datos, esto se traducirá en una clave ajena a la otra tabla.
micampo_id = fields.Many2one ('mimodulo.elotromodelo', string='Lo que se relaciona en el otro modelo')
Una ciudad almacena el país al que pertenece:

```
# La forma más común, en el Modelo Ciudad
pais_id = fields.Many2one('modulo.pais')
# Otra forma, con argumento
pais_id = fields.Many2one(comodel_name='modulo.pais')
```

Campos relacionales

One2many

La inversa del “Many2one”. Necesita que exista un “Many2one” en el otro modelo relacionado.
Este “field” no supone ningún cambio en la base de datos, ya que es el equivalente a hacer un ‘SELECT’ sobre las claves ajenas de la otra tabla.
loqueserelaciona_ids = fields.One2many(‘mimodulo.elotromodelo’, ‘loqueserelaciona’, string=‘Lo que se relaciona’)
(el campo string es opcional, sirve de aclaración)
En un país, se accede a sus ciudades:

```
# Esto en el modelo Ciudad, indica un many2one
pais_id = fields.Many2one('modulo.pais')

# En el modelo Pais.
# El nombre del modelo y el field que tiene el Many2one necesa
ciudades_ids = field.One2many('modulo.ciudad', 'pais_id')
```

Campos relacionales

Many2many

Relación muchos a muchos.
Esto se acaba mapeando como una tabla intermedia con claves ajenas a las dos tablas. Hacer los "Many2many" simplifica mucho la gestión de estas tablas intermedias y evita redundancias o errores.
loqueserelaciona_ids = fields.Many2many('mimodulo.elotromodelo', string='Lo que se relaciona')
Relación entre un alumno y sus módulos de un ciclo:

```
# Esto en el modelo Alumno
modulos_ids = fields.Many2many('modulo.modulo')

# Esto en el modelo Modulo.
alumnos_ids = field.Many2many('modulo.alumno')
```

Ejemplos de campos para relaciones

Relación de uno a varios

```
29
30      # relaciones entre tablas
31      coche_ids = fields.One2many('parking.coche', 'garaje_id', string='Coches')
32
```

Relaciones de muchos a uno, o muchos a muchos

```
49      # relaciones entre tablas
50      garaje_id = fields.Many2one('parking.garaje', string='Garaje')
51      mantenimiento_ids = fields.Many2many('parking.mantenimiento',
52                                           string='Mantenimientos')
53
```

2.3. Diseño de las vistas básicas

Se describen en el archivo *views.xml* de la carpeta *views*. Definen las vistas y las opciones de menú con sus acciones de ventana.

Tiene varios elementos para funcionar:

- **Definiciones de vistas**: Las propias definiciones de las vistas, guardadas en el modelo "ir.ui.view". Estos elementos tienen al menos los "fields" que se van a mostrar y pueden tener información sobre la disposición, el comportamiento o el aspecto de los "fields".
- **Menus**: Se distribuyen de forma jerárquica y se guardan en el modelo "ir.ui.menu".
- **Actions**: Enlazan una acción del usuario (como pulsar en un menú) con una llamada al servidor desde el cliente para pedir algo (como cargar una nueva vista). Las *actions* están guardadas en varios modelos dependiendo del tipo.

Vistas

Las vistas son objetos de tipo *record* de la tabla *ir.ui.view*. Su identificador sigue la nomenclatura *nombredelmodulo.nombredelmodelo_tipo* usando un solo punto en el nombre (el tipo es form, tree, kanban, ...). Se definen

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

- *name*: nombredelmodulo.nombredelmodelo.tipo (el nombre del id cambiando guión por punto)
- *model*: objeto al que se aplica la vista
- *arch*: tipo de archivo

Codificación de las vistas de lista y formulario

Vista de Árbol (Tree)

```
<record model="ir.ui.view" id="mimodulo.modelo1_list_view">
      <field name="name">mimodulo.modelo1.view.tree</field>
      <field name="model">mimodulo.modelo1</field>
      <field name="arch" type="xml">
        <tree>
          <field name="name"/>
          <field name="description"/>
        </tree>
      </field>
    </record>
```

Vista de formulario (Form)

```
<record model="ir.ui.view" id="mimodulo.modelo1_form_view">
    <field name="name">mimodulo.modelo1.view.form</field>
    <field name="model">mimodulo.modelo1</field>
    <field name="arch" type="xml">
      <form string="Formulario de modelo1">
        <sheet>
            <h1>
              <field name="name" placeholder="Dirección"/>
            </h1>
            <group>
              <group>
                <separator string="Información general"/>
                  <field name="plazas"/>
              </group>
              .....
            </group>
        </sheet>
```

Ejemplos de las vistas de lista y formulario

Vista de lista o árbol

```
<record model="ir.ui.view" id="parking.garaje_list_view">
  <field name="name">parking.garaje.view.tree</field>
  <field name="model">parking.garaje</field>
  <field name="arch" type="xml">
    <tree>
      <field name="name"/>
      <field name="plazas"/>
    </tree>
  </field>
</record>
```

Vista formulario

```
<record model="ir.ui.view" id="parking.garaje_form_view">
    <field name="name">parking.garaje.view.form</field>
    <field name="model">parking.garaje</field>
    <field name="arch" type="xml">
      <form string="Formulario de garaje">
```

2.4. Menús y acciones

Los elementos de menú responden a una jerarquía tipo árbol.

Los campos obligatorios a definir son:

- **name** que es el literal que se ve en el navegador,
- **id**, el identificador del menú en la aplicación
- **parent**, el menú del que surge
- **action**, la acción que se lanza al pulsar en el menú

Mientras no se asignen permisos a los usuarios de la plataforma, se pueden visualizar con la opción **Convertirse en superusuario**

Ejemplo de elementos de menú y acciones

Acciones de ventana

```xml
<record model="ir.actions.act_window" id="parking.coche_action_window">
  <field name="name">Coches</field>
  <field name="res_model">parking.coche</field>
  <field name="view_mode">tree,form</field>
</record>
```

Elementos de menú

```xml
<!-- Top menu item -->

<menuitem name="Parking" id="parking.menu_root" web_icon="garaje,static/description/icon.png" />

<!-- menu categories -->
<menuitem name="Garajes" id="parking.garaje_menu" parent="parking.menu_root" action="parking.garaje_action_window"/>
<menuitem name="Coches" id="parking.coche_menu" parent="parking.menu_root" action="parking.coche_action_window"/>
<menuitem name="Mantenimientos" id="parking.mantenimiento_menu" parent="parking.menu_root" action="parking.mantenimient
```

3. Permisos del módulo

Las políticas de seguridad de acceso se configuran mediante mecanismos de control de acceso.

Se trabaja creando el fichero *security.xml* y modificando el *ir.model.access.csv* en la carpeta *security*.

Se añade una línea *(record)* por cada uno de los roles (grupos) con sus permisos. Los grupos son registros de la clase *res.groups*. En el fichero *.csv* se especifican los permisos de cada rol. Cada control de acceso se asocia a un modelo, un grupo y un conjunto de permisos (lectura, escritura, crear y borrar).

Si se quiere que los grupos de una aplicación estén asociados, se les reúne en una categoría creando un registro de la clase *ir.module.category*

Ficheros de permisos

El fichero *security.csv* tiene como cabecera:

```
id,name,model_id:id,group_id:id,
    perm_read,perm_write,perm_create,perm_unlink
```

- *id*: identificador de la regla
- *name*: nombre para la regla
- *model_id:id*: tabla del modelo sobre la que se aplica la regla
- *group_id:id*: grupo al que aplica

Ejemplos de ficheros de permisos

ir.model.access.csv

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_parking_mantenimiento_usuario,parking.mantenimiento.usuario,model_parking_mantenimiento,parking.group_parking_usuario,1,1,1,1
access_parking_coche_usuario,parking.coche.usuario,model_parking_coche,parking.group_parking_usuario,1,1,1,1
access_parking_garaje_usuario,parking.garaje.usuario,model_parking_garaje,parking.group_parking_usuario,1,0,0,0
access_parking_garaje_director,parking.garaje.director,model_parking_garaje,parking.group_parking_director,1,1,1,1
```

security.xml

```xml
<record id="group_parking_director" model="res.groups">
    <field name="name">Director</field>
    <field name="category_id" ref="parking.module_category_garaje" />
    <field name="comment">Usuarios que gestionan los parkings</field>
    <!-- Los miembros de este grupo son también miembros del grupo de usuarios-->
    <field name="implied_ids" eval="[(4, ref('group_parking_usuario'))]" />
```

4. Carga de datos de prueba

En el directorio ***data*** se pueden incluir ficheros .xml que contienen los registros de datos que se cargarán en la base de datos junto con la aplicación obligatoriamente.

Si se quiere que esta carga sea opcional, el fichero con los datos de prueba se sitúa en el directorio ***demo***. En este caso hay que activar en la base de datos la carga de datos de prueba.

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

Ejemplo de dato de prueba

```
<record id="coche01" model="parking.coche">
    <field name="name">1234abc</field>
    <field name="modelo">Mercedes C</field>
    <field name="construido">1996-06-10</field>
    <field name="consumo">6.3</field>
    <field name="averiado">True</field>
    <field name="descripcion">Clasico del 96</field>
    <field name="garaje_id" ref="garaje01" />
</record>
```

5. Creación paso a paso

1. Creación de los modelos de la aplicación
    1. Diseño de los modelos y sus campos
    2. Codificación de los modelos en el fichero *models.py*
    3. Asignación de atributos de cada modelo
    4. Codificación de los campos de relaciones

4. Creación paso a paso

2. Instalación del módulo o aplicación
3. Creación de vistas asociadas en el fichero *views.xml*
    1. Creación de elementos de menús y acciones de ventana
    2. Creación de las vistas necesarias de cada modelo
    3. Actualizar el modelo o aplicación
4. Asignación de permisos a los grupos de usuarios
5. Generación de datos de prueba
6. Otros detalles de la aplicación: campos calculados, estética de formularios, ...

Anexo I: Base de datos Postgres

Instalación

La BD Postgres se ha instalado anteriormente a la instalación de Odoo. Las instrucciones de instalación son

```
sudo apt-get install postgresql postgresql-client
```

En la base de datos están creados los usuarios *postgres* y *odoo* con permisos de creación de bases de datos.

El servidor de bases de datos responde en el puerto **5432**

Gestión del SGBD postgres

Con la orden **psql** se puede acceder mediante terminal a la aplicación de gestión de las bases de datos Postgres

Algunas órdenes básicas:

- Conectarse a una base de datos usando psql:

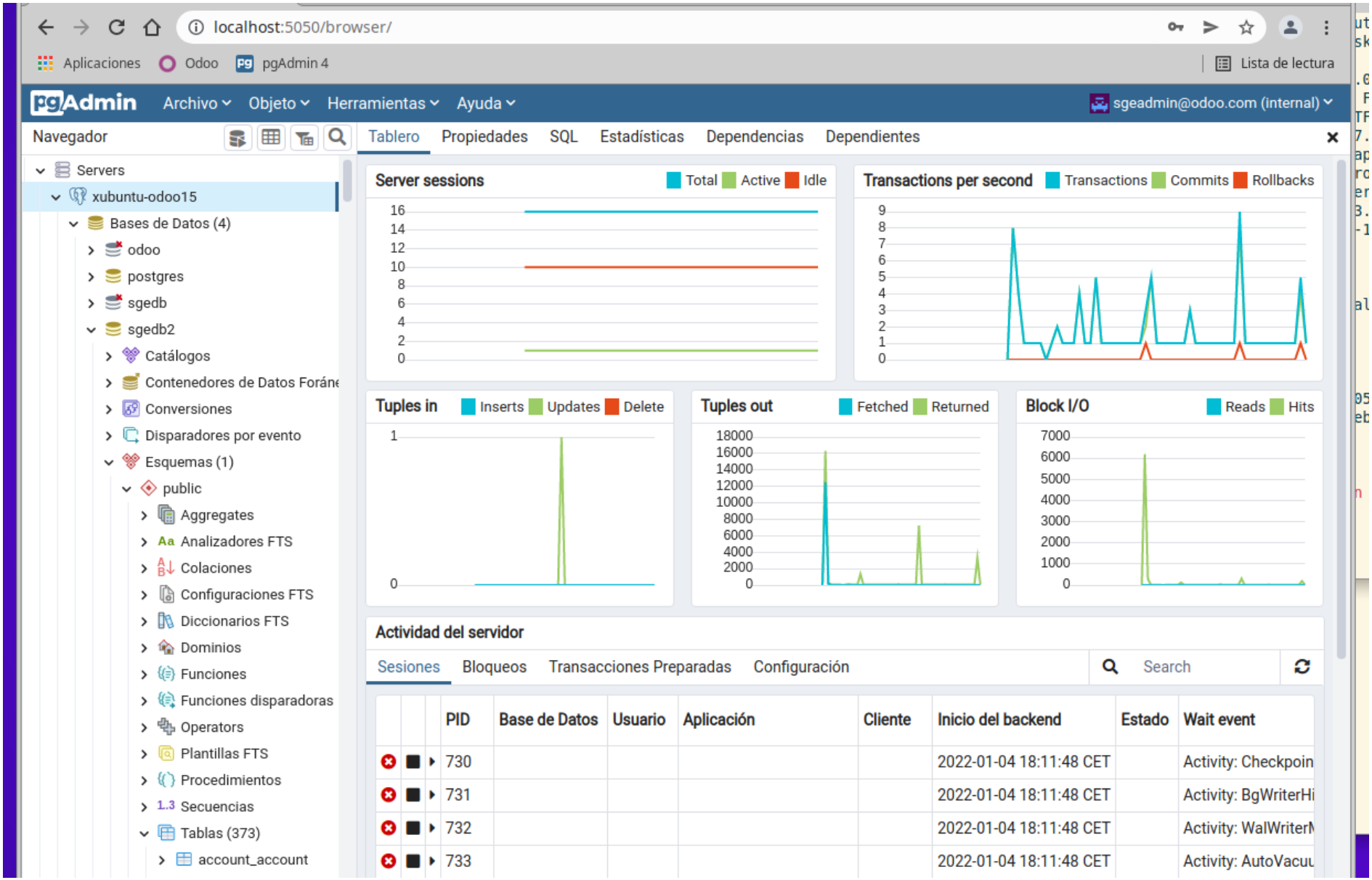
psql -h servidor -U usuario -d base-datos

- **l**: Lista de las BBDD
- **d**: Describe la base de datos en uso
- **c base-datos**: Cambia a otra base de datos
- **q** ó **Ctrl+D**: Salir de la sesión
- **createdb**: Crear nueva base de datos
- **dropdb**: Eliminar base de datos
- **psql -l**: Listar BBDD existentes

Instalación de **pgadminIV**

```
sudo mkdir /var/lib/pgadmin
sudo mkdir /var/log/pgadmin
sudo chown $USER /var/lib/pgadmin
sudo chown $USER /var/log/pgadmin
sudo apt install python3.9-venv

source pgadmin4/bin/activate

(pgadmin4) odoo@xubuntu-odoo15:~$ pip install pgadmin4
(pgadmin4) odoo@xubuntu-odoo15:~$ pgadmin4
Email address: sgeadmin@odoo.com
Password: password
```

Interfaz de **pgadminIV**



Gracias por su atención