odoo

Publicación 8.0

July 01, 2020

Índice general

Odoo Development Essentials Book

1.1 Copyright del libro

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2015

Production reference: 1300315

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-279-6

http://www.packtpub.com

1.2 Créditos

Autor

Daniel Reis

Revisores

Pedro M. Baeza

Nicolas Bessi

Alexandre Fayolle

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Subho Gupta

Content Development Editor Siddhesh Salvi

Technical Editors

Ankur Ghiye

Manali Gonsalves

Naveenkumar Jain

Copy Editors

Hiral Bhat

Pranjali Chury

Wishva Shah

Sameen Siddiqui

Project Coordinator

Nidhi J. Joshi

Proofreaders

Paul Hindle

Chris Smith

Indexer

Tejal Soni

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

1.3 Acerca del autor

Figura 1.1: Daniel Reis

Autor

Daniel Reis has worked in the IT industry for over 15 years, most of it for a multinational consultancy firm, implementing business applications for a variety of sectors, including telco, banking, and industry. He has been working with Odoo (formerly OpenERP) since 2010, is an active contributor to the Odoo Community Association projects, and has been a regular speaker at the OpenDays annual conference.

He currently works at Securitas, a global security services company, where he introduced Python and Odoo into the applications portfolio.

1.4 Acerca de los revisores

Figura 1.2: Pedro M. Baeza
Revisor técnico

Pedro M. Baeza is an Odoo freelance consultant, developer, and trainer with more than 16 years of experience in IT. He's been in the Odoo world for 4 years, and has been involved in its community since the beginning, first in the Spanish community, and then in the worldwide community that later formed the Odoo Community Association (OCA).

Currently, he is the Spanish localization PSC and website PSC team leader, and also an active reviewer and contributor for most of the community projects.

He doesn't have direct employees, but collaborates with other companies and freelancers to deploy Odoo implementations. He feels that the best part of this is having to contact a lot of awesome people to work with to get to a common goal and that this is the perfect environment for getting close to perfection!

I would like to thank the awesome community, which is spread around the world, for pushing me a little further and adding to my knowledge. I also want to thank my girlfriend (and future wife), Esther, for understanding why I'm unable to spend time with her because of the job and my current commitment to the community.

Figura 1.3: Nicholas Bessi
Revisor técnico

Nicholas Bessi has been an Odoo/OpenERP developer and consultant since 2006 when it was still TinyERP. He is the author of many modules including the "report webkit" add-on that was part of the official add-ons for many years, which inspired the actual QWeb report engine.

He's an active member of Odoo Community Association and is responsible for Swiss localization. He was recognized as an OpenERP top contributor in 2010, and is still an active partisan of Open Source values.

Nicholas is a technical leader at Camptocamp, a leading society in Open Source technologies that is a historical Odoo contibutor and partner. Camtocamp is actively working alongside Odoo to bring the solution to the next level.

Alexandre Fayolle installed his first Linux distribution in 1995 (Slackware at the time, before moving to Debian in 1996) and has never used another OS on his computers since. He started using Python in 1999 when he cofounded Logilab, where he was a CTO, software architect, and Agile coach. He got the opportunity to participate in a large number of FLOSS projects, including pyxml, Pypy, Cubicweb, and Pylint. In 2012, he joined Camptocamp to work on Odoo, which was still called OpenERP at the time. He became a very active member of the Odoo Community Association, both as a direct module contributor and as a mentor to new comers. He also happens to be a jazz vibraphone player.

Figura 1.4: Alexandre Fayolle
Revisor técnico

1.5 Prefacio

Odoo es una plataforma poderosa de código abierto para aplicaciones de negocio. Sobre esta se encuentra una suite de aplicaciones estrechamente integradas, que cubren todas las áreas de negocio desde CRM y Ventas hasta Contabilidad y Suministros.

Odoo tiene una comunidad dinámica y en constante crecimiento, constantemente se agregan características, conectores, y aplicaciones de negocio adicionales. Sin embargo, la curva de aprendizaje inicial para el desarrollo Odoo puede ser bastante cara.

El libro **Odoo Development Essentials** provee una guía paso a paso para el desarrollo con Odoo, para escalar rápidamente la curva de aprendizaje y emprender productivamente en la plataforma Odoo.

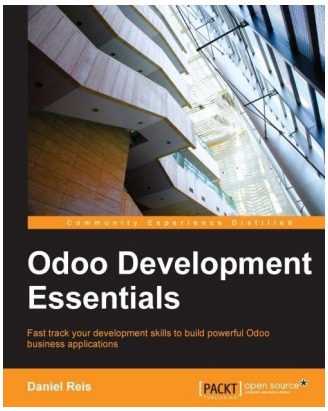

Figura 1.5: Portada del libro “Odoo Development Essentials”

Los primeros tres capítulos tiene como finalidad hacer que la persona que lee se sienta cómoda con Odoo, las técnicas básicas para configurar un entorno de desarrollo, el desarrollo de módulos y flujos de trabajo.

Los cinco capítulos siguientes explican en detalle varias áreas de desarrollo usadas en los módulos: archivos de datos, modelos, vistas, lógica de negocio, y QWeb.

Los dos capítulos finales son una guía a través de la integración de aplicaciones Odoo con aplicaciones externas y la discusión sobre las consideraciones a tener en cuenta al implementar instancias de Odoo para su uso en producción.

1.6 ¿Qué abarca este libro?

Capítulo 1, **Comenzar con el Desarrollo de Odoo**, abarca la configuración del entorno de desarrollo, instalación de Odoo desde el código fuente, y aprender como se gestionan las instancias del servidor Odoo.

Capítulo 2, **Desarrollar una Primera Aplicación con Odoo**, es una guía a través de la creación de un primer módulo de Odoo, cubre todas las capas involucradas: modelos, vistas y lógica de negocio.

Capítulo 3, **Herencia - Ampliar las Aplicaciones Existentes**, explica los mecanismos de herencia y sus usos para crear módulos de extensión que agregan o modifican características en otros módulos existentes.

Capítulo 4, **Serialización de Datos y Datos de Módulo**, abarca los formatos de archivo de datos más usados en Odoo, XML y CSV, identificadores externos, y como usar archivos de datos en los módulos y en la importación y exportación de datos.

Capítulo 5, **Modelos – Estructurar los Datos de la Aplicación**, describe en detalle la capa de Modelo con los tipos de modelos y campos disponibles, incluyendo los campos relacionales y computados.

Capítulo 6, **Vistas – Diseñar la Interfaz de Usuario**, abarca la capa de Vista, explicando en detalle los tipos de vistas y todos los elementos que pueden ser usados para crear interfaces dinámicas e intuitivas.

Capítulo 7, **Lógica de Aplicación ORM – Apoyar la Lógica de Negocio**, introduce a la programación de la lógica de negocio del lado del servidor, explorando los conceptos de ORM y sus características, también explica como usar wizards para una interacción más compleja con las usuarias y los usuarios.

Capítulo 8, **QWeb – Crear Vistas y Reportes Kanban**, recorre las plantillas QWeb de Odoo, usándolas para crear sofisticadas pizarras Kanban y reportes basados en HTML.

Capítulo 9, **API Externa – Integración con Otros Sistemas**, explica como usar la lógica del servidor Odoo desde aplicaciones externas, e introduce una popular librería de programación que puede también ser usada como cliente de línea de comando.

Capítulo 10, **Lista de Verificación de Implementación – En Vivo**, muestra como preparar un servidor para ser usado en producción, explica cuales configuraciones deben ser tomadas en cuenta y como configurar un proxy inverso Nginx para mejorar la seguridad y escalabilidad.

1.7 ¿Qué se necesita para este libro?

Se realiza la instalación de Odoo en un servidor con sistema Ubuntu o Debian, pero puede usar las herramientas de programación y el sistema operativo de su preferencia, sea Windows, Macintosh, o cualquier otro.

Aquí se proporciona una guía general para la configuración de na máquina virtual con Ubuntu Server. Puede elegir el software de virtualización que desee, como VirtualBox o VMware Player, ambos disponibles de forma gratuita. Si usa una estación de trabajo Ubuntu o Debian, no es necesaria una máquina virtual.

1.8 ¿A quién va dirigido este libro?

Este libro esta dirigido a desarrolladoras y desarrolladores con experiencia en la creación de aplicaciones de negocio con la disposición de convertirse en personas productivas con Odoo.

Se espera que cuente con conocimientos en el diseño de aplicaciones MVC y programación con lenguaje Python.

1.9 Convenciones

En este libro, encontrará diferentes estilos de texto para distinguir entre distintos tipos de información. Los siguientes son algunos ejemplos de estos estilos, y una explicación de su significado.

El manejo de palabras de código en texto, nombres de tablas de base de datos, nombres de carpetas, nombres de archivos, extensiones de archivos, nombres de ruta, URLs falsas, entrada de usuario, y Twitter se muestran como sigue: “Necesita poder ser importado por Python, por lo tanto tendrá también que ser un archivo **__init__.py**“

Un bloque de código es asentado de la siguiente manera:

```
{
    'name': 'To-Do Application',
    'description':  'Manage your personal Tasks with this module.',
    'author':   'Daniel Reis',
```

```
    'depends':  ['mail'],
    'application':  True,
}
```

Cualquier entrada de línea de comando es escrito como sigue:

```
$ mkdir ~/odoo-dev/custom-addons
```

Termino nuevos y palabras importantes son mostradas en negrita. Las palabras que se muestran en la pantalla, en los menús o ventanas de dialogo por ejemplo, aparecen en texto como este: "Seleccione la opción **Actualizar Listas de Módulos**."

Nota: Las advertencias o **notas importantes** aparecen en una caja como esta.

Truco: Los consejos y notas aparecen así.

1.9.1 Acerca del codigo fuente

El código fuente usado en las todas practicas de en libro **Odoo Development Essentials** para la versión Odoo 8.0 esta disponible en la siguiente dirección:

- https://github.com/dreispt/todo_app/tree/8.0

Puede obtener una copia localmente del código fuente, ejecutando el siguiente comando:

```
$ git clone https://github.com/dreispt/todo_app.git -b 8.0
```

Capítulos

2.1 Capítulo 1 - Iniciando

2.1.1 Comenzando con Odoo

Antes de sumergirse en el desarrollo de Odoo, es necesario configurar el entorno de desarrollo, y para esto se debe aprender las tareas básicas de administración.

En este capítulo, se aprenderá como configurar el entorno de desarrollo, donde luego se desarrollarán las aplicaciones Odoo.

Se aprenderá a configurar sistemas Debian o Ubuntu para alojar las instancias del servidor de desarrollo, y como instalar Odoo desde el código fuente en GitHub. Luego aprenderá a configurar archivos compartidos con Samba, permitiendo trabajar con archivos de Odoo desde una estación de trabajo con cualquier sistema operativo.

Odoo está desarrollado usando el lenguaje de programación Python y usa PostgreSQL como base de datos para almacenar datos, estos son los requisitos principales para trabajar con Odoo. Para ejecutar Odoo desde el código fuente, es necesario instalar las librerías Python de las cuales depende. Luego el código fuente de Odoo debe descargarse desde GitHub y ejecutado desde el código fuente. Aunque es posible descargar un zip o tarball, es mejor obtener el código fuente usando GitHub, así además tendría Odoo instalado en su equipo.

Configurar un equipo como servidor Odoo

Se recomienda usar sistemas Debian/Ubuntu para el servidor Odoo, aunque puede trabajar desde el sistema operativo de su preferencia, sea Windows, Macintosh, o Linux.

Odoo puede ser ejecutado en una gran variedad de sistemas operativos, entonces ¿por qué elegir Debian por encima de otros sistemas operativos? Debido a que Odoo es desarrollado principalmente para sistemas Debian/Ubuntu, el soporte para Odoo es mejor. Por lo tanto será más fácil encontrar ayuda y recursos adicionales si se trabaja con Debian/Ubuntu.

También son las plataformas más usadas por las personas que desarrollan aplicaciones, y donde se dan a conocer más implementaciones. Por esta razón, inevitablemente, se espera que los desarrolladores de Odoo se sientan a gusto con esta plataforma. Incluso quienes tiene una historial de trabajo con Windows, es importante que tengan algún conocimiento sobre estas plataformas.

En este capítulo, se aprenderá a configurar y trabajar con Odoo sobre un sistema Debian, usando únicamente la línea de comandos. Para quienes están acostumbrados a sistemas Windows, se describirá como configurar una máquina virtual para alojar un servidor Odoo. Adicionalmente, las técnicas aprendidas servirán para gestionar servidores Odoo en la nube donde el único acceso será a través de una **Shell Segura (SSH).**

Nota: Tenga en cuenta que estas instrucciones tienen como objetivo configurar un nuevo sistema para desarrollo. Si desea probarlas en un sistema existente, haga un respaldo a tiempo que le permita recuperar el sistema en caso de algún problema.

Disposiciones para un equipo Debian

Como se explicó antes, será necesario un equipo con Debian para alojar su servidor Odoo versión 8.0. Si estos son sus primeros pasos con Linux, le gustará saber que Ubuntu es una distribución Linux basada en Debian, por lo tanto son muy similares.

Nota: Odoo asegura su funcionamiento con la versión estable de Debian o Ubuntu. Al momento de elegir este libro, las versión estable para Debian es la versión 7 "Wheezy" y para Ubuntu la versión 14.04 "Trusty Tahr". Ambas se distribuyen con Python 2.7, necesario para ejecutar Odoo.

Si ya está ejecutando Ubuntu u otra distribución basada en Debian, todo esta listo para comenzar; ésta máquina también puede ser usada para alojar Odoo.

Para los sistemas operativos Windows y Macintosh, es posible tener Python, PostgreSQL, y todas las dependencias instaladas, y luego ejecutar Odoo desde el código fuente de forma nativa.

Sin embargo, esto puede ser un gran reto, por lo que su recomendación es usar una máquina virtual ejecutando Debian o Ubuntu Server. Puede usar su software de virtualización preferido para hacer funcionar Debian en una máquina virtual. Si necesita alguna ayuda, aquí hay algunos consejos: en lo que se refiere a software de vistualización, tiene muchas opciones, como Microsoft Hyper-V (disponible para algunas versiones de Windows), Oracle VirtualBox, o VMWare Player (o VMWare Fusion para Macintosh). VMWare Player es probablemente el más fácil de usar, y puede descargarse gratuitamente en https://my.vmware.com/web/vmware/downloads

Con relación a la imagen Linux a usar, Ubuntu Server es más amigable para los usuarios para instalar que Debian. Si esta comenzando con Linux, es recomendable que use una distribución lista para usar. TurnKey Linux provee imágenes fácil de usar, instaladas previamente en distintos formatos, incluyendo ISO. El formato ISO funcionara con cualquier software de virtualización de su preferencia, o incluso en cualquier equipo actual. Una buena opción sería una imagen LAPP, que puede hallarse en la siguiente dirección http://www.turnkeylinux.org/lapp

Una vez instalado el sistema e iniciado, debería ser capaz de ingresar en la línea de comando.

Si ingresa usado `root`, su primera tarea será crear un usuario para ser usado en el trabajo cotidiano, ya que es considerada una mala práctica trabajar como `root`. Particularmente, el servidor Odoo se rehusará a ejecutarse si está usando `root`.

Si está usando Ubuntu, probablemente no necesite esto ya que el proceso de instalación le habrá guiado en la creación de un usuario personal.

Creando una cuenta de usuario para Odoo

Primero, asegúrese que sudo este instalado. Su usuario de trabajo lo necesitará. Si ha accedido como root ejecute los siguientes comandos:

Instalar actualizaciones del sistema, ejecutando el siguiente comando:

```
$ apt-get update & apt-get upgrade
```

Asegurarse que 'sudo' esta instalada, ejecutando el siguiente comando:

```
$ apt-get install sudo
```

Con los siguientes comandos crearán un usuario odoo.

Cree un usuario 'Odoo' con poderes sudo, ejecutando el siguiente comando:

```
$ useradd -m -g sudo -s /bin/bash odoo
```

Solicite y configure una contraseña para el nuevo usuario, ejecutando el siguiente comando:

```
$ passwd odoo
```

Puede cambiar odoo por cualquier nombre que desee. La opción `-m` crea el directorio home. El `-g sudo` agrega al nuevo usuario a la lista de usuarios sudo, por lo tanto podrá ejecutar comandos como root, y `-s /bin/bash` configura la línea de comando predeterminada a bash, la cual es más amigable de usar que la fijada por omisión estándar `sh`.

Ahora puede acceder con el nuevo usuario y configurar Odoo.

2.1.2 Instalar Odoo desde el código fuente

Los paquetes de Odoo listos para instalar pueden ser encontrados en nightly.odoo.com, disponibles para Windows (.exe), Debian (.deb), CentOS (.rpm), y código fuente (.tar.gz).

Como desarrolladores, se prefiriere hacer la instalación directamente desde el repositorio GitHub. Esto les permitirá tener más control sobre las sucesivas versiones y actualizaciones.

Para mantener el orden de las cosas, se trabaja en el directorio `/odoo-dev` que se encuentra en su directorio `/home`. A lo largo del libro, se asume que este es el lugar donde está instalado el servidor Odoo.

Primero, asegúrese que ha accedido con el usuario creado anteriormente, o durante el proceso de instalación, y no como root. Asumiendo que su usuario es `odoo`, puede confirmar esto con el siguiente comando:

```
$ whoami
odoo
$ echo $HOME
/home/odoo
```

Ahora es posible usar este script. Muestra como instalar Odoo desde el código fuente en un sistema Debian:

Instalar las actualizaciones del sistema, ejecutando el siguiente comando:

```
$ sudo apt-get update & sudo apt-get upgrade
```

Instalar Git, ejecutando el siguiente comando:

```
$ sudo apt-get install git
```

Crear el directorio de trabajo, ejecutando el siguiente comando:

```
$ mkdir ~/odoo-dev
```

Ingresar en el directorio de trabajo, ejecutando el siguiente comando:

```
$ cd ~/odoo-dev
```

Obtener el código fuente de Odoo, ejecutando el siguiente comando:

```
$ git clone https://github.com/odoo/odoo.git -b 8.0
```

Instalar las dependencias del sistema Odoo, ejecutando el siguiente comando:

```
$ ./odoo/odoo.py setup_deps
```

Instalar PostgreSQL y el usuario administrador para un usuario Unix, ejecutando el siguiente comando:

```
$ ./odoo/odoo.py setup_pg
```

Al finalizar, Odoo estará listo para ser usado. El símbolo `~` es un atajo para su directorio raíz (por ejemplo, `/home/odoo`). La opción `git -b 8.0` explícitamente solicita descargar la rama 8.0 de Odoo. En el momento de escribir éste libro, esto es redundante, ya que 8.0 es la rama predeterminada, pero ésto puede cambiar, lo que hará más flexible lo aquí descrito.

Para iniciar una instancia del servidor Odoo, simplemente ejecute `odoo.py`:

```
$ ~/odoo-dev/odoo/odoo.py
```

De forma predeterminada, las instancias de Odoo escuchan a través del puerto 8069, si abre en su navegador la siguiente dirección `http://<server-address>:8069` se llegará a la instancia de Odoo. Cuando se accede por primera vez, se mostrará un asistente para crear una nueva base de datos, como se muestra en la siguiente imagen:

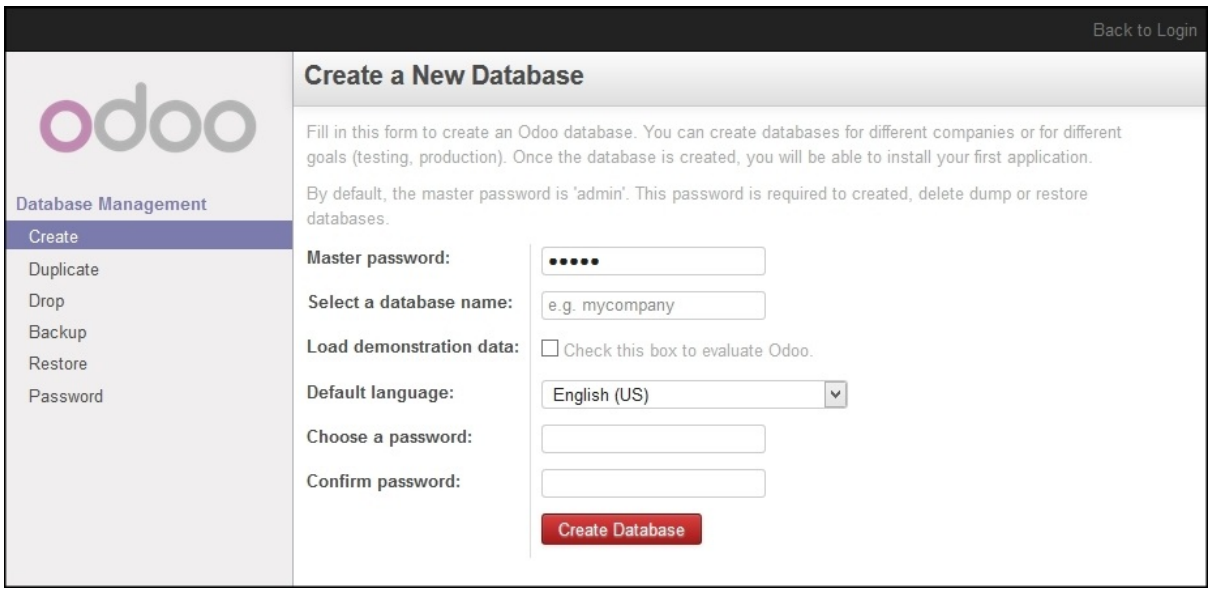

Figura 2.1: Gráfico 1.1 - Vista Crear una Nueva Base de datos

Pero aprenderá como inicializar bases de datos nuevas desde la línea de comando, ahora presione *Ctrl + C* para detener el servidor y volver a la línea de comandos.

Inicializar una base de datos nueva en Odoo

Para poder crear una base de datos nueva, su usuario debe ser un superusuario de PostgreSQL. Lo siguiente hace esto por usted `./odoo.py setup_pg`; de lo contrario use el siguiente comando para crear un superusuario PostgreSQL para el usuario Unix actual:

```
$ sudo createuser --superuser $(whoami)
```

Para crear una base de datos nueva use el comando `createdb`. Cree la base de datos `v8dev`:

```
$ createdb v8dev
```

Para inicializar ésta base de datos con el esquema de datos de Odoo debe ejecutar Odoo en la base de datos vacía usando la opción `-d`:

```
$ ~/odoo-dev/odoo/odoo.py -d v8dev
```

Tomará unos minutos inicializar la base de datos `v8dev`, y terminará con un mensaje de log INFO **Modules loaded**. Luego el servidor estará listo para atender las peticiones del cliente.

Por defecto, éste método inicializará la base de datos con los datos de demostración, lo cual frecuentemente es útil en bases de datos de desarrollo. Para inicializar una base de datos sin los datos de demostración, agregue la siguiente opción al comando anterior: `--without-demo-data=all`.

Para mostrar la pantalla de acceso abra en un navegador web `http://<server-name>:8069`. Si no conoce el nombre de su servidor, escriba el comando `hostname` en la terminal para averiguarlo, o el comando `ifconfig` para conocer la dirección IP.

Si está alojando Odoo en una máquina virtual probablemente necesite hacer algunas configuraciones de red para poder usarlo como servidor. La solución más simple es cambiar el tipo de red de la VM de NAT a Bridged. Con esto, en vez de compartir la dirección IP del equipo, la VM huésped tendrá su propia dirección IP. También es

posible usar NAT, pero esto requiere que configure el enrutamiento de puerto, así su sistema sabrá que algunos puertos, como el 8069, deben ser controlados por la VM. En caso de algún problema, con suerte estos detalles puedan ayudarle a encontrar ayuda en la documentación del software de virtualización de su preferencia.

La cuenta de usuario predeterminada es `admin` con la contraseña `admin`. Una vez acceda se mostrará el menú **Configuración**, revelando los módulos instalados. Elimine el filtro de **Instalado** y podrá ver e instalar cualquiera de los módulos oficiales.

En cualquier momento que desee detener la instancia del servidor Odoo y volver a la línea de comando, presione *Ctrl + C*. En consola, presiona la tecla de flecha Arriba para mostrar el comando anterior ejecutado, esta es una forma rápida de iniciar Odoo con las mismas opciones. Notará que *Ctrl + C* seguido de la flecha Arriba y *Enter* es una combinación frecuentemente usada para reiniciar el servidor Odoo durante el desarrollo.

Gestionar la base de datos

Ha visto como crear e inicializar bases de datos nuevas en Odoo desde la línea de comando. Existen más comandos que valen la pena conocer para gestionar bases de datos.

Ya sabe como usar el comando `createdb` para crear una base de datos vacía, pero también puede crear una base de datos copiando una existente, usando la opción `--template`.

Asegúrese que su instancia de Odoo este detenida y no tenga otra conexión abierta con la base de datos `v8dev` creada anteriormente, y ejecute:

```
$ createdb --template=v8dev v8test
```

De hecho, cada vez que se crea una base de datos, es usada una plantilla. Si no se especifica ninguna, se usa una predefinida llamada `template1`.

Para listar las bases de datos existentes en su sistema use la utilidad `psql` de PostgreSQL con la opción `-l`:

```
$ psql -l
```

Al ejecutar esto se debe listar las dos bases de datos creadas hasta los momentos: `v8dev` y `v8test`. La lista también mostrará la codificación usada en cada base de datos. La codificación predeterminada es UTF8, la cual es necesaria para las bases de datos Odoo.

Para eliminar una base de datos que ya no necesite (o necesita crear nuevamente), use el comando `dropdb`:

```
$ dropdb v8test
```

Ahora ya conoce lo básico para trabajar con varias bases de datos. Para aprender más sobre PostgresSQL, puede encontrar la documentación oficial en http://www.postgresql.org/docs/

> **Advertencia:** *Eliminar una base de datos destruirá los datos de forma irrevocable. Tenga cuidado al ejecutar esta acción y mantenga siempre respaldos de sus bases de datos.*

Unas palabras sobre las versiones de Odoo

A la fecha de publicación, la última versión estable de Odoo es la 8, marcada en GitHub como branch `8.0`. Ésta es la versión con la que se trabajará a lo largo de éste libro.

Es importante saber que las bases de datos de Odoo son incompatibles entre versiones principales de Odoo. Esto significa que si ejecuta un servidor Odoo 8 contra una base de datos Odoo/OpenERP 7, no funcionará. Es necesario un trabajo de migración significativo para que una base de datos pueda ser usada con una versión más reciente del producto.

Lo mismo pasa con los módulos: como regla general un módulo desarrollado para una versión más reciente de Odoo no funcionará con otras versiones. Cuando descargue módulos desde la Web desarrollados por la comunidad, asegúrese que estén dirigidos a la versión de Odoo que esté usando.

Por otro lado, los lanzamientos principales (7.0, 8.0) reciben actualizaciones frecuentes, pero en su mayoría deberán ser correcciones de fallos. Tiene la garantía de ser "estables para la API", lo que significa que las estructuras del modelo de datos y los identificadores de vista de los elementos se mantendrán estables. Esto es importante porque significa que no habrá riesgo de estropear los módulos personalizados por causa de cambios incompatibles en los módulos base.

Sea consciente que la versión en la rama `master` se convertirá en la próxima versión principal estable, pero hasta entonces no será "estable para la API" y no debe usarla para construir módulos personalizados. Hacer esto es como moverse en arena movediza: no hay formar de asegurar cuando algún cambio introducido hará que su módulo falle.

Más opciones de configuración del servidor

El servidor Odoo soporta unas pocas opciones más. Es posible verificar todas las opciones disponibles con la opción `--help`:

```
$ ./odoo.py --help
```

Vale la pena tener una idea general de las más importantes.

Archivos de configuración del servidor Odoo

La mayoría de las opciones pueden ser guardadas en un archivo de configuración. De forma predeterminada, Odoo usará el archivo `.openerp-serverrc` en su directorio home. Convenientemente, existe una opción `--save` para guardar la instancia actual de configuración dentro de ese archivo, ejecute el siguiente comando:

```
$ ~/odoo-dev/odoo/odoo.py --save --stop-after-init
```

Aquí también se usa la opción `--stop-after-init`, para que el servidor se detenga al finalizar las acciones. Ésta opción es usada frecuentemente para ejecutar pruebas y solicitar la ejecución de actualización de un módulo para verificar que se instala correctamente.

Ahora se puede inspeccionar lo que se guardó en este archivo de configuración, ejecutando el siguiente comando:

```
$ more ~/.openerp_serverrc
```

Esto mostrará todas las opciones de configuración disponibles con sus valores predeterminados. La edición se hará efectiva la próxima vez que inicie una instancia de Odoo. Escriba `q` para salir y retornar a la línea de comandos.

También es posible seleccionar un archivo específico de configuración, usando la opción `--conf=<filepath>`. Los archivos de configuración no necesitan tener todas las opciones de configuración que ha visto hasta ahora. Solo es necesario que estén aquellas opciones que cambian los valores predeterminados.

Cambiar el puerto de escucha

El comando `--xmlrpc-server=<port>` permite cambiar el puerto predeterminado 8069 desde donde la instancia del servidor escucha las peticiones. Esto puede ser usado para ejecutar más de una instancia al mismo tiempo, en el mismo servidor.

Intente esto. Abra dos ventanas de la terminal. En la primera ejecute:

```
$ ~/odoo-dev/odoo.py --xmlrpc-port=8070
```

y en la otra ejecute:

```
$ ~/odoo-dev/odoo.py --xmlrpc-port=8071
```

Y allí lo tiene: dos instancias de Odoo en el mismo servidor escuchando a través de diferentes puertos. Las dos instancias pueden ser usadas en la misma o en diferentes base de datos. Y ambas pueden ejecutar versiones iguales o diferentes de Odoo.

Registro

La opción `--log-level` permite configurar el nivel de detalle del registro. Esto puede ser muy útil para entender lo que esta pasando en el servidor. Por ejemplo, para habilitar el nivel de registro de depuración utilice: `--log-level=debug`

Los siguientes niveles de registro pueden ser particularmente interesantes:

- `debug_sql` para inspeccionar el SQL generado por el servidor.
- `debug_rpc` para detallar las peticiones recibidas por el servidor.
- `debug_rpc` para detallar las respuestas enviadas por el servidor.

La salida del registro es enviada de forma predeterminada a la salida estándar (la terminal), pero puede ser dirigida a un archivo de registro con la opción `--logfile=<filepath>`.

Finalmente, la opción `--debug` llamará al *depurador Python* (`pdb`) cuando aparezca una excepción. Es útil hacer un análisis post-mortem de un error del servidor. Note que esto no tiene ningún efecto en el nivel de detalle del registro. Se pueden encontrar más detalles sobre los comandos del depurador de Python aquí: https://docs.python.org/2/library/pdb.html#debugger-commands.

2.1.3 Desarrollar desde la estación de trabajo

Puede ejecutar Odoo con un sistema Debian/Ubuntu, en una máquina virtual local o en un servidor remoto. Pero posiblemente prefiera hacer el trabajo de desarrollo en su estación de trabajo personal, usando su editor de texto o IDE favorito.

Éste puede ser el caso para las personas que desarrollan en estaciones de trabajo con Windows. Pero puede también ser el caso para las personas que usan Linux y necesitan trabajar en un servidor Odoo desde una red local.

Una solución para esto es habilitar el uso compartido de archivos en el servidor Odoo, así los archivos son fáciles de editar desde su estación de trabajo. Para las operaciones del servidor Odoo, como reiniciar el servidor, es posible usar un intérprete de comando SSH (como PUTTY en Windows) junto a su editor favorito.

Usar un editor de texto Linux

Tarde o temprano, será necesario editar archivos desde la línea de comandos. En muchos sistemas Debian el editor de texto predeterminado es vi. Si no se siente a gusto con éste, puede usar una alternativa más amigable. En sistemas Ubuntu el editor de texto predeterminado es `nano`. Puede que prefiera usar éste ya que es más fácil de usar. En caso que no esté disponible en su servidor, puede instalarlo, ejecutando el siguiente comando:

```
$ sudo apt-get install nano
```

En las siguientes secciones se asumirá como el editor de preferencia. Si prefiere cualquier otro editor, siéntase libre de adaptar los comandos de acuerdo a su elección.

Instalar y configurar Samba

El proyecto Samba proporciona a Linux servicios para compartir archivos compatibles con sistemas Microsoft Windows. Se puede instalar en el servidor Debian/Ubuntu, ejecutando el siguiente comando:

```
$ sudo apt-get install samba samba-common-bin
```

El paquete `samba` instala el servicio para compartir archivos y el paquete `samba-common-bin` es necesario para la herramienta `smbpasswd`. De forma predeterminada los usuarios autorizados para acceder a los archivos compartidos necesitan ser registrados. Es necesario registrar el usuario `odoo` y asignarle una contraseña para su acceso a los archivos compartidos, ejecutando el siguiente comando:

```
$ sudo smbpasswd -a odoo
```

Después de esto el usuario `odoo` podrá acceder a un recurso compartido de archivos para su directorio *home*, pero será de solo lectura. Se requiere el acceso a escritura, así que es necesario editar los archivos de configuración de Samba para cambiar eso, ejecutando el siguiente comando:

```
$ sudo nano /etc/samba/smb.conf
```

En el archivo de configuración, busque la sección `[homes]`. Edite las líneas de configuración para que sean iguales a los siguientes ajustes:

```
[homes]
   comment = Home Directories
   browseable = yes
   read only = no
   create mask = 0640
   directory mask = 0750
```

Para que estos cambios en la configuración tengan efecto, reinicie el servicio, ejecutando el siguiente comando:

```
$ sudo /etc/init.d/smbd restart
```

Habilitar las herramientas técnicas

Odoo incluye algunas herramientas que son muy útiles para las personas que desarrollan, y usted hará uso de estas a lo largo del libro. Estas son las Características Técnicas y el Modo de Desarrollo.

Estas están deshabilitadas de forma predeterminada, así que aprenderá como habilitarlas.

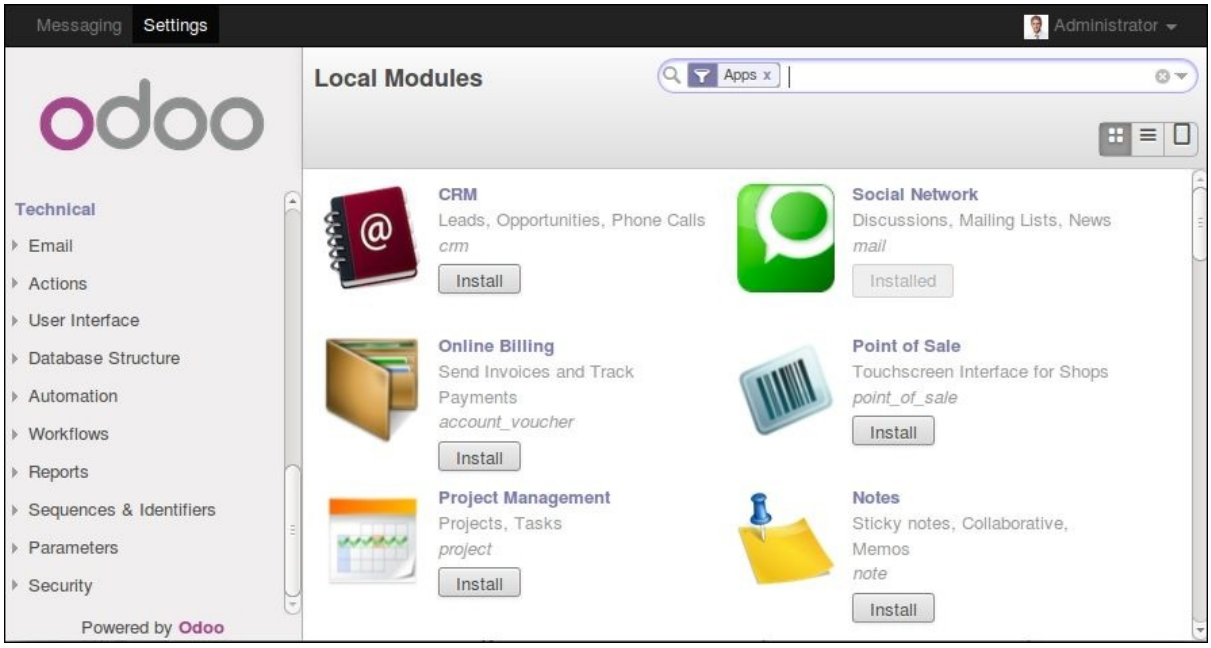

Figura 2.2: Gráfico 1.2 - Características Técnicas de Odoo

Activar las Características Técnicas

Las Características Técnicas proporcionan herramientas avanzadas de configuración del servidor.

Estas están deshabilitadas de forma predeterminada, y para habilitarlas, es necesario acceder con el usuario Administrador. En el menú **Configuración**, seleccione **Usuarios** y edite el usuario Administrador. En la pestaña **Derechos de Acceso**, encontrará una casilla de selección de **Características Técnicas**. Seleccione esa casilla y guarde los cambios.

Ahora es necesario recargar la página en el navegador web. Deberá poder ver en el menú **Configuraciones** una nueva sección **Técnico** que da acceso a lo interno del servidor Odoo.

La opción del menú **Técnico** permite inspeccionar y editar todas las configuraciones de Odoo almacenadas en la base de datos, desde la interfaz de usuario, a la seguridad y otros parámetros del sistema. Aprenderá más sobre esto a lo largo del libro.

Activar el modo de Desarrollo

El modo de Desarrollo habilita una caja de selección cerca de la parte superior de la ventana Odoo, haciendo accesible algunas opciones de configuración avanzadas en toda la aplicación. También deshabilita la modificación del código JavaScript y CSS usado por el cliente web, haciendo más fácil la depuración del comportamiento del lado del cliente.

Para habilitarlo, abra el menú desplegable en la esquina superior derecha de la ventana del navegador, en el nombre de usuario, y seleccione la opción **Acerca de Odoo**. En la ventana de dialogo **Acerca de**, haga clic sobre el botón **Activar modo desarrollador** en la esquina superior derecha.

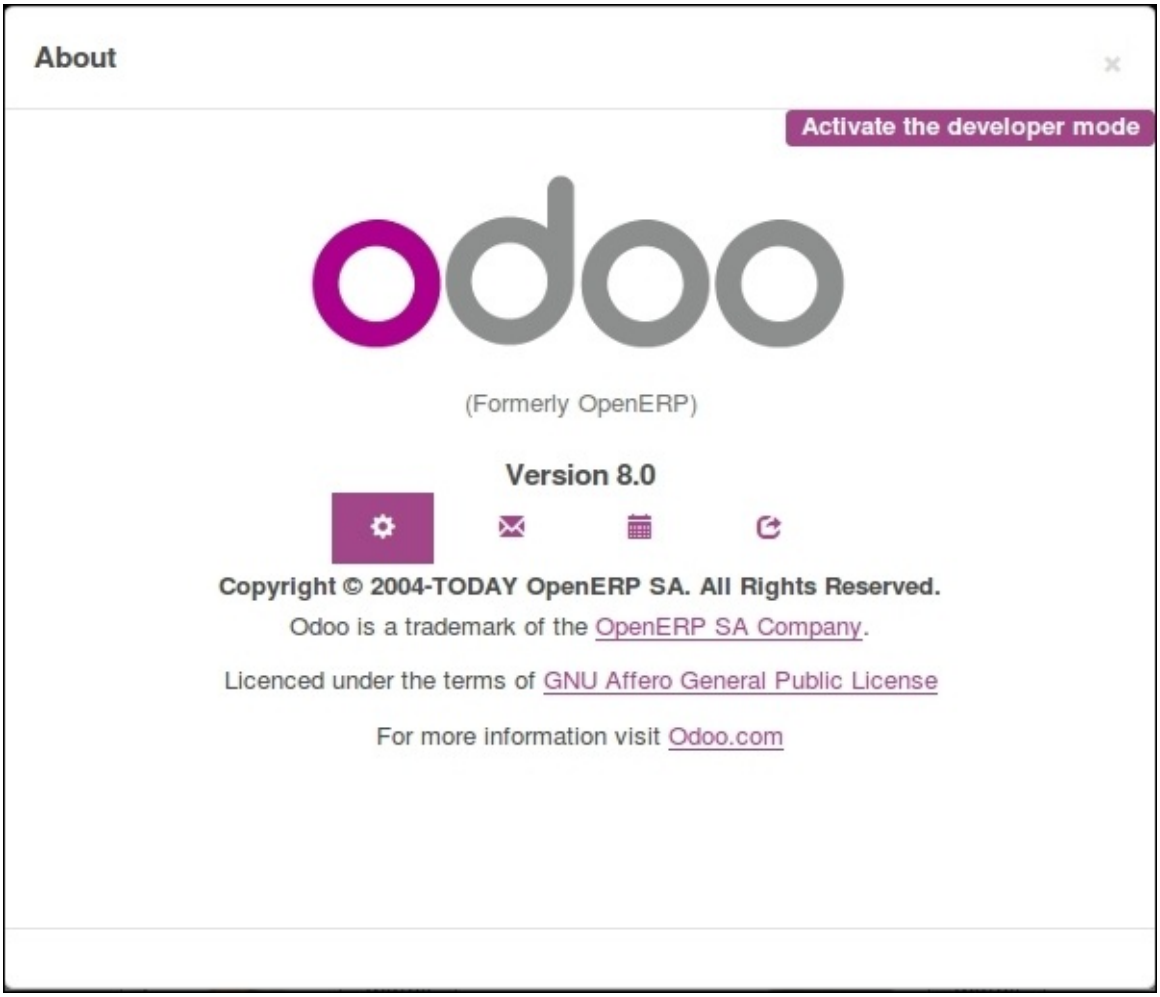

Figura 2.3: Gráfico 1.3 - Activar Modo de Desarrollo en Odoo

Luego de esto, verá una caja de selección **Depurar Vista** en la parte superior izquierda del área actual del formulario.

2.1.4 Instalar módulos de terceras partes

Hacer que nuevos módulos estén disponibles en una instancia de Odoo para que puedan ser instalados es algo que puede resultar confuso para las personas nuevas. Pero no necesariamente tiene que ser así, así que a continuación se desmitificará esta suposición.

Encontrar módulos de la comunidad

Existen muchos módulos para Odoo disponibles en Internet. El sitio web https://www.odoo.com/apps es un catalogo de módulos que pueden ser descargados e instalados. La **Odoo Community Association (OCA)** coordina las contribuciones de la comunidad y mantiene unos pocos repositorios en GitHub, en https://github.com/OCA.

Para agregar un módulo a la instalación de Odoo puede simplemente copiarlo dentro del directorio de complementos, junto a los módulos oficiales. En este caso, el directorio de complementos está en `~/odoo-dev/odoo/addons/`. Ésta puede que no sea la mejor opción para Ud., debido a que su instalación está basada en una versión controlada por el repositorio, y querrá tenerla sincronizada con el repositorio de GitHub.

Afortunadamente, es posible usar ubicaciones adicionales para los módulos, por lo que se puede tener los módulos personalizados en un directorio diferente, sin mezclarlos con los complementos oficiales.

Como ejemplo, se descargará el proyecto `department` de OCA y sus módulos se harán disponibles en la instalación de Odoo. Éste proyecto es un conjunto de módulos muy simples que agregan un campo Departamento en muchos formularios, como en el de Proyectos u Oportunidades de CRM.

Para obtener el código fuente desde GitHub:

```
$ cd ~/odoo-dev
$ git clone https://github.com/OCA/department.git -b 8.0
```

Se usó la opción `-b` para asegurar que se descargan los módulos de la versión 8.0.

Pero debido a que en el momento de escribir esto la versión 8.0 en la rama predeterminada del proyecto la opción `-b` podría haber sido omitida.

Luego, se tendrá un directorio `/department` nuevo junto al directorio `/odoo`, que contendrá los módulos. Ahora es necesario hacer saber a Odoo sobre este nuevo directorio.

Configurar la ruta de complementos

El servidor Odoo tiene una opción llamada `addons-path` que define donde buscar los módulo. De forma predeterminada este apunta al directorio `/addons` del servidor Odoo que se esta ejecutando.

Afortunadamente, es posible asignar no uno, sino una lista de directorios donde se pueden encontrar los módulos. Esto permite mantener los módulos personalizados en un directorio diferente, sin mezclarlos con los complementos oficiales. Se ejecutará el servidor con una ruta de complemento incluyendo el nuevo directorio de módulos:

```
$ cd ~/odoo-dev/odoo
$ ./odoo.py -d v8dev --addons-path="/department,./addons"
```

Si se observa con cuidado el registro del servidor notará una línea reportando la ruta de los complementos en uso: **INFO ? Openerp: addons paths:** (...). Confirmando que la instancia contiene su directorio `department`.

Actualizar la lista de módulos

Es necesario pedirle a Odoo que actualice su lista de módulos antes que estos módulos nuevos estén disponibles para ser instalados.

Para esto es necesario habilitar el menú **Técnico**, debido a que esta provee la opción de menú **Actualizar Lista de Módulos**. Esta puede ser encontrada en la sección **Módulos** en el menú **Configuración**.

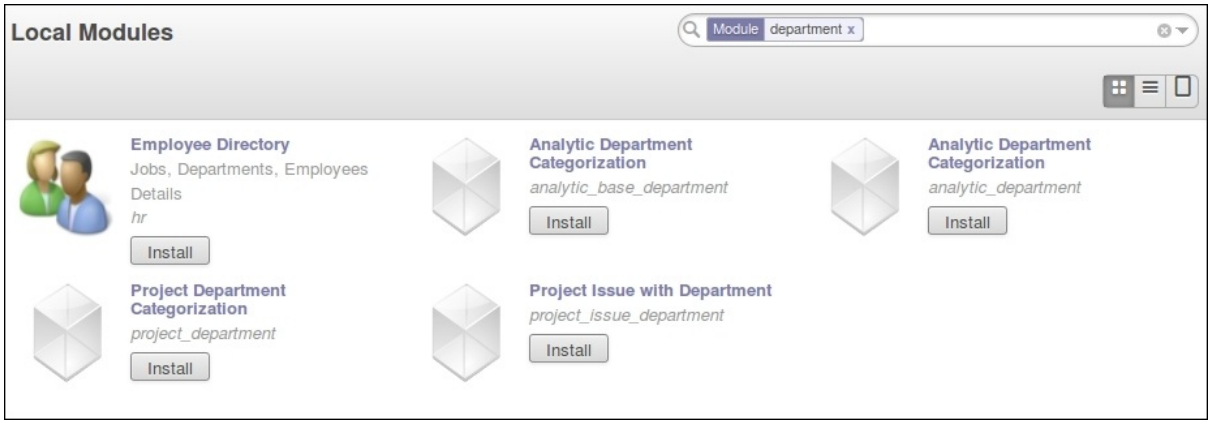

Figura 2.4: Gráfico 1.4 - Confirmar que la instancia Odoo reconoce el directorio 'department'

Luego de ejecutar la actualización de la lista de módulos se puede confirmar que los módulos nuevos están disponibles para ser instalados. En la lista de **Módulos Locales**, quite el filtro de `Aplicaciones en línea` y busque `department`. Debería poder ver los nuevos módulos disponibles.

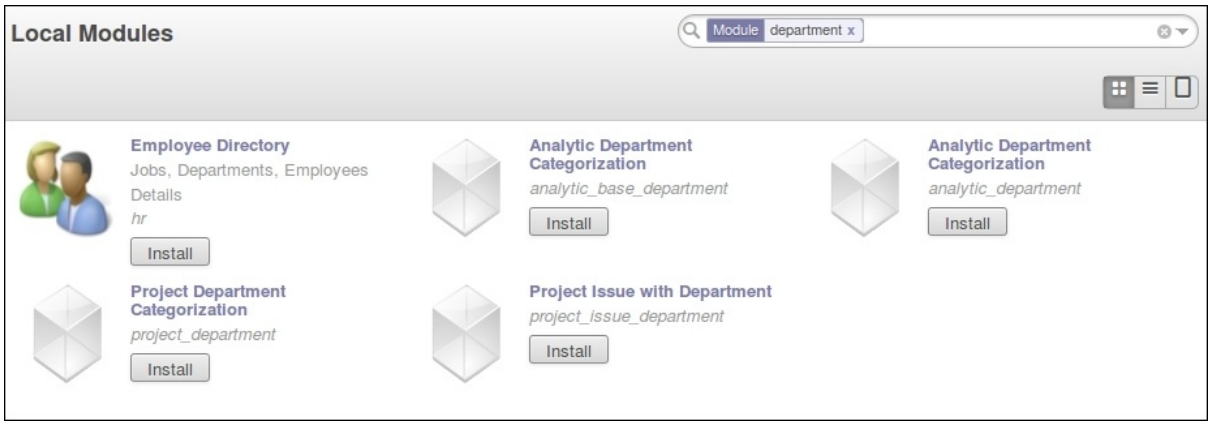

Figura 2.5: Gráfico 1.5 - Actualizar Lista de Módulos

2.1.5 Resumen

En este capítulo, aprendió como configurar un sistema Debian para alojar Odoo e instalarlo desde GitHub. También aprendió como crear bases de datos en Odoo y ejecutar instancias Odoo. Para permitir que las personas que desarrollan usen sus herramientas favoritas en sus estaciones de trabajo, se explicó como configurar archivos compartidos en el servidor Odoo.

En estos momentos debería tener un ambiente Odoo completamente funcional para trabajar, y sentirse a gusto con el manejo de bases de datos e instancias.

Con esto claro, es momento de ir directo a la acción. En el próximo capítulo se creará el primer módulo Odoo y entenderá los elementos principales involucrados.

¡Comience!

2.2 Capítulo 2 - Primera aplicación

2.2.1 Construyendo su primera aplicación con Odoo

Desarrollar en Odoo la mayoría de las veces significa crear sus propios módulos. En este capítulo, se creará la primera aplicación con Odoo, y se aprenderán los pasos necesarios para habilitarlas e instalarlas en Odoo.

Inspirados del notable proyecto todomvc.com, se desarrollará una simple aplicación para el registro de cosas por hacer. Deberá permitir agregar nuevas tareas, marcarlas como culminadas, y finalmente borrar de la lista todas las tareas finalizadas.

Aprenderá como Odoo sigue una arquitectura MVC, y recorrerá las siguientes capas durante la implementación de la aplicación:

- El **modelo**, define la estructura de los datos.
- La **vista**, describe la interfaz con el usuario.
- El **controlador**, soporta la lógica de negocio de la aplicación.

La capa modelo es definida por objetos Python cuyos datos son almacenados en una base de datos PostgreSQL. El mapeo de la base de datos es gestionado automáticamente por Odoo, y el mecanismo responsable por esto es el **modelo objeto relacional, (ORM - object relational model)**.

La capa vista describe la interfaz con el usuario. Las vistas son definidas usando XML, las cuales son usadas por el marco de trabajo (framework) del cliente web para generar vistas HTML de datos.

Las vistas del cliente web ejecutan acciones de datos persistentes a través de la interacción con el servidor ORM. Estas pueden ser operaciones básicas como escribir o eliminar, pero pueden también invocar métodos definidos en los objetos Python del ORM, ejecutando lógica de negocio más compleja. A esto es a lo que se refiere cuando se habla de la capa modelo.

Nota: Note que el concepto de controlador mencionado aquí es diferente al desarrollo de controladores web de Odoo. Aquellos son programas finales a los cuales las páginas web pueden llamar para ejecutar acciones.

Con este enfoque, podrá ser capaz de aprender gradualmente sobre los bloques básicos de desarrollo que conforman una aplicación y experimentar el proceso iterativo del desarrollo de módulos en Odoo desde cero.

2.2.2 Entender las aplicaciones y los módulos

Es común escuchar hablar sobre los módulos y las aplicaciones en Odoo. Pero, ¿Cual es exactamente la diferencia entre un módulo y una aplicación? Los **módulos** son bloques para la construcción de las aplicaciones en Odoo. Un módulo puede agregar o modificar características en Odoo. Esto es soportado por un directorio que contiene un archivo de manifiesto o descriptor (llamado `__openerp__.py`) y el resto de los archivos que implementan sus características. A veces, los módulos pueden ser llamados "add-ons" (complementos). Las **aplicaciones** no son diferentes de los módulos regulares, pero funcionalmente, éstas proporcionan una característica central, alrededor de la cual otros módulos agregan características u opciones. Estas proveen los elementos base para un área funcional, como contabilidad o RRHH, sobre las cuales otros módulos agregan características. Por esto son resaltadas en el menú Apps de Odoo.

Modificar un módulo existente

En el ejemplo que sigue a continuación, creara un módulo nuevo con tan pocas dependencias como sea posible.

Sin embargo, este no es el caso típico. Lo más frecuente serán situaciones donde las modificaciones y extensiones son necesarias en un módulo existente para ajustarlo a casos de uso específicos.

La regla de oro dice que no debe cambiar módulos existentes modificándolos directamente. Esto es considerado una mala práctica. Especialmente cierto para los módulos oficiales proporcionados por Odoo. Hacer esto no permitirá una clara separación entre el módulo original y sus modificaciones, y hace difícil la actualización.

Por el contrario, debe crear módulos nuevos que sean aplicados encima de los módulos que requiere modificar, e implementar esos cambios. Esta es una de las principales fortalezas de Odoo: provee mecanismos de "herencia" que permiten a los módulos personalizados extender los módulos existentes, bien sean oficiales o de la comunidad. La herencia el posible en todos los niveles, modelo de datos, lógica de negocio, e interfaz con el usuario.

Ahora, creara un módulo nuevo completo, sin extender ningún módulo existente, para enfocarse en las diferentes partes y pasos involucrados en la creación de un módulo. Solo se dará una breve mirada a cada parte, ya que cada una será estudiada en detalle en los siguientes capítulos. Una vez este a gusto con la creación de un módulo nuevo, podrá sumergirse dentro de los mecanismos de herencia, los cuales serán estudiados en el siguiente capítulo.

Crear un módulo nuevo

Nuestro módulo será una aplicación muy simple para gestionar las tareas por hacer. Estas tareas tendrán un único campo de texto, para la descripción, y una casilla de verificación para marcarlas como culminadas. También tendrá un botón para limpiar la lista de tareas de todas aquellas finalizadas.

Estas especificaciones son muy simples, pero a medida que avance en el libro ira agregando gradualmente nuevas características, para hacer la aplicación más interesante.

Basta de charla, comience a escribir código y crear su nuevo módulo.

Siguiendo las instrucciones del Capítulo 1, Comenzando con Odoo, debe tener el servidor Odoo en `/odoo-dev/odoo/`. Para mantener las cosas ordenadas, creare un directorio junto a este para guardar sus propios módulos:

```
$ mkdir ~/odoo-dev/custom-addons
```

Un módulo en Odoo es un directorio que contiene un archivo descriptor `__openerp__.py`. Esto es una herencia de cuando Odoo se llamaba OpenERP, y en el futuro se espera se convierta en `__odoo__.py`. Es necesario que pueda ser importado desde Python, por lo que debe tener un archivo `__init__.py`.

El nombre del directorio del módulo será su nombre técnico. Usara `todo_app` para el nombre. El nombre técnico debe ser un identificador Python valido: debe comenzar con una letra y puede contener letras, números y el carácter especial guión bajo. Los siguientes comandos crean el directorio del módulo y el archivo vacío `__init__.py` dentro de este:

```
$ mkdir ~/odoo-dev/custom-addons/todo_app
$ touch ~/odoo-dev/custom-addons/todo_app/__init__.py
```

Luego necesita crear el archivo descriptor. Debe contener únicamente un diccionario Python y puede contener alrededor de una docena de atributos, de los cuales solo el atributo `name` es obligatorio. Son recomendados los atributos `description`, para una descripción más larga, y `author`. Ahora agregue un archivo `__openerp__.py` junto al archivo `__init__.py` con el siguiente contenido:

```
{
    'name': 'To-Do Application',
    'description': 'Manage your personal Tasks with this module.',
    'author': 'Daniel Reis',
    'depends': ['mail'],
    'application': True,
}
```

El atributo `depends` puede tener una lista de otros módulos requeridos. Odoo los instalará automáticamente cuando este módulo sea instalado. No es un atributo obligatorio pero se recomienda tenerlo siempre. Si no es requerida alguna dependencia en particular, debería existir alguna dependencia a un módulo base especial. Debe tener cuidado de asegurarse que todas las dependencias sean explícitamente fijadas aquí, de otra forma el módulo podría fallar al instalar una base de datos vacía (debido a dependencias insatisfechas) o tener errores en la carga, si otros módulos necesarios son cargados después.

Para su aplicación, querrá que dependa del módulo `mail` debido a que este agrega el menú **Mensajería** en la parte superior de la ventana, y querrá incluir su nuevo menú de opciones allí.

Para precisar, escoja pocas claves del descriptor, pero en el mundo real es recomendable usar claves adicionales, ya que estas son relevantes para la app-store de Odoo:

- `summary`, muestra un subtitulo del módulo.
- `version`, de forma predeterminada, es 1.0. Se debe seguir las reglas de versionamiento semántico (para más detalles ver semver.org).
- `license`, de forma predeterminada es AGPL-3.
- `website`, es una URL para encontrar más información sobre el módulo. Esta puede servir a las personas a encontrar documentación, informar sobre errores o hacer sugerencias.
- `category`, es la categoría funcional del módulo, la cual de forma predeterminada es Sin Categoría. La lista de las categorías existentes puede encontrarse en el formato de Grupos (Configuraciones > Usuarios > menú Grupos), en la lista desplegable del campo Aplicación.

Estos descriptores también están disponibles:

- `installable`, de forma predeterminada es `True`, pero puede ser fijada `False` para deshabilitar el módulo.
- `auto_install`, si esta fijada en `True` este módulo es automáticamente instalado si todas las dependencias han sido instaladas. Esto es usado en módulos asociados.

Desde Odoo 8.0, en vez de la clave `description` podrá usar un archivo `README.rst` o `README.md` en el directorio raíz del módulo.

Agregar el módulo a la ruta de complementos

Ahora que tiene un módulo nuevo, incluso si es muy simple, querrá que esté disponible en Odoo. Para esto, debe asegurarse que el directorio que contiene el módulo sea parte de la ruta de complementos addons. Y luego tiene que actualizar la lista de módulos de Odoo.

Ambas operaciones han sido explicadas en detalle en el capítulo anterior, pero a continuación se presenta un resumen de lo necesario.

Posiciónese dentro del directorio de trabajo e inicia el servidor con la configuración de la ruta de complementos o addons:

```
$ cd ~/odoo-dev
$ odoo/odoo.py -d v8dev --addons-path="custom-addons,odoo/addons" --save
```

La opción `--save` guarda la configuración usada en un archivo de configuración. Esto evita repetirlo cada vez que el servidor es iniciado: simplemente ejecute `./odoo.py` y serán ejecutadas las últimas opciones guardadas.

Mira detenidamente en el registro del servidor. Debería haber una línea **INFO ? openerp: addons paths:** (...), y debería incluir su directorio `custom-addons`.

Recuerde incluir cualquier otro directorio que pueda estar usando. Por ejemplo, si siguió las instrucciones del último capítulo para instalar el repositorio `department`, puede querer incluirlo y usar la opción:

```
--addons-path="custom-addons,department,odoo/addons"
```

Ahora haga que Odoo sepa de los módulos nuevos que ha incluido.

Para esto, En la sección **Módulos** del menú **Configuración**, seleccione la opción **Actualizar lista de módulos**. Esto actualizará la lista de módulos agregando cualquier módulo incluido desde la última actualización de la lista. Recuerde que necesita habilitar las Características Técnicas para que esta opción sea visible. Esto se logra seleccionando la caja de verificación de **Características técnicas** para su cuenta de usuario.

Instalar el módulo nuevo

La opción **Módulos locales** le muestra la lista de módulos disponibles. De forma predeterminada solo muestra los módulos de **Aplicaciones en línea**. Debido a que crea un módulo de aplicación no es necesario remover este filtro. Escriba “todo” en la campo de búsqueda y debe ver su módulo nuevo, listo para ser instalado.

Haga clic en el botón **Instalar** y listo!

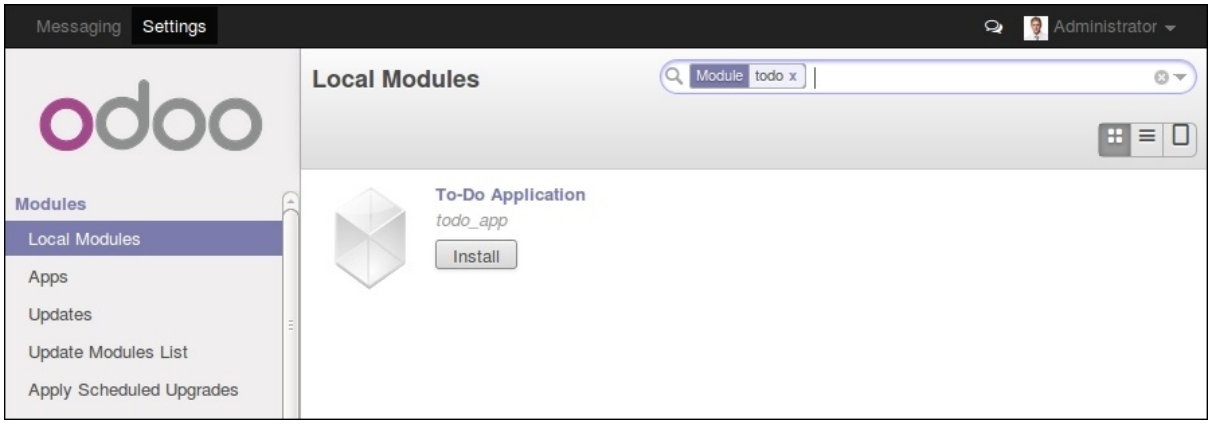

Figura 2.6: Gráfico 2.1 - Instalar nuevo módulo 'todo_app'

Actualizar un módulo

El desarrollo de un módulo es un proceso iterativo, y puede querer que los cambios hechos en los archivos fuente sean aplicados y estén visibles en Odoo.

En la mayoría de los casos esto es hecho a través de la actualización del módulo: busque el módulo en la lista de Módulos Locales y, ya que está instalado, debe poder ver el botón Actualizar.

De cualquier forma, cuando los cambios realizados son en el código Python, la actualización puede no tener ningún efecto. En este caso es necesario reiniciar la aplicación en el servidor.

En algunos casos, si el módulo ha sido modificado tanto en los archivos de datos como en el código Python, pueden ser necesarias ambas operaciones. Este es un punto común de confusión para las personas que se inician en el desarrollo con Odoo.

Pero afortunadamente, existe una mejor forma. La forma más simple y rápida para hacer efectivos todos los cambios en su módulo es detener (*Ctrl* + *C*) y reiniciar el proceso del servidor que requiere que sus módulos sean actualizados en la base de datos de trabajo.

Para hacer que el servidor inicie la actualización del módulo `todo_app` en la base de datos `v8dev`, usara:

```
$ ./odoo.py -d v8dev -u todo_app
```

La opción `-u` (o `--update` en su forma larga) requiere la opción `-d` y acepta una lista separada por comas de módulos para actualizar. Por ejemplo, podrá usar: `-u todo_app,mail`.

En el momento en que necesite actualizar un módulo en proceso de desarrollo a lo largo del libro, la manera más segura de hacerlo es ir a una ventana de terminal donde se este ejecutando Odoo, detener el servidor, y reiniciarlo con el comando visto anteriormente. Usualmente será suficiente con presionar la tecla de flecha arriba, esto debería devolver el último comando usado para iniciar el servidor.

Desafortunadamente, la actualización de la lista de módulos y la desinstalación son acciones que no están disponibles a través de la línea de comandos. Esto debe ser realizado a través de la interfaz web, en el menú Configuraciones.

Crear un modelo de aplicación

Ahora que Odoo sabe sobre la disponibilidad de su módulo nuevo, comience a agregarle un modelo simple.

Los modelos describen los objetos de negocio, como una oportunidad, una orden de venta, o un socio (cliente, proveedor, etc). Un modelo tiene una lista de atributos y también puede definir su negocio específico.

Los modelos son implementados usando clases Python derivadas de una plantilla de clase de Odoo. Estos son traducidos directamente a objetos de base de datos, y Odoo se encarga de esto automáticamente cuando el módulo es instalado o actualizado.

Algunas personas consideran como buena práctica mantener los archivos Python para los modelos dentro de un subdirectorio. Por simplicidad no seguirá esta sugerencia, así que va a crear un archivo `todo_model.py` en el directorio raíz del módulo `todo_app`.

Agregar el siguiente contenido:

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

class TodoTask(models.Model):
    _name = 'todo.task'
    name = fields.Char('Description', required=True)
    is_done = fields.Boolean('Done?')
    active = fields.Boolean('Active?', default=True)
```

La primera línea es un marcador especial que le dice al interprete de Python que ese archivo es UTF-8, por lo que puede manejar y esperarse caracteres non-ASCII. No usara ninguno, pero es más seguro usarlo.

La segunda línea hace que estén disponibles los modelos y los objetos campos del núcleo de Odoo.

la tercera línea declara su nuevo modelo. Es una clase derivada de `models.Model`. La siguiente línea fija el atributo `_name` definiendo el identificador que será usado por Odoo para referirse a este modelo. Note que el nombre real de la clase Python no es significativo para los otros módulos de Odoo. El valor de `_name` es lo que será usado como identificador.

Observe que éstas y las siguientes líneas tienen una sangría. Si no conoce muy bien Python debe saber que esto es sumamente importante: la sangría define un bloque de código anidado, por lo tanto estas cuatro líneas deben tener la misma sangría.

Las últimas tres líneas definen los campos del modelo. Vale la pena señalar que `name` y `active` son nombres de campos especiales. De forma predeterminada Odoo usara el campo `name` como el título del registro cuando sea referenciado desde otros modelos. El campo `active` es usado para desactivar registros, y de forma predeterminada solo los registros activos son mostrados. Lo usara para quitar las tareas finalizadas sin eliminarlas definitivamente de la base de datos.

Todavía, este archivo, no es usado por el módulo. Debe decirle a Odoo que lo cargue con el módulo en el archivo `__init__.py`. Edite el archivo para agregar la siguiente línea:

```
from . import todo_model
```

Esto es todo. para que sus cambios tengan efecto el módulo debe ser actualizado. Encuentre la aplicación **To-Do** en **Módulos Locales** y haga clic en el botón **Actualizar**.

Ahora podrá revisar el modelo recién creado en el menú **Técnico**. Vaya a **Estructura de la Base de Datos > Modelos** y busque el modelo `todo.task` en la lista. Luego haga clic en este para ver su definición:

Si no hubo ningún problema, esto le confirmará que el modelo y sus campos fueron creados. Si hizo algunos cambios y no son reflejados, intente reiniciar el servidor, como fue descrito anteriormente, para obligar que todo el código Python sea cargado nuevamente.

También podrá ver algunos campos adicionales que no declarara. Estos son cinco campos reservados que Odoo agrega automáticamente a cualquier modelo. Son los siguientes:

- `id`: Este es el identificador único para cada registro en un modelo en particular.
- `create_date` y `create_uid`: Estos les indican cuando el registro fue creado y quien lo creó, respectivamente.
- `write_date` y `write_uid`: Estos les indican cuando fue la última vez que el registro fue modificado y quien lo modificó, respectivamente.

Agregar entradas al menú

Ahora que tiene un modelo en el cual almacenar sus datos, haga que este disponible en la interfaz con el usuario.

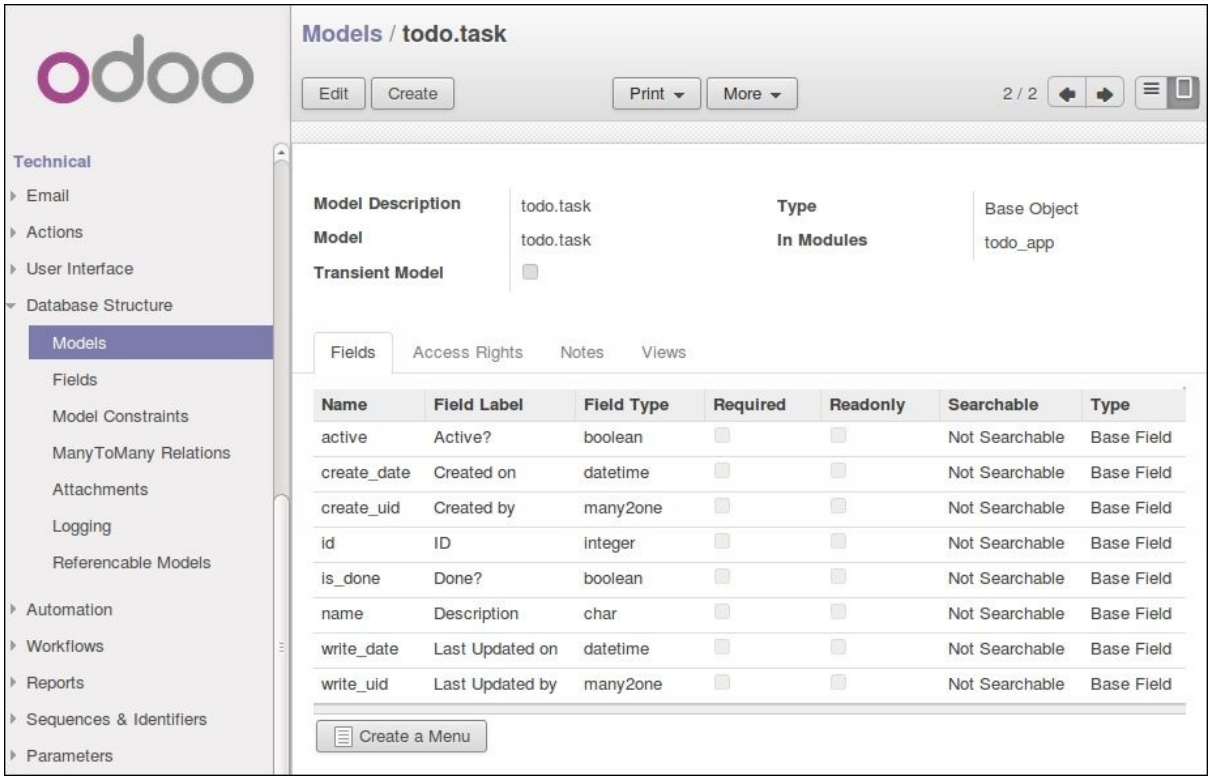

Figura 2.7: Gráfico 2.2 - Vista de Estructura de la Base de Datos de módulo 'todo_app'

Todo lo que necesita hacer es agregar una opción de menú para abrir el modelo de "To-do Task" para que pueda ser usado. Esto es realizado usando un archivo XML. Igual que en el caso de los modelos, algunas personas consideran como una buena practica mantener las definiciones de vistas en en un subdirectorio separado.

Creara un archivo nuevo `todo_view.xml` en el directorio raíz del módulo, y este tendrá la declaración de un ítem de menú y la acción ejecutada por este:

```
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
    <data>
        <!-- Action to open To-do Task list -->
        <act_window
            id="action_todo_task"
            name="To-do Task"
            res_model="todo.task"
            view_mode="tree,form"
        />
        <!-- Menu item to open To-do Task list -->
        <menuitem
            id="menu_todo_task"
            name="To-Do Tasks"
            parent="mail.mail_feeds"
            sequence="20"
            action="action_todo_task"
        />
    </data>
</openerp>
```

La interfaz con el usuario, incluidas las opciones del menú y las acciones, son almacenadas en tablas de la base de datos. El archivo XML es un archivo de datos usado para cargar esas definiciones dentro de la base de datos cuando el módulo es instalado o actualizado. Esto es un archivo de datos de Odoo, que describe dos registros para ser agregados a Odoo:

- El elemento <act_window> define una Acción de Ventana del lado del cliente para abrir el modelo todo.task definido en el archivo Python, con las vistas de árbol y formulario habilitadas, en ese orden.
- El elemento <menuitem> define un ítem de menú bajo el menú Mensajería (identificado por mail.mail_feeds), llamando a la acción action_todo_task, que fue definida anteriormente. el atributo sequence les deja fijar el orden de las opciones del menú.

Ahora necesita decirle al módulo que use el nuevo archivo de datos XML. Esto es hecho en el archivo __openerp__.py usando el atributo data. Este define la lista de archivos que son cargados por el módulo. Agregue este atributo al diccionario del descriptor:

```
'data' : ['todo_view.xml'],
```

Ahora necesita actualizar nuevamente el módulo para que estos cambios tengan efecto. Vaya al menú Mensajería y debe poder ver su nueva opción disponible.

Figura 2.8: Gráfico 2.3 - Agregar módulo 'todo_app' al menú de Odoo

Si hace clic en ella se abrirá un formulario generado automáticamente para su modelo, permitiendo agregar y modificar los registros.

Las vistas deben ser definidas por los modelos para ser expuestas a los usuarios, aunque Odoo es lo suficientemente amable para hacerlo automáticamente si no querrá, entonces podrá trabajar con su modelo, sin tener ningún formulario o vistas definidas aún.

Hasta ahora va bien. Mejorara ahora su interfaz gráfica. Intente las mejoras graduales que son mostradas en las secciones siguientes, haciendo actualizaciones frecuentes del módulo, y no tenga miedo de experimentar.

Truco: En caso que una actualización falle debido a un error en el XML, ¡no entre en pánico! Comente las últimas porciones de XML editadas, o elimine el archivo XML del __openerp__.py, y repita la actualización. El servidor debería iniciar correctamente. Luego lea detenidamente el mensaje de error en los registros del servidor - debería decirle donde está el problema.

Crear vistas - formulario, árbol y búsqueda

Como ha visto, si ninguna vista es definida, Odoo automáticamente generará vistas básicas para que puedas continuar. Pero seguramente le gustará definir sus propias vistas del módulo, así que eso es lo que hará.

Odoo soporta varios tipos de vistas, pero las tres principales son: list (lista, también llamada árbol), form (formulario), y search (búsqueda). Agregara un ejemplo de cada una a su módulo.

Todas las vistas son almacenadas en la base de datos, en el modelo ir.model.view. Para agregar una vista en un módulo, declaro un elemento <record> describiendo la vista en un archivo XML que será cargado dentro de la base de datos cuando el modelo sea instalado.

Creando una vista formulario

Edite el XML que recién ha creado para agregar el elemento `<record>` después de la apertura de la etiqueta `<data>`:

```
<record id="view_form_todo_task" model="ir.ui.view">
    <field name="name">To-do Task Form</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <form string="To-do Task">
            <field name="name"/>
            <field name="is_done"/>
            <field name="active" readonly="1"/>
        </form>
    </field>
</record>
```

Esto agregará un registro al modelo `ir.ui.view` con el identificador `view_form_todo_task`. Para el modelo la vista es `todo.task` y nombrada `To-do Task Form`. El nombre es solo para información, no tiene que ser único, pero debe permitir identificar fácilmente a que registro se refiere.

El atributo más importante es `arch`, que contiene la definición de la vista. Aquí se dice que es un formulario, y que contiene tres campos, y que decidió hacer al campo `active` de solo lectura.

Formatear como un documento de negocio

Lo anterior proporciona una vista de formulario básica, pero podrá hacer algunos cambios para mejorar su apariencia. Para los modelos de documentos Odoo tiene un estilo de presentación que asemeja una hoja de papel. El formulario contiene dos elementos: una `<head>`, que contiene botones de acción, y un `<sheet>`, que contiene los campos de datos:

```
<form>
    <header>
        <!-- Buttons go here-->
    </header>
    <sheet>
        <!-- Content goes here: -->
        <field name="name"/>
        <field name="is_done"/>
    </sheet>
</form>
```

Agregar botones de acción

Los formularios pueden tener botones que ejecuten acciones. Estos son capaces de desencadenar acciones de flujo de trabajo, ejecutar Acciones de Ventana, como abrir otro formulario, o ejecutar funciones Python definidas en el modelo.

Estos pueden ser colocados en cualquier parte dentro de un formulario, pero para formularios con estilo de documentos, el sitio recomendado es en la sección `<header>`.

Para su aplicación, agregara dos botones para ejecutar métodos del modelo `todo.task`:

```
<header>
    <button name="do_toggle_done" type="object" string="Toggle Done" class="oe_highlight" />
    <button name="do_clear_done" type="object" string="Clear All Done" />
</header>
```

Los atributos básicos para un botón son: `string` con el texto que se muestra en el botón, `type` que hace referencia al tipo de acción que ejecuta, y `name` que es el identificador para esa acción. El atributo `class` puede aplicar estilos CSS, como un HTML común.

Organizar formularios usando grupos

La etiqueta `<group>` permite organizar el contenido del formulario. Colocando los elementos `<group>` dentro de un elemento `<group>` crea una disposición de dos columnas dentro del grupo externo. Se recomienda que los elementos `Group` tengan un nombre para hacer más fácil su extensión en otros módulos.

Usara esto para mejorar la organización de su contenido. Cambio el contenido de `<sheet>` de su formulario:

```
<sheet>
    <group name="group_top">
        <group name="group_left">
            <field name="name"/>
        </group>
        <group name="group_right">
            <field name="is_done"/>
            <field name="active" readonly="1"/>
        </group>
    </group>
</sheet>
```

La vista de formulario completa

En este momento, su registro en `todo_view.xml` para la vista de formulario de `todo.task` debería lucir así:

```
<record id="view_form_todo_task" model="ir.ui.view">
    <field name="name">To-do Task Form</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <form>
            <header>
                <button name="do_toggle_done" type="object" string="Toggle Done" class="oe_highli
                <button name="do_clear_done" type="object" string="Clear All Done" />
            </header>
            <sheet>
                <group name="group_top">
                    <group name="group_left">
                        <field name="name"/>
                    </group>
                    <group name="group_right">
                        <field name="is_done"/>
                        <field name="active" readonly="1" />
                    </group>
                </group>
            </sheet>
        </form>
    </field>
</record>
```

Recuerde que para que los cambios tengan efecto en la base de datos de Odoo, es necesario actualizar el módulo. Para ver los cambio en el cliente web, es necesario volver a cargar el formulario: haciendo nuevamente clic en la opción de menú que abre el formulario, o volviendo a cargar la página en el navegador (*F5* en la mayoría de los navegadores).

Ahora, agregue la lógica de negocio para las acciones de los botones.

Agregar vistas de lista y búsqueda

Cuando un modelo se visualiza como una lista, se esta usando una vista `<tree>` Las vistas de árbol son capaces de mostrar líneas organizadas por jerarquía, pero la mayoría de las veces son usadas para desplegar listas planas.

Puede agregar la siguiente definición de una vista de árbol a `todo_view.xml`:

```
<record id="view_tree_todo_task" model="ir.ui.view">
    <field name="name">To-do Task Tree</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <tree colors="gray:is_done==True">
            <field name="name"/>
            <field name="is_done"/>
        </tree>
    </field>
</record>
```

Ha definido una lista con solo dos columnas, `name` y `is_done`. También agregue un toque extra: las líneas para las tareas finalizadas (`is_done==True`) son mostradas en color gris.

En la parte superior derecha de la lista Odoo muestra una campo de búsqueda. Los campos de búsqueda predefinidos y los filtros disponibles pueden ser predeterminados por una vista `<search>`.

Como lo hice anteriormente, agregara esto a `todo_view.xml`:

```
<record id="view_filter_todo_task" model="ir.ui.view">
    <field name="name">To-do Task Filter</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <search>
            <field name="name"/>
            <filter string="Not Done" domain="[('is_done','=',False)]"/>
            <filter string="Done" domain="[('is_done','!=',False)]"/>
        </search>
    </field>
</record>
```

Los elementos `<field>` definen campos que también son buscados cuando se escribe en el campo de búsqueda. Los elementos `<filter>` agregan condiciones predefinidas de filtro, usando la sintaxis de dominio que puede ser seleccionada por el usuario con un clic.

Agregar la lógica de negocio

Ahora agregara lógica a sus botones. Edite el archivo Python `todo_model.py` para agregar a la clase los métodos llamados por los botones.

Usara la API nueva introducida en Odoo 8.0. Para compatibilidad con versiones anteriores, de forma predeterminada Odoo espera la API anterior, por lo tanto para crear métodos usando la API nueva se necesitan en ellos decoradores Python. Primero necesita una declaración `import` al principio del archivo:

```
from openerp import models, fields, api
```

La acción del botón **Toggle Done** es bastante simple: solo cambia de estado (marca o desmarca) la señal **Is Done?**. La forma más simple para agregar la lógica a un registro, es usar el decorador `@api.one`. Aquí `self` representara un registro. Si la acción es llamada para un conjunto de registros, la API gestionara esto lanzando el método para cada uno de los registros.

Dentro de la clase `TodoTask` agregue:

```
@api.one
def do_toggle_done(self):
    self.is_done = not self.is_done
    return True
```

Como puede observar, simplemente modifica el campo `is_done`, invirtiendo su valor. Luego los métodos pueden ser llamados desde el lado del cliente y siempre deben devolver algo. Si no devuelven nada, las llamadas del cliente usando el protocolo `XMLRPC` no funcionará. Si no tiene nada que devolver, la práctica común es simplemente devolver `True`.

Después de esto, si reinicie el servidor Odoo para cargar nuevamente el archivo Python, el botón **Toggle Done** debe funcionar.

Para el botón **Clear All Done** querrá ir un poco más lejos. Este debe buscar todos los registros activos que estén finalizados, y desactivarlos. Se supone que los botones de formulario solo actúan sobre los registros seleccionados, pero para mantener las cosas simples hará un poco de trampa, y también actuará sobre los demás botones:

```
@api.multi
def do_clear_done(self):
    done_recs = self.search([('is_done', '=', True)])
    done_recs.write({'active': False})
    return True
```

En los métodos decorados con `@api.multi` el `self` representa un conjunto de registros. Puede contener un único registro, cuando se usa desde un formulario, o muchos registros, cuando se usa desde la vista de lista. Ignore el conjunto de registros de `self` y construirá su propio conjunto `done_recs` que contiene todas la tareas marcadas como finalizadas. Luego fija la señal activa como `False`, en todas ellas.

El `search` es un método de la API que devuelve los registros que cumplen con algunas condiciones. Estas condiciones son escritas en un dominio, esto es una lista de tríos. Explorara con mayor detalle los dominios más adelante.

El método `write` fija los valores de todos los elementos en el conjunto de una vez. Los valores a escribir son definidos usando un diccionario. Usar `write` aquí es más eficiente que iterar a través de un conjunto de registros para asignar el valor uno por uno.

Note que `@api.one` no es lo más eficiente para estas acciones, ya que se ejecutará para cada uno de los registros seleccionados. La `@api.multi` se asegura que su código sea ejecutado una sola vez incluso si hay más de un registro seleccionado. Esto puede pasar si una opción es agregada a la vista de lista.

Configurando la seguridad en el control de acceso

Debe haber notado, desde que cargo su módulo, un mensaje de alerta en el registro del servidor:

> **The model todo.task has no access rules, consider adding one**.

El mensaje es muy claro: su modelo nuevo no tiene reglas de acceso, por lo tanto puede ser usado por cualquiera, no solo por el administrador. Como súper usuario el `admin` ignora las reglas de acceso, por ello es capaz de usar el formulario sin errores. Pero debe arreglar esto antes que otros usuarios puedan usarlo.

Para tener una muestra de la información requerida para agregar reglas de acceso a un modelo, use el cliente web y diríjase a: **Configuración > Técnico > Seguridad > Lista controles de acceso**.

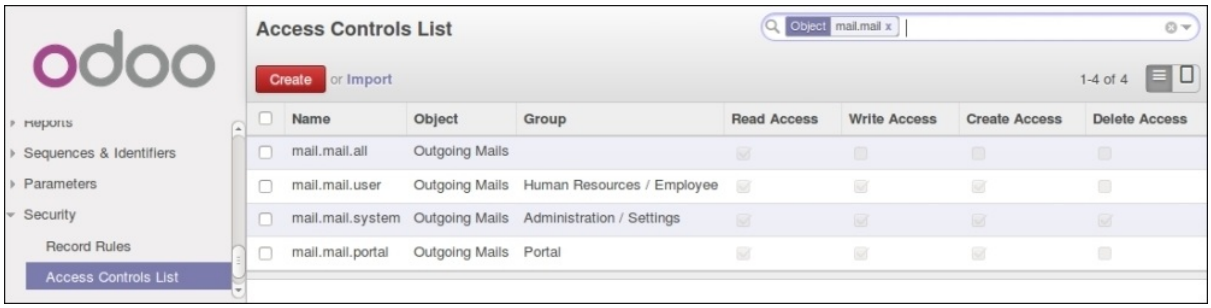

Figura 2.9: Gráfico 2.4 - Lista controles de acceso de Odoo

Aquí podrá ver la ACL para el modelo `mail.mail`. Este indica, por grupo, las acciones permitidas en los registros.

Esta información debe ser provista por el modelo, usando un archivo de datos para cargar las líneas dentro del modelo `ir.model.access`. Dará acceso completo al modelo al grupo empleado. Empleado es el grupo básico de acceso, casi todos pertenecen a este grupo.

Esto es realizado usualmente usando un archivo CSV llamado `security/ir.model.access.csv`. Los modelos generan identificadores automáticamente: para `todo.task` el identificador es `model_todo_task`. Los grupos también tienen identificadores fijados por los modelos que los crean. El grupo empleado es creado por el módulo base y tiene el identificador `base.group_user`. El nombre de la línea es solo informativo y es mejor si es único. Los módulos raíz usando una cadena separada por puntos con el nombre del modelo y el nombre del grupo. Siguiendo esta convención usara `todo.task.user`.

Ahora que tiene todo lo que necesita saber, va a agregar el archivo nuevo con el siguiente contenido:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_todo_task_group_user,todo.task.user,model_todo_task,base.group_user,1,1,1,1
```

No debe olvidar agregar la referencia a este archivo nuevo en el atributo "data" del descriptor en `__openerp__.py`, de la siguiente manera:

```
'data': [
    'todo_view.xml',
    'security/ir.model.access.csv',
],
```

Como se hizo anteriormente, actualice el módulo para que estos cambios tengan efecto. El mensaje de advertencia debería desaparecer, y puede confirmar que los permisos sean **correctos** accediendo con la cuenta de usuario demo (la contraseña es también demo) e intentar ejecutar la característica de "to-do tasks".

Reglas de acceso de nivel de fila

Odoo es un sistema multi-usuario, y querrá que la aplicación **to-do task** sea privada para cada usuario. Afortunadamente, Odoo soporta reglas de acceso de nivel de fila. En el menú **Técnico** pueden encontrarse en la opción **Reglas de Registro**, junto a la **Lista de Control de Acceso**. Las reglas de registro son definidas en el modelo `ir.rule`. Como es costumbre, necesita un nombre distintivo. También necesita el modelo en el cual operan y el dominio para forzar la restricción de acceso. El filtro de dominio usa la misma sintaxis de dominio mencionada anteriormente, y usado a lo largo de Odoo.

Finalmente, las reglas pueden ser globales (el campo `global` es fijado a `True`) o solo para grupos particulares de seguridad. En su caso, puede ser una regla global, pero para ilustrar el caso más común, la hará como una regla específica para un grupo, aplicada solo al grupo empleados.

Debe crear un archivo `security/todo_access_rules.xml` con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
    <openerp>
        <data noupdate="1">
            <record id="todo_task_user_rule" model="ir.rule">
                <field name="name">ToDo Tasks only for owner</field>
                <field name="model_id" ref="model_todo_task"/>
                <field name="domain_force">
                    [('create_uid','=',user.id)]
                </field>
                <field name="groups" eval="[(4,ref('base.group_user'))]"/>
            </record>
        </data>
    </openerp>
```

Nota el atributo `noupdate="1"`. Esto significa que estos datos no serán actualizados en las actualizaciones del módulo. Esto permitirá que sea personalizada más adelante, debido a que las actualizaciones del módulo no destruirán los cambios realizados. Pero ten en cuenta que esto será así mientras se esté desarrollando, por lo tanto es probable que quieras fijar `noupdate="0"` durante el desarrollo, hasta que estés feliz con el archivo de datos.

En el campo `groups` también encontraras una expresión especial. Es un campo de relación uno a muchos, y tienen una sintaxis especial para operar con ellos. En este caso la tupla `(4,x)` indica agregar `x` a los registros, y `x` es una referencia al grupo empleados, identificado por `base.group_user`.

Como se hizo anteriormente, debe agregar el archivo a `__openerp__.py` antes que pueda ser cargado al módulo:

```
'data': [
    'todo_view.xml',
    'security/ir.model.access.csv',
    'security/todo_access_rules.xml',
],
```

Agregar un ícono al módulo

Nuestro módulo se ve genial. ¿Por qué no añadir un ícono para que se vea aún mejor?. Para esto solo debe agregar al módulo el archivo `static/description/icon.png` con el ícono que usara.

Los siguientes comandos agregan un ícono copiado del módulo raíz `Notes`:

```
$ mkdir -p ~/odoo-dev/custom-addons/todo_app/static/description
$ cd ~/odoo-dev/custom-addons/todo_app/static/description
$ cp ../odoo/addons/note/static/description/icon.png ./
```

Ahora, si actualiza la lista de módulos, su módulo debe mostrarse con el ícono nuevo.

2.2.3 Resumen

Cree un módulo nuevo desde cero, cubriendo los elementos más frecuentemente usados en un módulo: modelos, los tres tipos base de vistas (formulario, lista y búsqueda), la lógica de negocio en los métodos del modelo, y seguridad en el acceso.

En el proceso, se familiarizó con el proceso de desarrollo de módulos, el cual incluye la actualización del módulo y la aplicación de reinicio del servidor para hacer efectivos en Odoo los cambios graduales.

Recuerde siempre, al agregar campos en el modelo, que es necesaria una actualización del módulo. Cuando se cambia el código Python, incluyendo el archivo de manifiesto, es necesario un reinicio del servidor. Cuando se cambian archivos XML o CSV es necesaria una actualización del módulo; incluso en caso de duda, realice ambas: actualización del módulo y reinicio del servidor.

En el siguiente capítulo, se aprenderá sobre la construcción de módulos que se acoplan a otro existentes para agregar características.

2.3 Capítulo 3 - Herencia

2.3.1 Herencia - Extendiendo la Funcionalidad de las Aplicaciones Existentes

Una de las características más poderosas de Odoo es la capacidad para agregar características sin modificar directamente los objetos subyacentes. Esto se logra a través de mecanismos de herencia, que funcionan como capas para la modificación por encima de los objetos existentes.

Estas modificaciones puede suceder en todos los niveles: modelos, vistas, y lógica de negocio. En lugar de modificar directamente un módulo existente, creara un módulo nuevo para agregar las modificaciones previstas.

Aquí, aprenderá como escribir sus propios módulos de extensión, confiriéndole facultades para aprovechar las aplicaciones base o comunitarias. Como ejemplo, aprenderá como agregar las características de mensajería y redes sociales de Odoo a sus propios módulos.

Agregar la capacidad de compartir con otros a la aplicación To-Do

La aplicación To-Do actualmente permite a los usuarios y gestionar de forma privada sus tareas por hacer. ¿No sería grandioso llevar su aplicación a otro nivel agregando características colaborativas y de redes sociales? Sera capaz de compartir las tareas y discutirlas con otras personas.

Hará esto con un módulo nuevo para ampliar la funcionalidad de la aplicación To-Do creada anteriormente y agregar estas características nuevas. Esto es lo que esperara lograr al final de este capítulo:

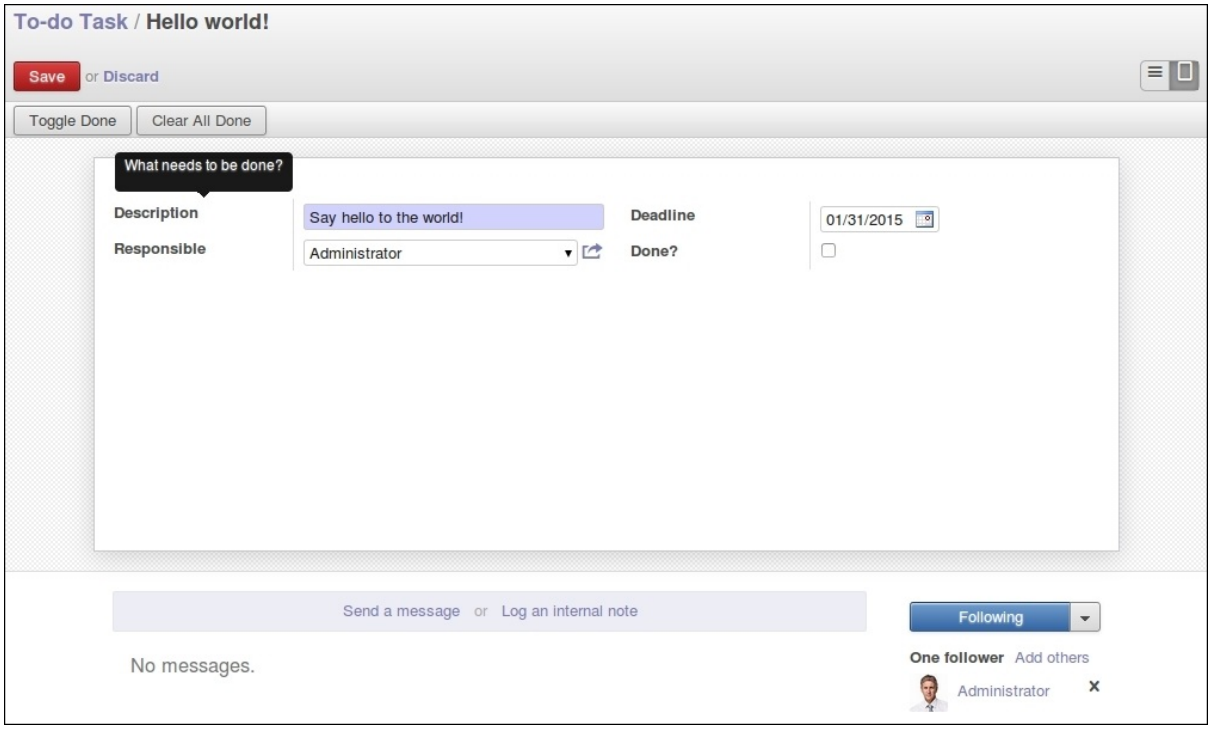

Figura 2.10: Gráfico 3.1 - Nuevo módulo para la aplicación To-Do

Camino a seguir para las características colaborativas

Aquí esta su plan de trabajo para implementar la extensión de funcionalidades:

- Agregar campos al modelo **Task**, como el usuario quien posee la tarea.
- Modificar la lógica de negocio para operar solo en la tarea actual del usuario, en vez de todas las tareas disponibles para ser vistas por el usuario.
- Agregar los campos necesarios a las vistas.
- Agregar características de redes sociales: el muro de mensajes y los seguidores.

Comience creando la estructura básica para el módulo junto al módulo `todo_app`. Siguiendo el ejemplo de instalación del Capítulo 1, sus módulos estarán alojados en `~/odoo-dev/custom-addons/`:

```
$ mkdir ~/odoo-dev/custom-addons/todo_user
$ touch ~/odoo-dev/custom-addons/todo_user/__init__.py
```

Ahora cree el archivo `__openerp__.py`, con el siguiente código:

```
{
    'name': 'Multiuser To-Do',
    'description': 'Extend the To-Do app to multiuser.',
    'author': 'Daniel Reis',
```

```
    'depends': ['todo_app'],
}
```

No ha hecho esto, pero incluir las claves “summary” y “category” puede ser importante cuando se publican módulos en la tienda de aplicaciones en línea de Odoo.

Ahora, podrá instalarlo. Debe ser suficiente con solo actualizar el **Lista de módulos** desde el menú **Configuración**, encuentre el módulo nuevo en la lista de **Módulos locales** y haga clic en el botón Instalar. Para instrucciones más detalladas sobre como encontrar e instalar un módulo puede volver al Capítulo 1.

Ahora, comience a agregar las nuevas características.

2.3.2 Ampliando el modelo de tareas por hacer

Los modelos nuevos son definidos a través de las clases Python. Ampliarlos también es hecho a través de las clases Python, pero usando un mecanismo específico de Odoo.

Para aplicar un modelo use una clase Python con un atributo `__inherit`. Este identifica el modelo que será ampliado. La clase nueva hereda todas las características del modelo padre, y solo necesite declarar las modificaciones que querrá introducir.

De hecho, los modelos de Odoo existen fuera de su módulo particular, en un registro central. Podrá referirse a este registro como la piscina, y puede ser accedido desde los métodos del modelo usando `self.env[<model name>]`. Por ejemplo, para referirse al modelo `res.partner` escribirá `self.env['res.partner']`.

Para modificar un modelo de Odoo obtiene una referencia a la clase de registro y luego ejecuta los cambios en ella. Esto significa que esas modificaciones también estarán disponibles en cualquier otro lado donde el modelo sea usado.

En la secuencia de carga del módulo, durante un reinicio del servidor, las modificaciones solo serán visibles en los modelos cargados después. Así que, la secuencia de carga es importante y debe asegurarse que las dependencias del módulo están fijadas correctamente.

Agregar campos a un modelo

Ampliara el modelo `todo.task` para agregar un par de campos: el usuario responsable de la tarea, y la fecha de vencimiento.

Cree un archivo `todo_task.py` nuevo y declare una clase que extienda al modelo original:

```
#-*- coding: utf-8 -*-
from openerp import models, fields, api
class TodoTask(models.Model):
    _inherit = 'todo.task'
    user_id = fields.Many2one('res.users', 'Responsible')
    date_deadline = fields.Date('Deadline')
```

El nombre de la clase es local para este archivo Python, y en general es irrelevante para los otros módulos. El atributo `_inherit` de la clase es la clave aquí: esta le dice a Odoo que esta clase hereda el modelo `todo.task`. Note la ausencia del atributo `_name`. Este no es necesario porque ya es heredado desde el modelo padre.

Las siguientes dos líneas son declaraciones de campos comunes. El `user_id` representa un usuario desde el modelo Users, `res.users`. Es un campo de `Many2one` equivalente a una clave foránea en el argot de base de datos. El `date_deadline` es un simple campo de fecha. En el Capítulo 5, se explica con más detalle los tipos de campos disponibles en Odoo.

Aun le falta agregar al archivo `__init__.py` la declaración `import` para incluirlo en el módulo:

```
from . import todo_task
```

Para tener los campos nuevos agregados a la tabla de la base de datos soportada por el modelo, necesita ejecutar una actualización al módulo. Si todo sale como es esperado, debería poder ver los campos nuevos cuando revise el modelo `todo.task`, en el menú **Técnico**, **Estructura de base de datos > Modelos**.

Modificar los campos existentes

Como puede ver, agregar campos nuevos a un modelo existente es bastante directo. Desde Odoo 8, es posible modificar atributos en campos existentes. Esto es hecho agregando un campo con el mismo nombre, y configurando los valores solo para los atributos que serán modificados.

Por ejemplo, para agregar un comentario de ayuda a un campo `name`, podrá agregar esta línea en el archivo `todo_task.py`:

```
name = fields.Char(help="What needs to be done?")
```

Si actualiza el módulo, va a un formulario de tareas por hacer, y posicione el ratón sobre el campo **Descripción**, aparecerá el mensaje de texto escrito en el código anterior.

Modificar los métodos del modelo

La herencia también funciona en la lógica de negocio. Agregar métodos nuevos es simple: solo declare las funciones dentro de la clase heredada.

Para ampliar la lógica existente, un método puede ser sobrescrito declarando otro método con el mismo nombre, y el método nuevo reemplazará al anterior. Pero este puede extender el código de la clase heredada, usando la palabra clave de Python `super()` para llamar al método padre.

Es mejor evitar cambiar la función distintiva del método (esto es, mantener los mismos argumentos) para asegurarse que las llamadas a este sigan funcionando adecuadamente. En caso que necesite agregar parámetros adicionales, hágalos opcionales (con un valor predeterminado).

La acción original de `Clear All Done` ya no es apropiada para su módulos de tareas compartidas, ya que borra todas las tareas sin importar a quien le pertenecen. Necesita modificarla para que borre solo las tareas del usuario actual.

Para esto, se sobrescribirá el método original con una nueva versión que primero encuentre las tareas completadas del usuario actual, y luego las desactive:

```
@api.multi
def do_clear_done(self):
    domain = [('is_done', '=', True), '|', ('user_id', '=', self.env.uid), ('user_id', '=', False
    done_recs = self.search(domain)
    done_recs.write({'active': False})
    return True
```

Primero se listan los registros finalizados sobre los cuales se usa el método `search` con un filtro de búsqueda. El filtro de búsqueda sigue una sintaxis especial de Odoo referida como `domain`.

El filtro "domain" usado es definido en la primera instrucción: es una lista de condiciones, donde cada condición es una tupla.

Estas condiciones son unidas implícitamente con un operador `AND` (`&` en la sintaxis de dominio). Para agregar una operación `OR` se usa una "tubería" (`|`) en el lugar de la tupla, y afectara las siguientes dos condiciones. Ahondara más sobre este tema en el Capítulo 6.

El dominio usado aquí filtra todas las tareas con su etapa finalizadas (`'is_done', '=', True`) que también tengan al usuario actual como responsable (`'user_id','=',self.env.uid`) o no tengan fijado un usuario (`'user_id', '=', False`).

Lo que acaba de hacer fue sobrescribir completamente el método padre, reemplazándolo con una implementación nueva.

Pero esto no es lo que usualmente querrá hacer. En vez de esto, ampliara la lógica actual y agregara operaciones adicionales. De lo contrario podrá dañar operaciones existentes. La lógica existente es insertada dentro de un método sobrescrito usando el comando `super()` de Python para llamar a la versión padre del método.

Vea un ejemplo de esto: podrá escribir una versión mejor de `do_toggle_done()` que solo ejecute la acción sobre las Tareas asignadas a su usuario:

```
@api.one
def do_toggle_done(self):
    if self.user_id != self.env.user:
        raise Exception('Only the responsible can do this!')
    else:
        return super(TodoTask, self).do_toggle_done()
```

Estas son las técnicas básicas para sobrescribir y ampliar la lógica de negocio definida en las clases del modelo. Vera ahora como extender las vistas de la interfaz con los usuarios.

2.3.3 Ampliar las vistas

Vistas de formulario, listas y búsqueda son definidas usando las estructuras de arco de XML. Para ampliar las vistas necesita una manera de modificar este XML. Esto significa localizar los elementos XML y luego introducir modificaciones en esos puntos. Las vistas heredadas permiten esto.

Una vista heredada se ve así:

```
<record id="view_form_todo_task_inherited" model="ir.ui.view">
    <field name="name">Todo Task form - User extension</field>
    <field name="model">todo.task</field>
    <field name="inherit_id" ref="todo_app.view_form_todo_task"/>
    <field name="arch" type="xml">
        <!-- ...match and extend elements here! ... -->
    </field>
</record>
```

El campo `inherit_id` identifica la vista que será ampliada, a través de la referencia de su identificador externo usando el atributo especial `ref`. Los identificadores externos serán discutidos con mayor detalle en el Capítulo 4.

La forma natural de localizar los elementos XML es usando expresiones XPath. Por ejemplo, tomando la vista que fue definida en el capítulo anterior, la expresión XPath para localizar el elemento `<field name="is_done">` es `//field[@name]='is_done'`. Esta expresión encuentra un elemento `field` con un atributo `name` igual a `is_done`. Puede encontrar mayor información sobre XPath en: https://docs.python.org/2/library/xml.etree.elementtree.html#xpath-support.

Tener atributos "name" en los elementos es importante porque los hace mucho más fácil de seleccionar como puntos de extensión. Una vez que el punto de extensión es localizado, puede ser modificado o puede tener elementos XML agregados cerca de él.

Como un ejemplo práctico, para agregar el campo `date_deadline` antes del campo `is_done`, debe escribir en `arch`:

```
<xpath expr="//field[@name]='is_done'" position="before">
    <field name="date_deadline" />
</xpath>
```

Afortunadamente Odoo proporciona una notación simplificada para eso, así que la mayoría de las veces podrá omitir la sintaxis *XPath*. En vez del elemento `xpath` anterior podrá usar el tipo de elementos que querrá localizar y su atributo distintivo.

Lo anterior también puede ser escrito como:

```
<field name="is_done" position="before">
    <field name="date_deadline" />
</field>`
```

Agregar campos nuevos, cerca de campos existentes es hecho frecuentemente, por lo tanto la etiqueta `<field>` es usada frecuentemente como el localizador. Pero cualquier otra etiqueta puede ser usada: `<sheet>`, `<group>`, `<div>`, entre otras. El atributo `name` es generalmente la mejor opción para hacer coincidir elementos, pero a veces, podrá necesitar usar `string` (el texto mostrado en un "label") o la clase CSS del elemento.

El atributo de posición usado con el elemento localizador es opcional, y puede tener los siguientes valores:

- `after`: Este es agregado al elemento padre, después del nodo de coincidencia.
- `before`: Este es agregado al elemento padre, antes del nodo de coincidencia.
- `inside` (el valor predeterminado): Este es anexado al contenido del nodo de coincidencia.
- `replace`: Este reemplaza el nodo de coincidencia. Si es usado con un contenido vacío, borra un elemento.
- `attributes`: Este modifica los atributos XML del elemento de coincidencia (más detalles luego de esta lista).

La posición del atributo le permite modificar los atributos del elemento de coincidencia. Esto es hecho usando los elementos `<attribute name="attr-name">` con los valores del atributo nuevo.

En el formulario de Tareas, tendrá el campo **Active**, pero tenerlo visible no es muy útil. Quizás podrá esconderlo al usuario. Esto puede ser realizado configurando su atributo `invisible`:

```
<field name="active" position="attributes">
    <attribute name="invisible">1</attribute>
</field>
```

Configurar el atributo `invisible` para esconder un elemento es una buena alternativa para usar el localizador de reemplazo para eliminar nodos. Debería evitarse la eliminación, ya que puede dañar las extensiones de modelos que pueden depender del nodo eliminado.

Finalmente, podrá poner todo junto, agregar los campos nuevos, y obtener la siguiente vista heredada completa para ampliar el formulario de tareas por hacer:

```
<record id="view_form_todo_task_inherited" model="ir.ui.view">
    <field name="name">Todo Task form - User extension</field>
    <field name="model">todo.task</field>
    <field name="inherit_id" ref="todo_app.view_form_todo_task"/>
    <field name="arch" type="xml">
        <field name="name" position="after">
            <field name="user_id" />
        </field>
        <field name="is_done" position="before">
            <field name="date_deadline" />
        </field>
        <field name="name" position="attributes">
            <attribute name="string">I have to...</attribute>
        </field>
    </field>
</record>
```

Esto debe ser agregado al archivo `todo_view.xml` en su módulo, dentro de las etiquetas `<openerp>` y `<data>`, como fue mostrado en el capítulo anterior.

Nota: Las vistas heredadas también pueden ser a su vez heredadas, pero debido a que esto crea dependencias más complicadas, debe ser evitado.

No podrá olvidar agregar el atributo datos al archivo descriptor `__openerp__.py`:

```
'data': ['todo_view.xml'],
```

Ampliando más vistas de árbol y búsqueda

Las extensiones de las vistas de árbol y búsqueda son también definidas usando la estructura XML `arch`, y pueden ser ampliadas de la misma manera que las vistas de formulario. Seguidamente se muestra un ejemplo de la ampliación de vistas de lista y búsqueda.

Para la vista de lista, querrá agregar el campo usuario:

```
<record id="view_tree_todo_task_inherited" model="ir.ui.view">
    <field name="name">Todo Task tree - User extension</field>
    <field name="model">todo.task</field>
    <field name="inherit_id" ref="todo_app.view_tree_todo_task"/>
    <field name="arch" type="xml">
        <field name="name" position="after">
            <field name="user_id" />
        </field>
    </field>
</record>
```

Para la vista de búsqueda, agregara una búsqueda por usuario, y filtros predefinidos para las tareas propias del usuario y tareas no asignadas a alguien.

```
<record id="view_filter_todo_task_inherited" model="ir.ui.view">
    <field name="name">Todo Task tree - User extension</field>
    <field name="model">todo.task</field>
    <field name="inherit_id" ref="todo_app.view_filter_todo_task"/>
    <field name="arch" type="xml">
        <field name="name" position="after">
            <field name="user_id" />
            <filter name="filter_my_tasks" string="My Tasks"
                    domain="[('user_id','in',[uid,False])]" />
            <filter name="filter_not_assigned" string="Not Assigned"
                    domain="[('user_id','=',False)]" />
        </field>
    </field>
</record>
```

No se preocupe demasiado por la sintaxis específica de las vistas. Se describirá esto con más detalle en el Capítulo 6.

2.3.4 Más sobre el uso de la herencia para ampliar los modelos

Ha visto lo básico en lo que se refiere a la ampliación de modelos "in place", lo cual es la forma más frecuente de uso de la herencia. Pero la herencia usando el atributo `_inherit` tiene mayores capacidades, como la mezcla de clases.

También tiene disponible el método de herencia delegada, usando el atributo `_inherits`. Esto permite a un modelo contener otros modelos de forma transparente a la vista, mientras por detrás de escena cada modelo gestiona sus propios datos.

Explore esas posibilidades en más detalle.

Copiar características usando herencia por prototipo

El método que use anteriormente para ampliar el modelo solo usa el atributo `_inherit`. Defina una clase que hereda el modelo `todo.task`, y le agrega algunas características. La clase `_name` no fue fijada explícitamente; implícitamente fue también `todo.task`.

Pero usando el atributo `_name` le permitió crear una mezcla de clases (mixin), incorporándolo al modelo que querrá ampliar. Aquí se muestre un ejemplo:

```
from openerp import models
class TodoTask(models.Model):
    _name = 'todo.task'
    _inherit = 'mail.thread'
```

Esto amplia el modelo `todo.task` copiando las características del modelo `mail.thread`. El modelo `mail.thread` implementa la mensajería de Odoo y la función de seguidores, y es reusable, por lo tanto es fácil agregar esas características a cualquier modelo.

Copiar significa que los métodos y los campos heredados estarán disponibles en el modelo heredero. Para los campos significa que estos serán creados y almacenados en las tablas de la base de datos del modelo objetivo. Los registros de datos del modelo original (heredado) y el nuevo modelo (heredero) son conservados sin relación entre ellos. Solo son compartidas las definiciones.

Estas mezclas son usadas frecuentemente como modelos abstractos, como el `mail.thread` usado en el ejemplo. Los modelos abstractos son como los modelos regulares excepto que no es creada ninguna representación de ellos en la base de datos. Actúan como plantillas, describen campos y la lógica para ser reusadas en modelos regulares.

Los campos que definen solo serán creados en aquellos modelos regulares que hereden de ellos. En un momento se discutirá en detalle como usar eso para agregar `mail.thread` y sus características de redes sociales a su módulo. En la práctica cuando se usan las mezclas rara vez hereda de modelos regulares, porque esto puede causar duplicación de las mismas estructuras de datos.

Odoo proporciona un mecanismo de herencia delegada, el cual impide la duplicación de estructuras de datos, por lo que es usualmente usada cuando se hereda de modelos regulares. Vea esto con mayor detalle.

Integrar Modelos usando herencia delegada

La herencia delegada es el método de extensión de modelos usado con menos frecuencia, pero puede proporcionar soluciones muy convenientes. Es usada a través del atributo `_inherits` (note la 's' adicional) con un mapeo de diccionario de modelos heredados con campos relacionados a él.

Un buen ejemplo de esto es el modelo estándar **Users**, `res.users`, que tiene un modelo **Partner** `res.partner` anidado:

```
from openerp import models, fields

class User(models.Model):
    _name = 'res.users'
    _inherits = {'res.partner': 'partner_id'}
    partner_id = fields.Many2one('res.partner')
```

Con la herencia delegada el modelos `res.users` integra el modelo heredado `res.partner`, por lo tanto cuando un usuario (User) nuevo es creado, un socio (Partner) también es creado y se mantiene una referencia a este a través del campo `partner_id` de User. Es similar al concepto de polimorfismo en la programación orientada a objetos.

Todos los campos del modelo heredado, Partner, están disponibles como si fueran campos de User, a través del mecanismo de delegación. Por ejemplo, el nombre del socio y los campos de dirección son expuestos como campos de User, pero de hecho son almacenados en el modelo Partner enlazado, y no ocurre ninguna duplicación de la estructura de datos.

La ventaja de esto, comparada a la herencia por prototipo, es que no hay necesidad de repetir la estructura de datos en muchas tablas, como las direcciones. Cualquier modelo que necesite incluir un dirección puede delegar esto a un modelo Partner vinculado. Y si son introducidas algunas modificaciones en los campos de dirección del socio o validaciones, estas estarán disponibles inmediatamente para todos los modelos que vinculen con él!

Nota: Note que con la herencia delegada, los campos con heredados, pero los métodos no.

Usar la herencia para agregar características redes sociales

El módulo de red social (nombre técnico `mail`) proporciona la pizarra de mensajes que se encuentra en la parte inferior de muchos formularios, también llamado Charla Abierta (Open Chatter), los seguidores se presentan junto a la lógica relativa a mensajes y notificaciones. Esto es algo que va a querer agregar con frecuencia a sus modelos, así que aprenda como hacerlo.

Las características de mensajería de red social son proporcionadas por el modelo `mail.thread` del modelo `mail`. Para agregarlo a un módulo personalizado necesita:

1. Que el módulo dependa de `mail`.
2. Que la clase herede de `mail.thread`.
3. Tener agregados a la vista de formulario los widgets `Followers` (seguidores) y `Threads` (hilos).
4. Opcionalmente, configurar las reglas de registro para seguidores.

Siga esta lista de verificación:

En relación a *#1*, debido a que su módulo ampliado depende de `todo_app`, el cual a su vez depende de `mail`, la dependencia de `mail` esta implícita, por lo tanto no se requiere ninguna acción.

En relación a *#2*, la herencia a `mail.thread` es hecha usando el atributo `_inherit`. Pero su clase ampliada de tareas por hacer ya está usando el atributo `_inherit`.

Afortunadamente, también puede aceptar una lista de modelos desde los cuales heredar, así que podrá usar esto para hacer que incluya la herencia a `mail.thread`:

```
_name = 'todo.task'
_inherit = ['todo.task', 'mail.thread']
```

El modelo `mail.thread` es un modelo abstracto. Los modelos abstractos son como los modelos regulares excepto que no tienen una representación en la base de datos; no se crean tablas para ellos. Los modelos abstractos no están destinados a ser usados directamente. Pero se espera que sean usados en la mezcla de clases, como acaba de hacer.

Podrá pensar en los modelos abstractos como plantillas con características listas para usar. Para crear una clase abstracta solo necesita usar modelos abstractos. AbstractModel en vez de `models.Model`.

Para la número *#3*, querrá agregar el widget de red social en la parte inferior del formulario. Podrá reusar la vista heredada que recién creada, `view_form_todo_task_inherited`, y agregar esto dentro de `arch`:

```
<sheet position="after">
    <div class="oe_chatter">
        <field name="message_follower_ids" widget="mail_followers" />
        <field name="message_ids" widget="mail_thread" />
    </div>
</sheet>
```

Los dos campos que ha agregado aquí no han sido declarados explícitamente, pero son provistos por el modelo `mail.thread`.

El paso final es fijar las reglas de los registros de seguidores, esto solo es necesario si su modelo tiene implementadas reglas de registro que limitan el acceso a otros usuarios. En este caso, necesita asegurarse que los seguidores para cada registro tengan al menos acceso de lectura.

Tendrá reglas de registro en su modelo de tareas por hacer así que necesita abordar esto, y es lo que hará en la siguiente sección.

Modificar datos

A diferencia de las vistas, los registros de datos no tienen una estructura de arco XML y no pueden ser ampliados usando expresiones XPath. Pero aún pueden ser modificados reemplazando valores en sus campos.

El elemento `<record id="x" model="y">` está realizando una operación de inserción o actualización en un modelo: si x no existe, es creada; de otra forma, es actualizada / escrita.

Debido a que los registros en otros módulos pueden ser accedidos usando un identificador `<model>.<identifier>`, es perfectamente legal para su módulo sobrescribir algo que fue escrito antes por otro módulo.

Nota: Note que el punto esta reservado para separar el nombre del módulo del identificador del objeto, así que no debe ser usado en identificadores. Para esto use la barra baja (_).

Como ejemplo, cambie la opción de menú creada por el módulo `todo_app` en "My To Do". Para esto agregar lo siguiente al archivo `todo_user/todo_view.xml`:

```
<!-- Modify menu item -->
<record id="todo_app.menu_todo_task" model="ir.ui.menu">
    <field name="name">My To-Do</field>
</record>
<!-- Action to open To-Do Task list -->
<record model="ir.actions.act_window" id="todo_app.action_todo_task">
    <field name="context">
        {'search_default_filter_my_tasks': True}
    </field>
</record>
```

Ampliando las reglas de registro

La aplicación Tareas-por-Hacer incluye una regla de registro para asegurar que cada tarea sea solo visible para el usuario que la ha creado. Pero ahora, con la adición de las características sociales, necesita que los seguidores de la tarea también tengan acceso. El modelo de red social no maneja esto por si solo.

Ahora las tareas también pueden tener usuarios asignados a ellas, por lo tanto tiene más sentido tener reglas de acceso que funcionen para el usuario responsable en vez del usuario que creo la tarea.

El plan será el mismo que para la opción de menú: sobrescribir `todo_app.todo_task_user_rule` para modificar el campo `domain_force` a un valor nuevo.

Desafortunadamente, esto no funcionará esta vez. Recuerde que el `<data no_update="1">` que use anteriormente en el archivo XML de las reglas de seguridad: previene las operaciones posteriores de escritura.

Debido a que las actualizaciones del registro no están permitidas, necesita una solución alterna. Este será borrar el registro y agregar un reemplazo para este en su módulo.

Para mantener las cosas organizadas, creara un archivo `security/todo_access_rules.xml` y agregara lo siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
    <openerp>
        <data noupdate="1">
            <delete model="ir.rule" search="[('id''=',ref('todo_app.todo_task_user_rule'))]" />
            <record id="todo_task_per_user_rule" model="ir.rule">
                <field name="name">ToDo Tasks only for owner</field>
                <field name="model_id" ref="model_todo_task"/>
                <field name="groups" eval="[(4, ref('base.group_user'))]"/>
                <field name="domain_force">
                    ['|', ('user_id','in', [user.id,False]), ('message_follower_ids','in',[user.pa
                </field>
            </record>
        </data>
    </openerp>
```

Esto encuentra y elimina la regla de registro `todo_task_user_rule` del módulo `todo_app`, y crea una nueva regla de registro `todo_task_per_user`. El filtro de dominio que usa ahora hace la tarea visible para el usuario responsable `user_id`, para todo el mundo si el usuario responsable no ha sido definido (igual a `False`), y para todos los seguidores. La regla se ejecutará en un contexto donde el usuario este disponible y represente la sesión del usuario actual. Los seguidores son socios, no objetos User, así que en vez de `user_id`, necesita usar `user.partner_id.id`.

Truco: Cuando se trabaja en campos de datos con `<data noupdate="1">` puede ser engañoso porque cualquier edición posterior no será actualizada en Odoo. Para evitar esto, use temporalmente `<data noupdate="0">` durante el desarrollo, y cámbielo solo cuando haya terminado con el módulo.

Como de costumbre, no debe olvidar agregar el archivo nuevo al archivo descriptor `__openerp__.py` en el atributo "data":

```
'data': [
    'todo_view.xml',
    'security/todo_access_rules.xml'
],
```

Note que en la actualización de módulos, el elemento `<delete>` arrojará un mensaje de advertencia, porque el registro que será eliminado no existe más. Esto no es un error y la actualización se realizará con éxito, así que no es necesario preocuparse por esto.

2.3.5 Resumen

Ahora debe ser capaz de crear módulos nuevos para ampliar los módulos existentes. Vio como ampliar el módulo To-Do creado en los capítulos anteriores.

Se agregaron nuevas características en las diferentes capas que forman la aplicación. Amplio el modelo Odoo para agregar campos nuevos, y amplié los métodos con su lógica de negocio. Luego, modifique las vistas para hacer disponibles los campos nuevos. Finalmente, aprendió como ampliar un modelo heredando de otros modelos, y use esto para agregar características de red social a su aplicación.

Con estos tres capítulos, tiene una vista general de las actividades mas comunes dentro del desarrollo en Odoo, desde la instalación de Odoo y configuración a la creación de módulos y extensiones.

Los siguientes capítulos se enfocarán en áreas específicas, la mayoría de las cuales ha tocado en estos primeros capítulos. En el siguiente capítulo, abordara la serialización de datos y el uso de archivos XML y CSV con más detalle.

2.4 Capítulo 4 - Serialización

2.4.1 Serialización de Datos y Datos de Módulos

La mayoría de las configuraciones de Odoo, desde interfaces de usuario hasta reglas de seguridad, son en realidad registros de datos almacenados en tablas internas de Odoo. Los archivos XML y CSV que se encuentran en los módulos no son usados para ejecutar aplicaciones Odoo. Ellos solo son un medio para cargar esas configuraciones a las tablas de la base de datos.

Los módulos pueden también tener datos iniciales y de demostración (accesorios). La serialización de datos permite añadir eso a sus módulos. Adicionalmente, entendiendo los formatos de serialización de datos de Odoo es importante para exportar e importar datos en el contexto de la implementación de un proyecto.

Antes de entrar en casos prácticos, primero explorara el concepto de identificador externo, el cual es la clave a la serialización de datos de Odoo

Entendiendo los Identificadores Externos

Todos los registros en la base de datos de Odoo tienen un identificador único, el campo `id`

Es un número secuencial asignado automáticamente por la base de datos. De cualquier forma, este identificador automático puede ser un desafío al momento de cargar datos interrelacionados: ¿cómo podrá hacer referencia a un registro relacionado si no podrá saber de antemano cual ID de base de datos le será asignado?

La respuesta de Odoo a esto es el identificador externo. Los identificadores externos solucionan este problema asignando identificadores con nombre a los registros de datos que serán cargados. Un identificador con nombre puede ser usado por cualquier otra pieza de dato registrada para referenciarla luego. Odoo se encargará de traducir estos nombres de identificación a los IDs reales asignados a ellos.

El mecanismo detrás de esto es muy simple: Odoo mantiene una tabla con el mapeo entre los IDs externos con nombre y sus correspondiente IDs numéricos en la base de datos. Ese es el modelo `ir.model.data`.

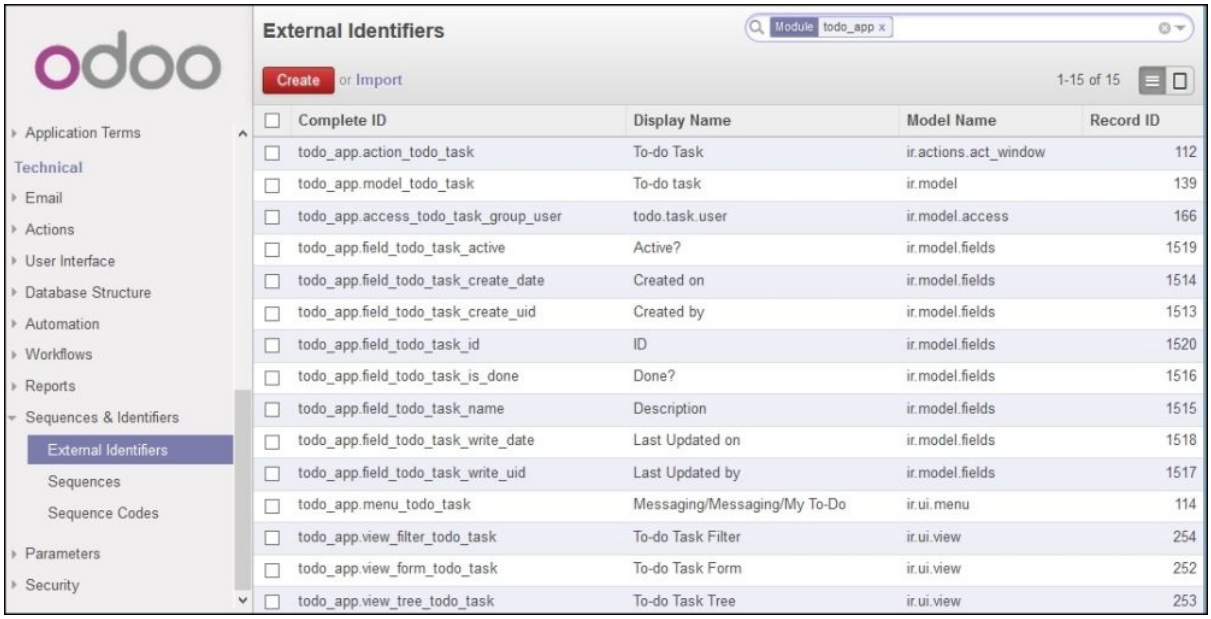

Figura 2.11: Gráfico 4.1 - Vista de 'External Identifiers'

Para inspeccionar la existencia de mapeo, diríjase a la sección **Técnico** en el menú **Configuración**, y seleccione **Secuencias e identificadores** haga el item de menú **Identificadores externos**.

Por ejemplo, si vuelve a visitar la lista de identificadores externos y filtra por el módulo `todo_app`, vera los identificadores externos creados previamente por el módulo.

Puede notar que los identificadores externos tienen una etiqueta ID completa. Esta compuesta por el nombre del módulo y el nombre de identificador unido por un punto, por ejemplo, `todo_app.action_todo_task`.

Debido a que solo es obligatorio que el ID completo sea único, el nombre del módulo sirve como `namespace` para los identificadores. Esto significa que el mismo identificador puede repetirse en diferentes módulos, y no tiene que preocuparnos por identificadores en su módulo que colisionen con identificadores en otros módulos.

Figura 2.12: Gráfico 4.2 - Detalles del ID 'todo_app.action_todo_task'

Al principio de la lista, puedes ver el ID de `todo_app.action_todo_task`. Esta es la acción del menú que crea para el módulo, el cual también es referenciado en el elemento de menú correspondiente. Haciendo clic en él, puede abrir un formulario con sus detalles: el `action_todo_task` en el módulo `todo_app` mapea hacia un ID de un registro específico en el modelo `ir.actions.act_window`.

Además de proporcionar una forma de hacer referencia a un registro de una manera fácil a otros registros, los ID externos también permiten evitar la duplicidad de datos en las importaciones repetidas. Si el ID externo esta

presente, el registro existente se actualiza, en ves de crear un nuevo registro. Esta es la razón de porque, en las siguientes actualizaciones del módulo, los registros cargados previamente se actualizaran en lugar de duplicarse.

Encontrando los identificadores externos

Cuando prepara archivos de datos para la demostración y configuración del módulo, tiene que mirar frecuentemente la existencia de IDs Externos que se necesitan para realizar las referencias.

Podrá utilizar identificadores externos en el menú mostrado anteriormente, pero el **menú de Desarrollo** puede proporcionar un método que sea más conveniente. Como recordara en el Capítulo 1, *Comenzando con el Desarrollo en Odoo*, el **menú de Desarrollo** es activado en la opción **Acerca de Odoo**, y entonces, estará disponible en la esquina superior izquierda de la vista del cliente web.

Para buscar el ID externo para un registro de datos, en el mismo formulario correspondiente, seleccione la opción **Ver metadatos** desde el **menú de Desarrollador**. Esto mostrará un cuadro de dialogo con el ID de la base de datos registrado y el ID externo (también conocido como ID XML)

Como Ejemplo, para buscar el ID del usuario Demo, podrá navegar hasta la vista de formulario (**Configuración > Usuarios**) y seleccionar la opción **Ver metadatos**, y se mostrará lo siguiente:

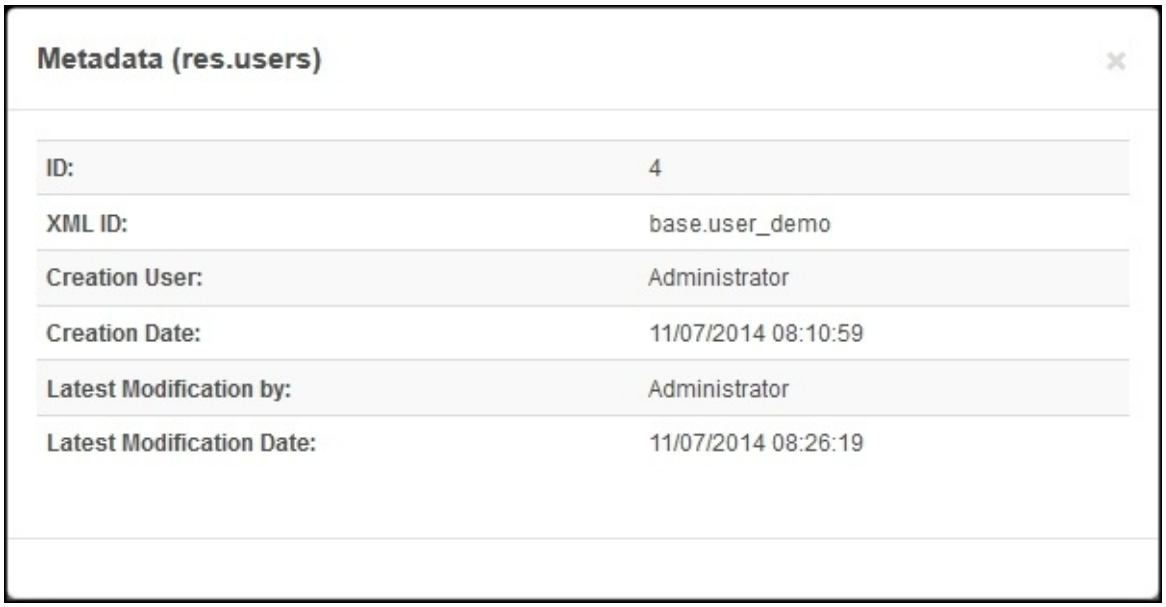

Figura 2.13: Gráfico 4.3 - Ver metadatos del módulo 'res.users'

Para buscar el ID externo de los elementos de vista, como formulario, árbol, buscador, y acción, el **menú de Desarrollo** es también de ayuda. Para esto, utilice la opción de **Obtener Campos de Vista** o abra la información para la vista deseada usando la opción Editar , y seleccione la opción Ver metadatos.

2.4.2 Exportar e importar datos

Va a empezar a trabajar en la exportación e importación de datos en Odoo, y desde allí, va a pasar a los detalles técnicos.

Exportando datos

Exportar datos es una característica estándar disponible en cualquier vista de lista. Para usarlo, primero tiene que seleccionar la fila de exportación seleccionando las casillas de verificación correspondiente en el extremo izquierdo, y luego selecciona la opción exportar en el botón "más".

Aquí esta un ejemplo, utilizando las reciente tareas creadas a realizar:

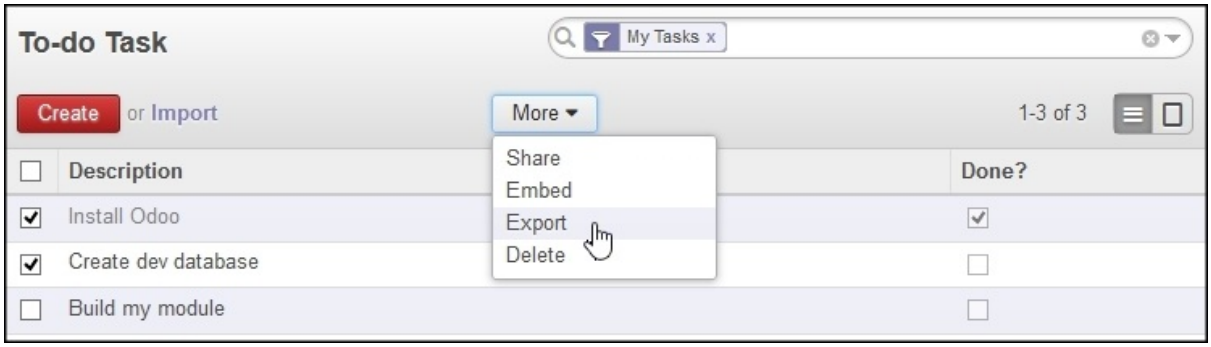

Figura 2.14: Gráfico 4.4 - Exportando datos del módulo 'To-Do'

La opción exportar le lleva a un dialogo, donde podrá elegir lo que se va a exportar. La opción exportar compatible se asegura de que el archivo exportado se pueda importar de nuevo a Odoo.

El formato de exportación puede ser CSV o Excel. Va a preferir archivos CSV para tener una mejor comprensión del formato de exportación. Continué, eligiendo las columnas que querrá exportar y hacer clic en el botón **Exportar a fichero**. Esto iniciara la descarga de un archivo con los datos exportados.

Si sigue estas instrucciones y selecciona los campos que se demuestran en la imagen anterior, debe terminar con un archivo de texto CSV similar a este:

```
"id","name","user_id/id","date_deadline","is_done" "__export__.todo_task_1","Install Odoo","base.u
```

Observe que Odoo exporta automáticamente una columna adicional identificada. Este es un ID externo que se genera automáticamente para cada registro. Estos identificadores externos generados utilizan `__export__` en lugar de un nombre real de módulo. Nuevos identificadores solo se asignan a los que no poseen uno asignado, y ya a partir de allí, se mantienen unidos al mismos registro. Esto significa que las exportaciones posteriores preservarán los mismos identificadores externos.

Importar datos

Primero tiene que asegurarse que la función de importar este habilitada. Esto se hace en el menú de **Configuración**, **Configuración** > opción de **Configuraciones Generales**. En **Importar/Exportar**, asegúrese que la opción **Permitir a los usuarios importar datos desde archivos CSV** esté habilitada.

Con esta opción habilitada, los puntos de vista de la lista muestran la opción de **Importar** junto al botón **Crear** en la parte superior de la lista.

Va a realizar una edición masiva en sus datos de tareas pendientes: se abre en una hoja de calculo o en un editor de texto el archivo CSV que acaba de descargar, a continuación, cambie algunos valores y añada algunas nuevas filas.

Como se mencionó antes,la primera columna de identificación proporciona un identificador único para cada fila permitiendo registros ya existentes que se actualizaran en ves de duplicarse cuando importe los datos de nuevo a Odoo. Para las nuevas filas que podrá añadir al archivo CSV, el `id` se deben dejar en blanco, y se creara un nuevo registro para ellos.

Después de guardar los cambios en el archivo CSV, haga clic en la opción **Importar** (junto al botón crear) y se presentara el asistente de importación. Hay que seleccionar la ubicación del archivo CSV en el disco y hacer clic en **Validar** para comprobar si el formato es correcto. Debido a que en archivo a importar esta basado en una importación de Odoo, es probable que es archivo sea correcto.

Ahora podrá hacer clic en **Importar** y allí va: sus modificaciones y nuevos registros deberían haberse cargado en Odoo.

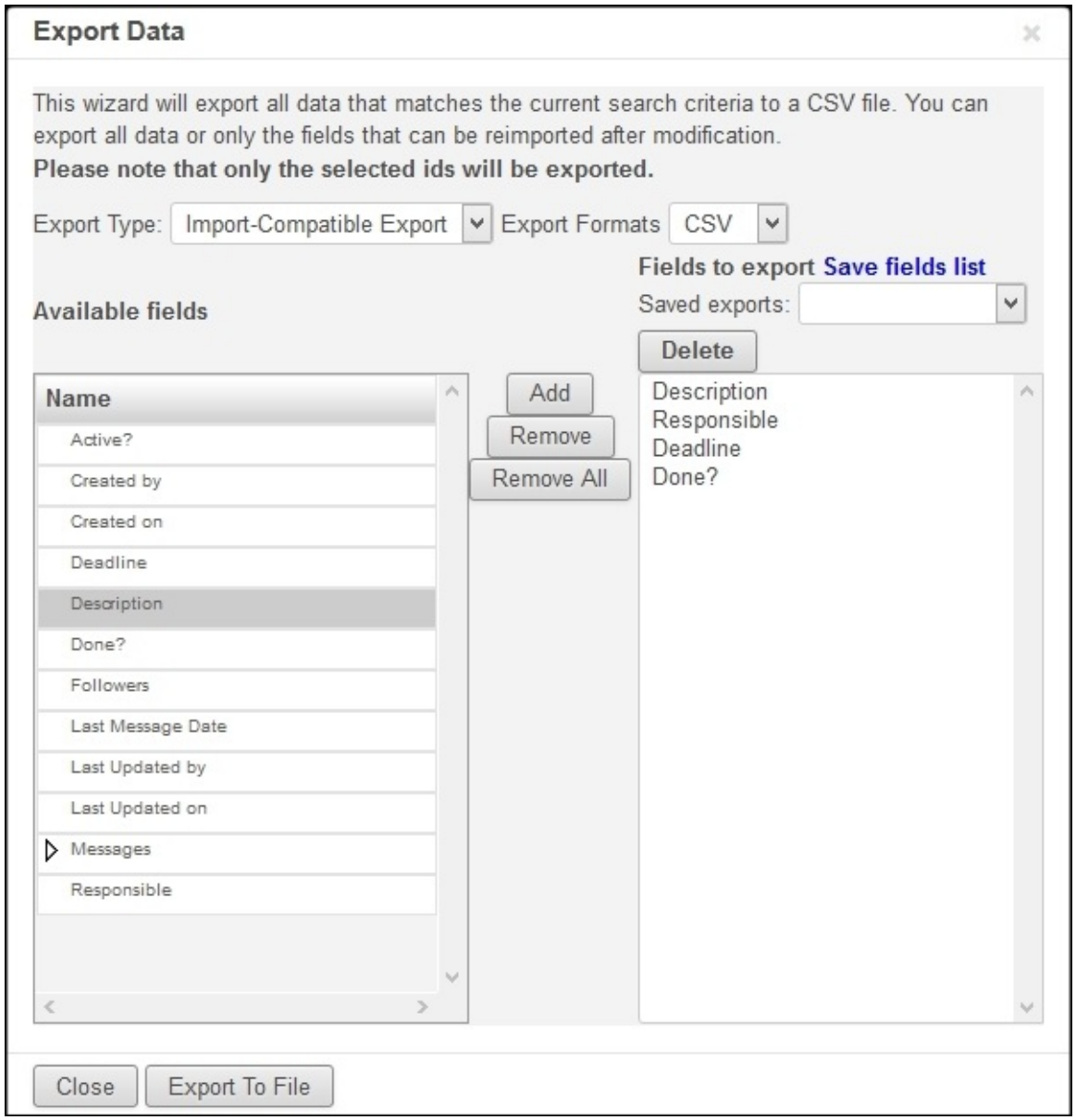

Figura 2.15: Gráfico 4.5 - Asistente para exportar datos del módulo 'To-Do'

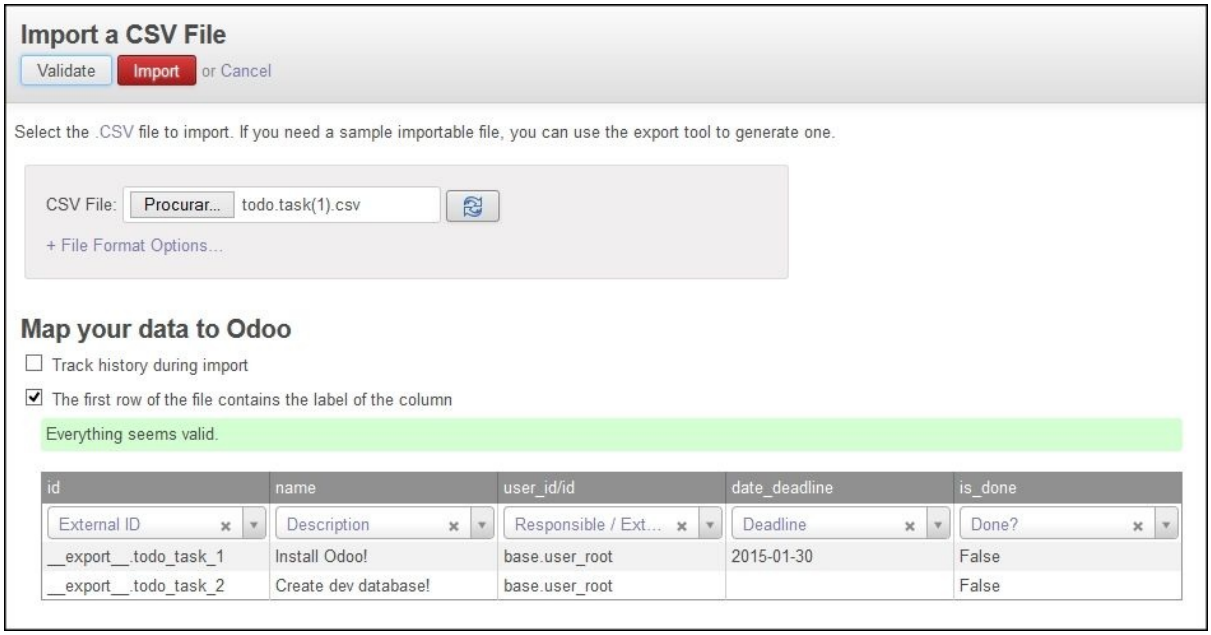

Figura 2.16: Gráfico 4.6 - Importar archivos de datos CSV

Registros relacionados en archivos de datos CSV

En el ejemplo visto anteriormente, el usuario responsable de cada tarea es un registro relacionado en el modelo de los usuarios, con la relación *many to one* - muchos a uno - (o foreign key - clave foránea). El nombre de la columna para ello fue `usuario_id/id` y los valores de los campos eran identificadores externos para los registros relacionados, tales como `base.user_root` para el usuario administrador.

Las columnas de relación deben tener `/id` anexo a su nombre, si se usan IDs externos, o `/.id`, si se usan IDs (numéricos) de base de datos. Alternativamente, dos puntos `(:)` se puede utilizar en lugar de la barra para el mismo efecto.

Del mismo modo, la relación *many to many* - muchos a muchos - son soportables. Un ejemplo de relación *many to many* es la que existe entre usuarios y grupos: cada usuario puede estar en muchos grupos, y cada grupo puede tener muchos usuarios. La columna nombre para este tipo de campo debería haber añadido un `/id`. Los valores de los campos aceptan una lista separada por comas de Id externos, entre comillas dobles.

Por ejemplo, los Seguidores de las tareas a realizar es una relación *many-to-many* entre Tareas por hacer y Socios. El nombre de la columna puede ser `follower_ids/id` y un valor de campo con dos seguidores podría ser: `"__export__.res_partner_1,__export__.res_partner_2"`

Finalmente, las relaciones *one to many* también se pueden importar a través de CSV. El ejemplo típico de esta relación es un documento “head” con varias “lines”.

Podrá ver un ejemplo de tal relación en el modelo de empresa (la vista de formulario esta disponible en el menú configuración): una empresa puede tener varias cuentas bancarias, cada una con sus propios detalles, y cada cuenta bancaria pertenece a (tiene una relación *many-to-one* con) solo una empresa.

Es posible importar las empresa junto con sus cuentas bancarias en un solo archivo. Para esto, algunas columnas corresponderán a empresas, y otras columnas corresponderán a cuentas bancarias detalladas. Los nombres de columnas de los detalles del banco deben ser precedidos de los campos con la relación *one-to-many* que vincula a la empresa con los bancos; `bank_ids` en este caso.

Los primeros datos de la cuenta bancaria van en la misma fila de los datos vinculados de la empresa. Los detalles de la próxima cuenta bancaria van en la siguiente fila, pero solo los datos bancarios de la columna relacionada deben tener valores; La columna de datos de la empresa debe tener esas lineas vacías.

Aquí esta un ejemplo cargando una empresa con datos de tres bancos:

```
id,name,bank_ids/id,bank_ids/acc_number,bank_ids/state base.main_company,YourCompany,__export__.re
,,__export__.res_partner_bank_6,1122334455,bank
```

Observe que las dos ultimas lineas comienzan con comas: Esto corresponde a valores en las dos primeras columnas, `id` y `name`, con respecto a los datos del encabezado de empresa. Pero las columnas restantes, con respecto a las cuentas bancarias, tienen valores para el segundo y tercer registro del banco.

Estos son los elementos esenciales en el trabajo con la exportación e importación en la GUI. Es útil para establecer los datos en nuevas instancias Odoo, o para prepara nuevos archivos de datos que se incluirán en los módulos Odoo.

A continuación va aprender más sobre el uso de los archivos de datos en los módulos.

Datos de los Módulos

Los módulos utilizan archivos de datos para cargar sus configuraciones en la base de datos, los datos iniciales y los datos de demostración. Esto se puede hacer utilizando tanto CSV y archivos XML. Para completar, el formato de archivo YAML también se puede utilizar, pero esto rara vez se utiliza para la carga de datos, por lo tanto no se discutirá.

Los archivos CSV utilizados por módulos son exactamente los mismos que los que ha visto y utilizado para la función de importación. Cuando se usa en módulos, la única restricción adicional es que el nombre del archivo debe coincidir con el nombre del modelo a la que se cargan los datos.

Un ejemplo común es el acceso de seguridad, para cargar en el modelo `ir.model.acess`. Esto se hace generalmente con archivos CSV, y que debe ser nombrado `ir.model.acess.csv`.

Datos de demostración

Los módulos Odoo pueden instalar datos de demostración. Esto es útil para proporcionar ejemplos de uso para un módulo y conjuntos de datos para ser utilizados en pruebas. Se considera una buena práctica para los módulos proporcionar datos de demostración. Los datos de demostración para un módulo se declara con el atributo `demo` del archivo de manifiesto `__openerp__.py`. Al igual que el atributo `data`, se trata de una lista de nombres de archivo con las rutas relativas correspondientes en el interior del módulo.

Estará agregando los datos de demostración en su módulo `todo_user`. Podrá comenzar con la exportación de algunos datos de las tareas a realizar, como se explico en la sección anterior. Luego debe guardar los datos en el directorio `todo_user` con el nombre del archivo `todo.task.csv`. Dado que esta información será propiedad de su módulo, debe editar los valores de `id` para reemplazar el prefijo `__export__` en los identificadores con el nombre técnico del módulo.

Como ejemplo su archivo de datos `todo.task.csv` podría tener este aspecto:

```
id,name,user_id/id,date_deadline todo_task_a,"Install Odoo","base.user_root","2015-01-30" todo_ta
```

No hay que olvidar agregar este archivo de datos en el atributo `demo` del `__openerp__.py`:

```
'demo':     ['todo.task.csv'],
```

La próxima vez que actualice el módulo, siempre y cuando se haya instalado con los datos de demostración habilitados, se importará el contenido del archivo. Tenga en cuenta que estos datos se reescribirán cada vez que se realiza una actualización del módulo.

Los archivos XML también pueden ser utilizados para los datos de demostración. Sus nombres de archivo no están obligados a coincidir con el modelo a cargar, porque el formato XML es mucho más rico y la información es proporcionada por los elementos XML dentro del archivo.

Va a aprender más sobre lo que los archivos de datos XML le permiten hacer y que los archivos CSV no.

Archivos de datos XML

Mientras que los archivos CSV proporcionan un formato simple y compacto para serializar los datos, los archivos XML son más potentes y dan un mayor control sobre el proceso de carga.

Ya ha utilizado los archivos de datos XML en los capítulos anteriores. Los componentes de la interfaz de usuario, tales como vistas y elementos de menú, se encuentran en los registros de datos almacenados en los modelos de sistemas. Los archivos XML en los módulos son un medio utilizado para cargar los registros en el servidor.

Para mostrar esto, va a añadir un segundo archivo de datos para el módulo `todo_user`, llamado `todo_data.xml`, con el siguiente contenido:

```
<?xml version="1.0"?>
<openerp>
  <data>
    <!-- Data to load -->
    <record model="todo.task" id="todo_task_c">
      <field name="name">Reinstall Odoo</field>
      <field name="user_id" ref="base.user_root" />
      <field name="date_deadline">2015-01-30</field>
    </record>
  </data>
</openerp>
```

Este XML es equivalente al archivo de datos CSV que acaba de ver en la sección anterior.

Los archivos de datos XML tienen un elemento `<openerp>` que contiene elementos `<data>`, dentro de los cuales podrá tener tienen varios elementos `<record>`, correspondientes a las filas de datos CSV.

Un elemento `<record>` tiene dos atributos obligatorios, `model` y `id` (el identificador externo para el registro), y contiene una etiqueta `<field>` para cada campo de texto.

Tenga en cuenta que la notación con barras en los nombres de campo no está disponible aquí: no podrá usar `<field name="user_id/id">`. En cambio, el atributo especial `ref` se utiliza para hacer referencia a los identificadores externos. Se hablara de los valores para el campo relacional "a muchos" en un momento.

El atributo de datos noupdate

Cuando se repite la carga de datos, los registros existentes de la ejecución anterior se reescriben.

Esto es importante a tener en cuenta: significa que la actualización de un módulo se sobrepone a los cambios manuales que podrían haber sido realizados en los datos. Cabe destacar que, si las vistas fueron modificadas con personalizaciones, esos cambios se perderán con la próxima actualización del módulo. El procedimiento correcto es crear vistas heredadas de los cambios que necesita, como se explica en el Capítulo 3.

Este comportamiento de sobrescribir es el valor predeterminado, pero se puede cambiar, por lo que cuando un registro ya creado se carga de nuevo no se realiza ningún cambio al mismo. Esto se hace añadiendo al elemento `<data>` un atributo `noupdate="1"`. Con esto, sus registros se crearán la primera vez que se cargan, y en mejoras de módulos subsiguientes no se hará nada para ellos.

Esto permite que las personalizaciones realizadas manualmente estén a salvo de las actualizaciones del módulo. Se utiliza a menudo con las reglas de acceso de registro, lo que les permite adaptarse a las necesidades específicas de aplicación.

También es posible tener más de una sección `<data>` en el mismo archivo XML. Podrá tomar ventaja de esto para tener un conjunto de datos con `noupdate="1"` y otro con `noupdate="0"`.

La etiqueta `noupdate` se almacena en la información de Identificador Externo para cada registro. Es posible editar la directamente utilizando el formulario de Identificador Externo disponible en el menú **Técnico** > opción **Secuencias e identificadores** > **Identificadores externos**, con la casilla de verificación **No actualizable**.

Truco: El atributo `noupdate` es difícil de manejar cuando se esta desarrollando el módulo, ya que los cambios hechos a los datos más tarde serán ignorados y Odoo no recogerá las modificaciones. Una solución es mantener

`noupdate="0"` durante el desarrollo y sólo ponerlo a * 1 *una vez terminado*.

Definición de registros en XML

Cada elemento `<record>` tiene dos atributos básicos, `id` y `model`, y contiene elementos `<field>` de la asignación de valores a cada columna. Como se mencionó antes, el atributo `id` corresponde ID Externo del registro y el `model` al el modelo de destino donde se escribirá el registro. Los elementos `<field>` tienen disponibles algunas maneras diferentes para asignar valores. Vea en detalle.

Configuración de los valores de campo

El elemento `<record>` define un registro de datos, y contiene elementos para establecer los valores de cada campo.

El atributo `name` del elemento `field` identifica el campo a ser escrito.

El valor a escribir es el contenido del elemento: el texto entre la etiqueta de apertura y la etiqueta de cierre del elemento `field`. En general, esto también es adecuado para establecer los valores que no son texto: para Booleanos, `0` y `1` o valores `False` y `True`; para fechas, fechas y horas, cadenas de texto como `YYYY-MM-DD` y `YYYY-MM-DD HH:MI:SS`, se realizará una correcta conversión.

Ajuste de valores utilizando expresiones

Una alternativa más avanzada para definir un valor de `field` es utilizar el atributo `eval`. Este evalúa una expresión Python y asigna el valor resultante al campo.

La expresión se evalúa en un contexto que, además de Python empotrado, también tiene algunos identificadores adicionales disponibles. Va a echar un vistazo a ellos.

Para manejar fechas, los siguientes módulos están disponibles: `time`, `datetime`, `timedelta` y `relativedelta`. Ellos permiten el cálculo de los valores de fecha, algo que se utiliza con frecuencia en los datos de demostración (y prueba). Por ejemplo, para establecer un valor de ayer se usaría:

```
<field name="expiration_date" eval="(datetime.now()+timedelta(-1)).strftime('%Y-%m-%d')"/>
```

También esta disponible en el contexto de evaluación la función `ref()`, que se utiliza para traducir un ID Externo al ID de base de datos correspondiente. Esto puede ser usado para establecer los valores para los campos relacionales. A modo de ejemplo, lo ha usado antes para ajustar el valor para el `user_id`:

```
<field name="user_id" eval="ref('base.group_user')" />
```

El contexto de evaluación también tiene una referencia, disponible para el Modelo actual, escrita a través de `obj`. Se puede utilizar junto con `ref()` para acceder a los valores de otros registros. He aquí un ejemplo del módulo de venta:

```
<value model="sale.order" eval="obj(ref('test_order_1')).amount_total" />
```

Configuración de los valores de los campos de relación

Acaba de ver cómo establecer un valor en un campo de relación muchos-a-uno, como `user_id`, usando el atributo `eval` con una función `ref()`. Pero hay una manera más sencilla.

El elemento `<field>` también tiene un atributo `ref` para establecer el valor de campo *many-to-one* utilizando un ID Externo. Usándolo, podrá establecer el valor de `user_id` con solo:

```
<field name="user_id" ref="base.group_user" />
```

Para campos *one-to-many* y *many-to-many*, se espera una lista de ID relacionados, por lo que es necesaria una sintaxis diferente, y Odoo proporciona una sintaxis especial para escribir sobre este tipo de campos.

El siguiente ejemplo, tomado de la aplicación de Flota, sustituye a la lista de registros relacionados de un campo `tag_ids`:

```
<field name="tag_ids" eval="[(6,0,[ref('vehicle_tag_leasing'),ref('fleet.vehicle_tag_compact'),re
```

Para escribir sobre un campo a-muchos se utiliza una lista de tripletas. Cada tripleta es un comando de escritura que hace cosas diferentes según el código utilizado:

- `(0,_,{'field':value})`: Esto crea un nuevo registro y lo vincula a ésta.
- `(1,id,{'field':value})`: Esto actualiza los valores en un registro ya vinculados.
- `(2,id,_)`: Esto desvincula y elimina un registro relacionado.
- `(3,id,_)`: Esto desvincula pero no elimina un registro relacionado.
- `(4,id,_)`: Esto vincula un registro ya existente.
- `(5,_,_)`: Esto desvincula pero no elimina todos los registros vinculados.
- `(6,_,[ids])`: Esto reemplaza la lista de registros vinculados con la lista proporcionada.

El símbolo guión bajo utilizado anteriormente representa valores irrelevantes, por lo general lleno de 0 o `False`.

Atajos para modelos de uso frecuente

Si se remonta al Capítulo 2, *La construcción de su primera aplicación Odoo*, podrá encontrar en los archivos XML otros elementos además de `<record>` , como `<act_window>` y `<menuitem>`.

Estos son los atajos convenientes para los modelos de uso frecuente, que también se pueden cargar utilizando elemento `<record>` regulares. Estos cargan datos en los modelos base y dan apoyo a la interfaz de usuario, se estudiarán con detalle más adelante, en el capítulo 6, *Vistas - Diseño de la interfaz de usuario*.

Como referencia, de manera que podrá comprender mejor los archivos XML que podrá encontrar en los módulos existentes, los siguientes elementos de acceso directo están disponibles con los modelos correspondientes donde cargan los datos:

- `<act_window>`: Este es el modelo de acciones de ventana `ir.actions.act_window`.
- `<menuitem>`: Este es el modelo de elementos de menú `ir.ui.menu`.
- `<report>`: Este es el modelo de acciones de reporte `ir.actions.report.xml`.
- `<template>`: Esto es el modelo de plantillas de vistas QWeb almacenadas en `ir.ui.view`.
- `<url>`: Este es el modelo de acciones de URL `ir.actions.act_url`.

Otras acciones en archivos de datos XML

Hasta ahora ha visto cómo añadir o actualizar datos mediante archivos XML. Pero los archivos XML también permiten realizar otro tipo de acciones, a veces necesarios para configurar los datos. En particular, son capaces de eliminar los datos, ejecutar métodos arbitrarios del modelo, e iniciar la ejecución de eventos de flujo de trabajo.

Eliminación de registros Para borrar un registro de datos se utiliza el elemento `<delete>`, siempre que sea con un `id` o un dominio de búsqueda para encontrar el registro de destino.

En el capítulo 3, *Herencia - Ampliación de aplicaciones existentes*, se tuvo la necesidad de eliminar una regla de registro añadida por la aplicación de tareas pendientes. En el archivo `todo_user/security/todo_access_rules.xml` se utilizó un elemento `<delete>`, con un dominio de búsqueda para encontrar el registro a eliminar:

```
<delete model="ir.rule" search="[('id','=',ref('todo_app.todo_task_user_rule'))]" />
```

En este caso, el mismo efecto se puede lograr mediante el atributo `id` para identificar el registro a eliminar:

```
<delete model="ir.rule" id="todo_app.todo_task_user_rule" />
```

Activación de las funciones y flujos de trabajo Un archivo XML también puede ejecutar métodos durante su proceso de carga a través del elemento `<function>`. Esto puede ser usado para establecer datos de demostración y de prueba. Por ejemplo, en el módulo de miembros se utiliza para crear facturas de demostración de membresía:

```
<function model="res.partner" name="create_membership_invoice"
          eval="(ref('base.res_partner_2'), ref('membership_0'), {'amount':180})" />
```

Esto llama al método `create_membership_invoice()` del modelo `res.partner`. Los argumentos se pasan como una tupla en el atributo `eval`. En este caso tiene una tupla con tres argumentos: el ID de socio, la identificación de membresía y un diccionario que contiene el importe de la factura.

Otra forma en que los archivos de datos XML pueden realizar acciones es mediante la activación de los flujos de trabajo Odoo, a través del elemento `<workflow>`.

Los flujos de trabajo pueden, por ejemplo, cambiar el estado de un pedido de cliente o convertirlo en una factura. He aquí un ejemplo tomado del módulo de venta, la conversión de un proyecto de orden de ventas para el estado confirmado:

```
<workflow model="sale.order" ref="sale_order_4" action="order_confirm" />
```

A estas alturas, `model` se explica por sí mismo, y `ref` identifica la instancia de flujo de trabajo sobre la cual esta actuando. `action` es la señal del flujo de trabajo enviada a la instancia de flujo de trabajo.

2.4.3 Resumen

Ha aprendido todo lo necesario sobre la serialización de datos, y ganado una mejor comprensión de los aspectos de XML que vio en los capítulos anteriores.

También paso algún tiempo comprendiendo los identificadores externos, un concepto central para el manejo de datos en general, y para las configuraciones de módulo en particular.

Los archivos de datos XML se explicaron en detalle. Aprendió sobre las distintas opciones disponibles para establecer los valores de los campos y también para realizar acciones como eliminar registros y llamar a métodos de modelo.

Los archivos CSV y las características de importación / exportación de datos también fueron explicadas. Estas son herramientas valiosas para la configuración inicial de Odoo o para la edición masiva de datos.

En el siguiente capítulo se estudiará con detalle cómo construir modelos Odoo y posteriormente obtener más información sobre la construcción de sus interfaces de usuario.

2.5 Capítulo 5 - Modelos

2.5.1 Modelos – Estructura de los Datos de la Aplicación

En los capítulos anteriores, vio un resumen de extremo a extremo sobre la creación de módulos nuevos para Odoo. En el Capítulo 2, se construyo una aplicación totalmente nueva, y en el Capítulo 3, exploro la herencia y como usarla para crear un módulo de extensión para su aplicación. En el Capítulo 4, se discute como agregar datos iniciales y de demostración a sus módulos.

En estos resúmenes, se toco todas las capas que componen el desarrollo de aplicaciones *"backend"* para Odoo. Ahora, en los siguientes capítulos, es hora de explicar con más detalle todas estas capas que conforman una aplicación: modelos, vistas, y lógica de negocio.

En este capítulo, aprenderá como diseñar las estructuras de datos que soportan una aplicación, y como representar las relaciones entre ellas.

Organizar las características de las aplicaciones en módulos

Como hizo anteriormente, usara un ejemplo para ayudar a explicar los conceptos. Una de las mejores cosas de Odoo es tener la capacidad de tomar una aplicación o módulo existente y agregar, sobre este, las características que necesite. Así que continuara mejorando sus módulos to-do, y pronto formaran una aplicación completa!

Es una buena práctica dividir las aplicaciones Odoo en varios módulos pequeños, cada uno responsable de una característica específica. Esto reduce la complejidad general y hace el mantenimiento y la actualización más fácil.

El problema de tener que instalar todos esos módulos individuales puede ser resuelto proporcionando un módulo de la aplicación que empaquete todas esas características, a través de sus dependencias. Para ilustrar este enfoque implementara las características adicionales usando módulos to-do nuevos.

Introducción al módulo todo_ui

En el capítulo anterior, primero creo una aplicación para tareas por hacer personales, y luego la aplico para que las tareas por hacer pudieran ser compartidas con otras personas.

Ahora querrá llevar a su aplicación a otro nivel agregándole una pizarra `kanban` y otras mejoras en la interfaz. La pizarra `kanban` nos permitirá organizar las tareas en columnas, de acuerdo a sus estados, como En Espera, Lista, Iniciada o Culminada.

Comenzara agregando la estructura de datos para permitir esa visión. Necesitara agregar los estados y sería bueno si añade soporte para las etiquetas, permitiendo que las tareas estén organizadas por categoría.

La primera cosa que tiene que comprender es como su data estará estructurada para que pueda diseñar los Modelos que la soportan. Ya tiene la entidad central: las tareas por hacer. Cada tarea estará en un estado, y las tareas pueden tener una o más etiquetas. Esto significa que necesitara agregar dos modelos adicionales, y tendrán estas relaciones:

- Cada tarea tiene un estado, y puede haber muchas tareas en un estado.
- Cada tarea puede tener muchas etiquetas, y cada etiqueta puede estar en muchas tareas.

Esto significa que las tareas tiene una relación muchos a uno con los estados, y una relación muchos a muchos con las etiquetas. Por otra parte, las relaciones inversas son: los estados tiene una relación uno a muchos con las tareas y las etiquetas tienen una relación muchos a muchos con las tareas.

Comenzara creando el módulo nuevo `todo_ui` y agregara los estados y los modelos de etiquetas.

Ha estado usado el directorio `~/odoo-dev/custom-addons/` para alojar sus módulos. Para crear el módulo nuevo junto a los existentes, podrá usar estos comandos en la terminal:

```
$ cd ~/odoo-dev/custom-addons
$ mkdir todo_ui
$ cd todo_ui
$ touch __openerp__.py
$ touch todo_model.py
$ echo "import todo_model" > __init__.py
```

Luego, debe editar el archivo manifiesto `__openerp__.py` con este contenido:

```
{
    'name': 'User interface improvements to the To-Do app',
    'description': 'User friendly features.',
    'author': 'Daniel Reis',
    'depends': ['todo_app']
}
```

Note que depende de `todo_app` y no de `todo_user`. En general, es buena idea mantener los módulos tan independientes como sea posible. Cuando un módulo aguas arriba es modificado, puede impactar todos los demás

módulos que directa o indirectamente dependen de el. Es mejor si puede mantener al mínimo el número de dependencias, e igualmente evitar concentrar un gran número de dependencias, como: `todo_ui` → `todo_user` → `todo_app` en este caso.

Ahora podrá instalar el módulo en su base de datos de trabajo y comenzar con los modelos.

2.5.2 Crear modelos

Para que las tareas por hacer tengan una pizarra `kanban`, necesita estados. Los estados son columnas de la pizarra, y cada tarea se ajustará a una de esas columnas.

Agregue el siguiente código al archivo `todo_ui/todo_model.py`:

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

class Tag(models.Model):
    _name = 'todo.task.tag'
    name = fields.Char('Name', 40, translate=True)

class Stage(models.Model):
    _name = 'todo.task.stage'
    _order = 'sequence,name'
    _rec_name = 'name'  # predeterminado
    _table = 'todo_task_stage' # predeterminado
    name = fields.Char('Name', 40, translate=True)
    sequence = fields.Integer('Sequence')
```

Aquí, crea los dos modelos nuevos, a los cuales, hará referencia en las tareas por hacer.

Enfocándose en los estados de las tareas, tiene una clase Python, Stage, basada en la clase `models.Model`, que define un modelo nuevo, `todo.task.stage`. También defina dos campos, `name` y `sequence`. Podrá ver algunos atributos del modelo, (con el guión bajo, _, como prefijo) esto es nuevo para nosotros. Dele una mirada más profunda.

Atributos del modelo

Las clases del modelo pueden tener atributos adicionales usados para controlar alguno de sus comportamientos:

- `_name`: Este es el identificador interno para el modelo que esta creando.
- `_order`: Este fija el orden que será usado cuando se navega por los registros del modelo. Es una cadena de texto que es usada como una clausula SQL `order by`, así que puede ser cualquier cosa permitida.
- `_rec_name`: Este indica el campo a usar como descripción del registro cuando se hace referencia a él desde campos relacionados, como una relación muchos a uno. De forma predeterminada usa el campo `name`, el cual esta frecuentemente presente en los modelos. Pero este atributo le permite usar cualquier otro campo para este propósito.
- `_table`: Este es el nombre de la tabla de la base de datos que soporta el modelo. Usualmente, se deja para que sea calculado automáticamente, y es el nombre del modelo con el carácter de piso bajo (_) que reemplaza a los puntos. Pero puede ser configurado para indicar un nombre de tabla específico.

Para completar, también podrá tener atributos `inherit` e `_inherits`, como se explicara en el Capítulo 3.

Modelos y clases Python

Los modelos de Odoo son representados por las clases Python. En el código precedente, tiene una clase Python llamada Stage, basada en la clase `models.Model`, usada para definir el modelo nuevo `todo.task.stage`.

Los modelos de Odoo son mantenidos en un registro central, también denominado como piscina - pool - en las versiones anteriores. Es un diccionario que mantiene las referencias de todas las clases de modelos disponibles en

la instancia, a las cuales se les puede hacer referencia por el nombre del modelo. Específicamente, el código en un método del modelo puede usar `self.env['x]` o `self.env.get('x')` para obtener la referencia a la clase que representa el modelo x.

Puede observar que los nombres del modelo son importantes ya que son la llave para acceder al registro. La convención para los nombres de modelo es usar una lista de palabras en minúscula unidas con puntos, como `todo.task.stage`. Otros ejemplos pueden verse en los módulos raíz de Odoo `project.project`, `project.task` o `project.task.type`.

Debe usar la forma singular: `todo.task` en vez de `todo.tasks`. Por cuestiones históricas se pueden encontrar módulos raíz, que no sigan dicha convención, como `res.users`, pero no es la norma.

Los nombres de modelo deben ser únicos. Debido a esto, la primera palabra deberá corresponder a la aplicación principal con la cual esta relacionada el módulo. En su ejemplo, es "todo". De los módulos raíz tiene, por ejemplo, `project`, `crm`, o `sale`.

Por otra parte, las clases Python, son locales para el archivo Python en la cual son declaradas. El identificador usado en ellas es solo significativo para el código en ese archivo.

Debido a esto, no se requiere que los identificadores de clase tengan como prefijo a la aplicación principal a la cual están relacionados. Por ejemplo, no hay problema en llamar simplemente Stage a su clase para el modelo `todo.task.stage`. No hay riesgo de colisión con otras posibles clases con el mismo nombre en otros módulos.

Se pueden usar dos convenciones diferentes para los identificadores de clase: **snake_case** o **CamelCase**. Históricamente, el código Odoo ha usado el snake_case, y es aún muy frecuente encontrar clases que usan esa convención. Pero la tendencia actual en usar CamelCase, debido a que es el estándar definido para Python por la convenciones de codificación PEP8. Puede haber notado que esta usando esta última forma.

Modelos transitorios y abstractos

En el código precedente, y en la vasta mayoría de los modelos Odoo, las clases están basadas en el clase `models.Model`. Este tipo de modelos tienen bases de datos persistentes: las tablas de las bases de datos son creadas para ellos y sus registros son almacenados hasta que son borrados explícitamente.

Pero Odoo proporciona otros dos tipos de modelo: modelos Transitorios y Abstractos.

Los **modelos transitorios** están basados en la clase `models.TransientModel` y son usados para interacción tipo asistente con el usuario. Sus datos son aún almacenados en la base de datos, pero se espera que sea temporal. Un proceso de reciclaje limpia periódicamente los datos viejos de esas tablas.

Los **modelos abstractos** están basados en la clase `models.AbstractModel` y no tienen almacén vinculado a ellos. Actúan como una característica de re-uso configurada para ser mezclada con otros modelos. Esto es hecho usando las capacidades de herencia de Odoo.

Inspeccionar modelos existentes

La información sobre los modelos y los campos creados con clases Python esta disponible a través de la interfaz. En el menú principal de **Configuración**, seleccione la opción de menú **Técnico > Estructura de base de datos > Modelos**. Allí, encontrará la lista de todos los modelos disponibles en la base de datos. Al hacer clic en un modelo de la lista se abrirá un formulario con sus detalles.

Esta es una buena herramienta para inspeccionar la estructura de un Modelo, ya que se tiene en un solo lugar el resultado de todas las adiciones que pueden venir de diferentes módulos. En este caso, como puede observar en el campo **En los módulos**, en la parte superior derecha, las definiciones de `todo.task` vienen de los módulos `todo_app` y `todo_user`.

En el área inferior, tiene disponibles algunas etiquetas informativas: una referencia rápida de los Campos del modelo, los Derechos de Acceso concedidos, y también lista las Vistas disponibles para este modelo.

Podrá encontrar el Identificador Externo del modelo, activando el **Menú de Desarrollo** y accediendo a la opción **Ver metadatos**. Estos son generados automáticamente pero bastante predecibles: para el modelo `todo.task`, el Identificador Externo es `model_todo_task`.

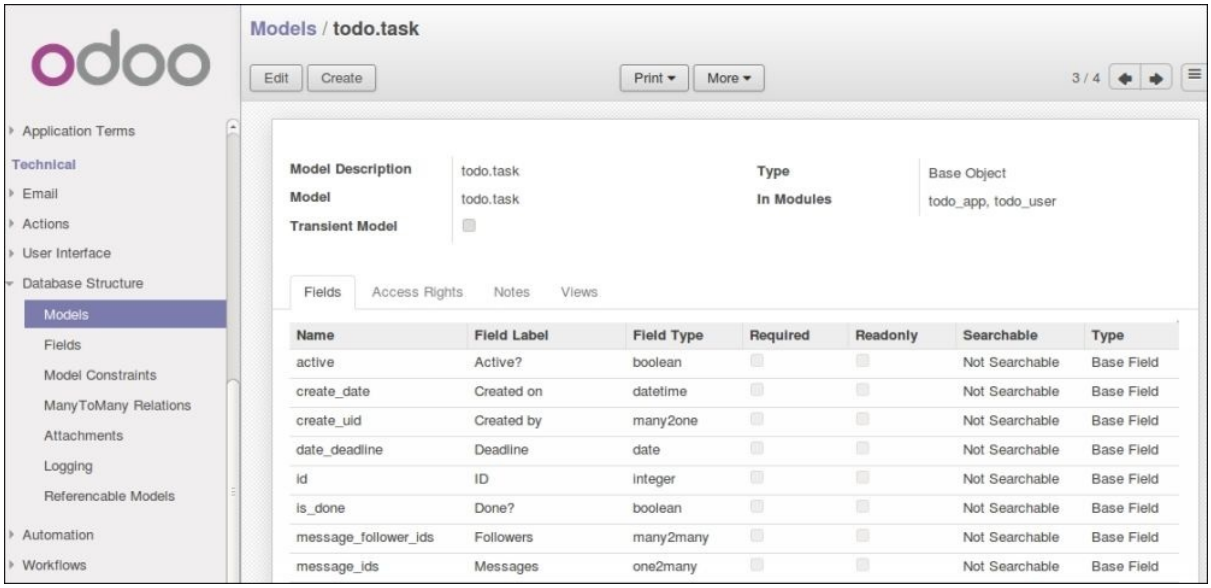

Figura 2.17: Gráfico 5.1 - Vista de la estructura de base de datos del modelo todo.task

Truco: Los formularios del Modelo pueden ser editados! Es posible crear y modificar modelos, campos y vistas desde aquí. Puede usar esto para construir prototipos antes de colocarlos definitivamente dentro de los propios modelos.

2.5.3 Crear campos

Después de crear un modelo nuevo, el siguiente paso es agregar los campos. Va a explorar diferentes tipos de campos disponibles en Odoo.

Tipos básicos de campos

Ahora tiene un modelo Stage y va a ampliarlo para agregar algunos campos adicionales. Debe editar el archivo `todo_ui/todo_model.py`, removiendo algunos atributos innecesarios incluidos antes con propósitos descriptivos:

```
class Stage(models.Model):
    _name  = 'todo.task.stage'
    _order = 'sequence,name'

    # Campos de cadena de caracteres:
    name  = fields.Char('Name',40)
    desc  = fields.Text('Description')
    state = fields.Selection(
        [
            ('draft','New'),
            ('open','Started'),
            ('done','Closed')
        ], 'State')
    docs  = fields.Html('Documentation')

    # Campos numéricos:
    sequence      = fields.Integer('Sequence')
    perc_complete = fields.Float('% Complete',(3,2))

    # Campos de fecha:
```

```
    date_effective = fields.Date('Effective Date')
    date_changed   = fields.Datetime('Last Changed')

    # Otros campos:
    fold  = fields.Boolean('Folded?')
    image = fields.Binary('Image')
```

Aquí tiene un ejemplo de tipos de campos no relacionales disponibles en Odoo, con los argumentos básicos esperados por cada función. Para la mayoría, el primer argumento es el título del campo, que corresponde al atributo palabra clave de cadena. Es un argumento opcional, pero se recomienda colocarlo. De lo contrario, sera generado automáticamente un título por el nombre del campo.

Existe una convención para los campos de fecha que usa `date` como prefijo para el nombre. Por ejemplo, debería usar `date_effective` en vez de `effective_date`. Esto también puede aplicarse a otros campos, como `amount_`, `price_` o `qty_`.

Algunos otros argumentos están disponibles para la mayoría de los tipos de campo:

- `Char`, acepta un segundo argumento opcional, `size`, que corresponde al tamaño máximo del texto. Es recomendable usarlo solo si se tiene una buena razón.
- `Text`, se diferencia de `Char` en que puede albergar texto de varias líneas, pero espera los mismos argumentos.
- `Selecction`, es una lista de selección desplegable. El primer argumento es la lista de opciones seleccionables y el segundo es la cadena de título. La lista de selección es una tupla `('value', 'Title')` para el valor almacenado en la base de datos y la cadena de descripción correspondiente. Cuando se amplía a través de la herencia, el argumento `selection_add` puede ser usado para agregar opciones a la lista de selección existente.
- `Html`, es almacenado como un campo de texto, pero tiene un manejo específico para presentar el contenido HTML en la interfaz.
- `Integer`, solo espera un argumento de cadena de texto para el campo de título.
- `Float`, tiene un argumento opcional, una tupla `(x,y)` con los campos de precisión: 'x' como el número total de dígitos; 'y' representa los dígitos decimales.
- `Date` y `Datetime`, estos datos son almacenados en formato UTC. Se realizan conversiones automáticas, basadas en las preferencias del usuario, disponibles a través del contexto de la sesión de usuario. Esto es discutido con mayor detalle en el Capítulo 6.
- `Boolean`, solo espera sea fijado el campo de título, incluso si es opcional.
- `Binary` también espera este único argumento.

Además de estos, también existen los campos relacionales, los cuales serán introducidos en este mismo capítulo. Pero por ahora, hay mucho que aprender sobre los tipos de campos y sus atributos.

Atributos de campo comunes

Los campos también tienen un conjunto de atributos los cuales podrá usar, y se explicara aquí con más detalle:

- `string`, es el título del campo, usado como su etiqueta en la UI. La mayoría de las veces no es usado como palabra clave, ya que puede ser fijado como un argumento de posición.
- `default`, fija un valor predefinido para el campo. Puede ser un valor estático o uno fijado anticipadamente, pudiendo ser una referencia a una función o una expresión `lambda`.
- `size`, aplica solo para los campos `Char`, y pueden fijar el tamaño máximo permitido.
- `translate`, aplica para los campos de texto, `Char`, `Text` y `Html`, hacen que los campos puedan ser traducidos: puede tener varios valores para diferentes idiomas.
- `help`, proporciona el texto de ayuda desplegable mostrado a los usuarios.
- `readonly = True`, hace que el campo no pueda ser editado en la interfaz.

- `required = True`, hace que el campo sea obligatorio.
- `index = True`, creara un índice en la base de datos para el campo.
- `copy = False`, hace que el campo sea ignorado cuando se usa la función Copiar. Los campos no relacionados de forma predeterminada pueden ser copiados.
- `groups`, permite limitar la visibilidad y el acceso a los campos solo a determinados grupos. Es una lista de cadenas de texto separadas por comas, que contiene los ID XML del grupo de seguridad.
- `states`, espera un diccionario para los atributos de la UI dependiendo de los valores de estado del campo. Por ejemplo: `states={'done':[('readonly', True)]}`. Los atributos que pueden ser usados son, `readonly`, `required` e `invisible`.

Para completar, a veces son usados dos atributos más cuando se actualiza entre versiones principales de Odoo:

- `deprecated = True`, registra un mensaje de alerta en cualquier momento que el campo sea usado.
- `oldname = 'field'`, es usado cuando un campo es re-nombrado en una versión nueva, permitiendo que la data en el campo viejo sea copiada automáticamente dentro del campo nuevo.

Nombres de campo reservados

Unos cuantos nombres de campo están reservados para ser usados por el ORM:

- `id`, es un número generado automáticamente que identifica de forma única a cada registro, y es usado como clave primaria en la base de datos. Es agregado automáticamente a cada modelo.

Los siguientes campos son creados automáticamente en los modelos nuevos, a menos que sea fijado el atributo `_log_access=False`:

- `create_uid`, para el usuario que crea el registro.
- `created_date`, para la fecha y la hora en que el registro es creado.
- `write_uid`, para el último usuario que modifica el registro.
- `write_date`, para la última fecha y hora en que el registro fue modificado.

Esta información esta disponible desde el cliente web, usando el **menú de Desarrollo** y seleccionando la opción **Ver metadatos**.

Hay algunos efectos integrados que esperan nombres de campo específicos. Debe evitar usarlos para otros propósitos que aquellos para los que fueron creados. Algunos de ellos incluso están reservados y no pueden ser usados para ningún otro propósito:

- `name`, es usado de forma predeterminada como el nombre del registro que será mostrado. Usualmente es un `Char`, pero se permiten otros tipos de campos. Puede ser sobre escrito configurando el atributo `_rec_name` del modelo.
- `active` (tipo `Boolean`), permite desactivar registros. Registros con `active==False` serán excluidos automáticamente de las consultas. Para acceder a ellos debe ser agregada la condición `('active','=', False)` al dominio de búsqueda o agregar `'active_test':False` al contexto actual.
- `sequence` (tipo `Integer`), si esta presente en una vista de lista, permite definir manualmente el orden de los registros. Para funcionar correctamente debe estar también presente en el `_order` del modelo.
- `state` (tipo `Selection`), representa los estados básicos del ciclo de vida del registro, y puede ser usado por el atributo `field` del estado para modificar de forma dinámica la vista: algunos campos de formulario pueden ser de solo lectura, requeridos o invisibles en estados específicos del registro.
- `parent_id`, `parent_left`, y `parent_right`; tienen significado especial para las relaciones jerárquicas padre/hijo. En un momento se discutirá con mayor detalle.

Hasta ahora ha discutido los valores escalares de los campos. Pero una buena parte de una estructura de datos de la aplicación es sobre la descripción de relaciones entre entidades. Vea algo sobre esto ahora.

2.5.4 Relaciones entre modelos

Viendo su diseño del módulo, tiene estas relaciones:

- Cada tarea tiene un estado – esta es una relación muchos a uno, también conocida como una clave foránea. La relación inversa es de uno a muchos, que significa que cada estado puede tener muchas tareas.
- Cada tarea puede tener muchas etiquetas – esta es una relación muchos a muchos. La relación inversa, obviamente, es también una relación muchos a muchos, debido a que cada etiqueta puede también tener muchas tareas.

Agregue los campos de relación correspondientes al archivo `todo_ui/todo_model.py`:

```
class TodoTask(models.Model):
    _inherit = 'todo.task'
    stage_id = fields.Many2one('todo.task.stage', 'Stage')
    tag_ids = fields.Many2many('todo.task.tag', string='Tags')
```

El código anterior muestra la sintaxis básica para estos campos. Configurando el modelo relacionado y el campo de título. La convención para los nombres de campo relacionales es agregar a los nombres de campos `_id` o `_ids`, para las relaciones de uno y muchos, respectivamente.

Como ejercicio puede intentar agregar en los modelos relacionados, las relaciones inversas correspondientes: La relación inversa de `Many2one` es un campo `One2many` en los estados: cada estado puede tener muchas tareas. Debería agregar este campo a la clase Stage. La relación inversa de `Many2many` es también un campo `Many2many` en las etiquetas: cada etiqueta puede ser usada en muchas tareas.

Vea con mayor detalle las definiciones de los campos relacionales.

Relaciones muchos a uno

`Many2one`, acepta dos argumentos de posición: el modelo relacionado (que corresponde al argumento de palabra clave del `comodel`) y la cadena de título. Este crea un campo en la tabla de la base de datos con una clave foránea a la tabla relacionada.

Algunos nombres adicionales de argumentos también están disponibles para ser usados con estos tipos de campo:

- `ondelete`, define lo que pasa cuando el registro relacionado es eliminado. De forma predeterminada esta fijado como `null`, lo que significa que al ser eliminado el registro relacionado se fija a un valor vacío. Otros valores posibles son `restrict`, que arroja un error que previene la eliminación, y `cascade` que también elimina este registro.
- `context` y `domain`, son significativos para las vistas del cliente. Pueden ser configurados en el modelo para ser usados de forma predeterminada en cualquier vista donde sea usado el campo. Estos serán explicados con más detalle en el Capítulo 6.
- `auto_join = True`, permite que el ORM use uniones SQL haciendo búsquedas usando esta relación. De forma predeterminada esto esta fijado como `False` para reforzar las reglas de seguridad. Si son usadas uniones, las reglas de seguridad serán pasadas por alto, y el usuario podrá tener acceso a los registros relacionados que las reglas de seguridad no le permitirían, pero las consultas SQL serán más eficientes y se ejecutarán con mayor rapidez.

Relaciones muchos a muchos

La forma más simple de la relación `Many2many` acepta un argumento para el modelo relacionado, y es recomendable también proporcionar el argumento de cadena con el título del campo.

En el nivel de base de datos, esto no agrega ninguna columna a las tablas existentes. Por el contrario, automáticamente crea una tabla nueva de relación de solo dos campos con las claves foráneas de las tablas relacionadas. El nombre de la tabla de relación es el nombre de ambas tablas unidos por un símbolo de guión bajo (_) con `_rel` anexado.

Estas configuraciones predeterminadas pueden ser sobre escritas manualmente. Una forma de hacerlo es usar la forma larga para la definición del campo:

```
# TodoTask class: Task <-> relación Tag (forma larga):
tag_ids = fields.Many2many('todo.task.tag', # modelo relacionado
                           'todo_task_tag_rel', # nombre de la tabla de relación
                           'task_id', # campo para "este" registro
                           'tag_id', # campo para "otro" registro
                           string='Tasks')
```

Note que los argumentos adicionales son opcionales. Podrá simplemente fijar el nombre para la tabla de relación y dejar que los nombres de los campos usen la configuración predeterminada.

Si prefiere, puede usar la forma larga usando los argumentos de palabra clave:

```
# TodoTask class: Task  <-> relación Tag (forma larga):
tag_ids = fields.Many2many(comodel_name='todo.task.tag', # modelo relacionado
                           relation='todo_task_tag_rel', # nombre de la tabla de relación
                           column1='task_id', # campo para "este" registro
                           column2='tag_id', # campo para "otro" registro
                           string='Tasks')
```

Como los campos muchos a uno, los campos muchos a muchos también soportan los atributos de palabra clave de dominio y contexto.

En algunas raras ocasiones tendrá que usar estas formas largas para sobre escribir las configuraciones automáticas predeterminadas, en particular, cuando los modelos relacionados tengan nombres largos o cuando necesite una segunda relación muchos a muchos entre los mismos modelos.

Truco: Los nombres de las tablas PostgreSQL tienen 63 caracteres como límite, y esto puede ser un problema si la tabla de relación generada automáticamente excede ese limite. Este es uno de los casos cuando tendrá que configurar manualmente el nombre de la tabla de relación usando el atributo `relation`.

Lo inverso a la relación `Many2many` es también un campo `Many2many`. Si también agrega un campo `Many2many` a las etiquetas, Odoo infiere que esta relación de muchos a muchos es la inversa a la del modelo de tareas.

La relación inversa entre tareas y etiquetas puede ser implementada así:

```
# class Tag(models.Model): #
    _name = 'todo.task.tag'

    #Tag class relación a Tasks:
    task_ids = fields.Many2many('todo.task', # modelo relacionado
                                string='Tasks')
```

Relaciones inversas de uno a muchos

La inversa de `Many2many` puede ser agregada al otro extremo de la relación. Esto no tiene un impacto real en la estructura de la base de datos, pero le permite navegar fácilmente desde "un" lado a "muchos" lados de los registros. Un caso típico es la relación entre un encabezado de un documento y sus líneas.

En su ejemplo, con una relación inversa `One2many` en estados, fácilmente podrá listar todas las tareas que se encuentran en un estado. Para agregar esta relación inversa a los estados, agregue el código mostrado a continuación:

```
# class Stage(models.Model): #
    _name = 'todo.task.stage'

    #Stage class relación con Tasks:
    tasks = fields.One2many('todo.task', # modelo relacionado
                            'stage_id', # campo para "este" en el modelo relacionado
                            'Tasks in this stage')
```

`One2many` acepta tres argumentos de posición: el modelo relacionado, el nombre del campo en aquel modelo que referencia este registro, y la cadena de título. Los dos primeros corresponden a los argumentos `comodel_name` e `inverse_name`.

Los parámetros adicionales disponibles son los mismos que para el muchos a uno: contexto, dominio, `ondelete` (aquí actúa en el lado "muchos" de la relación), y `auto_join`.

Relaciones jerárquicas

Las relaciones padre-hijo pueden ser representadas usando una relación `Many2one` al mismo modelo, para dejar que cada registro haga referencia a su padre. Y la inversa `One2many` hace más fácil para un padre mantener el registro de sus hijos.

Odoo también provee soporte mejorado para estas estructuras de datos jerárquicas: navegación más rápida a través de árboles hermanos, y búsquedas más simples con el operador `child_of` en las expresiones de dominio.

Para habilitar esas características debe configurar el atributo `_parent_store` y agregar los campos de ayuda: `parent_left` y `parent_right`. Tenga en cuenta que estas operaciones adicionales traen como consecuencia penalizaciones en materia de almacenamiento y ejecución, así que es mejor usarlo cuando se espere ejecutar más lecturas que escrituras, como es el caso de un árbol de categorías.

Revisando el modelo de etiquetas definido en el archivo `todo_ui/todo_model.py`, ahora edite para que luzca así:

```
class Tags(models.Model):
    _name          = 'todo.task.tag'
    _parent_store = True
    #_parent_name  = 'parent_id'
    name = fields.Char('Name')
    parent_id      = fields.Many2one('todo.task.tag','Parent Tag', ondelete='restrict')
    parent_left    = fields.Integer('Parent Left', index=True)
    parent_right   = fields.Integer('Parent  Right', index=True)
```

Aquí tiene un modelo básico, con un campos `parent_id` que referencia al registro padre, y el atributo adicional `_parent_store` para agregar soporte a búsquedas jerárquicas.

Se espera que el campo que hace referencia al padre sea nombrado `parent_id`. Pero puede usarse cualquier otro nombre declarándolo con el atributo `_parent_name`.

También, es conveniente agregar un campo con el hijo directo del registro:

```
child_ids = fields.One2many('todo.task.tag', 'parent_id', 'Child Tags')
```

Hacer referencia a campos usando relaciones dinámicas

Hasta ahora, los campos de relación que ha visto puede solamente hacer referencia a un modelo. El tipo de campo `Reference` no tiene esta limitación y admite relaciones dinámicas: el mismo campo es capaz de hacer referencia a más de un modelo.

Podrá usarlo para agregar un campo, "Refers to", a Tareas por Hacer que pueda hacer referencia a un User o un Partner:

```
# class TodoTask(models.Model):
    refers_to = fields.Reference([
        ('res.user', 'User'),('res.partner', 'Partner')
    ], 'Refers to')
```

Puede observar que la definición del campo es similar al campo `Selection`, pero aquí la lista de selección contiene los modelos que pueden ser usados. En la interfaz, el usuario seleccionará un modelo de la lista, y luego elegirá un registro de ese modelo.

Esto puede ser llevado a otro nivel de flexibilidad: existe una tabla de configuración de Modelos Referenciables para configurar los modelos que pueden ser usados en campos `Reference`. Esta disponible en el menú **Configuración > Técnico > Estructuras de base de datos**. Cuando se crea un campo como este podrá ajustarlo para que use cualquier modelo registrado allí, con la ayuda de la función `referencable_models()` en el módulo `openerp.addons.res.res_request`. En la versión 8 de Odoo, todavía se usa la versión antigua de la API, así que necesitara empaquetarlo para usarlo con la API nueva:

```
from openerp.addons.base.res import res_request

def referencable_models(self):
    return res_request.referencable_model(self, self.env.cr, self.env.uid, context=self.env.conte
```

Usando el código anterior, la versión revisada del campo "Refers to" sera así:

```
# class TodoTask(models.Model):
    refers_to = fields.Reference(referencable_models, 'Refers to')
```

2.5.5 Campos calculados

Los campos pueden tener valores calculados por una función, en vez de simplemente leer un valor almacenado en una base de datos. Un campo calculado es declarado como un campo regular, pero tiene el argumento `compute` adicional con el nombre de la función que se usará para calcularlo.

En la mayoría de los casos los campos calculados involucran alguna lógica de negocio, por lo tanto este tema se desarrollara con más profundidad en el Capítulo 7. Igual podrá explicarlo aquí, pero manteniendo la lógica de negocio lo más simple posible.

Trabaje en un ejemplo: los estados tienen un campo "fold". Agregue a las tareas un campo calculado con la marca "Folded?" para el estado correspondiente.

Debe editar el modelo `TodoTask` en el archivo `todo_ui/todo_model.py` para agregar lo siguiente:

```
# class TodoTask(models.Model):
    stage_fold = fields.Boolean('Stage Folded?', compute='_compute_stage_fold')

    @api.one
    @api.depends('stage_id.fold')
    def _compute_stage_fold(self):
        self.stage_fold = self.stage_id.fold
```

El código anterior agrega un campo nuevo `stage_fold` y el método `_compute_stage_fold` que sera usado para calcular el campo. El nombre de la función es pasado como una cadena, pero también es posible pasarla como una referencia obligatoria (el identificador de la función son comillas).

Debido a que esta usando el decorador `@api.one`, `self` tendrá un solo registro. Si en vez de esto usa `@api.multi`, representara un conjunto de registros y su código necesitará gestionar la iteración sobre cada registro.

El `@api.depends` es necesario si el calculo usa otros campos: le dice al servidor cuando re-calcular valores almacenados o en cache. Este acepta uno o más nombres de campo como argumento y la notación de puntos puede ser usada para seguir las relaciones de campo.

Se espera que la función de calculo asigne un valor al campo o campos a calcular. Si no lo hace, arrojara un error. Debido a que `self` es un objeto de registro, su calculo es simplemente para obtener el campo "Folded?" usando `self.stage_id.fold`. El resultado es conseguido asignando ese valor (escribiéndolo) en el campo calculado, `self.stage_fold`.

No trabajara aún en las vistas para este módulo, pero puede hacer una edición rápida al formulario de tareas para confirmar si el campo calculado esta funcionando como es esperado: usando el menú de **Desarrollo** escoja la opción **Editar Vista** y agregue el campo directamente en el XML del formulario. No se preocupe: será reemplazado por una vista limpia del módulo en la próxima actualización.

Buscar y escribir en campos calculados

El campo calculado que acabo de crear puede ser leído, pero no se puede realizar una búsqueda ni escribir en el. Esto puede ser habilitado proporcionando funciones especiales para esto. A lo largo de la función de calculo también podrá colocar una función de búsqueda, que implemente la lógica de búsqueda, y la función inversa, que implemente la lógica de escritura.

Para hacer esto, su declaración de campo calculado se convertirá en esto:

```
# class TodoTask(models.Model):
    stage_fold = fields.Boolean
        string   = 'Stage Folded?',
        compute  ='_compute_stage_fold',
        # store=False) # predeterminado
        search   ='_search_stage_fold',
        inverse  ='_write_stage_fold')
```

Las funciones soportadas son:

```
def _search_stage_fold(self, operator, value):
    return [('stage_id.fold', operator, value)]

def _write_stage_fold(self):
    self.stage_id.fold = self.stage_fold
```

La función de búsqueda es llamada en cuanto es encontrada en este campo una condición `(campo, operador, valor)` dentro de una expresión de dominio de búsqueda.

La función inversa realiza la lógica reversa del cálculo, para hallar el valor que sera escrito en el campo de origen. En su ejemplo, es solo escribir en `stage_id.fold`.

Guardar campos calculados

Los valores de los campos calculados también pueden ser almacenados en la base de datos, configurando `store` a `True` en su definición. Estos serán calculados cuando cualquiera de sus dependencias cambie. Debido a que los valores ahora estarán almacenados, pueden ser buscados como un campo regular, entonces no es necesaria una función de búsqueda.

2.5.6 Campos relacionados

Los campos calculados que implemento en la sección anterior son un caso especial que puede ser gestionado automáticamente por Odoo. El mismo efecto puede ser logrado usando campos Relacionados. Estos hacen disponibles, de forma directa en un módulo, los campos que pertenecen a un modelo relacionado, que son accesibles usando la notación de puntos. Esto posibilita su uso en los casos en que la notación de puntos no pueda usarse, como los formularos de UI.

Para crear un campo relacionado, declare un campo del tipo necesario, como en los campos calculados regulares, y en vez de calcularlo, use el atributo `related` indicando la cadena de notación por puntos para alcanzar el campo deseado.

Las tareas por hacer están organizadas en estados personalizables y a su vez esto forma un mapa en los estados básicos. Los pondrá disponibles en las tareas, y usara esto para la lógica del lado del cliente en la próximo capítulo.

Agregara un campo calculado en el modelo tarea, similar a como hizo a “stage_fold”, pero ahora usando un campo `related`:

```
# class TodoTask(models.Model):
    stage_state = fields.Selection(
        related='stage_id.state',
        string='Stage State'
    )
```

Detrás del escenario, los campos "Related" son solo campos calculados que convenientemente implementan las funciones de búsqueda e inversa. Esto significa que podrá realizar búsquedas y escribir en ellos sin tener que agregar código adicional.

2.5.7 Restricciones del Modelo

Para reforzar la integridad de los datos, los modelos también soportan dos tipos de restricciones: SQL y Python.

Las restricciones SQL son agregadas a la definición de la tabla en la base de datos e implementadas por PostgreSQL. Son definidas usando el atributo de clase `_sql_constraints`. Este es una lista de tuplas con el nombre del identificador de la restricción, el SQL para la restricción, y el mensaje de error que se usara.

Un caso común es agregar restricciones únicas a los modelos. Suponga que no querrá permitir que el mismo usuario tenga dos tareas activas con el mismo título:

```
# class TodoTask(models.Model):
    _sql_constraints = [(
        'todo_task_name_uniq',
        'UNIQUE (name, user_id, active)',
        'Task title must be unique!'
    )]
```

Debido a que esta usando el campo `user_id` agregado por el módulo `todo_user`, esta dependencia debe ser agregada a la clave `depends` del archivo manifiesto `__openerp__.py`.

Las restricciones Python pueden usar un pedazo arbitrario de código para verificar las condiciones. La función de verificación necesita ser decorada con `@api.constrains` indicando la lista de campos involucrados en la verificación. La validación es activada cuando cualquiera de ellos es modificado, y arrojara una excepción si la condición falla:

```
from openerp.exceptions import ValidationError

# class TodoTask(models.Model):
    @api.one
    @api.constrains('name')
    def _check_name_size(self):
        if len(self.name) < 5:
            raise ValidationError('Must have 5 chars!')
```

El ejemplo anterior previene que el título de las tareas sean almacenados con menos de 5 caracteres.

2.5.8 Resumen

Vio una explicación minuciosa de los modelos y los campos, usándolos para ampliar la aplicación de Tareas por Hacer con etiquetas y estados de las tareas. Aprendió como definir relaciones entre modelos, incluyendo relaciones jerárquicas padre/hijo. Finalmente, vi ejemplos sencillos de campos calculados y restricciones usando código Python.

En el próximo capítulo, trabajara en la interfaz para las características "back-end" de ese modelo, haciéndolas disponibles para las vistas que se usan para interactuar con la aplicación.

2.6 Capítulo 6 - Vistas

2.6.1 Vistas – Diseñar la Interfaz

Este capítulo le ayudara a construir la interfaz gráfica para sus aplicaciones. Hay varios tipos disponibles de vistas y widgets. Los conceptos de contexto y dominio también juegan un papel fundamental en la mejora de la experiencia del usuario, y aprenderá más sobre esto.

El módulo `todo_ui` tiene lista la capa de modelo, y ahora necesita la capa de vista con la interfaz. Agregara elementos nuevos a la Interfaz de Usuario y modificara las vistas existentes que fueron agregadas en capítulos anteriores.

La mejor manera de modificar vistas existentes es usar la herencia, como se explico en el Capítulo 3. Sin embargo, para mejorar la claridad en la explicación, sobre escribirá las vistas existentes, y las reemplazara por unas vistas completamente nuevas. Esto hará que los temas sean más fáciles de entender y seguir.

Es necesario agregar un archivo XML nuevo al módulo, así que comience por editar el archivo manifiesto `__openerp__.py`. Necesita usar algunos campos del módulo `todo_user`, para que sea configurado como una dependencia:

```
{ 'name': 'User interface improvements to the To-Do app',
    'description': 'User friendly features.',
    'author': 'Daniel Reis',
    'depends': ['todo_user'],
    'data': ['todo_view.xml']
}
```

Comience con las opciones de menú y las acciones de ventana.

Acciones de ventana

Las acciones de ventana dan instrucciones a la interfaz del lado del cliente. Cuando un usuario hace clic en una opción de menú o en un botón para abrir un formulario, es la acción subyacente la que da instrucciones a la interfaz sobre lo que debe hacer.

Comience por crear la acción de ventana que será usada en las opciones de menú, para abrir las vistas de las tareas por hacer y de los estados. Cree el archivo de datos `todo_view.xml` con el siguiente código:

```
<?xml version="1.0"?>
<openerp>
  <data>
    <act_window id="action_todo_stage" name="To-Do Task Stages" res_model="todo.task.stage" view_
    <act_window id="todo_app.action_todo_task" name="To-Do Tasks" res_model="todo.task"
                view_mode="tree,form,calendar,gantt,graph" target="current"
                context="{'default_user_id':uid}" domain="[]" limit="80"/>
    <act_window id="action_todo_task_stage" name="To-Do Task Stages"
                res_model="todo.task.stage" src_model="todo.task" multi="False"/>
  </data>
 </openerp>
```

Las acciones de ventana se almacenan en el modelo `ir.actions.act_window`, y pueden ser definidas en archivos XML usando el acceso directo `<act_window>` que recién uso.

La primera acción abre el modelo de estados de la tarea, y solo usa los atributos básicos para una acción de ventana.

La segunda acción usa un ID en el espacio de nombre de `todo_app` para sobre escribir la acción original de tareas por hacer del módulo `todo_app`. Esta usa los atributos de acciones de ventana más relevantes.

- `name`: Este es el título mostrado en las vistas abiertas a través de esta acción.
- `res_model`: Es el identificador del modelo de destino.
- `view_mode`: Son los tipos de vista que estarán disponibles. El orden es relevante y el primero de la lista será la vista que se abrirá de forma predeterminada.
- `target`: Si es fijado como `new`, la vista se abrirá en una ventana de dialogo. De forma predeterminada esta fijado a `current`, por lo que abre la vista en el área principal de contenido.
- `context`: Este fija información de contexto en las vistas de destino, la cual puede ser usada para establecer valores predeterminados en campos o filtros activos, entre otras cosas. Verá más detalles sobre esto en este mismo capítulo.

- `domain`: Es una expresión de dominio que establece un filtro para los registros que estarán disponibles en las vistas abiertas.
- `limit`: Es el número de registros por cada página con vista de lista, 80 es el número predefinido.

La acción de ventana ya incluye los otros tipos de vista las cuales estará examinando en este capítulo: calendar, Gantt y gráfico. Una vez que estos cambios son instalados, los botones correspondientes serán mostrados en la esquina superior derecha, junto a los botones de lista y formulario. Note que esto no funcionará hasta crear las vistas correspondientes.

La tercera acción de ventana demuestra como agregar una opción bajo el botón "Más", en la parte superior de la vista. Estos son los atributos usados para realizar esto:

- `multi`: Si esta fijado a `True`, estará disponible en la vista de lista. De lo contrario, estará disponible en la vista de formulario.

Opciones de menú

Las opciones de menú se almacenan en el modelo `ir.ui.menu`, y pueden ser encontradas en el menú Configuraciones navegando a través de **Técnico > Interfaz de Usuario > Opciones de Menú**. Si busca Mensajería, verá que tiene como submenú Organizador. Con la ayuda de las herramientas de desarrollo podrá encontrar el ID del XML para esa opción de menú: la cual es `mail.mail_my_stuff`.

Reemplazará la opción de menú existente en Tareas por Hacer con un submenú que puede encontrarse navegando a través de **Mensajería > Organizador**. En el `todo_view.xml`, después de las acciones de ventana, agregue el siguiente código:

```
<menuitem id="menu_todo_task_main" name="To-Do" parent="mail.mail_my_stuff"/>
<menuitem id="todo_app.menu_todo_task" name="To-Do Tasks" parent="menu_todo_task_main"
          sequence="10" action="todo_app.action_todo_task"/>
<menuitem id="menu_todo_task_stage" name="To-Do Stages" parent="menu_todo_task_main"
          sequence="20" action="action_todo_stage"/>
```

La opción de menú "data" para el modelo `ir.ui.menu` también puede cargarse usando el elemento de acceso directo `<menuitem>`, como se uso en el código anterior.

El primer elemento del menú, "To-Do", es hijo de la opción de menú Organizador `mail.mail_my_stuff`. No tiene ninguna acción asignada, debido a que será usada como padre para las próximas dos opciones.

El segundo elemento del menú re escribe la opción definida en el módulo `todo_app` para ser re ubicada bajo el elemento *"To-Do"* del menú principal.

El tercer elemento del menú agrega una nueva opción para acceder a los estados. Necesitará un orden para agregar algunos datos que permitan usar los estados en las tareas por hacer.

Contexto y dominio

Se ha referido varias veces al contexto y al dominio. También se ha visto que las acciones de ventana pueden fijar valores en estos, y que los campos relacionales pueden usarlos en sus atributos. Ambos conceptos son útiles para proveer interfaces más sofisticadas. Vea como.

Contexto de sesión

El contexto es un diccionario que contiene datos de sesión usados por las vistas en el lado del cliente y por los procesos del servidor. Puede transportar información desde una vista hasta otra, o hasta la lógica del lado del servidor. Es usado frecuentemente por las acciones de ventana y por los campos relacionales para enviar información a las vistas abiertas a través de ellos.

Odoo estable en el contexto alguna información básica sobre la sesión actual. La información inicial de sesión puede verse así:

```
{'lang': 'en_US', 'tz': 'Europe/Brussels', 'uid': 1}
```

Tiene información del ID de usuario actual, y las preferencias de idioma y zona horaria para la sesión de usuario.

Cuando se usa una acción en el cliente, como hacer clic en un botón, se agrega información al contexto sobre los registros seleccionados actualmente:

- `active_id` es el ID del registro seleccionado en el formulario,
- `active_model` es el modelo de los registros actuales,
- `active_ids` es la lista de los ID seleccionados en la vista de árbol/lista.

El contexto también puede usarse para proveed valores predeterminados en los campos o habilitar filtros en la vista de destino.

Para fijar el valor predeterminado en el campo `user_id`, que corresponda a la sesión actual de usuario, debe usar:

```
{'default_user_id': uid}
```

Y si la vista de destino tiene un filtro llamado `filter_my_task`, podrá habilitarlo usando:

```
{'search_default_filter_my_tasks': True}
```

Expresiones de dominio

Los dominios se usan para filtrar los datos de registro. Odoo los analiza detenidamente para formar la expresión *SQL* `WHERE` usada para consultar a la base de datos.

Cuando se usa en una acción de ventana para abrir una vista, el dominio fija un filtro en los registros que estarán disponibles en esa vista. Por ejemplo, para limitar solo a las Tareas del usuario actual:

```
domain=[('user_id', '=', uid)]
```

El valor `uid` usado aquí es provisto por el contexto de sesión. Cuando se usa en un campo relacional, limitara las opciones disponibles de selección para ese campo. El filtro de dominio puede también usar valores de otros campos en la vista. Con esto podrá tener diferentes opciones disponibles dependiendo de lo seleccionado en otros campos. Por ejemplo, un campo de persona de contacto puede ser establecido para mostrar solo las personas de la compañía seleccionada previamente en otro campo.

Un dominio es una lista de condiciones, donde cada condición es una tupla `('field', 'operator', 'value')`.

El campo a la izquierda es al cual se aplicara el filtro, y puede ser usada la notación de punto en los campos relaciones.

Los operadores que pueden ser usados son:

- =, `like` para coincidencias con el valor del patrón donde el símbolo de guión bajo (_) coincida con cualquier carácter único, y `%` coincida con cualquier secuencia de caracteres.
- `like` para hacer coincidir con el patrón SQL `%value%` sensible a mayúsculas, e `ilike` para coincidencias sin sensibilidad de mayúsculas.
- Los operadores `not like` y `not ilike` hacen la operación inversa.
- `child_of` encuentra los hijos directos e indirectos, si las relaciones padre/hijo están configuradas en el modelo de destino.
- `in` y `not` verifican la inclusión en una lista. En este caso, el valor de la derecha debe ser una lista Python. Estos son los únicos operadores que pueden ser usados con valores de una lista. Un caso especial es cuando el lado izquierdo es un campo "a-muchos": aquí el operador `in` ejecuta una operación `contains`.

Están disponibles los operadores de comparación usuales:

- < menor.

- > mayor.
- <= menor o igual que.
- >= mayor o igual que.
- = igual.
- != distinto.

El valor dela derecha puede puede ser una constante o una expresión Python a ser evaluada. Lo que puede ser usado en estas expresiones depende del contexto disponible (no debe ser confundido con el contexto de sesión, discutido en la sección anterior). Existen dos posibles contextos de evaluación para los dominios: del lado del cliente y del lado del servidor.

Para los dominios de campo y las acciones de ventana, la evaluación es realizada desde el lado del cliente. El contexto de evaluación incluye aquí los campos disponibles para la vista actual, y la notación de puntos no esta disponible. Puede ser usados los valores del contexto de sesión, como `uid` y `active_id`. Están disponibles los módulo de Python `datetime` y `time` para ser usado en las operaciones de fecha y hora, y también esta disponible la función `context_today()` que devuelve la fecha actual del cliente.

Los dominios usados en las reglas de registro de seguridad y en el código Python del servidor son evaluados del lado el servidor. El contexto de evaluación tiene los campos los registros actuales disponibles, y se permite la notación de puntos. También están disponibles los registros de la sesión de usuario actual. Al usar `user.id` es equivalente a usar `uid` en el contexto de evaluación del lado del cliente.

Las condiciones de dominio pueden ser combinadas usando los operadores lógicos:

- `&` para el operador lógico `AND` (el predeterminado).
- `|` para el operador lógico `OR`.
- `!` para el operador lógico de negación.

La negación es usada antes de la condición que será negada. Por ejemplo, para encontrar todas las tareas que no pertenezca al usuario actual: `['!', ('user_id','=', uid)]`.

Los operadores lógicos `AND` y `OR` operan en las dos condiciones siguientes. Por ejemplo:

Para filtrar las tareas del usuario actual o sin un usuario *(responsable)* asignado:

```
['|', ('user_id', '=', uid), ('user_id', '=', False)]
```

Un ejemplo más complejo, usado en las reglas de registro del lado del servidor:

```
['|', ('message_follower_ids', 'in', [user.partner_id.id]), '|', ('user_id', '=', user.id), ('use
```

El dominio filtra:

- Todos los registros donde los seguidores (un campo de *muchos a muchos*) contienen al usuario actual además del resultado de la siguiente condición.
- La siguiente condición es, nuevamente, la unión de otras dos condiciones: los registros donde el `user_id` es el usuario de la sesión actual o no esta fijado.

2.6.2 Vistas de Formulario

Como se ha visto en capítulos anteriores, las vistas de formulario cumplir con una diseño simple o un diseño de documento de negocio, similar a un documento en papel.

Ahora verá como diseñar vistas de negocio y usar los elementos y widgets disponibles. Esto es hecho usualmente heredando la vista base. Pero para hacer el código más simple, creará una vista completamente nueva para las tareas por hacer que sobre escribirá la definida anteriormente.

De hecho, el mismo modelo puede tener diferentes vistas del mismo tipo. Cuando se abre un tipo de vista para un modelo a través de una acción, se selecciona aquella con la prioridad más baja. O como alternativa, la acción puede especificar exactamente el identificador de la vista que se usará. La acción que definió al principio de

este capítulo solo hace eso; el `view_id` le dice a la acción que use específicamente el formulario con el ID `view_form_todo_task_ui`. Esta es la vista que creará a continuación.

Vistas de negocio

En una aplicación de negocios podrá diferenciar los datos auxiliares de los datos principales del negocio. Por ejemplo, en su aplicación los datos principales son las tareas por hacer, y las etiquetas y los estados son tablas auxiliares.

Estos modelos de negocio pueden usar diseños de vista de negocio mejorados para mejorar la experiencia del usuario. Si vuelve a ejecutar la vista del formulario de tarea agregada en el Capítulo 2, notará que ya sigue la estructura de vista de negocio.

La vista de formulario correspondiente debe ser agregada después de las acciones y los elementos del menú, que agrego anteriormente, y su estructura genérica es esta:

```
<record id="view_form_todo_task_ui" model="ir.ui.view">
    <field name="name">view_form_todo_task_ui</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <form>
            <header><!-- Buttons and status widget --></header>
            <sheet><!-- Form content --></sheet>
            <!-- History and communication: -->
            <div class="oe_chatter">
                <field name="message_follower_ids" widget="mail_followers" />
                <field name="message_ids" widget="mail_thread" />
        </div>
        </form>
    </field>
</record>
```

Las vistas de negocio se componen de tres área visuales:

- Un encabezado, `header`.
- Un `sheet` para el contenido.
- Una sección al final de historia y comunicación, “history and communication”.

La sección historia y comunicación, con los widgets de red social en la parte inferior, es agregada por la herencia de su modelo de `mail.thread` (del módulo `mail`), y agrega los elementos del ejemplo XML mencionado anteriormente al final de la vista de formulario. También vio esto en el Capítulo 3.

La barra de estado del encabezado

La barra de estado en la parte superior usualmente presenta el flujo de negocio y los botones de acción.

Los botones de acción son botones regulares de formulario, y lo más común es que el siguiente paso sea resaltarlos, usando `class="oe_highlight"`. En el archivo `todo_ui/todo_view.xml` podrá ampliar el encabezado vacío para agregar le una barra de estado:

```
<header>
    <field name="stage_state" invisible="True" />
    <button name="do_toggle_done" type="object"
            attrs="{'invisible' [('stage_state','in',['done','cancel'])]}"
            string="Toggle Done" class="oe_highlight" />
    <!-- Add stage statusbar:   ... -->
</header>
```

Los botones de acción disponible puede diferir dependiendo en que parte del proceso se encuentre el documento actual. Por ejemplo, un botón Marcar como Hecho no tiene sentido si ya está en el estado “Hecho”.

Esto se realiza usando el atributo `states`, que lista los estados donde el botón debería estas visible, como esto: `states="draft,open"`.

Para mayor flexibilidad podrá usar el atributo `attrs`, el cual forma condiciones donde el botón debería ser invisible: `attrs="{'invisible' [('stage_state','in', ['done','cancel'])]`.

Estas características de visibilidad también están disponibles para otros elementos de la vista, y no solo para los botones. Verá esto en detalle más adelante en este capítulo.

El flujo de negocio

El flujo de negocio es un widget de barra de estado que se encuentra en un campo el cual representa el punto en el flujo donde se encuentra el registro. Usualmente es un campo de selección "State", o un campo "Stage" muchos a uno. En ambos casos puede encontrarse en muchos módulos de Odoo.

El "Stage" es un campo muchos a uno que se usa en un modelo donde los pasos del proceso están definidos. Debido a esto pueden ser fácilmente configurados por el usuario final para adecuarlo a sus procesos específicos de negocio, y son perfectos para el uso de pizarras `kanban`.

El "State" es una lista de selección que muestra los pasos estables y principales de un proceso, como Nuevo, En Progreso, o Hecho. No pueden ser configurados por el usuario final, pero son fáciles de usar en la lógica de negocio. Los "States" también tienen soporte especial para las vistas: el atributo `state` permite que un elemento este habilitado para ser seleccionado por el usuario dependiendo en el estado en que se encuentre el registro.

Truco: Es posible obtener un beneficio de ambos mundos, a través del uso de `stages` que son mapeados dentro de los "states". Esto fue lo que hizo en el capítulo anterior, haciendo disponible a "State" en los documentos de tareas por hacer a través de un campo calculado.

Para agregar un flujo de "stage" en su encabezado de formulario:

```
<!-- Add stage statusbar: ... -->
<field name="stage_id" widget="statusbar" clickable="True"
       options="{'fold_field': 'fold'}" />
```

El atributo `clickable` permite hacer clic en el widget, para cambiar la etapa o el estado del documento. Es posible que no querrá esto si el progreso del proceso debe realizarse a través de botones de acción.

En el atributo `options` podrá usar algunas configuraciones específicas:

- `fold_fields`, cuando de usa el atributo `stages`, es el nombre del campo que usa el atributo `stage` del modelo usa para indicar en cuales etapas debe ser mostrado en **negritas** o "**fold**".
- `statusbar_visible`, cuando se usa el atributo `states`, lista los estados que deben estar siempre visibles, para mantener ocultos los estados de excepción que se usan para casos menos comunes. Por ejemplo: `statusbar_visible="draft,open.done"`.

La hoja `canvas` es el área del formulario que contiene los elementos principales del formulario. Esta diseñada para parecer un documento de papel, y sus registros de datos, a veces, puede ser referidos como documentos.

La estructura general del documento tiene estos componentes:

- Información de título y subtítulo.
- Un área de botón inteligente, es la parte superior derecha de los campos del encabezado del documento.
- Un cuaderno con páginas en etiquetas, con líneas de documento y otros detalles.

Título y subtítulo

Cuando se usa el diseño de hoja, los campos que están fuera del bloque `<group>` no se mostrarán las etiquetas automáticamente. Es responsabilidad de la persona que desarrolla controlar si se muestran las etiquetas y cuando.

También se puede usar las etiquetas HTML para hacer que el título resplandezca. Para mejores resultados, el título del documento debe estar dentro de un elemento HTML `div` con la clase `oe_title`:

```
<div class="oe_title">
    <label for="name" class="oe_edit_only"/>
    <h1><field name="name"/></h1>
    <h3>
        <span class="oe_read_only">By</span>
        <label for="user_id" class="oe_edit_only"/>
        <field name="user_id" class="oe_inline" />
    </h3>
</div>
```

Aquí podrá ver el uso de elementos comunes de HTML como `div`, `span`, `h1` y `h3`.

Etiquetas y campos

Las etiquetas de los campos no son mostradas fuera de las secciones `<group>`, pero podrá mostrarlas usando el elemento `<label>`:

- El atributo `for` identifica el campo desde el cual tomará el texto de la etiqueta.
- El atributo `string` sobre escribe el texto original de la etiqueta del campo.
- Con el atributo `class` también podrá usar las clases CSS para controlar la presentación. Algunas clases útiles son:
- `oe_edit_only` para mostrar lo solo cuando el formulario este modo de edición.
- `oe_read_only` para mostrar lo solo cuando el formulario este en modo de lectura.

Un ejemplo interesante es reemplazar el texto con un ícono:

```
<label for="name" string=" " class="fafa-wrench"/>
```

Odoo empaqueta los íconos "Font Awesome", que se usan aquí. Los íconos disponibles puede encontrar se en http://fontawesome.org.

Botones inteligentes

El área superior izquierda puede tener una caja invisibles para colocar botones inteligentes. Estos funcionan como los botones regulares pero pueden incluir información estadística. Como ejemplo agregará un botón para mostrar el número total de tareas realizadas por el dueño de la tarea por hacer actual.

Primero necesita agregar el campo calculado correspondiente a `todo_ui/todo_model.py`. Agregue lo siguiente a la clase `TodoTask`:

```
@api.one
def compute_user_todo_count(self):
    self.user_todo_count = self.search_count([('user_id', '=', self.user_id.id)])
    user_todo_count      = fields.Integer('User To-Do   Count', compute='compute_user_todo_count'
```

Ahora agregará la caja del botón con un botón dentro de ella. Agregue lo siguiente justo después del bloque `div oe_title`:

```
<div name="buttons" class="oe_right oe_button_box">
    <button class="oe_stat_button" type="action" icon="fa-tasks"
            name="%(todo_app.action_todo_task)d" string=""
            context="{'search_default_user_id': user_id, 'default_user_id': user_id}"
            help="Other to-dos for this user">
        <field string="To-dos" name="user_todo_count" widget="statinfo"/>
    </button>
</div>
```

El contenedor para los botones es un elemento HTML `div` con las clases `oe_button_box` y `oe_right`, para que este alineado con la parte derecha del formulario.

En el ejemplo el botón muestra el número total de las tareas por hacer que posee el documento responsable. Al hacer clic en el, este las inspeccionara, y si se esta creando tareas nuevas el documento responsable original será usado como predeterminado.

Los atributos usados para el botón son:

- `class="oe_stat_button"`, es para usar un estilo rectángulo en vez de un botón.
- `icon`, es el ícono que será usado, escogido desde el conjunto de íconos de *Font Awesome*.
- `type`, será usualmente una acción para la acción de ventana, y `name` será el ID de la acción que será ejecutada. Puede usarse la formula `%(id-acción-externa)d`, para transformar el ID externo en un número de ID real. Se espera que esta acción abra una vista con los registros relacionados.
- `string`, puede ser usado para agregar texto al botón. No se usa aquí porque el campo que lo contiene ya proporciona un texto.
- `context`, fija las condiciones estándar en la vista destino, cuando se haga clic a través del botón, para los filtros de datos y los valores predeterminados para los registros creados.
- `help`, es la herramienta de ayuda que será mostrada.

Por si solo el botón es un contenedor y puede tener sus campos dentro para mostrar estadísticas. Estos son campos regulares que usan el widget `statinfo`.

El campo debe ser un campo calculado, definido en el módulo subyacente. También podrá usar texto estático en vez de o junto a los campos de `statinfo`, como: `<div>User's To-dos</div>`

2.6.3 Organizar el contenido en formulario

El contenido principal del formulario debe ser organizado usando etiquetas `<group>`. Un grupo es una cuadrícula con dos columnas. Un campo y su etiqueta ocupan dos columnas, por lo tanto al agregar campos dentro de un grupo, estos serán apilados verticalmente.

Si anido dos elementos `<group>` dentro de un grupo superior, tendrá dos columnas de campos con etiquetas, una al lado de la otra.

```
<group name="group_top">
    <group name="group_left">
        <field name="date_deadline" />
        <separator string="Reference"/>
        <field name="refers_to"/>
    </group>
    <group name="group_right">
        <field name="tag_ids" widget="many2many_tags"/>
    </group>
</group>
```

Los grupos pueden tener un atributo `string`, usado para el título de la sección. Dentro de una sección de grupo, los títulos también pueden agregarse usando un elemento `separator`.

Truco: Intente usar la opción Alternar la Disposición del Esquema del Formulario del menú de Desarrollo: este dibuja líneas alrededor de cada sección del formulario, permitiendo un mejor entendimiento de como esta organizada la vista actual.

Cuaderno con pestañas

Otra forma de organizar el contenido es el cuaderno, el cual contiene múltiples secciones a través de pestañas llamadas páginas. Esto puede usarse para mantener algunos datos fuera de la vista hasta que sean necesarios u

organizar un largo número de campos por tema.

No necesitará esto en su formulario de tareas por hacer, pero el siguiente es un ejemplo que podría agregar en el formularios de etapas de la tarea:

```
<notebook>
    <page string="Whiteboard" name="whiteboard">
        <field name="docs"/>
    </page>
    <page name="second_page">
        <!-- Second page content -->
    </page>
</notebook>
```

Se considera una buena practica tener nombres en las páginas, esto hace que la ampliación de estas por parte de otros módulo sea más fiable

Elementos de la vista

Ha visto como organizar el contenido dentro de un formulario, usando elementos como encabezado, grupo y cuaderno. Ahora, podrá ahondar en los elementos de campo y botón y que podrá hacer con ellos.

Botones

Los botones soportar los siguientes atributos:

- `icon`. A diferencia de los botones inteligentes, los íconos disponibles para los botones regulares son aquellos que se encuentran en `addons/web/static/src/img/icons`.
- `string`, es el texto de descripción del botón.
- `type`, puede ser `workflow`, `object` o `action`, para activar una señal de flujo de trabajo, llamar a un método Python o ejecutar una acción de ventana.
- `name`, es el desencadenante de un flujo de trabajo, un método del modelo, o la ejecución de una acción de ventana, dependiendo del `type` del botón.
- `args`, se usa para pasar parámetros adicionales al método, si el `type` es `object`.
- `context`, fija los valores en el contexto de la sesión, el cual puede tener efecto luego de la ejecución de la acción de ventana, o al llamar a un método de Python. En el último caso, a veces puede ser usado como un alternativa a `args`.
- `confirm`, agrega un mensaje con el mensaje de texto preguntando por una confirmación.
- `special="cancel"`, se usa en los asistentes, para cancelar o cerrar el formulario. No debe ser usado con `type`.

Campos

Los campos tiene los siguientes atributos disponibles. La mayoría es tomado de los que fue definido en el modelo, pero pueden ser sobre escritos en la vista. Los atributos generales son:

- `name`: identifica el nombre técnico del campo.
- `string`: proporciona la descripción de texto de la etiqueta para sobre escribir aquella provista por el modelo.
- `help`: texto de ayuda a ser usado y que reemplaza el proporcionado por el modelo.
- `placeholder`: proporciona un texto de sugerencia que será mostrado dentro del campo.
- `widget`: sobre escribe el widget predeterminado usado por el tipo de campo. Explorará los widgets disponibles más adelante en este mismo capítulo.

- `options`: contiene opciones adicionales para ser usadas por el widget.
- `class`: proporciona las clases CSS usadas por el HTML del campo.
- `invisible="1"`: invisibiliza el campo.
- `nolabel="1"`: no muestra la etiqueta del campo, solo es significativo para los campos que se encuentran dentro de un elemento `<group>`.
- `readonly="1"`: no permite que el campo sea editado.
- `required="1"`: hace que el campo sea obligatorio.

Atributos específicos para los tipos de campos:

- `sum`, `avg`: para los campos numéricos, y en las vistas de lista/árbol, estos agregan un resumen al final con el total o el promedio de los valores.
- `password="True"`: para los campos de texto, muestran el campo como una campo de contraseña.
- `filename`: para campos binarios, es el campo para el nombre del archivo.
- `mode="tree"`: para campos `One2many`, es el tipo de vista usado para mostrar los registros. De forma predeterminada es de árbol, pero también puede ser de formulario `form`, `kanban` o gráfico.

Para los atributos *Booleanos* en general, podrá usar `True` o `1` para habilitarlo y `False` o `0` *(cero)* para deshabilitarlo. Por ejemplo, `readonly="1"` y `realonly="True"` son equivalentes.

Campos relacionales

En los campos relacionales, podrá tener controles adicionales referentes a los que el usuario puede hacer. De forma predeterminada el usuario pueden crear nuevos registros desde estos campos (también conocido como creación rápida) y abrir el formulario relacionado al registro. Esto puede ser deshabilitado usando el atributo del campo `options`:

```
options={'no_open': True, 'no_create': True}
```

El contexto y el dominio también son particulares en los campos relacionales. El contexto puede definir valores predeterminados para los registros relacionados, y el dominio puede limitar los registros que pueden ser seleccionados, por ejemplo, basado en otro campo del registro actual. Tanto el contexto como el dominio pueden ser definidos en el modelo, pero solo son usados en la vista.

Widgets de campo

Cada tipo de campo es mostrado en el formulario con el widget predeterminado apropiado. Pero otros widget adicionales están disponible y pueden ser usados:

Widgets para los campos de texto:

- `email`: convierte al texto del correo electrónico en un elemento “mail-to” ejecutable.
- `url`: convierte al texto en un URL al que se puede hacer clic.
- `html`: espera un contenido en HTML y lo representa; en modo de edición usa un editor WYSIWYG para dar formato al contenido sin saber HTML.

Widgets para campos numéricos:

- `handle`: específicamente diseñado para campos de secuencia, este muestra una guía para dibujar líneas en una vista de lista y re ordenarlos manualmente.
- `float_time`: da formato a un valor decimal como tiempo en horas y minutos.
- `monetary`: muestra un campo decimal como un monto en monedas. La moneda a usar puede ser tomada desde un campo como `options="{'currency_field': 'currency_id'}"`.

- `progressbar`: presenta un decimal como una barra de progreso en porcentaje, usualmente se usa en un campo calculado que computa una tasa de culminación.

Algunos widget para los campos relacionales y de selección:

- `many2many_tags`: muestran un campo muchos a muchos como una lista de etiquetas.
- `selection`: usa el widget del campo Selección para un campo mucho a uno.
- `radio`: permite seleccionar un valor para una opción del campo de selección usando botones de selección simple.
- `kanban_state_selection`: muestra una luz de semáforo para la lista de selección de esta vista `kanban`.
- `priority`: representa una selección como una lista de estrellas a las que se puede hacer clic.

Eventos on-change

A veces necesita que el valor de un campo sea calculado automáticamente cuando cambia otro campo. El mecanismo para esto se llama `on-change`.

Desde la versión o, los eventos `on-change` están definidos en la capa del modelo, sin necesidad de ningún marcado especial en las vistas. Es se hace creando los métodos para realizar el calculo y enlazándolos al campo(s) que desencadenara la acción, usando el decorador `@api.onchenge('field1','field2')`.

En las versiones anteriores, ente enlace era hecho en la capa de vista, usando el atributo `onchange` para fijar el método de la clase que sería llamado cuando el campo cambiara. Esto todavía es soportado, pero es obsoleto. Tenga en cuenta que los métodos `on-change` con el estilo viejo no pueden ser ampliados usando la API nueva. Si necesita hacer esto, deberá usar la API vieja.

2.6.4 Vistas dinámicas

Los elementos visibles como un formulario también pueden ser cambiados dinámicamente, dependiendo, por ejemplo de los permisos de usuario o la etapa del proceso en la cual esta el documento.

Estos dos atributos le permiten controlar la visibilidad de los elemento en la interfaz:

- `groups`: hacen al elemento visible solo para los miembros de los grupos de seguridad específicos. Se espera una lista separada por coma de los ID XML del grupo.
- `states`: hace al elemento visible solo cuando el documento esta en el estado especificado. Espera una lista separada por coma de los códigos de "State", y el modelo del documento debe tener un campo "state".

Para mayor flexibilidad, podrá fijar la visibilidad de un elemento usando expresiones evaluadas del lado del cliente. Esto puede hacerse usando el atributo `attrs` con un diccionario que mapea el atributo `invisible` al resultado de una expresión de dominio.

Por ejemplo, para hacer que el campo `refers_to` sea visible en todos los estados menos `draft`:

```
<field name="refers_to" attrs="{'invisible': [('state','=','draft')]}" />
```

El atributo `invisible` esta disponible para cualquier elemento, no solo para los campos. Podrá usarlo en las páginas de un cuaderno o en grupos, por ejemplo.

El atributo `attrs` también puede fijar valores para otros dos atributos: `readonly` y `required`, pero esto solo tiene sentido para los campos de datos, convirtiéndolos en campos que no pueden ser editados u obligatorios. Con esto podrá agregar alguna lógica de negocio haciendo a un campo obligatorio, dependiendo del valor de otro campo, o desde un cierto estado más adelante.

Vistas de lista

Comparadas con las vistas de formulario, las vistas de listas son mucho más simples. Una vista de lista puede contener campos y botones, y muchos de los atributos de los formularios también están disponibles.

Aquí se muestra un ejemplo de una vista de lista para su Tareas por Hacer:

```
<record id="todo_app.view_tree_todo_task"   model="ir.ui.view">
    <field name="name">To-do Task Tree</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <tree editable="bottom" colors="gray:is_done==True" fonts="italic: state!='open'" delete=
            <field name="name"/>
            <field name="user_id"/>
        </tree>
    </field>
</record>
```

Los atributos para el elemento `tree` de nivel superior son:

- `editable`: permite que los registros sean editados directamente en la vista de lista. Los valores posibles son `top` y `bottom`, los lugares en donde serán agregados los registros nuevos.
- `colors`: fija dinámicamente el color del texto para los registros, basándose en su contenido. Es una lista separada por punto y coma de valores `color:condition`. `color` es un color válido CSS (vea http://www.w3.org/TR/css3-color/#html4), y `condition` es una expresión Python que evalúa el contexto del registro actual.
- `fonts`: modifica dinámicamente el tipo de letra para los registro basándose en su contexto. Es similar al atributo `colors`, pero este fija el estilo de la letra a `bold`, `italic` o `underline`.
- `create`, `delete`, `edit`: si se fija a `false` (en minúscula), deshabilita la acción correspondiente en la vista de lista.

Vistas de búsqueda

Las opciones de búsqueda disponibles en las vistas son definidas a través de una vista de lista. Esta define los campos que serán buscados cuando se escriba en la caja de búsqueda. También provee filtros predefinidos que pueden ser activados con un clic, y opciones de agrupación de datos para los registros en las vistas de lista o `kanban`.

Aquí se muestra una vista de búsqueda para las tareas por hacer:

```
<record id="todo_app.view_filter_todo_task" model="ir.ui.view">
    <field name="name">To-do Task Filter</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <search>
            <field name="name" domain_filter="['|', ('name','ilike',self),('user_id','ilike',self
            <field name="user_id"/>
            <filter name="filter_not_done" string="Not Done" domain="[('is_done','=',False)]"/>
            <filter name="filter_done" string="Done" domain="[('is_done','!=',False)]"/>
            <separator/>
            <filter name="group_user" string="By User" context="{'group_by':'user_id'}"/>
        </search>
    </field>
</record>
```

Podrá ver dos campos que serán buscados: `name` y `user_id`. En `user_id` tendrá una regla de filtro que hace la "búsqueda si" tanto en la descripción como en el usuario responsable. Luego tendrá dos filtros predefinidos, filtrando las "tareas no culminadas" y "tareas culminadas". Estos filtros pueden ser activados de forma independiente, y serán unidos por un operador `OR` si ambos son habilitados. Los bloques de `filters` separados por un elemento `<separator/>` serán unidos por un operador `AND`.

El tercer filtro solo fija un contexto o "group-by". Esto le dice a la vista que agrupe los registros por ese campo, `user_id` en este caso.

Los elementos `filed` pueden usar los siguientes atributos:

- `name`: identifica el campo.
- `string`: proporciona el texto de la etiqueta que será usado, en vez del predeterminado.
- `operator`: le permite usar un operador diferente en vez del predeterminado - = para campos numéricos y `ilike` para otros tipos de campos.
- `filter_domain`: puede usarse para definir una expresión de dominio específica para usar en la búsqueda, proporcionando mayor flexibilidad que el atributo `operator`. El texto que será buscado se referencia en la expresión usando `self`.
- `groups`: permite hacer que la búsqueda en el campo solo este disponible para una lista de grupos de seguridad (identificado por los Ids XML)

Estos son los atributos disponibles para los elementos `filter`:

- `name`: en un identificador, usado para la herencia o para habilitar la a través de la clave `search_default_` en el contexto de acciones de ventana.
- `string`: proporciona el texto de la etiqueta que se mostrará para el filtro (obligatorio)
- `domain`: proporciona la expresión de dominio del filtro para ser añadida al dominio activo.
- `context`: es un diccionario de contexto para agregarlo al contexto actual. Usualmente este fija una clave `group_by` con el nombre del filtro que agrupara los registros.
- `groups`: permite hacer que el filtro de búsqueda solo este disponible para una lista de grupos.

2.6.5 Otros tipos de vista

Los tipos de vista que se usan con mayor frecuencia son los formularios y las listas, discutidos hasta ahora. A parte de estas, existen otros tipos de vista, y dará un vistazo a cada una de ellas. Las vistas `kanban` no serán discutidas aquí, ya que las verá en el Capítulo 8.

Recuerde que los tipos de vista disponibles están definidos en el atributo `view_mode` de la acción de ventana correspondiente.

Vistas de Calendario

Como su nombre lo indica, esta presenta los registros en un calendario. Una vista de calendario para las tareas por hacer puede ser de la siguiente manera:

```
<record id="view_calendar_todo_task" model="ir.ui.view">
    <field name="name">view_calendar_todo_task</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <calendar date_start="date_deadline" color="user_id" display="[name], Stage[stage_id]">
            <!-- Fields used for the text of display attribute -->
            <field name="name" />
            <field name="stage_id" />
        </calendar>
    </field>
</record>
```

Los atributos de `calendar` son los siguientes:

- `date_start`: El campo para la fecha de inicio (obligatorio).
- `date_end`: El campo para la fecha de culminación (opcional).
- `date_delay`: El campo para la duración en días. Este puede ser usado en vez de `date_end`.

- `color`: El campo para colorear las entradas del calendario. Se le asignará un color a cada valor en el calendario, y todas sus entradas tendrán el mismo color.
- `display`: Este es el texto que se mostrará en las entradas del calendario. Los campos pueden ser insertados usando `[<field>]`. Estos campos deben ser declarados dentro del elemento `calendar`.

Vistas de Gantt

Esta vista presenta los datos en un gráfico de Gantt, que es útil para la planificación. Las tareas por hacer solo tiene un campo de fecha para la fecha de límite, pero podrá usarla para tener una vista funcional de un gráfico Gantt básico:

```
<record id="view_gantt_todo_task" model="ir.ui.view">
    <field name="name">view_gantt_todo_task</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <gantt date_start="date_deadline" default_group_by="user_id" />
    </field>
</record>
```

Los atributos que puede ser usados para las vistas Gantt son los siguientes.

- `date_start`: El campo para la fecha de inicio (obligatorio).
- `date_stop`: El campo para la fecha de culminación. Puede ser reemplazado por `date_delay`.
- `date_delay`: El campo con la duración en días. Puede usarse en vez de `date_stop`.
- `progress`: Este campo proporciona el progreso en porcentaje (entre 0 y 100).
- `default_group_by`: Este campo se usa para agrupar las tareas Gantt.

Vistas de Gráfico

Los tipos de vista de gráfico proporcionan un análisis de los datos, en forma de gráfico o una tabla pivote interactiva.

Agregará una tabla pivote a las tareas por hacer. Primero, necesita agregar un campo. En la clase `TodoTask`, del archivo `todo_ui/todo_model.py`, agregue este línea:

```
effort_estimate = fields.Integer('Effort Estimate')
```

También debe ser agregado al formulario de tareas por hacer para que podrá fijar datos allí. Ahora, agregue la vista de gráfico con una tabla pivote:

```
<record id="view_graph_todo_task" model="ir.ui.view">
    <field name="name">view_graph_todo_task</field>
    <field name="model">todo.task</field>
    <field name="arch" type="xml">
        <graph type="pivot">
            <field name="stage" type="col" />
            <field name="user_id"   />
            <field name="date_deadline" interval="week" />
            <field name="effort_estimate" type="measure" />
        </graph>
    </field>
</record>
```

El elemento `graph` tiene el atributo `type` fijado a `pivot`. También puede ser `bar` (predeterminado), `pie` o `line`. En el caso que sea "bar", gráfico de barras, adicionalmente se puede usar `stacked="True"` para hacer un gráfico de barras apilado.

`graph` debería contener campos que pueden tener estos posibles atributos:

- `name`: Identifica el campo que será usado en el gráfico, así como en otras vistas.

- `type`: Describe como será usado el campo, como un grupo de filas (predeterminado), “row”, como un grupo de columnas, “col”, o como una medida, “mesure”.
- `interval`: Solo es significativo para los campos de fecha, es un intervalo de tiempo para agrupar datos de fecha por `day`, `week`, `month`, `quarter` o `year`.

2.6.6 Resumen

Aprendió más sobre las vistas e Odoo que son usadas para la construcción de la interfaz. Comenzó agregando opciones de menú y acciones de ventana usadas para abrir las vistas. Fueron explicados en detalle los conceptos de contexto y dominio.

También aprendió como diseñar vistas de lista y configurar opciones de búsqueda usando las vistas de búsqueda. Luego, se describieron de modo general los otros tipos de vista disponibles: calendario, Gantt y gráfico. Las vistas Kanban serán estudiadas más adelante, cuando aprenda como usar Qweb.

Ya ha vistos los modelos y las vistas. En el próximo capítulo, aprenderá como implementar la lógica de negocio del lado del servidor.

2.7 Capítulo 7 - Lógica ORM

2.7.1 Lógica de la Aplicación ORM – Apoyo a los Procesos de Negocio

En este capítulo, aprenderá como escribir código para soportar la lógica de negocio en sus modelos y también como puede esto ser activado en eventos y acciones de usuario. Podrá escribir lógica compleja y asistentes usando la API de programación de Odoo, lo que les permitirá proveer una interacción más dinámica con el usuario con estos programas.

Asistente de tareas por hacer

Con los asistentes, podrá pedir a los usuarios que ingresen información para ser usada en algunos procesos. Suponga que los usuarios de su aplicación necesitan fijar fechas límites y personas responsables, regularmente, para un largo número de tareas. Podrá usar un asistente para ayudar con esto. El asistente permitirá escoger las tareas que serán actualizadas y luego seleccionar la fecha límite y/o la persona responsable.

Comenzara por crear un módulo nuevo para esta característica: `todo_wizard`. Nuestro módulo tendrá un archivo Python y un archivo XML, por lo tanto la descripción `todo_wizard/__openerp__.py` será como se muestra en el siguiente código:

```
{
    'name': 'To-do Tasks Management Assistant',
    'description': 'Mass edit your To-Do backlog.',
    'author': 'Daniel Reis',
    'depends': ['todo_user'],
    'data': ['todo_wizard_view.xml'],
}
```

El código para cargar su código en el archivo `todo_wizard/__init__.py`, es solo una línea:

```
from . import todo_wizard_model
```

Luego, necesita describir el modelo de datos que soporta su asistente.

Modelo del asistente

Un asistente muestra una vista de formulario al usuario, usualmente dentro de una ventana de dialogo, con algunos campos para ser llenados. Esto será usado luego por la lógica del asistente.

Esto es implementado usando la arquitectura modelo/vista usada para las vistas regulares, con una diferencia: El modelo soportado esta basado en `models.TransientModel` en vez de `models.Model`.

Este tipo de modelo también es almacenado en la base de datos, pero se espera que los datos sean útiles solo hasta que el asistente sea completado o cancelado. El servidor realiza limpiezas regulares de los datos viejos en los asistentes desde las tablas correspondientes de la base de datos.

El archivo `todo_wizard/todo_wizard_model.py` definirá los tres campos que necesita: la lista de tareas que serán actualizadas, la persona responsable, y la fecha límite, como se muestra aquí:

```
# -*- coding: utf-8 -*-
from openerp import models, fields, api
from openerp import exceptions # will be used in the code
import logging_logger = logging.getLogger(__name__)

class TodoWizard(models.TransientModel):
    _name        = 'todo.wizard'
    task_ids     = fields.Many2many('todo.task', string='Tasks')
    new_deadline = fields.Date('Deadline to Set')
    new_user_id  = fields.Many2one('res.users',string='Responsible to Set')
```

No vale de que nada, si usa una relación uno a muchos tener que agregar el campo inverso muchos a uno. Debería evitar las relaciones muchos a uno entre los modelos transitorios y regulares, y para ello use relaciones muchos a muchos que tengan el mismo propósito sin la necesidad de modificar el modelo de tareas por hacer.

También esta agregando soporte al registro de mensajes. El registro se inicia con dos líneas justo después del TodoWizard, usando la librería estándar de registro de Python. Para escribir mensajes en el registro podrá usar:

```
_logger.debug('A DEBUG message')
_logger.info('An INFO message')
_logger.warning('A WARNING message')
_logger.error('An ERROR message')
```

Vea más ejemplos de su uso en este capítulo.

Formularios de asistente

La vista de formularios de asistente luce exactamente como los formularios regulares, excepto por dos elementos específicos:

- Puede usarse una sección `<footer>` para colocar botones de acción.
- Esta disponible un tipo especial de botón de cancelación para interrumpir el asistente sin ejecutar ninguna acción.

Este es el contenido del archivo `todo_wizard/todo_wizard_view.xml`:

```
<openerp>
    <data>
        <record id="To-do Task Wizard" model="ir.ui.view">
            <field name="name">To-do Task Wizard</field>
            <field name="model">todo.wizard</field>
            <field name="arch" type="xml">
                <form>
                    <div class="oe_right">
                        <button type="object" name="do_count_tasks" string="Count"/>
                        <button type="object" name="do_populate_tasks" string="Get All"/>
                    </div>
                 <field name="task_ids"/>
                 <group>
                     <group>
                         <field name="new_user_id"/>
                     </group>
                     <group>
```

```
                        <field name="new_deadline"/>
                    </group>
                </group>
                <footer>
                    <button type="object" name="do_mass_update" string="Mass Update"
                            class="oe_highlight"
                            attrs="{'invisible': [('new_deadline','=',False), ('new_user_id','=
                    <button special="cancel" string="Cancel"/>
                </footer>
            </form>
        </field>
    </record>
    <!-- More button Action -->
    <act_window id="todo_app.action_todo_wizard" name="To-Do Tasks Wizard"
                src_model="todo.task" res_model="todo.wizard" view_mode="form"
                target="new" multi="True"/>
  </data>
</openerp>
```

La acción de ventana que ve en el XML agrega una opción al botón "Más" del formulario de tareas por hacer, usando el atributo `src_model`. `target=new` hace que se abra como una ventana de dialogo.

También debe haber notado el atributo `attrs` en el botón "Mass Update" usado para hacer al botón invisible hasta que sea seleccionada otra fecha límite u otro responsable.

Así es como lucirá su asistente:

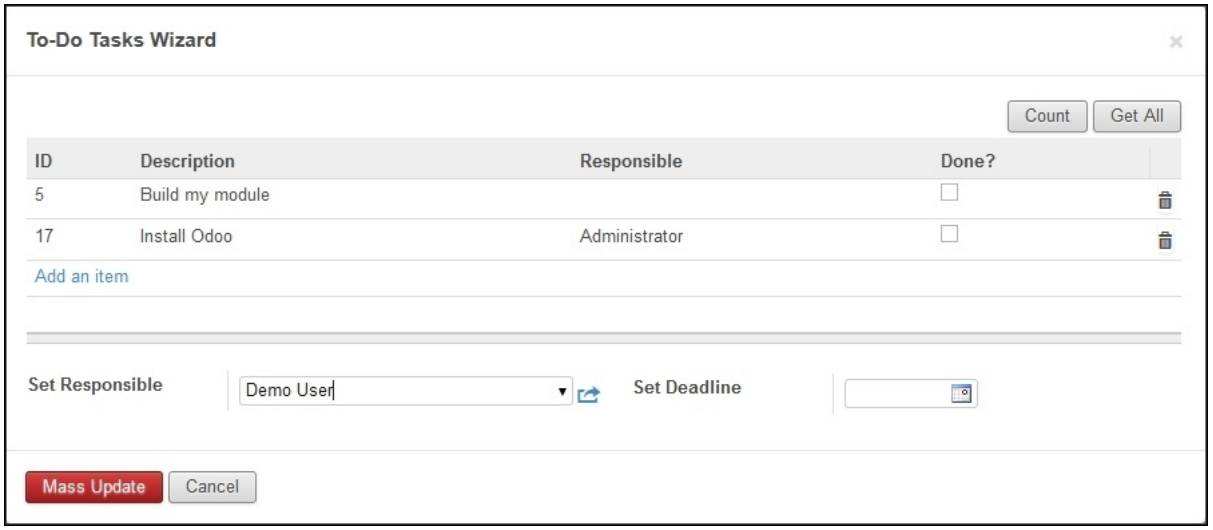

Figura 2.18: Gráfico 7.1 - Vista ToDo Tasks Wizard

Lógica de negocio del asistente

Luego necesita implementar las acciones ejecutadas al hacer clic en el botón "Mass Update". El método que es llamado por el botón es `do_mass_update` y debe ser definido en el archivo `todo_wizard/todo_wizard_model.py`, como se muestra en el siguiente código.

```
@api.multi
def do_mass_update(self):
    self.ensure_one()
    if not (self.new_deadline   or self.new_user_id):
        raise  exceptions.ValidationError('No data to update!') #
    else:
        _logger.debug('Mass update on Todo Tasks %s',self.task_ids.ids)
```

```
        if self.new_deadline:
            self.task_ids.write({'date_deadline': self.new_deadline})
            if self.new_user_id:
                self.task_ids.write({'user_id': self.new_user_id.id})
                return True
```

Nuestro código puede manejar solo una instancia del asistente al mismo tiempo. Puede que haya usado `@api.one`, pero no es recomendable hacerlo en los asistentes. En algunos casos querrá que el asistente devuelva una acción de ventana, que le diga al cliente que hacer luego. Esto no es posible hacerlo con `@api.one`, ya que esto devolverá una lista de acciones en vez de una sola.

Debido a esto, prefiere usar `@api.multi` y luego use `ensure_one()` para verificar que `self` representa un único registro. Debe tenerse en cuenta que `self` es un registro que representa los datos en el formulario del asistente. El método comienza validando si se ha dado una nueva fecha límite o un nuevo responsable, de lo contrario arroja un error. Luego, se hace una demostración de la escritura de un mensaje en el registro del servidor. Si pasa la validación, escriba los nuevos valores dados a las tareas seleccionadas. Esta usando el método de escritura en un conjunto de registros, como los `task_id` a muchos campos para ejecutar una actualización masiva.

Esto es más eficiente que escribir repetidamente en cada registro dentro de un bucle. Ahora trabajara en la lógica detrás de los dos botones en la parte superior. “Count” y “Get All”.

Elevar excepciones

Cuando algo no esta bien, querrá interrumpir el programa con algún mensaje de error. Esto se realiza elevando una excepción. Odoo proporciona algunas clases de excepción adicionales a aquellas disponibles en Python. Estos son ejemplos de las más usadas:

```
from openerp import exceptions

raise exceptions.Warning('Warning message')
raise exceptions.ValidationError('Not valid message')
```

El mensaje de advertencia también interrumpe la ejecución pero puede parecer menos severo que un ValidationError. Aunque no es la mejor interfaz, les aprovechará de esto para mostrar un mensaje en el botón “Count”:

```
@api.multi def do_count_tasks(self):
    Task  = self.env['todo.task']
    count = Task.search_count([])

    raise exceptions.Warning('There are %d active tasks.' % count)
```

Recarga automática de los cambios en el código

Cuando esta trabajando en el código Python, es necesario reiniciar el servidor cada vez que el código cambia. Para hacer le la vida más fácil a las personas que desarrollan esta disponible la opción `--auto-reload`. Esta realiza un monitoreo del código fuente y lo recarga automáticamente si es detectado algún cambio. Aquí se muestra un ejemplo de su uso:

```
$ ./odoo.py -d v8dev --auto-reload
```

Pero esta es una característica única en sistemas Linux. Si esta usando Debian/Ubuntu, como se recomendó en el Capítulo 1, entonces debe funcionar. Se requiere el paquete Python pyinotify, y debe ser instalado a través de `apt-get` o `pip`, como se muestra a continuación:

Usando paquetes OS, ejecutando el siguiente comando:

```
$ sudo apt-get install python-pyinotify
```

Usando `pip`, posiblemente en un entorno virtual creado por el paquete `virtualenv`, ejecutando el siguiente comando:

```
$ pip install pyinotify
```

Acciones en el dialogo del asistente

Ahora suponga que querrá tener un botón que selecciona automáticamente las todas las tareas por hacer para ahorrar le la tarea al usuario de tener que escoger una a una. Este es el objetivo de tener un botón "Get All" en el formulario. El código detrás de este botón tomará un conjunto de registros de tareas activas y los asignará a las tareas en el campo muchos a muchos.

Pero hay una trampa aquí. En las ventanas de dialogo, cuando un botón es presionado, la ventana de asistente es cerrada automáticamente. No se les presento este problema con el botón "Count" porque este usa una excepción para mostrar el mensaje; así que la acción falla y la ventana no se cierra.

Afortunadamente podrá trabajar este comportamiento para que retorne una acción al cliente que abra de nuevo el mismo asistente. Los métodos del modelo pueden retornar una acción para que el cliente web la ejecute, de la forma de un diccionario que describa la acción de ventana que será ejecutada. Este diccionario usa los mismos atributos que se usan para definir las acciones de ventana en el XML del módulo.

Usara una función de ayuda para el diccionario de la acción de ventana para abrirse de nuevo la ventana del asistente, así podrá ser usada de nuevo en varios botones, como se muestra a continuación:

```
@api.multi
def do_reopen_form(self):
    self.ensure_one()
    return {
        'type': 'ir.actions.act_window',
        'res_model': self._name, # this model
        'res_id': self.id, # the current wizard record
        'view_type': 'form',
        'view_mode': 'form',
        'target': 'new'
    }
```

No es importante si la acción de ventana es cualquier otra cosa, como saltas a un formulario y registro específico, o abrir otro formulario de asistente para pedir al usuario el ingreso de más datos.

Ahora que el botón "Get All" puede realizar su trabajo y mantener al usuario trabajando en el mismo asistente:

```
@api.multi
def do_populate_tasks(self):
    self.ensure_one()
    Task = self.env['todo.task']
    all_tasks = Task.search([])
    self.task_ids = all_tasks # reopen wizard form on same wizard record
    return self.do_reopen_form()
```

Aquí podrá ver como obtener una referencia a un modelo diferente, el cual en este caso es `todo.task`, para ejecutar acciones en el. Los valores del formulario del asistente son almacenados en un modelo transitorio y pueden ser escritos y leídos como en los modelos regulares. También podrá ver que el método fija el valor de''task_ids'' con la lista de todas las tareas activas.

Note que como no hay garantía que `self` sea un único registro, lo valida usando `self.ensure_one()`. No debe usar el decorador `@api.one` porque envuelve el valor retornado en una lista. Debido a que el cliente web espera recibir un diccionario y no una lista, no funcionaría como es requerido.

Trabajar en el servidor

Usualmente su código del servidor se ejecuta dentro de un método del modelo, como es el caso de `do_mass_update()` en el código precedente. En este contexto, `self` representa el conjunto de registro desde los cuales se actúa.

Las instancias de las clases del modelo son en realidad un conjunto de registros. Para las acciones ejecutadas desde las vistas, este será únicamente el registro seleccionado actualmente. Si es una vista de formulario, usualmente es un único registro, pero en las vistas de árbol, pueden ser varios registros.

El objeto `self.env` le permite acceder a su entorno de ejecución; esto incluye la información de la sesión actual, como el usuario actual y el contexto de sesión, y también acceso a todos los otros modelos disponibles en el servidor.

Para explorar mejor la programación del lado del servidor, podrá usar la consola interactiva del servidor, donde tiene un entorno similar al que encontró dentro de un método del modelo.

Esta es una nueva característica de la versión 9. Ha sido portada como un módulo para la versión 8, y puede ser descargada en https://www.odoo.com/apps/modules/8.0/shell/. Solo necesita ser colocada en algún lugar en la ruta de sus add-ons, y no se requiere instalación, o puede usar los siguientes comandos para obtener el código desde GitHub y hacer que el módulo este disponibles es su directorio de add-ons personalizados:

```
$ cd ~/odoo-dev
$ git clone https://github.com/OCA/server-tools.git -b 8.0
$ ln -s server-tools/shell custom-addons/shell
$ cd ~/odoo-dev/odoo
```

Para usar esto, ejecute `odoo.py` desde la línea de comandos con la base de datos a usar, como se muestra a continuación:

```
$ ./odoo.py shell -d v8dev
```

Puede ver la secuencia de inicio del servidor en la terminal culminando con un el símbolo de entrada de Python `>>>`. Aquí, `self` representa el registro para el usuario administrador como se muestra a continuación:

```
>>> self res.users(1,)
>>> self.name u'Administrator'
>>> self._name 'res.users'
>>> self.env
<openerp.api.Environment object at 0xb3f4f52c>
```

En la sesión anterior, se hizo una breve inspección de su entorno. `self` representa al conjunto de registro `res.users` el cual solo contiene el registro con el ID 1 y el nombre Administrator. También podrá confirmar el nombre del modelo del conjunto de registros con `self._name`, y confirmar que `self.env` es una referencia para el entorno.

Como es usual, puede salir de la usando *Ctrl + D*. Esto también cerrará el proceso en el servidor y le llevara de vuelta a la línea de comandos de la terminal.

La clase `Model` a la cual hace referencia `self` es de hecho un conjunto de registros. Si se itera a través de un conjunto de registro se retornará registros individuales.

El caso especial de un conjunto de registro con un solo registro es llamado “singleton”. Los “singletons” se comportan como registros, y para cualquier propósito práctico con la misma cosa. Esta particularidad quiere decir que se puede usar un registro donde sea que se espere un conjunto de registros.

A diferencia de los conjuntos de registros multi elementos, los “singletons” pueden acceder a sus campos usando la notación de punto, como se muestra a continuación:

```
>>> print self.name Administrator
>>> for rec in self: print rec.name Administrator
```

En este ejemplo, se realiza un ciclo a través de los registros en el conjunto `self` e imprime el contenido del campo `name`. Este contiene solo un registro, por lo tanto solo se muestra un nombre. Como puede ver, `self` es un “singleton” y se comporta como un registro, pero al mismo tiempo es iterable como un conjunto de registros.

Usar campos de relación

Como ya ha visto, los modelos pueden tener campos relacionales: muchos a uno, uno a muchos, y muchos a muchos. Estos tipos de campos tienen conjuntos de registros como valores.

En en caso de muchos a uno, el valor puede ser un "singleton" o un conjunto de registros vacío. En ambos casos, podrá acceder a sus valores directamente. Como ejemplo, las siguientes instrucciones son correctas y seguras:

```
>>> self.company_id res.company(1,)
>>> self.company_id.name u'YourCompany'
>>> self.company_id.currency_id res.currency(1,)
>>> self.company_id.currency_id.name u'EUR'
```

Convenientemente un conjunto de registros vacío también se comporta como un singleton, y el acceder a sus campos no retorna un error simplemente un `False`. Debido a esto, podrá recorrer los registros usando la notación de punto sin preocuparse por los errores de valores vacíos, como se muestra a continuación:

```
>>> self.company_id.country_id res.country()
>>> self.company_id.country_id.name False
```

Consultar los modelos

Con `self` solo podrá acceder a al conjunto de registros del método. Pero la referencia a `self.env` le permite acceder a cualquier otro modelo.

Por ejemplo, `self.env['res.partner']` devuelve una referencia al modelo Partners (la cual es un conjunto de registros vacío). Por lo tanto podrá usar `search()` y `browse()` para generar el conjunto de registros.

El método `search()` toma una expresión de dominio y devuelve un conjunto de registros con los registros que coinciden con esas condiciones. Un dominio vacío `[]` devolverá todos los registros. Si el modelo tiene el campo especial "active", de forma predeterminada solo los registros que tengan `active=True` serán tomados en cuenta. Otros argumentos opcionales están disponibles:

- `order`: Es una cadena de caracteres usada en la clausula `ORDER BY` en la consulta a la base de datos. Usualmente es una lista de los nombres de campos separada por coma.
- `limit`: Fija el número máximo de registros que serán devueltos.
- `offset`: Ignora los primeros "n" resultados; puede usarse con `limit` para realizar la búsqueda de un bloque de registros a la vez.

A veces solo necesita saber el número de registros que cumplen con ciertas condiciones. Para esto podrá usar `search_count()`, la cual devuelve el conteo de los registros en vez del conjunto de registros.

El método `browse()` toma una lista de Ids o un único ID y devuelve un conjunto con esos registros. Esto puede ser conveniente para los casos en que ya sepa los Ids de los registros que desea.

Algunos ejemplos de su uso se muestran a continuación:

```
>>> self.env['res.partner'].search([('name','like','Ag')]) res.partner(7,51)
>>> self.env['res.partner'].browse([7,51]) res.partner(7,51)
```

Escribir en los registros

Los conjuntos de registros implementan el patrón de registro activo. Esto significa que podrá asignas les valores, y esos valores se harán permanentes en la base de datos. Esta es una forma intuitiva y conveniente de manipulación de datos, como se muestra a continuación:

```
>>> admin = self.env['res.users'].browse(1)
>>> admin.name = 'Superuser'
>>> print admin.name Superuser
```

Los conjuntos de registros tienes tres métodos para actuar sobre los datos: `create()`, `write()`, `unlink()`.

El método `create()` toma un diccionario para mapear los valores de los campos y devuelve el registro creado. Los valores predeterminados con aplicados automáticamente como se espera, como se puede observar aquí:

```
>>> Partner = self.env['res.partner']
>>> new = Partner.create({'name':'ACME','is_company': True})
>>> print new res.partner(72,)
```

El método `unlink()` borra los registros en el conjunto, como se muestra a continuación:

```
>>> rec = Partner.search([('name','=','ACME')])
>>> rec.unlink()
True
```

El método `write()` toma un diccionario para mapear los valores de los registros. Estos son actualizados en todos los elementos del conjunto y no se devuelve nada, como se muestra a continuación:

```
>>> Partner.write({'comment':'Hello!'})
```

Usar el patrón de registro activo tiene algunas limitaciones; solo actualiza un registro a la vez. Por otro lado, el método `write()` puede actualizar varios campos de varios registros al mismo tiempo usando una sola instrucción de basa de datos. Estas diferencias deben ser tomadas en cuenta en el momento cuando el rendimiento pueda ser un problema.

También vale la pena mencionar a `copy()` para duplicar un registro existente; toma esto como un argumento opcional y un diccionario con los valores que serán escritos en el registro nuevo. Por ejemplo, para crear un usuario nuevo copiando lo desde "Demo User":

```
>>> demo = self.env.ref('base.user_demo')
>>> new = demo.copy({'name': 'Daniel', 'login': 'dr', 'email':''})
>>> self.env.cr.commit()
```

Recuerde que los campos con el atributo `copy=False` no serán tomados en cuenta.

Transacciones y SQL de bajo nivel

Las operaciones de escritura en la base de datos son ejecutadas en el contexto de una transacción de base de datos. Usualmente no tiene que preocuparse por esto ya que el servidor se encarga de ello mientras se ejecutan los métodos del modelo.

Pero en algunos casos, necesitara un mayor control sobre la transacción. Esto puede hacerse a través del cursor `self.env.cr` de la base de datos, como se muestra a continuación:

- `self.env.cr.commit()`: Este escribe las operaciones de escritura cargadas de la transacción.
- `self.env.savepoint()`: Este fija un punto seguro en la transacción para poder revertirla.
- `self.env.rollback()`: Este cancela las operaciones de escritura de la transacción desde el último punto seguro o todo si no fue creado un punto seguro.

> **Truco:** En una sesión de la terminal, la manipulación de los datos no se hará efectiva hasta no usar `self.env.cr.commit()`.

Con el método del cursor `execute()`, podrá ejecutar SQL directamente en la base de datos. Este toma una cadena de texto con la sentencia SQL que se ejecutará y un segundo argumento opcional con una tupla o lista de valores para ser usados como parámetros en el SQL. Estos valores serán usados donde se encuentre el marcador `%s`.

Si esta usando una sentencia `SELECT`, debería retornar los registros. La función `fetchall()` devuelve todas las filas como una lista de tuplas y `dictfetchall()` las devuelve como una lista de diccionarios, como se muestra en el siguiente ejemplo:

```
>>> self.env.cr.execute("SELECT id, login FROM res_users WHERE login=%s OR id=%s",('demo',1))
>>> self.env.cr.fetchall()
[(4, u'demo'), (1, u'admin')]
```

También es posible ejecutar instrucciones en *lenguaje de manipulación de datos (DML)* como `UPDATE` e `INSERT`. Debido a que el servidor mantiene en memoria (cache) los datos, estos puede hacerse inconsistente con los datos reales de la base de datos. Por lo tanto, cuando se use *DML*, la memoria (cache) debe ser limpiada después de su uso, a través de `self.env.invalidate_all()`.

Advertencia: Ejecutar SQL directamente en la base de datos puede tener como consecuencia la generación de inconsistencias en los datos. Debe usarse solo cuando tenga la seguridad de lo que esta haciendo.

Trabajar con hora y fecha

Por razones históricas, los valores de fecha, y de fecha y hora se manejan como cadenas en vez de sus tipos correspondientes en Python. Además los valores de fecha y hora de almacenan en la base de datos en hora UTC. Los formatos usados para representar las cadenas son definidos por:

```
openerp.tools.misc.DEFAULT_SERVER_DATE_FORMAT
openerp.tools.misc.DEFAULT_SERVER_DATETIME_FORMAT
```

Estas se esquematizan como `%Y-%m-%d` y `%Y-%m-%d %H:%M:%S` respectivamente.

Para ayudar a manejar las fechas, `fields.Date` y `fields.Datetime` proveen algunas funciones. Por ejemplo:

```
>>> from openerp import fields
>>> fields.Datetime.now()
'2014-12-08 23:36:09'
>>> fields.Datetime.from_string('2014-12-08 23:36:09')
datetime.datetime(2014, 12, 8, 23, 36, 9)
```

Dado que las fechas y horas son tratadas y almacenadas por el servidor en formato UTC nativo, el cual no toma en cuenta la zona horaria y probablemente es diferente a la zona horaria del usuario, a continuación se muestran algunas otras funciones que pueden ayudar con esto:

- `fields.Date.today()`: Este devuelve una cadena con la fecha actual en el formato esperado por el servidor y usando UTC como referencia. Es adecuado para calcular valores predeterminados.
- `fields.Datetime.now()`: Este devuelve una cadena con la fecha y hora actual en el formato esperado por el servidor y usando UTC como referencia. Es adecuado para calcular valores predeterminados.
- `fields.Date.context_today(record, timestamp=None)`: Este devuelve una cadena con la fecha actual en el contexto de sesión. El valor de la zona horaria es tomado del contexto del registro, y el parámetro opcional es la fecha y hora en vez de la hora actual.
- `fields.Datetime.context_timestamp(record, timestamp)`: Este convierte una hora y fecha nativa (sin zona horaria) en una fecha y hora consciente de la zona horaria. La zona horaria se extrae del contexto del registro, de allí el nombre de la función.

Para facilitar la conversión entre formatos, tanto el objeto `fields.Date` como `fields.Datetime` proporcionan estas funciones:

- `from_string(value)`: convierte una cadena a un objeto fecha o de fecha y hora.
- `to_string(value)`: convierte un objeto fecha o de fecha y hora en una cadena en el formato esperado por el servidor.

Trabajar con campos de relación

Mientras se usa el patrón de registro activo, se pueden asignar conjuntos de registros a los campos relacionales.

- Para un campo muchos a uno, el valor asignado puede ser un único registro (un conjunto de registros `singleton`).

- Para campos a-muchos, sus valores pueden ser asignados con un conjunto de registros, reemplazando la lista de registros enlazados, si existen, con una nueva. Aquí se permite un conjunto de registros de cualquier tamaño.

Mientras se usan los métodos `create()` o `write()`, donde se asigna los valores usando diccionarios, no es posible asignar conjuntos de registros a los valores de los campos relacionales. Se debería usar el ID correspondiente o la lista de Ids.

Por ejemplo, en ves de `self.write({'user_id': self.env.user})`, debería usar `self.write({'user_id':self.env.user.id})`.

Manipular los conjuntos de registros

Seguramente querrá agregar, eliminar o reemplazar los elementos en estos campos relacionados, y esto lleva a la pregunta: ¿como se pueden manipular los conjuntos de registros?

Los conjuntos de registros son inmutables pero pueden ser usados para componer conjuntos de registros nuevos. A continuación se muestran algunas de operaciones soportadas:

- `rs1 | rs2`: Como resultado se tendrá un conjunto con todos los elementos de ambos conjuntos de registros.
- `rs1 + rs2`: Esto también concatena ambos conjuntos en uno.
- `rs1 & rs2`: Como resultado se tendrá un conjunto con los elementos encontrados, que coincidan, en ambos conjuntos de registros.
- `rs1 - rs2`: Como resultado se tendrá un conjunto con los elementos de `rs1` que no estén presentes en `rs2`.

También se puede usar notación de porción, como se muestra a continuación:

- `rs[0]` y `rs[-1]`, retornan el primer elemento y el último elemento.
- `rs[1:]`, devuelve una copia del conjunto sin el primer elemento. Este produce los mismos registros que `rs - rs[0]` pero preservando el orden.

En general, cuando se manipulan conjuntos de registro, debe asumir que el orden del registro no es preservado. Aun así, la agregación y en "slicing" son conocidos por mantener el orden del registro.

Podrá usar estas operaciones de conjuntos para cambiar la lista, eliminando o agregando elementos. Puede observar esto en el siguiente ejemplo:

- `self.task_ids |= task1`: Esto agrega el elemento `task1` si no existe en el conjunto de registro.
- `self.task_ids -= task1`: Elimina la referencia a `task1` si esta presenta en el conjunto de registro.
- `self.task_ids = self.task_ids[:-1]`: Esto elimina el enlace del último registro.

Una sintaxis especial es usada para modificar a muchos campos, mientras se usan los métodos `create()` y `write()` con valores en un diccionario.

Esto fue explicado en el Capítulo 4, en la sección *Configurar valores para los campos de relación*.

Se hace referencia a las siguientes operaciones de ejemplo equivalentes a las precedentes usando `write()`:

- `self.write([(4, task1.id, False)])`: Agrega `task1` al miembro.
- `self.write([(3, task1.id, False)])`: Desconecta (quita el enlace) `task1`.
- `self.write([(3, self.task_ids[-1].id, False)])`: Desconecta (quita en enlace) el último elemento.

Otras operaciones de conjunto de registros

Los conjuntos de registro soportan operaciones adicionales.

Podrá verificar si un registro esta o no incluido en un conjunto, haciendo lo siguiente: record in recordset, record not in recordset. También estas disponibles estas operaciones:

- `recordset.ids`: Esto devuelve la lista con los Ids de los elementos del conjunto.
- `recordset.ensure_one()`: Verifica si es un único registro (*singleton*); si no lo es, arroja una excepción `ValueError`.
- `recordset.exists()`: Devuelve una copia solamente con los registros que todavía existen.
- `recordset.filtered(func)`: Devuelve un conjunto de registros filtrado.
- `recordset.mapped(func)`: Devuelve una lista de valores mapeados.
- `recordset.sorted(func)`: Devuelve un conjunto de registros ordenado.

A continuación se muestran algunos ejemplos del uso de estas funciones:

```
>>> rs0 = self.env['res.partner'].search([])
>>> len(rs0) # how many records?
68
>>> rs1 = rs0.filtered(lambda r: r.name.startswith('A'))
>>> print rs1 res.partner(3, 7, 6, 18, 51, 58, 39)
>>> rs2 = rs1.filtered('is_company')
>>> print rs2 res.partner(7, 6, 18)
>>> rs2.mapped('name') [u'Agrolait', u'ASUSTeK', u'Axelor']
>>> rs2.mapped(lambda r: (r.id, r.name)) [(7, u'Agrolait'), (6, u'ASUSTeK'), (18, u'Axelor')]
>>> rs2.sorted(key=lambda r: r.id, reverse=True)
res.partner(18, 7, 6)
```

El entorno de ejecución

El entorno provee información contextual usada por el servidor. Cada conjunto de registro carga su entorno de ejecución en `self.env` con estos atributos:

- `env.cr`: Es el cursor de base de datos usado actualmente.
- `env.uid`: Este es el ID para el usuario de la sesión.
- `env.user`: Es el registro para el usuario de la sesión.
- `env.context`: Es un diccionario inmutable con un contexto de sesión.

El entorno es inmutable, por lo tanto no puede ser modificado. Pero podrá crear entornos modificables y luego usarlos para ejecutar acciones.

Para esto pueden usarse los siguientes métodos:

- `env.sudo(user)`: Si esto es provisto con un registro de usuario, devuelve un entorno con este usuario. Si no se proporciona un usuario, se usa el usuario de administración, el cual permite ejecutar diferentes sentencias pasando por encima de las reglas de seguridad.
- `env.with_context(dictionary)`: Reemplaza el contexto con uno nuevo.
- `env.with_context(key=value, ...)`: Fija los valores para las claves en el contexto actual.

La función `env.ref()` toma una cadena con un ID externo y devuelve un registro, como se muestra a continuación.

```
>>> self.env.ref('base.user_root')
res.users(1,)
```

Métodos del modelo para la interacción con el cliente

Ha visto los métodos del modelo más importantes usados para generar los conjuntos de registros y como escribir en ellos. Pero existen otros métodos disponibles para acciones más específicas, se muestran a continuación:

- `read([fields])`: Es similar a browse, pero en vez de un conjunto de registros, devuelve una lista de filas de datos con los campos dados como argumentos. Cada fila es un diccionario. Proporciona una representación serializada de los datos que puede enviarse a través de protocolos RPC y esta previsto que sea usada por los programas del cliente y no por la lógica del servidor.
- `search_read([domain], [fields], offset=0, limit=None, order=None)`: Ejecuta una operación de búsqueda seguida por una lectura a la lista del registro resultante. Esta previsto que sea usado por los cliente RPC y ahorrarles el trabajo extra cuando se hace primero una búsqueda y luego una lectura.
- `load([fields], [data])`: Es usado para importar datos desde un archivo CSV. El primer argumento es la lista de campos que se importarán, y este se asigna directamente a la primera fila del CSV. El segundo argumento es una lista de registros, donde cada registro es una lista de valores de cadena de caracteres para para analizar e importar, y este se asigna directamente a las columnas y filas de los datos del CSV. Implementa las características de importación de datos CSV descritas en el Capítulo 4, como el soporte para Ids externos. Es usado por la característica Import del cliente web. Reemplaza el método obsoleto `import_data`.
- `export_data([fields], raw_data=False)`: Es usado por la función Export del cliente web. Devuelve un diccionario con una clave de datos que contiene la lista "data-a" de filas. Los nombres de los campos pueden usar los sufijos `.id` y `/id` usados en los archivos CSV. El argumento opcional `raw_data` permite que los valores de los datos sean exportados con sus tipos en Python, en vez la representación en cadena de caracteres usada en CSV.

Los siguientes métodos son mayormente usados por el cliente web para representar la interfaz y ejecutar la interacción básica:

- `name_get()`: Devuelve una lista de tuplas (ID, name) con un texto que representa a cada registro. Es usado de forma predeterminada para calcular el valor `display_name`, que provee la representación de texto de los campos de relación. Puede ser ampliada para implementar representaciones de presentación personalizadas, como mostrar el código del registro y el nombre en vez de solo el nombre.
- `name_search(name='', args=None, operator='ilike', limit=100)`: Este también devuelve una lista de tuplas (ID, name), donde el nombre mostrado concuerda con el texto en el argumento `name`. Es usado por la UI mientras se escribe en el campo de relación para producir la lista de registros sugeridos que coinciden con el texto escrito. Se usa para implementar la búsqueda de productos, por nombre y por referencia mientras se escribe en un campo para seleccionar un producto.
- `name_create(name)`: Crea un registro nuevo únicamente con el nombre de título. Se usa en el UI para la característica de creación rápida, donde puede crear rápidamente un registro relacionado con solo proporcionar el nombre. Puede ser ampliado para proveer configuraciones predeterminadas mientras se crean registros nuevos a través de esta característica.
- `default_get([fields])`: Devuelve un diccionario con los valores predeterminados para la creación de un registro nuevo. Los valores predeterminados pueden depender de variables como en usuario actual o el contexto de la sesión.
- `fields_get()`: Usado para describir las definiciones del campo, como son vistas en la opción Campos de Vista del menú de desarrollo.
- `fields_view_get()`: Es usado por el cliente web para devolver la estructura de la vista de la UI. Puede darse el ID de la vista como un argumento o el tipo de vista que querrá usando `view_type='form'`.

  Vea el siguiente ejemplo:

```
rset.fields_view_get(view_type='tree')
```

Sobre escribir los métodos predeterminados

Ha aprendido sobre los métodos estándares que provee la API. Pero lo que podrá hacer con ellos no termina allí! También podrá ampliarlos para agregar comportamientos personalizados a sus modelos.

El caso más común es ampliar los métodos `create()` y `write()`. Puede usarse para agregar la lógica desencadenada en cualquier momento que se ejecuten estas acciones. Colocando su lógica en la sección apropiada de los métodos personalizados, podrá hacer que el se ejecute antes o después que las operaciones principales.

Usando el modelo `TodoTask` como ejemplo, podrá crear un `create()` personalizado, el cual puede ser de la siguiente forma:

```
@api.model
def create(self, vals):
    # Code before create
    # Can use the `vals
    dict new_record = super(TodoTask, self).create(vals)
    # Code after create
    # Can use the `new` record created
    return new_record
```

Un método `write()` personalizado seguiría esta estructura:

```
@api.multi
def write(self, vals):
    # Code before write
    # Can use `self`, with the old values
    super(TodoTask, self).write(vals)
    # Code after write
    # Can use `self`, with the new (updated) values
    return True
```

Estos son ejemplos comunes de ampliación, pero cualquier método estándar disponibles para un modelo puede ser heredado en un forma similar para agregar lo a su lógica personalizada.

Estas técnicas abren muchas posibilidades, pero recuerde que otras herramientas que se ajustan mejor a tareas específicas también esta disponibles, y deben darse le prioridad:

- Para tener un valor de campo calculado basado en otro, debe usar campos calculados. Un ejemplo de esto es calcular un total cuando los valores de las líneas cambian.
- Para tener valores predeterminados de campos calculados dinámicamente, podrá usar un campo predeterminado enlazado a una función en vez de a un valor escalar.
- Para fijar valores en otros campos cuando un campos cambia, podrá usar funciones `on-change`. Un ejemplo de esto es cuando escoge un cliente para fijar el tipo de moneda en el documento para el socio correspondiente, el cual puede luego ser cambiado manualmente por el usuario. Tenga en cuenta que `on-change` solo funciona desde las interacciones de ventana y no directamente en las llamadas de escritura.
- Para las validaciones, podrá funciones de restricción decoradas con `@api.constraints(fdl1,fdl2,...)`. Estas son como campos calculados pero se espera que arrojen errores cuando las condiciones no son cumplidas en vez de valores calculados.

Decoradores de métodos del Modelo

Durante su jornada, los métodos que ha encontrado usan los decoradores de la API como `@api.one`. Estos son importantes para que el servidor sepa como manejar los métodos. Ya ha dado alguna explicación de los decoradores usados; ahora recapitule sobre aquellos que están disponibles y de como deben usarse:

- `@api.one`: Este alimenta a la función con un registro a la vez. El decorador realiza la iteración del conjunto de registros por usted y se garantiza que `self` sea un *singleton*. Este es el que debe usar si su lógica solo requiere trabajar con cada registro. También agrega el valor retornado de la función en una lista en cada registro, la cual puede tener efectos secundarios no intencionados.
- `@api.multi`: Este controla un conjunto de registros. Debe usarlo cuando su lógica pueda depender del conjunto completo de registros y la visualización de registros aislados no es suficiente o cuando necesita que el valor de retorno no sea una lista como un diccionario con una acción de ventana. Este es el que más se usa en la práctica ya que `@api.one` tiene algunos costos y efectos de empaquetado de listas en los valores del resultado.

- `@api.model`: Este es un método estático de nivel de clase, y no usa ningún dato de conjunto de registros. Por consistencia, `self` aún es un conjunto, pero su contenido es irrelevante.
- `@api.returns(model)`: Este indica que el método devuelve instancias del modelo en el argumento para el modelo actual, como `res.partner` o `self`.

Los decoradores que tiene propósitos más específicos y que fueron explicados en el Capítulo 5, se muestran a continuación:

- `@api.depends(fld1,...)`: Este es usado por funciones de campos calculados para identificar los cambios en los cuales se debe realizar el (re) calculo.
- `@api.constraints(fld1,...)`: Este es usado por funciones de validación para identificar los cambios en los que se debe realizar la validación.
- `@api.onchange(fld1,...)`: Este es usado por funciones `on-change` para identificar los campos del formulario que detonarán la acción.

En particular, los métodos `on-change` pueden enviar mensajes de advertencia a la interfaz. Por ejemplo, lo siguiente podría advertir al usuario que la cantidad ingresada del producto no esta disponible, sin impedir al usuario continuar. Esto es realizado a través de un método `return` con un diccionario que describa el siguiente mensaje:

```
return {
    'warning': {
        'title': 'Warning!',
        'message': 'The warning text'
    }
}
```

Depuración

Sabe que una buena parte del trabajo de desarrollo es la depuración del código. Para hacer esto frecuentemente hace uso del editor de código que puede fijar pontos de quiebre y ejecutar su programa paso a paso. Hacer esto con Odoo es posible pero tiene sus dificultades.

Si esta usando Microsoft Windows como su estación de trabajo, configurar un entorno capaz de ejecutar en código de Odoo desde la fuente no es una tarea trivial. Además el hecho que Odoo sea un servidor que espera llamadas de un cliente para actuar, lo hace diferente a la depuración de programas del lado del cliente.

Mientras que esto puede ser realizado con Odoo, puede decirse que no es la forma más pragmática de resolver el asunto. Hará una introducción sobre algunas estrategias básicas para la depuración, las cuales pueden ser tan efectivas como algunos IDEs sofisticados, con un poco de práctica.

La herramienta integrada para la depuración de Python, `pdb`, puede hacer un trabajo decente de depuración. Podrá fijar un punto de quiebre insertando la siguiente línea en el lugar deseado:

```
import pdb; pdb.set_trace()
```

Ahora reinicie el servidor para que se cargue la modificación del código. Tan pronto como la ejecución del código alcance la línea, una (`pdb`) linea de entrada de Python será mostrada en la ventana de la terminal en la cual el servidor se esta ejecutando, esperando por el ingreso de datos.

Esta línea de entrada funciona como una línea de comandos de Python, donde puede ejecutar cualquier comando o expresión en el actual contexto de ejecución. Esto significa que las variables actuales pueden ser inspeccionadas e incluso modificadas. Estos son los comandos disponibles más importantes:

- `h`: Es usado para mostrar un resumen de la ayuda del comando `pdb`.
- `p`: Es usado para evaluar e imprimir una expresión.
- `pp`: Este es para una impresión más legible, la cual es útil para los diccionarios y listas muy largos.
- `l`: Lista el código alrededor de la instrucción que será ejecutada a continuación.
- `n` *(next)*: Salta hasta la próxima instrucción.

- `s` *(step)*: Salta hasta la instrucción actual.
- `c` *(continue)*: Continua la ejecución normalmente.
- `u` *(up)*: Permite moverse hacia arriba de la pila de ejecución.
- `d` *(down)*: Permite moverse hacia abajo de la pila de ejecución.

El servidor Odoo también soporta la opción `--debug`. Si se usa, el servidor entrara en un modo *post mortem* cuando encuentre una excepción, en la línea donde se encuentre el error. Es una consola `pdb` y les permite inspeccionar el estado del programa en el momento en que es encontrado el error.

Existen alternativas al depurador de Python. Puede provee los mismos comandos que `pdb` y funciona en terminales de solo texto, pero usa una visualización gráfica más amigable, haciendo que la información útil sea más legible como las variables del contexto actual y sus valores.

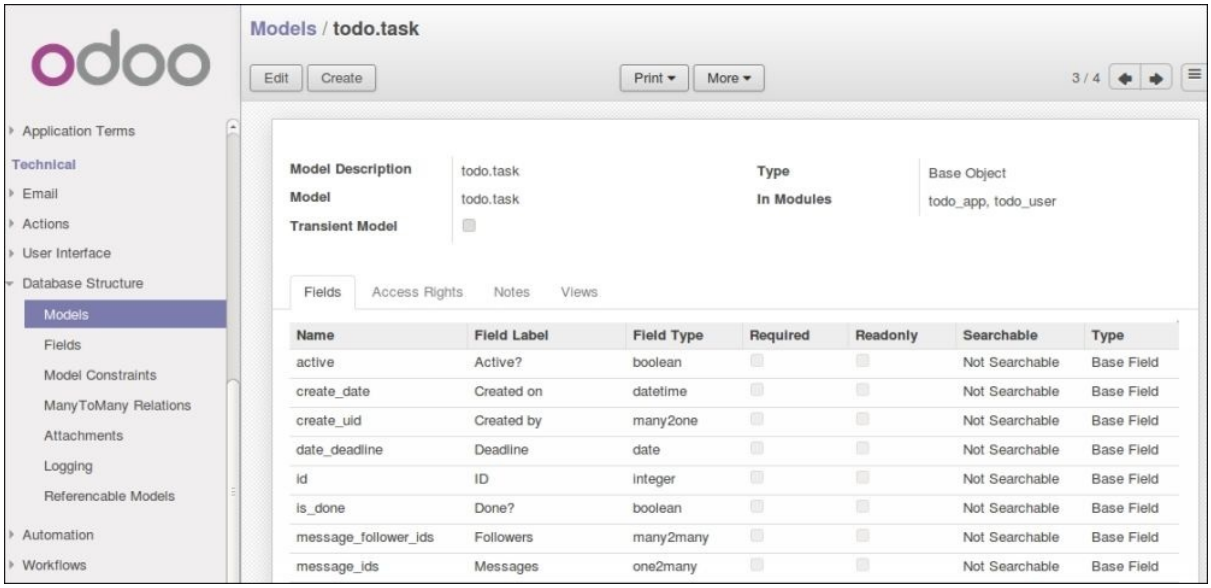

Figura 2.19: Gráfico 7.2 - Vista del modelo todo.task

Puede ser instalado a través del sistema de paquetes o por `pip`, como se muestra a continuación:

```
$ sudo apt-get install python-pudb # using OS packages
$ pip install pudb # using pip, possibly in a virtualenv
```

Funciona como `pdb`; solo necesita usar `pudb` en vez de `pdb` en el código.

Otra opción es el depurador *Iron Python*, `ipdb`, el cual puede ser instalado:

```
$ pip install ipdb
```

A veces solo necesita inspeccionar los valores de algunas variables o verificar si algunos bloques de código son ejecutados. Una sentencia `print` de Python puede perfectamente hacer el trabajo sin parar el flujo de ejecución. Como esta ejecutando el servidor en una terminal, el texto impreso será mostrado en la salida estándar. Pero no será guardado en los registros del servidor si esta siendo escrito en un archivo.

Otra opción a tener en cuenta es fijar los mensajes de registros de los niveles de depuración en puntos sensibles de su código si siente que podrá necesitar investigar algunos problemas en la instancia de despliegue. Solo se requiere elevar el nivel de registro del servidor a `DEBUG` y luego inspeccionar los archivos de registro.

2.7.2 Resumen

En los capítulos anteriores, vio como construir modelos y diseñar vistas. Aquí fue un poco más allá para aprender como implementar la lógica de negocio y usar conjuntos de registros para manipular los datos del modelo.

También pudo ver como la lógica de negocio interactúa con la interfaz y aprendió a crear ayudantes que dialoguen con el usuario y sirvan como una plataforma para iniciar procesos avanzados.

En el próximo capítulo, se enfocara nuevamente en la interfaz, y aprenderá como crear vistas `kanban` avanzadas y a diseñar sus propios reportes de negocio.

2.8 Capítulo 8 - QweB

2.8.1 QweB - Creando vistas Kanban y Reportes

QWeb es un motor de plantillas (*engine template*) por Odoo. Está basado en XML y es utilizado para generar fragmentos y páginas HTML. QWeb fue introducido por primera vez en la versión 7.0 para habilitar vistas Kanban más ricas, y con las versión 8.0, también se usa para la generación de reportes y páginas web CMS (CMS: Sistemas de Gestión de Contenido).

Aquí aprenderás acerca de la sintaxis QWeb y como usarla para crear tus propias vistas Kanban reportes personalizados.

Para entender los tableros Kanban, **Kanban** es una palabra de origen japonés que es usada para representar un método de gestión de colas (queue) de trabajo. Fue inspirado del Sistema de Producción y Fabricación Ligera (lean) de Toyota, y se ha vuelto popular en la la industria del software con su adopción en las metodologías Ágiles.

El **tablero Kanban** es una herramienta para visualizar la cola de trabajo. Los elementos de trabajo están representados por tarjetas que son organizadas en columnas representando las **etapas** (stages) del proceso de trabajo. Nuevos elementos de trabajo inician en la columna más a la izquierda y viaja a través del tablero hasta que alcanzan la columna más a la derecha, representando el trabajo completado.

Iniciándose con el tablero Kanban

La simplicidad y el impacto visual del tablero Kanban los hace excelente para soportar procesos de negocio simples. Un ejemplo básico de un tablero Kanban puede tener tres columnas, como se muestra en la siguiente imagen: ToDo “Por hacer”, Doing “Haciendo” y Done “Hecho”, pero, por supuesto puede ser extendido a cualquier paso de un proceso específico que necesite:

Figura 2.20: Gráfico 8.1 - tablero Kanban

Las vistas Kanban una característica distintiva de Odoo, haciendo fácil implementar estos tableros. Aprenda a cómo usarlos.

Vistas Kanban

En las vistas de formulario, usa mayormente elementos XML específicos, tales como `<field>` y `<group>`, y algunos elementos HTML, tales como `<h1>` o `<div>`. Con las vistas Kanban, es un poco lo opuesto; ellas son plantillas basadas en HTML y soportan solo dos elementos específicos de Odoo, `<field>` y `<button>`.

El HTML puede ser generado dinámicamente usando el motor de plantilla Qweb. Éste procesa los atributos de etiqueta especiales en los elementos HTML para producir el HTML final para ser presentado por el cliente web. Esto proporciona mucho control sobre cómo renderizar el contenido, pero también permite hacer diseños de vistas más complejas.

Las vistas Kanban son tan flexibles que pueden haber muchas formas diferentes de diseñarlas, y puede ser difícil proveer una receta para seguir. Una buena regla general es encontrar un vista Kanban existente similar a lo que querrá alcanzar, y crear su nuevo trabajo de vista Kanban basada en ella.

Observando las vistas Kanban usadas en los módulos estándar, es posible identificar dos estilos de vistas Kanban principales: *viñeta* y *tarjeta*

- Estilo **viñeta**: Muchos ejemplos de las vistas Kanban de estilo **viñeta** pueden ser encontrados en las aplicaciones `CRM`, `Inventario` > `Productos`, y también, **Aplicaciones y Módulos**. Ellos usualmente no tienen borde y son decorados con imágenes en el lado de la izquierda, tal como se muestra en la siguiente imagen:

Figura 2.21: Gráfico 8.2 - Ejemplo de vistas Kanban tipo Viñeta

- Estilo **tarjeta**: Es usualmente usada para mostrar tarjetas organizadas en columnas para las etapas de procesos. Ejemplo de esto son las `CRM` > `Oportunidades` y en `Proyectos` > `Tareas`. El contenido principal es mostrado en el área superior de la tarjeta y la información adicional puede ser mostrada en las áreas inferior derecha e inferior izquierda, tal como se muestra en la siguiente imagen:

Figura 2.22: Gráfico 8.3 - Ejemplo de estilo de tarjeta Kanban

Vera el esqueleto y elementos típicos usados en ambos estilos de vistas tal que puedas sentirte cómodo adaptándolos a tus casos de usos particular.

Diseña vistas Kanban

La primera cosa es crear un nuevo módulo agregando sus vistas Kanban a la lista de tareas por hacer. En un trabajo del mundo real, una situación de uso de un módulo para esto podría ser, probablemente, excesiva y ellas podrían ser perfectamente agregadas directamente en el módulo `todo_ui`. Pero para una explicación más clara, usara un nuevo módulo y evitara demasiados, y posiblemente confusos, cambios en archivos ya creados. Lo nombrara `todo_kanban` y creara los archivos iniciales tal como sigue:

```
$ cd ~/odoo-dev/custom-addons
$ mkdir todo_kanban
$ touch todo_kanban/__init__.py
```

Ahora, edita el archivo descriptor `todo_kanban/__opernerp__.py` tal como sigue:

```
{
    'name': 'To-Do Kanban',
    'description': 'Kanban board for to-do tasks.',
    'author': 'Daniel Reis',
    'depends': ['todo_ui'],
    'data': ['todo_view.xml']
}
```

A continuación, cree el archivo XML donde irán sus nuevas y brillantes vistas Kanban y configurar Kanban como la vista por defecto en la acción `action` de ventana de la aplicación *tareas por hacer*, tal como se muestra a continuación:

```
<?xml version="1.0"?>
<openerp>
  <data>
    <!-- Agrega el modo de vista kanban al menu Action: -->
    <act_window id="todo_app.action_todo_task" name="To-Do Tasks"
                res_model="todo.task" view_mode="kanban,tree,form,calendar,gantt,graph"
                context="{'search_default_filter_my_tasks':True}" />
    <!-- Agregar vista kanban -->
    <record id="To-do Task Kanban" model="ir.ui.view">
      <field name="name">To-do Task Kanban</field>
      <field name="model">todo.task</field>
      <field name="arch" type="xml">
        <!-- vacío por ahora, pero el Kanban irá aquí! -->
      </field>
    </record>
  </data>
</openerp>
```

Ahora tiene ubicado el esqueleto básico para su módulo. Las plantillas usada en las vistas `kanban` y los reportes son extendidos usando las técnicas regulares usadas para otras vistas, por ejemplos usando expresiones XPATH. Para más detalles, ve al Capítulo 3, Herencia – Extendiendo Aplicaciones Existentes.

Antes de iniciar con las vistas kanban, necesita agregar un para de campos en el modelo de la aplicación *tareas por hacer*.

Prioridad y estado Kanban

Los dos campos que son frecuentemente usados en las vistas `kanban` son: priority y kanban state.

- **Priority** permite a los usuarios organizar sus elementos de trabajo, señalando lo que debería estar ubicado primero.
- **Kanban state** señala cuando una tarea está lista para pasar a la siguiente etapa o si es bloqueada por alguna razón. Ambos son soportados por campos `selection` y tienen widgets específicos para ser usados en las vistas de formulario y kanban.

Para agrega estos campos a su modelo, agregara al archivo `todo_kanban/todo_task.py`, tal como se muestra a continuación:

```
from openerp import models, fields

class TodoTask(models.Model):
    _inherit = 'todo.task'

    priority = fields.Selection([
                                ('0','Low'),
                                ('1','Normal'),
                                ('2','High')],
                               'Priority',default='1')
    kanban_state = fields.Selection([
                                    ('normal', 'In Progress'),
                                    ('blocked', 'Blocked'),
                                    ('done', 'Ready for next stage')],
                                   'Kanban State', default='normal')
```

No olvide el archivo `todo_kanban/__init__.py` que cargará el código precedente:

```
from . import todo_model
```

Elementos de la vista kanban

La arquitectura de la vista kanban tiene un elemento superior y la siguiente estructura básica:

```
<kanban>
  <!-- Fields to use in expressions... -->
  <field name="a_field" />
  <templates>
    <t t-name="kanban-box">
      <!-- HTML Qweb template ... -->
    </t>
  </templates>
</kanban>
```

El elemento contiene las plantillas para los fragmentos HTML a usar —uno o más. La plantilla principal a ser usada debe ser nombrada `kanban-box`. Otras plantillas son permitidas para fragmentos HTML para se incluido en la plantilla principal.

Las plantillas usan html estándar, pero pueden incluir etiquetas `<field>` para insertar campos del modelo. También pueden ser usadas algunas directivas especiales de Qweb para la generación dinámica de contenido, tal como el `t-name` usado en el ejemplo previo.

Todos los campos del modelo usados deben ser declarados con una etiqueta `<field>`. Si ellos son usados solo en expresiones, tiene que declararlos antes de la sección `<templates>`. Uno de esos campos se le permite tener un valor agregado, mostrado en en el área superior de las columnas `kanban`. Esto se logra mediante la adición de un atributo con la agregación a usar, por ejemplo:

```
<field name="effort_estimated" sum="Total Effort" />
```

Aquí, la suma para el campo de estimación de esfuerzo es presentada en el área superior de las columnas `kanban` con la etiqueta Total Effort. Las agregaciones soportadas son `sum`, `avg`, `min`, `max` y `count`.

El elemento superior también soporta algunos atributos interesantes:

- `default_group_by`: Establece el campo a usar para la agrupación por defecto de columnas.
- `default_order`: Establece un orden por defecto para usarse en los elementos `kanban`.
- `quick_create="false"`: Deshabilita la opción de creación rápida en la vista `kanban`.
- `class`: Añade una clase CSS al elemento raíz en la vista `kanban` renderizada.

Ahora de una mirada más de cerca a las plantillas Qweb usadas en las vistas `kanban`.

Las plantillas QWeb de la vista de viñetas `kanban`, su estructura lucen así:

```
<t t-name="kanban-box"/>
  <div class="oe_kanban_vignette">
    <!-- Left side image: -->
    <img class="oe_kanban_image" name="..." >
    <div class="oe_kanban_details">
      <!-- Title and data -->
      <h4>Title</h4>
      <br>Other data <br/>
      <ul>
        <li>More data</li>
      </ul>
    </div>
  </div>
</t>
```

Puedes ver las dos (02) clases CSS principales provistas para los estilos de viñeta `kanban`: `oe_kanban_vignette` para el contenedor superior y `oe_kanban_details` para el contenido de datos.

La vista completa de viñeta `kanban` para las tareas por hacer es como sigue:

```
<kanban>
    <templates>
        <t t-name="kanban-box">
          <div class="oe_kanban_vignette">
            <img t-att-src="kanban_image('res.partner',
                                         'image_medium',
                                         record.id.value)"
                 class="oe_kanban_image"/>
            <div class="oe_kanban_details">
                <!-- Title and Data content -->
                <h4>
                  <a type="open">
                    <field name="name"/>
                  </a>
                </h4>
                <field name="tags" />
                <ul>
                  <li><field name="user_id" /></li>
                  <li><field name="date_deadline"/></li>
                </ul>
                <field name="kanban_state" widget="kanban_state_selection"/>
                <field name="priority" widget="priority"/>
            </div>
          </div>
        </t>
    </templates>
</kanban>
```

Podrá ver los elementos discutidos hasta ahora, y también algunos nuevos. En la etiqueta , tiene el atributo QWeb especial `t-att-src`. Esto puede calcular el contenido `src` de la imagen desde un campo almacenado en la base de datos. Se explicara esto en otras directivas QWeb en un momento. También podrá ver el uso del atributo especial `type` en la etiqueta `<a>`. Eche un vistazo más de cerca.

Acciones en las vistas Kanban

En las plantillas Qweb, la etiqueta para enlaces puede tener un atributo `type`. Este establece el tipo de acción que el enlace ejecutará para que los enlaces puedan actuar como los botones en los formularios regulares. En adición a los elementos `<button>`, las etiquetas `<a>` también pueden ser usadas para ejecutar acciones Odoo.

Así como en las vistas de formulario, el tipo de acción puede ser acción u objeto, y debería ser acompañado por atributo nombre, que identifique la acción específica a ejecutar. Adicionalmente, los siguientes tipos de acción también están disponibles:

- `open`: Abre la vista formulario correspondiente.
- `edit`: Abre la vista formulario correspondiente directamente en el modo de edición.
- `delete`: Elimina el registro y remueve el elemento de la vista kanban.

La vista de tarjeta kanban El **tarjeta** de `kanban` puede ser un poco más complejo. Este tiene un área de contenido principal y dos sub-contenedores al pie, alineados a cada lado de la tarjeta. También podría contener un botón de apertura de una acción de menú en la esquina superior derecha de la tarjeta.

El esqueleto para esta plantilla se vería así:

```
<t t-name="kanban-box">
  <div class="oe_kanban_card">
    <div class="oe_dropdown_kanban oe_dropdown_toggle">
    <!-- Top-right drop down menu -->
    </div>
    <div class="oe_kanban_content">
      <!-- Content fields go here... -->
      <div class="oe_kanban_bottom_right"></div>
      <div class="oe_kanban_footer_left"></div>
    </div>
  </div>
</t>
```

Una **tarjeta** `kanban` es más apropiada para las tareas to-do, así que en lugar de la vista descrita en la sección anterior, mejor debería usar la siguiente:

```
<t t-name="kanban-box">
  <div class="oe_kanban_card">
    <div class="oe_kanban_content">
        <!-- Option menu will go here! -->
        <h4>
          <a type="open">
            <field name="name" />
          </a>
        </h4>
        <field name="tags" />
        <ul>
            <li><field name="user_id" /></li>
            <li><field name="date_deadline" /></li>
        </ul>
        <div class="oe_kanban_bottom_right">
            <field name="kanban_state" widget="kanban_state_selection"/>
        </div>
        <div class="oe_kanban_footer_left">
            <field name="priority" widget="priority"/>
        </div>
    </div>
  </div>
</t>
```

Hasta ahora ha visto vistas `kanban` estáticas, usando una combinación de HTML y etiquetas especiales (`field`, `button`, `a`). Pero podrá tener resultados mucho más interesantes usando contenido HTML generado dinámicamente. Vea como podrá hacer eso usando Qweb.

Agregando contenido dinámico Qweb

El analizador Qweb busca atributos especiales (directivas) en las plantillas y las reemplaza con HTML generado dinámicamente.

Para las vistas `kanban`, el análisis se realiza mediante Javascript del lado del cliente. Esto significa que las evaluaciones de expresiones hechos por Qweb deberían ser escritas usando la sintaxis Javascript, no Python.

Al momento de mostrar una vista kanban, los pasos internos son aproximadamente los siguientes:

- Obtiene el XML de la plantilla a renderizar.
- Llama al método de servidor `read()` para obtener la data de los campos en las plantillas.
- Ubica la plantilla `kanban-box` y la analiza usando Qweb para la salida de los fragmentos HTML finales.
- Inyecta el HTML en la visualización del navegador (el DOM).

Esto no significa que sea exacto técnicamente. Es solo un mapa mental que puede ser útil para entender como funcionan las cosas en las vistas kanban.

A continuación explorara las distintas directiva Qweb disponibles, usando ejemplos que mejorarán su tarjeta `kanban` de la tarea to-do.

Renderizado Condicional con t-if

La directiva `t-if`, usada en el ejemplo anterior, acepta expresiones JavaScript para ser evaluadas. La etiqueta y su contenido serán renderizadas si la condición se evalúa verdadera.

Por ejemplo, en la tarjeta `kanban`, para mostrar el esfuerzo estimado de la Tarea, solo si este contiene un valor, después del campo `date_deadline`, agrega lo siguiente:

```
<t t-if="record.effort_estimate.raw_value > 0">
    <li>Estimate <field name="effort_estimate"/></li>
</t>
```

El contexto de evaluación JavaScript tiene un objeto de registro que representa el registro que está siendo renderizado, con las campos solicitados del servidor. Los valores de campo pueden ser accedidos usando el atributo `raw_value` o el `value`:

- `raw_value`: Este es el valor retornado por el método de servidor `read()`, así que se ajusta más para usarse en expresiones condicionales.
- `value`: Este es formateado de acuerdo a las configuraciones de usuario, y está destinado a ser mostrado en la interfaz del usuario.

El contexto de evaluación de Qweb también tiene referencias disponibles para la instancia JavaScript del cliente web. Para hacer uso de ellos, se necesita una buena comprensión de la arquitectura de cliente web, pero no podrá llegar a ese nivel de detalle. Para propósitos referenciales, los identificadores siguientes están disponibles en la evaluación de expresiones Qweb:

- `widget`: Esta es una referencia al objeto widget `KanbanRecord`, responsable por el renderizado del registro actual dentro de la tarjeta `kanban`. Expone algunas funciones de ayuda útiles que podrá usar.
- `record`: Este es un atajo para `widget.records` y provee acceso a los campos disponibles, usando notación de puntos.
- `read_only_mode`:
- `widget`: Esta es una referencia al widget actual `KanbanRecord` objeto, responsable de la representación del registro actual en un tarjeta `kanban`. Expone algunas funciones `helper` útiles que puede usar.
- `record`: Este es un acceso directo para `widget.records` y proporciona acceso a los campos disponibles, utilizando la notación de puntos.
- `read_only_mode`: Esto indica si la vista actual está en modo de lectura (y no en modo de edición). Es un atajo para `widget.view.options.read_only_mode`.
- `instance`: Esta es una referencia a la instancia completa del cliente web.

También es digno de mención que algunos caracteres no están permitidos dentro expresiones El signo inferior a (<) es un caso así. Puedes usar un negado >= en su lugar. De todos modos, hay símbolos alternativos disponibles para operaciones de desigualdad de la siguiente manera:

- `lt`: Esto es para *menor que*.
- `lte`: Esto es para *menor o igual que*.
- `gt`: Esto es para *mayor que*.
- `gte`: Esto es para *mayor o igual que*.

Renderinzando valores con t-esc y t-raw

Usted ha utilizado el elemento para representar el contenido del campo. Pero los valores de campo también se puede presentar directamente sin una etiqueta. La directiva `t-esc` evalúa una expresión y representa su valor escapado de HTML, como se muestra en el seguimiento:

```
<t t-esc="record.message_follower_ids.raw_value" />
```

En algunos casos, y si se garantiza que los datos de origen sean seguros, la directiva `t-raw` puede se utilizará para representar el valor sin procesar del campo, sin ningún escape, como se muestra en el siguiente código:

```
<t t-raw="record.message_follower_ids.raw_value" />
```

Bucle de renderizado con t-foreach

Un bloque de HTML puede repetirse iterando a través de un bucle. Usted podrá usar para agregar los avatares de los seguidores de tareas a las tareas que comienzan por representando solo las ID de socio de la tarea, de la siguiente manera:

```
<t t-foreach="record.message_follower_ids.raw_value" t-as="rec"/>
  <t t-esc="rec" />
</t>
```

La directiva `t-foreach` acepta una expresión JavaScript que evalúa colección para iterar. En la mayoría de los casos, este será solo el nombre de un campo de relación *a muchos*. Se utiliza con una directiva `t-as` para establecer el nombre que se utilizará para referirse a cada elemento en la iteración.

En el ejemplo anterior, recorre los seguidores de la tarea, almacenados en el campo `message_follower_ids`. Como hay espacio limitado en la tarjeta `kanban`, podría haber usado la función de JavaScript `slice()` para limitar el número de seguidores a mostrar, como se muestra a continuación:

```
t-foreach="record.message_follower_ids.raw_value.slice(0, 3)"
```

La variable `rec` contiene cada avatar de iteraciones almacenado en la base de datos. Las vistas Kanban proporcionan una función auxiliar para generar convenientemente eso: `kanban_image()`. Acepta como argumentos el nombre del modelo, el nombre del campo sosteniendo la imagen que quiere y la ID para recuperar el registro.

Con esto, puede reescribir el bucle de seguidores de la siguiente manera:

```
<div>
  <t t-foreach="record.message_follower_ids.raw_value.slice(0, 3)" t-as="rec">
      <img t-att-src="kanban_image(
                            'res.partner',
                            'image_small', rec)"
           class="oe_kanban_image oe_kanban_avatar_smallbox"/>
  </t>
</div>
```

Lo usa para el atributo `src`, pero cualquier atributo puede ser dinámicamente generado con un prefijo `t-att-`.

Sustitución de cadenas en atributos con los prefijos `t-attf-`.

Otra forma de generar dinámicamente atributos de etiqueta es usar cadena sustitución. Esto es útil para generar partes de cadenas más grandes dinámicamente, como una dirección URL o nombres de clase CSS.

La directiva contiene bloques de expresión que serán evaluados y reemplazado por el resultado. Estos están delimitados por `{{ and }}` o por `#{ and }`. El contenido de los bloques puede ser cualquier expresión JavaScript válida y puede usar cualquiera de las variables disponibles para las expresiones QWeb, como registro y widget.

Ahora va a modificar para usar una sub-plantilla. Deberá comenzar agregando otra plantilla para su archivo XML, dentro del elemento, después del nodo `<t t-name="kanban-box">`, como se muestra a continuación:

```
<t t-name="follower_avatars">
    <div>
        <t t-foreach="record.message_follower_ids.raw_value.slice(0, 3)" t-as="rec">
        <img t-att-src="kanban_image('res.partner', 'image_small', rec)"
             class="oe_kanban_image oe_kanban_avatar_smallbox"/>
        </t>
  </div>
</t>
```

Llamarlo desde la plantilla principal de `kanban-box` es bastante sencillo para cada uno existe en el valor del llamador al realizar la llamada de sub-plantilla como sigue:

```
<t t-call="follower_avatars">
    <t t-set="arg_max" t-value="3" />
</t>
```

Todo el contenido dentro del elemento `t-call` también está disponible para sub-plantilla a través de la variable mágica `0`. En lugar del argumento de las variables, puede definir un fragmento de código HTML que podría insertarse en la sub-plantilla usando la sintaxis `<t t-raw="0" />`.

2.8.2 Otras directivas QWeb

Usted ha revisado las directivas Qweb más importantes, pero hay algunos más que debe tener en cuenta. Usted ha visto lo básico sobre Vistas kanban y plantillas QWeb. Todavía hay algunas técnicas que puede utilizar para brindar una experiencia de usuario más rica a nuestras tarjetas `kanban`.

Adición de un menú de opciones de la tarjeta Kanban

Las tarjetas `kanban` pueden tener un menú de opciones, ubicado en la parte superior derecha. Las acciones usuales son para editar o eliminar el registro, pero cualquier acción invocable desde un el botón es posible. También hay disponible un widget para configurar la tarjeta.

```
      </a>
    </li>
  </t>
  <t t-if="widget.view.is_action_enabled('delete')">
    <li><a type="delete">Delete</a></li>
  </t>
  <!-- Color picker option: -->
  <li>
    <ul class="oe_kanban_colorpicker"
        data-field="color"/>
    </ul>
  </li>
</div>
```

Básicamente es una lista HTML de elementos. Las opciones **Editar** y **Eliminar** usa QWeb para hacerlos visibles solo cuando sus acciones estén habilitadas en el ver. La función `widget.view.is_action_enabled` nos permite inspeccionar si las acciones de edición y eliminación están disponibles y para decidir qué hacer disponible para el usuario actual.

Adición de colores para tarjetas Kanban

La opción del selector de color permite al usuario elegir el color de una tarjeta `kanban`. El color se almacena en un campo modelo como un índice numérico.

Debería comenzar agregando este campo al modelo de tareas pendientes, agregando al archivo `todo_kanban/todo_model.py` en la siguiente línea:

```
color = fields.Integer('Color Index')
```

Aquí usa el nombre habitual para el campo, el color, y esto es lo que es esperado en el atributo de campo `data-` en el selector de color.

A continuación, para que los colores seleccionados con el selector tengan algún efecto en el tarjeta, debe agregar algunos CSS dinámicos basados en el valor del campo de color. En la vista `kanban`, justo antes de la etiqueta, también debe declarar el color campo, como se muestra a continuación:

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

Y, necesita reemplazar el elemento superior de la tarjeta `kanban`:

```
<div class="oe_kanban_card">
```

con lo siguiente:

```
<div t-attf-class="oe_kanban_card
                   #{kanban_color(record.color.raw_value)}"/>
```

La función auxiliar `kanban_color` hace la traducción del índice de color al nombre de la clase CSS correspondiente.

Y eso. Una función auxiliar para esto está disponible en vistas `kanban`.

Por ejemplo, para limitar nuestros títulos de tareas pendientes a los primeros 32 caracteres, debe reemplazar el elemento con lo siguiente:

```
<t t-esc="kanban_text_ellipsis(record.name.value, 32)" />
```

Archivos CSS y JavaScript personalizados

Como usted ha visto, las vistas `kanban` son principalmente HTML y hacen un uso intensivo de clases CSS. Usted ha estado introduciendo algunas clases CSS de uso frecuente proporcionado por el producto estándar. Pero para obtener mejores resultados, los módulos también pueden agregar su propio CSS.

Usted no va a entrar en detalles aquí sobre cómo escribir CSS, pero funciona, dado que no tiene HTML en PDF. Probablemente no sea lo que obtendrá ahora su sistema. Deje mostrar usted necesita `Wkhtmltopdf` para imprimir un pdf versión de la biblioteca de tiempo de informes

- `user`: Este es el registro del usuario que ejecuta el informe.
- `res_company`: Este es el registro para el usuario actual. Diseño del Interfaz de usuario, con un widget adicional para configurar el widget a usar para representar el campo.

Un ejemplo común es un campo monetario, como se muestra a continuación:

```
<span t-field="o.amount"
      t-field-options='{
                "widget": "monetary",
                "display_currency": "o.pricelist_id.currency_id"}'/>
```

Un caso más sofisticado es el widget de contacto, utilizado para formatear direcciones, como se muestra a continuación:

```
<div t-field="res_company.partner_id"
     t-field-options='{
        "widget": "contact",
        "fields": ["address", "name", "phone", "fax"],
        "no_marker": true}' />
```

Por defecto, algunos pictogramas, como un teléfono, se muestran en la dirección. La opción `no_marker="true"` los desactiva.

Habilitando la traducción de idiomas en reportes

Una función auxiliar, `translate_doc()`, está disponible para dinámicamente traducir el contenido del informe a un idioma específico.

Necesita el nombre del campo donde se puede encontrar el idioma a utilizar. Con frecuencia será el Socio (Partner) al que se enviará el documento, generalmente almacenado en `partner_id.lang`. En su caso, también tiene un método menos eficiente.

Si puede ganar importancia en el conjunto de herramientas Odoo. Finalmente tuviste una descripción general sobre cómo crear informes, también utilizando el motor QWeb.

2.8.3 Resumen

En el siguiente capítulo, explorara cómo aprovechar la API RPC para interactuar con Odoo desde aplicaciones externas.

2.9 Capítulo 9 - API Externa

2.9.1 API Externa – Integración con otros Sistemas

Hasta ahora, usted ha estado trabajando con el código del lado del servidor. Sin embargo, el servidor Odoo también proporciona una API externa, que es utilizada por su cliente web y también está disponible para otras aplicaciones cliente.

En este capítulo, aprenderá cómo usar la API externa de Odoo de su propios programas de clientes HTTP. Para simplificar, se centraran en la clientela disponible para Python.

Configurar un cliente Python

Se puede acceder a la API de Odoo externamente usando dos (02) protocolos diferentes: `XML-RPC` y `JSON-RPC`. Cualquier programa externo capaz de implementar un el cliente para uno de estos protocolos podrá interactuar con un servidor Odoo. Para evitar introducir lenguajes de programación adicionales, se seguirá usando Python para explorar la API externa.

Hasta ahora, usted ha estado ejecutando código Python solo en el servidor. Esta vez, usara Python en el lado del cliente, por lo que es posible que pueda necesita hacer una configuración adicional en su estación de trabajo.

Para seguir los ejemplos de este capítulo, deberá poder ejecutar archivos Python en su computadora de trabajo. El servidor Odoo requiere **Python 2**, pero su cliente `RPC` puede estar en cualquier idioma, por lo que **Python 3** estará bien. Sin embargo, dado que algunos lectores pueden estar ejecutando el servidor en el mismo máquina en la que están trabajando (¡hola usuarios de Ubuntu!), será más simple para que todos sigan si sigue usando a **Python 2**.

Si está utilizando *Ubuntu* o *Macintosh*, probablemente **Python** ya esté instalado Abra una consola de terminal, escriba `python`, y debería estar recibido con algo como lo siguiente:

```
Python 2.7.8 (default, Oct 20  2014, 15:05:29)
[GCC 4.9.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Nota: Los usuarios de Windows pueden encontrar un instalador y también ponerse al día rápidamente. Los paquetes de instalación oficiales se pueden encontrar en https://www.python.org/downloads/.

2.9.2 Llamando a la API Odoo usando XML-RPC

El método más simple para acceder al servidor es usar `XML-RPC`. Usted puede usar la biblioteca `xmlrpclib` de la biblioteca estándar de **Python** para esto. Recuerda que esta programando un cliente para conectarse a un servidor, entonces necesita una instancia del servidor Odoo ejecutándose para conectarse. En sus ejemplos, se asumirá que una instancia del servidor Odoo se está ejecutando en la misma máquina (`localhost`), pero puede usar cualquier dirección IP o nombre de servidor, si el servidor se está ejecutando en otra máquina.

Abriendo una conexión XML-RPC

Usted va a tener un primer contacto con la API externa. Iniciar una consola de **Python** y escriba lo siguiente:

```
>>> import xmlrpclib
>>> srv, db = 'http://localhost:8069', 'v8dev' >>> user, pwd = 'admin', 'admin'
>>> common = xmlrpclib.ServerProxy('%s/xmlrpc/2/common' % srv)
>>> common.version()
{'server_version_info': [8, 0, 0, 'final', 0], 'server_serie': '8.0', 'server_version': '8.0', 'p
```

Aquí, importa la biblioteca `xmlrpclib` y luego la configura algunas variables con la información para la ubicación del servidor y las credenciales de conexión. Siéntase libre de adaptarlos a su configuración específica.

A continuación, configura el acceso a los servicios públicos del servidor (no requiere un inicio de sesión), expuesto en el *endpoint* `/xmlrpc/2/common`. Uno de los métodos que están disponibles es `version()`, que inspecciona la versión del servidor. Este se usa para confirmar que puede comunicar con el servidor.

Otro método público es `authenticate()`. De hecho, esto no crea un sesión, como puede ser llevado a creer. Este método solo confirma que el nombre de usuario y la contraseña son aceptados y devuelve la identificación de usuario que debe usarse en solicitudes en lugar del nombre de usuario, como se muestra aquí:

```
>>> uid = common.authenticate(db, user, pwd, {})
>>> print uid
1
```

Leyendo data desde el servidor

Con `XML-RPC`, no se mantiene ninguna sesión y la autenticación de las credenciales se envían con cada solicitud. Esto agrega algo de sobrecarga al protocolo, pero hace que sea más fácil de usar. A continuación, configure el acceso a métodos de servidor que necesitan un inicio de sesión para acceder. Estos están expuestos en el punto final `/xmlrpc/2/object`, como se muestra a continuación:

```
>>> api = xmlrpclib.ServerProxy('%s/xmlrpc/2/object' % srv)
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'search_count', [[]])
70
```

Aquí, esta haciendo su primer acceso a la API del servidor, realizando un conteo con los registros de socios (*Partners*). Los métodos se llaman usando el método `execute_kw()` que toma los siguientes argumentos:

- El nombre de la base de datos a conectarse.
- La conexión ID de usuario.

- La contraseña de usuario.
- El nombre del modelo de destino identificador.
- El método para llamar Una lista de argumentos posicionales.
- Un diccionario opcional con argumentos de palabras clave.

El ejemplo anterior llama al método `search_count` del modelo `res.partner` con un argumento posicional, `[]`, y sin argumentos de palabras clave. Los argumento posicional es un dominio de búsqueda; ya que esta proporcionando una lista vacía, cuenta todos los socios (*Partners*).

Las acciones frecuentes son `search` y `read`. Cuando se llama desde el `RPC`, el método `search` devuelve una lista de ID que coinciden con un dominio. El método de navegación no está disponible desde el `RPC`, y el método `read` debe usarse en su lugar para, dada una lista de ID de registro, recupere sus datos, como se muestra en el siguiente código:

```
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'search', [[('country_id', '=', 'be'), ('parent_id
[43,  42]
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'read', [[43]], {'fields': ['id', 'name', 'parent_
[{'parent_id': [7, 'Agrolait'], 'id':43, 'name': 'Michel Fletcher'}]
```

Tenga en cuenta que para el método `read`, esta utilizando un argumento posicional para la lista de ID, `[43]` y un argumento de palabra clave, campos. También puede observar que los campos relacionales se recuperan como un par, con los ID de registro y nombre para mostrar. Eso es algo a tener en cuenta cuando procesando los datos en su código.

La combinación de búsqueda y lectura es tan frecuente que un método `search_read` se proporciona el método para realizar ambas operaciones en un solo paso. El mismo resultado ya que los dos pasos anteriores se pueden obtener con lo siguiente:

```
>>> api.execute_kw(
        db, uid, pwd, 'res.partner', 'search_read', [
        [('country_id', '=', 'be'), ('parent_id', '!=', False)]
        ], {'fields': ['id', 'name', 'parent_id']}
    )
```

El método `search_read` se comporta como leído, pero espera como primero argumento posicional un dominio en lugar de una lista de ID. Merece la pena mencionando que el argumento de campo en `read` y `search_read` no es obligatorio. Si no se proporciona, se recuperarán todos los campos.

2.9.3 Llamando otros métodos

Todos los métodos de modelo restantes están expuestos a través de `RPC`, excepto aquellos que comienzan con _ que se consideran privados. Esto significa que usted puede usar `create`, `write` y `unlink` para modificar datos en el servidor como sigue:

```
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'create', [{'name':'Packt'}])
75
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'write', [[75], {'name': 'Packt Pub'}])
True
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'read', [[75], ['id', 'name']])
[{'id':  75, 'name': 'Packt Pub'}]
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'unlink', [[75]])
True
```

Una limitación del protocolo `XML-RPC` es que no admite los valores `None`. La implicación es que los métodos que no devuelven nada no ser utilizable a través de `XML-RPC`, ya que están devolviendo implícitamente `None`. Es por eso que los métodos siempre deben terminar con al menos una declaración de retorno `True`.

Escribir una aplicación de escritorio de **Notes** haga algo interesante con la *API RPC*. ¿Qué pasaría si los usuarios pudieran administrar sus tareas pendientes de Odoo directamente desde el escritorio de su computadora? Usted va a escribir una aplicación Python simple hacer exactamente eso, como se muestra en la siguiente captura de pantalla:

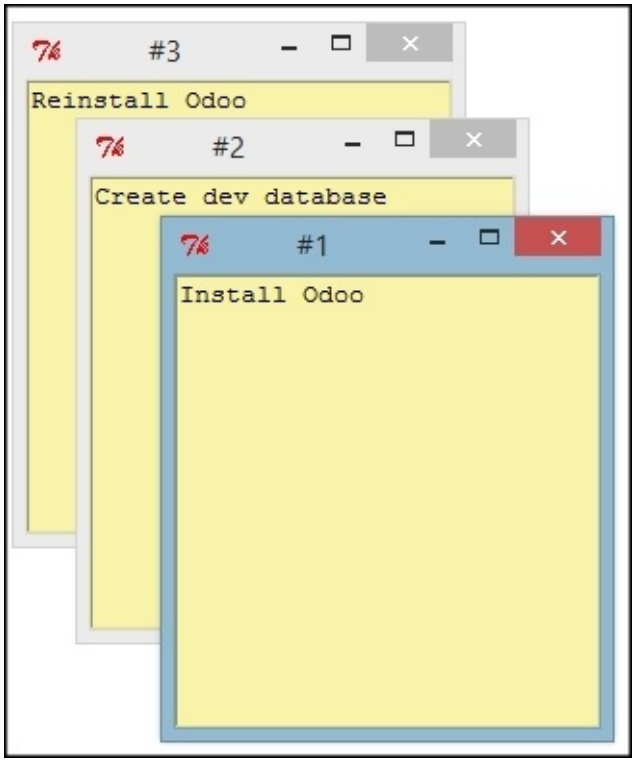

Figura 2.23: Gráfico 9.1 - Cliente Python Tk

Para mayor claridad, lo divide en dos archivos: uno para interactuar con el servidor backend, en el archivo `note_api.py`, y otro con la interfaz gráfico de usuario, en el archivo `note_gui.py`.

Capa de comunicación con Odoo

Cree una clase para configurar la conexión y almacenar su información. Debería exponer dos métodos:

- El método `get()` para recuperar datos de la tarea.
- El método `set()` para crear o actualizar tareas.

Seleccione un directorio para alojar los archivos de aplicación y cree el archivo `note_api.py`. Puede empezar por agregando el constructor de clase, de la siguiente manera:

```
import  xmlrpclib

class NoteAPI():

    def __init__(self, srv, db, user, pwd):

        common = xmlrpclib.ServerProxy('%s/xmlrpc/2/common' % srv)
        self.api = xmlrpclib.ServerProxy('%s/xmlrpc/2/object' % srv)
        self.uid = common.authenticate(db, user, pwd, {})
        self.pwd = pwd
        self.db = db
        self.model = 'todo.task'
```

Aquí almacena en el objeto creado toda la información necesaria para ejecutar llamadas en un modelo: la referencia API, `uid`, `cpassword`, `database name` y el `model` a usar. A continuación definirá un método helper para ejecutar las llamadas. Aprovecha los datos almacenados del objeto para proporcione una firma de función más pequeña, como se muestra a continuación:

```
def execute(self, method, arg_list, kwarg_dict=None):
    return self.api.execute_kw(self.db,
                               self.uid,
                               self.pwd,
                               self.model,
                               method,
                               arg_list,
                               kwarg_dict or {})
```

Ahora puede usarlo para implementar los métodos de nivel superior `get()` y `set()`. El método `get()` aceptará una lista opcional de ID para recuperar. Si ninguno está en la lista, todos los registros serán devueltos, como se muestra aquí:

```
def get(self, ids=None):
    domain = [('id', 'in', ids)]
    if ids else []
    fields = ['id', 'name']
    return  self.execute('search_read', [domain, fields])
```

El método `set()` tendrá como argumentos el texto de la tarea a escribir, y un ID opcional. Si no se proporciona ID, se creará un nuevo registro. Eso devuelve la ID del registro escrito o creado, como se muestra aquí:

```
def set(self, text, id=None):
    if id:
        self.execute('write', [[id], {'name': text}])
    else:
        vals = {'name': text, 'user_id': self.uid}
        id = self.execute('create', [vals])
    return id
```

Termine el archivo con un pequeño fragmento de código de prueba que se ejecutará si ejecuta el archivo Python:

```
if __name__ == '__main__':
    srv, db = 'http://localhost:8069', 'v8dev'
    user, pwd = 'admin', 'admin'
    api =  NoteAPI(srv, db, user, pwd)
    from pprint import pprint
    pprint(api.get())
```

Si ejecuta el script **Python**, debería ver el contenido de su tareas pendientes impresas. Ahora que tiene un contenedor simple alrededor de su backend de Odoo, trate con la interfaz de usuario de escritorio.

2.9.4 Creando la GUI

Su objetivo aquí era aprender a escribir la interfaz entre una aplicación externo y el servidor Odoo, y esto se hizo en el anterior sección. Pero sería una pena no ir más allá y, de hecho, poniéndolo a disposición del usuario final.

Para mantener la configuración tan simple como posible, usara la librería `Tkinter` para implementar la interfaz gráfica de usuario. Como es parte de la biblioteca estándar, no requiere ninguna instalación adicional. No es el objetivo explicar cómo funciona `Tkinter`, por lo que faltarán explicaciones al respecto.

Cada tarea debe tener una pequeña ventana amarilla en el escritorio. Estas ventanas tendrá un solo widget de texto. Al presionar *Ctrl + N* se abrirá una nueva *Nota*, y presionando *Ctrl + S* escribirá el contenido de la nota actual al servidor Odoo.

Ahora, junto con el archivo `note_api.py`, cree un nuevo archivo `note_gui.py`. Primero importará los módulos y widgets de `Tkinter` que usara, y luego la clase `NoteAPI`, como se muestra a continuación:

```
from Tkinter import Text, Tk
import tkMessageBox
from note_api import NoteAPI
```

A continuación, cree su propio widget de texto derivado del `Tkinter`. Cuando al crear una instancia, esperará una referencia de API que se utilizará para guardar la acción, y también el texto y la ID de la tarea, como se muestra a continuación:

```
class NoteText(Text):
    def __init__(self, api, text='', id=None):
        self.master = Tk()
        self.id = id
        self.api = api
        Text.__init__(self, self.master, bg='#f9f3a9',
                      wrap='word', undo=True)
        self.bind('<Control-n>', self.create)
        self.bind('<Control-s>', self.save)
        if id:
            self.master.title('#%d' % id)
            self.delete('1.0', 'end')
            self.insert('1.0', text)
            self.master.geometry('220x235')
            self.pack(fill='both', expand=1)
```

El método constructor `Tk()` crea una nueva ventana de IU y el widget de texto coloca dentro de él, de modo que crear una nueva instancia de `NoteText` automáticamente abre una ventana de escritorio. A continuación, implementara las acciones `create` y `save`. La acción `create` abre una nueva ventana vacía, pero será almacenado en el servidor solo cuando se realiza una acción `save`, como se muestra en el siguiente código:

```
def create(self, event=None):
    NoteText(self.api, '')

def save(self, event=None):
    text = self.get('1.0', 'end')
    self.id = self.api.set(text,  self.id)
    tkMessageBox.showinfo('Info', 'Note %d Saved.' % self.id)
```

La acción `save` se puede realizar en tareas existentes o nuevas, pero no hay necesidad de preocuparse por eso aquí ya que esos casos ya están manejado por el método `set()` de la clase `NoteAPI`.

Finalmente, agregara el código que recupera y crea todas las ventanas notas cuando se inicia el programa, como se muestra en el siguiente código:

```
if __name__ == '__main__':
    srv, db = 'http://localhost:8069', 'v8dev'
    user, pwd = 'admin', 'admin'
    api = NoteAPI(srv, db, user, pwd)
    for note in api.get():
        x = NoteText(api, note['name'], note['id'])
        x.master.mainloop()
```

El último comando ejecuta `mainloop()` en la última ventana de Nota creada, para iniciar a esperar eventos de ventana.

Esta es una aplicación muy básica, pero el punto aquí es hacer un ejemplo de formas interesantes de aprovechar la API de Odoo RPC.

2.9.5 Introduciendo al cliente ERPpeek

`ERPpeek` es una herramienta versátil que se puede utilizar tanto como una aplicación interactiva de interfaz de línea de comandos (*Command-line Interface - CLI*) y como biblioteca de **Python**, con más API conveniente que la proporcionada por `xmlrpclib`. Está disponible desde el índice PyPi y se puede instalar con lo siguiente:

```
$ pip install -U erppeek
```

En un sistema Unix, si lo está instalando en todo el sistema, es posible que necesite anteponer `sudo` al comando.

La API ERPpeek

La biblioteca `erppeek` proporciona una interfaz de programación, envolviendo la biblioteca `xmlrpclib`, que es similar a la interfaz de programación que tiene para el código del lado del servidor. Su punto aquí es proporcionar una idea de lo que `ERPpeek` tiene para ofrecer, y no para proporcionar una explicación completa de todas sus características.

Puede comenzar reproduciendo sus primeros pasos con la biblioteca `xmlrpclib` usando `erppeek` como lo sigue:

```
>>> import  erppeek
>>> api = erppeek.Client('http://localhost:8069', 'v8dev', 'admin', 'admin')
>>> api.common.version()
>>> api.count('res.partner', [])
>>> api.search('res.partner', [('country_id', '=', 'be'), ('parent_id', '!=', False)])
>>> api.read('res.partner', [43], ['id',  'name', 'parent_id'])
```

Como puede ver, las llamadas a la API usan menos argumentos y son similares a las contrapartes del lado del servidor.

Pero `ERPpeek` no se detiene aquí, y también proporciona una representación para *Modelos*. Tiene las siguientes dos formas alternativas de obtener una instancia para un modelo, ya sea utilizando el método `model()` o accediendo a un atributo en caso de camello:

```
>>> m = api.model('res.partner')
>>> m = api.ResPartner
```

Ahora puede realizar acciones en ese modelo de la siguiente manera:

```
>>> m.count([('name', 'like', 'Packt%')])
1
>>> m.search([('name', 'like', 'Packt%')])
[76]
```

También proporciona representación de objetos del lado del cliente para registros como sigue:

```
>>> recs = m.browse([('name', 'like', 'Packt%')])
>>> recs <RecordList 'res.partner,[76]'>
>>> recs.name ['Packt']
```

Como puede ver, `ERPpeek` recorre un largo camino desde el simple `xmlrpclib`, y hace es posible escribir código que se pueda reutilizar del lado del servidor con poco o sin modificaciones.

El CLI ERPpeek

No solo se puede usar como una biblioteca de Python, sino que también es una CLI que se puede usar para realizar acciones administrativas en el servidor. Donde el comando *odoo shell* proporcionó una sesión interactiva local en el servidor host, `erppeek` proporciona una sesión interactiva remota en un cliente a través de la red.

Al abrir una línea de comando, puede echar un vistazo a las opciones disponibles, como se muestra a continuación:

```
$ erppeek --help
```

Vea una sesión de muestra de la siguiente manera:

```
$ erppeek --server='http://localhost:8069' -d v8dev -u admin

Usage (some commands): models(name)

# List models matching pattern model(name)
# Return a Model instance (...)
Password for 'admin':
Logged in as 'admin' v8dev
>>> model('res.users').count()
```

```
3 v8dev
>>> rec = model('res.partner').browse(43)
v8dev
>>> rec.name 'Michel Fletcher'
```

Como puede ver, se realizó una conexión con el servidor y la ejecución del contexto proporcionó una referencia al método `model()` para obtener el modelo instancias y realizar acciones sobre ellos.

La instancia `erppeek.Client` utilizada para la conexión también está disponible a través de la variable cliente. En particular, proporciona una alternativa a la cliente web para gestionar los siguientes módulos instalados:

- `client.modules()`: Esto puede buscar y enumerar módulos disponibles o instalados
- `client.install()`: Esto realiza la instalación del módulo
- `client.upgrade()`: Esto ordena que los módulos se actualicen
- `client.uninstall()`: Esto desinstala módulos

Entonces, `ERPpeek` también puede proporcionar un buen servicio como administración remota herramienta para servidores Odoo.

2.9.6 Resumen

El objetivo para este capítulo fue aprender cómo funciona la API externa y de lo que es capaz. Usted inicio a explorarlo usando un simple cliente `XML-RPC` en Python, pero la API externa se puede usar desde cualquier programación idioma. De hecho, los documentos oficiales proporcionan ejemplos de código para Java, PHP y Ruby.

Hay varias bibliotecas para manejar `XML-RPC` o `JSON-RPC`, algunas genéricos y algunos específicos para usar con Odoo. No intento señalar ninguno bibliotecas en particular, a excepción de `erppeek`, ya que no es solo un contenedor comprobado para el `XML-RPC` *Odoo/OpenERP* pero porque también es un herramienta invaluable para la gestión e inspección remota del servidor.

Hasta ahora, utiliza sus instancias de servidor Odoo para desarrollo y pruebas. Pero para tener un servidor de grado de producción, hay seguridad adicional y configuraciones de optimización que deben hacerse. En el siguiente capitulo, Usted se centrara en ellos.

2.10 Capítulo 10 - Despliegue

2.10.1 ¡Lista de Verificación para Despliegue – En Vivo!

En este capítulo, aprenderá como preparar su servidor Odoo para usarlo en un entorno de producción.

Existen varias estrategias y herramientas posibles que pueden usarse para el despliegue y gestión de un servidor de producción Odoo. Se le guiara a través de una de estas formas de hacerlo.

Esta es la lista de verificación para la configuración que seguirá:

- Instalar Odoo desde la fuente.
- Crear archivo de configuración de Odoo.
- Configuración de multiproceso de trabajos
- Configuración del servicio del sistema Odoo
- Configuración de un proxy inverso (reverse proxy) con soporte SSL

Comience.

2.10.2 Instalar Odoo

Odoo tiene paquetes disponibles para su instalación en sistemas Debian/Ubuntu. Así, se obtiene un servidor que se comienza a funcionar automáticamente cuando el sistema se inicia. Este proceso de instalación es bastante directo, y puede encontrar todo lo que necesita en http://nightly.odoo.com.

Aunque esta es una forma fácil y conveniente de instalar Odoo, aquí se prefiere ejecutar una versión controlada desde el código fuente, debido a que proporciona mejor control sobre lo que se esta desplegando.

Instalación desde el código fuente

Tarde o temprano, su servidor necesitará actualizaciones y parches. Una versión controlada por repositorio puede ser de gran ayuda cuando estos momentos lleguen.

Use `git` para obtener el código desde un repositorio, como hizo para instalar su entorno de desarrollo. Por ejemplo:

```
$ git clone https://github.com/odoo/odoo.git -b 8.0 --depth=1
```

Este comando obtiene el código fuente de la rama 8.0 desde GitHub dentro del subdirectorio `odoo/`. En el momento de escribir esto, la 8.0 es la rama predeterminada, por la tanto la opción `-b 8.0` es opcional. La opción `--depth=1` se usa para obtener una copia superficial del repositorio, sin toda la historia de la versión. Esto reduce el espacio en disco usado y hace que el proceso de clonación sea más rápido.

Puede valer la pena tener una configuración un poco más sofisticada, con un entorno de prueba junto al entorno de producción.

Con esto, podrá traerle la última versión de código fuente y probarlo en el entorno de prueba, sin perturbar el entorno de producción. Cuando la nueva versión este lista, puede desplegarla desde el entorno de pruebas a producción.

Considerara que el repositorio en `~/odoo-dev/odoo` será su entorno de pruebas. Fue clonado desde GitHub, por lo tanto un `git pull` dentro de este traerá y mezclara los últimos cambios. Pero el mismo es un repositorio, y podrá clonarlo desde su entorno de producción, como se muestra en el siguiente ejemplo:

```
$ mkdir ~/odoo-prd && cd ~/odoo-prd
$ git clone ~/odoo-dev/odoo ~/odoo-prd/odoo/
```

Esto creará el repositorio de producción en `~/odoo-prd/odoo` clonado desde en entorno de prueba `~/odoo-dev/odoo`. Se configurara para que monitorear este repositorio, por lo que un `git pull` dentro de producción traerá y mezclara las últimas versiones desde pruebas. Git es lo suficientemente inteligente para saber que esto es una clonación local y usar enlaces duros al repositorio padre para ahorrar espacio en disco, por lo tanto la opción `--depth` no es necesaria.

Cuando sea necesario resolver un problema en el entorno de producción, podrá verificar en el entorno de prueba la versión del código de producción, y depurar para diagnosticar y resolver el problema sin tocar el código de producción. Luego, el parche de la solución puede ser entregado al historial Git de prueba, y desplegado al repositorio de producción usando un comando `git pull`.

Nota Git sera una herramienta invaluables para gestionar las versiones de los despliegues de Odoo. Solo ha visto la superficie de lo que puede hacerse para gestionar las versiones de código. Si aun no conoce Git, vale la pena aprender más sobre este. Un buen sitio para comenzar es http://git-scm.com/doc.

2.10.3 Crear el archivo de configuración

Al agregar la opción `--save` cuando se inicia un servidor Odoo almacena la configuración usada en el archivo `~/.openerp_serverrc`. Podrá usar este archivo como punto de partida para su configuración del servidor, la cual será almacenada en `/etc/odoo`, como se muestra a continuación:

```
$ sudo mkdir /etc/odoo
$ sudo chown $(whoami) /etc/odoo
$ cp ~/.openerp_serverrc /etc/odoo/openerp-server.conf
```

Este tendrá los parámetros de configuración para ser usados en su instancia del servidor. Los siguientes son los parámetros para que el servidor trabaje correctamente:

- `addons_path`: Es una lista separada por coma de las rutas de directorios donde se buscarán los módulos, usando los directorios de izquierda a derecha. Esto significa que los directorios más a la izquierda tienen mayor prioridad.
- `xmlrpc_port`: Es el número de puerto en el cual escuchara el servidor. De forma predeterminada es el puerto `8069`.
- `log_level`: Este es la cantidad de información en el registro. De forma predeterminada es el nivel "info", pero al usar el nivel "debug_rpc", más descriptivo, agrega información importante para el monitoreo del desempaño del servidor.

Las configuraciones siguientes también son importantes para una instancia de producción:

- `admin_passwd`: Es la contraseña maestra para acceder a las funciones de gestión de base de datos del cliente web. Es importante fijarlo con una contraseña segura o con un valor vacío para desactivar la función.
- `dbfilter`: Es una expresión regular interpretada por Python para filtrar la lista de base de datos. Para que no sea requerido que el usuario seleccione una base de datos, debe fijarse con `^dbname$`, por ejemplo, `dbfilter = ^v8dev$`.
- `logrotate = True`: Divide el registro en archivos diarios y mantendrá solo un historias de registro mensual.
- `data_dir`: Es la ruta donde son almacenados los archivos adjuntos. Recuerde tener respaldo de estos.
- `withput_demo = True`: Se fija en los entornos de producción para que las bases de datos nuevas no tengan datos de demostración.

Cuando se usa un proxy inverso (reverse proxy), se deben considerar las siguientes configuraciones:

- `proxy_mode = True`: Es importante fijarlo cuando se usa un proxy inverso.
- `xmlrpc-interface`: Este fija las direcciones que serán escuchadas. De forma predeterminada escucha todo 0.0.0.0, pero cuando se usa un proxy inverso, puede configurarse a 127.0.0.1 para responder solo a solicitudes locales.

Se espera que una instancia de producción gestione una carga de trabajo significativa. De forma predeterminada, el servidor ejecuta un proceso y es capaz de gestionar solo una solicitud al mismo tiempo. De todas maneras, el modo multiproceso esta disponible para que puedan gestionarse solicitudes concurrentes.

La opción `workers=N` fija el número de procesos de trabajo que serán usados. Como guía puede intentar fijarlo a `1+2*P` donde P es el número de procesos. Es necesario afinar la mejor configuración para cada caso, debido a que depende de la carga del servidor y que otros servicios son ejecutados en el servidor (como PostgreSQL).

Podrá verificar el efecto de las configuraciones ejecutando el servidor con la opción `-c` o `--config` como se muestra a continuación:

```
$ ./odoo.py -c /etc/odoo/openerp-server.conf
```

2.10.4 Configurar como un servicio del sistema

Ahora, quiere configurar Odoo como un servicio del sistema y que sea ejecutado automáticamente cuando el sistema sea iniciado.

El código fuente de Odoo incluye un script de inicio, usado para las distribuciones Debian. Podrá usarlo como su script de inicio con algunas modificaciones menores, como se muestra a continuación:

```
$ sudo cp ~/odoo-prd/odoo/debian/init /etc/init.d/odoo
$ sudo chmo +x /etc/init.d/odoo
```

En este momento, quizás quiera verificar el contenido del script de inicio. Los parámetros claves son a variables al inicio del archivo. A continuación se muestra un ejemplo:

```
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin
DAEMON=/usr/bin/openerp-server
NAME=odoo
DESC=odoo
CONFIG=/etc/odoo/openerp-server.conf
LOGFILE=/var/log/odoo/odoo-server.log
PIDFILE=/var/run/${NAME}.pid
USER=odoo
```

La variable `USER` es el usuario del sistema bajo el cual se ejecutara el servidor, y probablemente quiera cambiarlo. Las otras variables deberían ser las correctas y preparare el resto de la configuración teniendo en mente estos valores predeterminados. `DAEMON` es la ruta a el ejecutable del servidor, `CONFIG` es el archivo de configuración que será usado, y `LOGFILE` es la ubicación del archivo de registro.

Los ejecutables en `DAEMON` pueden ser un enlace simbólico a su ubicación actual de Odoo, como se muestra a continuación:

```
$ sudo ln -s ~/odoo-prd/odoo/odoo.py /usr/bin/openerp-server
$ sudo chown $(whoami) /usr/bin/openerp-server
```

Luego debe crear el directorio `LOGFILE` como sigue:

```
$ sudo mkdir /var/log/odoo
$ sudo chown $(whoami) /etc/odoo
```

Ahora debería poder iniciar y parar el servicio de Odoo:

```
$ sudo /etc/init.d/odoo start
Starting odoo: ok
```

Debería ser capaces de obtener una respuesta del servidor sin ningún error en la archivo de registro, como se muestra a continuación:

```
$ curl http://localhost:8069
<html><head><script>window.location = '/web' + location.hash;</script> </head></html>
```

Muestre el archivo de registro de Odoo, ejecutando el siguiente comando:

```
$ less /var/log/odoo/odoo-server.log
```

La parada del servicio se hace de forma similar:

```
$ sudo /etc/init.d/odoo stop
Stopping odoo: ok
```

Truco: Ubuntu proporciona el comando más fácil de recordar para gestionar los servicios, si lo prefiere puede usar `sudo service odoo start` y `sudo service odoo stop`.

Ahora solo necesita que el servicio se ejecute automáticamente cuando se inicia el sistema:

```
$ sudo update-rc.d odoo defaults
```

Luego de esto, al reiniciar el servidor, el servicio de Odoo debería comenzar a ejecutarse automáticamente son errores. Es un buen momento para verificar que todo este funcionando como se espera.

2.10.5 Usar un proxy inverso

Mientras que Odoo puede entregar páginas web por si mismo, es recomendable usar un proxy inverso delante de Odoo. Un proxy inverso actúa como un intermediario que gestiona el tráfico entre los clientes que envían solicitudes y el servidor Odoo que responde a esas solicitudes. Usar un proxy inverso tiene múltiples beneficios.

De cara a la seguridad, puede hacer lo siguiente:

- Gestionar (y reforzar) los protocolos HTTPS para cifrar el tráfico.
- Esconder las características internas de la red.
- Actuar como un "aplicación firewall" limitando el número de URLs aceptados para su procesamiento.

Y del lado del desempeño, puede proveer mejoras significativas:

- Contenido estático cache, por lo tanto reduce la carga en los servidores Odoo.
- Comprime el contenido para acelerar el tiempo de carga.
- Balancea la carga distribuyendo la entre varios servidores.

Apache es una opción popular que se usa como proxy inverso. Nginx es una alternativa reciente con buenos argumentos técnicos. Aquí usara `nginx` como proxy inverso y mostrará como puede usarse para ejecutar las funciones mencionadas anteriormente.

Configurar nginx como proxy inverso

Primero, debe instalar `nginx`. Querrá que escuche en los puertos HTTP predeterminados, así que debe asegurarse que no estén siendo usados por otro servicio. Ejecutar el siguiente comando debe arrojar un error, como se muestra a continuación:

```
$ curl http://localhost
curl:   (7) Failed to connect to localhost port 80
```

De lo contrario, deberá deshabilitar o eliminar ese servicio para permitir que `nginx` use esos puertos. Por ejemplo, para parar un servidor Apache existente, deberá hacer lo siguiente:

```
$ sudo /etc/init.d/apache2 stop
```

Ahora podrá instalar `nginx`, lo cual es realizado de la forma esperada:

```
$ sudo apt-get install nginx
```

Para conformar que este funcionando correctamente, debería ver una página que diga **"Welcome to nginx"** cuando se ingrese la dirección del servidor en la navegador o usando `curl http://localhost`

Los archivos de configuración de `nginx` siguen el mismo enfoque que los de Apache: son almacenados en `/etc/nginx/available-sites/` y se activan agregando un enlace simbólico en `/etc/nginx/enabled-sites/`. Debería deshabilitar la configuración predeterminada que provee la instalación de `nginx`, como se muestra a continuación:

```
$ sudo rm /etc/nginx/sites-enabled/default
$ sudo touch /etc/nginx/sites-available/odoo
$ sudo ln -s /etc/nginx/sites-available/odoo /etc/nginx/sites-enabled/odoo
```

Usando un editor, como `nano` o `vi`, edite sus archivo de configuración `nginx` como sigue:

```
$ sudo nano /etc/nginx/sites-available/odoo
```

Primero agregue los `upstreams`, los servidores traseros hacia los cuales `nginx` redireccionará el tráfico, en su caso el servidor Odoo, el cual escucha en el puerto `8069`, como se muestra a continuación:

```
upstream backend-odoo {
    server 127.0.0.1:8069;
}

server {
    location / {
        proxy_pass http://backend-odoo;
    }
}
```

Para probar que la configuración es correcta, use lo siguiente:

```
$ sudo nginx -t
```

En caso que se encuentren errores, verifique que el archivo de configuración esta bien escrito. Además, un problema común es que el HTTP este tomado de forma predeterminada por otro servicio, como Apache o la página web predeterminada de `nginx`. Realice una doble revisión de las instrucciones dadas anteriormente para asegurarse que este no sea el caso, luego reinicio `nginx`. Luego de esto, podrá hacer que `nginx` cargue la nueva configuración:

```
$ sudo /etc/init.d/nginx reload
```

Ahora podrá verificar que `nginx` este redirigiendo el tráfico al servidor de Odoo, como se muestra a continuación:

```
$ curl http://localhost
<html><head><script>window.location = '/web' + location.hash;</script> </head></html>
```

2.10.6 Reforzar el HTTPS

Ahora, debería instalar un certificado para poder usar *SSL*. Para crear un certificado auto-firmado, siga los pasos a continuación:

Crear y acceder al directorio `ssl`, ejecutando el siguiente comando:

```
$ sudo mkdir /etc/nginx/ssl && cd /etc/nginx/ssl
```

Genere certificado SSL, ejecutando el siguiente comando:

```
$ sudo openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem - days 365 -nodes
```

hace a los archivos de solo lectura, ejecutando el siguiente comando:

```
$ sudo chmod a-wx *
```

acceso solamente al grupo `www-data`, ejecutando el siguiente comando:

```
$ sudo chown www-data:root *
```

Esto crea un directorio `ssl/` dentro del directorio `/etc/nginx/` y un certificado auto-firmado sin contraseña. Cuando se ejecute el comando `openssl`, se solicitara más información, y se generaran un certificado y archivos llave. Finalmente, estos archivos serán propiedad del usuario `www-data`, usado para ejecutar el servidor web.

Nota: Usar un certificado auto-firmado puede plantear algunos riesgos de seguridad, como ataques **"man-in-the-middle"**, y pueden no ser permitidos por algunos navegadores. Para una solución más robusta, debe usar un certificado firmado por una autoridad de certificación reconocida. Esto es particularmente importante si se esta ejecutando un sitio web comercial o de *e-commerce*.

Ahora que tiene un certificado SSL, podrá configurar `nginx` para usarlo.

Para reforzar HTTPS, redireccionara todo el tráfico HTTP. Reemplace la directiva `server` que defina anteriormente con lo siguiente:

```
server {
    listen 80;
    add_header Strict-Transport-Security max-age=2592000;
    rewrite ^/.*$ https://$host$request_uri? permanent;
}
```

Si recargue la configuración de `nginx` y acceda al servidor con el navegador web, vera que la dirección `http://` se convierte en `https://`.

Pero no devolverá ningún contenido antes que configura el servicio HTTPS apropiadamente, agregando la siguiente configuración a `server`:

```
server {
    listen 443 default;
    # ssl settings
    ssl on;
    ssl_certificate /etc/nginx/ssl/cert.pem;
    ssl_certificate_key /etc/nginx/ssl/key.pem;
    keepalive_timeout 60;
    # proxy header and settings
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forward-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_redirect off;

    location / {
        proxy_pass http://backend-odoo;
    }
}
```

Esto escuchara al puerto HTTPS y usará los archivos del certificado `/etc/nginx/ssl/` para cifrar el tráfico. También agregue alguna información al encabezado de solicitud para hacer que el servicio de Odoo sepa que esta pasando a través de un proxy. Por razones de seguridad, es importante para Odoo asegurarse que el parámetro `proxy_mode` este fijado a `True`. Al final, la directiva `location` define que todas las solicitudes sean pasadas al `upstream` "backend-oddo".

Recargue la configuración, y debería poder tener su servicio Odoo trabajando a través de HTTPS, como se muestra a continuación:

```
$ sudo nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
$ sudo service nginx reload *
Reloading nginx configuration nginx ...done.
$ curl -k https://localhost
<html><head><script>window.location = '/web' + location.hash;</script></head></html>
```

La última salida confirma que el cliente Odoo esta siendo servido sobre HTTPS.

2.10.7 Optimización de Nginx

Es hora para algunas mejoras en las configuraciones de `nginx`. Estas son recomendadas para habilitar el búfer de respuesta y compresión de datos que debería mejorar la velocidad del sitio web. También fije una localización específica para los registros.

Las siguientes configuraciones deberían ser agregadas dentro de `server` que escucha en el puerto `443`, por ejemplo, justo después de las definiciones del proxy:

```
# odoo log files access_log /var/log/nginx/odoo-access.log;
error_log /var/log/nginx/odoo-error.log;
# increase proxy buffer size
proxy_buffers 16 64k;
proxy_buffer_size 128k;
# force timeouts if the backend dies
proxy_next_upstream error timeout invalid_header http_500 http_502 http_503;
# enable data compression
gzip on;
gzip_min_length 1100;
gzip_buffers 4 32k;
gzip_types text/plain application/x-javascript text/xml text/css;
gzip_vary on;
```

También podrá activar el caché de contenido para respuestas más rápidas para los tipos de solicitudes mencionados

en el código anterior y para impedir su carga en el servidor Odoo. Después de la sección `location /`, agregue una segunda sección `location`:

```
location ~* /web/static/ {
    # cache static data
    proxy_cache_valid 200 60m;
    proxy_buffering on;
    expires 864000;
    proxy_pass http://backend-odoo;
}
```

Con esto, se hace caché de los datos estáticos por 60 minutos. Las solicitudes siguientes de esas solicitudes en este intervalo de tiempo serán respondidas directamente por `nginx` desde el caché.

2.10.8 Long polling

"Long polling" es usada para soportar la aplicación de mensajería instantánea, y cuando se usan trabajos multi-proceso, esta es gestionada en un puerto separado, el cual de forma predeterminada es el puerto `8072`.

Para su proxy inverso, esto significa que las solicitudes "longpolling" deberían ser pasadas por este puerto. Para soportar esto, necesita agregar un nuevo `upstream` a su configuración `nginx`, como se muestra en el siguiente código:

```
upstream backend-odoo-im { server 127.0.0.1:8072; }
```

Luego, debería agregar otra `location` al `server` que gestiona las solicitudes HTTPS, como se muestra a continuación:

```
location /longpolling { proxy_pass http://backend-odoo-im; }
```

Con estas configuraciones, `nginx` debería pasar estas solicitudes al puerto apropiado del servidor Odoo.

2.10.9 Actualización del servidor y módulos

Una vez que el servidor Odoo este listo y ejecutándose, llegara el momento en que necesite instalar actualizaciones. Lo cual involucra dos pasos: primero, obtener las nuevas versiones del código fuente (servidor o módulos), y segundo, instalarlas.

Si ha seguido el enfoque descrito en la sección *Instalación desde el código fuente*, podrá buscar y probar las nuevas versiones dentro del repositorio de preparación. Es altamente recomendable hacer una copia de la base de datos de producción y probar la actualización en ella. Si `v8dev` es su base de datos de producción, esto podría ser realizado con los siguientes comandos:

```
$ dropdb v8test ; createdb v8test
$ pg_dump v8dev | psqlpsql -d v8test
$ cd ~/odoo-dev/odoo/
$ ./odoo.py -d v8test --xmlrpc-port=8080 -c /etc/odoo/openerp-server.conf -u all
```

Si todo resulta bien, debería ser seguro realizar la actualización en el servicio en producción. Recuerde colocar una nota de la versión actual de referencia Git, con el fin de poder regresar, revisando esta versión otra vez. Hacer un respaldo de la base de datos antes de realizar la actualización es también recomendable.

Luego de esto, podrá hacer un `git pull` de las nuevas versiones al repositorio de producción usando Git y completando la actualización, como se muestra aquí:

```
$ cd ~/odoo-prd/odoo/
$ git pull
$ ./odoo.py -c /etc/odoo/openerp-server.conf --stop-after-init -d v8dev -u all
$ sudo /etc/init.d/odoo restart
```

2.10.10 Resumen

En este capítulo, aprendió sobre los pasos adicionales para configurar y ejecutar Odoo en un servidor de producción basado en *Debian*. Fueron vistas las configuraciones más importantes del archivo de configuración, y aprendió como aprovechar el modo multiproceso.

También aprendió como usar `nginx` como un proxy inverso frente a su servidor Odoo, para mejorar la seguridad y la escalabilidad.

Ojala que esto cubra lo esencial de lo que es necesario para ejecutar un servidor Odoo y proveer un servicio estable y seguro a sus usuarios.

CAPÍTULO 3

Acerca de esta iniciativa

Esta es una iniciativa de un grupo de desarrolladoras y desarrolladores, entusiastas del Software Libre desde la República Bolivariana de Venezuela, para hacer más accesibles los conocimientos sobre la tecnología Odoo.

3.1 Comunidades

Figura 3.1: Comunidad de Desarrollo del ERP Nacional - Bachaco-VE - Venezuela

3.2 Empresas

Figura 3.2: Packt Publishing - Reino Unido

3.3 Colaboradores

A continuación una lista de los colaboradores que han hecho posible esta es una iniciativa para hacer más accesibles los conocimientos sobre Odoo.

Truco: Para obtener una lista detallada (quizás más actualizada) de todos los colaboradores visite la siguiente dirección: https://github.com/Covantec/formacion-bachaco/graphs/contributors

Figura 3.3: Odoo S.A. - Bélgica

Figura 3.4: Covantec R.L. - Venezuela

Figura 3.5: Leonardo J. Caballero G.
Mantenedor, Traductor y Redactor

Figura 3.6: Victor Inojosa
Traductor y Redactor

Figura 3.7: Jhuliana Delgado
Traductor y Redactor

Figura 3.8: Francisco Palm
Traductor y Redactor

Figura 3.9: Francisco Vielma
Traductor y Redactor

Figura 3.10: Germana Oliveira Blazetic
Traductor y Redactor

Figura 3.11: Gustavo de Oliveira
Traductor y Redactor

Figura 3.12: Jorge Escalona
Redactor

Figura 3.13: Carlos Gustavo Nuñez.
Redactor

Figura 3.14: Samy Pessé
Redactor