

# Tryton: Capacitación técnica

Una introducción al desarrollo en Tryton

**Autores:** Mario Puntin, Adrián Bernardi <[contacto@silix.com.ar](mailto:contacto@silix.com.ar)>  
**Versión:** 4.4  
**Fecha:** Octubre de 2017

# Temario

## 1. Instalación

- 1.1 Características generales de Tryton
- 1.2 Herramientas y entornos de ejecución
- 1.3 Instalación e inicialización del servidor

## 2. Un módulo básico

- 2.1 Crear un modelo
- 2.2 Agregar campos a un modelo
- 2.3 Los campos del modelo: su representación gráfica y en base de datos
- 2.4 La presentación gráfica de los datos
- 2.5 Responder a las acciones del usuario: *on\_change*

## 3. Características avanzadas

- 3.1 Workflows
- 3.2 Mejorando las vistas
- 3.3 Los campos *function*
- 3.4 Wizards (asistentes)
- 3.5 Cómo extender objetos preexistentes

## 4. Temas adicionales

- 4.1 Reportes
- 4.2 Permisos y reglas de acceso
- 4.3 *proteus*

## 5. Referencias y fuentes

# 1. Instalación

## 1.1. Características generales de Tryton

Tryton es una plataforma para aplicaciones de propósito general. Puede utilizarse para desarrollar sistemas con diferentes objetivos: ERP para empresas u organizaciones, software para establecimientos de salud (como GNU Health) o para universidades, tiendas virtuales para ventas on-line, etc.

### **Características**

. **Escrito en Python:** Para ejecutar Tryton (versión 4.4) se requiere Python 2.7 o superior.

. **Modular**

. **Arquitectura de tres capas**

- Cliente de escritorio en GTK (tryton) y cliente web en javascript (sao)
- Servidor (trytond)
- Gestor de base de datos (DBMS): PostgreSQL (también soporta MySQL)

## 1.2. Herramientas y entornos de ejecución

Es recomendable utilizar algunas herramientas que facilitan la instalación y gestión de Tryton

### **pip**

Herramienta para instalar y gestionar paquetes de Python

### **virtualenv**

Herramienta para crear entornos controlados de Python. Permite tener diferentes versiones de algunas librerías y controlar las dependencias instaladas para cada entorno

### **virtualenvwrapper**

Conjunto de extensiones hechas sobre *virtualenv* para simplificar su uso.

### **Sistemas de Control de Versiones**

Hay que mencionar además la gran utilidad que tienen sistemas como **Mercurial** o **Git**, que permiten llevar registro de las distintas modificaciones hechas al código.

### **Instalación de pip y virtualenvwrapper**

En primer lugar podemos instalar el paquete *pip* que trae el sistema operativo. Luego podemos usar *pip* para instalar *virtualenvwrapper*.

```
# apt-get install python-pip
$ pip install --user virtualenvwrapper
```

Si se quiere utilizar Python 3 en un sistema que por defecto trae Python 2, los comandos a ejecutar serían:

```
# apt-get install python3-pip
$ pip3 install --user virtualenvwrapper
```

## Uso de pip

### La opción --user

La opción **--user** de *pip* hace que instale dentro de la carpeta *.local/* del *home* del usuario. De esa manera no es necesario utilizar carpetas del sistema o recurrir al superusuario para instalar mediante *pip*

## El repositorio de pip: Python Package Index

El comando *pip* funciona por medio de un repositorio público localizado en <https://pypi.python.org> cuyo propósito es ser un catálogo exhaustivo de todos los paquetes de Python escritos en código abierto

## Configuración de virtualenvwrapper

Hay que indicarle al sistema cómo trabajar con *virtualenvwrapper*. Podemos crear una carpeta llamada *.virtualenvs/*, en el *home*, y usarla para contener los entornos creados.

Agregar en el archivo */home/usuario/.bashrc* las siguientes líneas:

```
export WORKON_HOME=$HOME/.virtualenvs
source $HOME/.local/bin/virtualenvwrapper.sh
```

De esta manera, en el *home* tendremos dos carpetas:

- ***\$HOME/.local/***:  
todo lo que se instale con *pip* con la opción *--user*
- ***\$HOME/.virtualenvs/***:  
los entornos virtuales creados con *virtualenv[wrapper]*

## Incorporar .local/bin al PATH

En algunas circunstancias podría ocurrir que los comandos de *virtualenv* o *virtualenvwrapper* no se encuentren disponibles para ser ejecutados porque la carpeta *.local/bin* del *home* no está incluida en la ruta *PATH* de los ejecutables. En esos casos puede ser necesario editar el archivo *~/.profile* y agregar la línea:

```
PATH="$HOME/.local/bin:$PATH"
```

Luego, para que tome esta nueva configuración en línea de comandos, ejecutar:

```
$ source ~/.profile
```

## Creación del entorno

Se crea el entorno virtual, al que llamamos, por ejemplo, *tryton-4.4*, pero podría tener cualquier otro nombre:

```
$ mkvirtualenv tryton-4.4
(tryton-4.4)$ pip install pycopg2==2.7.3
```

Al crear el entorno virtual el comando *mkvirtualenv* también activa este entorno: el prompt de la consola indica **(tryton-4.4)\$**, es decir que antepone el nombre del entorno en el que nos encontramos.

En este punto ya se puede empezar a instalar paquetes por medio de *pip*.

Hay que tener en cuenta que *pip* permite indicar cuál es la versión del paquete que se desea instalar. Basta con escribir dos veces el signo de igual (==) y a continuación la versión, todo junto, sin espacios intermedios.

Si no se indicara versión y solamente se escribiera el nombre del paquete, entonces se entiende que debe instalarse la última versión. Hay que tener presente esto, porque no siempre queremos instalar la última versión de un paquete.

## ***pip dentro de un entorno virtual***

Al ejecutar *pip dentro* del entorno no hace falta indicarle *--user*, porque directamente instala dentro del entorno virtual, que se encuentra en *\$HOME/.virtualenvs/*

### ***Dependencias de Tryton***

Es importante chequear cuáles son las dependencias para ejecutar Tryton en determinadas condiciones. Por ejemplo, para ejecutarlo en combinación con el gestor PostgreSQL es necesario instalar el paquete *psycopg2* (se puede utilizar *pip*)

Para una referencia completa, dentro de la carpeta del servidor Tryton leer las indicaciones del archivo *INSTALL*

Para comenzar a trabajar en un entorno de python y dejar de trabajar en él utilizamos los comandos *workon* y *deactivate*

```
$ workon tryton-4.4
(tryton-4.4)$ ...
(tryton-4.4)$ deactivate
$
```

El paquete *virtualenvwrapper* ofrece varios comandos útiles, uno de los cuales es *cdsitepackages*, que permite posicionarse fácilmente en la carpeta donde se instalan los paquetes y módulos de Tryton:

```
(tryton-4.4)$ cdsitepackages
(tryton-4.4)~/virtualenvs/tryton-4.4/lib/python3.5/site-packages$
```

El comando *pip list* permite ver los paquetes instalados. También se puede utilizar *pip freeze*.

```
(tryton-4.4)$ pip list
Genshi (0.7)
lxml (4.0.0)
pip (9.0.1)
polib (1.0.8)
psycopg2 (2.7.3.1)
python-dateutil (2.6.1)
python-sql (0.9)
relatorio (0.7.0)
setuptools (36.5.0)
six (1.11.0)
[...]
```

Finalmente, si quisiéramos también podríamos eliminar todo el entorno junto con los paquetes instalados:

```
$ rmvirtualenv tryton-4.4
```

El efecto de este comando sería borrar la subcarpeta *.virtualenvs/tryton-4.4*, con todo su contenido.

## **'pyenv' y las versiones de Python**

Es posible que el sistema donde se intenta instalar Tryton tenga alguna versión antigua del lenguaje Python. Por eso es importante verificar en la página oficial, o en la documentación de los paquetes disponibles para descarga, cuál es la versión mínima de Python necesaria para ejecutar el servidor de Tryton o el programa cliente.

Para los casos en que el sistema no tenga disponible una versión adecuada, existe una herramienta llamada *pyenv*, que se puede utilizar en combinación con *virtualenv*.

*pyenv* permite instalar distintas versiones de Python y activar cualquiera de ellas en distintas circunstancias, según se requiera.

En primer lugar se necesita instalar *pyenv* siguiendo las instrucciones de su página oficial: <https://github.com/pyenv/pyenv>. Una vez disponible en nuestro sistema, podemos solicitarle que instale una determinada versión de Python, por ejemplo:

```
$ pyenv install 3.6.0
```

A continuación es posible activar esta versión de Python para todo el sistema, con el comando

```
$ pyenv global 3.6.0
```

Finalmente, si se quiere utilizar esta versión de Python para crear el entorno virtual, hay que recurrir a *virtualenv*. Por ejemplo, dentro de la carpeta *~/.virtualenvs/*, que es donde suelen localizarse los entornos virtuales, se puede crear el entorno *tryton-4.4*

```
$ virtualenv ~/.virtualenvs/tryton-4.4
```

Si la versión de Python deseada fue activada correctamente con *pyenv*, el comando *virtualenv* creará el entorno virtual utilizando esa versión de Python.

Cuando se quiera activar el entorno virtual, simplemente hay que ejecutar el comando *activate*:

```
$ source ~/.virtualenvs/tryton-4.4/bin/activate
```

Finalmente, si se usó el comando *pyenv global* para activar una nueva versión de Python, pero luego se desea volver a dejar la versión original del sistema, hay que ejecutar:

```
$ pyenv global system
```

### **Entender pyenv**

Se recomienda estudiar la documentación para comprender las distintas posibilidades y herramientas de configuración que ofrece *pyenv*.

## 1.3. Instalación e inicialización del servidor

### **Instalación de módulos**

Los módulos oficiales de Tryton, como cualquier paquete o módulo Python, pueden instalarse con las diversas herramientas ofrecidas por ese lenguaje. Podrían incluso también descargarse y descomprimirse en algún lugar del sistema de archivos, y ejecutarse sin recurrir a herramientas de instalación.

En nuestro caso veremos la instalación por medio de *pip*, a fin de aprovechar las ventajas y la comodidad que ofrece.

Siempre dentro del entorno creado, para instalar la última versión del servidor de Tryton (*trytond*) bastaría con ejecutar.

```
(tryton-4.4)$ pip install trytond
```

Si se pretende instalar una versión específica, o si se prefiere indicar la versión de manera explícita, el comando tendría la forma siguiente:

```
(tryton-4.4)$ pip install trytond==4.4.3
```

Además del servidor, pueden instalarse otros módulos de Tryton, por ejemplo:

```
(tryton-4.4)$ pip install trytond_account
```

Esto instala el módulo *account* (cuentas contables), junto con sus dependencias: *company*, *currency*, *country*, *party*, dentro de la carpeta *modules* del servidor *trytond*, en el entorno virtual. Es decir que en este caso instalamos cinco módulos.

### **Cuidado con las versiones**

Las distintas versiones de Tryton se manejan por el tradicional esquema de tres números separados por un punto: *mayor.menor.parche*. Por ejemplo, 4.4.1. Para toda instalación de Tryton, debemos asegurarnos de que los dos primeros números (*mayor.menor*) coincidan tanto en el servidor, como en el resto de los módulos y los clientes web o de Escritorio. No se debe mezclar una versión 4.4.3 con una 4.2.1. Se puede ignorar el último de los tres dígitos, pero los dos primeros deben coincidir.

### **El archivo *trytond.conf***

Algunas características de funcionamiento de Tryton pueden configurarse por medio de un archivo denominado *trytond.conf*, que tiene un formato semejante a los conocidos archivos *ini*. Consiste de secciones definidas entre corchetes [], a las que siguen entradas definidas por nombre = valor.

```
[jsonrpc]
listen=*:8000
data=/var/www/localhost/tryton

[database]
uri=postgres://usuario:password@localhost:5432
path=/var/lib/tryton/data # directorio donde se guardaran los adjuntos
```



```
[session]
timeout=3600
```

Tryton podría funcionar sin un archivo *trytond.conf*. Pero normalmente queremos tenerlo. Es posible ubicarlo en cualquier lugar del sistema de archivos, y no hay que olvidar indicarle al servidor dónde encontrarlo, con la opción **-c**

Por ejemplo, este archivo podría guardarse dentro de */etc/*, en el sistema de archivos de GNU/Linux. Por lo tanto, para invocarlo al ejecutar el servidor hay que indicar: **-c /etc/trytond.conf**

## Creación e inicialización de la base de datos

Para poder utilizar la base de datos será necesario tener un usuario. Este usuario será el que se configurará en *trytond.conf*. En nuestro caso, utilizamos PostgreSQL, y llamaremos *dbadmin* al usuario. Desde la línea de comandos, ejecutar:

```
$ createuser -s -P dbadmin
```

A continuación se debe crear la base de datos, llamada en este caso 'tryton44'

```
$ createdb -U dbadmin --encoding=UTF-8 tryton44
```

Es necesario editar el archivo *trytond.conf*, en la entrada *uri* de la sección *[database]*. Allí hay que indicar el usuario y contraseña que se ha definido.

```
uri=postgresql://dbadmin:miclave@localhost:5432
```

El siguiente paso es inicializar la base de datos 'tryton44'. Para esto es necesario tener activado el entorno virtual. El comando a utilizar es *trytond-admin*. El parámetro **-p** hará que al finalizar este proceso se pida una contraseña de usuario *admin*.

Hay que tener en claro que este usuario *admin* es el usuario administrador del sistema Tryton cuya base de datos estamos inicializando. Es distinto del usuario *dbadmin*, de PostgreSQL.

```
(tryton-4.4) $ trytond-admin -v -p -c /etc/trytond.conf -d tryton44 --all
```

Al terminar, estaremos en condiciones de ejecutar el servidor Tryton, para lo cual se utiliza el comando *trytond*. Con el parámetro **-c** se indicará la ubicación del archivo de configuración. Por su parte, **-v** (verbose) hará que el comando vaya generando información en pantalla, mientras que **-d** permitirá indicar la base de datos a utilizar.

```
(tryton-4.4) $ trytond -c /etc/trytond.conf -v -d tryton44
```

## 2. Un módulo básico

Un módulo mínimo de Tryton consiste de dos archivos:

**\_\_init\_\_.py**

Archivo estándar de todos los módulos de Python

**tryton.cfg**

Permite establecer cuáles son las dependencias del módulo y los archivos XML adicionales que deben ser utilizados

Ejemplo de *tryton.cfg*

```
[tryton]
version=4.4.1
depends:
    company
xml:
    account.xml
```

### 2.1. Crear un modelo

En Tryton un *modelo* es una clase normal de Python, que hereda de la clase *Model*. Pero para que el modelo persista en la base de datos hay que heredar de *ModelSQL*

Para nuestro ejemplo vamos a crear un archivo llamado *opportunity.py*, en el que ingresaremos el siguiente código:

En *opportunity.py*:

```
from trytond.model import ModelSQL

__all__ = ['Opportunity']

class Opportunity(ModelSQL):
    'Opportunity'
    __name__ = 'training.opportunity'
```

Para que Tryton tome nota de la existencia de este modelo es necesario registrarlo en el *Pool* de modelos

En *\_\_init\_\_.py*:

```
from trytond.pool import Pool
from .opportunity import *

def register():
    Pool.register(
        Opportunity,
        module='training', type_='model')
```

## 2.2. Agregar campos a un modelo

Volvemos a `opportunity.py`:

```
from trytond.model import ModelSQL, ModelView, fields

__all__ = ['Opportunity']

class Opportunity(ModelSQL, ModelView):
    'Opportunity'
    __name__ = 'training.opportunity'
    _rec_name = 'description'
    description = fields.Char('Description', required=True)
    start_date = fields.Date('Start Date', required=True)
    end_date = fields.Date('End Date')
    party = fields.Many2One('party.party', 'Party', required=True)
    comment = fields.Text('Comment')
```

Para definir los atributos (campos) de la clase se utiliza *fields*

Existen distintos tipos de campos, por ejemplo *Char*, *Integer*, *Boolean*, *Date*, entre otros.

Cada uno de ellos tiene atributos que establecen su comportamiento:

### **string**

Nombre o etiqueta para el campo

### **required**

Si se le asigna *True*, el campo es requerido

### **readonly**

Si se le asigna *True*, no se permite editar el campo desde la interfaz de usuario

### **domain**

Una lista que define un dominio o subconjunto de registros válidos

### **states**

**Es un diccionario de Python con al menos alguna de las siguientes claves:**

*required*, *readonly* e *invisible*

Nótese que las dos primeras coinciden con atributos ya mencionados.

Los valores posibles para estas claves son expresiones **pyson** que se evalúan con valores de cada registro. De esta manera es posible cambiar dinámicamente los atributos del campo. Por ejemplo, que determinado campo no sea de solo lectura (*readonly*) siempre, sino en determinada circunstancia o estado del registro.

## 2.3. Los campos del modelo: su representación gráfica y en base de datos

Al desarrollar una aplicación utilizando la plataforma Tryton tenemos la ventaja de que fácilmente podemos ver representados nuestros campos en una base de datos (capa de persistencia de datos) y también de manera gráfica (capa de presentación).

En cierta medida estamos hablando del *mapeo objeto-relacional* (conocido por su nombre en inglés, Object-Relational mapping, o sus siglas *ORM*).

Veamos algunos ejemplos:

---

### fields.Char:

En un módulo Tryton:

```
name = fields.Char('Nombre', required=True)
```

En la interfaz gráfica:

Nombre:

En la base de datos (SQL):

```
name VARCHAR NOT NULL
```

---

### fields.Date:

En un módulo Tryton:

```
fecha_factura = fields.Date('Fecha factura')
```

En la interfaz gráfica:

Fecha factura:

En la base de datos (SQL):

```
fecha_factura DATE
```

---

### fields.Many2One:

En un módulo Tryton:

```
party = fields.Many2One('party.party', 'Entidad', required=True)
```

En la interfaz gráfica:

Entidad:

En la base de datos (SQL):

```
party INTEGER NOT NULL  
FOREIGN KEY(party) REFERENCES party_party(id)
```

## 2.4. La presentación gráfica de los datos

Para poder usar la capa de presentación, el modelo debe heredar *ModelView*

```
class Opportunity(ModelSQL, ModelView):
```

La representación gráfica se determina por medio de archivos XML.

Hay que indicarle a Tryton de dónde tomar la definición XML. Y para ello hay que crear un archivo llamado *opportunity.xml*

En el archivo *tryton.cfg*:

```
xml :
    opportunity.xml
```

Una vista (*View*) es un objeto más de Tryton; se almacena en base de datos y guarda información que toma el servidor para construir la representación gráfica.

Hay dos tipos principales de vista:

- **Tree:** es una lista de registros
- **Form:** es una vista de formulario, que permite crear y editar un registro

### Vista Tree

En la definición se hace referencia al objeto View (*ir.ui.view*) que va a contener la vista.

En los campos de este objeto se define el modelo que está asociado a la vista (*training.opportunity*), el tipo (*tree*) y el nombre del archivo que contiene la definición: *opportunity\_list*

```
<record model="ir.ui.view" id="opportunity_view_list">
    <field name="model">training.opportunity</field>
    <field name="type">tree</field>
    <field name="name">opportunity_list</field>
</record>
```

El nombre de archivo al que se alude está alojado por defecto en una subcarpeta del módulo: *view/opportunity\_list.xml*

```
<tree>
    <field name="party" />
    <field name="description" />
    <field name="start_date" />
    <field name="end_date" />
</tree>
```

### Vista Form

También en este caso tenemos la referencia al objeto View (*ir.ui.view*).

La diferencia es que el tipo es *form*, y el nombre del archivo que contiene la definición es *opportunity\_form*

```
<record model="ir.ui.view" id="opportunity_view_form">
    <field name="model">training.opportunity</field>
```

```

    <field name="type">form</field>
    <field name="name">opportunity_form</field>
</record>

```

Definición de *view/opportunity\_form.xml*:

```

<form>
  <label name="party" />
  <field name="party" />
  <label name="description" />
  <field name="description" />
  <label name="start_date" />
  <field name="start_date" />
  <label name="end_date" />
  <field name="end_date" />
  <separator name="comment" colspan="4" />
  <field name="comment" colspan="4" />
</form>

```

## Armando las piezas

Definidas las vistas, sólo falta *armarlas* o *pegarlas*, para que actúen de manera coordinada.

Para ello se utiliza el objeto *ActionActWindow*, que define un tipo particular de acción, en este caso, abrir una ventana.

En *opportunity.xml*:

```

<record model="ir.action.act_window" id="act_opportunity_open">
  <field name="name">Opportunities</field>
  <field name="res_model">training.opportunity</field>
</record>

```

Y vinculadas a esta nueva acción, dos *acciones/vistas*

```

<record model="ir.action.act_window.view" id="act_opportunity_open_view1">
  <field name="sequence" eval="10" />
  <field name="view" ref="opportunity_view_list" />
  <field name="act_window" ref="act_opportunity_open" />
</record>
<record model="ir.action.act_window.view" id="act_opportunity_open_view2">
  <field name="sequence" eval="20" />
  <field name="view" ref="opportunity_view_form" />
  <field name="act_window" ref="act_opportunity_open" />
</record>

```

Como se puede ver, el campo *act\_window* enlaza la **acción** de abrir una ventana con las **vistas** a las que se alude por medio del campo *view*.

## Agregar entradas al menú

Por último, esta acción es disparada por el usuario por medio de una entrada del menú de Tryton, que se puede crear de la siguiente manera:

```
<menuitem name="Training" id="menu_training"/>

<menuitem parent="menu_training" action="act_opportunity_open"
  id="menu_opportunity_form"/>
```

## 2.5. Responder a las acciones del usuario: *on\_change*

Tryton provee un mecanismo para responder a acciones del usuario, de modo que si cambia algún valor otros campos resulten modificados.

Este mecanismo es el de *on\_change*

Se puede aplicar de dos maneras:

### **on\_change\_<nombre campo>**

Cuando se modifica un campo, se envía al servidor una lista de valores. En base a ellos, el servidor devuelve nuevos valores para los campos.

### **on\_change\_with\_<nombre campo>**

El valor de un campo se calcula cuando cualquier campo de una lista de campos es modificada. El cálculo lo realiza el servidor.

En ambos casos, la lista de campos que se envía al servidor se especifica por medio de un *decorador* de Python **@fields.depends**.

### **Ejemplo de on\_change**

```
@fields.depends('party')
def on_change_party(self):
    self.cuit = None
    if self.party:
        self.cuit = self.party.cuit
```

**Ejemplo de on\_change\_with:** *end\_date* toma el valor de *start\_date* pero con siete días más

```
@fields.depends('start_date')
def on_change_with_end_date(self):
    if self.start_date:
        return self.start_date + datetime.timedelta(days=7)
    return None
```

## 3. Características avanzadas

### 3.1. Workflows

Los *workflows* o flujos de trabajo permiten establecer estados para determinadas tareas, en este caso relacionadas con alguna clase de Tryton.

Para que una clase pueda manejar flujos de trabajo debe heredar la clase *Workflow*

```
class Opportunity(Workflow, ModelSQL, ModelView):
```

También debe tener un campo *state*.

```
state = fields.Selection([
    ('opportunity', 'Opportunity'),
    ('converted', 'Converted'),
    ('lost', 'Lost'),
], 'State', required=True, readonly=True, sort=False)
```

El flujo además requiere definir las posibles *transiciones* entre un estado y otro. Esto se hace en la función `__setup__()` por medio de *tuplas*:

```
@classmethod
def __setup__(cls):
    super(Opportunity, cls).__setup__()
    cls._transitions |= set((
        ('opportunity', 'converted'),
        ('opportunity', 'lost'),
    ))
```

Finalmente, cada transición debe estar relacionada con un método de clase:

```
@classmethod
@ModelView.button
@Workflow.transition('converted')
def convert(cls, opportunities):
    pool = Pool()
    Date = pool.get('ir.date')
    cls.write(opportunities, {
        'end_date': Date.today(),
    })
```

#### **Agregar botones en la vista**

La transición de un estado a otro es disparada normalmente por el usuario. Esto se puede hacer agregando un botón en la vista del formulario definido en XML:

```
<button name="convert" string="Convert" icon="tryton-go-next"/>
```

Primero, en `__setup__()` se declara la existencia del botón. Luego se *decora* el método con `@ModelView.button` para que el método pueda ser invocado por medio del botón.



```

@classmethod
def __setup__(cls):
    [...]
    cls._buttons.update({
        'convert': {
            'invisible': ~Eval('state').in_(['opportunity']),
        },
    })

@classmethod
@ModelView.button
@Workflow.transition('converted')
def convert(cls, opportunities):
    [...]

```

Los botones pueden estar activos o visibles de acuerdo con el estado actual del objeto, que define el flujo posible a seguir. Para eso se utiliza el atributo *invisible* de los botones.

Hay que tener presente que como el método está vinculado con la transición del *workflow*, al ejecutarse cambia de forma automática el estado de los objetos afectados por la transición. En otras palabras, no es necesario modificar manualmente el valor del campo *state*.

## 3.2. Mejorando las vistas

En el caso de los *workflows*, es importante mostrar el estado y los botones disponibles.

Es posible agrupar los elementos a mostrar, como una forma de controlar mejor la forma en que se muestran:

```

<group col="2" colspan="2" id="state">
    <label name="state"/>
    <field name="state"/>
</group>
<group col="2" colspan="2" id="buttons">
    <button name="lost" string="Lost" icon="tryton-cancel"/>
    <button name="convert" string="Convert" icon="tryton-go-next"/>
</group>

```

### Formularios dinámicos

Es frecuente que se quiera que campos o botones estén visibles o no, o sean de solo lectura u obligatorios, de acuerdo con determinadas condiciones variables. Estas condiciones se evalúan utilizando el lenguaje *Python*, que se verá a continuación.

Ya se habló del atributo *states*, que define un conjunto de valores posibles para **readonly**, **required**, **invisible**

```

start_date = fields.Date('Start Date', required=True, states={
    'readonly': Eval('state') != 'opportunity',
}, depends=['state'])
end_date = fields.Date('End Date', readonly=True, states={
    'required': Eval('state').in_(['converted', 'lost']),
}, depends=['state'])

```

## Pyson

Pyson es una forma de representar una declaración que puede ser evaluada, es decir que de ella siempre se obtiene un resultado.

Es necesario importar aquellos tipos de declaración que se van a utilizar.

```
from trytond.pyson import Eval, Equal, Not
```

*Eval* evalúa el valor de un campo y lo devuelve, en este caso para realizar una comparación:

```
'readonly': Eval('state') != 'opportunity'
```

Las declaraciones Pyson se pueden combinar y anidar, agregando como en este caso *Equal* y *Not*

```
'readonly': Not(Equal(Eval('state'), 'opportunity'))
```

En los dos casos citados, el campo al que se le aplique esta definición será de solo lectura (*'readonly': True*) cuando el valor del campo *state* sea distinto de *'opportunity'*. Caso contrario, *readonly* tomará el valor *falso*: (*'readonly': False*)

## 3.3. Los campos *function*

Un campo de tipo *function* es un campo que puede emular otro tipo cualquiera de campo, y es calculado de forma dinámica. Su valor **no** es almacenado en la base de datos.

```
duration = fields.Function(fields.Integer('Duration'), 'get_duration')

def get_duration(self, name=None):
    if not self.start_date or not self.end_date:
        return None
    return (self.end_date - self.start_date).days
```

El primer parámetro a definir es el tipo del campo (*Integer* en el ejemplo). Cualquier tipo de campo puede ser usado, siempre que se provea un mecanismo para calcularlo (un método *getter*, que en el ejemplo es *get\_duration*).

### ***El getter como on\_change\_with***

Hay que notar que el método *getter* que se define es semejante en su estructura al *on\_change\_with*: en virtud de ciertos valores o condiciones, se obtiene un valor para el campo.

Por este motivo, no es extraño encontrar que la misma función es usada con ese fin: en lugar de algo como *'get\_duration'*, se la escribe como *'on\_change\_with\_duration'*

## Setter y searcher

Cuando estos campos *function* operan en base a cálculos hechos sobre otros campos relacionados, es posible también modificar valores por medio de un método *setter*. Asimismo es posible definir un *searcher*, para que se puedan realizar búsquedas por el campo *function*

## ***Campos function que operan por lotes***

El cálculo del campo *function* como lo hemos visto hasta ahora se ejecuta una vez por cada registro que se muestra.

No obstante, es posible calcular los valores de manera masiva, por lotes. Para ello el método *getter* debe ser un método de clase y debe retornar un diccionario que mapee el valor devuelto con su respectivo *id* de registro.

```
description_length = fields.Function(fields.Integer('Description Length'),
    'get_description_length')

@classmethod
def get_description_length(cls, opportunities, name):
    cursor = Transaction.cursor()

    opportunity = cls.__table__()
    query = opportunity.select(
        opportunity.id, CharLength(opportunity.description))
    cursor.execute(*query)

    return dict(cursor.fetchall())
```

## **3.4. Wizards (asistentes)**

### ***Agregar acciones a un modelo***

En ocasiones se puede requerir alguna funcionalidad o acción de cierta complejidad, que requiere una intervención adicional del usuario. Para estas situaciones la solución es crear un *wizard* o asistente.

Un *wizard* se compone de dos elementos:

- Vistas o formularios que permiten al usuario ingresar ciertos valores o tomar una decisión.
- Una *máquina de estados* que permite determinar lo que se debe o se puede hacer.

### ***Vistas de wizard***

Las vistas de un wizard son *ModelView* como los que ya se han visto. En este caso, no se hace persistir la información en base de datos, por lo cual no es necesario heredar de *ModelSQL*

```
class ConvertOpportunitiesStart(ModelView):
    'Convert Opportunities'
    __name__ = 'training.opportunity.convert.start'
```

Por lo demás, es posible establecer valores por defecto o utilizar las funcionalidades de tipo *on\_change*, como con cualquier otro modelo.

### ***Una máquina de estados***

Las clases que constituyen el wizard heredan, precisamente, de *Wizard*, y en ellas se pueden definir diferentes estados

```

from trytond.wizard import Wizard, StateView, StateTransition, Button

[...]

class ConvertOpportunities(Wizard):
    'Convert Opportunities'
    __name__ = 'training.opportunity.convert'

    start = StateView('training.opportunity.convert.start',
        'training.opportunity_convert_start_view_form', [
            Button('Cancel', 'end', 'tryton-cancel'),
            Button('Convert', 'convert', 'tryton-ok', default=True),
        ])
    convert = StateTransition()

    def transition_convert(self):
        pool = Pool()
        Opportunity = pool.get('training.opportunity')
        opportunities = Opportunity.browse(Transaction().context['active_ids'])
        Opportunity.convert(opportunities)
        return 'end'

```

### El estado final

Por defecto, todo wizard tiene un estado final: *end*. No es necesario definirlo.

### Activación del wizard

Para activar el wizard es necesario definirlo en el archivo XML:

```

<record model="ir.action.wizard" id="act_convert_opportunities">
    <field name="name">Convert Opportunities</field>
    <field name="wiz_name">training.opportunity.convert</field>
    <field name="model">training.opportunity</field>
</record>
<record model="ir.action.keyword" id="act_convert_opportunities_keyword">
    <field name="keyword">form_action</field>
    <field name="model">training.opportunity,-1</field>
    <field name="action" ref="act_convert_opportunities"/>
</record>

```

En este caso, con *ir.action.keyword* se indica que el wizard puede dispararse desde el botón de acciones (*form\_action*) del modelo *training.opportunity*. Es posible seleccionar uno o más registros y lanzar el wizard vinculado con esos registros.

Sin embargo, también podría lanzarse el wizard asociándolo a un botón en un formulario. Para eso se crea el botón y en el método que tiene asociado se utiliza el decorador `@ModelView.button_action`.

```

@classmethod
@ModelView.button_action('training.act_convert_opportunities')
def convert_opportunities(cls, opportunities):
    pass

```

Finalmente, es necesario registrar el modelo (el wizard) en el archivo `__init__.py`, en este caso indicando su tipo.

```
def register():
    [...]
    Pool.register(
        ConvertOpportunities,
        module='training', type_='wizard')
```

## 3.5. Cómo extender objetos preexistentes

### La metacalse *PoolMeta*

Algunas veces lo que se desea no es crear modelos u objetos nuevos, sino simplemente agregar información.

Existe una clase llamada *PoolMeta*, que es una *metacalse*. Una *metacalse* es una clase que facilita la creación de otras clases. En este caso, estamos hablando de clases ya existentes en el Pool de modelos de Tryton.

```
from trytond.model import fields
from trytond.pool import PoolMeta

__all__ = ['Party']
__metaclass__ = PoolMeta

class Party:
    __name__ = 'party.party'
    opportunities = fields.One2Many('training.opportunity',
        'party', 'Opportunities')
```

Al especificar el atributo `__metaclass__ = PoolMeta` ya no hace falta indicar de qué clases hereda la clase *Party*, es decir, no hace falta que especifiquemos: `class Party(ModelSQL, ModelView)`

La clase *Party*, identificada por el atributo `__name__ = 'party.party'`, ya fue reconocida por Tryton, cuando se cargó, previamente, el módulo *party*.

Tryton se da cuenta de que se está trabajando con el mismo modelo y por lo tanto se produce todo el mecanismo de herencia, y finalmente se agrega el atributo deseado, *opportunities*.

### Extender las vistas

Para extender o modificar las vistas que ya existen hay que agregar un registro en un archivo XML.

En el registro se indica con el atributo *inherit* cuál es la vista que se va a modificar. En el valor de *ref* hay que indicar `<módulo>.<id_vista>`

```
<record model="ir.ui.view" id="party_view_form">
    <field name="model">party.party</field>
    <field name="inherit" ref="party.party_view_form"/>
    <field name="name">party_form</field>
</record>
```

La definición de aquello que se va a modificar se realiza por medio de *XPATH*, que permite identificar el elemento a modificar o la ubicación de referencia para agregar algo.

```
<xpath expr="/form/notebook/page[@id='accounting']"
  position="after">
  <page name="opportunities" col="1">
    <separator name="opportunities"/>
    <field name="opportunities"/>
  </page>
</xpath>
```

## 4. Temas adicionales

### 4.1. Reportes

Para crear reportes se utilizan archivos de tipo *.odt* (Open Document) a modo de plantilla. Se los puede generar con herramientas como LibreOffice, aunque también es posible tomar como base algún reporte ya existente.

Para procesar los reportes se utilizan las librerías de Python *relatorio* y *Genshi*.

Se debe definir una clase que herede *Report*. Para personalizar el reporte se puede *sobrecargar* o *extender* el método *get\_context()*, que se encuentra en *trytond/report/report.py*. De esa manera se pueden realizar cálculos adicionales para obtener datos que se utilizarán en el reporte.

```
from trytond.report import Report

class ReporteConsultas(Report):
    __name__ = 'servicio.reporte.consultas'

    @classmethod
    def get_context(cls, records, data):
        report_context = super(ReporteConsultas, cls).get_context(records, data)
        # Con 'super' se invoca el método 'get_context'
        # de la clase que se hereda (Report)
        # 'report_context' es entonces un diccionario
        # que trae ya por defecto un conjunto de valores
        # que pueden ser utilizados en el reporte

        # En este lugar se pueden realizar operaciones o invocar
        # otros métodos para obtener valores adicionales
        # que se utilizarán en el reporte que se va a generar

        # Finalmente, los nuevos valores calculados u obtenidos
        # se agregan en el diccionario 'report_context'
        report_context['consultas'] = consultas
        report_context['hoy'] = Date.today()
        return report_context
```

Además hay que registrar este nuevo objeto en el Pool de Tryton, en el archivo *\_\_init\_\_.py*. El módulo de este ejemplo se llama *servicio*

```
def register():
    Pool.register(
        ReporteConsultas,
        module='servicio', type_='report')
```

Por otra parte, para poder lanzar el reporte debemos crear un objeto de tipo *ActionReport*. El modelo asociado a este reporte se llama *servicio.consulta* y es un modelo que contiene, en nuestro ejemplo hipotético, un listado de consultas recibidas.

También hay que indicar desde dónde se lanzará esta acción: se define un *ActionKeyword* de tipo *form\_print*, con lo cual se asocia la acción al botón de *Imprimir informe* que se muestra en la barra de acciones de la interfaz gráfica.

```

<record model="ir.action.report" id="act_servicio_reporte_consultas">
  <field name="name">Reporte de consultas</field>
  <field name="model">servicio.consulta</field>
  <field name="report_name">servicio.reporte.consultas</field>
  <field name="report">servicio/report/report_consultas.odt</field>
</record>
<record model="ir.action.keyword" id="servicio_reporte_consultas_keyword">
  <field name="keyword">form_print</field>
  <field name="model">servicio.consulta,-1</field>
  <field name="action" ref="act_servicio_reporte_consultas"/>
</record>

```

En la definición del *ActionReport* se indica dónde se encuentra la plantilla (el archivo .odt), dentro de la carpeta del módulo.

Esta plantilla podría tener un formato como el siguiente:

Reporte de consultas				
Fecha informe: <FORMAT_DATE(HOY, PARTY.LANG)>				
Descripción	Referencia	Fecha	Duración	Solución
<FOR EACH="CONSULTA IN CONSULTAS">				
<CONSULTA.DESCRIPCION>	<CONSULTA.REFERENCIA IF CONSULTA.REFERENCIA ELSE ">	<FORMAT_DATE(CONSULTA.FECHA, PARTY.LANG)>	<CONSULTA.DURACION>	<CONSULTA.SOLUCION>
</FOR>				

Se utilizan los valores *ho*y y *consultas*, definidos en el diccionario *report\_context*.

*consultas* contiene una lista de elementos (las consultas) sobre los cuales se itera, para que se muestren una debajo de la otra. Esta iteración se realiza con el comando `<for each> ... </for>`

Esta incorporación de datos de forma dinámica se realiza por medio de funciones especiales en la plantilla. En el archivo .odt (LibreOffice) hay que ir a *Insertar > Campos > Otros* y luego en la pestaña *Funciones > Marcador de posición > Texto*

La edición de estos campos se realiza apretando Botón derecho sobre ellos y eligiendo *Campos...*

## 4.2. Permisos y reglas de acceso

En Tryton es posible definir grupos de usuarios, y luego establecer un conjunto de permisos para cada grupo.

Los permisos se pueden determinar a nivel de modelo, de campo, de botón o de registro.

En el caso de los permisos para modelos y campos, es posible desactivar el chequeo si para determinada transacción se da al atributo *check\_access* el valor *False*.



```
with Transaction().set_context(_check_access=False):
    [...]
    # Aquí se realiza la acción que afecta al modelo o campo en cuestión
```

En el caso de botones y registros, las reglas de acceso siempre se imponen.

## Definición de reglas de acceso

Para establecer reglas de acceso a modelos se utilizan registros de tipo *ir.model.access*, en los cuales se define, para cada par *modelo/grupo*, los permisos de lectura, escritura, creación y eliminación. Si el usuario en cuestión pertenece a un grupo con permisos, entonces puede realizar la operación. En caso contrario, no se permite.

En el caso de accesos a un campo, el mecanismo es el mismo, pero se utilizan registros de tipo *ir.model.field.access*.

Para los botones, en el registro *ir.model.button* se define la lista de grupos de usuarios habilitados para ejecutarlo.

## Grupos de usuarios

Para asignar permisos a un grupo de usuarios primero se debe crear ese grupo y, eventualmente, asociarlo a uno o más usuarios.

Por ejemplo, se puede crear un grupo *Administrador de Cuentas*, y además asociar ese grupo al usuario *admin* que tiene por defecto el sistema

```
<record model="res.group" id="group_account_admin">
    <field name="name">Grupo Administrador de Cuentas</field>
</record>

<record model="res.user-res.group" id="user_admin_group_account_admin">
    <field name="user" ref="res.user_admin"/>
    <field name="group" ref="group_account_admin"/>
</record>
```

Una vez hecho esto, se crean dos registros en *ir.model.access*: en el primero no se define ningún grupo, de modo que es genérico y se aplica a todos los usuarios. Se niegan todos los permisos, salvo el de lectura (*perm\_read*).

En el segundo registro se asocia al grupo *Administrador de Cuentas* (*group\_account\_admin*), y se le otorgan permisos para todo.

De esta manera, nadie que no pertenezca al grupo *Administrador de Cuentas* podrá modificar el modelo *account.account*

```
<record model="ir.model.access" id="access_account_general">
    <field name="model" search="['model', '=', 'account.account']"/>
    <field name="perm_read" eval="True"/>
    <field name="perm_write" eval="False"/>
    <field name="perm_create" eval="False"/>
    <field name="perm_delete" eval="False"/>
</record>

<record model="ir.model.access" id="access_account_admin">
    <field name="model" search="['model', '=', 'account.account']"/>
    <field name="group" ref="group_account_admin"/>
</record>
```

```

<field name="perm_read" eval="True"/>
<field name="perm_write" eval="True"/>
<field name="perm_create" eval="True"/>
<field name="perm_delete" eval="True"/>
</record>

```

Toda esta información se guarda en un archivo que puede llamarse *access\_rights.xml*, dentro de una carpeta denominada *security*.

Y como ya se ha visto en otros casos, no hay que olvidar indicar a Tryton que procese este archivo. En *tryton.cfg*, en el apartado *xml* hay que agregar una línea como la siguiente:

```

xml:
    security/access_rights.xml

```

### 4.3. proteus

*proteus* es una librería que permite acceder a los modelos de Tryton como si fuera una aplicación cliente. Sin embargo, no es necesario que el servidor Tryton esté ejecutándose para poder utilizarlo: el servidor Tryton es utilizado como un módulo más.

Se puede usar *proteus* para instalar módulos, crear entidades (*Party*) o cualquier otro modelo, e incluso ejecutar *wizards*, todo sin intervención directa de un usuario.

Es necesario configurar una conexión con la base de datos:

```

config.set_trytond(password='admin', database_name='training')

```

En el siguiente ejemplo se toma un archivo con datos de tipo CSV (valores separados por coma) y se lo procesa.

El objetivo es que nuevas oportunidades (*Opportunity*) se carguen en la base de datos y se asocien a una entidad (*Party*). Si esta entidad no existiera, se la crea primero.

```

import csv
import datetime
import sys
from proteus import config, Model

def main(args):
    Party = Model.get('party.party')
    Opportunity = Model.get('training.opportunity')
    csv_file = csv.DictReader(open(args[1], 'r'))
    for line in csv_file:
        parties = Party.find([('name', '=', line['Party'])])
        if not parties:
            # no existe la Entidad: se la crea
            party = Party(name=line['Party'])
            party.save()
        else:
            # la Entidad existe ya
            party = parties[0]
        line_date = datetime.datetime.strptime(line['Date'], '%d/%m/%y').date()

```

```
new_opportunity = Opportunity(party=party,
                              description=line['Description'], start_date=line_date)
new_opportunity.save()

if __name__ == '__main__':
    config.set_trytond(password='admin', database_name='training')
    main(sys.argv)
```

Si suponemos que este código Python se almacena en un archivo llamado *load\_opportunities.py*, para ejecutarlo habría que escribir:

```
$ python load_opportunities.py archivo_con_datos.csv
```

Se asume además que se utiliza una base de datos llamada *training*, y que el password del *admin* es 'admin'

## 5. Referencias y fuentes

- . La inspiración principal de este documento corresponde a una presentación de Nicolas Évrard: <http://downloads.tryton.org/TUL2014/crash-course.pdf>
- . El módulo *training* utilizado en los ejemplos se puede encontrar en forma de parches (*patches* de *Mercurial*) en <http://hg.tryton.org/>
- . Documentación de Tryton: <http://doc.tryton.org/>
- . Tryton es español: <http://www.tryton-erp.es/>