

2^{do} Encuentro: Tryton (desarrollo rápido en Python)

Capacitación técnica

Una introducción al desarrollo en Tryton
— Segunda Parte —

02/06/2017

Autores

Francisco M. Moyano Casco
Francisco Arata
Carlos Scotta
Fernando Sassetti
Ingrid Spessotti



Temario

1. Presentación

- ~~1.1 Características generales de Tryton~~
- ~~1.2 Herramientas de uso e instalación~~

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 Estructura de nuestro directorio~~
- ~~2.6 Arrancando el cliente~~
- ~~2.7 Instalando el modulo creado~~
- 2.8 Responder a las acciones del usuario: on_change
- 2.9 Listas y mas listas



Temario

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



2.8 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` y 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`.

¿QUE ES UN DECORADOR?

PEP 318 -- Decorators for Functions and Methods



2.8 Responder a las acciones del usuario: on_change

Ejemplo on_change_<nombre campo>

Como parte de la clase Opportunity, agregamos el siguiente método. Si no estamos utilizando el servidor en modo desarrollo, lo paramos y lo actualizamos como ya se ha visto en la primer parte.

```
@fields.depends('party')
def on_change_party(self):
    self.description = None
    if self.party:
        self.description = 'Taller Uner 2017'
        self.comment = 'Bienvenidos a la segunda parte del Taller'
```

The screenshot shows the top part of a web form. At the top is a toolbar with various icons. Below it, the 'Party' field is an empty text input. To its right is the 'Description' field, also empty. Below 'Party' is the 'Start Date' field, and below 'Description' is the 'End Date' field. At the bottom of this section is a large 'Comment' text area, which is currently empty.

This screenshot shows the same form after the 'Party' field has been updated to 'Pedro'. The 'Description' field now contains the text 'Taller Uner 2017'. The 'Comment' text area now contains the text 'Bienvenidos a la segunda parte del Taller'. An arrow points from the first screenshot to this one, indicating the state change.

2.8 Responder a las acciones del usuario: on_change

Ejemplo on_change_with_<nombre campo>

El siguiente metodo, también dentro de la clase Opportunity, precisara además de un atributo extra, delta_days, e importar un modulo de python para utilizar la función timedelta.

```
from datetime import datetime, timedelta
```

 ← Cerca de la cabecera del archivo

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

Party:	<input type="text" value="Pedro"/>	Description:	<input type="text" value="Taller Uner 2017"/>
Start Date:	<input type="text" value="30/05/2017"/>	Delta days:	<input type="text"/>
End Date:	<input type="text"/>		
Comment			
<input type="text" value="Bienvenidos a la segunda parte del Taller"/>			

Party:	<input type="text" value="Pedro"/>	Description:	<input type="text" value="Taller Uner 2017"/>
Start Date:	<input type="text" value="30/05/2017"/>	Delta days:	<input type="text" value="10"/>
End Date:	<input type="text" value="09/06/2017"/>		
Comment			
<input type="text" value="Bienvenidos a la segunda parte del Taller"/>			

Implementemos algo interesante con lo visto hasta ahora y con la documentación básica

0. Implementamos el código visto.

1. Necesitamos saber cuantas veces ha cambiado de valor el campo *party*.

2. Queremos saber cuantos caracteres tiene el atributo *name* del campo *party*.

3. Es necesario que todos los caracteres del campo *description* estén en minúsculas

4. En un campo indique que porcentaje del formulario ha sido completado hasta el momento.



2.9 Listas y mas listas

De un registros a muchos: *One2Many*

Es una relación de uno a muchos. Requiere tener el opuesto Many2One o una Reference definida en el modelo al que apunta. Son listas que no se comparten

```
class Chance(ModelSQL, ModelView):
    'Chance'
    __name__ = 'training.chance'
    __rec_name__ = 'description'

    name = fields.Char('Name')

    opportunity = fields.Many2One('training.opportunity', 'Opportunity')
```

```
class Opportunity(Workflow, 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')
    delta_days = fields.Integer('Delta days')
    party = fields.Many2One('party.party', 'Party', required=True)
    comment = fields.Text('Comment')
    chances = fields.One2Many('training.chance', 'opportunity', 'Chances')
```


2.9 Listas y mas listas

One2Many

Ejemplo en pantalla



2.9 Listas y mas listas

De muchos registros a muchos registros: Many2Many

Es una relación de muchos a muchos. Requiere tener un registro intermedio con los opuestos Many2One o una Reference definidas correspondientemente a los modelos a los que apunta. Son listas cuyos elementos se pueden compartir entre distintos registros.

```
class Involved(ModelSQL, ModelView):
    'Involved'
    __name__ = 'training.involved'
    _rec_name = 'party'

    party = fields.Many2One('party.party', 'Involved', required=True)

    role = fields.Selection([
        ('opportunity_admin', 'Opportunity Admin'),
        ('helper', 'Helper'),
        ('hard_worker', 'Hard Worker'),
        ('prime_mate', 'Prime Mate'),
        ('rookie', 'Rookie')
    ], 'Role', sort=False, required=True)

    def get_rec_name(self, name):
        return self.party.name
```

```
class OpportunityInvolved(ModelSQL, ModelView):
    'Opportunity Involveds'
    __name__ = 'training.opportunity_training.involved'
    _rec_name = 'involved'

    opportunity = fields.Many2One('training.opportunity', 'Opportunity')

    involved = fields.Many2One('training.involved', 'Involved')
```

Necesitaremos el siguiente campo en Opportunity

```
involved = fields.Many2Many('training.opportunity_training.involved',
    'opportunity', 'involved', 'Involveds')
```

Los parámetros del campo son:

- la relación que une a los modelos:
'training.opportunity_training.involved'
- el campo que apunta al origen
'opportunity'
- y el campo que apunta al objetivo (target)
'involved'

Party: Description:
Start Date: Delta days:
End Date:
Comment:

Involveds

Involved	Role
----------	------

tryton://localhost:8100/taller_des_uner/model/training.opportunity;context=?

2.9 Listas y mas listas

Many2Many

Ejemplo en pantalla



Implementemos algo interesante con lo visto hasta ahora y con la documentación básica

0. Implementamos el código visto.

1. Implemente alguna manera de que un campo *One2Many* se comporte lo mas aproximado a un *Many2Many*

2. Necesitamos un campo que lleve la cuenta de cuantos elementos tiene un campo *One2Many*

3. ¿Que ocurriría si el *target* del *Many2Many* no fuera un *Many2One* y fuera un *One2Many*? ¿Se puede? **PARA LA CASA**

4. Como podríamos hacer para que en el ejemplo de *Many2Many*, no se pueda seleccionar un colaborador igual al party del formulario *opportunity*. **PARA LA CASA**





3. CARACTERÍSTICAS AVANZADAS

```
from trytond.model import ModelSQL, ModelView, fields, Workflow
from trytond.pool import Pool
from trytond.wizard import StateTransition
from trytond.pyson import Eval, Not, Bool, PYSONEncoder, Equal, And, Or, If
from datetime import datetime, timedelta
```



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 haber un campo *state*:

```
state = fields.Selection([
    (None, ''),
    ('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'),
        ('converted', 'opportunity'),
    ))
```

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(),
    })
```

```
@classmethod
@ModelView.button
@Workflow.transition('opportunity')
def inconvert(cls, opportunities):
    cls.write(opportunities, {
        'end_date': None,
        'delta_days': None,
    })
```



3.1 Workflows

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:

opportunity_form.xml

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

Definida las transiciones, en `__setup__()` se declara la existencia del botón y su actualización.

Por ultimo, deberemos definir el valor por defecto del campo state

```
@staticmethod
def default_state():
    return 'opportunity'
```

```
@classmethod
def __setup__(cls):
    super(Opportunity, cls).__setup__()

    cls._transitions |= set((
        ('opportunity', 'converted'),
        ('opportunity', 'lost'),
        ('converted', 'opportunity'),
    ))

    cls._buttons.update({
        'convert': {
            'invisible': Eval('state').in_(['converted']),
        },
        'invert': {
            'invisible': Eval('state').in_(['opportunity', 'lost']),
        },
    })
```

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 otros palabras, no es necesario modificar manualmente el valor del campo state.



Implementemos algo interesante con lo visto hasta ahora y con la documentación básica

0. Implementamos el código visto.

1. Preguntas?????



3.1 Mejorando las vistas

Una manera de mantener un mejor orden en el formulario, de manera que no se nos pierda nada por irse “muy abajo”, es organizar, por ejemplo, algunas partes del mismo dentro las *pages* en un solo elemento *notebook*.

opportunity_form.xml

```
<notebook>
  <page id="chances" string="Chances">
    <separator name="chances" colspan="4"/>
    <field name="chances" colspan="4"/>
  </page>
  <page id="involveds" string="Involveds">
    <separator name="involved" colspan="4"/>
    <field name="involved" colspan="4"/>
  </page>
</notebook>
```



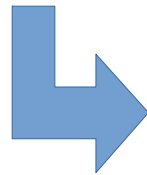
The mockup shows a form with two tabs: 'Chances' and 'Involveds'. The 'Chances' tab is selected and displays a table with a single row labeled 'Chance 1' under the 'Name' header. The 'Involveds' tab is also visible but empty.



3.1 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="convert" string="Convert" icon="tryton-go-next"/>
  <button name="unconvert" string="Unconvert" icon="tryton-go-previous"/>
</group>
```



3.1 Mejorando las vistas

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*.

Ya se habló del atributo *states*, que define un conjunto de valores posibles para *readonly*, *required*, *Invisible*. En este caso utilizamos el atributo *state*, para generar una condición de cambio de estado de los atributos *start_date* y *end_date*.

Agregamos lo siguiente cerca de la cabecera del archivo

```
from trytond.pyson import Eval, Not, Bool, PYSONEncoder, Equal, And, Or, If
```

Modificamos los campos en la clase Opportunity

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



3.1 Mejorando las vistas

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, Not, Bool, PYSONEncoder, Equal, And, Or, If
```

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)



Implementemos algo interesante con lo visto hasta ahora y con la documentación sugerida

1. Implemente una manera de que el formulario tenga todos los campos en solo lectura, excepto uno que cuando se complete, se pueda editar algún otro, y así sucesivamente hasta tener completos todos los campos. Cuando llegue al ultimo, y se complete, el formulario debe quedar solo en lectura. **PARA LA CASA**

2. Lo mismo, excepto que el primero debe ser obligatorio y que con un botón (o mas), pueda habilitarse uno a uno los otros campos, que deberán ser obligatorios, y que al llegar al ultimo, se tornen solo de lectura (no se puedan editar). **PARA LA CASA**

3. Habilite o deshabilite la visibilidad de algún campo y del *group* de la vista con un campo *Bool* (*fields.Boolean*). **ESTE NO ES PARA LA CASA**




Ver documentación Oficial





¿ Para dónde vamos ?





Repetir el encuentro para seguir avanzando en/con/para el aprendizaje/enseñanza


¿FECHA SUGERIDA?





¿Qué proponemos?





Que practiquen lo aprendido para el próximo
encuentro

Que vengan (de venir, no de vengarse)

Que traigan sus dudas (serán muchas y atendidas!)

Que no se queden con las ganas!!!

Que vean los canales y la documentación sugerida





Fuentes de consulta!!!!!!!

www.tryton.org Sitio oficial de tryton, donde puede descargarse los códigos fuentes y principales enlaces a otras fuentes

doc.tryton.org/3.8 documentación oficial de la versión 3.8

www.gnusolidario.org Sitio oficial del proyecto gnu solidario

<https://groups.google.com/forum/#!forum/tryton> Grupo oficial de tryton. También esta el grupo español y el argentino.

moyanocasco.franciscom@gmail.com el mas lerdo para responder.





Mas Fuentes de consulta!!!!!!!

<http://www.python.org.ar/> Comunidad Python argentina. Mucha documentación sobre el lenguaje, lista de correo, noticias de eventos, foro de consultas.

www.linux-malaga.org/index.php?s=file_download&id=9 *Python en 14* páginas. Guía de referencia rápida de python.

<https://www.python.org/dev/peps/pep-0020/> *PEP 20*. El Zen de python. **!!!Lectura obligatoria!!!**

<https://www.python.org/dev/peps/pep-0008/> *PEP 8* ("Pepocho"). Guía de estilo para codificar en Python.

Luego de leer PEP 20.





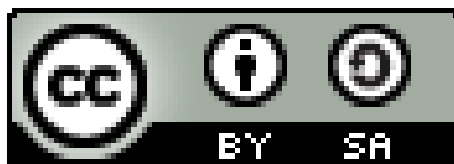
Muchas gracias por venir!

Cátedra de Salud Pública – Facultad de Ingeniería (UNER)

E-mail: saludpublica@bioingenieria.edu.ar

Facebook: www.facebook.com/catedraSaludPublica





Basado en el trabajo *Tryton: Capacitación técnica. Una introducción al desarrollo en Tryton* de los autores: Mario Puntin, Adrián Bernardi <contacto@silix.com.ar>; Versión:0.2; Fecha:Mayo de 2015

Todo el contenido esta licenciado bajo Creative Commons Attribution-ShareAlike 4.0 International License.

Las capturas de pantalla a la interfaz de Tryton fueron tomadas desde mi computadora personal

