apuntes_de_tryton Documentation

Release 1.0

gcoop

February 11, 2016

Contents

Aca compartimos las notas que tomamos, en el curso dictado por Thymbra, durante el 2012.

Son apuntes para empezar a desarrollar con Tryton.

Correcciones y agregados son bienvenidos.

CHAPTER 1

Funcional

Estas notas corresponden a la parte funcional del curso.

Indice

1.1 Instalar módulos

Para instalar, por ejemplo, el modulo Party:

- De ser necesario, crear el directorio 'trytond/modules'.
- Copiar el contenido del módulo party dentro de 'tryton/modules/party'.
- Ejecutar:

```
./trytond --all --config=ejemplo.conf --database=ejemplo
```

- Dentro del cliente, marcar para instalar.
- Ir a vista de lista.
- Click en rombo de acción.

- Instalar todos los módulos seleccionados.
- Iniciar instalación.

1.2 Entidades

Entidad: Cualquier tercero con el cual realicé una transacción.

1.3 Moneda

- El módulo Currency incluye bocha de monedas.
- Una de ellas es la moneda pivot, en Tryton es EURO.

1.4 Compras

Es el módulo Purchase.

- Primer estado: Draft (No genera movimientos de ningún tipo).
- Tryton propone 3 workflows posibles:
  - Basado en el pedido.
  - Basado en el envío.
  - Manual.
- Cuando se valida la OC se genera una factura, es decir, se crea un registro con la factura que me va a mandar el proveedor.

1.4.1 Carga de la OC

- Vamos al menú Compras / Compras.
- Tryton permite establecer plazos de pago con reglas lógicas que son acumulables.
- En el formulario de compras, si quiero que sólo me traiga las entidades que son proveedores, hay que agregarle un dominio a ese campo Tercero (Entidad). Esto es con código.
- Cargamos las líneas de la OC:
  - El producto no es requerido y la descripción sí porque puedo pedir algo que no está catalogado.
  - Si defino la orden de compra en dólares y los productos los tengo valorizados en pesos, en las líneas transforma ese precio en Pesos a los dólares correspondientes. Para esto utiliza la última tasa de cambio.
  - Una vez guardado el Draft de compras, si le doy a Presupuestar, significa que el proveedor me confirmó la OC.
  - Si le doy click a Confirmar, desencadena el workflow.
  - Al darle click al botón Imprimir genera la factura. Por defecto el reporte genera un .odt pero bien podría sacar un pdf.

1.4.2 Producto

- Si el producto es "Consumible" no se maneja stock.
- Usar cuentas de la categoría: Se configura una cuenta de ingresos y egresos a una categoría y el producto hereda las cuentas de esa categoría. Así no es necesario cargar cuentas por cada producto.
- Precio de Lista / Precio de costo : La moneda es la de la compañia (la mía).

1.5 Stock

Funciona utilizando el método de partida doble y deben configurarse diferentes ubicaciones dentro de la organización.

Hay que configurar:

- Depósitos
- Ubicaciones
  - Una ubicación de tipo View (Vista) no va a tener movimientos, si no sus hijos.

1.5.1 Inventarios

- Generar una entrada en inventarios genera los movimientos necesarios para ajustar el stock esperado al stock real.
- Supongamos que tengo 20 unidades del producto A en sistema, pero cuando voy a la estantería y los cuento, me encuentro con que hay 15. Entonces voy a Inventario y cargo 15 en inventario. Esto dispara automáticamente los movimientos necesarios para ajustar el stock de sistema, en particular, como funciona por el método de partida doble, saca 5 unidades de sistema y los pasa a la cuenta "Lost and Found".

Hay 2 tipos de stock: Real y Virtual

Al stock Real lo alimentan:

- Remitos.
- Inventario.
- Movimientos Internos.

Al stock virtual lo alimentan:

- Ordenes de Compra.
- Ordenes de Venta.
- Ordenes de Producción.

En "Movimientos" (Moves) vemos el listado de movimientos de Stock. Todos aquellos que están en estado "Done" modifican el stock real y los que están en estado "Draft" modifican el stock virtual.

Cuando compro se generan Órdenes de Compra. Estas órdenes de compra generan movimientos de stock en estado Draft. Cuando el producto que compré llega viene con un remito del proveedor. Con ese remito, voy a "Supplier Shipments" y cargo el remito. Cuando confirmo el remito, por un lado confirma los movimientos anteriores (pasa los movimientos de Supplier a Input Zone, y de Draft a Done) y genera otros movimientos en estado Draft que van de "Input Zone" a "Storage Zone".

1.6 Módulo Ventas

Muy similar al módulo de compras. Cuando se genera una venta y se le da click al botón "Quote" (Presupuesto) se genera una cotización. Cuando confirmo la orden, la Cotización pasa a ser una venta confirmada "Confirmed Sale".

Instalar el módulo stock_lot para tener manejo de lotes tanto en las ventas como en las compras.

1.7 Contabilidad

Módulo Financial

La contabilidad es online, es decir, automáticamente hace los asientos.

Vamos a necesitar:

- Un plan de cuentas.
- Definir años y períodos fiscales.
  - El año fiscal va a estar definido en períodos.
- Configurar los Sub-diarios.
- Configurar impuestos (IVA e impuestos de cálculo simple).
  - Las retenciones al momento de pagar se maneja con la localización.
- Configurar los términos de pago.

1.7.1 Configurar cuentas

Lo que marca la funcionalidad de una cuenta es el campo Kind.

Tipos de cuenta:

- View: No son cuentas para imputar, son para ser cuenta padre o jerarquizar.
- Revenue y Expense : Son cuentas de ganancias y gastos (Cuentas de resultado positivo y negativo).
- Payable: Cuentas a pagar.
- Receivable: Cuentas a cobrar.
- Stock: Para valorización de stock.
- Other: Ninguna de las anteriores.

La moneda principal es por defecto la de la compañia, o se puede especificar. El campo Reconcile (Conciliable), se usa para aquellas cuentas que haga falta conciliar: pago a proveedores, deudores por venta, etc.

- Plan de cuentas (View)
  - Activo (View)
    - Deudores por venta (Receivable - A cobrar)
    - Caja (Other)
    - Banco (Other)
    - IVA crédito fiscal (Other)
  - Pasivo (View)

        * Proveedores (Payable)
        * IVA débito fiscal (Other)
    - Patrimonio Neto
    - Resultado positivo
        * Ingresos (Revenue)
    - Resultado negativo (View)
        * Gastos (Expense)
        * Costo mercadería vendida (Expense)

1.7.2 Años fiscales

- Tiene fecha de inicio y fin.
- No se crean automáticamente.
- Si estás dentro de un año fiscal no podés operar.

Períodos

Se pueden cargar manuelamente o via wizard períodos mensuales o trimestrales.

A cada producto se le tienen que configurar las cuentas Revenue y Expense, o bien puede tomar las cuentas de la categoría.

1.7.3 Diarios

Los diarios de tipo CASH son por ejemplo el diario de caja, bancos, etc. Los diarios de tipo SITUACION son para registrar cierres o cosas por el estilo. Los diarios de tipo GENERAL son para diarios de ajuste, o cosas que no se engloban en las categorías anteriores.

Cada diario puede tener definido un tipo de vista diferente.

1.7.4 Impuestos

Hay 3 formas de cálculo: Porcentaje, Fijo o ninguno. Los impuestos se calculan por línea de factura. Antes había una opción para meter código python, hoy no hay mas.

1.7.5 Plazos de Pago

Puede configurarse que sea fijo, remanente, porcentaje sobre total o sobre el remanente

1.8 Localización

La localización que hace Thymbra está en el Wiki de Tryton Argentina que es un wiki que mantienen ellos, y aparentemente quieren que sea comunitario.

1.8.1 Módulo account_ar

Bajamos e instalamos el módulo:

account_ar

Hay que ir al módulo Contabilidad -> Planes contables -> Crear plan contable desde plantilla y seleccionar el Template Plan contable argentino.

Luego instalamos el módulo:

account_bank_ar

que son todos los bancos de argentina con sus datos

Luego de instalado esto hay que ir a Entidades -> Empresas -> Emrpesas seleccionar mi empresa y configurarle la cuenta bancaria, en nuestro caso elegimos el Credicoop ;-)

Instalamos el módulo: account_voucher_ar

que es un módulo que provee pagos y cobros.

- Creamos una OC.
- Esa OC genera una factura de proveedor.
- Vamos a Contabilidad -> Comprobantes -> Pagos.
- Desde ahí genero un pago que cancele una o más facturas.
- Luego se le da click al botón Pagar y genera todos los asientos contables, conciliaciones, etc.

Para el caso de las ventas es similar:

- Creamos una Venta.
- Esa Venta genera una factura por cobrar.
- Vamos a Contabilidad -> Comprobantes -> Recibos.
- Desde ahí genero un cobro que cancele una o más facturas.
- Luego se le dá click al botón Pagar (Debería ser Cobrar :P) y genera todos los asientos contables, conciliaciones, etc.

Instalamos el módulo: account_check_ar

Este módulo agrega el menú de Tesorería dentro de Contabilidad. Además agrega una solapa en los Diarios. Ahí configuro las cuentas contables donde van los cheques recibidos y los cheques emitidos. (Para la ver 2.4 falta la opción de cheque rechazado).

CHAPTER 2

Desarrollo

Estas notas corresponden a la parte técnica del curso.

Indice

2.1 Tryton como framework

Por un lado el curso que estoy haciendo (LucianoRossi) es la parte técnica. El enfoque del curso es mirar a tryton como un framework y no como un ERP.

La instalación en la que nos basamos originalmente es la versión 2.4, actualizada a la 3.4.

La documentación oficial de instalación: http://doc.tryton.org/3.4/ .

Según la guia de instalación debemos instalar previamente algunos paquetes además de postgresql. Tryton puede trabajar con base de datos como postgresql, mysql, sqlite, etc.

Tryton tiene un sistema de comunicación CLIENTE - SERVIDOR. La comunicación que se usa es jsonrpc. Uno podría desarrollar una aplicación cliente y comunicarse vía jsonrpc con el servidor tryton. Las modificaciones que vamos a hacer durante el curso son a los módulos que corren en el servidor. Tryton es bien modular y por lo visto, no es necesario hacer modificaciones al core.

Por un lado tenemos los .py que van a contener las clases, métodos. Luego tenemos archivos .xml que contiene las definiciones de las vistas. Las vistas se diferencian entre las que son tree y form. Tree son listas (listview) y form son detail y edit.

2.1.1 Instalación

Para la versión 3.4 estamos instalando de la siguiente manera:

- instalar los paquetes necesarios para ejecutar y desarrollar sobre el servidor:

```
apt-get install \
      python-dev swig python-pip python-lxml libxml2-dev libxslt-dev \
      postgresql postgresql-client postgresql-server-dev-all libpq-dev \
      make git mercurial
```

  - si queremos generar pdfs, es necesario instalar dos paquetes más:

```
apt-get install libreoffice-java-common python-uno
```

- instalar el virtualenvwrapper:

```
pip install virtualenvwrapper
```

- crear el entorno virtual para nuestro servidor y activarlo:

```
mkvirtualenv tryton
workon tryton
```

- instalar los módulos de python:

```
pip install \
      trytond psycopg2 pytz vatnumber suds vobject proteus Genshi \
      argparse lxml polib python-dateutil python-sql relatorio six
```

- configurar el usuario en postgresql:

```
sudo su postgres -c 'createuser --createdb --no-adduser -P Tryton'
```

  - para la clave podemos usar `tryton`.
  - si no cambiamos la configuración el servidor postgresql responderá sólo a localhost.

- modificar la configuración del servidor `trytond` en `etc/trytond.conf`:

```
[jsonrpc]
listen=*:8000
data=/var/lib/trytond

[database]
```

```
    uri=postgresql://tryton:tryton@localhost:5432
    path=/var/lib/trytond # directorio donde se guardaran los adjuntos

    [session]
    timeout=3600
    super_pwd=hrNNibAnqs1ng
```

  - el crypt indicado en `super_pwd` corresponde a la clave `tryton`. de ser necesario puede generarse un crypt para otra clave con el siguiente comando:

```
    python -c 'import getpass,crypt,random,string; print crypt.crypt(getpass.getpass(), \
              "".join(random.sample(string.ascii_letters + string.digits, 8)))'
```

- para correr el servidor ejecutar el comando `trytond`.

2.2 Modulos

Los módulos (party, company, account, sale, etc) se instalan en trytond/trytond/modules. Pero en el curso se toma la idea de instalar los modulos por fuera de la carpeta de trytond y lo que hacemos es crear enlaces simbólicos hacia la carpeta donde dejaremos tanto los módulos oficiales, como los módulos extras.

Cuando bajamos un módulo tryton debemos mirar los archivos:

```
tryton.cfg
__init__.py
```

En tryton.cfg encontramos alguna descripción del módulo y también cuales son sus dependencias. Debemos verificar si tenemos todos los módulos que nos piden para poder activarlos. Si el módulo se instaló usando `pip` sus dependencias normalmente se resuelven automágicamente.

Al crear los enlaces simbólicos y tenerlos en la carpeta modulos, debemos updatear el servidor, para que popule nuevamente la base de datos avisandole que han habido modificaciones y que tenemos nuevos módulos para poder instalar:

```
bin/trytond --config=etc/tryton.conf --database=tryton --all
```

2.2.1 Actualización de módulos

¿Cuando se debe ejecutar la opción `--all`?

Al modificar los archivos .py los cambios se toman al momento.

Si se agrega un nuevo field, o se modifica la vista (xml) se debe ejecutar –update=all para que cree en field en la base de datos, o popule la metadata de la vista nuevamente. Esto aclara que NO esta leyendo el xml cada vez que ejecuta el servidor.

2.3 Primer Modulo

2.3.1 Fields

Dentro de la clase fields estan los tipos de campos: Char, Many2One, Many2Many, Boolean, etc. Impactan tanto en la base de datos, como en las vistas.

Un ejemplo de declaración de campos para un modelo:

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

class Curso(ModelSQL, ModelView):
    'Curso'
    _name = 'academia.curso'
    _description = 'Clase Tryton. Curso'

    name = fields.Char('Name', size=None, required=True)
    descripcion = fields.Text('Description', required=True)
    user_id = fields.Many2One('res.user','Responsable')
    sesion_id = fields.One2Many('academia.sesion','course_id','Sesion')

Curso()
```

En el caso del campo Selection, si se agrega el atributo readonly=True, entonces el usuario no puede cambiar el campo, pero va cambiando cuando va pasando a través del workflow.

Note: Recordar: al agregar un modelo nuevo SIEMPRE se debe agregar al archivo __init__

La forma de agregar al archivo __init__

```
from .curso import *
```

2.3.2 Vistas

Las vistas estan contenidas en los archivos xml

El contenido de los xml deben estar dentro de los tags <tryton> y <data>. Todo tag que se abre, debe tener un tag de cierre.

Ejemplo:

```
<tryton
    <data>
        <field ... />
    </data>
</tryton>
```

tags

Hablare sobre los distintos tipos de tags que podemos encontrar en un archivo xml.

1. tag <menuitem> A nivel representación esta en el menu izquierdo del cliente. Para darle jerarquia se maneja como padres e hijos. Atributos posibles:

```
- parent: Si no tiene atributo parent, entonces usa el root del menu.
- action: Se usan para abrir una vista de formulario, de lista, wizard, reporte.
- id: El id es la forma en la que vamos a referenciar al menu. Ejemplo, padre -> hijo.
- sequence: La secuencia en que se muestran los items de menu.
```

Ejemplo de tag <menuitem>

```
<menuitem name="Academias" sequence="0" id="menu_academia" />
<menuitem id="menu_academia_curso" parent="menu_academia" action="act_academia_curso_form"/>
```

#. tag <record> La Convencion es que primero se define el action (record) y luego el menu que lo asocia. Atributos posibles:

```
- model: nos dice que modelo trabaja y en que tabla lo inserta.
- res_model: le especificamos vista de que modelo estamos llamando.
```

Ejemplo de tag <record>

```
<record model="ir.action.act_window" id="act_academia_curso_form">
    <field name="name">Cursos</field>
    <field name="res_model">academia.curso</field> # MODELO
</record>
```

2.3.3 Vistas form, tree

Las vistas de tipo tree serias los listview en sugar y form serian los edit/detail. Por defecto las vista se reparte en 4 columnas. Entonces tenemos algo como:

```
LABEL | FIELD | LABEL | FIELD
```

El atributo record model siempre es *ir.ui.view*

Ejemplo de vista form

```
<!-- Vista curso -->
<record model="ir.ui.view" id="academia_curso_view_form">
<!--
     Por convención el id es el nombre de la clase y tipo de vista.
 -->

    <field name="model">academia.curso</field>
    <!--
         Especificamos que modelo usamos. El record de action lo usa como
         referencia.
    -->

    <field name="type">form</field>
    <!--
         Estamos diagramando la vista del form, sino deberia decir tree
     -->

    <field name="arch" type="xml">
        <![CDATA[
       <!--
            Es un tag de xml que formatea el codigo para insertarlo en el campo form.
        -->

        <form string="Curso">
                <label name="name" />
                <!--
                     Se muestra el label. Si no agregamos este dato, no se muestra
                 -->

                <field name="name" />
                <!--
                    Se muestra el campo de name="name"
                 -->
```

```
                <label name="user_id" />
                <field name="user_id" />
            <notebook colspan="4">
            <!--
                 Es un contenedor de tags `page`. Por cada page nos crea un tab nuevo.
                 Si a este tag le agregamos colspan=4 le estamos indicando que ocupe
                 4 columnas.
             -->

                <page string="Descripcion" id="curso_descripcion">
                <!--
                    Dentro de pages se agregan los tags <fields>. Las pages tiene
                    atributo id. Con el id se puede heredar la page desde otro modulo.
                    Se le puede agregar el atributo col="4". Sirve para setear la page
                    en 4 columnas. En este caso es redundante.
                 -->

                    <field name="descripcion" />
                </page>
                <page string="Sesiones" id="curso_sesiones">
                    <field name="sesion_id" />
                </page>
            </notebook>
        </form>
        ]]>
    </field>
</record>
```

Ejemplo de record para vista tree

```
<record model="ir.ui.view" id="academia_curso_view_tree">
    <field name="model">academia.curso</field>
    <field name="type">tree</field>
    <field name="arch" type="xml">
    <![CDATA[
        <tree string="Cursos">
            <field name="name" />
        </tree>
    ]]>
    </field>
</record>
```

Note: Los form tree se agregan los fields sin los tags label. Los campos que se agregan a la vista, se cargan como filtros de búsqueda. NO hay forma de agregar filtros de búsqueda si no estan en el form tree.

2.4 Campos relacionables

2.4.1 One2Many

Un registro hacia muchos. La representación en el cliente se muestra como campos grilla que uno va agregando registro.

La forma de declaración:

```
fields.One2Many(`nombre de la clase extranjera`, `campo clase extranjera`, `label`)
```

Note: El campo de la clase extranjera debe ser definida como Many2One

```
class Curso(ModelSQL, ModelView):
    'Clase Local. Curso'
    _name = 'academia.curso'
    _description = 'Clase Tryton. Curso'

    sesion_id = fields.One2Many('academia.sesion','course_id','Sesion')

Curso()

class Sesion(ModelSQL, ModelView):
    'Clase Extranjera. Sesion'
    _name = 'academia.sesion'
    _description = 'Clase Tryton. Sesion'

    course_id = fields.Many2One('academia.curso','Curso')

Sesion()
```

2.4.2 Many2One

Relacion Muchos contra un Registro. Siempre son entre dos clases la relación. La representación en el cliente se visualiza con una lupa al costado del text field. Almacena el id del reigstro extranjero.

La forma de declaración:

```
fields.Many2One(`nombre de la clase extranjera`,`label`)
```

```
class Sesion(ModelSQL, ModelView):
    'Clase Extranjera. Sesion'
    _name = 'academia.sesion'
    _description = 'Clase Tryton. Sesion'

    course_id = fields.Many2One('academia.curso','Curso')

Sesion()
```

2.4.3 Many2Many

Clase A y Clase B. Se relacionan usando una tabla intermedia, y voy almacenar idA, idB Si tengo clase Local y Extranjera. Lo declaro en Local y en clase intermedia. Lo declaro con dos campos, uno que apunta a local y otra a extranjera.

La forma de declaración:

```
fields.Many2Many( `clase intermedia`, `campo idA`, `campo idB`, `label`)
```

```
class Sesion(ModelSQL, ModelView):
    'Sesion'
    _name = 'academia.sesion'
    _description = 'Clase Tryton. Sesion'
```

```
    alumno_id = fields.Many2Many('academia.sesion-academia.alumno','sesion_id','alumno_id','Alumno')

Sesion()

class SesionAlumno(ModelSQL):
    'Clase intermedia Alumno - Sesion'
    _name = 'academia.sesion-academia.alumno'
    _table = 'sesion_alumno_rel'

    _description = 'Tabla relacion sesion - alumno'
    sesion_id = fields.Many2One('academia.sesion', 'Sesion')
    alumno_id = fields.Many2One('academia.alumno', 'Alumno')

SesionAlumno()
```

La clase intermedia hereda unicamente de ModelSQL. Por convencion se nombra a la clase de la forma: **nombre.de.la.clase.local-nombre.de.la.clase.extranjera** También se puede renombrar el nombre de la tabla, y se toma por convención que sea: **claselocal_claseextranjera_rel**

Note: Puedo especificar el nombre de la tabla usando _table = 'nombre_tabla_rel'

2.5 Herencia

Clase 3 del curso Thymbra

La mayoria se resuelve usando herencia por extensión.

Se crea un nuevo modulo que hereda del anterior. **Todo en tryton es heredable**. Tanto las vistas como los modelos. Podemos agregar funciones nuevas, atributos, usar campos de la original. Por eso, radica la importancia del id en vistas (xml).

Important: No tiene el mismo mecanismo de herencia de python.

Note: Es bueno que la definicion del nuevo modulo haga poco, pero bien. Método KISS

2.5.1 Herencia de modelo

¿Cómo heredo en Tryton? Si seteamos el atributo **_name** con el nombre del modelo de otro módulo, entonces heredaremos los atributos y vistas de dicho modelo.

```
class Hola (Clase1, Clase2) # Heredo de Clase2, Clase3.
    "Clase Hola"
    _name = 'party.party' # Estoy indicando que heredo del objeto party.party

    instructor = fields.Boolean('instructor')

Hola()
```

2.5.2 Herencia de Vistas

Como modifico vista usando herencia.

Important: El id NO debe ser repetido.

```
<record model="ir.ui.view" id="academia_curso_party_view_form">
    <field name="model">party.party</field>
    <field name="inherit" ref="party.party_view_form" />
    <field name="type">form</field>
    <field name="arch" type="xml">
```

1. Indico que heredo del modelo party.party
2. Estoy referenciando desde que vista voy a modificar. Se debe pasar "NOMBRE_MODULO.ID_VISTA". Si no se pasa el nombre de modulo, entonces busca el id de vista en el xml de mi modulo. Si tengo otro xml en mi modulo, también buscará allí.

Important: Recordar, debe ser nombre del modulo, NO el nombre del modelo.

Para poder indicarle cual es el form o tree que voy a agregar o modificar debo llegar al mismo usando xpath.

Navego a través de formulario u/o arbol. Entro por form, paso por notebook, page lo identifico por id. Ahi me paro en un campo. La manera de posicionarme lo busco por name. Ejemplo name="description". Entre los atributos que puedo pasar es la position:

```
- after
- before
- replace. Reemplaza, y elimina el campo que estas posicionado.
- inside. Se usa mucho en los group. Se puede usar para moverse por ejemplo en una page vacia.
```

```
<![CDATA[
<data>
<xpath expr="/form/field[@name=&quot;code&quot;]" position="after">
        <label name="instructor" />
        <field name="instructor" />
</xpath>
</data>
]]>
```

2.5.3 def __init__

replace_attributes. Como hacerlo desde el modelo. Agrego mas items a un dom de un combo. Esta logica se puede usar para cualquier campo.

```
def  __init__(self):
    super(Sesion, self).__init__()

    self._error_messages.update({
        'alumnos_out': 'No se pueden agregar mas alumnos. Numero de asientos!',
    })

    self._transitions |= set((      # Si existe te lo reescribe. NO podes tener una transition repeti
        ('draft', 'confirmed'),   # (origen, destino)
```

```
        ('confirmed', 'done'),  # Podria tener la de (done,confirmed)
        ('done', 'cancel'),
        ))
```

2.6 Explorando los xml

Ahora vamos a ver distintos tags de XML y explicar algunos atributos.

Atributo mode. Le estas explicitando que primero queres que se vea el form, y luego el tree:

```
<field name="addresses" mode="form,tree" colspan="4" ..
```

Le explicitas que vista usa tanto para form, como para tree respectivamente. Los id de view estan definidos mas arriba en el archivo party.xml:

```
<field name="lines" colspan"4"
view_ids="party.address_view_form,party.address_view_tree_sequence"/>
```

Explicito que widget podes usar. En el caso del **field lang** se veria una lupa y te dejaria crear un registro nuevo. Directamente le pones widget=”selection”. ¿Widget posibles? Todos los tipos de campo tienen un widget:

```
<label name="lang"/>
<field name="lang" widget="selection"/>
```

2.6.1 ir.action.keyword

Indicas desde donde se lanzará los reportes, actions o wizards. Si queres que se agreguen al rombo de action

```
<record model="ir.action.keyword" id="act_marcar_presente_keyword">
    <field name="keyword">form_action</field>
    <field name="model">academia.sesion,-1</field>  <!-- Que vista de modelo aparece -->
    <field name="action" ref="act_marcar_presente_wizard"/>
</record>
```

2.6.2 ir.action.wizard

Se usa para declarar los wizard y el keyword para poder llamarlo.

```
<record model="ir.action.wizard" id="act_marcar_presente_wizard">
    <field name="name">Marcar Presente</field>
    <field name="wiz_name">academia.presents</field>
    <field name="model">academia.sesion</field>
</record>
```

2.6.3 Permisos (ACL)

Podemos definir los ACL desde el cliente. También podemos agregar un grupo o setear los permisos desde el XML.

TODO:

```
Buscar algún ejemplo.
```

2.7 Definición de 'domain'

El **domain**, sirve como regla de que registros, pueden verse y seleccionar desde el otro modelo.

La condicion se evalua contra el campo de la clase extranjera. Ejemplo: domain=[('instructor', '=', True)]

Important: domain funcionan únicamente sobre campos relaciones.

```
instructor_id = fields.Many2One('party.party','Instructor', domain=[('instructor','=',True)])

Primer valor: El campo de modelo extranjero. ('instructor')
Segundo valor: El filtro. Ej: ('=')
Tercer valor: El valor que queremos recibir. (True)
```

También podemos usar PYSON, que es una forma de comparación: http://doc.tryton.org/2.4/trytond/doc/topics/pyson.html

2.8 funciones on_change

Clase del Jueves en Curso Thymbra

El evento esta en el cliente. Manda una signal al servidor. En el servidor, tiende a ejecutar una función.

¿Cuando se ejecuta? Se ejecuta hasta que yo no quite el foco. Se define como atributo del campo.

```
alumnos = fields.BlaBla('Duration', on_change_with=['nombre_del_campo', 'nombre_campo_2'])

alumno_id = fields.Many2Many('academia.sesion-academia.alumno','sesion_id','alumno_id','Alumno', on_c
```

En el modelo de ese atributo, debemos crear una funcion que se llame **def on_change_with_alumnos(self, vals):** donde **alumnos** es el nombre del campo que seteamos en el atributo **on_change_with**. En el agumento vals viene un diccionario con los valores de los campos que seteamos en el atributo on_change_with.

```
def on_change_with_alumno_id(self, vals):

    alumnos = vals.get('alumno_id')

    if alumnos > vals['number_of_seats']:
        ## mensaje de error. sold_out es un key que contiene el mensaje de error
        self.raise_user_error('sold_out')

    [...]
    pool = Pool()
    party_obj = pool.get('party.party')
    address_obj = pool.get('party.address')
    payment_term_obj = pool.get('account.invoice.payment_term')
    res = {
        'invoice_address': None,
        'shipment_address': None,
        'payment_term': None,
    }

    return res

def  __init__(self):
    super(Sesion, self).__init__()
```

```
    self._error_messages.update({
        'sold_out': 'Tuvimos un error de seets',
    })
```

1. Las funciones on_change_with DEBEN devolver un diccionario.
2. Cada una de las claves deben ser el nombre de un campo:

```
    res['num_of_seats'] = 'Valor que le asigno al campo num_of_seats'
```

Si vamos a reescribir un campo relacionable, se debe pasar una estructura especial:

```
res['relacionable'].setdefault('add':[])
# Luego le hacemos un append de un diccionario con los nombre_campo : valor
res['relacionable'].setdefault('add':[])
```

2.9 Wizards

Con la accion wizard me abre un popup (modal). Ejemplo clasico, lanzamiento de reportes.

- Ingreso fechas y luego genero un reporte.
- Ingreso datos en mi formulario para generar datos.
- Wizards que exportan archivos.

Cuando se ejecuta el wizard, hace un llamado al servidor, y el server me devuelve la vista del wizard. Los datos que se insertan, se vuelven a mandar al servidor y el server devuelve el resultado. El resultado puede ser el final ('end') o trae otra ventana de wizard.

2.9.1 Algunas consideraciones

1. Para poder trabajar con wizard hacemos algunos import adicionales.

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

2. StateTransition -> estado entre una y otra
3. Button -> Botones para pasar de un estado a otro.

2.9.2 Ingredientes para hacer un wizard

Archivo .py

1. Voy a necesitar un model view. Designo campos, pero SIN inserción en la base de datos. Solo heredo de ModelView NO heredo de ModelSQL
2. Una clase Wizard. Declaro los StateView, los StateTransition, los Button, funciones.

2.9.3 Mirando el codigo

Lo primero que se define es un start. Generalmente se da una instancia de StateView para que nos muestre algo.

StateView (modelo del modelview que queremos mostrar, la vista de ese modelo, lista de botones [Button ('Label', 'Nombre de la próxima transition'), Button()]

```
class MarcarPresents(Wizard):
    'Presents'
    _name = 'academia.presents'

    start = StateView('academia.presents.start',
    'academia.marcar_presentes_form', [
    Button('Cancel', 'end', 'tryton-cancel'),
    Button('Presents', 'presents', 'tryton-ok', True),
    ])

    presents = StateTransition()

    def transition_presents(self, session):
        if session.data['start']['presente']:
            sesion_obj = Pool().get('academia.sesion').browse(Transaction().context.get('active_ids')

        ## Estoy iterando sobre objetos de alumnos
        #for alumno in sesion_obj.alumno_id:
        #   alumno.write(alumno.id, {'presente':True})

            for sesion in sesion_obj:
                sesion.write(sesion.id, {'presente':True})

        return 'end'

MarcarPresents()
```

transition *end*: Le estoy diciendo que finaliza el wizard. Palabra reservada.

transition *presents*: Es una transition que tengo que definir. Entonces presents = StateTransition(). Tryton va a buscar la function presents donde esta mi transition definida. Mi funcion deberia ser def transition_presents

Archivo .xml

Pasos:

1. View que corresponde al model view que declare en el modelo.
2. Action que voy a llamar al wizard, -> ir.action.wizard
3. keyword -> ir.action.keyword

2.9.4 Mirando el codigo

1. Es un form comun y corriente.

```
<!-- Vista Wizard MarcarPresente -->
<record model="ir.ui.view" id="marcar_presentes_form">
    <field name="model">academia.presents.start</field>
    <field name="type">form</field>
    <field name="arch" type="xml">
        <![CDATA[
        <form string="Marcar Presente">
                <label name="presente" />
                <field name="presente" />
        </form>
        ]]>
```

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

2. <record model=”ir.action.wizard” ..> <field name=”model”>academia.sesion</field> únicamente este wizard va a estar disponible en mi modelo academia.sesion <field name=”wiz_name”>academia.presents</field> Nombre del modelo wizard. academia.presents

```
<record model="ir.action.wizard" id="act_marcar_presente_wizard">
    <field name="name">Marcar Presente</field>
    <field name="wiz_name">academia.presents</field>
    <field name="model">academia.sesion</field>
</record>
```

3. keyword: form_action (le estamos diciendo que va a abrir una action form). model: academia.sesion, -1 action: act_wizard_presents. Es la referencia a lo que voy a ejecutar. En vez de llamarlo desde un menuitem, lo llamamos desde un keyword.

```
<record model="ir.action.keyword" id="act_marcar_presente_keyword">
    <field name="keyword">form_action</field>
    <field name="model">academia.sesion,-1</field>
    <field name="action" ref="act_marcar_presente_wizard"/>
</record>
```

2.10 ORM

Cuando estamos en un wizard, generalmente queremos acceder a los datos que tenemos en esa session:

```
session.data.start.presentes == session.data['start']['presentes'] son distinas formas de acceder a l
```

Para poder instanciar algún modelo debemos usar el ORM de Tryton. (import Pool)

```
from trytond.pool import Pool # ORM
```

El código para poder instanciar sería de la forma:

```
Pool() #es la forma que nos deja instanciar.
Pool().get('academia.sesion') #me trae un objeto instanciado de academia.sesion.
```

Métodos que probablemente use diariamente.

browse(): Devuelve Objeto o lista de objetos. El browse recive un ID o lista de IDS.

Transaction(): Contiene el contexto (el active_id, el language). context.get[’active_id’] -> Es el id o ids sobre el que tengo foco en la grilla de mi modelo. Este caso, Sesion.

write(): Simplemente escribe sobre los atributos del objeto que tengo. Es parecido a un update de SQL.

Ejemplos:

```
sesion_obj = Pool().get('academia.sesion').browse(Transaction().context.get('active_ids'))

alumno.write(alumno.id, dictionario con el nombre clave(atributo) y value  {'presente': True}).

alumno.create(dictionary con los valores del nuevo registro) #devuelve el id del nuevo registro que
alumno.search(lista de condiciones. Cada condicion es una tupla.  [('name','=','PEPE')]) # ver operac
alumno.delete(pasas id o ids)
alumno.read(id, nombre de campos que queremos leer) #lee
```

2.11 Proteus

Nos lo podemos bajar desde el sitio.

Trabaja sobre el nivel de aplicación. Lo importamos como si fuera un paquete trytond.

```
import sys
sys.path.append('path/al/trytond')
import proteus
from proteus import config, Model, Wizard

config #configuro la conexion contra la DB
Model #me deja manejar los modelos de trytond
```

Mirar documentacion de proteus. Contiene muchos muchos ejemplos. Por ejemplo, uno de instalacion de modulos usando proteus. Entonces se puede automatizar una puesta en marcha.

Note: No es necesario que el servidor este corriendo para ejecutar proteus

2.12 Reportes

La libreria relatorios de python es lo que usa tryton para hacer los llamados.

Desde el template ODT vamos hacer funciones for, if, else, y cuestiones que son del lenguaje **relatorio**. Esas sentencias de relatorio son las que me dan conexión con los objetos. Voy a tener una parte estática y una dinámica que la toma de parte del objeto.

Si queremos ver un ejemplo muy completo, mirar el módulo account.

2.12.1 XML

El XML va a necesitar tener un record keyword y un action ir.action.report

```
<record model="ir.action.report" id="act_report_sesion">
    <field name="name">Reporte 1</field>
    <field name="model">academia.sesion</field>
    <field name="report_name">academia.sesion</field>
    <field name="report">academia/sesion.odt</field>
    <field name="style">company/header_A4.odt</field>
</record>

<record model="ir.action.keyword" id="act_report_sesion_keyword">
    <field name="keyword">form_print</field>
    <field name="model">academia.sesion,-1</field>
    <field name="action" ref="act_report_sesion"/>
</record>
```

2.12.2 py

Si quiero el reporte en sesion, va a tener que heredar de CompanyReport (es la clase de tryton que me deja manejar reportes)

```
from trytond.modules.company import CompanyReport

class SesionReport(CompanyReport):
    _name = 'academia.sesion'

SesionReport()
```

Note: El en xml se define el tag report_name que debe ser igual al _name del modelo SesionReport

2.12.3 ODT

El lenguaje que usamos es el relatorio.

Primero es necesario acceder a todos los atributos de los objetos:

```
<FOR EACH="SESION IN OBJECTS">
# En este caso accedo a los objetos de sesion
```

El tag `for each` va a estar detntro de un campo de tipo **Marcador de posición**

¿Cómo inserto un campo desde LibreOffice?: Insertar->Campos->Otros->(tab funciones)Marcador de posicion->Texto

tag SETLANG: Viene a setear el lenguaje del reporte. Toma el lenguaje de mi party, sino usa EN_US

tag IF TEST="SALE.INVOICE_ADDRESS": Si tiene el campo INVOCE_ADDRESS

Ejemplo de tags que funcionen como **switch**:

```
<CHOOSE TEST>
<WHEN TEST="XXXX"> </WHEN>
<OTHERWISE> </OTHERWISE>
</WHEN>
```

tag FORMATLANG(SALE.SALE_DATE, PARTY.LANG, DATE=True): Formateo la fecha segun el idioma

¿Se puede imprimir a pdf, csv, etc?: Fijarse que libreria faltaría, pero se puede tocar desde la interfaz cliente.

Sino, fijarse desde el XML, agregar el **field report_extension="pdf"**

2.13 Traducciones

Se manejan con los archivos .po. Usar el poedit para editar los archivos .po

Tryton nos deja desde el cliente traducir la interfaz.

1. Primero se debe sincronizar desde los .po.
2. Cuando se ha terminado de traducir desde la interfaz, se puede exportar y pisar el archivo anterior.
3. En el archivo __tryton__ agregar la linea translation['locale/es_AR.po']

2.14 Workflow

Modelo workflow. Hereda de Workflow, ModelSQL, ModelView.

El workflow tiene una transición que va de un lado a otro.

1. Lo primero que se deben definir son las transiciones. Estos datos se debe definir en el __init__ heredo las transitions ya creadas de Workflow

```
def  __init__(self):
    super(Sesion, self).__init__()

    self._error_messages.update({
        'alumnos_out': 'No se pueden agregar mas alumnos. Numero de asientos!',
    })

    self._transitions |= set((      # Si existe te lo reescribe. NO podes tener una transition repeti
        ('draft', 'confirmed'),   # (origen, destino)
        ('confirmed', 'done'),  # Podria tener la de (done,confirmed)
        ('done', 'cancel'),
        ))

    self._buttons.update({
        'confirmed': {  ## Nombre del boton. Nombre de la function
            'invisible': ~Eval('state').in_(['draft']),
        },
        'done': {  ## Nombre del boton. Nombre de la function
            'invisible': ~Eval('state').in_(['draft']),
        },
    })
```

2. Debemos definir en el modelo el campo 'state'.

```
class Sesion(Workflow, ModelSQL, ModelView):
'Sesion'
_name = 'academia.sesion'
_description = 'Clase Tryton. Sesion'

state = fields.Selection([  # Se debe llamar SI o SI state
    ('draft','Draft'), #transition, label
    ('confirmed','Confirm'),
    ('done', 'Done'),
    ('cancel', 'Cancel'),
], 'State', readonly=True)

Sesion()
```

3. El valor del campo que va a tener el campo al abrir el formulario

Important: SI no definimos el state default, el workflow NO comienza

```
def default_state(self):
    return 'draft'
```

4. Nombre de la funcion que va a ejecutar cuando se apriete el boton confirmed Se debe llamar igual que el name="" del button en el xml/py

Argumentos: ids viene el id metido en una lista.

Lo que ejecutamos acá esta entre medio de un estado y el otro. Si faltan datos para concretar alguna operacion, y no puede pasar al estado siguiente tira una exception

```
@ModelView.button
@Workflow.transition('done') ## Se define hacia que transition queremos ir
def confirmed(self, ids):
    print "Call def confirmed" ## Estoy logueando en el servidor
    pass
```

Botones se debe definir en self._buttons. Por convención, las claves van a ser el nombre de los botones. Atributos de los botones. Lo usual es que sea invisible o no. También se deben definir los botones en la vista.

El botón definido en el XML quedaría de la forma:

```
<group col="5" colspan="4" id="buttons">
    <label name="state" />
    <field name="state" />
    <button name="confirmed" string="Confirmar" type="object"
    icon="tryton-go-previous" />
</group>
```

Important: El button en el xml DEBE tener un name. Necesito que tenga name ya que con eso asocio la function definida en el py.

Se debe agregar en un group, ya que los botones iran apareciendo y desapareciendo mientras pasemos de un state a otro. Para que los botones no aparezcan en cualquier lado, se los engloba en un group.

tag type => object: Llama a funcion de py

tag type => action: llama a una action del xml. Ejemplo, un action wizard, o un action report.

Como esto es python, podemos usar herencia. Usando herencia _name, definiendo la function del button y usando super(). En este caso, nos sirve para redefinir el button de **cancel** en sale.sale

2.15 Ideas que quedaron de la implementación Imprentas

1. Wizard loco. Capaz se puede hacer que segun condicion on_change_with hacer el tag invisible.
2. tag -> image name="" name: the name of the image. It must be the name with the extension of an image from tryton/share/pixmaps/
3. Mirar modulos pagos para ver esto de heredar el boton cancel de sale y agregarle un wizard para guardar el comentario.
4. Mirar funcion duplicate -> se puede heredar, y si quiero hacer algo en el medio. Sale pasa de confirmed a draft.
5. Margen de ganancia. La primera idea sería agregar campo "inflacion" y actualizar (on_change_with). Luego de sumar una linea, producto el total. Segunda idea es mirar el campo fields.Function (total). Herardar esa funcion, y sumarle si tengo algo en mi campo inflacion.

2.16 Tryton Web

2.16.1 Flask

Sacado de la web de tryton: https://code.google.com/p/tryton/wiki/TrytonFlask

Flask is a micro webdevelopment framework for Python. This howto will show how Tryton could be used with it. A Minimal Application

A minimal Flask application using Tryton looks something like this:

```
from flask import Flask
from trytond.pool import Pool
from trytond.transaction import Transaction

app = Flask(__name__)
app.config['TRYTON_DATABASE'] = 'test'

Pool.start()
Pool(app.config['TRYTON_DATABASE']).init()


@app.before_request
def before_request():
    Transaction().start(app.config['TRYTON_DATABASE'], 0)


@app.teardown_request
def teardown_request(exception):
    Transaction().stop()


@app.route("/")
def hello():
    pool = Pool()
    user_obj = pool.get('res.user')
    user = user_obj.browse(0)
    return "%s, Hello World!" % user.name

if __name__ == "__main__":
    app.run()
```

Just save it as hello.py (or something similar) and run it with your Python interpreter.

$ python hello.py

- Running on http://127.0.0.1:5000/

Now head over to http://127.0.0.1:5000/, and you should see your hello world greeting.

Note: Each function is responsible to commit to the database by calling: Transaction().cursor.commit()

2.16.2 Nereid

Nereid is a web framework built over Flask, with Tryton as a Backend.

http://nereid.openlabs.co.in/

Install

Instalando desde el repo github:

```
$ git clone git://github.com/openlabs/nereid.git
$ cd nereid
$ python setup.py install
```

O sino usando pip

```
$ pip install nereid
```