Outline

1. Overview and Installation
   - Tryton Overview
   - Prerequisites
   - Initializing tryton

2. A basic module

3. Some more advanced features

4. Additional topics
Global overview

- A general purpose application platform
  - Web shop
  - Insurance software
  - GNU Health
  - Traditional ERP
- Covers lot of fields
  - Accounting
  - Sales Management
  - Stock Management
  - ... 
- Free Software
  - GPL-3
  - Tryton foundation
  - TU*
Technical Overview

- Written in Python
- Three tiers architecture
  - PyGTK client (*tryton*) and a JavaScript client (*sao*) in its infancy
  - Application Server (*trytond*)
  - DBMS, usually *PostgreSQL* but you can use *SQLite* or even *MySQL*
- Modularity of business functionalities
Python good practices (pip & virtualenv)

**pip** is THE tool for installing and managing Python packages. **virtualenv** is THE tool used to create isolated Python environment.

- create isolated Python environments
  - Do not mix different version of your libraries / applications
  - Installation of packages in the user $HOME
- **pip install virtualenv** (or your distro package manager)
- **virtualenv --system-site-packages .venv/trytond**
- **source .venv/trytond/bin/activate**

**hgnested** is a mercurial extension. The tryton project use it to easily apply the same command on nested repositories.
Configuring mercurial

In order to activate both the `hgnested` and MQ extensions of Mercurial we will need to add those lines to your `.hgrc`

```plaintext
[extensions]
hgext.mq =
hgnested =
```
Installing trytond

Example

```
$ hg nclone http://hg.tryton.org/3.4/trytond -b 3.4
$ cd trytond
$ pip install -e .
$ pip install vobject
```
Setting up a trytond config file

Here is a minimal example of a configuration file. You should save it in $HOME/.trytond.conf

Example

```
[database]
path = /home/training/databases
```

We will set the TRYTOND_CONFIG environment variable

Example

```
$ export TRYTOND_CONFIG=$HOME/.trytond.conf
```
Initializing a minimal trytond database

Example

$ touch ~/databases/test.sqlite
$ ./bin/trytond -d test -u ir res

trytond will ask you for the admin password at the end of the installation process.
Adding a new modules

In this tutorial we will use a MQ repository in order to progress step by step.

Example

$ cd trytond/modules
$ hg init training
$ cd training/.hg
$ hg clone http://hg.tryton.org/training -b 3.4 patches
Outline

1. Overview and Installation

2. A basic module
   - A minimal module
   - Defining some object / tables
   - Different fields
   - Defining a tree and a form view
   - Default values and on_change calls

3. Some more advanced features

4. Additional topics
A minimal trytond modules

A minimal trytond modules needs two files:

- `__init__.py` the usual file needed by all python modules
- `tryton.cfg` the file that helps tryton glue together model and view definitions
A basic module

A minimal module

The content of tryton.cfg

Example

[tryton]
version=0.0.1
depends:
  ir
  res
Creating a model

A trytond model is a python class inheriting from `Model`. To enable the SQL persistence the model must inherit of `ModelSQL`.

Example

```python
from trytond.model import ModelSQL

__all__ = ['Opportunity']

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

N. Évrard (B²CK)
Register the model

In order for your object to be "known" by trytond they must be registered into the pool.

Example

```python
from tryton.pool import Pool
from .opportunity import *

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

Example

```python
class Opportunity (ModelSQL):
    '__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')
```
**fields arguments**

- **string**: A string for label of the field.
- **required**: A boolean if True the field is required.
- **readonly**: A boolean if True the field is not editable in the user interface.
- **domain**: A list that defines a domain constraint.
- **states**: A dictionary. Possible keys are `required`, `readonly` and `invisible`. Values are pyson expressions that will be evaluated with record values. This allows to change dynamically the attributes of the field.

The whole list in `trytond/model/fields/field.py`
**fields.Char**

In a tryton module:

```python
Example
description = fields.Char('Description', required=True)
```

In the interface:

**Example**

**Description:**

In SQL:

```sql
Example
description VARCHAR NOT NULL
```
fields.Date

In a tryton module:

Example

```
start_date = fields.Date('Start Date', required=True)
```

In the interface:

Example

```
Start Date:
```

In SQL:

Example

```
start_date DATE NOT NULL
```

**fields.Text**

In a tryton module:

```python
Example

comment = fields.Text('Comment')
```

In the interface:

```plaintext
Example
```

In SQL:

```sql
Example

comment TEXT
```
**fields.Many2One**

In a tryton module:

Example

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

In the interface:

Example

![Party:](image)

In SQL:

Example

```sql
party integer NOT NULL,
FOREIGN KEY(party) REFERENCES party_party(id)
```
Other relation fields

- fields.One2Many
- fields.Many2Many
- fields.One2One
Displaying data

To use the presentation layer your model must inherit from `ModelView`

**Example**

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

You must also add the xml presentation file in the `tryton.cfg` configuration file

**Example**

```xml
xml:
    opportunity.xml
```
Defining a view

- View objects are normal tryton objects (`trytond/ir/ui/view.py`)
- Two kind of view:
  - **Tree view** a list of record
  - **Form view** a view for editing/creating one record
A tree view

Example

```
<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>
```

Example

```
<tree string="Opportunities">
    <field name="party"/>
    <field name="description"/>
    <field name="start_date"/>
    <field name="end_date"/>
</tree>
```
A form view

Example

```xml
<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>
```

Example

```xml
<form string="Opportunity">
    <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>
```
Gluing them together

Example

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

<record model="ir.action.act_window.view"
  id="act_opportunity_form_view1">
  <field name="sequence" eval="10"/>
  <field name="view" ref="opportunity_view_tree"/>
  <field name="act_window" ref="act_opportunity_form"/>
</record>

<record model="ir.action.act_window.view"
  id="act_opportunity_form_view2">
  <field name="sequence" eval="20"/>
  <field name="view" ref="opportunity_view_form"/>
  <field name="act_window" ref="act_opportunity_form"/>
</record>
```
Adding menu entries

Example

```xml
<menuitem name="Training" id="menu_training" />
<menuitem parent="menu_training"
    action="act_opportunity_form"
    id="menu_opportunity_form"/>
```
Adding default values

Create a method in the object with the name `default_<field_name>`

Example

```python
@staticmethod
def default_start_date():
    pool = Pool()
    Date = pool.get('ir.date')
    return Date.today()
```
Reacting to user input: \texttt{on\_change}

Tryton provides a way to react on the user input by changing the value of other fields. This mechanism is called the \texttt{on\_changes}.

They come in two flavours:

\texttt{on\_change\_<field name>} When a field is changed a list of values is sent to the server. The server sends the new values of some fields.

\texttt{on\_change\_with\_<field name>} A field value is computed when any field in a list of fields is modified. The computation occurs on the server.

In both cases, the list of fields sent to the server is specified thanks to the decorator \texttt{@fields.depends}.
Example

```python
@fields.depends('party')
def on_change_party(self):
    address = None
    if self.party:
        address = self.party.address_get(type='invoice')
    return {
        'address': address,
    }
```
on_change_with

Example

```python
@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
```
Outline

1. Overview and Installation

2. A basic module

3. Some more advanced features
   - Workflows
   - More beautiful views
   - Function fields
   - Wizards
   - Extending pre-existing tryton objects

4. Additional topics
Adding workflow to object

Your object should inherit from Workflow

Example

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

It must have a state field:

Example

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

A workflow is composed of transitions:

Example

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

Each transition must be linked a class method:

Example

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

Now we need to add buttons in the view

Example

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

The method must be "button decorated" to be callable and defined.

Example

```python
@classmethod
def __setup__(cls):
    ...
    cls._buttons.update({
        'convert': {},
        'lost': {}})

@classmethod
@ModelView.button
@Workflow.transition('converted')
def convert(cls, opportunities):
    ...
```
Let’s add the state and the buttons to the opportunity view. There is also a way of grouping widgets

**Example**

```xml
<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>
```
Adding dynamicity in the form

Of course sometimes you want to make fields readonly/invisible/required under certain conditions. This behavior can be implemented using the `states` attribute of fields:

**Example**

```python
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** is used to encode statements that can be evaluated.

http://doc.tryton.org/2.6/trytond/doc/topics/pyson.html
Adding dynamicity to buttons

**PYSON** expression can also be used on buttons to hide them.

**Example**

```python
@classmethod
def __setup__(cls):
    ...
    cls._buttons.update({
        'convert': {
            'invisible': ~Eval('state')
                .in_(['opportunity']),
        },
        'lost': {
            'invisible': ~Eval('state')
                .in_(['opportunity']),
        }
    })
```
Defining a simple function field

A function field is a field that is computed into python its data is not persistently store into the database.

Example

```python
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
```

Any tryton type can be used. Note also that since this signature is the same as the one of an `on_change_with` then the same function can be used for both. Of course a `setter` and a `searcher` can also be defined in order to modify or search corresponding data.
Defining a function field operating by batch

The previous example makes one call per record. Tryton provides a way to compute the values by batch. In order to do so the getter must be a **classmethod** and it must return a dictionary mapping the id to the function value.

**Example**

```python
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())
```

Adding actions to model

Sometime you want to add functionalities to a model that do not suite the use of a button. For this kind of use case the wizard is your solution. A wizard is composed of two things:

- A set of views that represent the form to gather the user input
- A "state machine" that define what should be done
Wizard views

Those are standard ModelView, you can define on_change, on_change_with and default values on them.

Example

class ConvertOpportunitiesStart (ModelView):
    'Convert Opportunities'
    __name__ = 'training.opportunity.convert.start'
Some more advanced features

Wizards

Wizard "state machine"

This is a class that inherits from Wizard. You’ll be able to define different states on it.

Example

```python
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'
```

N. Évrard (B2CK)
Tryton Technical Training
September 18, 2015 45 / 56
Activating the wizard

Example

```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,#_</field>
    <field name="action" ref="act_convert_opportunities"/>
</record>
```
Sometimes you want to extend existing objects to add miscellaneous information. Doing so is just a matter of (tryton) inheritance:

**Example**

```python
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')
```

N. Évrard (B₂CK)
Extending existing views

Modifying existing view is done by adding record in the XML files that specify an XPath and what to do with the resulting node.

Example

```xml
<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>
```

Example

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

1. Overview and Installation
2. A basic module
3. Some more advanced features
4. Additional topics
   - Reporting
   - proteus
   - Table Queries
Adding a report

To add a report you must create an odt template that is compatible with relatorio (http://relatorio.openhex.org/)
This report engine allows you to create opendocument files from templates.
Additional topics

Reporting

Making trytond aware of the report

You must define a python class that will register a report object

Example

```python
from trytond.report import Report
class OpportunityReport(Report):
    pass
```

You might want to overload the `parse` method to define a specific behavior.
Creating the report object

Example

```xml
<record model="ir.action.report" id="report_opportunity">
    <field name="name">Opportunity</field>
    <field name="model">training.opportunity</field>
    <field name="report_name">training.opportunity</field>
    <field name="report">training/opportunity.odt</field>
</record>

<record model="ir.action.keyword" id="report_opportunity_keyword">
    <field name="keyword">form_print</field>
    <field name="model">training.opportunity</field>
    <field name="action" ref="report_opportunity" />
</record>
```
Using proteus to script tryton

You might want to script your tryton server. For this there is the **proteus** library. It will allow you to interact with it in a python way.
With models you will be able to use the find, edit, delete and create objects
You will also have the possibility to use the wizards.
Setting up proteus

To set up proteus, you first need a connection to a Tryton server:

- either through XML-RPC using the `set_xmlrpc` method
- or on the same compute using the `set_trytond` method

The latter will use the possibility to import trytond to create and set up an internal trytond server.
An example of proteus

Example

```python
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:
            party = Party(name=line['Party'])
            party.save()
        else:
            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)
```

N. Évrard (B2CK)
Creating a 'view' object

`table_query` allows you to define a SQL-query to use as source of the records.

Example

```python
class OpportunityByParty(ModelSQL, ModelView):
    'Opportunities by Party'
    __name__ = 'training.opportunity_by_party'

    party = fields.Many2One('party.party', 'Party', readonly=True)
    opportunity_count = fields.Integer('Opportunity count', readonly=True)

    @classmethod
    def table_query(cls):
        pool = Pool()
        opportunity_table = pool.get('training.opportunity').__table__()
        columns = [
            Min(opportunity_table.id).as_('id'),
            Max(opportunity_table.create_uid).as_('create_uid'),
            Max(opportunity_table.create_date).as_('create_date'),
            Max(opportunity_table.write_uid).as_('write_uid'),
            Max(opportunity_table.write_date).as_('write_date'),
            opportunity_table.party,
            Count(opportunity_table.id).as_('opportunity_count'),
        ]
        group_by = [opportunity_table.party]

        return opportunity_table.select(*columns, group_by=group_by)
```

N. Évrard (B2CK)