Tryton Documentation

Release 0.1

Tryton Community

Jun 02, 2017

Contents

Contents:

CHAPTER 1

User Guide

Contents:

Basic Concepts

Tryton is a three-tier high-level general purpose computer application platform on top of which is built an Enterprise resource planning (ERP) business solution through a set of Tryton modules. The three-tiers architecture consists of the Tryton client, the Tryton server and the Database management system (mainly PostgreSQL).

Technical Features

The client and the server applications are written in Python, the client use GTK+ as graphical toolkit. Both are available on Linux, OS X. The kernel provides the technical foundations needed by the most business applications. However it is not linked to any particular field hence constituting a general purpose framework:

- Data persistence: ensured by accessor objects called Models, they allow easy creation, migration and access to records.
- User Management: the kernel comes with the base features of user management: user groups, access rules by models and records, etc.
- Workflow Engine: allows to activate a workflow on any business model.
- Report Engine: the report engine is based on relatorio that uses ODT files as templates and generate ODT or PDF reports.
- Internationalisation: Tryton is available in English, French, German, Spanish,and Italian. New translations can be added directly from the client interface.
- Historical data: data historization may be enabled on any business model allowing for example to get the list of all the past value of the cost price of any product. It also allows to dynamically access historized record at any time in the past: for instance the customer information on each open invoice will be the ones of the day the invoice was opened.

- Support for DAV protocols: WebDAV, CalDAV, and CardDAV. This allow out-of-the-box document management and synchronizations of calendars and contacts.
- Support for XML-RPC and JSON-RPC protocols.
- Database independence is allowed since the 1.2 series and is used in the 1.4 series for the SQLite backend.

Being a framework, Tryton can be used as a platform for the development of various other solutions than just business ERPs. A very prominent example is GNU Health, a free Health and Hospital Information System based on Tryton.

License

The platform, along with the official modules, are Free Software, licensed under the GPLv3.

Product

Basic idea about products

Party

A party can be a person, a company or any organisation that one want to consider as the same entity. A party is defined by a name, a code, a language, a VAT code, categories, contact mechanisms and a list of addresses.

Using the Tryton Desktop Client

Tryton is a desktop client application, which means we need to know the server url and the port to connect to it. We also need to know the database that we are going to work on. There is also the possibility of creating a new database if needed.

Getting the Tryton Desktop Client

The desktop client can be downloaded from the downloads page of the Tryton website.

> Warning: To be able to connect to a server you must use a client version of the same series as server. So if your server is series 3.0 you must use 3.0 series client.

Connecting to your server

The first time you open the Tryton client you will see the following screen.

In this window you can enter the connection parameters of your server:

- yourserver: It's your server URL. Something like demo.tryton.org
- port: It's the port where your server is running. The default value is 8000
- database: It's the name of the database you will work on.

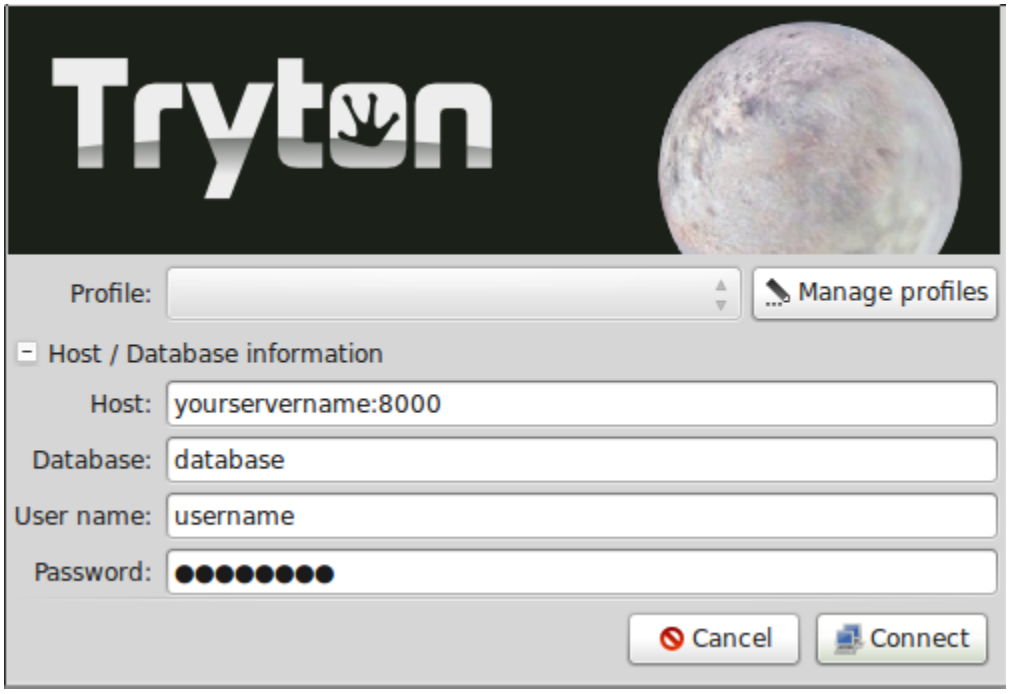


Fig. 1.1: Tryton's client login window

- user: Introduce the username you want to use for connecting.
- password: Introduce your password to connect to the server.

Managing profiles

The Tryton client has the ability of managing connection profiles, so you only have to introduce the details of the server only one time. If you click the Manage Profiles button you will see the following screen:

In the left part of the screen, you will see all the available profiles. You can add a new one by using the Add button. You can also delete old profiles with the Remove button. Use the right part of the screen to modify current profiles or to add the details of new created profiles.

The client will try to connect to the server to load the databases. It will fail if you can't connect to the server.

You can open the Tryton client numerous times. This is useful when you need to access diferent servers or databases at the same time.

Password window

For security reasons the client will ask you to enter your password, if your screen has been inactive for some time.

Client options

You can customize the behaviour of your client by selecting the options menu. The most common options are:

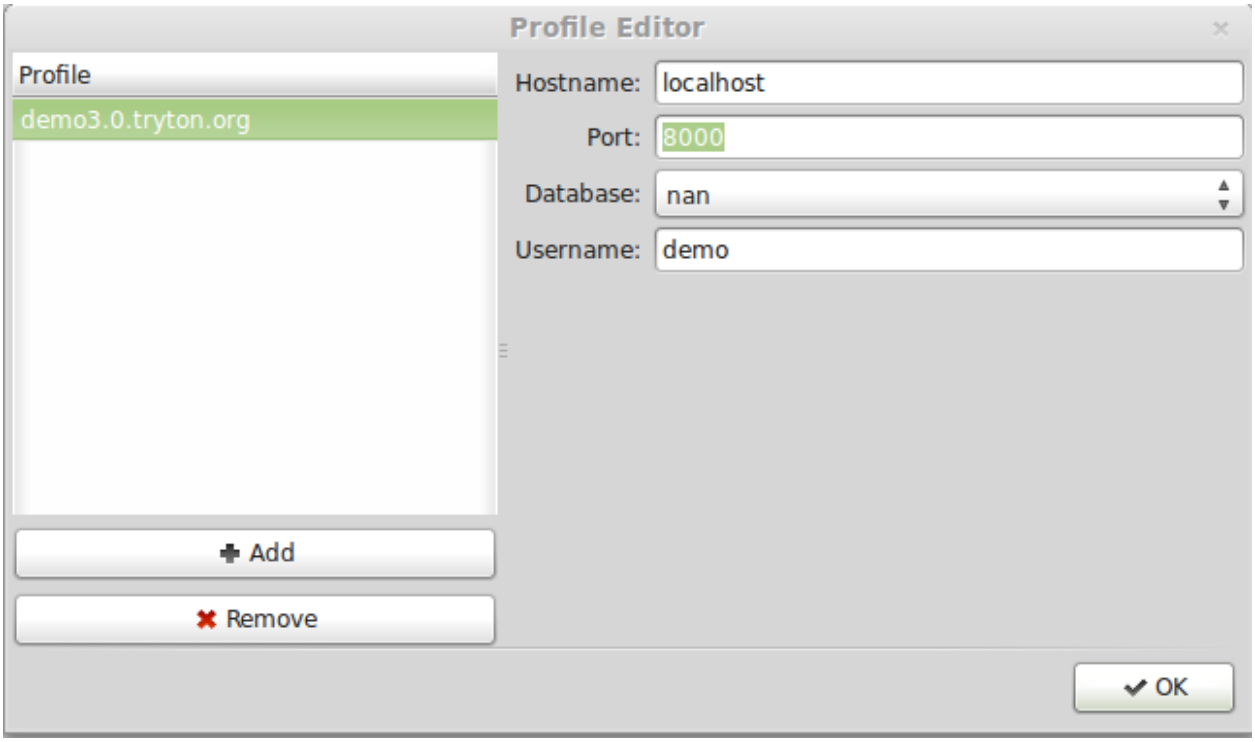


Fig. 1.2: Tryton connection profiles.

- Toolbar: Used to change the toolbar view. You can select to show only icons, only text and text and icons. The default value will show only icons.
- Form: Used to change the view of the Tryton client. The following options are available:
  - Save Width/Height: If marked, the client will remember the width of the columns if you modify it.
  - Save Tree Status: If marked, the client will remember the opened nodes in tree view.
  - Tabs positions: You can define where the forms tabs are shown. Possible values are: Top, Left, Right, Bottom.
- Search Limit: Used to change the number of records the client will search for. The default value is 1000.
- Email: You can introduce the command that will be executed when you use the send report by email option.

Toolbar options

In every Tryton tab the following toolbar option is available.

You can change toolbar view, just go to "option" on the top of the screen and select the toolbar options and select one from the following:

- Default
- Text and icons
- Icons
- Text

Text and icon view can be seen below:

Toolbar icons

They are distributed in four groups:

- Edition: Used to edit records.
- Navigation: Used to navigate through the records.
- Action: Used to interact with the records.
- Report: Used to generate reports.

In the Edition group the following icons are available (in order):

- New: Used to create a new record.
- Save: Used to save changes in the current record. If you haven't changed the current record it will be disabled.
- Switch: Used to switch the current view of the data.
- Reload: Used to refresh the data.

In the Navigation group the following icons are available (in order):

- Previous: Go to previous record.
- Next: Go to next record.

In the Action group the following icons are available (in order):

- Attachment: Used to show attachments of the records.
- Action: Used to perform some action on the selected records.
- Relate: Used to navigate to relate info of the record. For instance, if you're on a Party record you can use this button to show all the invoices related to this party.

In the Report group the following icons are available (in order):

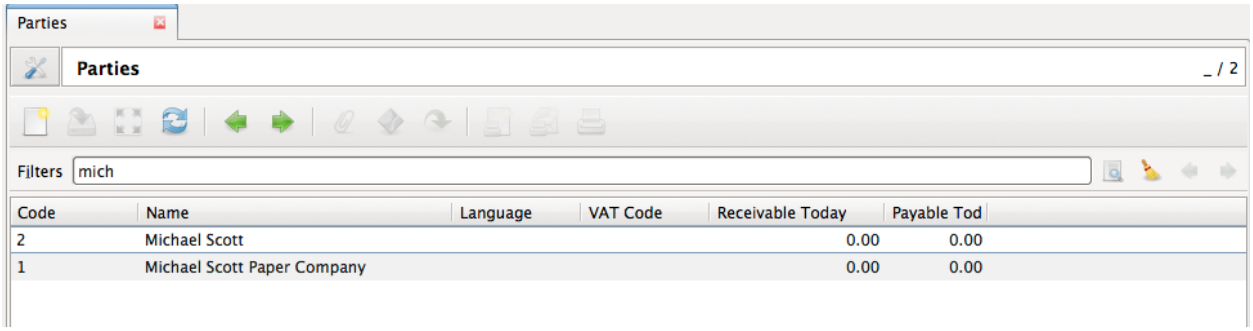
- Report: Generate a report and show it on the screen.
- Email: Will generate a report and open your email client to send it.
- Print: Print a report directly on a printer.

Searching

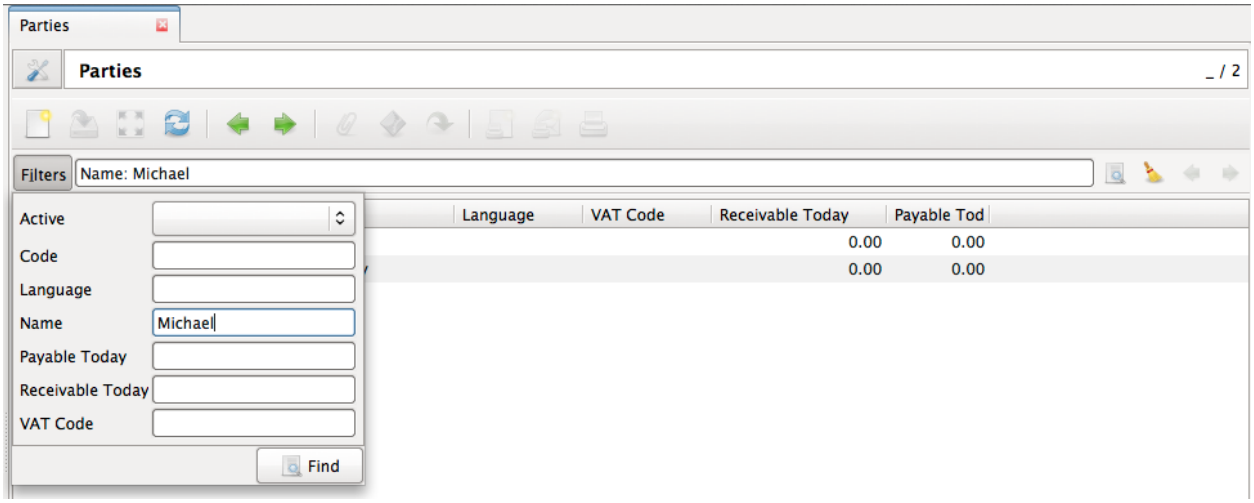
A query is composed to search clauses. A clause is composed of a field name (with : at the end), an operator and a value. The field name is optional and defaults to the record name. The operator is also optional and defaults to a case insensitive search on the name of the record.

Examples:

mich



Name: Michael



Operators

The following operators can be used:

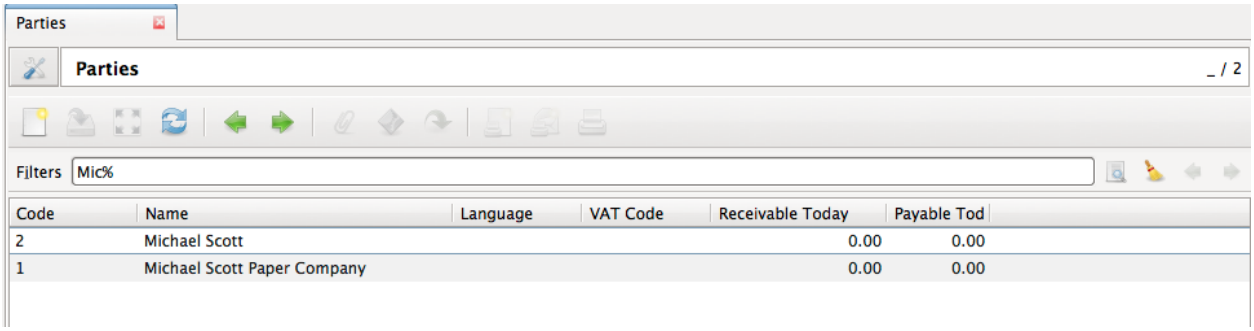
- =: equal to
- <: less then
- <=: less then or equal to
- >: greater then
- >=: greater then or equal to
- !=: not equal
- !: not equal or not like (depending of the type of field)

For example: Name: != Dwight

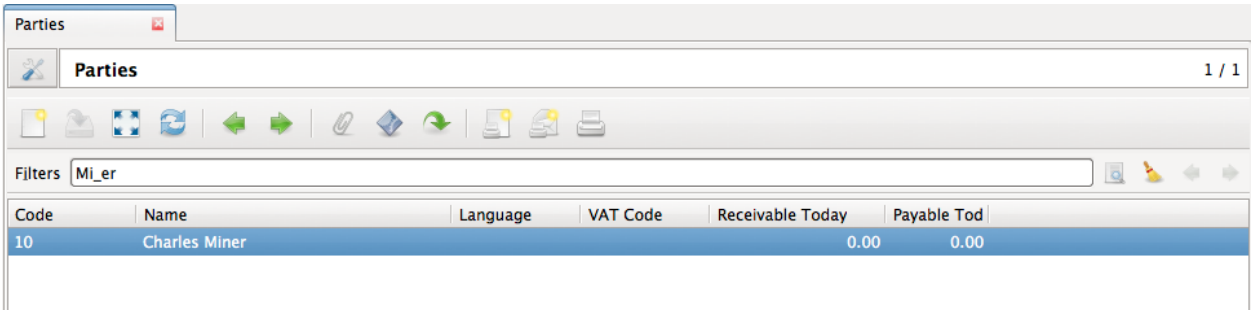
Wildcards

There are two wildcards:

%: matches any string of zero or more characters



'_': matches any single character



It is possible to escape special characters in values by using double quotes. For example: Name: "Michael:Scott" Here it will search with the value Michael:Scott.

Clause composition

The clauses can be composed using the two boolean operators and and or. By default, there is an implicit and between each clause if no operator is specified.

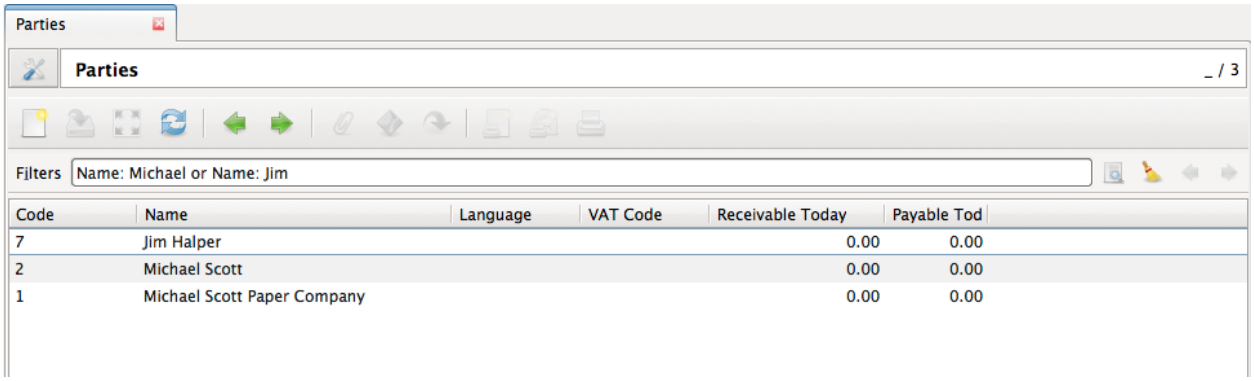
For example:

```
Name: Michael Amount: 100
```

is the same as:

```
Name: Michael and Amount: 100
```

Example using or



The and operator has a highest precedence than "or" but you can change it by using parenthesis.

For example:

```
(Name: Michael or Name: Pam) and Amount: 100
```

is different than:

```
Name: Michael or Name: Pam and Amount: 100
```

which is evaluated as:

```
Name: Michael or (Name: Pam and Amount: 100)
```

Tips and Tricks

- Refresh button: If you want to discard your changes on a record you can use the refresh button to reload server values.
- Favorites: You can save you favorites menu options by clicking on their star. Your favorites will be shown in the Favorites menu of the client.
- Global search: The text box in the top of the menu option is used to perform a text search on all global search enabled models. You can also access the global search by typing Ctrl+K from any place of the Tryton client
- Saving searches
- Hide the menu
- Right clicking on tree view

Attachments

On Tryton you can add attachments to any record. You can use the clip icon to attach some documents in a record.

The clip icon will show you yellow mark if the record has attachments.

How to get Help

Mailing List

tryton@googlegroups.com

Most of the tryton community follows this mailing list. You will usually be able to find people willing to help you, provided you follow some simple rules:

- Use english. There are language specific mailing lists, and posting in spanish or french on the standard mailing list will irritate and pollute those who do not understand the language.
- Be descriptive. Do not hesitate to give plenty of details on your problem. People will not be able to help you if the only thing they read is "I have a problem".
- Follow the Netiquette. At the very least, be polite and avoid top-posting.

IRC (Chat)

You can find us on the Freenode IRC. You can access with it your browser on Freenode WebChat.

We have different channels depending on the language you are comfortable with.

- English: #tryton (#tryton Archive)
- Deutsch: #tryton.de (#tryton.de Archive)
- Español: #tryton-es (#tryton-es Archive)
- Français: #tryton-fr (#tryton-fr Archive)
- РУССКИЙ: #tryton-ru (#tryton-ru Archive)

All the conversations are archived and you can browse it on the tryton website.

IRC is a great way to get support from the tryton community. People are usually responsive, and willing to help. Again, some rules are necessary for it to work properly:

- Do not copy-paste huge chunks of text in the chat, it is annoying and makes it unreadable. Use an online temporary hosting for it like Debian Paste or Pastebin is more convienent for everyone.
- People on the chat have no way to know what you are wanting to do. Be exhaustive in your explanation. Explain what your are trying to do, and what it is that makes it not possible.
- If somebody answers you, it's always good to write his/her nickname when you are referring to him. It helps getting faster responses as they will have a notification on their client. It also helps to clarify conversations because sometimes there are more than one conversation in the same time.
- Understand that people in the chat may have other things to do. We try to help, but most of us are working and may not be immediately available.

What can I do if nobody answers my question?

If no-one answers you after a few minutes (30 minutes), you should probably try to reach a bigger audience to get the question replied. If you asked on a localized channel it is always to ask the question to the main channel (the english one). If you have already asked your question on the main channel it is always a good idea to send a mail to the mailing list and wait for the reply.

CHAPTER 2

Installation and Configuration Guide

Contents:

Quick install guide

Before you can start using Tryton, you'll need to get it installed. We have a detailed installation guide that covers all the possibilities; this guide will guide you to a simple, minimal installation that'll work while you explore, test or evaluate tryton.

Components

Tryton is separated into three different parts:

1. The tryton server named trytond (d for daemon)
2. The tryton desktop client.
3. The modules that extend or implement features on server (like inventory management, sales management, accounting, invoicing etc.)

Neso

Neso is a standalone suite which includes the server, client and modules and uses a light weight database called SQLite. Neso is a great way to explore tryton and using Tryton for a single user.

You could download the installer for windows or install it from source.

Install Python

Being a python based system, Tryton requires Python to run. Tryton works with Python 2.7. Python 2.7 also includes a light weight database called SQLite so you won't need to set up a database just yet.

You can verify that Python is installed by typing python from your shell; you should see something like:

```
Python 2.7.5 (default, Oct  8 2013, 17:30:18)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.75)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Setup a database

This step is only necessary if you would like to work with a more powerful database engine like PostgreSQL or MySQL. To install such a database consult the Basic Database Configuration section. If all you want is to get started quickly on tryton and use SQLite as a database, add the following to your trytond.conf (normally /etc/trytond.conf):

```
db_type = sqlite
data_path = /var/lib/trytond
```

and make sure data_path is an existing directory

Install Tryton

You've got three easy options to install Tryton:

- Install a version of Tryton provided by your operating system distribution. This is the quickest option for those who have operating systems that distribute Tryton.
- Install an official release. This is the best approach for users who want a stable version number and aren't concerned about running a slightly older version of Tryton.
- Install the latest development version. This is best for users who want the latest-and-greatest features and aren't afraid of running brand-new code.

Verifying

To verify that Tryton is installed, type trytond --version from your shell; you should see something like:

```
trytond 3.2.1
```

This indicates the version of Tryton installed.

Installation Procedure

Tryton is separated into independent parts:

- the server named trytond
- the GTK client named tryton
- and several modules to extends server capabilities (ie: account, bank, party, project...)

Installation instructions are slightly different depending on whether you're installing a distribution-specific package, downloading the latest official release, or fetching the latest development version.

It's easy, no matter which way you choose.

Installing from system distributions

Many third-party distributors are now providing versions of Tryton integrated with their package-management systems. These can make installation and upgrading much easier for users of Tryton since the integration includes the ability to automatically install dependencies (like database adapters) that Tryton requires.

If you're using Linux or a Unix installation, such as Ubuntu, check with your package manager if they already package Tryton. The Tryton Wiki also contains instructions for installation on several distributions.

Specific packages are available for Windows and MacOSX, which can be downloaded from the tryton download page.

Tip: While most packages are based on the latest stable versions of Tryton, if the version you need is not available, you'll need to follow the instructions for installing an official release or development versions.

Installing an official release with pip

This is the recommended way to install Tryton. The packages are downloaded automatically from the Python package index (PYPI).

1. Install pip. The easiest is to use the standalone pip installer. If your distribution already has pip installed, you might need to update it if it's outdated. (If it's outdated, you'll know because installation won't work.)
2. (optional but recommended) Take a look at virtualenv and virtualenvwrapper. These tools provide isolated Python environments, which are more practical than installing packages systemwide. They also allow installing packages without administrator privileges. It's up to you to decide if you want to learn and use them.
3. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command sudo pip install trytond at the shell prompt. If you're using Windows, start a command shell with administrator privileges and run the command pip install trytond. This will install Tryton server in your Python installation's site-packages directory.

   Similarly, you can install the Tryton client with the command pip install tryton and modules with pip install trytond_MODULE_NAME

Tip: If you're using a virtualenv, you don't need sudo or administrator privileges, and this will install Tryton in the virtualenv's site-packages directory.

Tip: You might be interested by a list of available modules.

Useful pip command combinations

- Ensure that PIP is the latest version:

```
pip install -U pip
```

- Installing the server

```
pip install trytond
```

- Installing the GTK client

```
pip install tryton
```

- Installing any module for server

```
pip install trytond_MODULE_NAME
```

Replace MODULE_NAME with the name of the module. Example below.

```
pip install trytond_sale
```

remember that it installs the most recent released version, as of this writing that is the 3.2 series.

- Installing the most recent version from a previous series:

```
pip install "trytond_sale>=3.0,<3.1"
```

Installs latest version in the 3.0 series

- Upgrading a module

```
  pip install -U trytond_sale

remember that this would install the most recent series which as of this
writing is 3.2
```

- Upgrading within the same series

```
pip install -U "trytond_sale>=3.0,<3.1"
```

- Upgrade only the module, without any dependencies

```
pip install -U --no-deps "trytond_sale>=3.0,<3.1"
```

- Installing a module from source code

```
pip install /path/to/module/folder
```

- Forcefully reinstall a module

```
pip install -U --force trytond_sale
```

Installing an official release manually

1. Download the latest release from our download page.
2. Untar the downloaded file (e.g. tar xzvf trytond-X.Y.Z.tar.gz, where X.Y.Z is the version number of the release). If you're using Windows, you can download the command-line tool bsdtar to do this, or you can use a GUI-based tool such as 7-zip.
3. Change into the directory created in step 2 (e.g. cd trytond-X.Y.Z).
4. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command sudo python setup.py install at the shell prompt. If you're using Windows, start a command shell with administrator

privileges and run the command python setup.py install. This will install Trytond in your Python installation's site-packages directory.

Removing an old version

If you use this installation technique, it is particularly important that you remove old versions of Tryton. TODO: explain how

---

Installing from source

You can browse the Source Code Repository <http://hg.tryton.org/> and download source code thanks to your favorite version control system:

1. Make sure that you have Git or Hg installed and that you can run its commands from a shell. (Enter git help or hg help to test this).
2. Checkout the development branch of the source code from the VCS (See examples below).
3. Make sure that the Python interpreter can load the downloaded code. The most convenient way to do this is via pip. Run the following command:

```
sudo pip install -e trytond-trunk/
```

(If using a virtualenv you can omit sudo.)

> Warning: Don't run sudo python setup.py install, because you've already carried out the equivalent actions in step 3.

Official mercurial repos

- Get server source code

```
hg clone http://hg.tryton.org/trytond/ trytond-trunk  # For the server
```

- Get GTK client source code

```
hg clone http://hg.tryton.org/tryton/ tryton-trunk  # For the client
```

- Get official modules source code

```
hg clone http://hg.tryton.org/modules/MODULE_NAME MODULE_NAME-trunk
```

You might be interested by a list of actual module repositories <http://hg.tryton.org/modules>.

When you want to update your copy of the Tryton source code, just run the command hg pull -U from within the corresponding directory. When you do this, Mercurial will automatically download any changes.

Unofficial Git mirror

And up-to-date, but non-official git repositories are maintained on github:

- Get server source code

```
git clone https://github.com/tryton/trytond.git trytond-trunk
```

- Get GTK client source code

```
git clone https://github.com/tryton/tryton.git tryton-trunk
```

- Get official modules source code

```
git clone https://github.com/tryton/MODULE_NAME.git MODULE_NAME-trunk
```

When you want to update your copy of the Tryton source code, just run the command git pull from within the corresponding directory. When you do this, Git will automatically download any changes.

Preparing Application Servers

TODO

Basic Database Configuration

Postgres is the recommended database engine for tryton Install Postgres database. Steps for installing Postgres can be found from Postgres Installation Install the database and give a new password to the postgres database user.

Access Control & management

1. For models/fields which have no access rules, everyone by default has access.
2. All users are subject to access rules that have group left blank. Users are also subject to access rules that apply to groups to which they belong.
3. If contradictory access rules apply to a single user, the most permissive applies.
   (a) Example: If a global (group blank) access rule denies access, but a group-specific access rule grants access, then access is allowed.
   (b) Example: If a global (group blank) access rule allows access, then everyone will have access; no group-specific rules can block it. (Thus, globally granting access is a bad idea. Better to just leave the target alone, in which case the default rule will apply, permitting access.)
4. Access "rules" can be set for Groups. These make access dependent on whether the circumstances meet administrator-specified tests. Using these may hurt system speed.

Menuitems Access

These simplify the interfaces for affected users by hiding menuitems that they have no need to access. They do not, however, necessarily keep data safe. If model 'X' is used as an associative field of model 'Y', then a user with access to model 'Y' can access model 'X' through a pop-up, even if the menuitem to access model 'X' is hidden. Further, a sophisticated user could access the Tryton server with a tool other than the Tryton client and thereby have access to such hidden data.

Database management

Install a database of your choice. Here Posgres database is taken as sample database. Steps for installing Postgres can be found from Postgres Installation Install the database and give a new password to the postgres database user.

Creating Database

TODO

Upgrading a Database

To upgrade the database there are 2 ways to do that:

- By typing the command:

```
$ trytond -c ../etc/trytond.conf -u module_name -d database_name
```

- By using tryton client and connecting with your database. On left bar Under Modules > Modules. There you will see the modules installed on your tryton client along with the modules that are not in installed state. You can simply click on Mark for Installation for the module you wish to install and then click on Launch action to Perform Pending Installation/Upgrade. All the modules that you mark for installation will be installed.

Installing Modules

Each tryton database can be customized depending on the modules that are installed on it. A module is set of files that adds new functionalities to a tryton database.

A module can do one (or more) of the following things:

- Create new models
- Create new fields on existing models
- Create new views
- Modify existing views

To install a new module you can go to the Administration/Modules menú which contains a list of all the available modules on the server. To install a new module, you can push the Mark for install button. When you have finished selecting all the modules you want to install, you must use the Action button on the toolbar to launch the Perform Pending Installation/Upgrade wizard.

Once the wizard is finished, the new modules should be available on your database.

> Warning: Some modules add new groups, so in order to see their options you will have to modifiy your user to add it to the newly added groups.

Uninstalling Modules

Although tryton has the option to uninstall a module on a database, occasionally it might not work as expected, as is still in the beta status. We recomend you to avoid uninstalling modules as much as possible.

If you want to test a module, you can install it on a test database and then install it on the production database if that module fits your needs. If you need some data to test your module you can create a copy of your production database and restore it with a diferent name.

Backup & Restore

The Tryton client allows you to create and restore backup of your databases.

To backup/restore databases, you will need the admin_passwd that was configured on your server. The default value for this password is admin.

Backup a database

Open your Tryton client and go to File/Databases/Backup Database. You will see a screen, which will display the following details:

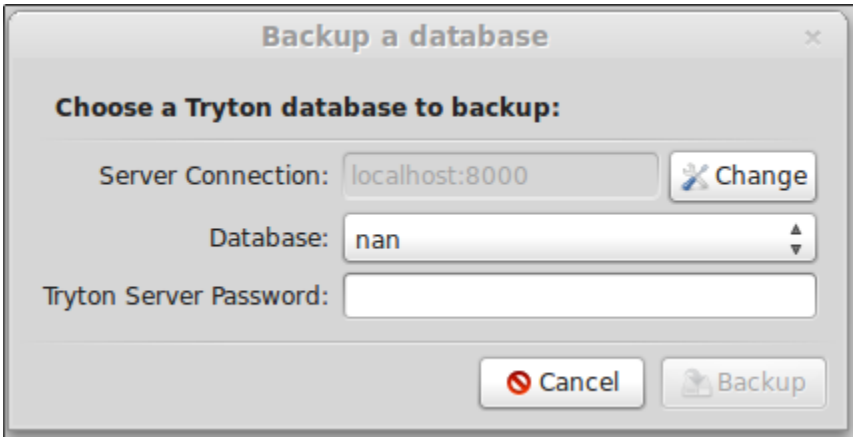


Fig. 2.1: Tryton's client backup option

Once you have selected one database from your Tryton server and entered the admin password the backup will be triggered. When the backup finishes, you will be asked to select a folder to save the backup.

All the backups generated with the client can also be restored by the client.

Restore a database

To restore a backup generated with the tryton client you must open your Tryton client and go to File/Databases/Restore Database. You will be promted to enter the name of the backup file you want to restore. Once the name is entered, you will see a screen which displays the following details:

You must enter your server password and the database name to the new restored database.

If you don't want to upgrade the database after restoring you must unselect the Update Database option. We encourage you to keep this option selected as it is always a good idea to update your database after restoring.

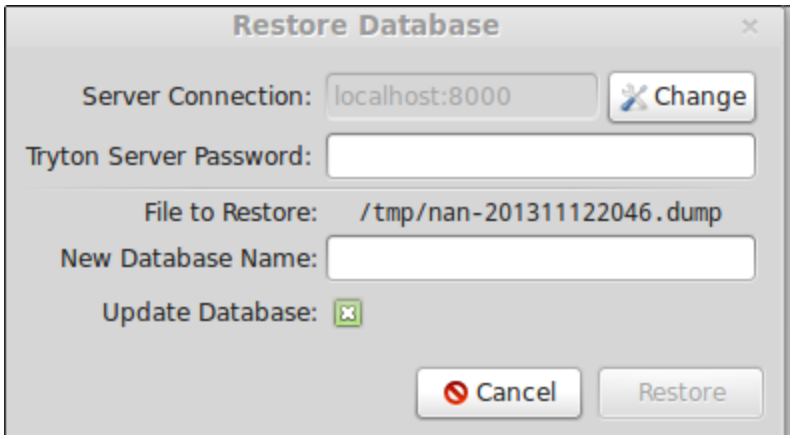


Fig. 2.2: Tryton's client restored option

Monitoring Tryton

Watching for Errors

Sentry

Instrumentation

Instrumenting and monitoring requests

Logging

Logging for production

Migration

The migration between tryton versions is fully suported. In order to migrate an existing database to a newer version you must take the following actions:

1. Obtain the new version of trytond and all the instaled modules. You can update your existing installation or create a new one.
2. Update the database with trytond -u all -d <database>, where <database> refers to your database name.

Migrating custom modules

If you have developed custom modules it's possible that you have to adapt your code in order to get it working with the new tryton version.

Normally, there is an entry on the wiki with the changes that must be done and an example of the change in one official module. You can find this entries under the Release process entry.

Data Storage

TODO

Attachments

How are attachments stored

Using Amazon S3

TODO

CHAPTER 3

Developer Guide

Getting Started

Setting up a development environment

In the previous chapter you have seen how to install tryton client on your machine. Now, lets start with setting up the development environment for tryton.

Steps for setting up a development environment for tryton.

1. First thing is to have Python2 running on your system. You can check if python is installed by typing python2 in a console window and pressing Enter. If python is not installed, consult your package manager for your operating system. Or go to www.python.org and select "Downloads" If python is installed an interactive shell will open, printing out version info and >>> You can exit by calling the function exit() or by pressing ctrl+d
2. Next, Install Posgres database. Steps for installing Postgres can be found from Postgres Installation Install the database and give the database user postgres a new password.
3. Set up the virtual environment and install tryton client and trytond. You can directly install Tryton using pip command-line tool in your virtualev.

   ```
   $ pip install trytond
   $ pip install tryton_module_name
   ```

   Replace module_name with the name of the module you want to install

Using HgNested

HgNested is a mercuarial extension used to work on nested repositories. In order to install hgnested you must run:

```
pip install hgnested
```

don't forget to add the hgnested extension to your `~/.hgrc`:

```
[extensions]
hgnested =
```

Once installed you can checkout the lastest sources by executing:

```
hg nclone http://hg.tryton.org/trytond
```

This will also clone all the modules in trytond/modules.

In order to run the tryton server you must install all the requirements as described on the Wiki.

After this you can run your server with:

```
trytond/bin/trytond
```

Using virtualenvwrappers templates

The Openlabs guys have created a virtualenvwrapper template to get a virtualenv created with all the required dependencies (except your database drivers).

You can install it executing:

```
pip install virtualenvwrapper.tryton
```

You can create a virtualenv with the latest version of tryton running:

```
mkproject -t tryton virtualenv_name
```

In the virtualenv you can run your tryton server by executing:

```
trytond
```

For more information about available templates please refer to virtualenvwrapper.tryton repository

Basic Concepts

Models

Model([id[,**kwargs]]) This is the base class that every kind of model inherits.

The most commonly used type of models are:

- ModelSQL (Objects to be stored in an Sql-Database)
- ModelView (Objects to be viewed in the client)
- Workflow (Objects to have different states and state-transitions)

For API-Reference about Models in tryton refer to Trytond docs: model

Most likely your custom Model will inherit from ModelSql and ModelView at least, so it can be stored and viewed in the client.

Each ModelSQL can hold a set of tryton-fields to represent its attributes. For a complete list of field types available on tryton you are refered to Trytond docs: fields

Pool

Tryton provides a Pool to serve all your Models in a thread-safe way. The Pool can contain the following types:

- model
- wizard
- report

You can therefore obtain a model-class from anywhere in your module with the help of a Pool() instance:

```
Books = Pool().get('library.book')  # Model
book_no_one = Books(1)  # Instance
```

Model Inheritance

To extend an existing model (like Company), one only needs to instantiate a class with the same __name__ attribute:

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

__all__=['Company']
__metaclass__ = PoolMeta

class Company:
    __name__ = 'company.company'
    company_code = fields.Char('Company Code')
```

and register the model to the pool (using a different module name)

```
Pool().register(
    Company,
    module='company_customized', type='model')
```

Records

Most of trytons behaviour is itself defined by records of internal models (ir). All records are stored in the database and they can created within the client or statically predefined in xml files. When you are unsure about how to define the records, you are encouraged to explore the models in:

```
/trytond
    /ir
    /res
```

Views

The views are used to display records of an object to the user. In tryton, models can have several views, it is the action, that opens the window, that tells which views must be used. The view are built using XML that is stored in the module's view diectory or can be stored in database with the object.ir.ui.view. So generally, they are defined in xml files with this kind of xml:

```
<record model="ir.ui.view" id="view_id">
    <field name="model">model name</field>
    <field name="type">type name</field>
    <field name="inherit" ref="inherit_view_id"/>
</record>
```

Extending Views

Each inherit view must start with data tag. xpath tag is used which specifies the location where the field is to be added.

- expr: the xpath expression to find a node in the inherited view.
  - selecting elements starting from "/"
  - selecting one of a set of elements by querying attributes: [@attribute='value']
- position: Define the position of xml-injection.
  - before
  - after
  - replace
  - inside
  - replace_attributes (which will change the attributes)

Example

```
    <data>
        <xpath
            expr="/form/notebook/page/separator[@name='signature']"
            position="before">
            <label name="company_code"/>
            <field name="company_code"/>
            <label name="company"/>
            <field name="company"/>
            <label name="employee_code"/>
            <field name="employee_code"/>
        </xpath>
    </data>
```

Wizard

A wizard is a finite state machine.

Wizard(session_id) This is the base for any wizard. It contains the engine for the finite state machine. A wizard must have some state instance attributes that the engine will use.

Class attributes are: Wizard.__name__ It contains the unique name to reference the wizard throughout the platform.

Active Records

TODO

Tryton by example: library

Almost every Tryton functionality that you are going to develop on a daily basis is enclosed into the modules. In order to get an idea of available modules you should take a look inside trytond/modules. You will see many folders in the modules. Each one comprises one set of facilities that can be installed and will be available to the end users. Some examples are company, country, currency and party.

Module structure

First thing in the birth of a new module is the creation of a python package with the following structure:

```
/library
    /tryton.cfg
    /__init__.py
    /library.py
    /library.xml
    /view
        /book_form.xml
        /book_tree.xml
```

Let's see the contents and purpose of each one in detail:

tryton.cfg

This file must be present at the root of your module's directory. It contains the server-version the module was developed for, a list of the xml files the module contains and the modules it depends on:

```
[tryton]
version=3.2.1
depends:
    ir
xml:
    library.xml
```

In this case we are creating a module for trytond 3.2 series. We state that library.xml has to be parsed and our module doesn't have any real dependencies (ir is a builtin module)

library.xml

All our static data is usually put into xml files.

- actions
- views
- reports
- users/groups
- access restrictions

Observe that this file is a regular xml file. So it starts with the ordinary xml version declaration at the top, and it has as its master element the tryton element, followed by a data element. The other elements will all be children of data

```
<?xml version="1.0"?>
<tryton>
    <data>
        <!-- All our definitions come here -->
    </data>
</tryton>
```

__init__.py

This file must be present at the root of your module's directory. It serves two main purposes: it transforms your directory into a Python visible package (according to Python general rules) and it also registers in the Pool the entity classes of the module.

You can think of the Pool as a "in memory synchronized image" of your database, because Tryton follows the so called active record pattern. Tryton takes care of database table creation and of the mapping between the in-memory representation of the entity and the respective columns in the database. It also takes care of the synchronization of the data loaded in your in-memory entities and the persistent data on the database.

Whenever we are building a module in Tryton, we deal with a high-level, object-oriented representation of our entities. Generally, we are free from writing explicit SQL or python-sql instructions, but in order for this magic to happen, Tryton's Pool must be "aware" of the existence of your entity classes.

```
from trytond.pool import Pool
from .library import Book

def register():
    Pool.register(
        Book,
        module='library', type_='model'
    )
```

In the example above, we are registering the Book class into the Pool. Whenever the trytond service runs, it starts with initializing every module that is installed (more on that in the coming lines), i.e., it performs the regular Python initialization of packages. That means the execution of the code contained inside the __init__.py.

If you are unfamiliar with the package initialization, you can think of it as performing an analogous role as the __init__ method inside a Python class, but, in this case, it performs initialization tasks semantically relative to the whole package.

library.py

This file must be present at the root of your module's directory. According to a domain model, it contains the entity classes.

If your domain model is a commercial enterprise, your domain model would contain entities such as SaleOrder, Product, Customer and so on. Our tutorial here is proposing a library domain model, where you would expect to find Book, Author, Publisher, etc. A domain model encompasses real world objects that your software solution is expected to deal with.

In our tutorial, we are going to have a simple Book model. It has some fields associated with it: title, isbn, subject, abstract.

Each field has a Type. This type determines many aspects and behaviours of the application. For instance,

- Char field will be created as a Char Varying column inside the database.

- Text field will be displayed as a large text box in the Tryton Client window and so on.

In order to know every field avaliable, you can consult official Trytond docs: fields

Defining the model

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

# list of all classes in the file
__all__ = ['Book']


class Book(ModelSQL, ModelView):
    # description (mandatory on first declaration)
    'Book'

    # Internal class name. Always used as a reference inside Tryton
    # default: '<module_name>.<class_name>' on Tryton
    # becomes '<module_name>_<class_name>' in the database
    __name__ = 'library.book'

    title = fields.Char('Title', required=True)
    isbn = fields.Char('ISBN')
    subject = fields.Char('Subject')
    abstract = fields.Text('Abstract')
```

In our example we have defined four fields in the class. Tryton will automatically create a table in the database called library_book, consisting of nine columns: the four defined above and another five that are present on every column of the database:

- id
- create_date
- write_date
- create_uid
- write_uid

The first column is the surrogate primary key of the table. The following ones are self-explanatory, and are created for auditing purposes. In general, we should not worry about those columns, because Tryton takes care of them for us.

If you access the defined database, you are going to see the the aforementioned table created.

Creating the View

As we need our model to appear in the client we have to define a view. A complete list of all the available views can be found in Tryton docs, but in this tutorial we're only going to define the following for our module:

- tree view: to display a list of all our books
- form view: to view and modify all the details of one single book at a time

Each view is defined by its own xml-file which has to be placed in the 'view' folder of the module. Again this is a regular xml file with the following structure:

```
<?xml version="1.0"?>
<!-- This file is part of Tryton.  The COPYRIGHT file at the top level of
this repository contains the full copyright notices and license terms. -->
<form string="Books" col="6">
    <label name="title"/>
    <field name="title" colspan="3"/>
    <label name="isbn"/>
    <field name="isbn"/>
</form>
```

in our simple case we only need labels to put a translated version of our field name and fields to input/view field-data. There is a lot more formatting tags available which can be looked up from Tryton docs

adding a menu

In order to create a new menu we have to edit the library.xml file so it will contain the declaration of our menu and its respective menu item (submenu):

```
<menuitem name="Library" sequence="0" id="menu_library"/>
<menuitem name="Books" parent="menu_library" id="menu_books" action="act_library_window"/>
```

In the xml file above we have declared two menuitems. The first one, named Library will be placed on the root menu of Tryton client. Observe that it has, besides the name attribute, a sequence, that indicates the position of the menu, and an id, that must be unique. This id will identify this element to the rest of the software. It will be placed on the root menu because it has no parents.

The second menuitem, named Books has another element: a parent element, which points to the id of the former menu (id="menu_library"), indicating that it is going to be nested on the first one. this menu-item also has an associated action to call: 'act_library_window'.

Associating the views

there is four types of actions we could call from our menu-entry:

- ir.action.act_window
- ir.action.report
- ir.action.wizard
- ir.action.url

obviously we want to use act_window, which should open up a new tab in the client:

```
<record model="ir.action.act_window" id="act_library_window">
    <field name="name">Books</field>
    <field name="res_model">library.book</field>
</record>
```

Note: Our action has to be defined before referencing in the menu

We can then add different window-views to our newly created window:

```
<record model="ir.action.act_window.view" id="act_library_view1">
    <field name="sequence" eval="10"/>
    <field name="view" ref="library_view_tree"/>
    <field name="act_window" ref="act_library_window"/>
</record>
```

which themselves point to views

```
<record model="ir.ui.view" id="library_view_tree">
    <field name="model">library.book</field>
    <field name="type">tree</field>
    <field name="name">book_tree</field>
</record>
```

Where the "name" field points to our xml-file (form/book_tree.xml) containing the actual layout of the view.

Installing the package

When installing your package you can either link directly in the modules folder of tryton or use python setuptools (recommended). to use python-setuptools:

- obtain the contrib-module-setup.tmpl from hg.tryton.org/tryton-tools
- replace 'MODULE' and 'PREFIX' with your desired values
- save the file in your module-root as 'setup.py'
- install like any other module (refer to the installation guide)

Applying changes

In order for your changes to be applied we need to insert the module in the database. You can either achieve this by installing the module within the client or directly from command line using -i (insert):

```
TRYTOND_HOME/trytond/bin/trytond -d NAME_OF_THE_DATABASE -i MODULE_NAME
```

Whenever you make changes to the module, those changes can be applied by using the -u flag (update):

```
TRYTOND_HOME/trytond/bin/trytond -d NAME_OF_THE_DATABASE -u MODULE_NAME
```

Let's also restart the Tryton client now. Remember to start it with the -d (development) flag, so it can update the cache and show the changes we have just made:

```
TRYTON_HOME/tryton/bin/tryton -d
```

When you log in again on the client, you are going to see that the menu Library and the submenu Books have been created.

Tryton by example (2): library_rent

Now that we have created a working library-module which gives us the possibility to store information about all our books, we might find ourselves in a situation where we want to rent those books out and keep track of our friends who currently hold them. Because we read our books from time to time we also want to know if its currently available or not.

Field functions

As we already learned about trytons inheritance-model from basic concepts, we create a new module folder called 'library_rent' in the same structure as the one from last chapter, and we include 'library' as its dependency in tryton.cfg. We start by adding a simple char field straight away (we don't know any better yet):

```
class Book:
    __name__ = 'library.book'
    renter = fields.Char('Rented by')
```

Next thing is to find out if the book is available or not, but we don't want to manually set the state each time we rent out a book. Tryton has some handy field-functions to offer so we can automate this task.

Note: Field-functions are always executed on the server, but they may also be triggered from the client.

Field-Relationships

If you have a pair of fields that influence each others value, you can define functions to update the fields values whenever a change is detected. There is two use-cases for this:

Updating a field should trigger an update on a number of fields

- define a function named on_change_<field_name>
- return a dictionary containing {'field_name': value} for all fields to be updated
- decorate the function with @fields.depends(*keys) containing all keys to be updated or required for calculation. this ensures that all the fields get submitted by the client.

```
class Book:
    available = fields.Boolean('Available for rent')

    @fields.depends('available')
    def on_change_renter(self):
        if self.renter:
            return {'available': False}
        else:
            return {'available': True}
```

Updating a set of fields should trigger an update on a single field

- define a function named on_change_with_<field_B_name>
- return the fields new value
- decorate the function with @fields.depends(*keys) using all the keys that may influence the field

```
class Book:
    available = fields.Boolean('Available for rent')

    @fields.depends('renter')
```

```
    def on_change_with_available(self):
        return self.renter == ''
```

Note: on_change_* and on_change_with_* are called from the client

Function fields

The previous 'on_change_owner' example could have been solved without storing a new key in the database by calculating its value on the fly. To achieve this we need to add a function field:

```
class Book:
    available = fields.Function(fields.Boolean('Available'), 'get_renter_information')

    def get_renter_information(self, name):
        return self.renter == ''
```

where name is the fields name. This special field can be accessed just as if it was a normal field of the type specified but gets computed each time (on the server)

Note: function fields are calculated on the server and may be incorrect when a value is changed in the client

Combining on_change with a Function field

Now that we know about those concepts we can even have all the advantages of Function fields (no extra database-column) and on_change_* functions (updated in the client) by combining them:

```
class Book:
    available = fields.Function(fields.Boolean('Available for rent'), 'on_change_with_available')

    @fields.depends('renter')
    def on_change_with_available(self, name=None):
        return self.renter == ''
```

So we have that problem solved.

Default values

After a while, we recognize that whenever there is a new book in our library, we want to read it first. We don't want to rent it out by accident, so by default we should be the renter on a new library-entry.

You can define default values for fields by adding a 'default_<field_name>' function to your model:

```
class Book:
    __name__ = 'library.book'
    renter = fields.Char('Rented by')

    def default_renter():
        return 'me'
```

Relational Fields

It turns out that actually there is a lot more than just a hand of forename-friends who want to rent our books, so we want to store a bit of additional information with our renters. We have a good run and find out that tryton already has a useful Party Module to offer, so all we have to do is to extend our book model by a new field to reference a party who currently rents the book from our library.

Like any ORM Tryton offers relational fields, which enable you to connect model(s) to its related model(s). You can use any of these:

- Many2Many - for example (Many) models can belong to a category but also to other (Many) categories
- Many2One - Connect a set of (Many) models to a parent (One) (example: a company field in company.employee Model)
- One2Many - A field representing (Many) connected model instances (example employees field in company.company model)
- One2One

Given that information, we could solve our Library example a bit more elegant by using Trytons built-in Party model and rent books only to registered parties:

```
class Book:
    __name__ = 'library.book'
    renter = fields.Many2One('party.party', 'Renter', required=False)

class User:
    __name__ = 'party.party'
    rented_books = fields.One2Many('library.book', 'renter', 'Rented Books')
```

Note: The One2Many field requires a Many2One field to be referred in the related Model.

Transactions

TODO

Creating Reports

Add the following line to the file 'library.xml' into the /data tag :

```
<!-- First thing: define the report itself,
model: Target-Model
report_name: the report class ' __name__
report: template ods-file
-->
<record model="ir.action.report" id="report_library">
    <field name="name">Book</field>
    <field name="model">library.book</field>
    <field name="report_name">library.book</field>
    <field name="report">library/book.odt</field>
</record>
<!-- Second we register a keyword
(so we can call the report from tryton client) -->
<record model="ir.action.keyword" id="report_library_book">
```

```
    <field name="keyword">form_print</field>
    <field name="model">library.book,-1</field>
    <field name="action" ref="report_library"/>
</record>
```

Now create the file book.odt inside your module. In this file add the following lines by adding a placeholder in your odt file.

```
<for each="library in objects">
<library.title>
</for>
```

Tip: placeholders can be inserted in libreoffice by pressing ctrl+f2 functions -> placeholder -> text

In case you are dealing with ods file. For adding a placeholder you have to add a hyperlink.

Wizard

A wizard is a fine state machine.

Wizard(session_id) This is the base for any wizard. It contains the engine for the finite state machine. A wizard must have some state instance attributes that the engine will use.

Class attributes are: Wizard.__name__ It contains the unique name to reference the wizard throughout the platform.

Wizard.start_state It contains the name of the starting state.

Wizard.end_state It contains the name of the ending state.

Wizard.__rpc__ Same as trytond.model.Model.__rpc__.

Wizard.states It contains a dictionary with state name as key and State as value

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

class PrintLibraryReportStart(ModelView):
    'Print Library Report'
    __name__ = 'library.print_report.start'

class PrintLibraryReport(Wizard):
    'Print Library Report'
    __name__ = 'library.print_report'

    start = StateView(
        'library.print_report.start', 'library.print_view_form',
        [
            Button('Cancel', 'end', 'tryton-cancel'),
            Button('Print', 'print_', 'tryton-print', default=True),
        ]
    )
    print_ = StateAction('library.book')

    def do_print_(self, action):
        data = {
            'library': self.start.book.id,
        }
```

```
        return action, data

    def transition_print_(self):
        return 'end'
```

Register the Wizard model name in __init__.py and add the xml files in tryton.cfg file.

```
#Register type_='wizard' in __init__.py
Pool.register(
  PrintLibraryReport,
  module='library', type_='wizard'
)
```

Add the record tag for the wizard in library.xml

```
<record model="ir.action.wizard" id="book_print">
    <field name="name">Print Library Book</field>
    <field name="wiz_name">library.print_report</field>
</record>
```

WebServices

TODO

How to get help

IRC (Chat)

Usually there are developers on the IRC chat, and they will try to help you if you provide a good information about your problem. You should supply as a minimum:

- Version of tryton you are running (found in the Help/About menu selection)
- What operating system and version of OS are you running
- What module is the problem
- Ideally a screenshot of the error you are encountering
- The sequence of steps to cause the problem

You can find more information on the IRC (Chat) section of user documentation.

Mailing List

- tryton-dev@googlegroups.com
- tryton-contrib@googlegroups.com

TODO: Elaborate how to use both

Advanced Concepts:

Domains

What is a domain ?

Domains are sets of search criteria. They apply constraints on the model that is searched so that only a sub-set of its instances are valid regarding the domain.

You can consider a domain as a filtering expression for your target model. This is an example of a very basic domain:

```
[('amount', '>', 0)]
```

This domain will filter all records whose amount field is greater than 0.

A domain clause is the building block of a domain. A full domain combines multiple domain clauses, as well as the OR and AND operators. Domain Clause:

```
('field_name', 'operator', value, <optional parameter>)
```

The 'field_name' part of the domain clause is much more than just a field name. It is possible to chain fields, in order to apply contraints on a Many2One fields. For instance,

```
('invoice.date', '>', Date.today())
```

is a valid domain.

> Warning: Though it is theoratically possible to chain fields at will, be carfeul that you will still be limited by the database performance. Usually, more than two chains (i.e. field1.field2.field3) should be avoided if possible.

Domain (the AND operator is implicit between domain clauses):

```
[domain_clause1, domain_clause2]
[domain_clause1, ['OR',
    [domain_clause2, domain_clause3],
    [domain_clause4]]]
```

Domain Usage

Domains are everywhere in tryton. As soon as you need to limit the result of a research on a model, chances is that you are going to need a domain.

Relation Fields

All relation fields (Many2One, One2Many, Many2Many) support domains. Setting a domain on a relation field means that the value(s) of the field must comply with it in order for the record to be valid. For instance:

```
party = fields.Many2One('party.party', 'Party', domain=[
    ('name', '=', 'John')])
```

It is now mandatory for all instances of the model in which this field is defined that its value's name is 'John'.

In administrative tools

Some of Tryton's internal models use domains: * Views * Act Windows * Rules

See the dedicated documentation for details regarding their usage. .. TODO : Link this to the actual documentation of views / act windows.

Anywhere in the code

When using Model.search(...), the first argument you need to pass is a domain. Almost all the queries that you will write will (and should) use the search method associated to a domain. Directly building queries through python-sql, though possible, should be restricted to the case of complex queries. For instance, aggregate accounting queries would be way to inefficient if they used the domain pattern as the search function does not provide aggregate functions.

Advanced Domains

Domains support a lot of advanced features which make them very flexible and adapted to a lot of needs.

Usage of Pyson

Using Pyson in domains is supported. This is awesome. This allows you to create dynamic domains depending on, for instance, the value of the current record field. The context is evaluated as well when computing the domain, so it is possible to use the active_id in domains set on views.

> Warning: When using Pyson, be very careful to know what field_names are those of the source record (and thus will be evaluated), and which one just represent the field names of the target model.

Tuning a domain clause with Pyson:

```
('amount', If(Bool(Eval('active')), '>', '<'), 0)
```

The above domain clause behaves as follow: if the field 'active' of the current record is True, it will accept all target model records for which the field 'amount' is greater than 0. If 'active' evals to False, it will accept records for which the 'amount' is less than 0. This gives you a wide range of possibilities to define precisely the domain you need for what you want to do.

Using Pyson to dynamically set the search value:

```
('product.category', '=', Eval('category'))
```

Here, product.category is the field on which the domain clause should be applied. category is the current record's category field value.

Tuning a full domain with Pyson:

```
[(domain_clause1, If(Eval('active'), domain_clause2, domain_clause3))]
```

You can enclose whole domain clauses in Pyson. A typical use case is to test whether the record's id is set, or if we are working in the scope of a company.

Using the context:

```
('company', '=', Eval('context', {}).get('company', None))
```

Limiting the target record to one which matches the current company is usual, here is how to do this.

Searcher

Function fields may be used in domains. That is the purpose of the searcher keyword argument in fields.Function. This argument refers to the name of a function (the searcher function), which transforms a domain_clause using the function field in a database compatible clause.

For instance, let's assume that the model we are working on has a function field whose value is the first letter of a Char field. The searcher function will then look like this:

```
@classmethod
def search_my_function_field(cls, name, clause):
    return ('my_char_field', clause[1],
        clause[2][0] if len(clause[2]) else '')
```

We are basically forwarding the clause from the function field to the actual char field. We convert the operator part of the clause to only use the first letter of the search value to match the function field definition.

Debugging Trytond

Tryton is a wonderful platform to work with, populated with equally wonderful modules. Unfortunately, even the most skilled programmers sometimes make mistakes, and then comes what developers do 90 % of their time: debugging.

This is what all debugging is about:

- Find what went wrong
- Understand why it went wrong

Once the source of the problem is pinpointed, you can correct it. Usually, correcting a bug without knowing how it appeared will prove useless in the long run, so one got to be able to discover the core of the problem.

Remember though, a not consistently reproductible bug is your worse nightmare. This category of bug requires some special treatment that will be detailed in a dedicated place. The following assumes that your are able to consistently and easily reproduce the problem.

Tryton Configuration

The tryton server comes bundled with a few options to make the debugging easier. Every developement server should have the following setting enabled in their configuration file:

```
auto_reload = True
```

This makes the server reload itself every time one of its resources (python files, view files...) That allows you to modify your code to ease debugging with the client running, and restart the action that caused the problem to either obtain more data on the problem, or to check if a modification you made changes something.

Tip: Keep in mind though that the modifications that require creating new records in the database to be effective still require a database upgrade. That includes adding new fields, creating new records in xml (views / actions...).

Usage of print

The most basic debugging method with python is printing. It is particularly efficient with the server properly configured as described in the previous section.

Of course, printing needs to be intelligent to be effective. Usually, you will want to do the following:

- Find a context in which the bug arises. This is particularly important when the method in which the bug occurs is often used. For instance, when calling a method on a list of ids, you need to detect which instance made the method crash. This can easily be achieved by printing the method arguments at the top of the call / the iteration values at the start of a loop. Another option would be a try / except around the bad line.
- Once you can design a test that you are confident allows you to detect the problem's context, you can exhaustively use print to get all the context information you need to understand what happens.

Server Logging

Use python's logging module to write down useful data for debugging. If you use the 'DEBUG' loglevel, it will not appear anywhere. It is interesting to use it in tricky places of the code in which for instance not all cases can be properly tested.

Currently, the tryton server does not allow easy configuration of the output log level. To achieve this, you need to edit trytond/server.py and replace occurences of logging.WARNING with logging.DEBUG in TrytonServer.__init__

Client Logging

The tryton gtk client provides useful fonctionnalities for debugging: debug mode and verbose mode. Those are arguments on the command line of the tryton gtk client:

```
tryton -d -v
```

The debug mode force the client to fetch the definition of each view you want to display, every time you want to display it, from the server. That includes fields definition and xml structure. Useful when your problem is a field dependency in on_change(_with) / depends. Just change your source, reload the view (close the current tab and reopen it), and it will be up to date.

The verbose mode is more useful from a debugging point of view.

Every action in the client triggers one / multiple server requests. For instance, opening a view requires the client to fetch the view definition, the access right data, the records data, etc. Once you nailed down the action in the client that triggers the problem, it may trigger tens of requests to the server, and you got to know which one of those caused the crash. Enabling the verbose mode will make every request from the client to the server displayed in the server log this way:

```
INFO:tryton.rpc:model.model_name.method_name(_, _, arg1, arg2, ..., context)
DEBUG:tryton.rpc:something
```

The first arguments of the method call are json-rpc (xml-rpc is similar) specific parameters like the session token. The answer is the json encoded method result

Using this properly allows you to know precisely what the server was asked to do, which is a step toward resolution.

Pdb

Pdb is the Python Debugger. It may be useful in particularly complex cases, or when debugging the client itself. It basically provides you a way to place breakpoints in your code (which is particularly good combined with the server auto-reloading feature).

Once your running application arrives at the line at which you set the breakpoint, it stops, and give you the possibility to explore the current state. It features stack exploration (goin up / down), symbol evaluation, step by step execution... A good use case is once you know precisely where is the problem, but you cannot figure out exactly what is going on.

It is no the prefered way to debug as it is some sort of overkill, but definitely useful in some situations.

Note that for vim users, there exists a python vim binding named Vimpdb which allows to use vim as an interface for pdb, which allows for a better view of the surrounding code.

Setup trytond for debugging

There are some traces that are very useful to set up in the server in order to check for the usual suspects.

Debug those annoying Error 500

In the trytond/protocols/jsonrpc.py file, in SimpleJSONRPCDispatcher._marshaled_dispatch, you should enclose the

```
return json.dumps(response, cls=JSONEncoder)
```

statement in a try / except + traceback + raise to know what really failed when you got an error 500 client side.

Know where functional errors where thrown

Add those lines at the start of the raise_user_error method of the WarningErrorMixin class of the trytond/error.py:

```
import traceback
traceback.print_stack()
```

That will make it so that everytime a user error is thrown somewhere in the server, the server log will print the current stack before displaying the error to the user.

Debug Functional Errors

Write

```
print cls.__name__, field_name, value
```

in ModelStorage._validate.required_test (modelstorage.py). This will give you some info in case of "The field ... is required"

Write

```
print cls.__name__, field_name, value, test
```

in ModelStorage._validate at the cls.raise_user_error('selection_validation_record') line. That way you will know why "The value ... is not in the selection"

How to deal with non-reproductible errors / client errors

Those are the worst thing you can encounter. The solution for debugging them is the same: consider you got only one go:

- When the error occurs server-side and is not reproductible, the only thing you can do is make it so that you get the maximum information out of it the few times it occurs.
- The client does not have the nice autoreload feature of the server (it is only possible in the server as it runs separate threads). So everytime you change the code, you need to fully restart it. The bottom line is the same: you got to make those runs worth it.

So basically, use logging extensively. Logging is nice because you can just go on something else, until the error occurs. Once it does occur, you should be able to get relevant information about the error context, which hopefully will make it possible to pinpoint it and understand how to reproduce it.

Another option is to use Pdb's post-mortem debug mode. This allows you to try / except your error, then trigger Pdb in the except block. Doing so will make python enter debugging mode in the context of the error when it occurs.

Usual errors and how to debug them

How to write unit tests

TODO: Elaborate

Security Guidelines

Writing SQL ?

- Why not to do it
- Using python-sql

API Reference:

> TODO:Full API Reference generated by Udo

CHAPTER 4

Indices and tables

- genindex
- modindex
- search